Reducing the Complexity of the Issue Logic

Ramon Canal and Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3 Mòdul D6
Barcelona, E-08034

ABSTRACT

The issue logic of dynamically scheduled superscalar processors is one of their most complex and power-consuming parts. In this paper we present alternative issue-logic designs that are much simpler than the traditional scheme while they retain most of its ability to exploit ILP. These alternative schemes are based on the observation that most values produced by a program are used by very few instructions, and the latencies of most operations are deterministic.

Keywords: instruction issue logic, out-of-order issue, complexity-effective design, wide-issue superscalar.

1. INTRODUCTION

An out-of-order issue scheme is a common trend in today's high-performance microprocessor design due to its higher potential to exploit instruction-level parallelism (ILP) for some applications whose behavior is hard to predict at compile time (e.g. non-numeric applications such as SpecInt).

However, the hardware structures required by an out-of-order issue scheme are rather complex, which translates in significant delays that may challenge the cycle time [17]. In addition, the issue logic is responsible for a significant part of the energy consumed by a high-performance microprocessor [7]. The complexity and energy consumption of the issue logic depends on a number of microarchitectural factors, mainly the size of the instruction queue and the issue width [16,17,27]. These two parameters have experienced a continuous increase, and future projections suggest that this trend will continue in the near future. Therefore, the delay and energy consumption of the issue logic is expected to be even more critical in the future.

The main complexity of the issue logic comes from the associative search that is required by the issue logic, which relies on a *wake-up* and *select* mechanism [20]. The wake-up uses long wires to broadcast the tags (and data, sometimes) to the non-ready instructions, and a large number of comparators that compare each broadcast tag with every source operand's tag.

Besides, the issue logic is not suitable for a pipelined implementation [17] since this would significantly degrade performance by introducing some delay between the back-to-back execution of dependent instructions. Therefore, the issue logic may significantly impact the clock-cycle time. At the same time, the issue logic becomes a significant consumer of power. Recently, Gowan, Biro and Jackson [10] reported that in an Alpha 21264 operating at the maximum operating frequency the issue logic accounts for the 18% of the total power consumption. This percentage is even higher than the consumption of caches, which accounted just for 15% of the total.

To address this problem, various techniques have been proposed. These techniques attempt to partition the central instruction window by means of a clustered architecture where the partition is performed by either considering each instruction in turn [3] or managing larger instruction units such as trace cache lines [18] or loop iterations [13]. Another approach is to rely on the compiler to take this decision, VLIW [8] architectures are an example of that.

We take a different approach in this work to tackle this problem. This approach is based on the observed properties of typical dynamic instruction streams. We first note that a vast majority of the register values generated by a program are read at most once. For instance, only about one out of four values generated by the Spec95 benchmarks is read more than once [5]. This feature suggests that the wake-up function could be implemented through a simple table that is indexed by the register identifier, avoiding the associative search. The second observation is that the latencies of most instructions are known (except for the memory unit) and thus the time when a source operand will be available can be deterministically determined in most cases. This suggests a issue logic that schedules instructions based on the availability time of their operands.

We present different implementations of the issue logic based on the above two concepts and show that these schemes can significantly reduce the issue logic complexity with a minor impact on the IPC rate.

This paper is organized as follows. Sections 2 and 3 present alternative issue logic schemes. Section 4 analyzes the performance of these schemes and Section 5 reviews the related work. Finally, Section 6 summarizes the main conclusions of this work.

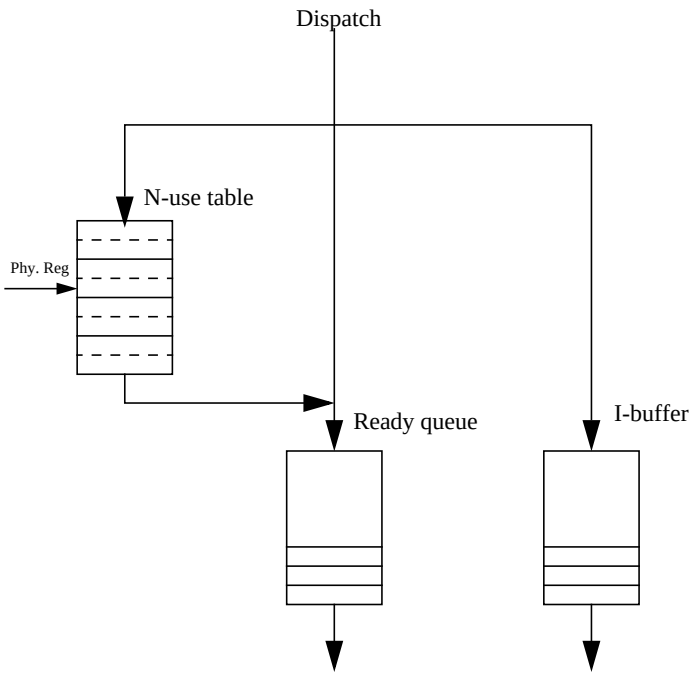

Figure 1: N-use issue logic design

2. N-USE ISSUE SCHEME

The *N-use scheme* is designed upon the observation that most register values are read very few times. For instance, only 22% of the values generated by the SpecInt95 and 25% of the FP register values produced by SpecFP95 are read more than once [5]. The N-use issue scheme is based on a table (we refer to it as N-use table) that, for each physical register, stores the first N instructions that read it in sequential order. We refer to the parameter N as the associativity degree of the issue logic. The scheme works as follows.

