Cheap Out-of-Order Execution using Delayed Issue

J.P. Grossman

Abstract

In superscalar architectures, out-of-order issue mechanisms increase performance by dynamically rescheduling instructions that cannot be statically reordered by the compiler. While such mechanisms are effective, they are also expensive in terms of both complexity and silicon area. There is a need for cost-effective alternatives when area efficiency becomes a concern, such as when multiple processors are placed on a single die. 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.

The goal of this paper is to explore the concept of delayed issue in detail. We explain its advantages and identify the challenges and correctness issues that arise. We show that an inexpensive implementation is possible by describing simple mechanisms for inserting instructions into delay queues and maintaining the queues in a hazard- and deadlock-free state. Quantitative performance evaluations are the subject of future research.

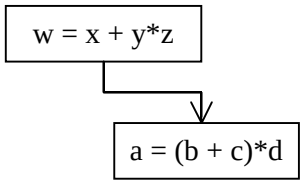
1 Introduction

The order in which instructions are executed on a superscalar architecture can have a dramatic impact on performance. To some extent optimizing compilers are able to address this by statically reordering assembly code. However, even a trace scheduling compiler [Fisher81] with perfect knowledge of the underlying hardware cannot always optimally schedule instructions as:

- It is difficult to optimize across basic block boundaries when the program flow graph has high fan-in (e.g. a subroutine called from many places) or high fan-out (e.g. an indexed or indirect jump)
- Memory references cannot be reordered if one of them is a store and the compiler is unable to disambiguate the addresses

The familiar solution to this problem is to allow the hardware to issue instructions out of order as soon as it is safe to do so. 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). If the hardware supports out-of-order issue, then the same four instructions take 7 cycles to execute (Figure 1d).

Project Aries Technical Memo ARIES-TM-06
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA



(a) Program flow

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

(b) Instruction sequence

```
1 _ _ 2 3 _ _ 4 _ _
```

(c) In order execution: 10 cycles

```
1 3 _ 2 4 _ _
```

(d) Out of order execution: 7 cycles

Figure 1: Out of order execution

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 scalar performance with much less area overhead.

Area efficiency is the motivation for **Delayed Issue**, the central idea of which is to allow the compiler to specify explicit delays for instructions that would ordinarily stall due to data dependencies. These delays are used to place instructions into per-functional unit delay queues. Instructions within a queue are executed in the order that they appear; this is not necessarily the original program order as individual instructions may be inserted into a queue ahead of or behind other instructions depending on their relative delays. Thus, delayed issue achieves out-of-order execution without the complexity of dynamic out-of-order issue hardware.

The next section discusses previous related work, then in section 3 we present a more detailed description of delayed issue. The main challenge of delayed issue is avoiding data hazards and deadlock within the queues; section 4 presents mechanisms for accomplishing this under some simplifying assumptions. In section 5 we explore how some of these assumptions can be relaxed. We estimate the complexity of the presented mechanisms in section 6, and finally we conclude in section 7 and outline our plans for future work.

2 Previous 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.

One of the most well known architectural alternatives is the Very Long Instruction Word (VLIW) architecture [Fisher83]. 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 order to achieve high performance on a VLIW machine it is necessary to schedule instructions on a coarser granularity than the basic block level; the technique of trace scheduling [Fisher81] was developed for this purpose. While trace scheduling was originally introduced to compact microcode and was later used for VLIW compilation, it is a very general technique that can produce highly optimized code for any superscalar architecture. VLIW has also been extended in various ways to improve its performance. In [Rau93] instructions were split into two phases corresponding to initiation and completion of the original instruction, with the second phase being placed in a delayed issue 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. In [Hara96] a predicating mechanism was described which allows the compiler to perform speculative code motions across conditional branches. It was hypothesized that a VLIW machine with predicating would have a smaller cycle time than a superscalar out-of-order processor, and it was shown that under this assumption the VLIW

machine would have the higher performance. Allowing the compiler to reorder instructions across branch boundaries was also studied for general multiple-instruction-issue processors in [Chang91]. Their approach was to add a set of non-trapping instructions in order to guarantee the safety of upward code motion; the speedups obtained were very near those of an out-of-order issue processor.

