On Reducing Misspeculations in a Pipelined Scheduler

R. Gran
University of Zaragoza-CPS
rgran@unizar.es

E. Morancho, A. Olivé and J.M. Llabería
Universitat Politècnica de Catalunya-DAC
{enricm,angel,llaberia}@ac.upc.edu

Abstract

Pipelining the scheduling logic, which exposes and exploits the instruction level parallelism, degrades processor performance. In a 4-issue processor, our evaluations show that pipelining the scheduling logic over two cycles degrades performance by 10% in SPEC-2000 integer benchmarks. Such a performance degradation is due to sacrificing the ability to execute dependent instructions in consecutive cycles.

Speculative selection is a previously proposed technique that boosts the performance of a processor with a pipelined scheduling logic. However, this new speculation source increases the overall number of misspeculated instructions, and this unuseful work wastes energy.

In this work we introduce a non-speculative mechanism named Dependence Level Scheduler (DLS) which not only tolerates the scheduling-logic latency but also reduces the number of misspeculated instructions with respect to a scheduler with speculative selection. 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. On average, DLS reduces the number of misspeculated instructions with respect to a speculative scheduler by 17.9%. From the IPC point of view, the speculative scheduler outperforms DLS by 0.3%. Moreover, we propose two non-speculative improvements to DLS.

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 selection) 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, this could also increase processor's cycle time. 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 10% with respect to an unpipelined scheduling logic. Our results are similar to those reported by other authors ([4], [13], [23], [31]).

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 [31] or to issue speculatively [4], other proposal does not rely on speculation [11]. Also, while proposals [11] and [31] require two logical Wakeup Matrices to wakeup dependent instructions, proposal [4] requires only one Wakeup Matrix.

Speculative mechanisms boost the performance of a pipelined scheduling logic [4][31]. However, this new speculation source increases the overall number of misspeculations. For instance, our experimental results, using the simulation environment described in Section 4, show that the speculative mechanism Select-

Work partly funded by the Diputacion General de Aragón (Grupo Consoli Investigacion), EU Network of Excellence HiPEAC-2(FP7/ICT/217068) Ministerio de Educacion y Ciencia (TIN 2007-60625).

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

Free[4] boosts the performance of a two-cycle pipelined scheduling, but it increases the overall number of misspeculated issued instructions, on average, by 10.0%. Such unuseful work wastes energy in both processor front-end and processor back-end.

In this paper, we propose the Dependence Level Scheduler (DLS), a non-speculative mechanism which not only tolerates the latency of the scheduling logic but also reduces the overall number of misspeculated issued instructions with respect to Select-Free mechanism by 17.9%. Moreover, DLS mechanism also boosts IPC of a two-cycle pipelined processor and it reduces the overall number of misspeculated issued instructions by 9.7%.

Dependence Level Scheduler 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 compete for selection after all producer-level instructions have been selected for execution. As DLS mechanism is not speculative, both false selections [31] and pileup victims [4] are avoided. Moreover, DLS mechanism, as Select-Free, requires one Wakeup Matrix, in contrast with [11] and [31] which require two logical Wakeup Matrices. As we describe in the evaluation section, we consider load-latency prediction and memory dependence prediction in contrast to [4][11] and [31]. We compare DLS mechanism with Select-Free mechanism, which hardware costs are similar, and we show that DLS mechanism is outperformed by Select-Free, on average, by 0.3%. In this paper, we also propose two improvements to DLS mechanism. The best DLS improvement performs within 1.5% of an ideal scheduler, it outperforms by 0.2% Select-Free and it reduces by 18.7% the overall number of misspeculated issued instructions with respect to 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 DLS mechanism. Section 6 describes two enhancements to the DLS mechanism and evaluates them. Section 2 discusses related work and Section 8 concludes this paper.

2. Baseline processor model

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

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.

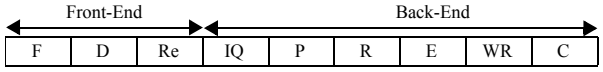

Figure 1 Processor Pipeline. F: Fetch, D: Decode, Re: Rename, IQ: Issue Queue, P: Read Payload, R: Read Register File, E: Execution; WR: Write Register File, C: Commit.

The scheduling logic is composed by two phases: wakeup and selection. The wakeup phase identifies instructions with available source operands, these instructions are named ready instructions. To wakeup instructions, the Wakeup Logic uses a wired-OR style array [4][10][28]. 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 (Fig. 2.a). Fig. 2.b shows a scheduling logic pipelined over two stages. That is, a two-cycle latency scheduling-loop.

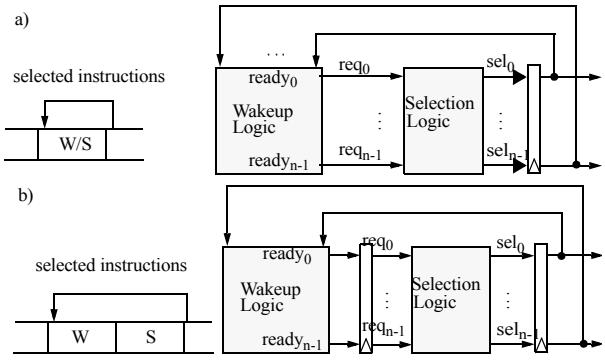

Figure 2 Diagrams of scheduling loops. a) one-cycle latency, b) two-cycle latency. (W: Wakeup, S: Selection)

Fig. 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.

