On Improving a Pipelined Scheduling Logic

Ruben Gran, Enric Morancho, Angel Olive and J.M. Llaberia.
Department of Computer Architecture. Polithecnic University of Catalonia
{rgran, enricm, angel, llaberia}@ac.upc.edu

Abstract

Pipelining the scheduling logic, which exposes and exploits the instruction level parallelism, degrades processor performance. Our evaluations show that pipelining the scheduling logic over two cycles degrades performance a 14% in SPEC-2000 integer benchmarks. Such a performance degradation is due to sacrificing the ability to execute dependent instructions in consecutive cycles.

In this work we introduce a non-speculative mechanism named Dependence Level Scheduler (DLS) which tolerates the scheduling-logic latency. In DLS, the selection of a group of one-cycle instructions (producer level) is overlapped with the wake up in advance of its group of dependent instructions. DLS is not speculative because the group of woken in advance instructions will compete for selection only after issuing all producer-level instructions.

Moreover, we compare it with a speculative mechanism. In SPEC-2000 integer benchmarks, DLS performs within 4.0% of an ideal scheduler (unpipelined) and, on average, outperforms the speculative mechanism.

1. Introduction

An option to improve processor performance is enlarging the issue queue (or scheduling window). The issue queue is in charge of exposing and exploiting instruction level parallelism (ILP). Instructions wait in the issue queue until their source operands are ready (wakeup) and appropriate execution units are available (selection).

Both issue-queue phases (wakeup and select) constitute a hardware loop, the scheduling loop, because an instruction must be selected before waking its dependents instructions up. This hardware loop is critical because its latency must be only one cycle in order to execute dependent instructions in consecutive cycles (back-to-back).

Issue-queue timing directly depends on issue-queue size. Therefore, though increasing issue-queue size could improve IPC, issue-queue size is determined by processor's clock frequency. Pipelining the scheduling logic is an option to mitigate this timing restriction. However, this option may degrade IPC because back-to-back execution of dependent instructions is impossible when the execution latency of a producer instruction is shorter than the scheduling-loop latency. Our experimental results with SPEC-2000 integer benchmarks in a 4-issue processor show that pipelining the scheduling logic over two cycles degrades IPC, on average, about 14% with respect to an unpipelined scheduling logic. Our results are similar to those reported by other authors ([3], [11], [18], [23]).

In order not to sacrifice the ability to execute back-to-back dependent instructions, several works propose to overlap the selection phase of a group of one-cycle instructions (from now on, producer level) with the wakeup phase of its dependent instructions (consumer level). While some of these proposals allow the consumer level to compete for selection speculatively [23] and to issue speculatively [3], other proposal does not relay on speculation [9]. Also, proposals [9] and [23] require two logical wakeup matrices to wakeup dependent instructions, however proposal [3] requires only one Wakeup Matrix.

In this paper we propose the Dependence Level Scheduler (DLS), a non-speculative mechanism that allows pipelining the critical hardware scheduling loop without sacrificing the ability to execute dependent instructions in consecutive cycles. In DLS mechanism, the selection phase of producer-level instructions is overlapped with the wakeup phase of the group of their dependent instructions. The group of instructions woken up in advance will not compete for selection until all producer-level instructions have been selected for execution. As DLS mechanism is not speculative, both false selections [23] and pileup victims [3] are avoided. Moreover DLS mechanism requires only a Wakeup Matrix as [3] in contrast with [9] and [23] which require two.

The performance of base DLS mechanism in SPEC-2000 integer benchmarks is within a 4.0% of an ideal unpipelined scheduler. Also we compare DLS mechanism with Select Free [3], which hardware costs are similar, and we show that DLS outperforms, on average, Select-Free.

This paper is structured as follows: Section 2 outlines the processor model being used and motivates the work. Section 3 describes the DLS mechanism. Section 4 details the simulation environment. Section 5 evaluates the base DLS mechanism. Section 6 discusses related work and Section 7 concludes this paper.

2. Baseline processor model

Figure 1 shows the pipeline of a dynamically scheduled processor. Each stage may take more than one cycle.

