Reusing cached schedules in an out-of-order processor with in-order issue logic

Oscar Palomar*†, Toni Juan‡ and Juan J. Navarro*
** Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya.*
†Barcelona Supercomputing Center. ‡Intel® Corporation.
{*opaloma,juanjo*}*@ac.upc.edu, toni.juan@intel.com*

***Abstract*— The complex and powerful out-of-order issue logic dismisses the repetitive nature of the code, unlike what caches or branch predictors do. We show that 90% of the cycles, the group of instructions selected by the issue logic belongs to just 13% of the total different groups issued: the issue logic of an out-of-order processor is constantly re-discovering what it has already found. To benefit from the repetitive nature of instruction issue, we move the scheduling logic after the commit stage, out of the critical path of execution. The schedules created there are cached and reused to feed a simple in-order issue logic, that could result in a higher frequency design. We present the complete design of our ReLaSch processor, that achieves the same average IPC than a conventional out-of-order processor, and a 1.56 speed-up over the IPC of an in-order processor. We actually surpass the out-of-order IPC in 23 out of 40 SPEC benchmarks, mainly because the broader vision of the code after the commit stage allows creating better schedules.**

I. Introduction

The out-of-order processing logic allows to achieve high IPC but has serious impact in the achievable frequency. For throughput oriented and server workloads, simpler in-order processors that allow more cores per die and higher design frequencies are becoming the preferred choice (Power6 [1], Niagara 2 [2]). However, there are many workloads where it is still important to get good single thread performance. Our ReLaSch processor aims to enable high IPC cores capable of running at high clock frequencies by processing the instructions in an in-order issue logic and caching instruction groups that are dynamically scheduled out of the critical path and only when really needed.

A. Repeated issue in the out-of-order processors

Programs use functions and loops, so most of the time the out-of-order issue logic processes a limited amount of different instructions and many cycles it ends up issuing together the same group of independent instructions. Eventually, the schedule adapts to new situations such as a cache miss.

The issue logic is designed to extract as much ILP as possible each cycle, ignoring the parallelism found before. It does not benefit from the repetitive behavior of code, unlike other processor elements, i.e. caches or branch predictors, that indeed rely on this characteristic of the programs to perform as expected.

To show that the issue logic repeats most of its work, we have captured the *issue-groups* (an issue-group is the set of instructions issued in the same cycle) created by an out-of-order processor on 100-Minstruction simulations of each SPECcpu2000 benchmark. Experimental environment is detailed in section IV. Fig. 1 shows the percentage of different issue-groups that add up to 90% of the cycles. From this data, a 90/10-like rule of thumb can be postulated: 13% of the issue-groups appear 90% of the cycles. Regrettably, the issue logic is constantly creating the same issue-groups in the critical path of execution.

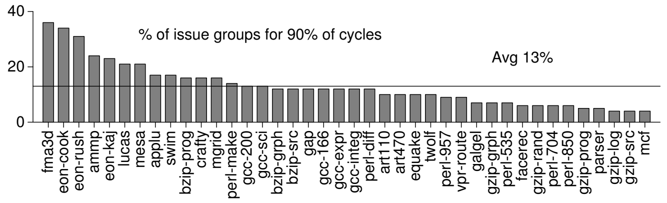

Fig. 1. Amount of repeated work done by the conventional out-of-order issue logic.

B. Proposal: in-order issue and post-commit scheduling

We present a complete processor, named *ReLaSch* after Reused Late Schedules, in which the creation of issue-groups is removed from the critical path of execution. ReLaSch uses a simple and small in-order issue logic, because it just wakes-up and selects the instructions of a single issue-group each cycle instead of a whole issue window, which is much larger than one issue-group.

A new logic at the end of the conventional pipeline schedules the committed instructions into *rgroups* (which are sequences of issue-groups). R means Reuse, as in other elements of ReLaSch that are prefixed with an R.

The rgroups are stored in a cache. Whenever is possible, an rgroup is read and its instructions executed: the schedules are reused, lowering the pressure on the scheduling logic.

The conventional out-of-order processors use branch prediction and memory aliasing speculation to find more available instructions. Besides, their issue logic adapts to variable latency instructions. The ReLaSch processor predicts the branch targets, memory aliases and latencies at schedule time, out of the critical path. Average branch misprediction rate is higher in ReLaSch than that of a conventional branch predictor, though ReLaSch predicts better in some cases. A conventional branch predictor is used when no rgroup is found the cache.

The out-of-order issue logic is able to react immediately to changes in code behavior. ReLaSch relies on the repetitive nature of code, so on a change the schedule mispredicts or stalls due to an unexpected latency. To reduce the wasted cycles, it replaces the rgroups that repeatedly fail.

The ReLaSch processor takes some techniques and elements from the conventional out-of-order processors. It has a Reorder Buffer (ROB) and a Commit stage to retire the instructions in-order and provide precise interruptions. It also allows to commit the correct part of an rgroup when it includes a misprediction. Besides, the registers are renamed to eliminate false dependences. Since the instructions in an issue-group are independent by construction, the renaming logic can process all the instructions in the issue-group in parallel, so it is simpler than in a traditional out-of-order processor, that must detect the dependences between the instructions renamed in the same cycle.

Our ReLaSch processor is able to outperform the IPC of a conventional out-of-order processor, because the post-commit scheduler has a broader vision of the code. Our scheduler can place two independent but distant instructions in the same issue-group when it creates the schedule. Nevertheless, a conventional out-of-order processor fetches the instructions and inserts them in the issue-window in-order and for instructions that are distant in the code, the fetch, decode and rename width prevent them to be present at the same time in the issue window.