In Fig. 3.a, as the scheduling loop is unpipelined, the instruction I2 can be issued one cycle after issuing the instruction I1. In Fig. 3.b, the scheduling loop is pipelined over two cycles. 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.

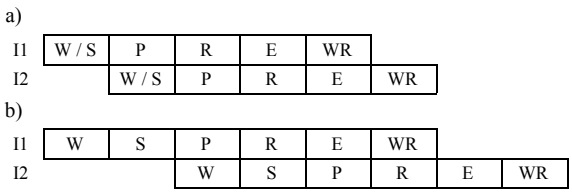

Figure 3 Scheduling of a one-cycle instruction and a dependent instruction assuming several scheduling-loop latencies: a) one cycle, b) two cycles. (W: Wakeup, S: Selection).

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 [4], [13], [23] and [31] concluded that back-to-back is a must for high-performance processors.

In SPEC-2000 benchmarks, we have counted the number of one-cycle committed instructions that update the register file. We observe that integer benchmarks double the percentage of one-cycle instructions with respect to floating-point benchmarks (44.3% vs. 23.6%); 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 a 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.

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 (woken up in advance). Moreover, in order to avoid an speculative selection phase, woken up in advance instructions will compete for selection after 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 and to use them safely. Note that the execution latency of multi-cycle instructions hides the scheduling-loop latency and therefore they can be executed back-to-back with their dependent instructions.

Fig. 4 shows an example of the scheduling of an instruction sequence, assuming 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 it does not compete yet for selection. Finally, the S label means that the instruction is selected for execution.

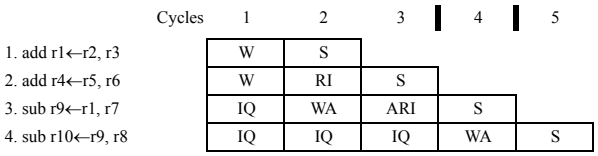

Figure 4 Scheduling example of the DLS mechanism. A bar between cycles indicates that all producer-level instructions have been issued, then consumer-level can compete for selection next cycle safely.

In Fig. 4, we assume that 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.

DLS mechanism is equivalent to a one-cycle schedul-

ing-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, their woken up in advance consumer instructions are prevented from competing for selection, then 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 already 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 scheduler 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 Fig. 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 Fig. 5, D-Logic and ZDL are in charge of this task.In DLS, instructions are classified according to two criteria: their own execution latency and their producers' execution latency.

In decode phase, instructions are classified according to their execution latency. One-cycle instructions are classified as *wakeup in advance*; multi-cycle instructions are classified as *wakeup in selection*. In Fig. 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 a 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.

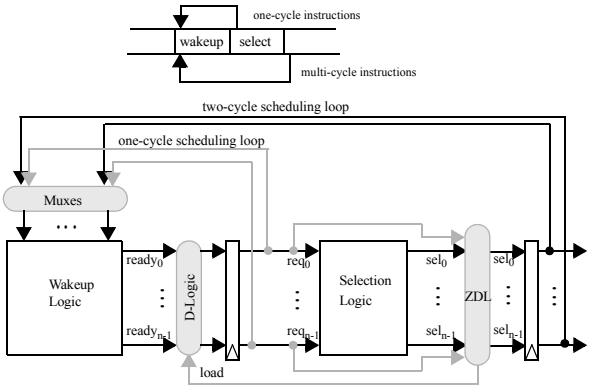

Figure 5 DLS design.

In Fig. 5, the Wakeup Logic sets ready bits ($ready_0$... $ready_{n-1}$) 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-1}). Zero Detection Logic (ZDL) and D-Logic determine if ready *woken up in advance* instructions are allowed to compete for selection next cycle.

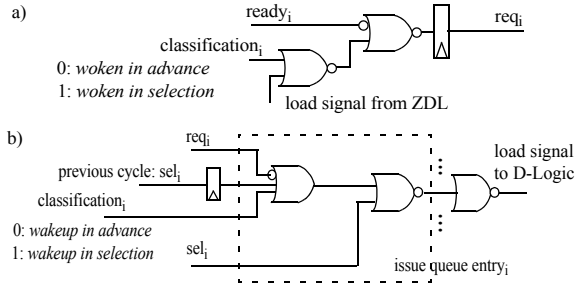

Figure 6 a) D-Logic. Slice corresponding to an issue-queue entry. "ready" stands for a request signal from Wakeup Matrix. b) ZDL. Slice corresponding to an issue-queue entry

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 allows *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 (Fig. 6.a). Fig. 6.b 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, instructions either selected in previous cycles or classified as *woken in selection* are not accounted by ZDL.

4. Simulation environment

4.1. Processor model

We have modified SimpleScalar 3.0d [1] to model a Re-Order Buffer (ROB) and separate issue queues (integer IQ and floating-point IQ). Also, our processor model predicts both load latencies and memory dependences. The speculative instruction issue and the recovery mechanisms have been carefully modelled. We assume an out-of-order processor with eight stages from Fetch to IQ and two stages between IQ and Execution. Table 1 details other processor and memory parameters.