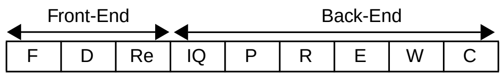

Figure 1 Processor Pipeline. F: Fetch, D: Decode, Re: Rename, IQ: Issue Queue, P: Read Payload, R: Read Register File, E: Execution; WR: Write Register File, C: Commit.

In the front-end stages of the pipeline (fetch, decode and rename stages), instructions are brought from the instruction cache, decoded and renamed (to remove false register dependencies). After that, the instructions are dispatched into the issue queue. Each instruction must wait there for the availability of its source operands (wakeup phase). Once its required execution resource is available, the instruction can be selected for execution (selection phase). Then, its payload and its source registers are read. Next, the instruction is executed and its result is written into the register file. Finally, the instruction waits until it is committed in program order.

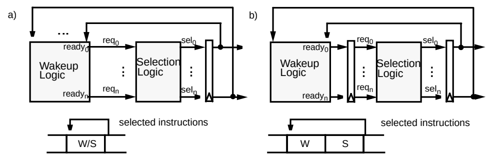

Figure 2 Diagrams of scheduling loops. a) one-cycle latency, b) two-cycle latency. (W: Wakeup, S: Selection)

In the scheduling logic there are two phases: wakeup and selection. The wakeup phase identifies instructions with available source operands, which are called ready instructions. To wakeup instructions, the Wakeup Logic uses a wire-OR style array [3][8]. Each issue-queue entry corresponding to a ready instruction activates a request signal in order to notify its readiness. The selection phase picks the oldest ready instruction taking into account available resources in each issue port. These two phases constitute a hardware loop because each instruction must be selected before waking its dependent instructions up. The latency of this hardware loop must be one cycle, otherwise back-to-back execution of dependent instructions is sacrificed. That is, instructions selected by the Selection Logic wake their dependent instructions up in the next clock cycle (Figure 2.a). Figure 2.b shows a scheduling logic pipelined over two stages. That is, a two-cycle latency scheduling-loop.

Figure 3 shows an example of the influence of the scheduling-loop latency on dependent-instruction scheduling. We consider two instructions, I1 and I2; the instruction I2 is dependent on the instruction I1, which execution latency is one cycle.

a)	I1	W / S	P	R	E	W			
	I2		W / S	P	R	E	W		
b)	I1	W	S	P	R	E	W		
	I2			W	S	P	R	E	W

Figure 3 Scheduling of an one-cycle instruction and a dependent instruction assuming several scheduling-loop latencies: a) one cycle, b) two cycle. (W: Wakeup, S: Select).

In Figure 3-a, as the scheduling loop is unpipelined, the instruction I2 can be issued one cycle after issuing the instruction I1. In Figure 3-b, the scheduling loop is pipelined over two cycles (one cycle for each phase). In this scenario, when the instruction I1 is selected, it wakes the instruction I2 up on next cycle. Consequently, back-to-back execution of instructions I1 and I2 is not possible.

As a general rule, back-to-back execution is possible only if the execution latency of the producer instruction is greater than or equal to the scheduling-loop latency. Previous works [3], [11], [18] and [23] concluded that back-to-back execution is a must for high-performance processors.

Table 1 shows the distribution of committed instructions in the SPEC-2000 benchmarks considering their execution latency and if the instructions update the register file (Section 4 details benchmarks, simulated intervals and the execution latency of the instructions). We observe that integer benchmarks double the amount of one-cycle instructions with respect to floating-point benchmarks; consequently, integer benchmarks will be more sensitive to the scheduling-loop latency than floating-point benchmarks.

In this paper, the baseline processor uses a two-cycle latency scheduling loop. Then there is, at least, a two-cycle delay between issuing an instruction and issuing its dependent instructions. In order not to degrade performance with respect to the unpipelined scheduling logic, the pipelined scheduling logic must be able to exploit ILP in the issue cycle between issuing an one-cycle instruction and issuing its dependent instruction. For multi-cycle producer instructions (greater than one-cycle latency), pipelining the scheduling logic does not degrade performance with respect to an unpipelined scheduling logic.

