Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors

Pierre Michaud, André Seznec
IRISA/INRIA
Campus de Beaulieu, 35042 Rennes Cedex, France
{pmichaud, seznec}@irisa.fr

Abstract

The performance of out-of-order processors increases with the instruction window size. In conventional processors, the effective instruction window cannot be larger than the issue buffer. Determining which instructions from the issue buffer can be launched to the execution units is a time-critical operation which complexity increases with the issue buffer size. We propose to relieve the issue stage by reordering instructions before they enter the issue buffer. This study introduces the general principle of data-flow prescheduling. Then we describe a possible implementation. Our preliminary results show that data-flow prescheduling makes it possible to enlarge the effective instruction window while keeping the issue buffer small.

1. Introduction

Processor performance is strongly correlated with the clock cycle. Shorter clock cycle has been allowed by both improvements in silicon technology and careful processor design.

As a consequence of this evolution, the IPC (average number of instructions committed per clock cycle) of future processors may decrease rather than increase [1]. This IPC decay comes from the dispersion of instruction latencies. In particular, load latencies in CPU cycles tend to increase across technology generations, while ALU operation latency remains one cycle.

A solution for overcoming the IPC decay is to enlarge the processor instruction window [6, 15], both physically (issue buffer, physical registers...) and logically, through better branch prediction accuracy or branches removed by predication. However, the instruction window should be enlarged without impairing the clock cycle. In particular, the *issue buffer* and issue logic are among the most serious obstacles to enlarging the physical instruction window [11].

In this paper, we study the addition of a *preschedule* stage before the issue stage to combine the benefit of a large instruction window and a short clock cycle.

We introduce data-flow prescheduling. Instructions are sent to the issue buffer in a predicted data-flow order instead of the sequential order, allowing a smaller issue buffer. The rationale of this proposal is to avoid using entries in the issue buffer for instructions which operands are known to be yet unavailable.

In our proposal, this reordering of instructions is accomplished through an array of *schedule lines*. Each schedule line corresponds to a different depth in the data-flow graph. The depth of each instruction in the data-flow graph is determined, and the instruction is inserted in the corresponding schedule line. Lines are consumed by the issue buffer sequentially.

Section 2 briefly describes issue buffers and discusses related works. Section 3 describes our processor model and experimental set-up. Section 4 presents the general principle of prescheduling and introduces data-flow prescheduling. Section 5 describes a possible implementation for data-flow prescheduling. Section 6 analyses the efficiency of the implementation proposed based on experimental results. Finally, Section 7 gives some directions for future research.

2. Background and related works

The issue buffer is the hardware structure materializing the instruction window. Instructions wait in the issue buffer until they are ready to be launched to the execution units. Unlike the *reorder buffer* [14], instructions can be removed from the issue buffer soon after issuing, to make room for new instructions.

The two main phases of instruction issue are the *wake-up* phase and the *selection* phase [11]. The wake-up phase determines which instructions have their data dependencies resolved. The selection phase resolves resource conflicts and determines which instructions can effectively issue. The delay of the wake-up and selection phases increases with the issue buffer size [11], which makes a large issue buffer hardly compatible with a short clock cycle.

In some processors like the Alpha 21264 [8], the issue buffer is collapsable in order to maintain instructions in sequential order and facilitate the insertion of new instructions. Maintaining the sequential order allows the selection logic to give priority to older instructions.

In currently available processors, separate issue buffers are implemented for integer and floating-point instructions, typically 2 to 4 times smaller than the reorder buffer (in number of instructions). The integer issue buffer typically does not exceed 20 entries in current processors (20-entry integer queue in the Alpha 21264, 18-entry integer scheduler in the AMD Athlon, 20-entry reservation station in the Intel P6 ...).

Both micro-architectural and circuit-level solutions have been proposed for enabling the use of a large instruction window. In [11, 12], it was proposed to distribute the issue logic among multiple clusters of execution units. This solution trades global communications for fast local communications. The *trace processor* [13] is an example of such proposition. A characteristic of these propositions is that the instruction window size is proportional to the number of execution units.

A circuit-level approach was proposed recently for tackling the window size problem specifically [5] : the reorder buffer and the issue buffer are merged, and parallel-prefix circuits are used for the wake-up and selection phases.