There is previous work that proposes moving the scheduling logic out of the critical path and/or try to reuse the schedules [3], [4], [5], [6], [7]. In section VI the main differences and similarities are highlighted.

C. Summary of results and scope

Our experiments (see details in section V) show that ReLaSch achieves the same average IPC as our reference out-of-order processor and is clearly better than the reference in-order processor (1.56 IPC speed-up). In all cases it outperforms the in-order processor, and in 23 SPEC benchmarks out of 40 it outperforms the out-of-order processor.

The results presented in this paper assume that all the processors use the same frequency and just evaluate IPC. However, it would be reasonable to assume that the in-order processor and our proposed ReLaSch processor can achieve a higher frequency.[1] This would translate into a performance speed-up over the out-of-order processor higher than the IPC speed-up shown here. However, in this paper we do not evaluate cycle time but describe the microarchitecture and compare the IPC.

Also, note that in our approach the power-hungry scheduling logic of a conventional OoO processor, which is active all the time, is replaced by a scheduling logic that is out of the critical path that is only active a small fraction of the time (30% of the cycles in average). In the out-of-order Alpha 21264, the issue units use 18% of the power budget,

[1]Using a technology independent model, the in-order Power6 processor doubles the frequency of the out-of-order Power5 [1].

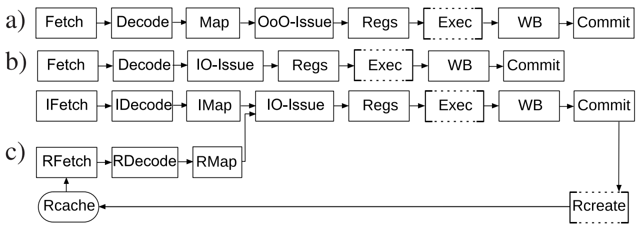

Fig. 2. The pipeline of the (a) OoO, (b) IO and (c) ReLaSch processors.

only surpassed by the clock network (32%) [8]. However, in this paper we do not measure power.

II. General description

Fig. 2.a shows the pipeline of an improved out-of-order processor (OoO) based on the 21264 Alpha [9], enhanced with better memory alias detection and branch target prediction. The Fetch stage reads the instructions from the Icache and accesses the branch predictor. The next stage completes the branch prediction and decodes the instructions. The Map stage renames the registers and inserts the instructions in the ROB and the issue window. The Issue stage wakes-up and selects the available instructions in the issue window, up to 4 integer and 2 floating point. The registers are read in the next stage, and execution happens in the corresponding functional unit afterwards. The registers are written and the Dcache accessed in the WriteBack stage. Finally, instructions are retired in-order in the Commit stage.

Fig. 2.b shows the pipeline of the in-order processor (IO) we use as reference. Although based on the 21264 Alpha, it has an in-order Issue stage. It also does not rename the instructions nor uses a ROB, so the Map stage is eliminated. The Decode stage simply puts the instructions in a buffer (the issue buffer).

The pipeline of the ReLaSch processor is shown in fig. 2.c. The Fetch, Decode and Map stages of the OoO processor form the Ifront-end and are prefixed with an I in the figure. They are coupled with additional logic to process the rgroups: Rfetch, Rdecode and Rmap (that forms the Rfront-end). Also, there is a new sequence of stages after the Commit stage: the Rcreate logic. Besides, the Rcache is added to the processor.

Fig. 3 shows a diagram of the main blocks of the ReLaSch processor, including the execution pipeline, the different elements of the Rcreate logic and the caches. The diagram also indicates the number of entries of the tables and queues, and the size of the different caches used in our experiments (see section IV).

The Rcreate logic schedules the committed instructions into rgroups. Our baseline uses 256-instruction rgroups, with issue-groups of 4 integer and 2 floating point instructions. The proposed scheduling is a simple list-based one. However, a complex one could be used if it improves performance. Rcreate also makes predictions based on the last execution and partially renames the registers. The resulting rgroups are stored in the Rcache. The Rfetch logic accesses the Rcache and the Rdecode logic decodes the instructions. However,

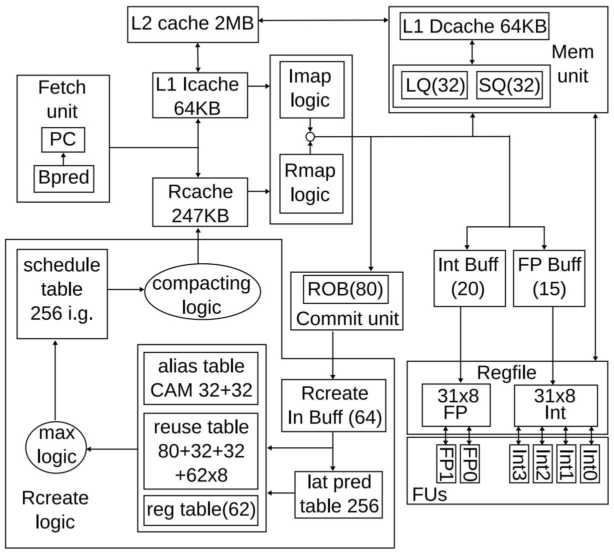

Fig. 3. Block diagram of the ReLaSch processor.

they don't access the branch predictor. The Rmap logic completes the renaming and inserts the instructions in the ROB and the issue buffer.