Table 1 Distribution of committed instructions according to the execution latency in SPEC-2000 benchmarks.

	update the register file		do not update the register file
	execution latency		
	one-cycle	multi-cycle	
Integer benchmarks	44.3%	32.0%	23.7%
Floating Point benchmarks	23.6%	62.4%	14.0%

3. Dependence-Level Scheduler

In this section, we describe the Dependence-Level Scheduler (DLS), a non-speculative mechanism that allows pipelining over two cycles the critical scheduling loop without sacrificing the ability to execute back-to-back one-cycle instructions and their dependent instructions.

The idea is overlapping the selection phase of producer-level instructions with the wakeup phase of their dependent instructions. That is, dependent instructions are woken up before their producer instructions have been selected for issuing. Moreover, in order to avoid an speculative selection phase, woken up in advance instructions will not compete for selection until all producer-level instructions have been issued. The goal of DLS mechanism is to look for opportunities for back-to-back execution of dependent instructions. Note that the execution latency of multi-cycle instructions hides the scheduling-loop latency and therefore back-to-back execution with their dependents instructions is possible.

Figure 4 shows an example of the scheduling of an instruction sequence, in which we assume that only one instruction can be issued per cycle. The IQ label means that the instruction is waiting to be ready in the issue queue. The W and WA labels mean, respectively, that the instruction wakes up and that the instruction wakes up in advance. The RI label symbolizes that the instruction is ready and it is competing for selection. The ARI (advanced ready instruction) label means that the instruction has been waken up in advance but does not compete yet for selection. Finally, the S label means that the instruction is selected for execution.

In Figure 4, the source operands of instructions 1 and 2 are available in cycle 1. Both instructions wake up and become the current producer level. In cycle 2, the current producer level competes for selection, and also it wakes up in advance its dependents. Consequently, instruction 3 is woken up in advance in cycle 2, and it becomes the consumer level. This consumer level will not compete for selection until the current producer level is completely scheduled. At the end of cycle 3, the producer level has been completely scheduled. Then in cycle 4, the consumer level (instruction 3) will be allowed to compete for selection, and consequently, it becomes the current producer level. Also in cycle 4, the current producer level wakes its consumer level (instruction 4) up in advance. Because in cycle 4 the current producer level is completely scheduled, in cycle 5 the current consumer level will be allowed to compete for selection.

Cycles	1	2	3 ▌	4 ▌	5
1. add r1←r2, r3	W	S			
2. add r4←r5, r6	W	RI	S		
3. sub r9←r1, r7	IQ	WA	ARI	S	
4. sub r10←r9, r8	IQ	IQ	IQ	WA	S

Figure 4 Scheduling example of the DLS mechanism. A bar between cycles indicates that all producer-level instructions have been issued, then consumer level can compete for selection next cycle.

DLS mechanism is equivalent to an one-cycle scheduling-loop if, every cycle, all producer-level instructions are scheduled. In this scenario, back-to-back execution is performed and oldest-first selection policy is observed. Otherwise, if producer-level instructions require several cycles to be scheduled and their woken up in advance consumer instructions are prevented from competing for selection, then oldest-first selection policy is not always observed and DLS performance depends on whether the available ILP can maintain the throughput of issued instructions. A harmful performance scenario takes place when issue width is not fully exploited and there are woken up in advance instructions prevented from competing for selection whose producer-level instructions have been scheduled in previous cycles.

3.1. Hardware Design

Next, we describe the implementation of DLS by extending the base two-cycle scheduling logic

(Section 2). Main differences of DLS with respect to a Base two-cycle scheduling logic are:

- In the Base model, each instruction wakes its dependent instructions up only after being selected. In DLS, one-cycle instructions wake their dependent instructions up before being selected. They wake their dependent instructions up in advance using the one-cycle scheduling loop shown in Figure 5.
- In the Base model, instructions start competing for selection the cycle after becoming ready. In DLS, the selection phase of ready instructions dependent on producer-level instructions may be delayed. This is necessary to prevent them from being selected speculatively. In Figure 5, D-Logic and ZDL are in charge of this task.

