A Low-Complexity Issue Logic

Ramon Canal and Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3 Mòdul D6
08034 Barcelona, Spain
{rcanal,antonio}@ac.upc.es

Abstract

One of the main concerns in today's processor design is the issue logic. Instruction-level parallelism is usually favored by an out-of-order issue mechanism where instructions can issue independently of the program order. The out-of-order scheme yields the best performance but at the same time introduces important hardware costs such as an associative look-up, which might be prohibitive for wide issue processors with large instruction windows. This associative search may slow-down the clock-rate and it has an important impact on power consumption. In this work, two new issue schemes that reduce the hardware complexity of the issue logic with minimal impact on the average number of instructions executed per cycle are presented.

Keywords: instruction issue logic, wide-issue superscalar, out-of-order issue, in-order issue.

1. Introduction

A roadmap to higher processor performance is based on increasing the instruction-level parallelism (ILP), which typically is measured as the average number of instructions committed per cycle (IPC). Dynamically scheduled superscalar processors are the predominant organization nowadays. In general, their out-of-order issue mechanism can achieve higher IPC rates than an in-order issue, although the difference may not be high for some numeric applications, which are typical for example in the embedded market.

A high IPC rate obviously implies an issue mechanism capable of issuing multiple instructions in parallel. Dynamically-scheduled superscalar processors that can issue a large number of instructions per cycle will become feasible with the advances in manufacturing techniques [14]. On the other hand, providing a wide issue logic is not sufficient for exploiting greater degrees of ILP. With a wider issue logic, the instruction window has also to be enlarged [18]. However, designing a large instruction window together with a wide issue logic does not come for free. The delay of the issue logic has been shown to depend quadratically on the product of instruction issue width times window size [12]. This delay is mainly due to the associative search that is required by the issue logic, which relies on a *broadcast* and *select* mechanism [16]. This delay is caused by the long wires used to broadcast the tags (and data, sometimes) to the non-ready instructions, and the large number of comparators needed to implement an associative search in the instruction window [1, 10] in order to wake-up the instructions that are waiting for the result produced. As discussed by other authors [12], the issue logic is not suitable for a pipelined implementation since it would introduce a delay in the back-to-back execution of dependent instructions and thus, it may significantly impact the clock-cycle time.

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 [13] or loop iterations [9].

In this work, we target this problem through a different approach that is based on the observed properties of typical dynamic instruction streams. The first feature is the fact that a very high percentage of the register values are read at most once. For instance, only about one out of every four values generated by the Spec95 benchmarks are read more than once [4]. This observation motivates an issue logic with a limited (if any) associative look-up logic that is required only by very few instructions. The second proposed approach relies on the fact that the latencies of

Copyright notice for the Proceedings of the 14th International Conference on Supercomputing (ICS-00). Santa Fe (NM-USA). May 8-11, 2000.

