An Enhancement for a Scheduling Logic Pipelined over two Cycles

Rubén Gran, Enric Morancho, Àngel Olivé, José M. Llabería,
Departamento de Arquitectura de Computadores. Universidad Politécnica de Cataluña.
{rgran, enricm, angel, llaberia}@ac.upc.edu

Abstract - **Out of order processors use the dynamic scheduling logic both to expose and to exploit parallelism. Pipelining this logic may sacrifice the ability to execute dependent instructions in consecutive cycles. Several previous studies have shown that pipelining the scheduling logic over two cycles degrades performance; our evaluations, in a 4-way machine, on SPEC-2000 integer benchmarks show a performance degradation about 11% compared to an unpipelined scheduling logic.**

In this work, we present two non-speculative enhancements for a scheduling logic pipelined over two cycles. The idea is computing in advance which instructions will be woken-up by all instructions that are currently competing for selection. Once all of them have been selected, the pre-computed group of instructions can compete for selection in next cycle. The enhancement goal is to tolerate the scheduling-loop latency when not enough ILP is available through the scheduling of dependent instructions in consecutive cycles.

Our results in a 4-way machine show that our two proposed enhancements perform, on average, slightly better than two previously proposed speculative schedulers. The performance of our proposals is within a 2.6% and 2% of an unpipelined ideal scheduler.

Index Terms - **Back-to-back execution, dynamic scheduler, pipelined scheduling logic.**

I. Introduction

The dynamic scheduling logic allows both exposing and exploiting the instruction-level parallelism (ILP). The scheduling task is divided into two phases: wakeup and select. The wakeup logic marks instructions as ready when their data dependencies are satisfied. The select logic picks instructions for execution from the pool of ready instructions by considering instruction priorities and available resources.

Both the wakeup logic and the select logic form a hardware loop, the scheduling loop, because an instruction cannot be scheduled until its producer instructions have been scheduled. Assuming that the producer-instruction latency is one cycle then, in order to execute its dependent instructions in consecutive cycles, the scheduling task must be performed in one cycle. A producer instruction and its consumer instruction are executed back-to-back when the consumer instruction consumes the produced result as soon as it is available.

Manuscript received May 8, 2006. 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.

Enlarging the issue queue to expose more ILP may increase the latency needed to wakeup and select instructions, which may require reducing clock frequency. An approach to either maintaining or increasing clock frequency is pipelining the scheduling logic over several cycles, but then the IPC may decrease because the scheduling logic sometimes is unable to issue dependent instructions in consecutive cycles. Our experimental results with SPEC-2000 integer benchmarks in a 4-issue machine show that pipelining the scheduling logic over two cycles degrades IPC, on average, about 11% compared to an unpipelined scheduling logic. Other authors report similar results ([2], [16], [20]).

Techniques that allow pipelining the scheduling logic without sacrificing the back-to-back execution of dependent instructions are an option to design high-frequency processors. However, some previously proposed techniques ([2], [20]) are speculative.

In this paper, we enhance a scheduling logic pipelined over two cycles to increase its performance. The proposed enhancement is non-speculative and able to execute dependent instructions in consecutive cycles when not enough ILP is available. Consequently, we manage to tolerate the scheduling-logic latency. The idea of the enhancement is computing in advance which instruction group will be woken up by all one-cycle execution-latency instructions that are currently competing for selection. Then, once all these instructions have been selected, the precomputed instruction group can compete for selection in next cycle and back-to-back execution may be performed.

Our results show that our two proposed enhancements outperform, on average, two previously proposed speculative schedulers ([2], [20]) on SPEC-2000 integer benchmarks. The performance of our proposals is within a *2.6%* and *2%* of an ideal scheduler (unpipelined).

This paper is structured as follows: Section II outlines the processor model being used and motivates the work. Section III describes the proposed enhancement. Section IV details the simulation environment. Section V evaluates the proposed models and compares them to two previously proposed mechanisms. Section VI discusses related work and Section VII concludes this paper.

II. Baseline processor model

Figure 1 shows the pipeline of a dynamically scheduled processor, where each stage can take one or more cycles.

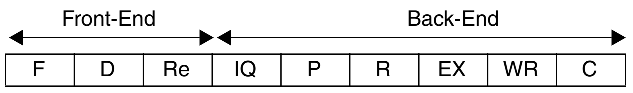


Figure 1 Processor Pipeline. F: Fetch, D: Decode, Re: Rename, IQ: Issue Queue, P: Read Payload, R: Read Register File, EX: Execution; WR: Write Register File, C: Commit.