After being decoded, each instruction is dispatched in a different way depending on the availability of its source operands:

a) If all its operands are available, it is dispatched to a queue of ready instructions.

b) If for each non-ready source operands there is a free slot in the N-use table in the corresponding operand, the instruction is dispatched to the table entries corresponding to the non-ready operands.

c) If for any of the non-ready source operands there is not a free slot in the N-use table entry corresponding to the source operand, the dispatch of instructions is stalled until the operand becomes ready. Alternatively, the issue logic can be extended with an instruction queue where such instructions are dispatched either in-order or out-of-order. This queue could be small since it is used by few instructions.

Figure 1 shows a block diagram of the N-use issue scheme. This scheme consists of two main hardware parts. The first one is a ready queue, which contains instructions that have all their operands available. This queue issues the instructions to the functional units in-order. Instructions are dispatched to this queue if they meet the conditions in the above paragraph (a). The second component (N-use table) is a table that contains the first N instructions that read each physical register that is not available. The table will have N times the number of physical registers (N x #Physical Regs). Instructions are dispatched to this queue when they meet the conditions in the above paragraph (b). Since an instruction can have up to two source registers, it can be stored into two different entries, if both operands are not available. Each entry of the N-use table has an additional field that may point to another entry of the table. When an instruction is placed in two entries, each entry's pointer is set to point to the other entry. The pointer size is:

$$\lceil \log_2(N \times \#\,\text{Physical Regs}) \rceil .$$

When the execution of an instruction completes, its physical register identifier is used to index the N-use table. If an instruction is found in the corresponding entry, then the pointer field is analyzed. If it does not point to any other entry (i.e., the pointer's value is NIL), the instruction is forwarded to the ready queue, since this indicates that this register was the only one that was not available for that instruction. Otherwise, the pointer's value is used to access the other entry of the N-use table where the same instruction resides, and the pointer of that other entry is set to point to NIL. When the physical register corresponding to the entry is

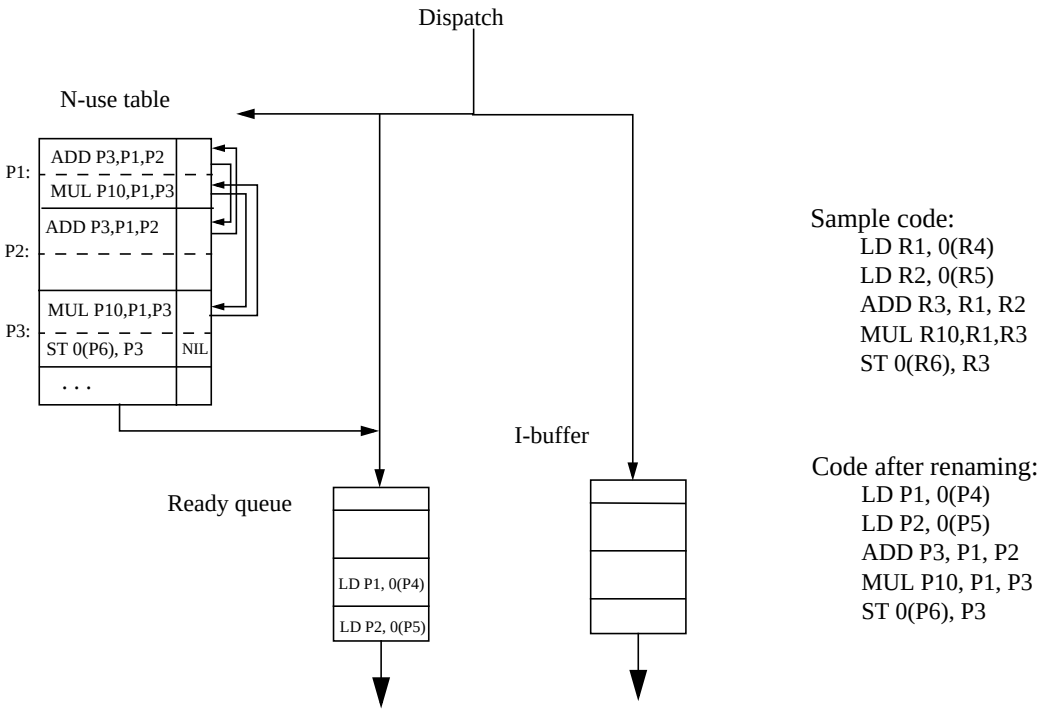

Figure 2: Example of the 2-use scheme

available, the instruction will be forwarded to the ready queue since the pointer that the processor will find when it accesses this entry will be NIL. Alternatively, the pointers could be eliminated if the N-use table would keep just the instructions with one (and only one) source operand which is in the first N-uses of that value. The difference with the previous scheme is that the instructions that have two source registers and both are in the first N-uses would now be sent to the I-buffer. Nevertheless, this alternative is beyond the scope of this work and its evaluation is currently on its way.