Unlike the 21264 Alpha and the OoO processors, that use a shared pool of physical registers, ReLaSch uses a register file that has a fixed set of physical registers for each logical register, based on the Flywheel's register file [5]. The registers within a set are always assigned in the same order, which eases the renaming process. We use the Flywheel register file because a ReLaSch processor modified to use a conventional register file has shown: a) to yield a slightly lower IPC than the baseline ReLaSch; b) that its Rmap renaming logic and Rcreate stages are more complex; and c) the Rcache requires storing more bits per instruction.

A. Execution modes

ReLaSch has two execution modes: the Icache mode, when the Ifront-end is used; and the Rcache mode, when the scheduled instructions are fetched from the Rcache and are processed by the Rfront-end.

When the processor is in the Icache mode, each cycle it accesses with the PC the Icache and the Rcache in parallel. On an Rcache hit (an rgroup begins with this instruction), the processor changes to the Rcache mode. After an rgroup has been completely fetched, another one is read from the Rcache if possible. Otherwise, the processor changes to the Icache mode. The identifier of the next rgroup is stored with the current rgroup in the Rcache.

Regardless of the mode, when the pipeline is flushed (on a branch misprediction or a memory order violation), the Rcache is accessed with the recovery PC. On a hit the processor executes the instructions in the Rcache mode, and in the Icache mode otherwise.

Besides, the Rcreate logic can be in one of two modes: in the Schedule mode or in the Idle mode. It creates new rgroups only in the Schedule mode. The instructions executed in the Icache mode are always processed in the Schedule mode. On a change to the Rcache mode, Rcreate changes to the Idle mode, but first it completes the current rgroup. It changes back to the Schedule mode: a) when it finds an instruction executed in the Icache mode; and b) to re-schedule Rcache-mode instructions of an rgroup that frequently aborts its execution. Each rgroup in the Rcache has a saturating counter (5 bits) to detect this kind of rgroups.

B. ReLaSch processor operation principle

In the Schedule mode, the Rcreate logic places each instruction in the issue-group in which its source registers will be available and assigns its destination physical register. During execution, the Rmap logic needs to adjust this renaming to the actual registers used by the previous rgroup. So Rmap will complete the renaming, by simply adding an offset to the identifier of the physical registers. Besides, Rmap checks if the destination physical register is free, while the Issue stage checks if the source physical registers are ready.

Rcreate assigns to each instruction its identifier in the ROB and the LQ (Load Queue) or the SQ (Store Queue). The Rmap logic adds an offset to these identifiers and checks their availability; it also inserts the instructions in-order in the issue buffer. Rcreate tracks in which issue-group each ROB entry can be reused to avoid the possible deadlocks, as shown in section III-A.1. The Issue stage checks if the functional unit needed for each instruction is available, while Rcreate tracks the usage of the functional units in the issue-groups of the schedule.

Rcreate uses saturating counters to predict the latency of the memory instructions, using either the L1 hit latency or the latency of the last execution. The last addresses accessed are used to predict whether the memory instructions will alias and to schedule them accordingly. If a predicted aliased store-load pair cannot forward the data from the SQ (due to different access-size), the load saves a copy of the store's identifier. Rmap adds the offset to this identifier, and the Issue stage blocks the load until the store actually commits. The Writeback stage checks the memory order violations, and the Commit stage replays the offending instruction.

In Rcreate, each dynamic branch scheduled in an rgroup is predicted to repeat its outcome, although an rgroup can contain several copies of the same static branch, each one predicted independently. This prediction implicitly uses the whole rgroup as path, so it can capture complex patterns. The Commit stage checks that the prediction is correct. Our experiments show that 8% of all dynamic branches were predicted by Rcreate to change from their previous behavior in the schedule. 84% of them are predicted correctly, while a PC-indexed 1-bit predictor would have mispredicted all them.

III. Implementation details

A. The Rcreate logic

The Rcreate logic uses a table to schedule the instructions. Each entry corresponds to one issue-group. The logic schedules the instructions in the earliest possible issue-group, according to the dependences, the latencies of the FUs and

Code:

```
A: MUL R5, R4, R3; ROB[0]
B: ADD R3, R2, R1; ROB[1]
C: ADD R4, R2, R6; ROB[2]
D: ADD R6, R7, R7; ROB[3]
E: ADD R1, R8, R8; ROB[0]
F: ADD R7, R2, R2; ROB[1]
G: ADD R4, R5, R5; ROB[2]
```

Rgroup
(deadlock)

```
0: A C
1: D G
2: -
3: B
4: E F
```

Rgroup (ok)

```
0: A C
1: D
2: -
3: B
4: E F
5: G
```

ROB-safe:

id:	0	1	2	3
safe:	1	4	4	0

Fig. 4. Example of a deadlock using a 4-entry ROB and 2-instruction issue-width. Latencies: MUL 3 cycles, ADD 1 cycle. From left to right, and top to bottom: the original code, the rgroup with a deadlock, the ROB-safe table after C is scheduled, and the correctly scheduled rgroup.

the availability of the resources. The Rcreate logic must deal with the following problems:

1) Deadlocks: When creating an rgroup, Rcreate assigns in order the identifiers in the ROB, beginning with identifier 0. But, at execution time instructions are inserted in the ROB out-of-order by the Rmap logic, and committed in-order. A careless schedule, such as that in fig. 4, creates a deadlock: when Rmap processes the issue-group 0, both instructions A and C occupy the ROB identifiers 0 and 2. The next cycle, D is Rmapped, but G stalls since its ROB identifier is still in use by C. The stall prevents B to execute, so the identifier is never freed since C cannot commit before B.