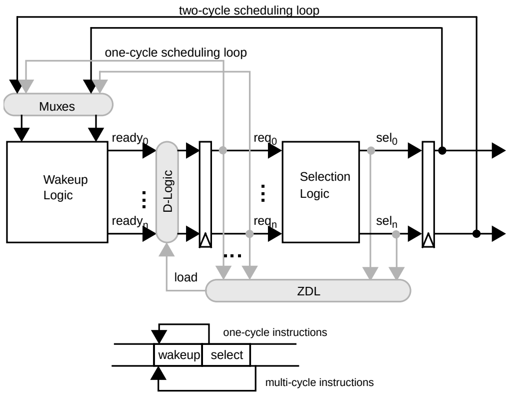

Figure 5 Base DLS design.

In DLS instructions are classified according to two criteria: attending to their own execution latency and attending to their producers' execution latency.

In decode phase, instructions are classified attending to their execution latency. One-cycle instructions are classified as *wakeup in advance*; multi-cycle instructions are classified as *wakeup in selection*. In Figure 5, wakeup signals to Wakeup Logic come from one-cycle and two-cycle latency scheduling loops. Only one loop is significant for each instruction to wake its dependent instructions up. The wakeup signal is selected by a multiplexer controlled by the classification of the instruction.

In rename phase, producer instructions of every instruction are identified. DLS also classifies instructions according to the latency of their producer instructions. Instructions that depend on at least an one-cycle instruction are classified as *woken up in advance*. Instructions that do not depend on any one-cycle instruction are classified as *woken up in selection*. This later class also contains instructions whose source operands are available in rename phase.

In Figure 5, the Wakeup Logic sets ready bits ($ready_0$ ... $ready_n$) for those instructions that have been waken up. Ready *woken up in selection* instructions will compete for selection next cycle after waking up. However, ready *woken up in advance* instructions must be prevented from competing for selection until selecting all one-cycle instructions currently competing for selection (req_0 ... req_n). Zero Detection Logic (ZDL) and D-Logic determine if ready *woken up in advance* instructions can compete for selection next cycle. Every cycle, ZDL determines if all requesting *wakeup in advance* instructions have been selected. If this condition is met, then load input in D-Logic is set. Consequently, D-Logic lets *woken up in advance* instructions compete for selection next cycle; otherwise, D-Logic will retain *woken up in advance* instructions at least one more cycle (Figure 6).

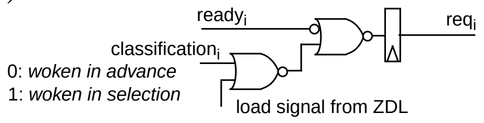

Figure 6 D-Logic. Slice corresponding to one issue-queue entry. "ready" stands for a request signal of Wakeup Matrix.

Figure 7 shows an slice of ZDL. Non continuous box contains the logic which is replicated for every issue-queue entry. For every requesting instruction in the issue queue, ZDL checks if they have been selected the current cycle. Moreover, either instructions selected in previous cycles or classified as *woken in selection* are not accounted by ZDL.

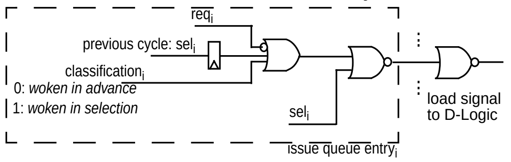

Figure 7 ZDL. Slice corresponding to one issue-queue entry

4. Simulation Environment

4.1. Processor model

We have modified SimpleScalar 3.0d [1] to model a Re-Order Buffer and separate issue queues (IQ). We assume an out-of-order processor with eight stages from Fetch to IQ and two stages between IQ and Execution. Table 2 details other processor and memory parameters.

Our processor splits store instructions into two instructions: STA (store address computation) and STD (store data). Two issue-queue entries are allocated to each store instruction.