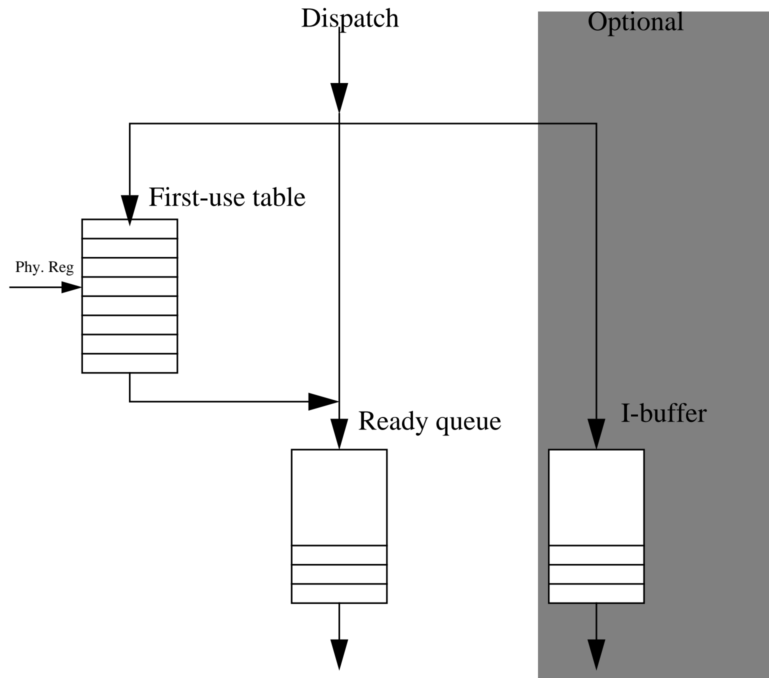

Figure 1: First-use issue logic design.

the functional units are known (except for the memory unit) and thus the time when an output register will be available can be determined if the starting time is known. Therefore, many instructions could be scheduled according to the cycle where their operands will be ready. We show that these schemes can significantly reduce the issue logic complexity with a minor impact on the IPC rate.

The rest of this paper is organized as follows. The two proposed issue schemes are presented in Sections 2 and 3 respectively. 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.

2. First-Use Issue Scheme

The first proposed scheme relies on the fact that most output register values are read at most once. In particular, only 22% of the values generated by the SpecInt95 and 25% of the FP register values produced by SpecFP95 are read more than once [4]. Based on this observation, we propose an issue logic that is based on keeping track of the instruction that is the first use of each produced register value. 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 all non-ready source operands represent the first use of the corresponding operand values (i.e., this instruction is the first one in sequential order that reads these operand values), the instruction is dispatched to a table that is indexed by the physical register identifier.

c) If any non-ready source operand is not the first use of the corresponding operand value, the dispatch of instructions is stalled until the operand becomes ready. Alternatively, the issue logic could be extended with an instruction queue where such instructions are dispatched. This queue could be small since it is used by few instructions. We have investigated the trade-off between performance and size of this queue. Besides, we have evaluated two alternative organizations for this queue: 1) an in-order issue scheme (i.e., instructions of this queue are issued in the same order as they are dispatched to the queue) or 2) an out-of-order issue (i.e., instructions of this queue can be issued in any order).

Figure 1 shows a block diagram of the First-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 and there is a separate issue queue for each functional unit type.