Table 1 Processor and memory parameters

Processor model			
Fetch and Decode width	4	Issue-queue size: Integer / Floating point	32 / 20
Branch predictor: hybrid (bimodal, gshare)	16 bits	Functional Units: Integer / Floating point	4 / 2
Store sets predictor	1024 / 128	Memory access ports	2
ROB/LSQ size	128/64	Issue width: Integer / Floating point	4/2
Memory hierarchy			
L1 I-cache *L1 D-cache*	32KB, 4-way, 2 cycles	L2-Main memory bus	8bytes / 2 cycles
Line size	32 B	Main memory latency	100 cycles
L2 Unified Cache *Line size*	256 KB, 4-way, 12 cycles 32 B	Load latency prediction	Blind (L1 hit)

Table 2 lists the instruction latencies assumed in this work. ALU label stands for: integer add, integer subtract, logical operations and branches.

Table 2 Execution latency (in cycles) of the instructions.

	Latency
ALU	1
Load	3
Integer (*,/)	10 /15 unpipelined

	Latency
2-FP (+,*)	4 fully pipelined
FP (/)	15 unpipelined
FP (sqrt)	24 unpipelined

We use a memory-dependence predictor (store sets [7]) to predict which store instructions write to the same memory location that a younger load reads. Then, when a load instruction is predicted to depend on a store instruction, the scheduler delays issuing the load instruction until issuing the store instruction. A memory-order violation is handled when the store instruction commits and, at this moment, the recovery is initiated from the offending load. The replay is non-selective (squash recovery) and is performed from a buffer, located before dispatch phase, that keeps all renamed instructions in order to reduce the misspeculation penalty on a memory-order violation. A store instruction is ready to compete for selection when all input operands are available.

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 [16]. A register scoreboard keeps the status of each register (dependent or independent on a miss-cheduling). Each load instruction that misses L1 marks its destination register as unavailable. In register-read stage, each issued instruction accesses the scoreboard to check the status of their source registers and it 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. Considerations about scheduling-loop critical path

Select logic is implemented using a matrix of bits [10][28], in which the number of rows and columns is equal to the number of instructions in IQ (N). Each row codes the priority of an instruction with respect to the other valid instructions in IQ. Each column indicates if the instruction is requesting for execution. The evaluation of this bit matrix has 3 phases. First, row-request signals are propagated horizontally in parallel until they reach their transposed columns. Second, request signals are propagated vertically in parallel in order to inhibit younger requesting instructions. Third, for each row, sel_i signal is evaluated by checking its priority with all requesting columns. Calling a hop [28] to either horizontal or vertical distance between two contiguous matrix elements, circuit delay requieres 3xN hops to evaluate sel_i signal.

ZDL has three gate levels (Fig. 6.b). First two levels are replicated per IQ-entry. First gate level of ZDL is evaluated in parallel with first phase of selection logic. In the second gate level of ZDL, the NOR_i gate evaluates just after sel_i signal has been evaluated. Third gate level is a N-input NOR gate implemented as a chain of two-input NOR gates. After 2xN hops, the first sel_i signal (sel_{N-1}) is stable, and just after NOR gates of the chain evaluate while rows evalutate their sel_i signals. Therefore, evaluation time for ZDL equals to 2xN+N+1 hops, this means a similar circuit delay as a bit matrix for an N+1-entry IQ.

We can suppose that the delay increment due to ZDL is similar to enlarging the issue queue by one entry in processor models different from DLS. In the evaluation, we conservatively enlarge the IQ by two entries to all processor models not using DLS.

4.3. Workload

We use SPECInt-2000 integer benchmarks compiled into Alpha ISA. We simulate a contiguous run of 100M-instruction from SimPoints [30] after a warming-up of 100M-instruction. Table 3 shows input data sets.

Table 3 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 processor. For comparison purposes, we also evaluate three additional processors.

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

Second, an ideal processor (ID) with a two-stage scheduling logic, but it uses a one-cycle latency hardware loop to wakeup/select instructions. Thus, dependent instructions can be scheduled back-to-back in spite of producer's execution latency, and the pipeline depth is kept consistent with the other processors (like in [4]).

Third, a processor that uses the Select-Free (SF) mechanism proposed by M. D. Brown et al. in [4]. 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 misspeculate selections because a consumer instruction may be selected at the same time as its producer instructions. SF mechanism 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 mechanism checks the availability of register source operands in register-read stage using the scoreboard structure to check latency mispeculation.

Firstly, we present IPC results. Secondly, we show results about the number of misspeculated instructions. Third, we show the impact of latency between selection and execution stages.

5.1. IPC evaluation

Table 4 shows, for each benchmark, the IPC achieved by B processor in a 4-way processor (Table 1) and the harmonic mean across the benchmark set considering and not considering *mcf* (due to its biased memory behaviour).

Table 4 IPC of the B processor (34 IQ-entries) and two harmonic means (with and without *mcf*).

bzip2	crafty	eon	gap	gcc	gzip	mcf
1.95	1.94	2.07	2.03	1.57	1.73	0.12
parser	perl	twolf	vortex	vpr	HM	HM-mcf
1.03	1.59	0.91	2.40	0.76	0.76	1.44

Fig. 7 shows the speedup of the other evaluated processors with respect to the B processor. We present individual results for each SPEC-2000 integer benchmark and two average values: Avg (including all benchmarks) and Avg-*mcf* (including all benchmarks but *mcf*). These average values are calculated as the ratio of the harmonic means of the IPC. As the number of simulated committed instructions is the same for all benchmarks, we weight all benchmarks equally.