Table 2 Processor and memory parameters

Processor model		Memory hierarchy	
Fetch and Decode width	4	*L1 I-cache L1 D-cache*	32KB, 4-way, 2 cycles
Branch predictor: hybrid (bimodal, gshare)	16 bits	Line size	32 B
ROB size	128	*L2 Unified Cache*	256 KB, 4-way, 12 cycles
LSQ size	64		
Issue-queue size: Integer / Floating point	32 / 20	Line size	32 B
Functional Units: Integer / Floating point	4 / 2	L2-Main memory bus	8bytes / 2 cycles
Memory access ports	2	Main memory latency	100 cycles
Issue width: Integer / Floating point	4/2	Load latency prediction	Blind (always L1 hit)

Table 3 lists the instruction latencies assumed in this work. ALU label stands for: integer add, integer subtract, logical operations and branches.

Table 3 Execution latency (in cycles) of the instructions.

	Latency		Latency
ALU	1	FP (+,*)	4 fully pipelined
Load	3	FP (/)	15 unpipelined
Integer (*,/)	10 /15 unpipelined	FP (sqrt)	24 unpipelined

A load instruction can be issued only after issuing all the STA instructions corresponding to the store instructions older than the load instruction. Consequently, the processor makes each load instruction dependent on all its older STA instructions.

Load instructions are variable-latency instructions. To cope with this variability, load instructions are blindly predicted to hit L1 and their dependent instructions are scheduled accordingly. To deal with misschedulings, our processor implements a *delayed selective replay* mechanism that replays the misscheduled instructions from the Issue Queue [13]. A register scoreboard keeps the status of each register (dependent or independent on a misscheduling). Each load instruction that misses L1 marks its destination register as unavailable. Each issued instruction accesses, in register-read stage, the scoreboard to check the status of their source registers and propagates the status to its destination register. To recover from misschedulings, each instruction remains in the issue queue until verifying that it is not dependent on misschedulings. Instructions dependent on misschedulings are replayed from the issue queue after resolving the cache miss.

4.2. Workload

We use SPECInt-2000 integer benchmarks compiled into Alpha ISA. We simulate a contiguous run of 100M-instruction from SimPoints [22] after a warming-up of 100M-instruction. Table 4 shows input data sets.

Table 4 Simulated benchmarks and their input data set.

Bench	Data set	Bench	Data set	Bench	Data set
bzip2	program-ref	gzip	program-ref	twolf	ref
crafty	ref	mcf	ref	vortex	one-ref
eon	rushmeier-ref	parser	ref	vpr	route-ref
gcc	166-ref	perl	diffmail-ref		

5. Evaluation of DLS

In this section we evaluate the performance of DLS. For comparison purposes three more models are evaluated.

First, a baseline model (B) with a two-cycle scheduling loop that sacrifices the execution of dependent instructions in consecutive cycles if producer instructions are one-cycle instructions.

Second, an ideal model (ID) with an one-cycle scheduling loop. Dependent instructions can be scheduled back-to-back. However, in order to remove the effect of a branch-misprediction penalty shorter than in the other models, its pipeline depth is kept consistent with them by adding to the ID model one extra stage in the processor-pipeline front-end.

The third model has been proposed by M. D. Brown et al. in [3] and it is named Select-Free (SF). They propose to remove the Selection Logic from the critical scheduling loop. Ready instructions wake their dependent instructions up, that is, the selection phase of each producer instruction is overlapped with the wakeup phase of its consumer instructions. Thus, all woken up instructions compete speculatively for selection. Contention for issue ports may produce misspeculations because a consumer instruction may be selected at the same time as its producer instructions. SF checks the availability of the source operands of each issued instruction before execution stage. In our simulations we model the SF mechanism on a two-cycle scheduling loop; SF checks the availability of source operands in register-read stage using the scoreboard structure previously described to check latency misspeculations.

Table 5 shows, for each benchmark, the IPC achieved by B model in a 4-way processor (Table 2). Figure 8 shows the speedup of the other models with respect to the B model. We present individual results for each SPEC-2000 integer benchmark and two harmonic average values: HM (including all benchmarks) and HM-*mcf* (including all benchmarks but *mcf* due to its biased memory behaviour).