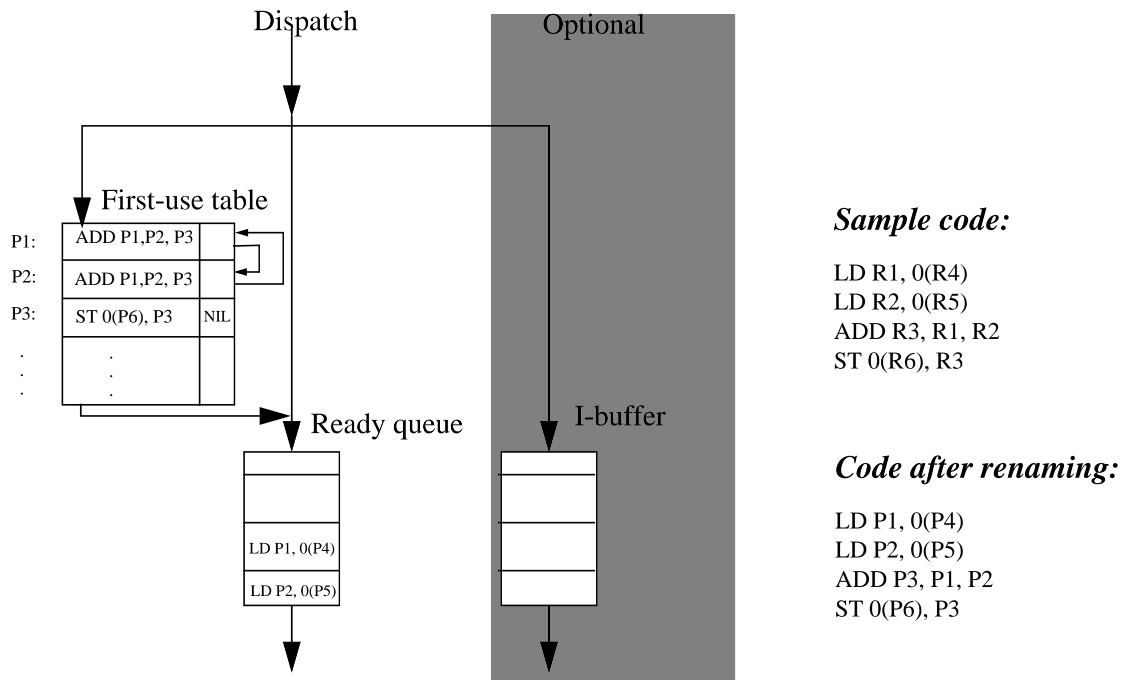

Figure 2: Example of the First-use scheme.

Instructions are dispatched to this queue if they meet the conditions in the above paragraph (a). The second component (First-use table) is a table that contains the oldest instruction that reads a register (i.e., its First-use) for each physical register that is not available. 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 copied into two different entries, if both operands correspond to the first use of the corresponding physical registers. For this case, each entry of the First-use table has an additional field that points 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.

When the execution of an instruction completes, its physical register identifier is used to index the First-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 First-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 available, the instruction will be forwarded to the ready queue since the pointer that the processor will find when it access this entry will be NIL.

In the basic First-use scheme described above, if a decoded instruction has a source operand that is not ready and it does not correspond to the first use 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 the hardware shown in the shaded part of Figure 1. Basically, it consists of a buffer (I-buffer) where the instructions that have non-ready operands that are not the first use 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 costs: 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 in respect to a conventional out-of-order issue mechanism.

In Figure 2, we can see an example of the use of the First-use scheme for a sample code. We assume that all four instructions can be dispatched in the current cycle and that P4, P5 and P6 are available at this cycle. Since the two

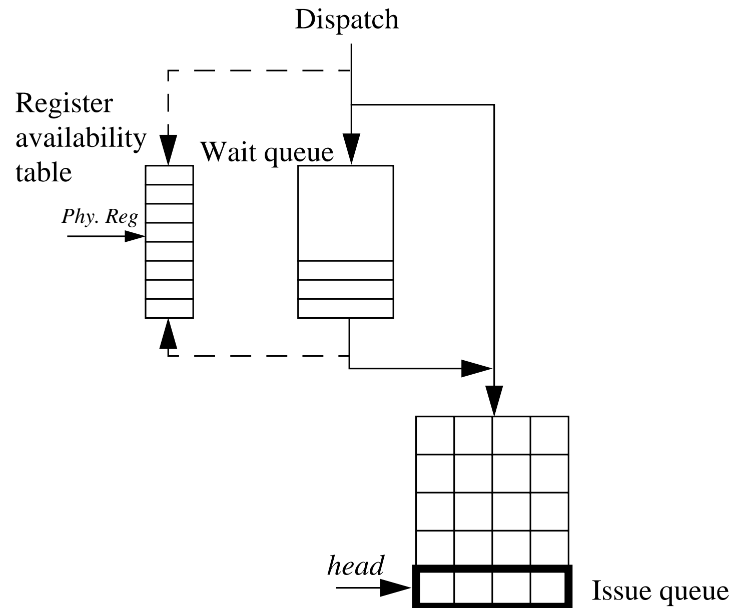

Figure 3: Distance scheme issue-logic.

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 its operands ready and it is the first use of them both. Thus, this instruction is steered to the First-use table into the entries corresponding to its source registers since both are the first use. Furthermore, each entry is made to point to the other one. Regarding the store instruction, the operand P3 is not available and thus the store is stored in the corresponding First-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 First-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 First-use table to the Ready queue.

3. Distance 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. In this case, the scheme provides an extra hardware that will keep the instructions as far as the cycle when their operands will be available is not known. We refer to this scheme as Distance issue logic.