In the front-end stages of the pipeline (fetch, decode and rename stages), instructions are brought from the instruction cache, decoded and false register dependencies are removed. After that, the instructions are dispatched into the issue queue and wait there for the availability of both their source operands and execution resources. When an instruction is selected for execution, the payload and its source registers are read in following cycles. With its source operands, the instruction is executed and its result is written into the register file. Finally, the instruction waits for committing in program order.

Wakeup logic. We use a wired-OR style wakeup logic array ([2], [8]). Dependencies are indicated using an *instructions-instruction* (each wakeup-matrix row and each wakeup-matrix column corresponds to an instruction inserted into the issue queue) wakeup matrix [2] (or *physical registers-instructions* wakeup matrix [8]). Bit vectors (rows) perform dependence tracking. Each bit in the vector represents the dependence on a parent instruction [2] (or on the data availability of a physical register [8]). When an instruction is issued, it sets the wakeup line (column) corresponding to its own issue-queue entry [2] (or to its destination physical register [8]). Each instruction monitors the readiness of its source operands every cycle by checking if all wakeup lines of matching dependence bits are set. Each issue-queue entry corresponding to a ready instruction activates a request signal in order to notify its readiness.

Select logic. The input of the select logic are request signals from the wakeup logic plus priority information. The select logic picks the oldest ready instructions considering available resources at each issue port. Instructions selected by the select logic become the input of the wakeup logic in next clock cycle in order to wakeup instructions dependent on the selected ones.

Figure 2 shows diagrams of both one-cycle latency (unpipelined) and two-cycle latency scheduling loops. 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. In [2], [16], [20], their authors have concluded that back-to-back execution is a performance goal.

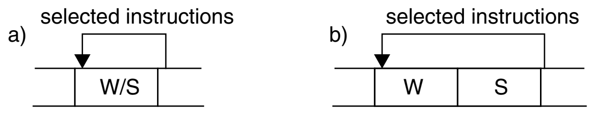


Figure 2 Diagrams of scheduling loops. a) one-cycle latency, b) two-cycle latency. (W: Wakeup, S: Select)

Table 1 shows the distribution of committed instructions on SPEC-2000 benchmarks considering their execution latency and if the instructions produce a value that is stored in the register file (Section IV details benchmarks, simulated intervals and the execution latency of the instructions). We observe that integer benchmarks double the amount of one-cycle execution-latency instructions compared to floating-point benchmarks; consequently, integer benchmarks will be more sensitive to the scheduling-loop latency.

TABLE I. DISTRIBUTION OF COMMITED INSTRUCTIONS ACCORDING TO THEIR EXECUTION LATENCY ON SPEC-2000.

	updating register file		not updating register file
	execution latency		
Benchmarks	one cycle	multicycle	
Integer	44.30%	32.05%	23.65%
Floating Point	23.62%	62.40%	13.98%

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

III. Enhanced scheduling logic

In this section we describe a non-speculative enhancement that improves the performance of a two-cycle latency scheduling logic.

A. Base enhancement (E)

The enhancement is applied only to instructions that may be woken up by instructions with an execution latency shorter than the scheduling-loop latency. The remaining instructions use the conventional two-cycle latency scheduling loop.

The idea to enhance the two-cycle latency scheduling logic is to compute in advance which instructions will be woken up by all one-cycle latency instructions which are currently competing for selection. Once all of them have been selected, the precomputed group of instructions competes for selection next cycle. Therefore, back-to-back execution is possible.

Figure 3 shows a scheme of a two-cycle latency scheduling logic with the proposed enhancement logic. In Figure 3, two parts are distinguished: a) the base two-cycle latency scheduling logic and b) the enhancement logic. The elements of the base two-cycle latency scheduling logic are the wakeup matrix A and the select logic. Remaining elements in Figure 3 enhance the base scheduling logic.

At dispatch time, all instructions are stored in wakeup matrix A. The output of the select logic becomes the input of the wakeup matrix A in next cycle. Each selection cycle, the select logic picks the oldest instructions that remain in its input, considering resource availability at issue ports.