In the basic N-use scheme described above, if a decoded instruction has a source operand that is not ready and it does not correspond to the first N uses of such value, the decode and dispatch of instructions is stalled until this operand becomes ready. Then, it is dispatched to the ready queue. Alternatively, the basic scheme could be extended for increased performance with an extra buffer (I- buffer) shown in Figure 1. Basically, it consists of a buffer (I-buffer) where the instructions that have non-ready operands that are not in the first N uses of them are dispatched. The instructions from this I-buffer are issued to the functional units when their operands are available. We have investigated two different organizations for the I-buffer, with very different hardware cost and complexity: in-order and an out-of-order issue policies.

Note that both the basic scheme and the extended scheme where the I-buffer uses an in-order issue policy do not require any associative search for the issue logic, which is a significant simplification with respect to a conventional out-of-order issue mechanism.

In Figure 2, we can see an example of the use of the 2-use scheme for a sample code. We assume that all five instructions can be dispatched in the current cycle and that P4, P5 and P6 are available at this cycle. Since the two loads have their operands ready they are sent to the Ready queue. When dispatching the ADD instruction, the processor detects that this operation does not have their operands ready and that they both correspond to the first use of them. Thus, this instruction is steered to the N-use table into the entries corresponding to its source registers. Furthermore, each entry is made to point to the other one. As far as the multiply is concerned, it is the first use of P3 and the second of P1. Since this implementation allows 2 instructions (uses) per register in the N-use table this instruction is kept in the 2-use table. Regarding the store instruction, the operand P3 is not available and it is the second use of P3, thus the instruction is stored in the corresponding N-use table entry. The pointer is set to NIL since P3 is the only unavailable operand.

Let us suppose that the first load finishes before the second one. At that time, the entry P1 of the N-use table is checked. Since it has an instruction and its pointer is not NIL, the pointer of the pointed entry (i.e. the pointer of entry P2) is set to NIL. When the second load finishes, since the entry corresponding to its destination register (P2) has an instruction and its pointer is NIL, the ADD instruction is forwarded from the N-use table to the Ready queue.

In order to handle instruction squashing, for example in the case of a branch miss-prediction, each entry in the N-use table will hold a tag that indicates the position of that instruction in the ROB (ReOrder Buffer). When flushing the instructions, the tag of the misspredicted branch is sent to the N-use table. If an entry of the N-Use table holds an instruction, the tag of that will be compared to the one of the misspredicted branch. In case that the instruction has been fetched after the branch that entry will be set to invalid and thus squashed.

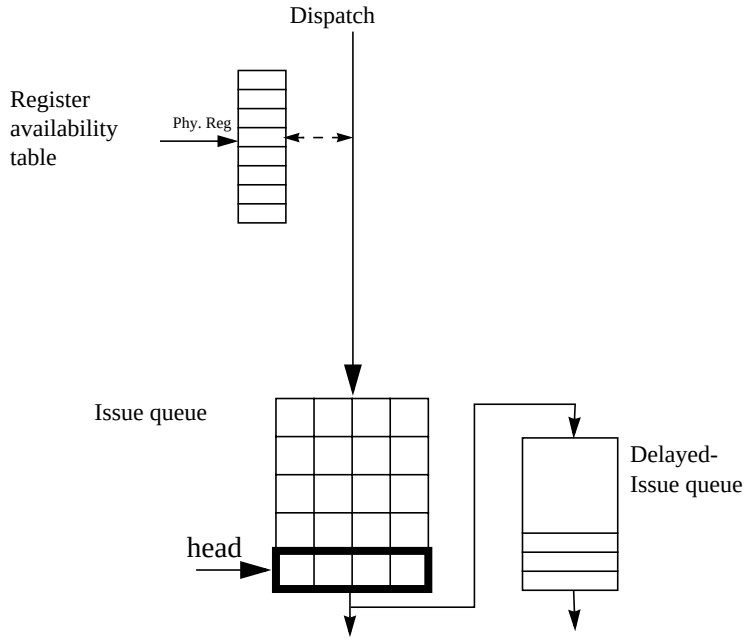

Figure 3: Deterministic Latency issue logic

3. DETERMINISTIC LATENCY SCHEME

Another approach to the issuing logic is based on the fact that the latency of most instructions is known when they are decoded. Thus, we could dynamically schedule the instructions following a similar approach to that implemented by a static scheduler. In other words, the order in which instructions will be executed is determined at the decode stage. The only problem this scheme has to face is the varying latency of the memory accesses. There are two main approaches when considering memory accesses. The first one is to assume that the latency is not known and thus, any instruction depending on a memory access will have to wait to be scheduled till the latency of the access is known. The second alternative is to assume that memory accesses have a deterministic latency and thus, the instructions depending on the memory access will be scheduled according to the assumed latency. Nevertheless, the issue will be stalled when the instruction depending on the memory access has to issue and the memory access has not finished yet. Alternatively, there could be a queue where the dependant instructions are kept if it is its expected issue time but they are not ready. The first scheme was previously studied and was referred to as *Distance* issue logic in [4]. The second alternative is called Deterministic Latency issue logic and will be described and evaluated in this work.

Figure 3 shows a block diagram of the Deterministic Latency issue logic. This scheme consists of three main blocks. First, we have a table where for each physical register it contains the cycle when its value will be produced. This table is called the register-availability table.