To prevent deadlocks, Rcreate records the safe issue-group for all the resources it assigns (physical registers and ROB, LQ and SQ identifiers). The safe issue-group, in which a resource can be reused, is the highest issue-group where an instruction is scheduled just after the assignment of the resource. In the example, after the ROB identifier 2 is assigned to C, it can be reused in the issue-group 3, where B is scheduled. Actually, the number of cycles until B commits is added, to avoid stalling the instruction that reuses the resource. Just one cycle is added in the example, so G should be scheduled in the issue-group 4; since it is full, G is placed in the issue-group 5.

2) Registers and dependences: Rcreate uses the register-info table to record: a) the last physical register assigned to each logical register, b) in which issue-group is the data available and c) the safe issue-group of the physical registers. Rx.y denotes the physical register y of the logical register x. A source register Rx not yet scheduled as destination within the rgroup reads the physical register Rx.0. The example in fig. 5 shows how the table is used to schedule an instruction: the logical registers are renamed to R3.7, R2.3 and R1.0. The instruction is scheduled in the issue-group 17, which is the maximum of 15 (Read), 17 (Read) and 10 (Safe[Phy+1]). The destination register's entry is updated after the instruction is scheduled. The example assumes there is a previously scheduled instruction in the issue-group 19, so the new safe[Phy] is 20.

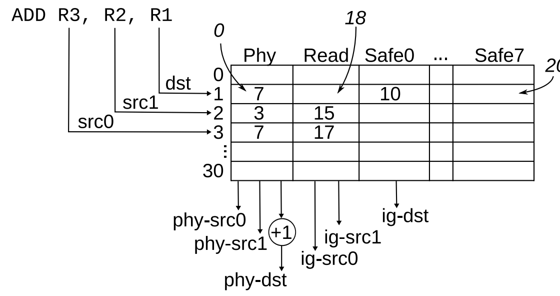

Fig. 5. An example of use of the register-info table. It provides the physical registers (phy) and the issue-groups (ig). The example assumes a register file with 8 physical registers per each logical register.

3) Closing rules: An rgroup is closed if: a) it is full (256 instructions is the maximum in our experiments); b) the issue-group for an instruction is beyond the schedule table (512 in our experiments); c) the table of indirect branch targets is full (10 in our experiments); or d) a system call instruction is scheduled. Then the compacting logic reads the rgroup from the schedule table, skipping the empty issue-groups and slots, placing all the instructions contiguously, in the Rcache format.

4) Branch prediction: The Rcreate logic always schedules from the stream of committed instructions: it predicts that the branches will repeat the behavior of its last execution. The instructions of any other path are not available. An option would be to close the rgroup after the branch, creating shorter rgroups, but this results in lower IPC.

To detect during execution the mispredicted branches, the conditional branches record a taken/not-taken bit, and the indirect branches store the predicted target PC (separated from the instructions). Patterns in multi-target indirect branches can also be captured with more precision than a conventional BTB does since history is implicitly taken into account. The reference OoO processor uses an enhanced BTB to cope with multi-target indirect branches.

5) Latency of the memory accesses: The scheduler predicts a latency for each load instruction to schedule the dependent instructions. If the load is predicted to hit and it actually misses, the first dependent instruction stalls at Issue. If a miss is predicted and at execution the L1 cache hits, no instruction stalls, but the dependent instructions execute later than necessary, decreasing the IPC.

Rcreate tries to detect the biased loads, with a PC-indexed table of 1-bit saturating counters. Counters are increased on an L1 hit, and decreased on an L1 miss. If the counter is set, the scheduler uses the L1-hit latency, and the latency of the last execution otherwise. Loads that frequently miss in the L2 cache are more likely to benefit from re-scheduling, so when a load executed in the Rcache mode misses in the L2, the rgroup's counter in the Rcache is decremented.

6) Memory aliasing prediction: The last addresses accessed are used to predict whether two memory instructions will alias. Unaliased instructions are freely reordered. Two aliased loads cannot be reordered at execution, so they are scheduled in the same issue-group. An aliased store-

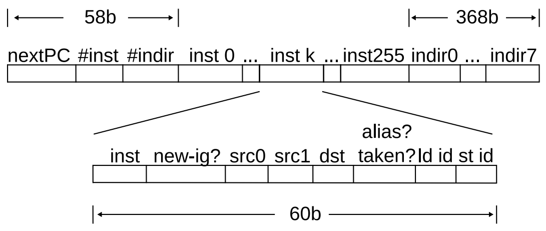

Fig. 6. The information of an rgroup stored in an Rcache line.

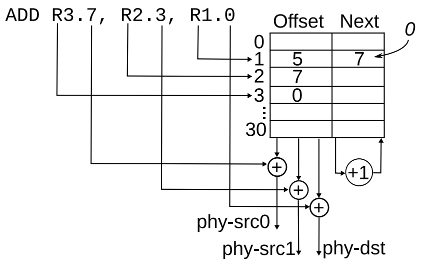

Fig. 7. An example of how the Rmap logic completes the renaming.

load pair is scheduled in consecutive issue-groups to allow data forwarding. When forwarding is impossible, the load is scheduled in the issue-group where the store commits, and it records the store's identifier in the SQ. The store instructions access the memory in-order and can be scheduled anywhere.