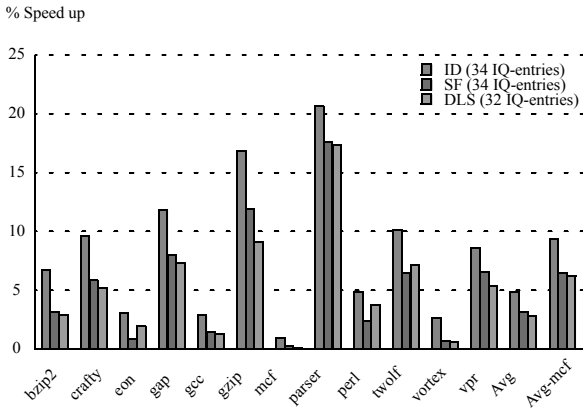

Figure 7 Speed up of ID, DLS and SF processors with respect to B processor (34 IQ-entries). Avg stands for ratio of harmonic means of IPC.

The ID processor outperforms B processor from 20.7% in *parser* to 1.0% in *mcf*, averaging 4.9% (Avg). However, when *mcf* is excluded from the average (Avg-mcf), then the improvement reaches the 9.4%. These results remark the importance of back-to-back execution of dependent instructions.

On average, DLS processor performs within 2.0% (Avg) of ID processor. Performance degradation with respect to ID processor 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 take more than one cycle.

On average, SF processor slightly outperforms DLS processor by 0.3% (Avg and Avg-*mcf*). But in *eon*, *perl* and *twolf*, DLS processor outperforms SF processor (at most, 1.4% in *perl)*. In case of *perl* and *twolf*, wrong-path execution explains performance degradation of SF.

In case of *eon*, the great amount of available ILP let DLS perform better. However, the key observation is that DLS processor misspeculates less issued instructions than SF processor. SF adds a new source of speculation that is aimed at dealing with the latency of a pipelined scheduling logic. This additional speculation source increases the overall number of misspeculations noticeably. Although such unuseful work boosts IPC in SF processor, it wastes energy in both processor frontend and processor backend. To point it out, we show an indirect measure of the wasted energy by classifying the misspeculated issued instructions by the source of their speculation. Later, we analyse the impact of latency between selection stage and execution stage.

5.2. Misspeculated instructions

The overall energy consumed while the processing of a task is proportional to the number of executed instructions and the average work required for processing an instruction. To estimate the energy consumption due to misspeculated instructions, we quantify the number of misspeculated issued instructions classified by the source of their misspeculation. Considering these sources, we know if a misspeculated issued instruction uses front-end processor resources, back-end processor resources or both. We differentiate four categories of misspeculations: a) wrong-path instructions, b) memory-order violations, c) latency misspredictions and d) misspeculated selections (this category applies only to SF processor). The first category uses both front-end and back-end processor resources. The second category uses dispatch and back-end processor resources. The later two categories use processor resources from IQ until register read stage.

In Fig. 8, for each benchmark, we show the misspeculated issued instructions per committed instruction, distributed by misspeculation category in both DLS and SF processors. The dark portion of each column represents the amount of misspeculations of DLS processor. The entire column represents the amount of misspeculations of SF processor. For each benchmark, we show four columns that correspond to the four misspeculation categories. DLS processor always issues less misspeculated instructions than SF processor for each misspeculation category in all benchmarks. In Fig. 8, sometimes this difference can not be appreciated.

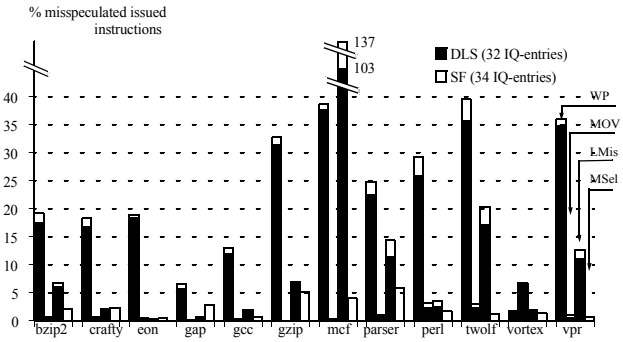

Figure 8 Misspeculated issued instructions per committed instruction, distributed by misspeculation category in both DLS and SF processors. WP stands for wrong-path instructions, MOV for memory-order violations, LMis for latency misspredictions and MSel for misspeculated selections.

In Fig. 9 we show the arithmetic means (Avg and Avg-mcf) of misspeculated issued instructions per committed instruction across SPECInt benchmarks in ID, B, SF and DLS processors. We show the four misspeculation categories and, for each category, we show a group of four columns that corresponds to ID, B, SF and DLS processors.

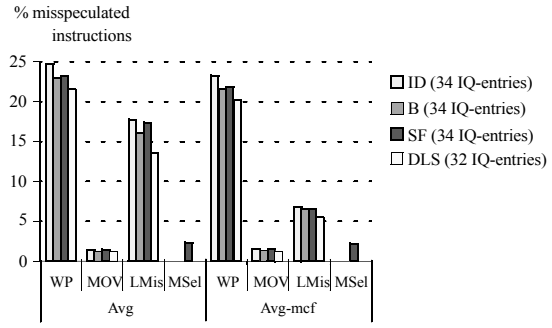

Figure 9 Arithmetic means of misspeculated issued instructions per committed instruction across SPEC benchmarks in ID, B, SF and DLS processors. WP stands for wrong-path instructions, MOV for memory-order violations, LMis for latency misspredictions and MSel for misspeculated selections.