The second block is the Delayed Issue queue, which holds the instructions that were scheduled to be issued too early, before their operands were ready. This queue has a complexity similar to a traditional instruction issue queue since every time an instruction finishes its execution, its physical destination register is broadcast to all entries of this queue. Any entry with a matching source operand takes note of its readiness and will be issued the next cycle that there is an available issue slot.

The third block is the Issue queue. For each instruction in the queue, this block contains information regarding the cycle when the instruction will be issued. Conceptually, it can be regarded as a circular buffer where for each entry there are as many instruction slots as the issue width and each entry corresponds to one cycle. Thus, this queue contains the instructions in the order that they will be executed and separated by a distance that ensures that depences will be obeyed if at every cycle the processor issues the instructions in the head entry. Thus, the issue logic for this queue is very simple: at each cycle the instructions in the head of the queue are issued and the head pointer is increased by one. In Section 4 the depth of the Issue queue is empirically determined. Instructions are placed on the issue queue only when the time when its source operands will be available is known. The location in the issue queue is computed as follows. First, the maximum of the availability time of its source operands (*MaxSource*) is calculated and then, the difference between this value and the current cycle indicates the displacement in respect to the head pointer. The instruction is placed on the first free slot starting at that cycle.

Once an instruction is placed on the issue queue, the time when its output register will be available is computed as MaxSource plus the latency of the instruction plus the additional delay due to conflicts in the issue queue with previously scheduled instructions. This value is used to update the register availability table.

Loads from memory are handled in the following way. They are dispatched as any other instruction and they are assumed to have the latency of a hit in the first level cache (alternatively, they

could have the latency of a second level cache hit). Instructions that depend on the load read the output time and will be scheduled according to it. Nevertheless, it can happen that an instruction depending on a memory access is not ready to execute when it is at the head of the issue queue since the memory access has not finished. For example, an instruction depending on a load may assume that the memory value will be available 1 cycle after the load's issue. If the load misses in the first level cache, the data will not be there at that time so the instruction will have to be taken apart (kept in the delayed-issue queue) or the issue will have to be stalled. In this paper, we assume the first alternative and investigate the trade-off between the size of the delayed issue queue and performance.

In this mechanism, it is assumed that loads are completed in a certain latency so the instructions depending on the values produced by loads will be scheduled according to the latency assigned to the memory accesses. Please, note that since stores do not produce any output register the scheduling is no affected by the store latency.

Instructions are issued first from the delayed queue, and then from the Issue queue. Any instruction that cannot be issued from the Issue queue head, either because it was depending on a load and it has not finished or because the instructions of the Delayed queue took its issue slot or functional unit, will be kept in the Delayed queue. If the Delayed queue is full and there are some instructions in the Issue queue that should move into the Delayed queue, the issue will be stalled till the "delayed" instructions fit in the Delayed Issue queue.

The technique used for instruction squashing is similar to that presented for the N-Use scheme in Section 2. In this case, the tags will be held in the Issue queue. No work needs to be done in the Register-Availability Table since during the squashing the physical registers will be deallocated (and thus not referenced). Next time the physical register is assigned (as an output register of a new operation) the availability time will be set by the new instruction.

4. PERFORMANCE EVALUATION

4.1. Experimental Framework

We have used a cycle-level timing simulator based on the SimpleScalar tool set [2] for performance evaluation. We extended the simulator to include register renaming through a physical register file and the issue mechanisms described in Section 2 and Section 3. See Table 1 for the main architectural parameters of the machine. We used the programs from the Spec95 suite with their reference inputs to conduct our evaluation. All the benchmarks were compiled with the Compaq-Alpha C compiler with the -O5 optimization flag. For each integer benchmark, 100 million instructions were run after skipping the first 100 million. For the FP programs, 100 million instructions were run after skipping the first 300 million. Performance results are reported as the harmonic mean for the whole benchmark suite. We have simulated four issue schemes: a conventional out-of-order scheme with a 64-entry instruction window, an in-order scheme, the N-use scheme (for several values of N) and the Deterministic Latency scheme (assuming that the latencies of the loads are either hit time in first level cache or hit time in second level cache). The next subsections present performance figures for all these schemes.

Table 1: Machine parameters (split into the integer datapath and the FP datapath if not common)

Parameter	Configuration	
Fetch width	8 instructions	
I-cache	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty	
Branch Predictor	Combined predictor of 1K entries with a Gshare with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 2K entries with 2-bit counters. Br. miss-prediction penalty: 3 cycles	
Decode/Rename width	8 instructions	
Max. in-flight instructions	64	
Retire width	8 instructions	
Functional units (latencies)	3 intALU (1) + 1 int mul/div (2, 14)	3 fpALU (2) + 1 fp mul/div (6,12)
Issue mechanism	4 instructions	4 instructions
	Depends on the mechanism studied Loads may execute when prior store addresses are known	
Physical registers	96	96
D-cache L1	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty	
	3 R/W ports	
I/D-cache L2	256 KB, 4-way set associative, 64-byte lines, 6-cycle hit time.	
	16 bytes bus bandwidth to main memory, 16 cycles first chunk, 2 cycles interchunk.	