The scheduler remembers the addresses of the last scheduled memory instructions, that can be in the LQ and SQ when the current load is executed. The safe issue-group is retrieved with a CAM-access with the load's address, masking the lower bits according to the size of the access. This method eliminates most load replays.

B. The Rcache

The Rcache stores one rgroup per line. Fig. 6 details the information stored per rgroup, divided into control information, indirect branch targets and instructions. It uses 60 bits per instruction and 46 per indirect target. With 8-target 256-instruction rgroups, the total adds up 1,974 bytes per rgroup, including 58 control bits.

Rcache lines are longer than usual cache lines. However, a line is processed sequentially, one issue-group per cycle, and there can be no random access in the middle of the line, so a low-bandwidth sequential access fetches the content.

The Rcache is indexed with the lower bits of the PC, the rest of the PC bits are the tag, matched with the PC of the first instruction of each rgroup in the Rcache set.

Each Rcache line has a saturating counter, increased if all the instructions in the rgroup commit, and decreased if the execution aborts or on a L2 miss. When an rgroup is fetched and its counter is 0, the rgroup's instructions are marked, and later re-scheduled in Rcreate.

Like the Trace cache [10], the Rcache usually contains duplicated instructions, i.e. several instances of a loop dynamically unrolled by the scheduler. Though not desirable, it is needed to create good rgroups.

C. The Rmap logic

To complete the renaming, an offset is added to the physical registers' identifiers. The offset of a logical register is the physical register of its last use as destination before the rgroup. The offsets are constant during the execution of an rgroup. To rename the instructions executed after an rgroup, the Rmap logic counts the uses as destination of each logical register (modulo the number of physical registers). Fig. 7 shows an example of an instruction renamed by Rmap. The final registers are R3.3, R2.2, R1.7. The Next counter of R1 is incremented (modulo 8).

Similarly, there is an ROB-offset, an LQ-offset and an SQ-offset, that are added to the corresponding identifiers. The offsets are updated when an rgroup begins execution, from the first free identifier in the ROB and the queues.

The logic checks, for each instruction, that its destination physical register and its identifier in the ROB, the LQ and the SQ are free. The instruction is inserted in the issue buffer.

The Rmap logic is similar in complexity to the Map stage of a conventional OoO processor, that also access a table indexed with the logical registers. The Rmap logic has also the offset-adders, but it removes the feedback loop.

D. The Issue stage

The ReLaSch processor uses two issue buffers: int and fp. Float-stores and float-to-integer instructions use both buffers, while loads and integer-to-float only use the int buffer.

The Issue stage processes each buffer in-order but independently, unless an instruction is present in both buffers. Only the first issue-width instructions of each buffer are processed. For each instruction, the issue logic checks that: a) the source registers are ready; b) the FU is available; c) the new-ig flag, that indicates the beginning of a new issue-group is cleared (except for the older instruction in the buffer); d) the aliased store has committed (only for some loads) and e) all older instructions in the buffer have been issued. When an instruction is present in the two buffers, both entries must be ready to issue it.

The new-ig flag is checked to avoid issuing together instructions of different issue-groups. When they are independent it is correct to issue them in parallel, improving IPC. However, this allows that an aliased store-load pair that is scheduled in consecutive issue-groups is issued simultaneously. In this case the data cannot be forwarded, so the load is replayed. Since our experiments show that the increase in load replays overweights the IPC gain, the issue-group boundaries are respected.

IV. Experimental set-up

We have modified the sim-alpha simulator [11] to model the ReLaSch and the reference OoO and IO processors. Sim-alpha is based on Simplescalar and was configured and validated against a real Alpha machine. Our reference IO and OoO processors are much like an Alpha 21264 [9], enhanced with Store Sets [12] and an improved BTB, similar to the Intel Pentium M processor's target predictor [13], using path instead of history since it works better with our benchmarks

TABLE I

Main simulation parameters for OoO, IO and ReLaSch.

	OoO	IO	ReLaSch
Issue width: 4 Int, 2 FP	*	*	*
Issue queue: 20 Int, 15 FP	*		
ROB: 80, Ld: 32, St: 32	*		*
FUs: 4 ialu, 4 imul, 1 fpalu, 1 fpmul	*	*	*
DL1 & IL1: 2-way 64KB 3-cycle hit lat	*	*	*
L2: 2MB 13-cycle hit, 84-cycle miss lat (extra cycles if bus contention)	*	*	*
DTLB: 128 entries, ITLB 128 ent.	*	*	*
Bpred: 4Kx2 choice, 4Kx2 global vs 2-level local (1Kx10 hist, 1Kx3 count.)	*	*	*
BTB: 1024-ent. 4-way pc-indexed, 32-ent. RAS	*	*	*
multi-target BTB: 1024-ent. path-indexed	*	*	
Store Sets: 4K-ent. SSIT 128-ent. LFST, 7-bit id	*	*	
StWait: 1024 1-bit table			*
41 int, 41 fp shared physical registers + 31 int, 31 fp arch. registers	*		
8 physical reg. per logical reg.			*
Rgroup: 256 inst, 8 indirect branches			*
Rcreate: 1 inst per cycle, 256 1-bit load lat pred schedule table of 512 issue-groups			*
Rcache: 4 ways, 32 sets, 1974B per rgroup 5-bit counter per line read 1st issue-group: 3 cycles-lat 1 issue-group/cycle afterwards			*

[14]. The ReLaSch processor doesn't require the Store Sets and the improved BTB, since the rgroups already solve the same problems. So ReLaSch uses the simpler original Alpha BTB and StWait bits.