Figure 3 shows a block diagram of the Distance 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, if it is known. This table is called the register-availability table.

The second block is the Wait queue, which holds the instructions that do not know when one (or both) of their operands will be produced. 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 availability time (i.e., the current cycle). When an instruction in the Wait queue knows the availability time of all its source operands, it is removed from the queue and placed in the Issue queue. Besides, the time when its destination register will be available is computed and it is used to update the register availability table. The identifier of its destination register along with its availability time is then broadcast to the Wait queue.

The third block is the Issue queue. Instructions are issued always from this 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 with 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 and it is broadcasted to the Wait queue.

Loads from memory are handled in the following way. They are dispatched as any other instruction but the entry in the register-availability table of the output register is set to *unknown*. Instructions that depend on the load will be held in the Wait queue since the availability time of their operands is unknown. When the load performs the write-back stage, it will update the entry in the register-availability table with the current cycle and it will broadcast the destination register of the load and the current cycle to the Wait queue. This may wake up some instructions that use the value produced by the load. Similarly, these instructions will update the entry of its output register in the register-availability table and wake up other instructions in the Wait queue, and so on.

3.1. Overlapping issue time computation and decode/renaming

The Distance issue logic presented above can work in parallel with the decode/renaming stage of the pipeline. The scheme requires some hardware modifications since the register-availability table should work with logical registers instead of physical ones in order to overlap the issue time computation with the renaming. Thus, the register-availability table will have as many entries as logical registers the ISA has since it will be looked up before renaming. In addition, there will be an extra table indexed by physical register identifiers that says if each physical register is the current mapping of a logical one. This table is needed for all the instructions that cannot write its output register availability time at dispatch time and thus, they are sent to the Wait queue. In this case, when they are woken up, they update the register-availability table entry corresponding to its destination logical register only if its physical destination register is the current mapping of the logical one. Note that if we stall the issue when an instruction has unknown times for one of its operands (i.e., there is no Wait queue), we do not need this extra table. This is due to the fact that we will always update the output register entry in the register-availability table in the dispatch stage. Furthermore, when recovering from a miss-prediction, these tables will have to be restored with the values they had before the misprediction, just as the register-renaming table is restored, since they work with logic registers and the tables are updated at dispatch time when it is not known whether one instruction is in the wrong-path or not.

To summarize, the activity of determining the issue cycle of an instruction can be totally overlapped with the renaming stage since it can work with the identifiers of the logic registers.

The benefit of overlapping the issue mechanism with the decode/renaming stage is that we could (depending on the distribution of the stages in the pipeline) reduce by one the depth of the pipeline, which has some benefits such as reducing the branch miss-prediction penalty. However, in the evaluation presented in this paper, this effect is not taken into account.

4. Performance Evaluation

4.1. Experimental Framework

We have used a cycle-based 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 SpecInt95 to conduct our evaluation with the inputs listed in Table 2. All the benchmarks were compiled with the Compaq-Alpha C compiler with the -O5 optimization flag. For each benchmark, 100 million instructions were run after skipping the first 100 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, an in-order

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.	
Decode/Rename width	8 instructions	
Max. in-flight instructions	64	
Retire width	8 instructions	
Functional units	3 intALU + 1 int mul/div	3 fpALU + 1 fp mul/div
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.	

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

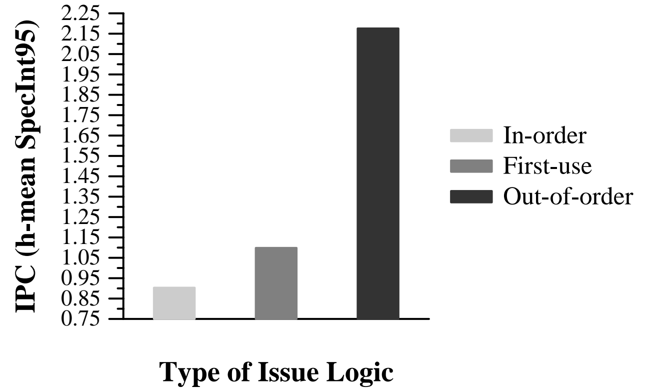

Figure 4: Performance of the basic First-use scheme (without I-buffer).