Another architecture which has been proposed is the decoupled access/execute (DAE) architecture [Smith82] in which programs are statically split into two instruction streams: a memory access stream and a computation stream. These two streams are executed in-order on separate processors which communicate via hardware queues. Surprisingly, speedups greater than two are possible due to the fact that the issue logic spends less time waiting to issue instructions than in a conventional in-order processor. It has been found that a DAE machine can provide speedups of more than 150% over scalar architectures [Smith86] and can outperform simple out-of-order superscalar architectures [Farrens93]. In [Love90] a comparison was made between VLIW and DAE architectures; it was found that the relative performances were heavily application dependent, and that the two architectures had roughly the same average performance.

One of the earliest simplifications of Tomasulo's algorithm was Thornton's scoreboarding approach used in the CDC 6600, which was conceptually similar but did not include register renaming. In [Weiss84] a restricted form of register renaming was proposed in which at most one reservation station can be waiting for a given result. This allows result tags to index a table of reservation stations and avoids the expensive associative lookup operation which is normally performed to distribute result data to reservation stations that need it.

An excellent presentation of ways to simplify superscalar microarchitectures 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.

3 Delayed Issue

In section 1 it was explained that a compiler cannot always optimally reorder instructions. A key observation is that while the compiler does not know in these situations what instructions *can* be issued, it is often able to determine which ones *can't* be issued. 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).

At this point we have made no mention of how to ensure correctness of execution or what to do when the delay slot specified by the compiler is already occupied by a previously-delayed instruction. We will defer discussion of these issues to section 4.

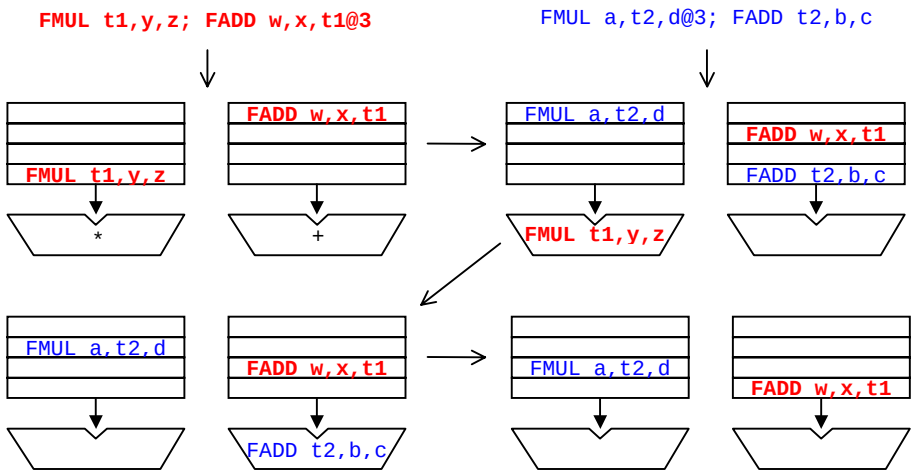


Figure 2: Delayed Issue: delay queues for adder and multiplier

3.1 Why it Works

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.

The circuitry used to determine when an instruction is ready to be issued is an expensive resource. 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, this resource 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 shameful waste of costly silicon. With delayed issue, this wastage is avoided; placing such instructions in delay queues keeps the issue mechanism available for instructions that can actually use it.

3.2 Ordering Semantics

Specifying delays does not affect the semantic ordering of instructions; it merely provides useful information to the hardware for computing the results of program-order execution. The only case in which delays affect instruction order is when instructions within the same group have different delays. In this case the delays are used to order the instructions within the group. Thus, for example, when the following sequence of instructions is executed:

```
1  A; B@3
2  D@2; E@4; C
3  F
```

the observed order of execution is always A, B, C, D, E, F.

As an aside, note that without delayed issue the instructions within a group would have to appear in execution order. Using delays to specify execution order makes it possible to place the instructions in functional unit order, which results in slightly simpler decode logic.

3.3 Loop 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. A scoreboard showing the utilization of the pipelined functional units is given in Figure 3c (we treat the memory interface as a functional unit capable of handling one memory request per cycle). Assuming (as before) that floating point addition and multiplication take 3 cycles, integer/pointer operations take 1 cycle, and the data resides in the cache so that memory references take 1 cycle, the total execution time is 8 cycles per iteration.

```
for (i = 0 ; i < N ; i++)
{
    y[i] = x[i] * x[i] + c;
}
```

(a) Simple loop

```
L0: LOAD  a2,[a0]; PADD a0,a0,1
    FMUL  a2,a2,a2
    FADD  a2,a2,a3
    STORE a2,[a1]; PADD a1,a1,1; LOOP L0
```

(b) Assembly code

memory	addition			multiplication			integer/pointer
LOAD							PADD
				FMUL			
					FMUL		
						FMUL	
	FADD						
		FADD					
			FADD				
STORE							PADD