Table I shows the main simulation parameters. Any other parameter maintains the default sim-alpha/21264 value [11].

We use most of the SPECcpu2000 benchmarks, the 8 missing ones had compilation problems. The benchmarks were compiled with -O3 or -O4. For each benchmark, a 100M-instruction segment is simulated. SimPoint [15] was used to find the most representative segment of each benchmark. The simulator is not trace-driven, but fetches the instructions from the binary, even after mispredicted branches. Average IPC stands for harmonic mean.

V. Experimental results

Fig. 8 shows the speed-up of the IPC obtained by the ReLaSch processor over the OoO processor, ordered by increasing speed-up and separating the FP and the INT benchmarks. Each bar starts at 1.0 speed-up. Speed-up lower than 1.0 indicates a performance loss.

The ReLaSch IPC is higher than the OoO IPC in 18 out of 28 INT benchmarks and in 5 out of 12 FP benchmarks. It has a 0.99 speed-up over the average INT OoO IPC, a 1.02 over FP and has the same average IPC when both FP and INT benchmarks are considered.

Fig. 9 shows the speed-up of ReLaSch over the IO processor. ReLaSch IPC is much higher than IO in all cases. It has a 1.51 speed-up over the average INT IO IPC, a 1.64 over FP and a 1.56 overall (INT and FP).

To outperform the OoO processor, several conditions are needed: a) that the scheduler frequently finds independent instructions that are distant in the code (instructions that usually the out-of-order issue logic doesn't schedule together); b) execute most of the instructions in Rcache mode; and c) have a low misprediction rate.

The first condition depends on each benchmark; the second is true for all the evaluated benchmarks: the average rate of committed instructions executed in Rcache mode is 94.3% (INT) and 99.3% (FP); the lowest is 82% (*gcc-sci* and *crafty*).

Branch prediction of ReLaSch can be more accurate than in the OoO processor: e.g. with a high number of branches in an rgroup, it can capture patterns that are longer than the history length of the 21264. However, branches correlated with local history are perfectly predicted by the baseline tournament predictor, but if the history is longer than the schedule, the rgroups misses them.

A. *Rcache size*

The branch misprediction rate is closely related to the number of rgroups that can be stored in the Rcache. Highly regular benchmarks just create a few rgroups that are usually completely executed, and fit in a small Rcache, but benchmarks that follow many different paths create more rgroups. Table II shows how the misprediction rate decreases when the Rcache size grows.

Table III summarizes the average speed-up for several Rcache sizes. Most FP benchmarks (and some INT benchmarks) are very regular and, creating few rgroups that fit in a small Rcache, they reach their top IPC with ReLaSch. On the other hand, many INT benchmarks create more rgroups and improve a lot with bigger Rcaches. However, some benchmarks with branches that are harder to predict using rgroups (*bzip-src* has a 16% miss-rate in ReLaSch vs. a 10% miss-rate in OoO) do not benefit from bigger Rcaches. The ReLaSch processor with a 4-4 Rcache (that stores 16 256-instruction rgroups and occupies 31KB) has a 1.33 (INT) and a 1.58 (FP) speed-up over the average IO IPC, and 0.87 (INT) and 0.98 (FP) over the OoO IPC. The actual optimal Rcache size depends also on the power consumption constraints, which are not evaluated in this paper.

TABLE II

Average branch misprediction rate (in %).

		sets-ways						
	OoO	4-4	8-4	16-4	32-4	64-4	128-4	256-4
INT	5.05	7.49	7.34	6.97	6.6	6.26	5.97	5.78
FP	1.70	3.54	3.17	2.88	2.45	2.33	2.18	2.08
ALL	4.00	6.24	6.02	5.67	5.27	5.00	4.74	4.58

TABLE III

Average speed-up of ReLaSch over the OoO processor for different Rcache sizes.

sets	4	8	16	32	64	128	256
FP	0.98	1.00	1.01	1.02	1.02	1.02	1.02
INT	0.87	0.91	0.96	0.99	1.01	1.03	1.04
ALL	0.90	0.94	0.97	1.00	1.02	1.03	1.04

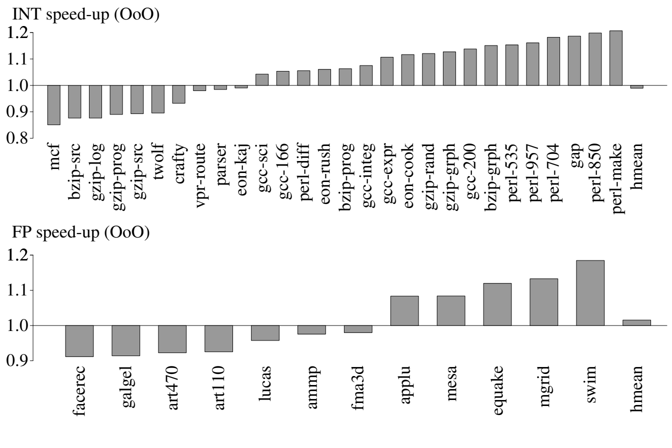

Fig. 8. IPC speed-up of the ReLaSch processor over the OoO processor.