The idea of prescheduling is not new. A *dependence-based* prescheduler was proposed in [11], that tries to form chains of dependent instructions in a set of FIFOs. This is further discussed in Section 4.1. An idea close to ours was proposed in [3], but with a different implementation.

Note on the issue buffer size. In some processors, instructions may have to be re-issued. For example, on the Alpha 21264 [7], when a load is predicted to hit in the data cache but actually misses, two issue cycles are annulled and the issue buffer state is restored. This requires that instructions remain valid in the issue buffer for a few cycle after they have been issued. These instructions constitute an "invisible" part of the issue buffer, which size depends on the issue width and on the number of pipeline stages between the issue stage and the execution stage. All issue buffer sizes reported in this study are for the "visible" part of the issue buffer.

3. Processor model and experimental set-up

The processor simulated in this paper is an out-of-order superscalar processor. The two processor configurations simulated, "ideal" and "8-way", are described on Tables 1 and 2 respectively.

The branch predictor simulated is a 3x16k-entry e-gskew predictor [10]. The size of the reorder buffer, i.e., the number of physical registers, was fixed large enough so that it does not interfere with our study. Branch misprediction recovery is performed as soon as a mispredicted branch is executed.

The cache latencies reported in Table 1 and 2 are "futuristic" values anticipating smaller feature sizes [1].

instruction cache	perfect
branch predictor	3x16k-entry e-gskew global history: 10 branches
fetch bandwidth	unlimited
front-end stages	1 (fetch/decode)
reorder buffer	4096 instructions
issue buffer	variable
issue width	N
execution units	N "universal" pipelined
back-end stages	issue, X execute, retire
main latencies (int)	most int: X=1 cycle mul: X=7 , div: X=20 load: X=1+4 (addr, cache) store: X=1+1 (addr, forward)
data cache	perfect
memory dependency predictor	perfect

Table 1. *"Ideal" configuration*

fetch	8 instructions
front-end stages	10 (+)
issue width	8
execution units	8 "universal"
L1 data cache	8 Kbytes, direct mapped 64-byte lines unlimited bandwidth
L2 data cache	perfect 15-cycle latency
store set predictor	SSIT : 16k entries (tagged) LFST : 128 entries (tagged)

Table 2. *"8-way" configuration : parameters not specified are identical to the "ideal" configuration.*

We assume the issue buffer is distinct from the reorder buffer. It schedules all the instructions, except those that are executed in the pipeline front-end, like unconditional jumps. The issue buffer is collapsable. When instructions are competing for the issue bandwidth, instructions that entered the issue buffer first are given priority.

As we focus our attention on the "visible" part of the issue buffer, we did not simulate the impact of pipeline stages between the issue stage and the execution stage, which is a distinct problem.

Load/store dependencies. When considering large instruction windows, we must pay attention to dependencies between loads and stores. Previous studies have shown that memory dependencies can be predicted with a high accuracy using past behavior. The memory dependency predic-

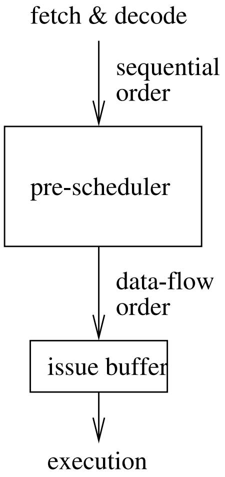

Figure 1. *The prescheduler sends instructions to the issue buffer in the order defined by data dependencies.*

tor used in this study for the "8-way" configuration is the *store set* predictor [4].

The *Store Set Identifier Table* (**SSIT**) is a 16k-entry tagged table (4-way set-associative). When a load misses in the SSIT, it is predicted to carry dependency with no in-flight store. A load is predicted to be dependent on the store encountered the more recently in its store set. The dependency is enforced by the issue logic : the load will issue after the store, so that it can catch the correct value. As recommended in [4], dependencies are enforced between stores belonging to the same store set in order to reduce the number of memory order violations.

In our simulations, the number of memory order violations did not exceed 2% of the number of branch mispredictions.