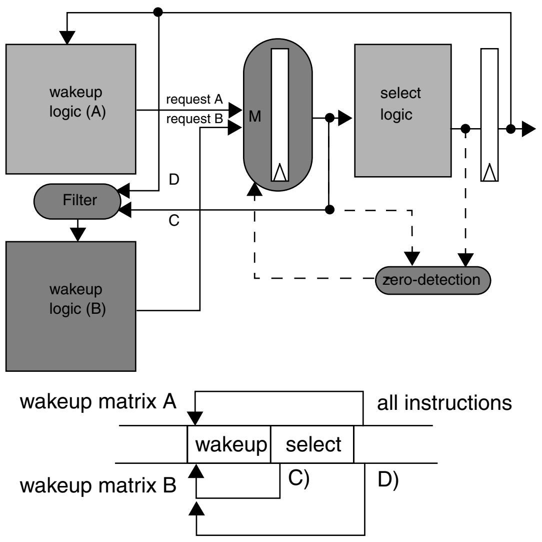


Figure 3 Block diagram of the base proposal.

The wakeup matrix B computes in advance which instructions will be woken up by the one-cycle latency instructions which are currently in the input of the select logic. Then, in wakeup matrix B are stored (the content of the entry allocated to an instruction in both matrices is the same) only instructions that may be woken up by one-cycle execution-latency instructions (their latency is shorter than the scheduling-logic latency). The classification of each instruction is performed in dispatch phase considering the latency of the instructions that wake it up (parent instructions).

Instructions dependent on at least one instruction which execution latency is shorter than the scheduling-logic latency are stored in wakeup matrix B. Therefore, inputs of the wakeup matrix B are: a) one-cycle execution-latency instructions in the input of the select logic and b) instructions selected by the select logic that have an execution latency greater than or equal to the scheduling-logic latency. These inputs are calculated by the filter shown in Figure 3 using an instruction classification performed in decode stage: instructions with an execution latency shorter than the scheduling-logic latency belong to a class, and the remaining instructions belong to the other class. Detailed implementation for a slice of the filter (Figure 3), is given in Figure 4.

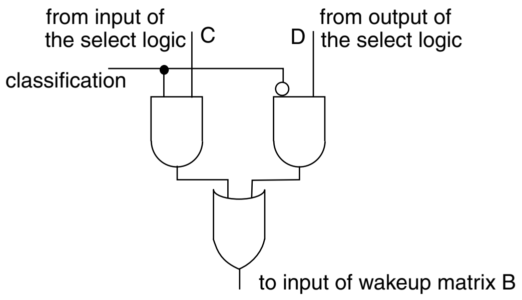

from input of the select logic
C
D
from output of the select logic
classification
to input of wakeup matrix B

Figure 4 Detail of slice of the filter logic. Logic filters signals coming from D and C (Figure 3). Signal D is discarded if the instruction latency is shorter than the scheduling-logic latency. Signal C is discarded if the instruction latency is greater than or equal to the scheduling-logic latency.

The zero-detection logic (ZDL) detects if all one-cycle execution-latency instructions currently competing for selection were actually selected, and therefore it is safe for a ready instruction in wakeup matrix B to proceed to selection.

Scheduling an instruction group with execution-latency shorter than the scheduling-loop latency in the input of the select logic may take one or several cycles.

If the scheduling takes one cycle, then the scheduling of the instruction group overlaps with the computation in advance of which instructions are woken up by this instruction group. The wakeup matrix B computes in advance woken up instructions and the ZDL detects that all instructions with execution latency shorter than the scheduling-logic latency have been scheduled. Then, in next cycle back-to-back execution of dependent instructions is performed.

If the scheduling takes several cycles, then behaviour in first cycle differs from behaviour in remaining ones. In first cycle, dependent instructions in wakeup matrix B wake up, and they wait for the activation of ZDL. In the elapsed time between the first and the last scheduling cycle, selected instructions with one-cycle execution latency wake dependent instructions up in wakeup matrix A. Next cycle after waking up an instruction in wakeup matrix A, its request signal will be in the input of the select stage, and then it will compete for selection. In the cycle next to the last scheduling cycle, back-to-back execution may be performed.

Next, we describe the functionality of the logic M. The request signal of an instruction that is woken up by an one-cycle execution latency instruction is activated in both wakeup matrices, but only one request signal should be observed by select logic. Request signals are activated either in the same cycle or in different ones.