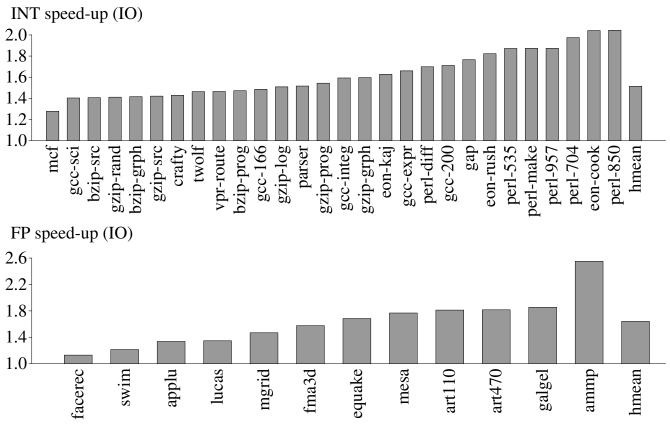

Fig. 9. IPC speed-up of the ReLaSch processor over the IO processor.

TABLE IV
AVERAGE SPEED-UP OF RELASCH OVER THE OOO PROCESSOR FOR DIFFERENT RGROUP SIZES.

inst-sets	64-128	128-64	256-32	512-16	1024-8
FP	0.90	0.97	1.02	1.03	1.04
INT	0.92	0.98	0.99	0.97	0.92
ALL	0.92	0.97	1.00	0.99	0.96

B. *Instructions per rgroup*

With more instructions per rgroup the scheduler is able to extract more ILP. But for a given Rcache size, doubling the rgroup size halves the number of rgroups, which lowers the IPC of benchmarks that create more rgroups. Table IV shows the ReLaSch average speed-up using several rgroup sizes with a 247KB Rcache.

In general, FP benchmarks benefit from larger rgroups while INT need more rgroups. With 64-instruction rgroups, the scheduler's reordering capability is too limited (less than the ROB size) and the reduced ILP is not compensated by the higher number of rgroups available: all the benchmarks perform worse than with 128-instruction rgroups. The baseline 256/32-configuration maximizes the overall performance.

C. *Rcreate stages*

All the results presented in this section assume that Rcreate is pipelined in 10 stages to schedule an instruction. Our experiments show that using more stages (i.e. 20) does not reduce performance.

VI. RELATED WORK

There are other proposed processors that schedule instructions outside the critical path of execution, and cache the schedules to feed the pipeline later. We present the differences in the approach and the objectives of those more similar to our work, followed by other related work that doesn't cache the schedules.

A. *Caching proposals*

DIF [3] creates VLIW schedules from RISC instructions. It has two cores: VLIW and in-order. It schedules instructions committed in the in-order core to feed the VLIW core. The in-order core is used when there isn't an available VLIW schedule. Each logical register has a fixed set of physical registers, but each physical register can be written just once in a schedule. Although this leads to use smaller schedules, write-once registers are combined with atomic committing of a schedule (either none or all its instructions commit), thus simplifying the commit logic.

rePLay [4] is mainly focused on trace optimization to reduce execution time. It has a conventional out-of-order pipeline and an optimizer of the committed instructions. Traces commit atomically to allow more aggressive optimizations, requiring that useful instructions are re-executed after a mispredicted branch. The out-of-order pipeline schedules and renames the instructions. However, there is an in-order rePLay [16], where the instructions in a trace are scheduled and renamed after optimization, and executed in an in-order pipeline. It uses a conventional register file, and a trace needs to record its live-in and live-out registers.

Flywheel [5], [17] reduces the energy consumed by the processor's front-end, by capturing what the Issue logic produces and caching the resulting traces. It has a conventional out-of-order pipeline. The cache substitutes the processor's front-end, that is switched off to save energy. The ReLaSch register file is based on the Flywheel proposal. The traces commit gradually, and their size varies depending on the branch misprediction rate. It has an out-of-order pipeline, so its performance usually degrades when the traces are used.

CTS [6] uses an in-order pipeline and schedules committed instructions. It is focused on dynamically analyzing the code to optimize it, using loop-unrolling and software pipelining. CTS caches only a subset of the schedules, that are stored in dedicated virtual memory pages. It filters the traces to schedule only the hottest parts of the code, so less instructions are scheduled, but it lowers pressure on the cache and the scheduler. CTS commits each loop iteration atomically, so it only needs to re-execute useful instructions on a load replay. It has several register windows, each one linked to a specific loop iteration and with a register for each logical register. The control instructions are not reordered, to ease code analysis and register window identification.

The Incremental Commit Groups proposal [7] improves a Transmeta-like processor [18] by allowing partial commit of the traces. It translates instructions to another language and does not compare with a conventional OoO processor.

Among this proposals, only the in-order rePLay has an in-order issue-logic and compares its performance with a conventional out-of-order processor. Although it doesn't outperform the out-of-order IPC in any benchmark, it is close in some cases. CTS, that also has an in-order issue logic, yields 1.16 speed-up over an in-order conventional processor.

B. Non-caching proposals

Other proposals [19] [20] [21] don't cache schedules but reuse the instructions directly from the issue window or the ROB, to save energy in the front-end. They benefit from small loops that fit in these structures.

Prescheduling [22] uses a Preschedule window formed by lines, consumed in-order to feed an out-of-order issue logic. It allows to reduce the size of the issue window without degrading IPC. The preschedule logic processes the decoded instructions at execution time. It just takes into account the data dependences and does not deal with renaming or resource assignment, since the out-of-order issue logic performs all these tasks. The preschedule logic assumes an L1-hit latency for all the load instructions.