Benchmarks. All simulations are trace-driven simulations using the IBS traces [16]. The eight traces reflect the execution of sequential applications on a MIPS-based workstation, including system activity.

With the L1 data cache simulated in the "8-way" configuration, there is an average 5% cache miss ratio on our benchmarks, *nroff* having the lowest (2 %) and *verilog* and *video_play* the highest (7-8 %)

With the branch predictor simulated, the average number of instructions between consecutive branch mispredictions lies between 100 (*real_gcc*) and 350 (*video_play*), and between 200 and 250 for other benchmarks.

4. Prescheduling

In today processors, instructions are pushed in the issue buffer in sequential order, therefore instructions depending on a long dependency chain occupy the issue buffer for a long time. All the issue buffer entries are checked on every cycle. This process is time consuming and the delay increases with the number of entries in the issue buffer. The general idea behind prescheduling is to allow only instructions which are likely to become fireable in the very next cycles to enter the issue buffer. Information on data dependencies and instruction latencies are known before the issue stage and can be used for prescheduling.

The principle of prescheduling is depicted on Figure 1. Instead of being sent to the issue buffer in sequential order, instructions are reordered by a prescheduler so that they enter the issue buffer in the data-flow order, i.e., the order of execution assuming unlimited execution resources, taking into account only data dependencies. The instructions wait in a **preschedule buffer** until they can enter the issue buffer.

If the predicted data-flow order is close enough to an optimal issue order, then the issue buffer can be very small as it is relieved from the task of buffering instructions not yet fireable. In fact, the issue buffer size should be closer to the issue width than to the effective instruction window size.

The job of the hardware prescheduler is somewhat similar to that of a compiler scheduling instructions within basic blocks. However, a hardware prescheduler works on large traces of several tens or hundreds of instructions discovered at run time and which length is not known *a priori*.

Deadlocks. To prevent deadlocks, the prescheduler must ensure that if instruction B is dependent on instruction A, A enters the issue buffer before B.

4.1. Dependence-based prescheduling

The dependence-based prescheduler presented in [11] is an example of a prescheduling scheme. The preschedule buffer consists of several FIFOs. The issue buffer is the set of all FIFOs heads, hence the issue buffer size is equal to the number of FIFOs. The prescheduling logic forms chains of dependent instructions in FIFOs : an instruction is steered to a FIFO such that it depends on the last instruction in the FIFO. If it is not possible to append an instruction to an existing chain, the instruction is steered to an empty FIFO. When this is not possible, the steering logic stalls until one FIFO gets empty.

We verified that, experimentally, a dependence-based prescheduler with N FIFOs is roughly equivalent to an issue buffer of $2N$ instructions. A first limitation comes from the complexity of the data-flow structure of programs. There are many very short chains ending on a branch or a store, some chains are merged because of dyadic instructions, several chains are forked when the same register value is used several times. There is another limitation : the optimal distribution in FIFOs would require to enqueue instructions out of the program order and take into account instruction latencies. Trying to find the optimal distribution on simple examples convinced us that this is a hard problem, and that it would be difficult to improve on the published heuristic.

To overcome these limitations, the data-flow prescheduler proposed in this paper takes a different approach. First,

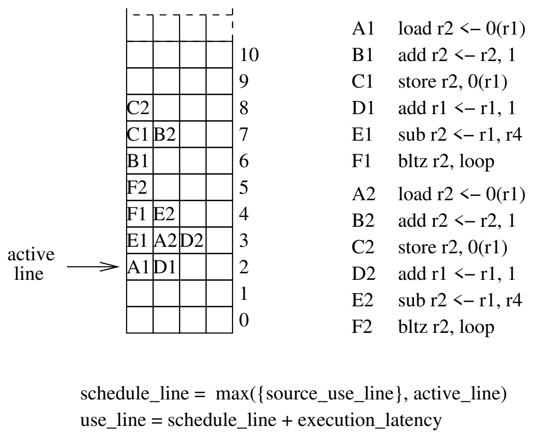

Figure 2. *Data-flow prescheduling example*

it defines a global data-flow order instead of a partial one. Second, it takes into account instruction latencies, in particular load latencies.

4.2. Data-flow prescheduling

Ideally, one would like to send instructions to the issue buffer only when they become fireable. We try to approach this ideal through real hardware. First we assume unlimited execution resources.