In Fig. 9, we observe significant differences in the category of latency misprediction between Avg and Avg-mcf due to the great number of latency misspredictions in *mcf* (Fig. 8).

The result of descendent sorting the processors by the overall number of misspeculated issued instructions is: SF, ID, B and DLS. SF processor boosts the performance of a two cycle scheduling logic using speculative selection to favour back-to-back execution and it increases with respect to B processor the overall number of misspeculated issued instructions by 10.0%. How-

ever, DLS processor looks for opportunities for back-to-back execution. DLS processor reduces with respect to B processor the overall number of misspeculated issued instructions by a 9.7%. Taking into account the categories WP, MOV and LMis, the result of descendent sorting the processors by the number of misspeculated issued instructions is: ID, SF, B and DLS (excluding latency misprediction category in Avg-mcf). Comparing ID and SF processor, the amount of speculation in branch misspredict ion and latency misspredict ion categories is greater in ID with respecto to SF. However, on behalf of a fair comparison, re-scheduled instructions due to miss-selections prevent SF from going deeper in wrong-path. Note that the sum of all misspeculations in ID and SF are similar.

In SF processor a misspeculated selection occurs when both producer and consumer instructions are issued at the same cycle. When an instruction is dependent on a misspeculated load and it also requires to be re-issued due to a misspeculated selection, we consider it as a misspeculated selection. On Average, the number of misspeculated selections represents a 2.3% of the committed instructions and a 5.3% of the misspredicted issued instructions.

In SF processor, due to speculative selection, branch instruction issuing (and, consequently, branch prediction checking) may be delayed with respect to DLS processor due to the re-issuing of the older misscheduled instructions or because the branch instruction belongs to a chain of misspeculated instructions. Then, SF processor goes into the wrong exection path deeper than B and DLS processors. On average, DLS processor reduces the number of wrong-path issued instructions by a 7.0% with respect to SF processor. DLS processor also executes less wrong-path instructions than B processor (15.4 of reduction), this is caused by the fact that DLS model can resolve branches before than B model because in some cases it can execute one-cycle-dependent instructions back-to-back.

The number of cache misses in SF and DLS processors are very similar and both processors check the latency prediction at the same pipeline stage. However, the number of latency misspeculated instructions in SF processor is greater than in both B and DLS processors due to the aggressive speculative policy in the scheduling logic, that allows SF processor issuing more instructions dependent on the missing load than B and DLS processors. DLS processor looks for safe opportunities for executing back-to-back dependent instructions, while SF processor always favours back-to-back. On average, DLS processor reduces the number of latency misspeculated instructions with respect to SF processor by 21.4%. DLS also reduces in a 5.8% the number of latency mispeculated instructions with respect to B. In DLS, an instruction that depends on a load and a one-cycle instruction is conservatively labelled as *woken up in advance*. After wakeup, this instruction must wait for complete scheduling of current producer level before competing for selection, nevertheless which producer was firstly issued (load instruction or one-cycle instruction). Assuming that one-cycle producer has already wake up in advance, on a L1-miss of producer load, *woken up in advance* dependent instructions scheduling is delayed until the current producer level is scheduled. This delay may prevent to these dependent instructions to be scheduled in the shadow of this L1-miss, and then the number of misspeculated instructions due to L1-misses are reduced. In mcf (Fig. 9), this situation is specially significant.

Also, the misspeculated issued instructions due to memory-order violations is smaller in DLS processor than in SF processor. On average, DLS processor reduces a 17.6% the number of misspeculated issued instruction in this category.

To sum up, DLS processor reduces the number of misspeculated issued instructions in each misspeculation category with respect to SF and B processors. DLS processor reduces the overall number of misspeculated issued instructions a 17.9% with respect to SF and a 9.7% with respect to B processor.

Finally, we have also simulated all processors with the same number of IQ entries (32). The results show that this parameter does not significantly impact the overall number of misspeculated issued instructions. The overall number of misspeculated issued instructions of SF processor is reduced by DLS processor a 17.3%. The number of misspeculated selections in SF processor represents a 6.5% of the misspeculated issued instructions. In this case, on average, DLS processor slightly outperforms SF processor by 0.1%.

5.3. Impact of the latency between selection stage and execution stage

Fig. 10 shows the performance impact of latency between selection stage and execution stage. We consider latencies from one cycle to four cycles, since two cycles is the default value in our evaluations. All processors show a performance degradation if the latency increases, because both the branch-misspredict ion penalty and the load-instruction shadow [16] increase. However, the performance degradation observed in SF processor is greater than in the other processors because the misspeculated-selection penalty increases with the latency. For instance, DLS processor reduces the overall number of misspeculated issued instructions with respect to SF processor by 7.8% and 10.0% (Avg and

Avg-mcf) for 1 cycle, meanwhile for 2 cycles, it becomes a 17.9% and 16.0% (Avg and Avg-mcf), for 3 cycles, it becomes a 33.5% and 23.3% (Avg and Avg-mcf), and for 4 cycles, it becomes a 35.5% and 23.8% (Avg and Avg-mcf).