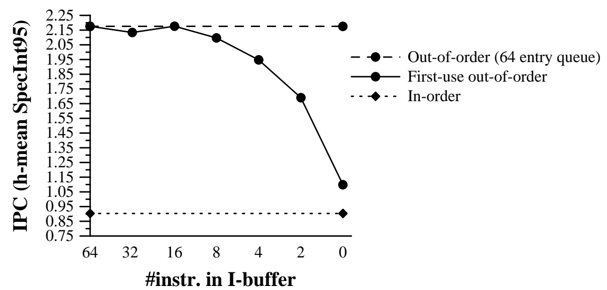

Figure 5: Evolution of the IPC for the First-use scheme for different I-buffer sizes with an out-of-order I-buffer.

scheme, the First-use scheme and the Distance scheme. The next subsections present performance figures for all these schemes.

4.2. First-Use Scheme

Figure 4 shows the IPC of the basic First-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 First-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 First-use scheme since the latter stalls the dispatch much more frequently. The First-use scheme achieves and speed-up of 21.6% over the in-order scheme but it slows down the out-of-order machine by half.

When the First-use scheme is extended with a a small out-of-order I-buffer (the shaded part of 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, if the I-buffer has 8 elements or more, the First-use scheme performs almost at the same level as the out-of-order issue mechanism. The peak at 16 is due to the fact that two benchmarks (m88ksim and li) have a very good performance in this case. This is due to the fact that we are using priorities when deciding which ready instruction will be executed (first branches, memory operations and long-latency operations in this order, and then program order). Due to this prioritization, younger instructions may take over older ones when we use a large I-buffer. When reducing the I-buffer from 32 to 16 entries, we can observe the same effect but less

Benchmark	go	li	gcc	compress	m88ksim	vortex	ijpeg	perl
Input	bigtest.in	*.lsp	insn-recog.i	50000 e 2231	ctl.raw, dcrand.lit	vortex.raw	pengin.ppm	primes.pl

Table 2: Benchmarks and their inputs

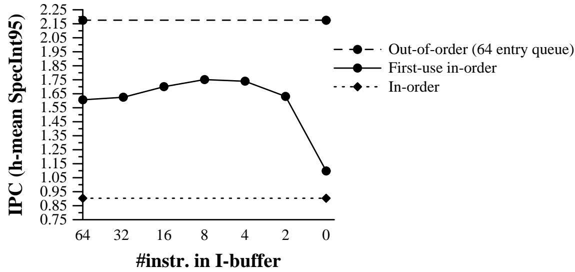

Figure 6: Performance of the First-use scheme for different in-order queue sizes of an in-order I-buffer.

frequently. What happens in these benchmarks is that some of these older instructions happen to be more critical than the younger ones that are executed earlier with a 16-entry I-buffer but not with a 32-entry one. Thus, the 16-entry I-buffer performs slightly better since critical instructions are executed earlier. When the I-buffer size is reduced further, the overall effect is negative since the ILP that can be exploited is significantly decreased.

Overall, the First-use scheme with an 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 8 entries (8 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 scheme (20%) but it performs much better than the in-order scheme (94%). 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 First-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.

4.3. Distance Scheme

Figure 7 shows the IPC of a basic implementation of the Distance scheme in comparison with an in-order issue approach and an out-of-order issue mechanism. This basic

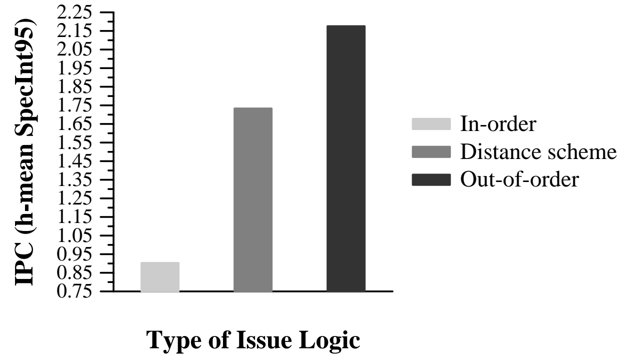

Figure 7: Performance of the basic Distance scheme (without Wait queue).