The request signals are activated in different cycles if an instruction is woken up by an one-cycle latency instruction. First, the request signal is activated in wakeup matrix B when the parent instruction is in the input of the select logic. Later, when the parent is selected, the request signal is activated in wakeup matrix A. Moreover, if during the elapsed time between both events the output of ZDL is not activated, then both requests will be concurrently activated.

The request signals are activated in the same cycle, when the latest arriving operand of an instruction stored in wakeup matrix B is produced by an instruction whose execution latency is greater than or equal to the scheduler-logic latency.

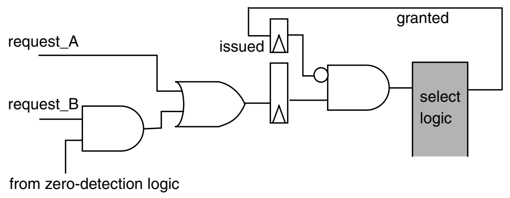


Figure 5 Logic M. Slice corresponding to one entry of the issue queue. Request_A and request_B stand for a request signal of the wakeup matrix A and B respectively. Issued bit nullifies request signal when the request signal has been previously granted.

Figure 5 shows detailed implementation of logic M. When the request signal of an instruction is activated in the wakeup matrix A, the request signal will be in the input of the select stage in the next cycle. The request signal of an entry in the wakeup matrix B will be in the input of the select stage in next cycle only if the output of the ZDL is activated.

Select logic observes only one request signal for every instruction (Figure 5). The issued bit filters out a request signal that arrives to the input of the select stage after selecting the instruction.

Note that the scheduling scheme may break the *oldest first* instruction-selection policy. Instructions woken up from matrix B can not be selected until selecting all one-cycle instructions in the input of the select logic. Then, an instruction woken up from matrix B may be waiting until selecting a younger (but one-cycle) instruction.

Figure 6 shows an example of the scheduling of a sequence of instructions 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. A and B labels mean that the instruction wakes up in wakeup matrix A and B respectively. The RI label symbolizes that the instruction is waiting for selection in the input of the select logic. The S label means that the instruction is selected for execution. Labels @, M1 and M2 mean, respectively, effective-address computation and the two cycles needed to access first-level cache. Shadowed rows indicate that instructions have been inserted in both wakeup matrices (non shadowed rows indicate that instructions are inserted only in matrix A).

In a cycle, an arrow indicates which instructions activate wakeup-matrix lines. Each arrow starts at the producer instruction that activates its corresponding wakeup-matrix line; each arrow ends at the consumer instructions. There are three kinds of arrows according to the involved wakeup matrix: filled arrows (matrix A), stick arrows (matrix B) and hollow arrows (both matrices at the same time).

We assume that the operands of instructions 1 and 2 are available at dispatch time, consequently both instructions wake up in cycle 1. Instruction 1 is selected in cycle 2 and, as its latency is larger than the scheduling-logic latency, its wakeup lines are activated in both matrices at the same time (cycle 4). In this example, as its dependent instructions are inserted only in matrix A, no instruction benefits from the wakeup-line activation in matrix B. Similarly, instruction 2 is selected in cycle 3 and both matrices are aware of the availability of its result in cycle 5.

In cycle 4, instruction 3 wakes up in wakeup matrix A. As it is an one-cycle instruction, while it competes for selection, in cycle 5, the availability of its result is notified to matrix B. The wakeup of instruction 4 is independent of the selection of instruction 3 in cycle 4. Moreover, once the instruction 3 is selected, the availability of its result is notified to matrix A in cycle 6.

In cycle 5, instruction 4 wakes up in matrix B and instruction 5 wakes up in matrix A. In cycle 6, both instructions compete for selection because, at the end of cycle 5, the ZDL activated the signal that allows ready instructions from wakeup matrix B to compete for selection. In cycle 6, the instruction 4 wakes up in matrix A; however, as the instruction 4 is already competing for selection, the M logic (by means of the issued bits shown in Figure 5) filters out this request signal in next cycle. The instruction 4 is selected in cycle 6, and the instruction 5 is selected in cycle 7.

Finally, the instruction 6 wakes up in both matrices in cycle 8 because its last arriving operand (r4) is produced by an instruction with an execution latency larger than or equal to the scheduling-logic latency. Selecting this kind of instructions is notified to both matrices at the same time.

Note that, in this example, our proposal allows issuing the instruction 4 one cycle earlier than the conventional scheduling logic pipelined over two cycles.