4.2. N-Use Scheme

Figure 4 shows the IPC of the basic N-use mechanism (i.e. without an I-buffer) in comparison to an in-order issue and an out-of-order issue. We can see that the 1-use scheme performs better than an in-order issue mechanism but significantly worse than an out-of-order issue scheme. Actually, what happens is that the out-of-order scheme can find instructions ready to execute further in the code sequence than the N-use scheme since the latter stalls the dispatch much more frequently. The 1-use scheme achieves and speed-up of 38% over the in-order scheme but it slows down the out-of-order machine by a 40%. For bigger associativities of the N-use table, the performance is very close to that of an out-of-order mechanism since with this configuration the issue is stalled less frequently due to the bigger capacity of the N-use table.

When the N-use scheme is extended with a a small out-of-order I-buffer (see Figure 1) the performance of this scheme significantly increases. Figure 5 shows the evolution of the IPC when the size of the I-buffer varies. We can see that for sizes of the I-buffer of 8 elements or more, the N-use scheme performs almost at the same level as the out-of-order issue mechanism. When reducing the I-buffer size further, the overall effect depends on the associativity of the N-use table. The higher the associativity the minor the effect of the smaller I-buffer. For a 4-entry I-buffer, the performance degradation is very low whereas without a I-buffer,

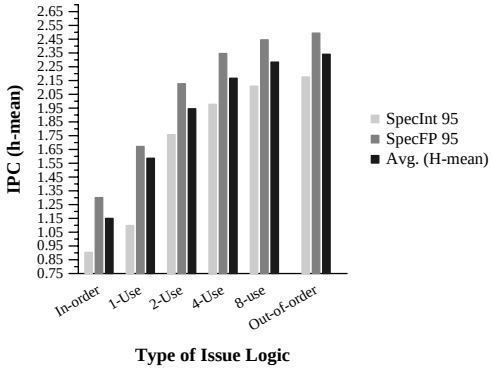

Figure 4: Performance of the N-use scheme (without I-buffer)

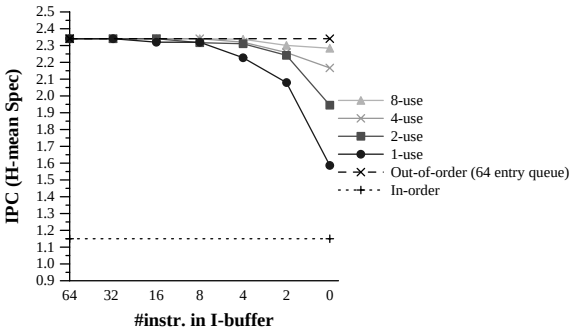

Figure 5: Evolution of the IPC for the N-use scheme for different I-buffer sizes with an out-of-order I-buffer

the degradation varies from 32% to 2.5% depending on the associativity. .

In particular, the 2-use scheme with a 2-entry I-buffer achieves an IPC comparable (~4% slow-down) to that of an out-of-order scheme and it reduces the associative look-up from 64 to 2 entries (32 times smaller). This restricted associative search will certainly result in a shorter issue delay which may in turn influence the clock-cycle time.

Alternatively, we could get rid of any associative search logic by implementing an in-order issue for instructions in the I-buffer. The performance of this alternative is shown in Figure 6.

We can see in Figure 6 that this scheme implies a decrease in IPC with respect to the out-of-order I-buffer configuration (see Figure 5) for the 1-use scheme whereas it is rather low for higher degrees of associativity. It is interesting to analyze the impact of the size of the in-order I-buffer on performance. A bigger I-buffer reduces the stalls in the dispatch. However, since instructions from the I-buffer are issued in order, once an instruction is placed on this buffer it must wait until all previous instructions have been issued. However, sometimes it is better to stall the dispatch for a few cycles and then issue the instruction to the N-use table, from where it can issue out of order. This trade-off explains why the IPC increases when the I-buffer size increases, but beyond a certain size (8 entries), the benefits of a larger I-buffer are more than offset by its drawbacks, which results in a decrease in performance. This effect

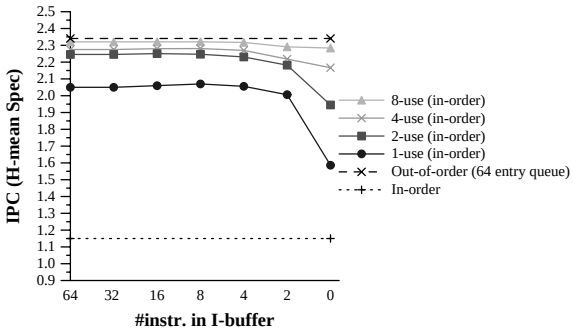

Figure 6: Performance of the N-use scheme for different in-order queue sizes of an in-order I-buffer

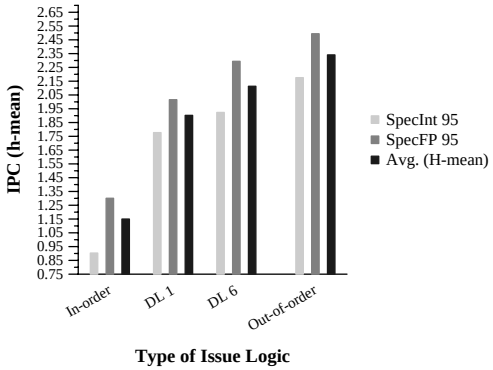

Figure 7: Performance of the basic Deterministic Latency scheme (without a Delayed Issue queue)

is minimized by the associativity of the N-use table. For smaller values of N the effect is more visible. Overall, the performance of the N-use scheme is quite close to that of an out-of-order scheme for a small I-buffer or associativities higher than 1. For associativity 1 and a large I-buffer the performance is somewhat lower (about 12% lower on average).