Cyclone [23] is a different way to simplify the OoO issue logic. It uses a simple in-order issue logic combined with latency prediction, replaying the instructions when the desired operand is not available. However, it is not able to achieve better IPC than a conventional OoO processor.

Runahead [24] proposes to increase the performance of an in-order processor by pre-executing instructions on a cache miss. We consider that this technique is orthogonal to our proposal and could be implemented in it.

VII. Conclusions and future work

We have shown that the out-of-order issue logic is constantly creating the same schedules, and that its complexity and capacity to find a high level of ILP is only seldom exploited. With the proposal of the ReLaSch processor, we have also shown that the scheduling-logic can be moved after the commit stage, out of the critical path, and achieve similar IPC than a conventional out-of-order processor with a simple in-order issue-logic and a cache of schedules. Placed after commit, the latency of the scheduler does not affect the performance of the processor and the scheduler has a broader vision of the code, sometimes finding more ILP than the out-of-order issue logic. Besides, the schedules capture the behavior of most control and memory instructions.

Our experiments show that, using a cache of 128 schedules (256 instructions each), we achieve an IPC similar to that of a conventional out-of-order processor: 0.99 INT and 1.02 FP speed-up, outperforming the out-of-order processor in many cases. Compared with a conventional in-order processor, ReLaSch offers a 1.52 INT and 1.65 FP speed-up.

There is room to continue improving the scheduler and create better schedules that would yield in higher IPC. For instance, the addresses accessed by the memory instructions can be used to predict their latency. On the other hand, the good IPC results obtained by ReLaSch encourage us to perform a detailed analysis of frequency and power.

VIII. Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN2007-60625. The authors would like to thank José R. Herrero for his useful revision of the draft.

References

[1] H. Le *et al.*, "IBM POWER6 microarchitecture," *IBM J. Res. & Dev.*, vol. 51, no. 6, pp. 639–662, November 2007.
[2] H. McGhan, "Niagara 2 opens the floodgates," *MPR*, November 2006.
[3] R. Nair and M. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proc. of the 24th ISCA*, 1997, pp. 13–25.
[4] S. Patel and S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Trans. on Computers*, vol. 50, no. 6, pp. 590–608, jun 2001.
[5] E. Talpes and D. Marculescu, "Execution cache-based microarchitecture for power-efficient superscalar processors." *IEEE Trans. VLSI Syst.*, vol. 13, no. 1, pp. 14–26, 2005.
[6] S. Narayanasamy *et al.*, "Creating converged trace schedules using string matching," in *Proc. of the 10th HPCA*, 2004, pp. 210–221.
[7] M. Yourst and K. Ghose, "Incremental commit groups for non-atomic trace processing," in *Proc. of the 38th MICRO*, 2005, pp. 67–80.
[8] M. Gowan, L. Biro, and D. Jackson, "Power considerations in the design of the alpha 21264 microprocessor," in *Proc. of the 35th Design Automaton Conf.*, 1998, pp. 726–731.
[9] *Compiler writer's guide for the Alpha 21264*, Compaq Computer Corporation, June 1999, eC-RJ66A-TE.
[10] E. Rotenberg *et al.*, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proc. of the 29th MICRO*, 1996, pp. 24–35.
[11] "Sim-alpha: a validated, execution-driven Alpha 21264 simulator," Dep. of Computer Sciences, UT-Austin, Tech. Rep. TR-01-23, 2001.
[12] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proc. of the 25th ISCA*, 1998, pp. 142–153.
[13] S. Gochman *et al.*, "The Intel Pentium M processor: Microarchitecture and performance," *Intel Technology Journal*, vol. 7, no. 2, May 2003.
[14] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *Proc. of the 24th ISCA*, 1997, pp. 274–283.
[15] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *Proc. of the 10th ASPLOS*, 2002, pp. 45–57.
[16] F. Spadini *et al.*, "Improving quasi-dynamic schedules through region slip," *Proc. of the 1st CGO*, pp. 149–158, 2003.
[17] E. Talpes and D. Marculescu, "Increased scalability and power efficiency by using multiple speed pipelines." in *Proc. of the 32nd ISCA*, 2005, pp. 310–321.
[18] J. C. Dehnert *et al.*, "The transmeta code morphing^{TM}software: Using speculation, recovery and adaptive retranslation to address real-life challenges," in *Proc. of the 1st CGO*, 2003, pp. 15–24.
[19] J. Hu *et al.*, "Scheduling reusable instructions for power reduction," in *Proc. of the DATE Conf.*, 2004, pp. 148–153.
[20] C. Yang and A. Orailoglu, "Power-efficient instruction delivery through trace reuse," in *Proc. of the 15th PACT*, 2006, pp. 192–201.
[21] F. Pratas *et al.*, "Low power microarchitecture with instruction reuse," in *Proc. of the Conf. on Computing Frontiers*, 2008, pp. 149–158.
[22] P. Michaud and A. Seznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," in *Proc. of the 7th HPCA*, 2001, pp. 27–36.
[23] D. Ernst, A. Hamel, and T. Austin, "Cyclone: A broadcast-free dynamic instruction scheduler with selective replay," in *Proc. of the 30th ISCA*, 2003, pp. 253–262.
[24] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proc. of the 11th Intl. Conf. on Supercomputing*, 1997, pp. 68–75.