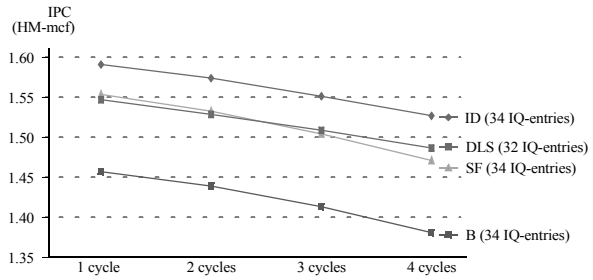

Figure 10 Performance impact of the latency between selection and execution stages in ID, DLS, B and SF processors. Plotted values are the harmonic mean of IPC excluding mcf benchmark

6. Improvements to the DLS mechanism

In this section we analyse DLS behaviour and we propose two improvements to DLS.

First row of Table 5 shows the percentage of execution cycles in which there are no woken in advance instructions. Next four rows show the percentage of execution cycles that Selection Logic devotes to schedule a producer-level that takes one, two, three and four or more cycles to be scheduled, and also, there exists at least one advanced waken up instruction. In case of two or more scheduling cycles, their woken in advance consumers are prevented from competing for selection until completely scheduling the producer-level.

Table 5 Execution cycle distribution according to producer-level scheduling duration in DLS (1, 2, 3, 4 or more cycles).

	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
no WA	34.3	37.6	52.0	34.6	61.1	29.4	94.9	49.5	49.3	59.3	38.1	70.0
1 cycle	38.0	37.6	37.2	46.7	28.9	37.9	4.5	40.1	34.4	32.2	35.1	23.1
2 cycle	21.6	19.1	8.3	16.3	7.3	21.1	0.6	7.9	12.7	6.8	16.5	6.4
3 cycle	4.8	4.9	1.9	1.8	2.3	10.5	0.0	2.1	3.2	1.5	7.5	0.5
≥ 4 cycles	1.3	0.8	0.6	0.6	0.4	1.1	0.0	0.4	0.4	0.2	2.8	0.0

In DLS processor, back-to-back execution of one-cycle producer instructions and their dependent instructions is achieved when the producer-level instructions are scheduled in one cycle. In Table 5 we observe that, except *mcf*, the percentage of cycles in which no woken in advance instructions (first row) plus the percentage of cycles in which the scheduling of producer-level instructions takes one cycle ranges from 67.3% (*gzip*) to 93.1% (*vpr*). During this percentage of cycles, DLS processor performs like the ideal processor (ID).

In DLS processor, when the scheduling of producer-level instructions takes several cycles, then woken in advance instructions are conservatively prevented from competing for selection although their producer instructions have been already selected. In this scenario, reflected by bottom three rows in Table 5, DLS processor expects that available ILP maintains throughput of issued instructions, that is, all issue ports will be busy every cycle. Thus during scheduling cycles of producer-level instructions, performance may be degraded due to two reasons: a) some issue ports are not busy and these issue ports could have been devoted to schedule woken in advance instructions whose producer instructions have been already scheduled and b) although issue bandwidth is fully exploited every cycle, some woken in advance instructions may be older than any instruction that competes for selection.

If the scheduling of producer-level instructions takes two cycles, DLS processor is not outperformed by the baseline processor (B). However when the scheduling takes even more cycles, then DLS processor may be outperformed by B processor.

The following subsections describe two improvements to DLS processor. The goal of these improvements is to allow competing for selection some instructions that in DLS processor were prevented from competing for selection. In following subsections we describe the improvements and in a later subsection we present their evaluation.

6.1. Taking into account instructions without consumer instructions in issue queue (DLS-WC)

The distance between a producer and its consumers instructions is usually short. However, there is a representative fraction of one-cycle instructions that do not have consumer instructions in IQ at issue time. In DLS processor, when these producer instructions belong to a producer-level, they could prevent the instructions that have been woken in advance by the remaining instructions of the producer-level from competing for selection. Table 6 shows the percentage of times, that all one-cycle latency instructions competing for selection do not have a consumer instruction in IQ. This percentage is classified according to the time devoted to schedule producer-level instructions that are currently competing for selection. The idea is that producer-level instructions without consumer instructions in the issue queue do not prevent woken in advance instructions from competing for selection. For this, at dispatch time, DLS-WC mechanism also tags as *without-consumer* any instruction classified as *wakeup in advance*. Later, at dispatch time of the first consumer instruction, DLS-WC mechanism untags the producer instruction if it still remains in the issue queue. An instruction tagged as *without-consumer*

is not accounted by ZDL and then, this instruction does not prevent waken in advance instructions from competing for selection.

Table 6 For each producer-level scheduling duration, percentage of times that all instructions competing for selection do not have consumer instructions in the issue queue.

Scheduling Cycle Duration	bzip2	crafty	eon	gap	gcc	gzip
2 cycle	4.3	2.4	1.5	3.0	1.6	3.5
≥ 3 cycles	0.9	1.1	0.5	0.6	0.5	2.2
	mcf	parser	perl	twolf	vortex	vpr
2 cycle	0.02	1.5	2.3	0.6	3.4	0.6
≥ 3 cycles	0.00	0.5	0.5	0.3	2.7	0.1

6.2. Letting woken up in advance instructions compete for selection (DLS-B)

In DLS mechanism, during the scheduling of producer-level instructions, woken in advance instructions are not allowed to compete for selection although their producer instructions may be already selected. The idea to improve DLS mechanism is to make safe use of instruction age in order to let some woken in advance instruction compete for selection before producer-level instructions had been completely scheduled.