4.3. Deterministic Latency Scheme

Figure 7 shows the IPC of a basic implementation of the Deterministic Latency scheme when memory instructions are scheduled assuming a 1-cycle -DL1- and 6-cycle -DL6- memory access latency. An in-order issue approach and an out-of-order issue mechanism are also shown for comparison. This basic implementation does not require any associative search since it assumes a zero-entry Delayed Issue queue (see Figure 3). We can see that the DL mechanism performs much better than an in-order machine (65% and 84% speed-up for the DL1 and DL6, respectively) and not very far from an out-of-order machine (10% slow-down for the DL6).

When a full implementation of the Deterministic Latency scheme is considered, the IPC is significantly improved. Figure 8 shows the evolution of the IPC when the size of the Delayed queue varies. The size of the queue hardly affects the performance of the DL6 scheme. Although the performance grows up when the size increases, there is a point where the extra cycles spent for every access (this scheme assumes that memory accesses take 6 cycles)

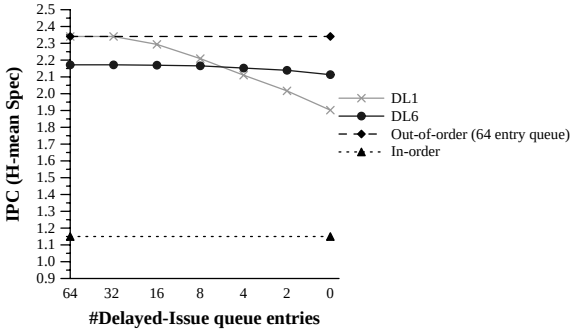

Figure 8: Performance of the Deterministic Latency scheme for different Delayed Issue queue sizes

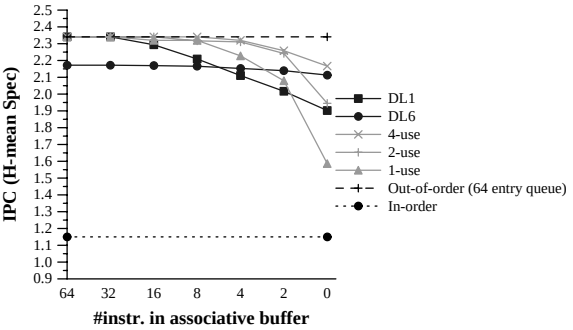

Figure 9: Performance of the two proposed schemes

limit the maximum performance achievable. As far as the DL1 is concerned, we can see that with 64 to 16 elements in the associative part of the mechanism (Delayed Issue queue) the scheme performs almost at the same level (~4% slow-down) as the out-of-order approach. For smaller sizes, the performance goes down significantly.

We have empirically determined that the maximum depth that the Issue queue of the Deterministic Latency scheme requires in order not to cause any stall is 60 entries of 4 instructions each and, on average, it is around 18 entries for the Spec benchmarks for the 1-cycle memory access latency configuration. Note also that for integer codes the size of the Issue queue is smaller due to the shorter latencies of the functional units. Studying the effect of a limited Issue queue is beyond the scope of this work.

4.4. N-Use vs. Latency

Figure 9 shows the performance of both the N-use and the Deterministic Latency schemes when varying the size of the associative hardware. We can see that when there is no associative buffer, the Deterministic Latency scheme and the 4-Use scheme perform at the same level. Besides if a small associative buffer (2 or 4 entries) is feasible, the N-use scheme performs better. And, in this case, the performance achieved is very close to that of an out-of-order mechanism.

Thus, we can conclude that if we can afford a small associative buffer (around 2-4 entries), the N-use scheme is very competitive in terms of IPC when compared to a fully out-of-order mechanism and it has the potential advantage of reducing the cycle time. If we want to completely avoid any associative search, both the Deterministic

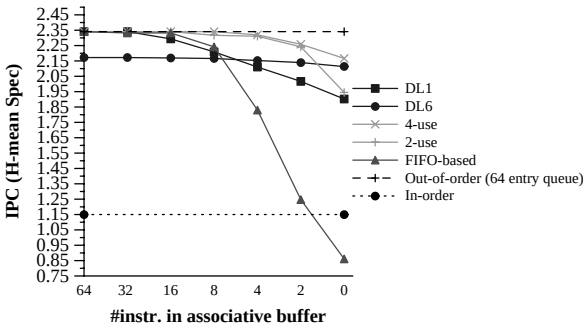

Figure 10: Performance of the FIFO scheme together with the N-use and the Deterministic Latency

Latency scheme without the Delayed queue and the N-Use scheme with a moderate associativity in the N-use table with an in-order I-buffer have about the same IPC, which is significantly higher than that of an in-order scheme (~84%) and not far from the performance of an out-of-order mechanism (~10%).

4.5. Comparison with the FIFO-Based Scheme

In this Section, we will compare the proposed mechanisms to the FIFO-based scheme proposed by Palacharla et al. [17]. In Palacharla's scheme instructions are dispatched to several FIFO queues depending on the following facts:

d) If the last instruction of a queue produces one of the registers of the instruction being dispatched, this instructions is sent to that queue.

e) Otherwise it is sent to an empty FIFO queue. If there are no queues available, the dispatch is stalled until this instruction has an empty FIFO.