(c) Scoreboard of hardware resource utilization

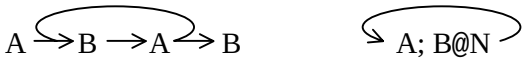
memory	addition			multiplication			integer/pointer
LOAD	*FADD*		**FADD**		***FMUL***		PADD
STORE		*FADD*		FMUL		***FMUL***	**PADD**
LOAD	***FADD***		*FADD*		FMUL		**PADD**
STORE		***FADD***		**FMUL**		FMUL	*PADD*
LOAD	FADD		***FADD***		**FMUL**		*PADD*
STORE		FADD		*FMUL*		**FMUL**	***PADD***
LOAD	**FADD**		FADD		*FMUL*		***PADD***
STORE		**FADD**		***FMUL***		*FMUL*	PADD

(d) Scoreboard for pipelined loop – different fonts correspond to different iterations

Figure 3: Simple loop before and after software pipelining

```
PR:   LOAD  b2,[b0]; PADD b0,b0,4
      FMUL  b2,b2,b2
      LOAD  c2,[c0]; PADD c0,c0,4
      FMUL  c2,c2,c2
      LOAD  d2,[d0]; FADD b2,b2,b3; PADD d0,d0,4
      FMUL  d2,d2,d2
L0:   LOAD  a2,[a0]; FADD c2,c2,c3; PADD a0,a0,4
      STORE b2,[b1]; FMUL a2,a2,a2; PADD b1,b1,4
      LOAD  b2,[b0]; FADD d2,d2,d3; PADD b0,b0,4
      STORE c2,[c1]; FMUL b2,b2,b2; PADD c1,c1,4
      LOAD  c2,[c0]; FADD a2,a2,a3; PADD c0,c0,4
      STORE d2,[d1]; FMUL c2,c2,c2; PADD d1,d1,4
      LOAD  d2,[d0]; FADD b2,b2,b3; PADD d0,d0,4
      STORE a2,[a1]; FMUL d2,d2,d2; PADD a1,a1,4;
                     LOOP L0
EP:   FADD  c2,c2,c3
      STORE b2,[b1]; PADD b1,b1,4
      FADD  d2,d2,d3
      STORE c2,[c1]; PADD c1,c1,4
      STORE d2,[d1]; PADD d1,d1,4
L1:   LOAD  a2,[a0]; PADD a0,a0,1
      FMUL  a2,a2,a2
      FADD  a2,a2,a3
      STORE a2,[a1]; PADD a1,a1,1; LOOP L1
```

(a) Pipelined loop; no delayed issue



(b) Prologue, Epilogue **(c)** Delayed issue

```
L0:   LOAD  a2,[a0];   FMUL a2,a2,a2@1; PADD a0,a0,4
      STORE a2,[a1]@7; FADD a2,a2,a3@3; PADD a1,a1,4@7
      LOAD  b2,[b0];   FMUL b2,b2,b2@1; PADD b0,b0,4
      STORE b2,[b1]@7; FADD b2,b2,b3@3; PADD b1,b1,4@7
      LOAD  c2,[c0];   FMUL c2,c2,c2@1; PADD c0,c0,4
      STORE c2,[c1]@7; FADD c2,c2,c3@3; PADD c1,c1,4@7
      LOAD  d2,[d0];   FMUL d2,d2,d2@1; PADD d1,d0,4
      STORE d2,[d1]@7; FADD d2,d2,d3@4; PADD d1,d1,4@7;
                       LOOP L0
L1:   LOAD  a2,[a0];   FMUL a2,a2,a2@1; PADD a0,a0,1
      STORE a2,[a1]@7; FADD a2,a2,a3@3; PADD a1,a1,4@7;
                       LOOP L1
```

(d) Loop unrolled four times with delayed issue

```
L0:   LOAD  a2,[a0];   FMUL a2,a2,a2@1; PADD a0,a0,4
      STORE a2,[a1]@7; FADD a2,a2,a3@3; PADD a1,a1,4@7;
                       LOOP L0; ROTATE
```

(e) Combining delayed issue and register rotation

Figure 4: Pipelining the loop with and without delayed issue

An inspection of the scoreboard reveals that the loop can be pipelined to obtain an execution time of 2 cycles per iteration. The scoreboard for the pipelined loop is shown in Figure 3d, and the corresponding assembly code is given in Figure 4a. Note that the number of instructions has increased dramatically from 6 to 48. Of these, 18 make up the prologue (PR) and epilogue (EP), 24 make up the pipelined loop (L0), and 6 form a copy of the

original loop (L1) which is used when N is not a multiple of 4. This does not include setup instructions for initializing registers and verifying that N > 3, which we have omitted for simplicity.