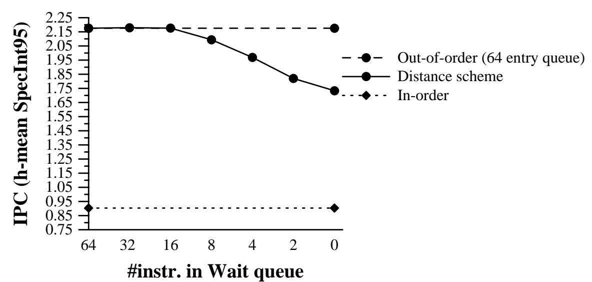

Figure 8: Performance of the Distance scheme for different Wait queue sizes.

implementation does not require any associative search since it assumes a zero-entry Wait queue (see Figure 3). We can see that the new mechanism performs better than an in-order machine (91% speed-up) and not very far from an out-of-order machine (21% slow-down).

When a full implementation of the Distance scheme is considered, the IPC is significantly improved. Figure 8 shows the evolution of the IPC when the size of the Wait queue varies. We can see that from 64 to 8 elements in the associative part of the mechanism (Wait queue) the scheme performs almost at the same level (~4% slow-down) as the out-of-order approach.

We have empirically determined that the maximum depth that the Issue queue of the Distance scheme requires in order not to cause any stall is 4 entries of 4 instructions each and, on average, it is around 2 entries for the SpecInt95 benchmarks. These numbers may be different for floating point benchmarks due to the larger latency of FP operations.

4.4. First-Use vs. Distance

Figure 9 shows the performance of both the First-use and the Distance schemes when varying the size of the associative hardware. We can see that both schemes perform at the same level while the associative part has 4

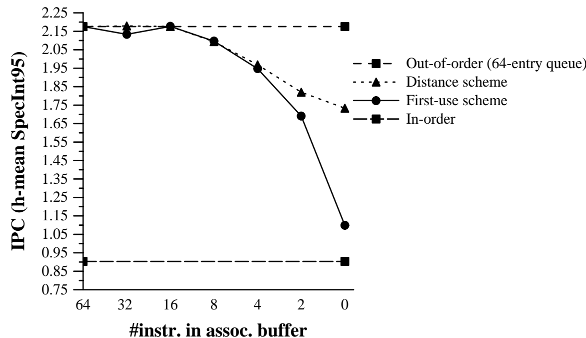

Figure 9: Performance of the two proposed schemes.

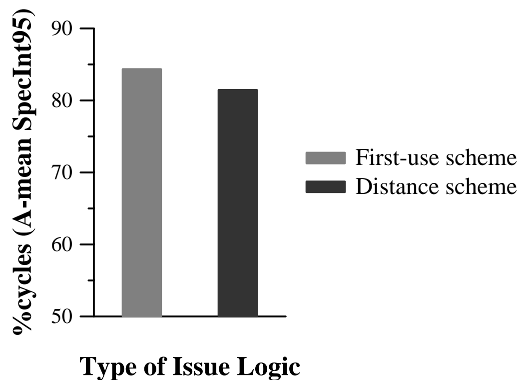

Figure 10: Percentage of cycles that the associative part of each mechanism was full (max. 8 entries).

entries or more. When the associative part is very small, the Distance scheme outperforms the First-use scheme. However, note that the complexity of the First-use scheme with an in-order I-buffer is almost independent of the I-buffer size since it does not require any associative search. In this case, the First-use scheme with an 8-entry in-order I-buffer (see Figure 6) outperforms the performance of the First-use with a very small out-of-order I-buffer and it has about the same performance as the Distance scheme with very few entries in its associative part.