Figure 10 shows the performance of the FIFO scheme in comparison to the N-use scheme and the Deterministic Latency scheme. For the FIFO-based scheme, the X-axis corresponds to the number of FIFO queues (note that the zero position, means zero entries in the extra buffer for the DL and N-use schemes and 1 single FIFO for the FIFO-based mechanism). The number of FIFO queues determines the degree of associativity of the issue logic since instructions of different queues are issued in any order whereas those in the same queue are issued in-order. The number of entries in the FIFO queues for each configuration is determined by the number of entries in the instruction window (64) divided by the number of queues. We can see that the performance of all the schemes is similar for a degree of associativity of 16 or more (except for the Deterministic Latency that assumes 6 cycles for the memory accesses). When reducing the associativity the FIFO scheme reduces dramatically its performance due to lack of empty FIFO queues when dispatching.

In conclusion, the N-use scheme and the Deterministic Latency scheme have a much better performance than the FIFO based for low degrees of associativity.

5. RELATED WORK

In our previous work on issue logic designs [4], we proposed the First-Use scheme and the Distance scheme. The former has been extended so that the First-use table now can hold more than one

instruction (the N-use table in this work). This significantly improves the performance of the scheme for low sizes of the I-buffer, as shown in the previous section. The Distance scheme is the complementary of the Deterministic Latency scheme. In that case, the memory access latency is assumed unknown (indeterministic) so all the instructions depending on a memory access are kept apart and are not placed into the Issue queue till the availability time of the data is known (in other words, when the load performs its writeback the availability time is updated). In the distance scheme there is no delayed-issue queue but needs a wait queue to keep the instructions whose operand's availability time is not known. That scheme performs lower than the Deterministic Latency schemes proposed in this work.

Michaud et al. [15] presented a prescheduling technique similar in concept to that presented in our previous work [4] and to the Deterministic Latency scheme in this work. In their case, they did not work directly on the issue logic but on preparing (prescheduling) the instruction for that stage. A deeper pipeline was used and thus more complex sturctures could be used.

Henry et al. [11] presented several circuit-level techniques for reducing the complexity of the issue logic. Their approach is orthogonal to ours and thus both could combine nicely.

S. Weiss and J.E. Smith [25] designed an issue logic similar to the basic 1-Use mechanism with no associativity explained in Section 2. In their work, they did not implement an I-queue and the first use check was done through a tag mechanism. We have shown that the mechanism suffers significant performance degradation when the I-buffer is not present.

S. Palacharla and J.E. Smith [17] presented an approach based on implementing the instruction queue through several FIFOs so that just the heads of the FIFOs need to be considered for issuing. In the previous section we have evaluated the performance of this scheme for different configurations and compared it to the mechanisms proposed in this work. We have showed that for a small degree of associativity (number of FIFOs), this mechanism reduces performance dramatically whereas for larger associativities its performance is similar to the schemes proposed in this work.

S. Öner and R. Gupta [16] proposed a mechanism which tried to chain the instructions according to the dependences among them. This scheme, builds the data dependence graph dynamically and limits the number of instructions that each instruction can wake up. In their study they assumed that all the FUs latencies are known and not variable, which simplifies the issue logic.

The most common approach to reduce the complexity of the issue logic is to take advantage of a VLIW architecture [8]. In this case the scheduling is done at compile time. This is the approach that reduces most the issue logic complexity but, on the other hand, it is not as flexible as a dynamic scheme since not all the information is available at compile time (e.g. memory access latencies).

More general approaches to reduce the complexity of the issue logic and, in general, of the microarchitecture are clustering and multithreading. These approaches try to reduce the complexity of the issue logic by partitioning it into several parts. The cluster approach [3, 6, 12, 17] partitions the datapath whereas in the multithreading approach [23, 26], each thread may have its own issue logic. Both architectural approaches are orthogonal with the research conducted in this work and both could combine nicely.

6. CONCLUSIONS

Out-of-order issue is a key component for high-performance in non-numeric applications. For instance, we have observed that for the SpecInt95, an out-of-order implementation achieves an IPC (instruction committed per cycle) that is about 2.5 times higher than that of a similar processor with an in-order issue scheme. This difference could be reduced by a compiler that was aware of the underlying in-order issue scheme and generated a more optimized code for this case, but we may still expect significant differences since the performance of instruction scheduling for non-numeric codes is significantly constrained by the small size of basic blocks and the large percentage of ambiguous memory references.

However, implementing an fully out-of-order issue scheme in the way that current microprocessors do, is very complex and energy-demanding, which poses significant constraints to its scalability. In this paper we have proposed alternative implementations of an out-of-order issue scheme that are much simpler and retain most, if not all, the ability of the fully out-of-order mechanism to exploit instruction-level parallelism.

In particular, the N-use scheme does not require any associative search and provides a performance very close the out-of-order scheme for N equal to 4. The cost of this scheme is a table with as many entries as number of physical registers but this table does not require any associative search, i.e., it is indexed by the operand register identifiers). Another alternative is the Deterministic Latency scheme that provides a performance very close to the out-of-order processor and at the same time it constraints the associative searches to a small queue of 8-16 entries.