The reason that the prologue and epilogue are necessary is that there are several instances of instructions A, B such that A logically precedes B, but in order to pipeline the loop B must be scheduled closer to the start of the loop than A. Without delayed issue, the only way to achieve this schedule is to actually place B earlier than A in the 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 4b). 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 4c). In Figure 4d we have rewritten the assembly code using delayed issue; the number of instructions has dropped from 48 to 30.

As a final note, observe that the code in Figure 4d essentially contains five copies of the same two lines; the only difference is the register names. If the hardware supports register rotation then we can rewrite the pipelined loop using only 6 instructions (Figure 4e).

4 Implementation Details

The previous section gave an intuitive introduction to delayed issue without considering the problems that can arise. In this section we present criteria for ensuring correctness of execution and we describe a simple implementation for delayed issue.

4.1 Data Hazards and Deadlock

Out of order execution raises the spectre of data hazards. To guarantee correct execution, we need to ensure that errors are not introduced by RAW, WAR and WAW hazards. A simple policy which achieves this is to always obey the following rules:

1. Two instructions in the same group with a data hazard dependency must have different compiler-specified delays
2. If an active instruction I writes to register x, no other instruction which reads or writes x may be issued until I completes
3. Two instructions may not be reordered with respect to one another if one of them writes a memory location/register that can potentially be accessed (read or written) by the other (when this is true for instructions A, B and A precedes B in program order, we write A < B)

Rule 1 applies to the compiler only, and allows the hardware to assume that instructions in the same group with the same delay are independent. Rule 2 is easy to implement using per-register busy bits [Tomasulo67]. A register's busy bit is set when an instruction is issued that writes to that register; the bit is cleared when the instruction completes. If an instruction accesses a busy register, that instruction is stalled.

Rule 3 can be implemented by requiring:

$$\text{delay}(A) < \text{delay}(B) \quad \text{whenever} \quad A < B \qquad \text{(ORD)}$$

where on any given cycle delay(X) refers the depth of instruction X within its delay queue (note that A and B may be in different queues). This restriction also guarantees that the queues will never deadlock, as it eliminates situations in which an instruction at the head of its queue stalls due to a dependency with an instruction that has not already been issued.

To see how ORD can be enforced, we begin with the simplifying assumption that all queues move in lockstep. This is not unreasonable if the hardware is pipelined and the compiler is able to use delays to avoid most stalls. In this case, if ORD is satisfied at a given time, it will automatically be satisfied on the next cycle for the instructions already in the queues. Thus, we only need to ensure that new instruction groups are placed in the queues in a manner that does not violate ORD.

4.2 Inserting New Instructions Groups

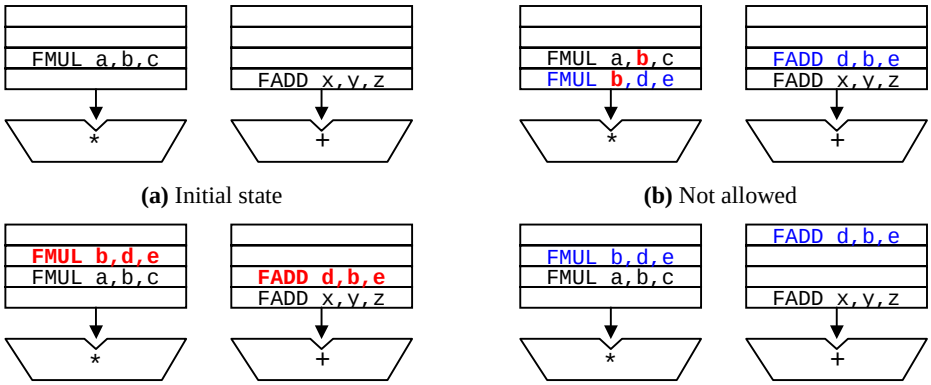
Suppose there are M functional units and each queue has depth N. Given an instruction group

$$I_1@d_1 \; ; \; I_2@d_2 \; ; \; \ldots \; ; \; I_M@d_M$$

we must find delays $0 \leq d_1', d_2', \ldots, d_M' < N$, where d_j' is the depth at which I_j is to be inserted in the j^{th} queue, such that:

1. $d_j' \geq d_j$ and the d_j' delay slot in the j^{th} queue is empty
2. Each d_j' is large enough such that ORD is satisfied with respect to all instructions already in the queues
3. $d_i' > d_j'$ whenever $d_i > d_j$

Figure 5 shows some simple examples of insertions that are or are not allowed according to these rules. In this section we will develop a mechanism for following these rules such that ORD is satisfied with respect to register hazards only; in section 4.3 we will extend the mechanism to deal with memory hazards.