The depth of each instruction in the data flow graph is computed, taking into account data dependencies and instruction latencies (for simplicity, we assume all loads hit in the L1 cache). The data-flow depth for an instruction corresponds to its ideal issue cycle, assuming unlimited execution resources. The reordering of instructions is done through a preschedule buffer implemented as an array of **schedule lines**. Each schedule line is associated with an issue cycle. An instruction is inserted in the schedule line corresponding to its ideal issue cycle. The issue buffer consumes the lines sequentially. Hence, assuming unlimited execution resources and perfect prescheduling, instructions spend a single cycle in the issue buffer, and instructions in the same line are issued simultaneously.

The principle of data-flow prescheduling is illustrated on an example in Figure 2. The **active line** is the line which is currently feeding the issue buffer. The schedule line number for an instruction is always higher than the current active line number. The schedule line is determined with the following sequential prescheduling algorithm :

$$\text{schedule\_line} = \max(\{\text{source\_use\_line}\}, \text{active\_line})$$
$$\text{use\_line} = \text{schedule\_line} + \text{execution\_latency}$$

For each instruction, we define the **use line** as the line where its result is available as a source operand for dependent instructions.

Real hardware implementations will require further trade-offs as shown in the next section.

5. A possible implementation for data-flow prescheduling

5.1. The preschedule buffer

The preschedule buffer is an array of schedule lines. Each line is associated with a line counter indicating how many instructions are currently stored in the line. We define the **line width** as the maximum line counter value, that is, the maximum number of instructions that we allow in the same line. The line counter is incremented each time an instruction is written into the line. If the line counter value is already equal to the line width, this is a **line overflow**.

In each cycle, as slots are freed in the issue buffer, instructions are taken from the current active line to fill these slots. Once all the instructions in the current active line have been consumed, the active line number is incremented. We assume the active line number is incremented at most once per cycle, and only after the current active line is totally consumed.

Note that the active line number keeps increasing monotonically. However in practice, the number of physical lines, which we define as the **preschedule window**, is limited. The active line, schedule line and use line numbers manipulated are virtual line numbers which we map onto physical lines circularly. When the current active line is consumed, the physical line is recycled and its line counter is reset.

In this study, we have chosen the following policies for preschedule window overflows and line overflows :

Preschedule window overflow. If the schedule line for an instruction is greater than or equal to the sum of the active line and the preschedule window, prescheduling is blocked, waiting for the active line to proceed and physical lines to be recycled.

Line overflow. Similarly, if the targeted schedule line is full, prescheduling is blocked waiting for the active line to proceed. The schedule line of the blocked instruction is simply recomputed with the new active line number, as many times as necessary, until the instruction can be written in the preschedule buffer.

Note on the preschedule buffer implementation. In this study, it is implicitly assumed that the preschedule buffer is implemented with a direct-mapped two-dimensional array : one dimension is the line number, and the other dimension is the line counter value. We did not focus on optimizing the size of the preschedule buffer, as the access to the preschedule buffer can be pipelined without impairing the performance excessively. It should be noted that this is not the only possible implementation. In particular, it may be interesting to introduce some associativity by using line numbers and/or line counter values as tags.

5.2. Schedule line computation

This section describes the hardware supports used for implementing the prescheduling algorithm introduced in Section 4.2.

5.2.1 Registers dependencies

The register use line numbers are stored in a *Register Use Line Table* (**RULT**) similar to a register rename table. Each RULT entry is associated with a logical register. For each instruction, we must

1. read the RULT entries corresponding to its source register operands,
2. compute the schedule line as the maximum of the current active line number and the two source registers use line numbers,
3. add the instruction execution latency to the schedule line number to determine the destination register use line.

5.2.2 Load/store dependencies.

We slightly modified the store set predictor for being able to preschedule a load on a line after all the stores in its store set. When a store set ID (SSID) is obtained from the SSIT for a load or a store, this SSID is used to index the *Last Fetched Store Table* (**LFST**) [4]. Each LFST entry holds the *inum* of the more recent store in that store set. The *inum* is used to enforce load-store and store-store dependencies.