Thus, we can conclude that if we can afford a small associative buffer (around 8 entries), both the First-use and the Distance schemes are very competitive in terms of IPC when compared to a fully out-of-order mechanism and both have the potential advantage of reducing the cycle time. If we want to completely avoid any associative search, both the Distance scheme without the Wait queue and the Fist-Use scheme with an 8-entry in-order I-buffer have about the same IPC, which is significantly higher than that of an in-order scheme (~90%) and not far from the performance of an out-of-order mechanism (-20%).

Figure 10 shows the percentage of cycles that the associative part of the mechanism (I-buffer in the First-use scheme and Wait queue in the Distance one) was full. These data show how likely the associative look-up may stall the instruction dispatch. The First-use scheme tends to fill the associative part a little bit more than the Distance scheme. The difference is due to the fact that the First-Use scheme keeps the instructions longer in the associative part than the Distance scheme since the instructions do not leave the I-buffer until they issue. On the other hand, the Distance scheme keeps the instructions in the associative part till the availability time of its operands is known. After the availability time is known, the instructions are kept in the Instruction queue until they are issued. Note that the availability time is usually known before the operands are ready.

5. Related Work

S. Weiss and J.E. Smith [19] designed an issue logic similar to the basic First-Use mechanism 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 [12] proposed 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. This mechanism reduces the number of instructions that have to be checked each cycle for issue.

S. Öner and R. Gupta [11] presented a proposal which tried to chain the instructions among them according to the dependences between them. This scheme, tries to build the data dependence datagraph 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, which simplifies the issue logic.

Another approach are VLIW architectures [6]. In this case the scheduling is done at compile time. This is the approach that reduces most the issue logic 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).

Other architectural approaches to the problem 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, 5, 8, 12] partitions the datapath whereas in the multithreading approach [17, 20], 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

One of the main concerns in today's processor design is the issue logic. The best performance is achieved by an out-of-order mechanism where instructions can issue independently of the program order. This scheme yields the best performance but at the same time introduces hardware restrictions which might be prohibitive for wide issue processors with large instruction windows. In this work, two new issue schemes that try to achieve the same performance with a lower hardware cost have been presented. The basic version of these schemes can achieve and speed-up of about 90% over an in-order machine but they perform 20% slower than the out-of-order scheme in terms of IPC. Besides, these schemes get rid of the associative look-up, and thus they could benefit from a shorter cycle time. The two proposed schemes can be extended with a very small associative buffer. In this case, we have measured that with an 8-entry associative buffer they can achieve almost the same IPC as an out-of-order scheme with a 64-entry instruction queue. Furthermore, one of the schemes can overlap the issue time computation with the decode/renaming stage of the processor, which may shorten the execution pipeline by one stage and consequently, reduce the penalty of branch mispredictions.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract CYCIT TIC98-0511-C02-01 and by the European Union through the ESPIRIT program under the MHAOTEU (EP26942) project. The research conducted in this paper has been developed using the resources of the CEPBA. The authors would like to thank Jim Smith for his comments on this work. Ramon Canal would like to thank his fellow PBC's for their patience and precious help.

References

[1] M.T. Bohr, "Interconnect Scaling - The Real Limiter to High Performance VLSI", in *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", in *Proc. of the Int. Symp. on High-Performance Computer Architecture,* pp. 133-142, 2000.

[4] J.Ll. Cruz, A. González, M. Valero, N. Topham, "Multiple-Banked Register File Architectures" in *Proc. of the 27nd Int. Symp. on Computer Architecture*, 2000.

[5] K.I. Farkas, P. Chow, N.P. Jouppi, Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning", in *Proc of the 30th. Ann. Symp. on Microarchitecture*, pp. 149-159, 1997.

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

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

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

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

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

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

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

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

[14] Semiconductor Industry Association, "The National Technology Roadmap for Semiconductors", 1997.

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

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

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

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

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

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