(a) Initial state **(b)** Not allowed

(c) Not allowed **(d)** Allowed

Figure 5: Inserting `FMUL b,d,e; FADD d,b,e@1`. **Bold** denotes a violation of the rules.

We start by computing bit vectors $q_1, q_2, \ldots, q_M$ of length N where $q_j[d] = 1$ if I_j can be inserted into the j^{th} queue at depth d subject to rules 1 and 2. Let R be the number of registers. For each depth d we maintain two bit vectors w_d and r_d of length R where $w_d[k] = 1$ if register k is written by some instruction at delay d or greater, and $r_d[k] = 1$ if register k is read *or* written by some instruction at delay d or greater. If I_j writes to register a_j and reads from registers b_j and c_j, then I_j has no data hazard dependency on any instruction at depth d or greater if and only if $r_d[a_j] + w_d[b_j] + w_d[c_j] = 0$; in this case I_j can be inserted at depth d without violating rule 2. We can compute the values $r_d[a_j] + w_d[b_j] + w_d[c_j]$ by arranging the vectors w_d, r_d in an Nx2R array with M sets of vertical address lines and M sets of horizontal wired-nor bit lines. Taking the j^{th} output of this array and masking out bits corresponding to delay slots that are occupied or have depth less than d_j, we obtain the desired bit vectors q_j.

We now must choose $d_1', \ldots, d_M'$ such that $q_j[d_j'] = 1$ and $d_i' > d_j'$ whenever $d_i > d_j$. While the former condition is easy enough to deal with in hardware, the latter is quite difficult. We therefore make another simplification by choosing a fixed "shift" s and setting $d_j' = d_j + s$. The shift s must be chosen such that $d_j + s < N$ and $q_j[d_j+s] = 1$ for each j. If we set $q_j[i] = 0$ for $i > N$ then the condition is simply $q_j[d_j+s] = 1$. Finally, if we define q_j' to be q_j shifted by d_j places and truncated to N bits, the condition is $Q[s] = 1$ where $Q = q_1' + q_2' + \ldots + q_M'$. We therefore take s to be the index of the first 1 in the bit vector Q. If $Q = 0$ then we have failed to find an allowable insertion and we stall until the next cycle.

When the instructions are inserted into the delay queues, the bit vectors w_i and r_i must be updated accordingly. This is conceptually easy but wire intensive as each instruction can affect any bit vector depending on its delay d_j'. Note that when the queues advance, w_i and r_i shift to w_{i-1} and r_{i-1} respectively, and w_{N-1}, r_{N-1} become zero.

4.3 Memory Hazards

The obvious problem with memory instructions is that the address being referenced may not be known when the instruction is decoded. In this case the instruction presents a possible data hazard with every other memory instruction, unless both are loads. Thus, if no attempt is made determine the address at instruction decode time, the only way to guarantee that ORD is satisfied is to strongly order stores with respect to all other memory operations within the memory delay queue. This can have a serious impact on performance; in particular the code of Figure 4d would not work as the second load would have to be scheduled after the first store.

The solution is to attempt to resolve the address when the instruction is decoded, and to store this address in the delay queue with the instruction. This can be as easy as reading the address from the register file, but only if we can determine that it will not change between the time that the instruction is decoded and the time that it is issued. If the address is read from register x, then we need to ensure that x is not written by any instruction already in the queues or by any instruction in the same group with smaller delay. The former condition translates to $w_0[x] = 0$, and the latter is easy enough to verify by comparing x to the registers written by the other M-1 instructions in the same group, while simultaneously comparing delays. If it is safe to do so, the address in x is read immediately and then stored with the instruction in the queue. Otherwise, an "unknown" bit is set for the address in the queue.

Using this address information, we can compute the delays at which a new memory instruction I can be inserted such that ORD is satisfied with respect to the other loads and stores in the queue. Suppose I references address a, and that a memory instruction J already in the queue references b. Then I has a hazard dependency on J if and only if they are not both loads, and either $a = b$ or one of a, b is unknown. Let h be N-bit vector such that $h[d] = 1$ if I has a hazard dependency on a memory instruction at delay d or greater. If the m^{th} delay queue is for memory instructions, $q_m h$ is a modified bit vector giving the depths at which I can be inserted taking other memory instructions into account. We then proceed as before, and ORD will be satisfied with respect to both registers and memory.