7. ACKNOWLEDGEMENTS

This work has been supported by the Spanish Ministry of Education under grant CICYT TIC98-0511, the Generalitat de Catalunya under grant 2001TDOC00049 and the IBM University Partnership Award. This work has been done using the resources of the European Center of Parallelism in Barcelona (CEPBA). Ramon Canal would like to thank his fellow PBC's for their patience and precious help.

8. REFERENCES

[1] M.T. Bohr. Interconnect Scaling - The Real Limiter to High Performance VLSI. *Proc. of the 1995 IEEE Int. Electron Devices Meeting*, pp. 241-244, 1995.

[2] D. Burger, T.M. Austin, S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. *Technical Report CS-TR-96-1308, University of Wisconsin-Madison, 1996.*

[3] R. Canal, J.M. Parcerisa, A. González. Dynamic Cluster Assignment Mechanisms. *Proc. of the 6th Int. Symp. on High-Performance Computer Arch.*, Toulouse, 2000, pp. 133-142.

[4] R. Canal, A. González. A Low-Complexity Issue Logic. *Proc. of the 2000 International Conference on Supercomputing . Santa Fe (NM-USA).* May, 2000, pp. 327-335.

[5] J.Ll. Cruz, A. González, M. Valero, N. Topham. Multiple-Banked Register File Architectures. *Proc. of the 27th Int'l Symp. on Computer Architecture, June 2000*, pp.316-324.

[6] K.I.Farkas, P.Chow, N.P.Jouppi, Z.Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. *Proc of the 30th. Ann. Symp. on Microarchitecture*, December 1997, pp149-159

[7] D. Folegnani, A. Gonzalez. Reducing Power Consumption of the Issue Logic. *Workshop on Complexity-Effective Design, Vancouver, June 2000*.

[8] J.A. Fisher. Very Long Instruction Word and ELI-512. *Proc. of the 10th Int. Symp. on Computer Architecture*, Stockholm, Sweden, June 1983, pp. 140-150.

[9] M. Franklin. The Multiscalar Architecture. *Ph.D. Thesis, Technical Report TR 1196, Computer Sciences Department, Univ. of Wisconsin-Madison, 1993.*

[10] M.K. Gowan, L.L. Biro, D.B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. *Proc. of the 35th ACM/IEEE conference on Design Automation Conference (DAC 98), San Francisco (CA-USA), 1998*, pp. 726-731

[11] D.S.Henry, D.C.Kuszmaul, G.H.Loh, R.Sami. Circuits for Wide-Window Superscalar Processors. *Proc. of the 27th Int. Symp. on Computer Architecture*, Vancouver, Canada, June 2000, pp. 236-247.

[12] G.A.Kemp, M.Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. *Proc. of the Int. Conf. on Parallel Processing*. 1996, v.1, pp 239-246.

[13] P. Marcuello, A. Gonzálex, J. Tubella. Speculative Multithreaded Processors. *Proc. of the Int. Conf. on Supercomputing*, pp. 77-84, July 1998.

[14] D.Matzke. Will Physical Scalability Sabotage Performance Gains. *IEEE Computer* Vol. 30, num. 9, pp.37-39, September 1997.

[15] P. Michaud, A. Seznec. Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. *Proc. of the 7th Int. Symp. on High-Performance Computer Architecture,* Monterrey, Mexico, 2001, pp. 27-36.

[16] S. Önder, R. Gipta. Superscalar Execution with Dynamic Data Forwarding. *Proc. Int. Conference on Parallel Architectures and Compilation Techniques*, pp. 130-135, 1998.

[17] S. Palacharla, N.P. Jouppi, J.E. Smith. Complexity-Effective Superscalar Processors. *Proc of the 24th. Int. Symp. on Comp. Architecture*, 1997, pp 1-13.

[18] E.Rotenberg, Q.Jacobson, Y.Sazeides and J.E.Smith. Trace Processors. *Proc of the 30th. Ann. Symp. on Microarchitecture*, 1997.

[19] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors. 1997.

[20] J.E. Smith, G.S. Sohi. The Mircoarchitecture of Superscalar Processors. Proc. of the IEE, vol. 83, no. 12, december 1995, pp. 1609-1624.

[21] G.S.Sohi, S.E.Breach, T.N.Vijaykumar. Multiscalar Processors. *Proc. of the 22nd Int. Symp. on Computer Architecture*. 1995, pp 414-425.

[22] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* vol 11, pp 25-33, 1967.

[23] D.M. Tullsen, S.J. Eggers, H.M.Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. of the Int. Symp. on Computer Architecture*, pp. 392-403, 1995.

[24] D.W. Wall. Limits of Instruction-Level Parallelism. *Techincal Report WRL 93/6*, Digital Western Research Lab, 1993.

[25] S. Weiss, J.E.Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE transactions on computers, vol. c-33, no.11*, pp 1013-1022, November 1984.

[26] W. Yamamoto, M. Nemirovsky. Increasing superscalar performance through multistreaming. *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 49-58, 1995

[27] V.V. Zyuban. Inherently Lower-Power High-Performance Supersalar Architectures. *PhD. Thesis, Dept. of Computer Science and Engineering, University of Notre Dame, (Indiana)*, January 2000.