In DLS-B mechanism, a woken in advance instruction is also allowed to compete for selection only if it is older than the oldest instruction which remains competing for selection. This approach is conservative because not all woken in advance instructions whose producer instructions are already selected are allowed to compete for selection. An instruction still competing for selection that is older than the woken in advance consumer instructions but younger than the already selected producer instructions, prevents the consumer instruction from competing for selection (note that a really ready consumer instruction will be disabled for selection by any older non-selected instruction that currently is competing for selection). The implementation of this selection policy is simple in a issue queue that maintains the instructions physically ordered by compacting the issue queue entries [10] or by using a circular structure [19]. In selection phase of these schedulers, a request signal generates a kill signal to disable lower-priority request signals. DLS-B mechanism adds another kill signal: a request signal from a producer-level instruction disables younger request signals from waken in advance instructions.

Fig. 11 shows an example of scheduling an instruction sequence with DLS-B mechanism. All four instructions are labelled as *wakeup in advance*, but only instruction 3 is labelled as *woken up in advance*. Instructions 1, 2 and 4 form a producer-level, and instruction 3 forms the consumer-level.

In cycle 1, instructions 1, 2 and 4 (one-cycle instructions) are ready. During cycle 2, instruction 3 is woken up in advance. In cycle 3, as instruction 3 is younger than instruction 2, instruction 3 is not allowed to compete for selection. Instruction 2 is older than instruction 4 and, it is selected in cycle 3. In cycle 4, instruction 3 is allowed to compete for selection because it is older than instruction 4; then instruction 3 is selected. Being the oldest instruction of both the producer-level and the woken in advance instructions ensures that data dependences of instruction 3 are satisfied.

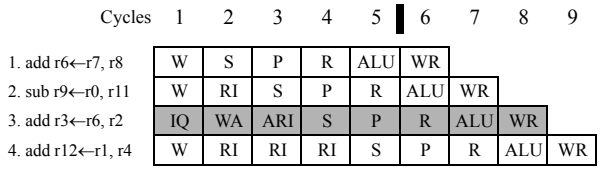

Figure 11 Scheduling example with DLS-B mechanism

6.3. Evaluation

In this section, we show the performance results for both improvements to the DLS processor (DLS-WC and DLS-B). Fig. 12 shows the speedup achieved by these processors with respect to the B processor.

Focusing on DLS-WC processor, on average, a slight improvement is achieved. DLS-WC processor outperforms DLS processor in all benchmarks except in *gap* and *vpr*. The performance increment depends on the benchmark and it is significant in *gzip* (2.8%).

DLS-B processor outperforms DLS-WC processor. On average, it boosts DLS processor by 0.5% and it is within 1.5% of ID processor. The speedup of DLS-B processor with respect to DLS processor in benchmarks *crafty*, *gzip* and *vpr* ranges from 1.3% to 3.4%. However, in *eon* benchmark, DLS processor and DLS-WC processor outperform DLS-B processor. In *eon* benchmark, DLS-B processor issues 0.22 million more instructions than DLS processor, however, DLS-B processor still issues less instructions than SF processor.

Also, these improvements to DLS processor reduce the overall number of misspeculated issued instructions with respect to SF processor up to 18.7% (Avg DLS-B) and 18.4% (Avg DLS-WC). In base DLS, *woken up in advance* instructions wait for the scheduling of the current producer-level, although their source operands are ready. DLS-WC and DLS-B try to promote these unnecessarily-waiting instructions and let them to compete for selection before the current producer-level has been scheduled. These promoted instructions may be either speculative or non-speculative, anyway more ILP is exposed and branches may be resolved earlier (less misspeculations).

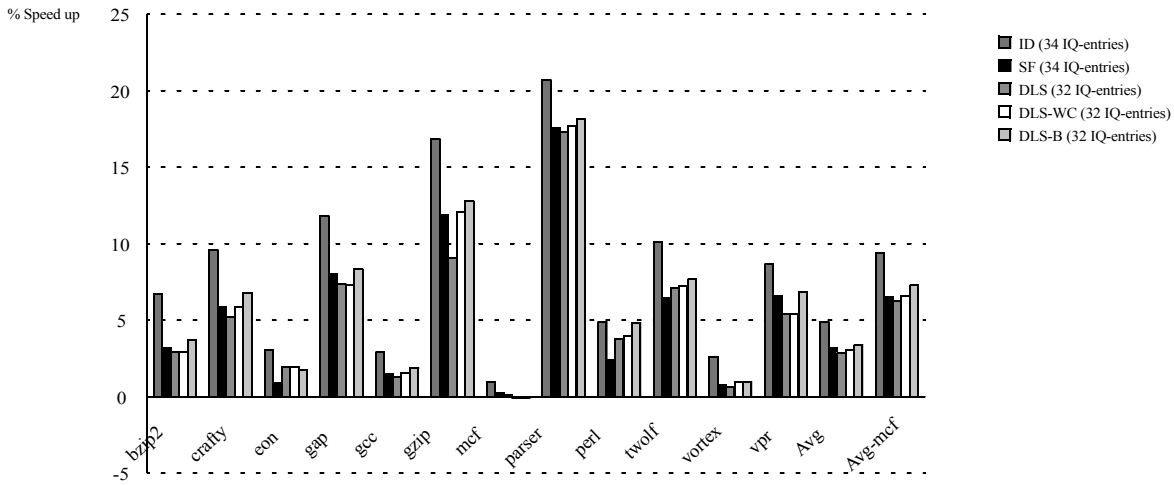

Figure 12 Speedup of ID, SF, DLS, DLS-WC and DLS-B processors with respect to B processor (34 IQ-entries). Avg stands for ratio of harmonic means of IPC.