4.4 The Omission of Register Renaming

It should be noted that WAR and WAW register hazards can be eliminated using register renaming; in fact this is common practice for modern architectures. Register renaming is an elegant mechanism which effectively changes registers from temporary named storage to edges in the program dataflow graph. However, as with out-of-order issue, it is also extremely expensive. Since the motivation behind delayed issue is the development of area efficient architectures, we assume that register renaming is not supported.

5 Improving Performance

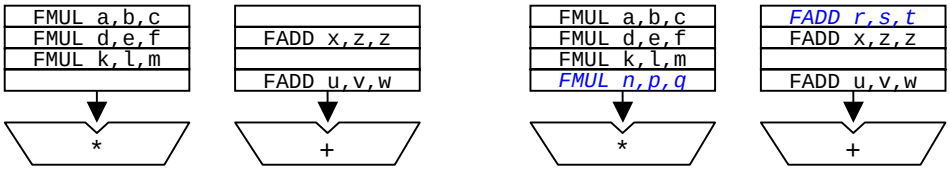
In the previous section we developed a set of mechanisms for ensuring the correct operation of delayed issue. To ease this task, we made certain simplifying assumptions about how the queues are managed. In this section we will show how to improve performance by relaxing these restrictions.

5.1 Insertions Revisited

The mechanism presented in section 4.2 for finding an allowable insertion restricts its search to insertions that are simply "shifts" of the compiler-specified delays. This allows the hardware to ignore the relative delays of instructions within a group, but it can have unfortunate consequences. Consider the problem of inserting the instruction group

```
FMUL n,p,q; FADD r,s,t@2
```

into the queues shown in Figure 6a. Although the insertion of Figure 6b is allowable according to the rules set out in section 4.2, it will not be found as it is not a shift of the specified delays. In fact, the insertion logic will end up stalling for four cycles in this example before it is finally able to insert the instruction group.



(a) Initial state of queues **(b)** Allowable insertion

Figure 6: Inserting `FMUL n,p,q; FADD r,s,t@2` into the queues

The situation can be improved by making two observations. First, consider the subset of one or more instructions in the group which have minimal delay. By the comments of section 4.1, these instructions have no inter-dependencies and can therefore be independently inserted into the queues. Second, it is perfectly acceptable to perform a "partial insertion" whereby some subset of the minimal-delay instructions are inserted and the remaining instructions in the group stall until the next cycle.

These observations are not difficult to translate into hardware. In parallel with the computation of the bit vectors q_j, we can compute $\min(d_j)$ to identify the instructions with minimal delay, which can be inserted into their queues immediately. If D is their maximum depth of insertion, taking D = N if one of them could not be inserted, we clear bits 0 to D of the remaining bit vectors q_j and then proceed as before.

Note that this mechanism has the desirable property that when the compiler issues a group of instructions with no delays, each instruction will be inserted in the corresponding queue at the minimum allowable delay. This was not true of the mechanism presented in section 4.2; for example in Figure 6 an insertion still would not have been found if no delay was specified for the add instruction.

5.2 Independent Queue Motion

In some architectures it may be desirable to allow the queues to move independently. For example, suppose that one of the functional units performs a high-latency non-pipelined divide. Without independent queue motion, two consecutive divide instructions will cause all queues to stall until the first divide completes, even if independent instructions are waiting to be issued to other functional units.

Let the queues be divided into K independent sets such that within each set the queues move in lockstep. On each cycle, some of these sets may stall due to dependencies at the heads of the queues; this in turn may cause other sets to stall due to inter-queue dependencies. We must somehow determine which sets are allowed to advance so that ORD is satisfied. This is certainly a non-trivial problem as, depending on the arrangement of the instructions within the queues, a stall in one set could imply a stall in any other set. For small K (say K = 2 or 3) we can simply compute these dependencies directly.

In section 4.2 we defined the bit vectors w_d, r_d for each depth d. With K sets of queues that can move independently, these become bit vectors w_d^s, r_d^s, $1 \le s \le K$, where $w_d^s[k] = 1$ if register k is written by some instruction at delay d or greater in the s^{th} set, and $r_d^s[k] = 1$ if register k is read *or* written by some instruction at delay d or greater in the s^{th} set. Note that for the purposes of insertion, we can compute $w_d' = w_d^1 + w_d^2 + \ldots + w_d^S$, $r_d' = r_d^1 + r_d^2 + \ldots + r_d^S$, and proceed as before.