B. *Adding instruction fusing (E-F)*

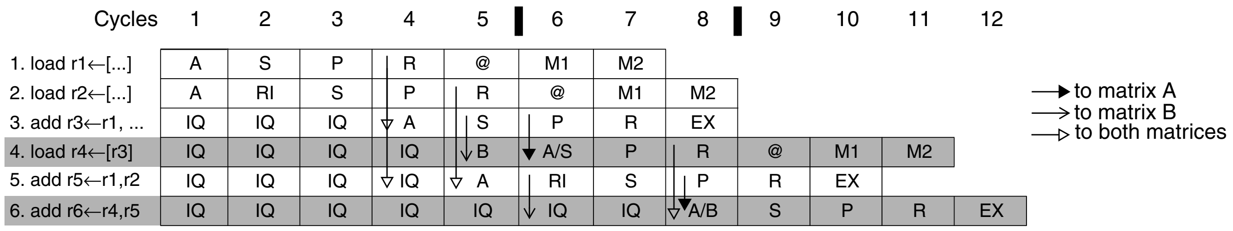
The proposed enhancement can be improved by taking advantage of a program characteristic: a large number of instructions has only one source operand (avg: *78.6%* of committed instructions in SPECInt 2000). Moreover, at dispatch time, some two-operand instructions have already available one of them. Consequently, only one operand must be tracked by the wakeup matrix.

Then, we make use of fusing instructions (a producer instruction and its dependent one) in order to favour back-to-back execution of dependent instructions. Two instructions are fused when producer instruction is an one-cycle execution-latency instruction and the consumer instruction only depends on this instruction.

In our model E-F, the advantages of fusing instructions are twofold. First, the consumer instruction can compete for selection once the producer instruction has been selected. Therefore, back-to-back execution is possible. Second, it is not necessary to store the consumer instruction in the wakeup matrix B for waking it up.

The possibility of fusion is detected in dispatch phase. The fused instructions must belong to the same dynamic basic block and the producer instruction must be in the issue queue.

The instruction fused with its producer instruction is the first one in program order that satisfies the previous conditions. Note that the proposed instruction fusing is simple because both producer and consumer instructions belong to the same basic block.



Cycles	1	2	3	4	5	6	7	8	9	10	11	12
1. load r1←[...]	A	S	P	R	@	M1	M2					
2. load r2←[...]	A	RI	S	P	R	@	M1	M2				
3. add r3←r1, ...	IQ	IQ	IQ	A	S	P	R	EX				
4. load r4←[r3]	IQ	IQ	IQ	IQ	B	A/S	P	R	@	M1	M2	
5. add r5←r1,r2	IQ	IQ	IQ	IQ	A	RI	S	P	R	EX		
6. add r6←r4,r5	IQ	IQ	IQ	IQ	IQ	IQ	IQ	A/B	S	P	R	EX

Figure 6 Scheduling example of the proposed mechanism. A bar between cycles indicates that the zero-detection logic activates its signal; therefore, in next cycle, the request signals from wakeup matrix B will be added to the input of select logic (ZDL).

In our evaluations, two issue-queue entries are allocated to fused instructions in wakeup matrix A and two issue cycles are needed to schedule them. Therefore, we maintain the same pressure than previous models over the issue-queue entries of wakeup matrix A and the issue ports.

IV. Simulation environment

A. Processor model

We have modified SimpleScalar 3.0d [2] in order to model a Reorder Buffer and separate issue queues (IQ). We assume an out-of-order processor with fifteen stages from Fetch to IQ and two stages between IQ and Execution. Table II details other processor and memory parameters. Table III lists the instruction latencies assumed in this work.

TABLE II. PROCESSOR & MEMORY PARAMETERS

	Model
Fetch and Decode width	4 inst/cycle
Branch predictor: hybrid (bimodal, gshare)	16 bits
ROB size / LSQ size	128 / 64 entries
Issue-queue size Integer / Floating point	32 / 20 entries
Functional Units Integer / Floating point	4 / 2
Memory access ports	2

Memory hierarchy	
L1 I-cache and L1 D-cache	32KB, 4-way, 2 cycles
Line size	32 B
L2 Unified Cache	256 KB, 4-way, 12 cycles
Line size	32 B
L2-Main memory bus	8B / 2 cycles
Main memory latency	100 cycles

TABLE III. EXECUTION LATENCY OF THE INSTRUCTIONS (CYCLES)

	Latency		Latency
ALU	1	Floating point add, mul	4 pipelined
Load	3	Floating point divide	15 not pipelined
integer multiply	10 not pipelined	others	1

We split store instructions into two instructions: STA (store address computation) and STD (store data). Therefore, two issue-queue entries are allocated to each store instruction.