7. Related works

Researchers have addressed the scalability and atomicity of instruction schedulers using different techniques. Prescheduling allows shortening or even eliminating the wakeup phase of instruction schedulers [5][6][9][20][23][25]. Wakeup latency is addressed by reducing the load capacitance of the wakeup tag bus [8][12][18] or by using index-based wakeup [5][6][12][14][21][29][32]. In several works the issue queue is partitioned or segmented in order to reduce the scheduling-loop latency[3][13]. Other proposals narrow the implementation of the issue queue taking into account that the distance between producer and consumer instructions is short [12] or that some producers do not have consumer instructions in the issue queue at issue time [28]. In the context of pipelined schedulers, some proposals hide the scheduling-loop latency by grouping instructions because grouping increases the scheduling granularity [2][15][17][26][27]; also, in order not to sacrifice the ability of back-to-back execution, other proposals schedule instructions speculatively.

Several works overlap the selection phase of a producer instruction with the wakeup phase of their dependent instructions. J. Stark et al. [31] proposed to speculatively wake instructions up by their grand-parents in order to tolerate the scheduling logic latency. An additional Wakeup Logic driven by parent instructions confirms if each selected instruction has their source operands available. R. Gran et al. [11] add a Wakeup Logic to a base two-cycle scheduling logic in order to detect opportunities to execute dependent instructions back-to-back. This added Wakeup Logic keeps instructions dependent on one-cycle instructions; they are woken up when the parent group of one-cycle instructions starts competing for selection. There exists an opportunity to execute back-to-back dependent instructions when all instructions of the parent group of one-cycle instructions have been selected. While [31] is speculative, [11] is not. Moreover, both proposals use two Wakeup Logics which duplicate the cost of the wakeup phase.

M.D. Brown et al. [4] 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. This mechanism wastes energy due to re-schedulings. Their evaluations assumes both perfect load latency prediction and perfect memory-dependence prediction. However, in our evaluations we predict both of them and we measure the impact of several misspeculation sources separately. In this scenario, the non-speculative mechanism proposed in this paper (DLS) reduces the overall number of misspeculated issued instructions with respect to SF by 17.9% and SF slightly outperforms DLS by 0.3%.

8. 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, the Dependency Level Scheduler (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 mechanism is to look for opportunities to execute dependent instructions back-to-back and use these opportunities safely. Key differences of DLS mechanism with respect to previous proposals, that tolerate the

latency in a pipelined scheduling logic, is that DLS mechanism is non-speculative and it does not duplicate hardware cost in Wakeup Logic.

We have shown that the number of misspeculated instructions issued by DLS processor is smaller than the misspeculated instructions issued by a speculative scheduler and thus, the wasted energy is smaller. We indirectly have evaluated the wasted energy by quantifying the amount of misspeculated issued instructions. DLS processor reduces the overall number of misspeculated issued instructions with respect to a speculative scheduler by 17.9% while it is outperformed by the speculative scheduler only by 0.3%. Also, we have shown that when the number of stages between issue and execution increases, then DLS processor reduces even more the number of misspeculated issued instructions with respect to the speculative scheduler, and moreover, DLS processor slightly outperforms the speculative scheduler.

DLS processor performs, on average, within 2.0% of an ideal processor (unpipelined scheduler). Improvements to DLS mechanism reduce this performance gap. The best of these improvements performs within the 1.5% of an ideal processor and it reduces the overall number of misspeculated instruction with respect to a speculative scheduler by a 18.7%.

References

[1] D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," UW Madison Computer Science T. R.

[2] A. Bracy et al. Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. Micro-2004, p. 18 - 29.

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

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

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

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

[7] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. ISCA-1998, p. 142-153.

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

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

[10] 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.

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

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

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

[14] 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.

[15] S. Hu et al. An Approach for Implementing Efficient Superscalar CISC Processors. HPCA-2006, p. 40-51.

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

[17] I. Kim and M.H. Lipasti. Macro-op Scheduling: Relaxing Scheduling Loop Constraints. Micro-2003, p. 277-288.

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

[19] 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

[20] P. Michaud et al.. Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. HPCA01. p. 27.

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

[22] S. Önder and R. Gupta. Instruction Wake-up in Wide Issue Superscalars. Europar-2001, p. 418-427.

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

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

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

[26] H. Sasaki et al.. Energy-Efficient Dynamic Instruction Scheduling Logic through Instruction Grouping. ISLPED-2006. p. 43-48.

[27] P. G. Sassone et al.. Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication. Micro-2004, p. 7 - 17.

[28] P. Sassone, J. Rupley, E. Brekelbaum. G. Loh and B. Black. Matrix Scheduler Reloaded. ISCA.2007, p. 335--346.

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

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

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

[32] S. Weiss and J.E. Smith. Instruction Issue Logic in Pipelined Supercomputers. ISCA. 1984. p. 110-118