Table 5 IPC of the baseline model (B).

bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
1.36	1.92	2.15	1.84	1.44	1.68	0.13	1.05	1.47	0.91	2.22	0.78

Figure 8 shows the speedup of ID, DLS and SF with respect to B model. The ID model outperforms B model from 24.8% in *parser* to 6.3 in *mcf*, and on average the ID model outperforms the B model around 10% (HM). However, excluding *mcf* from the average, then the improvement reaches a 14%. These results remark the importance of back-to-back execution of dependent instructions.

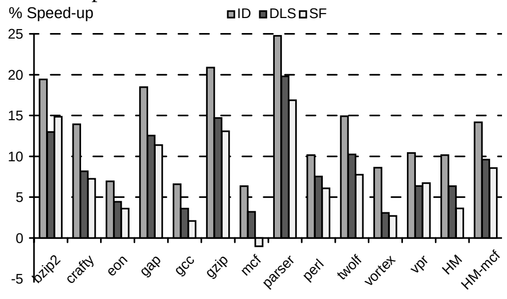

Figure 8 Speedup of ID, DLS and SF models with respect to B model.

On average, DLS model performs within a 3.6% (HM) and 4.0% (HM-mcf) of ID model. Performance degradation with respect to ID model is due to conservatively preventing woken in advance instructions from competing for selection despite their producer instructions have already been selected. This is caused by producer-level instructions whose scheduling takes more than one cycle.

Select Free performs within the 4.6% (HM) of ID model. Performance degradation with respect to ID model is caused by the re-scheduled instructions due to misspeculated selections. Table 6 shows the percentage of issued instructions which are re-scheduled in SF due to misspeculated selections. Note that SF will degrade performance only if another ready instruction with available source operands can be executed instead of the misspeculated one.

DLS outperforms, on average, SF. Only in *bzip2* and *vpr*, SF outperforms DLS (at most 2% in *bzip2)*. By comparing the number of issued instructions, SF counts, on average, a 3.9% more than DLS.

Table 6 Percentage of issued instructions rescheduled by SF.

bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
0.9	2.1	0.4	2.2	0.7	3.4	3.3	4.3	0.5	1.6	1.2	0.3

Figure 9 shows the performance impact of varying the latency between selection stage and execution stage from one cycle to three cycles, since two cycles is the default value in our evaluations. All models show a performance degradation if the latency increases, because both the branch-misprediction penalty and the load-instruction shadow [13] increase. However, the performance degradation observed in SF model is greater than in the other models because the misspeculated-selection penalty increases with the latency.

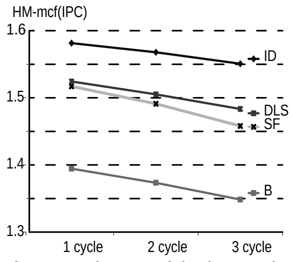

Figure 9 Performance impact of the latency between selection and execution stages in ID, DLS, B and SF models.

6. Related works

In order to reduce the scheduling latency, Palacharla et al. [18] proposed dispatching chains of dependent instructions into FIFO queues; the only instructions considered to be issued are the head instruction of each queue. Other works preschedule the instructions taking advantage of the fact that most instruction latencies are known at decode time ([4], [5], [7] [16] [20]). At dispatch time, instructions are sorted into a buffer according to their predicted issue cycle. The schemes mainly differ in the mechanism that deals with variable-latency instructions, e.g. load instructions, and their chains of dependent instructions; a structure like an issue queue is used for these cases. The structure is placed before the buffer in [4] and it tracks the availability of source data. In other works the structure is placed after the buffer in order to schedule all instructions [16] or just the mispredicted instructions and their dependent chains of instructions [5]. Ernst et al. [7] proposed organizing the buffer as a countdown cyclic queue. The instructions are scheduled from this buffer in order and, a latency-mispredicted instruction requires re-inserting into the queue its chain of dependent instructions. All these techniques require estimating the issue cycle of the instructions before inserting them in the buffer structure. This is a challenging task for wide-issue processors, although it can be approximated. Moreover, most prescheduling works also use an issue queue to deal with variable-latency instructions and their dependent instructions.