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

The IQ is divided into an integer IQ and a floating-point IQ. Our proposals are applied only to the integer IQ because the execution latency of most FP instructions is greater than the scheduling-loop latency.

B. Workload

We use SPEC2000 integer benchmarks compiled with full optimizations on an Alpha machine. We simulate a contiguous run of 100M-instruction from SimPoint [19] after a warming-up of 100M-instruction. Table IV shows their input data sets.

TABLE IV. 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		

V. Results

To evaluate the performance of our proposed enhancement we have simulated several models with a two-cycle latency scheduling loop.

- A baseline model (B) where back-to-back execution of dependent instructions is sacrificed when producer instructions have one-cycle execution latency. Also, we model instruction fusing (B-F) with same conditions than in Section B.
- Two models implementing our proposals: E and E-F.
- For comparison purposes, B-Double model doubles the number of integer issue-queue entries of the baseline model. This model is intended for showing us what is more cost-effective: dedicating added IQ entries to either expose more parallelism or favour the back-to-back execution of dependent instructions.
- For comparison purposes, we model the Speculative Wakeup (SW, [20]) and the Select-Free (SF, [2]) mechanisms. Both are speculative mechanisms designed to tolerate the scheduling-logic latency. They are described in Section VI. In our evaluations, the SW mechanism is implemented by using two wakeup matrices (the first one for tracking the parent instructions and the second one for tracking the grandparent instructions). In the SF mechanism, speculation is checked in register-read stage.

Moreover, we simulate an ideal model (ID) with unpipelined

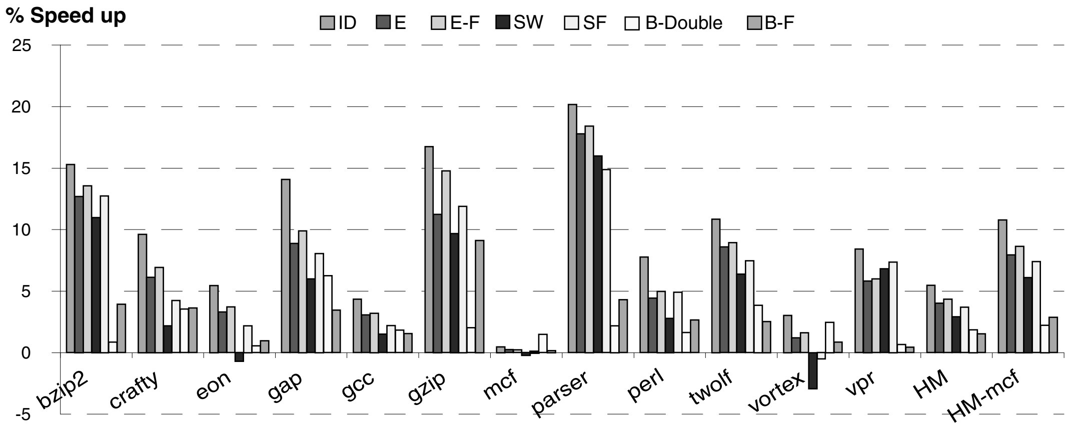


Figure 7 Speed-up with respect to the B model

scheduling-loop (that is, its latency is one cycle). 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 one extra stage in the front-end.

Table V presents our baseline IPC results.

TABLE V. IPC OF THE BASELINE MODEL (B).

bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
1.34	1.86	2.12	1.94	1.38	1.60	0.13	1.01	1.38	0.88	2.38	0.77

Figure 7 shows the speed-up of all models compared to the B model. We present individual results for each SPEC-2000 integer benchmark and two average values: for all benchmarks (HM) and for all benchmarks but *mcf* (HM-mcf) due to its biased memory behaviour.