Say that the set S **depends** on the set T if a stall in T implies a stall in S. Specifically, S depends on T if some instruction at delay d in S has a data hazard dependency on some instruction at delay $d-1$ in T (in this case if T stalled and S did not, these instructions would end up at the same delay, violating ORD). Let $h_{st} = 1$ if the s^{th} set depends on the t^{th} set. Then:

$$h_{st} = 1 \iff (w_0^t r_1^s + r_0^t w_1^s) + (w_1^t r_2^s + r_1^t w_2^s) + \ldots + (w_{N-2}^t r_{N-1}^s + r_{N-2}^t w_{N-1}^s) \neq 0$$

Now let $u_s = 1$ if the s^{th} set must stall due to a dependency at the head of a queue, and let $v_s = 1$ if the s^{th} set must stall for any reason at all. Then v_s is what we are interested in computing, and it can be defined recursively as follows:

$$v_s = u_s + (h_{s1}v_1 + h_{s2}v_2 + \ldots + h_{s(s-1)}v_{s-1} + h_{s(s+1)}v_{s+1} + \ldots + h_{s(K-1)}v_{K-1}) \qquad (*)$$

Since this recursive definition is monotonic, it must stabilize some time after the h_{st} and the u_s become stable. Although the critical path depends on the h_{st} and the u_s, it traverses at most K gates of the form (*) since each traversal results in one of the v_s becoming 1. This scheme is therefore practical for small K.

6 Complexity Estimates

Table 1 gives an estimate of the number of transistors required to implement delayed issue with 5 queues of depth 8, 64 registers, with all queues moving in lockstep (second column), and with two sets of queues that can move independently (third column). Shown are transistor counts for the buffers themselves and for the expensive computations. The remaining components (sense amps, row decoders, shifters, random logic, etc.) consist of a few hundred transistors each and are lumped together in "overhead". We conservatively estimate the overhead at 10,000 transistors for a single set of queues and 15,000 transistors for two sets of queues. This gives totals of 67,016 transistors and 104,784 transistors respectively. By comparison, transistor counts for dynamic out-of-order issue logic range from 141,000 for a small window size of 20 instructions [Farrell98] to as many as 850,000 for two 28 entry reorder buffers [Gaddis96].

	1 set of queues	**2 sets of queues**
instructions/addresses	12,800	12,800
address compares	1,696	1,696
use/maintain busy bits	2,852	2,852
maintain r_d, w_d	8,192	36,864
compute q_j	10,240	10,240
update r_d, w_d	21,236	21,236
compute h_{st}	0	4,096
overhead	10,000	15,000
Total	67,016	104,784

Table 1: Estimated transistor counts for 5 queues of depth 8, 64 registers

With all queues moving in lockstep, the instructions, addresses, and bit vectors r_d, w_d can all be stored in register SRAM with an index pointing to the head of the queues. However, with multiple sets of queues that can move independently, the bit vectors r_d^s, w_d^s must be physically arranged by depth or the computation of the h_{st} becomes intractable. Thus, in this case they are stored in more complicated shift registers. This accounts for one doubling in the transistor count for r_d, w_d in the table; the other doubling results from maintaining separate bit vectors for both set of queues.

It should be noted that the critical path in the delayed issue logic can become dangerously long with multiple sets of queues. On each cycle, the following must occur in order:

1. The decision of which sets to advance must be made
2. The bit vectors w_d' and r_d' must be computed for the state resulting from the decided-upon advancements
3. An insertion must be found for the instruction group being considered
4. The instructions must be inserted and the bit vectors w_d^s, r_d^s must be updated

Each of these steps incurs multiple gate delays. Thus, implementing delayed issue presents a circuit design challenge as well as an algorithmic challenge.

7 Discussion and Future Work

In domains for which area efficiency is of primary concern, delayed issue provides a cost-effective method to improve performance via out-of-order execution. While the mechanisms required to correctly implement delayed issue are not trivial, they are significantly 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, especially when combined with hardware register rotation.

There is no unique approach to ensuring the correctness of delayed issue. Some techniques were discussed in this paper; the purpose of this was not present an "optimal" implementation, but rather to show the existence of good low-cost solutions. It is likely that other techniques exist which are as good as or better than those described herein.