Some proposals exploit the fact that most register-writing instructions have, at most, one dependent instruction currently in the issue queue. Based on this observation, the proposed designs have structures that keep track of one or several instructions that consume a produced register value. So, in order to wake instructions up, associative searches are replaced by indexing. S. Weiss and J. Smith proposed, in a single-issue processor, a scheme named Direct Tag Search where each instruction has a pointer to its dependent instruction in order to waking it up directly [24]. The basic scheme can not implement direct wakeup in case of instructions with several consumers. Several alternatives have been proposed to overcome this restriction: linking the consumer instructions [17], using an associative structure to keep track of the consumers [5], using an issue queue with few entries [4][5], or using a register scoreboard to check the availability of the register values [21]. These techniques require additional hardware support for branch-misprediction recovery unless the recovery is initiated only when the branch instruction becomes the oldest instruction in flight.

Goshima et al. used two RAM bitmap arrays, one for each source operand, to identify all the successors of each instruction in the issue queue [10]. That is, this mechanism supports indexing-based wakeup for all instructions regardless of their number of dependent instructions. A new design that reduces the area cost for large issue queues was proposed by K. Hsiao and C. Chen in [12].

The observation that many instructions already have one or two ready source operands at dispatch time has been used to reduce the load capacitance of the wakeup tag bus in schedulers that use CAM schemes to wakeup; consequently, the wakeup latency may be reduced. D. Ernst and T. Austin used specialized issue queues and prediction of the last-arriving input operand [6]; consequently, it is a speculative mechanism. I. Kim and M. Lipasti proposed a sequential wakeup mechanism; while the predicted last-arriving operand is placed into a fast-wakeup entry, the other operand is placed into a slow-wakeup entry. Each selected instruction notifies the availability of its result to the fast-wakeup entries and, on next cycle, to the slow-wakeup entries [14]. The observation that the distance between dependent instructions is generally short is used in [10] to narrow the implementation of the issue queue with dependence matrices that supports indexing-based wakeup.

Brekelbaum et al. proposed a hierarchical scheduling window [2]. While critical instructions are placed in a small issue queue, latency-tolerant instructions are placed in a buffer that does not use power-hungry CAM logic, but it requires a longer latency for the wakeup operation. Hrishikesh et al. in [11] proposed a segmented issue queue to reduce scheduling latency; issue-queue entries are compacted after extracting the instructions. The instruction wakeup is pipelined among the segments and each segment has a different scheduling priority. In this mechanism, the scheduling loop must still be evaluated in a single cycle for a subset of instructions in the issue queue or instructions in a particular partition/segment of the window.

Several works overlap the selection phase of a producer instruction with the wakeup phase of their dependent instructions. J. Stark et al. [23] proposed to speculatively wake instructions up by their grand-parents in order to tolerate the scheduling logic latency. By using an additional Wakeup Logic driven by parent instructions, it is confirmed if each selected instruction has their source operands available. R. Gran et al. [9] proposed to add another Wakeup Logic to a base two-cycle scheduling logic. This additional Wakeup Logic is also driven by parent instructions, the difference is that they are competing for selection. Instructions that depends on one-cycle instructions are allocated in the additional Wakeup Logic. Some instructions included in additional Wakeup Logic will be woken up in Wakeup Logic of the base scheduling logic and they will compete for selection when this occurs. Other instructions will be woken up in advance by additional Wakeup Logic and they will be allowed to compete for selection when the group of parent instructions has been selected, thus overlapping the selection of parents instruction with wakeup of their dependent instructions. The former proposal [23] is speculative. Both proposals use two Wakeup Logic which duplicate the cost.

M.D. Brown et al. [3] proposed a design of the scheduling loop, named Select-Free in which Wakeup Logic forms a single cycle loop and the Selection Logic is removed from the critical scheduling loop. Thus, instructions, that compete for selection, speculatively wake their dependent instructions up, these instructions also speculatively compete for selection and they speculatively wakeup their dependent instructions. The mechanism wastes energy due to re-schedulings. However, DLS mechanism is not speculative and it outperforms Select-Free.