Our proposed models, on average, outperform both the B-Double and the B-F models. The B-Double model outperforms our proposed models in benchmarks *mcf* and *vortex*. However, in the other benchmarks, doubling the number of issue-queue entries to expose more parallelism is not cost-effective. It is better to favour the back-to-back execution of dependent instructions. Our proposed models also outperform B-F model in all benchmarks. B-F model outperforms, on average, about *2.8%* the B model. However, in *gzip*, performance improvement reaches *9.1%.* This speedup is explained by the great amount (23%) of fused instructions in dispatch. However, the number of fusions and the speedup are not correlated, by instance *bzip2* has a *20%* of fused instructions in dispatch and its speedup is smaller (*3.9%*) than in *gzip*.

We observe that our proposed models, on average, outperform the speculative models (SW and SF). The SW model outperforms our proposed models only in benchmark *vpr*; the SF model outperforms E model in benchmarks *bzip2, gzip, perl and vpr*. However, the E-F model is outperformed by the SF model only in benchmark *vpr.*

Performance of E and E-F models are, on average, within *2.6%* and *2%* of the ID model, respectively. In the E-F model, instruction fusing permits a consumer instruction, that has been fused with its producer, to avoid waiting for the next merge operation to compete for selection. Otherwise, in the E model, consumer instructions have to wait for scheduling all one-cycle execution-latency instructions, that are currently competing for selection. Therefore, in the E-F model, those instructions could save some cycles to reach the selection stage. On average, a *19.2%* of dispatched instructions are fused with their producer instruction.

While both the SF and the SW models are speculative, our proposed models are not. The SF model must re-schedule some instructions, that have been speculatively woken-up. This involves activity, which wastes energy, in both the select logic and the register file. Our evaluations show that, on average, in the SF model the re-schedulings affect to a 3.4% of committed instructions and a 3.0% of selections by the select logic.

The SW model may select instructions whose selection will be later nullified because parent instructions have not been issued. These *false selections* affect, on average, to a 7.6% of the committed instructions and a 4.6% of selections by the select logic.

The SW model and our proposed models use two wakeup matrices. In the SW model, all instructions are stored in both wakeup matrices. However, in our proposed models, the wakeup matrix B stores bit dependence vectors of an instruction only if it can be woken up by an one-cycle execution-latency instruction. In the proposed E model the average occupancy of wakeup matrix B is a 19% smaller than the average occupancy of wakeup matrix A. And for the model E-F, the occupancy of wakeup matrix B is, on average, a 34% smaller than the occupancy of wakeup matrix A.

In the SW model and in our proposed models, the empty entries of both wakeup matrices can be dynamically deactivated [1]. Then, our proposed models are more energy-efficient than SW model because the average occupancy of wakeup matrix B is smaller. Moreover, once the request signal of an instruction is detected by logic circuit M, its mirror entry in the other wakeup matrix can be deactivated.

VI. Related work

In order to reduce the scheduling latency, Palacharla et al. [16] proposed dispatching chains of dependent instructions into FIFO queues; the instructions considered to be issued are only the instructions heading each FIFO queue. Another works preschedule the instructions taking advantage of the fact that most instruction latencies are known at decode time ([5], [7], [14]). 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. All these techniques require estimating the issue cycle of instructions before inserting them in the buffer structure.

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 ([5], [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. Other proposal uses RAM bitmap arrays to identify all the successors of each instruction in the issue queue [9]. A new design that reduces the area cost for large issue queues was proposed by K. Hsiao and C. Chen in [10].

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 ([6], [9], [13]).

I.Kim and M. Lipasti proposed a hardware mechanism that dynamically detects dependent pairs of instructions and fuses them in order to be scheduled together [12]. So, the scheduling-loop latency (two cycles) is hidden because the scheduling granularity has been increased. A later work removes complexity from hardware and enables more sophisticated fusing heuristics using dynamic-translation software that becomes part of the processor design [11]. Other related works use intensive hardware to combine dependent operations [18] that are issued speculatively or need static compiler support [3].

Several works use speculation to break the scheduling loop. Stark et al. [20] proposed speculatively waking instructions up by their grandparents. This proposal allows pipelining the scheduling loop over two cycles. The speculative wakeup of an instruction is confirmed after their parents are selected. A false selected instruction affects performance only if it prevents really ready instructions from being selected for execution. Brown et al. [2] proposed a speculative technique, named Select-Free, which moves the select logic off the critical loop; this allows the scheduling loop to take just one cycle. The technique allows all woken-up instructions broadcasting the tags into the issue queue in the following cycle, even though some of them may have not been selected for execution yet. Contention for issue ports can produce the misspeculated wakeup of a chain of dependent instructions. Therefore, the availability of source operands of each issued instruction are checked before execution stage. Both proposals allow back-to-back scheduling of dependent instructions. Focusing in the SW model, we have measured that, on average, a *7.6%* of committed instructions are falsely selected, which unnecessarily utilize the select logic. In the SF model happens something similar. Measures show, on average, a *3.4%* of committed instructions must be re-scheduled. Obviously, re-scheduled instructions incur in an unnecessary utilization of both the select logic and the register file.