One of the premises of delayed issue is that the compiler has intimate knowledge of the target architecture so that it may correctly predict when instructions would stall and should therefore be delayed. In particular, this

means that if an architecture were to undergo a revision affecting the internal timing of the functional units, code would have to be recompiled in order to take full advantage of the benefits offered by delayed issue. Note that old code would still run correctly on the new architecture due to the sequential ordering semantics of delayed issue; it would simply not be as fast as recompiled code, and could conceivably run slower on the new architecture than on the old one. A slight modification which partially avoids this problem is to allow the compiler to specify delays relative to functional unit latencies, e.g. "delay of adder plus one"; the actual delays can then easily be computed by looking up these latencies in a small ROM.

Delayed issue certainly presents some novel and interesting challenges to the compiler. One challenge in particular that must be dealt with carefully concerns memory references; the compiler should take great pains to ensure that at the time a delayed memory operation is decoded its address can be accurately resolved. If the compiler misses by even a single cycle, the address will appear in the delay queue as 'unknown' which can prevent other independent memory references from being inserted ahead of the delayed instruction.

This paper has laid the groundwork for delayed issue by defining the mechanism, identifying and addressing the challenges, and providing some candidate implementations. The next stage of our research will consist of experimentation conducted in simulation to determine its performance advantages. Running benchmarks in simulation will allow us to compare the various implementations of delayed issue to both fully in-order and fully out-of-order execution, thus providing a more quantitative description of its benefits.

References

[Butler92] Michael Butler, Yale Patt, "An Investigation of the Performance of Various Dynamic Scheduling Techniques", Proc. MICRO-25, 1992, pp. 1-9

[Chang91] Pohua P. Chang, William Y. Chen, Scott A. Mahlke, Wen-mei W. Hwu, "Comparing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors", Proc. MICRO-24, 1991, pp. 25-33

[Farrel98] James A. Farrell, Timothy C. Fischer, "Issue Logic for a 600-MHz Out-of-Order Execution Microprocessor", IEEE Journal of Solid-State Circuits, Vol. 33, No. 5, May 1998, pp. 707-712

[Farrens93] Matthew K. Farrens, Pius Ng, Phil Nico, "A Comparison of Supersaclar and Decoupled Access/Execute Architectures", Proc. MICRO-26, pp. 100-103

[Fisher81] Joseph A. Fisher, "Trace scheduling: A technique for global microcode compaction", IEEE Trans. on computers, Vol. c-30, no. 7, July 1981, pp. 478-490

[Fisher83] Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", Proc. 10th International Symposium on Computer Architecture, 1983, pp. 140-150

[Gaddis96] N. B. Gaddis, J. R. Butler, A. Kumar, W. J. Queen, "A 56-Entry Instruction Reorder Buffer", 1996 Digest of Technical Papers, ISSCC 1996, pp. 212-213

[Hara96] Tetsuya Hara, Hideki Ando, Chikako Nakanishi, Masao Nakaya, "Performance Comparison of ILP Machines with Cycle Time Evaluation", Proc. 23rd International Symposium on Computer Architecture, 1996, pp. 213-224

[Love90] Carl E. Love, Harry F. Jordan, "An Investigation of Static Versus Dynamic Scheduling", Proc. 17th International Symposium on Computer Architecture, 1990, pp. 192-200

[Palacharla97] Subbarao Palacharla, Norman P. Jouppi, J.E. Smith, "Complexity-Effective Superscalar Processors", Proc. 24th International Symposium on Computer Architecture, 1997, pp. 206-218

[Rau93] B. Ramakrishna Rau, "Dynamically Scheduled VLIW Processors", Proc. 26th International Symposium on Computer Architecture, 1993, pp. 80-92

[Smith82] James E. Smith, "Decoupled Access/Execute Computer Architectures", Proc. 9th International Symposium on Computer Architecture, 1982, pp. 112-119

[Smith86] James E. Smith, Shlomo Weiss, Nicholas Y. Pang, "A Simulation Study of Decoupled Architecture Computers", IEEE Transactions on Computers, Vol. c-35, No. 8, August 1986, pp. 692-701

[Thornton70] J.E. Thornton, "Design of a Computer – the Control Data 6600", Glenview, IL: Scott, Foresman and Co., 1970

[Tomasulo67] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, January 1967, pp. 25-33

[Weiss84] Shlomo Weiss, James E. Smith, "Instruction Issue Logic in Pipelined Supercomputers", IEEE Transactions on computers, Vol. C-33, No. 11, November 1984, pp. 1013-1022