7. Conclusions

In high performance processors the scheduling loop is a critical loop. Pipelining this loop without significantly downgrading performance may allow to increase clock frequency and/or to enlarge the issue queue. We have proposed a mechanism (DLS) that tolerates the latency of a pipelined scheduling loop and it also boosts performance with respect to a pipelined scheduling logic. The idea of DLS is to look for opportunities to execute dependent instructions in consecutive cycles by overlapping selection phase of producer instructions with wakeup of dependent instructions. Key differences of DLS with respect to previous proposals that tolerate the latency in a pipelined scheduling logic is that DLS is non-speculative and it does not duplicate Wakeup Logic hardware cost.

Our evaluations have shown that DLS performs, on average, within a 4.0% (HM-mcf) of an ideal (unpipelined) scheduler. Also, on average, it outperforms SF proposal, which is speculative, and this difference increases with the number of stages between issue and execution due to SF speculation.

Acknowledgements

This work has been supported by the Ministry of Education and Science of Spain under the contract TIN-2004-07739-C02-01, the grant AP2003-4621 and the HiPEAC European Network of Excellence.

References

[1] D.C. Burger and T.M. Austin, “The SimpleScalar Tool Set, Version 2.0,” UW Madison Computer Science T. R. #1342, June 1997.

[2] E. Brekelbaum et al. Hierarchical Scheduling Windows. Micro-2002, p. 27-36.

[3] M. Brown et al. Select-Free Instruction Scheduling Logic. Micro-2001, p. 204-213.

[4] R. Canal and A. González. A Low-Complexity Issue Logic. ICS-2000, p. 327-335.

[5] R. Canal and A. González. Reducing the Complexity of the Issue Logic. ICS-2001, p. 312-320.

[6] D. Ernst and T.M. Austin. Efficient Dynamic Scheduling through Tag Elimination. ISCA-2002, p. 37-46.

[7] D. Ernst et al. Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay. ISCA-2003, p. 253-262.

[8] J.A. Farrel and T.C Fischer. Issue Logic for a 600 Mhz Out-of-order Execution Microprocessor. IEEE Journal of Solid-State Circuits, Vol 33(5), pp 707-712, 1998.

[9] R. Gran et al. An Enhancement for a Scheduling Logic Pipelined over Two-Cycles. ICCD-2006. p. 203-209.

[10] M. Goshima et al. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. Micro-2001, p. 225-236.

[11] M. Hrishikesh et al. The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays. ISCA-2002, p. 14-24.

[12] K. S. Hsiao and C.H. Chen. An Efficient Wakeup Design for Energy Reduction in High-Performance Superscalar Processors. Conf. on Computing frontiers, 2005, p. 353-360.

[13] I. Kim and M.H. Lipasti. Understanding scheduling replay schemes. HPCA-2004, p. 138-147.

[14] I. Kim and M.H. Lipasti. Half-Price Architecture. ISCA-2003, p. 28-38.

[15] J. Leenstra, et al.. A 1.8-GHz instruction window buffer for an out-of-order microprocessor core. IEEE Journal of Solid-State Circuits, Vol. 36, 11, Nov. 2001 p. 1628 - 1635

[16] P. Michaud and A. Seznec. Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. HPCA-2001, p. 27-36.

[17] S. Önder and R. Gupta. Superscalar Execution With Dynamic Data Forwarding. PACT-1998, p. 130-135.

[18] S.Palacharla et al. Quantifying the Complexity of Superscalar Processors. T.R. University of Wisconsin-Madison. Nov 1996.

[19] E. Perelman et al.. Picking Statistically Valid and Early Simulation Points. PACT-2003, p. 244-255.

[20] S. E. Raasch et al. A Scalable Instruction Queue Design Using Dependence Chains. ISCA-2002, p. 318-330.

[21] T. Sato et al, "Revisiting Direct Tag Search Algorithm on Superscalar Processors". WCED-2001.

[22] T. Sherwood et al., “Automatically Characterizing Large Scale Program Behaviour,” ASPLOS-2002, p. 45-57.

[23] J. Stark et al. On Pipelining Dynamic Instruction Scheduling Logic. Micro-2000, p. 57-66.

[24] S. Weiss and J.E. Smith. Instruction Issue Logic in Pipelined Supercomputers. IEEE Transactions on Computers, 33: p.1013-1022, November 1984.