VII. Conclusions

In this paper, we have proposed two non-speculative enhancements (E and E-F) for a scheduling logic pipelined over two cycles. These enhancements try to tolerate the latency of the scheduling loop when there is not enough available ILP. Our proposals compute in advance which instructions will be woken up by all one-cycle execution-latency instructions that are currently competing for selection. This pre-computed group of instructions can compete for selection once all previous one-cycle execution-latency instructions, which are currently competing for selection, have been selected. Moreover, we have improved the base enhancement (E) using instruction fusing (E-F).

Our evaluations show that E and E-F models perform, on average, within a *2%* and *2.6%* of an ideal (unpipelined) scheduler, respectively. Compared to baseline model B (scheduling loop pipelined over two cycles), E and E-F increase performance , on average, a *7.9%* and a *8.6%*, respectively. Also, E and E-F perform, on average, slightly better than two previously-proposed speculative schedulers (Select Free and Speculative Wakeup).

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. The authors thank Gabriel Loh and the anonymous reviewers for their useful comments and suggestions.

References

[1] Y. Bai; R. I. Bahar. A dynamically reconfigurable mixed in-order/out-of-order issue queue for power-aware microprocessors, Proc. IEEE Computer Society Annual Symposium on VLSI, p. 139 - 146, Feb. 2003.

[2] M. Brown et al. Select-Free Instruction Scheduling Logic. Int. Symp. on Microarchitecture, 2001, p. 204-213.

[3] A. Bracy et al. Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. Int. Symp. on Microarchitecture, 2004, p. 18 - 29.

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

[5] R. Canal and A. González. A Low-Complexity Issue Logic. Int. Conf. on Supercomputing, 2000, p. 327-335.

[6] D. Ernst and T.M. Austin. Efficient dynamic scheduling through tag elimination. Int. Symp. on Computer Architecture, 2002, p. 37-46.

[7] D. Ernst et al. Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay. Int. l Symp. on Computer Architecture. Jun 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] M. Goshima et al.. A high-speed dynamic instruction scheduling scheme for superscalar processors. Int. Symp. on Microarchitecture. 2001, p. 225-236.

[10] 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.

[11] S. Hu et al. An Approach for Implementing Efficient Superscalar CISC Processors. Int. Symp. on High Performance Computer Architecture, Feb. 2006, p. 40-51.

[12] I. Kim and M. H. Lipasti, Macro-op Scheduling: Relaxing Scheduling Loop Constraints. Int. Symp. on Microarchitecture, 2003, p. 277-288.

[13] I. Kim and M. H. Lipasti. Half-Price Architecture. Int. Symp. on Computer Architecture. 2003, p. 28-38.

[14] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. Int. Symp. on High Performance Computer Architecture. 2001, p. 27-36.

[15] S. Önder and R. Gupta. Instruction Wake-up in Wide issue superscalars. European Conf. on Parallel Processing, 2001, p. 418-427.

[16] S.Palacharla et al. Quantifying the complexity of superscalar processors. T.R. University of Wisconsin-Madison. Nov 1996.

[17] E. Perelman et al.. Picking Statistically Valid and Early Simulation Points. Int. Conf. on Parallel Architectures and Compilation Techniques. 2003, p. 244-255.

[18] P. G. Sassone and D. Scott Wills. Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication. Int. Symp. on Microarchitecture, 2004, p. 7 - 17.

[19] T. Sherwood et al., "Automatically Characterizing Large Scale Program Behaviour," in Proc. of ASPLOS, Oct. 2002, p. 45-57.

[20] J. Stark et al. On pipelining dynamic instruction scheduling logic. Int. Symp. on Microarchitecture. 2000, p. 204-213.

[21] S. Weiss and J.E. Smith. Instruction issue logic in pipelined supercomputers. IEEE Transactions on Computers, 33: p.1013-1022, November 1984.