We modified the LFST entry by adding a field indicating the *store set maximum use line* number (**SSMUL**). After prescheduling a store, we compare the schedule line number of the store with the SSMUL obtained from the LFST (in case there was a hit). Then we take the maximum of the two line numbers, and we write the result in the LFST entry.

We assume the instruction set architecture does not allow indexed addressing, so that loads have a single register dependency (e.g., MIPS, Alpha). When prescheduling a load, the first use line number is a register use line read from the RULT, and the second use line number is the SSMUL read from the LFST, so that the load is scheduled on a line after all the stores in its store set.

Although a store is forced to be dependent on previous stores in its store set, the SSMUL is not used for prescheduling stores because stores are dyadic instructions and this would require an extra input in the schedule line computation.

5.2.3 The preschedule pipeline stage

Data-flow prescheduling requires to add a few extra pipeline stages. In particular, a *preschedule* stage is necessary for computing the schedule line numbers. This preschedule stage is critical for performance, as prescheduling is basically a sequential task. Nevertheless, we show how it can be parallelized.

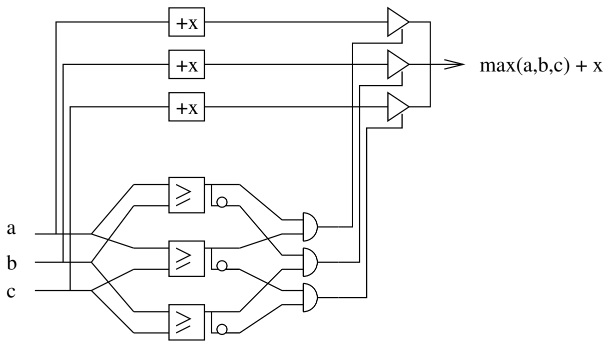

Figure 3. *Prescheduling operator computing* $max(a, b, c) + x$

The basic operation involved in data-flow prescheduling computes $max(a, b, c) + x$, with x being a small constant value depending on the instruction opcode. Figure 3 shows a possible implementation. For shortening the delay, the $+x$ operation can be performed in parallel with comparisons, as shown on Figure 3.

First, we show that the prescheduling logic may operate on small virtual line numbers, typically 12-bit wide. Therefore, the operator depicted on Figure 3 should have a propagation time shorter than a full 64-bit ALU. Second, we show that dependent $max(a, b, c) + x$ operations can be chained without increasing the circuit depth.

Maximum virtual line number. In Section 5.1, we did not consider the limitation of virtual line numbers. In practice however, virtual line numbers are coded with a limited number of bits.

When computing the use line of the result of an instruction, if the maximum virtual line number is exceeded, prescheduling is blocked until the processor instruction window gets completely drained. Then, the active line number and all RULT and LFST entries are reset to 0, and prescheduling resumes from the blocked instruction.

This method ensures that the content of the RULT is always coherent, which is important for avoiding deadlocks. We found that virtual line numbers can be coded with 12 bits with no significant performance loss. For example, with 12 bits, if 4 instructions are issued per cycle, a control-flow break is necessary every 16k instructions, which is an order of magnitude larger than the average distance between branch mispredictions.

Though this solution is simple, other solutions are possible for keeping the RULT content coherent. For example, we could invalidate a RULT entry when the last instruction which wrote in it leaves the preschedule buffer (this would require to change the definition of the *max* operation in the

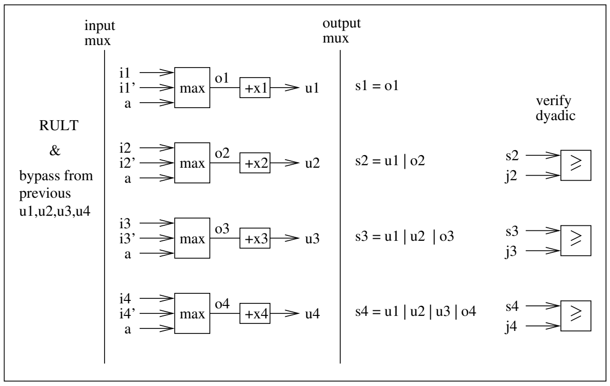

Figure 4. *Parallel computing of the use line u_n and schedule line s_n of a group of instructions.*

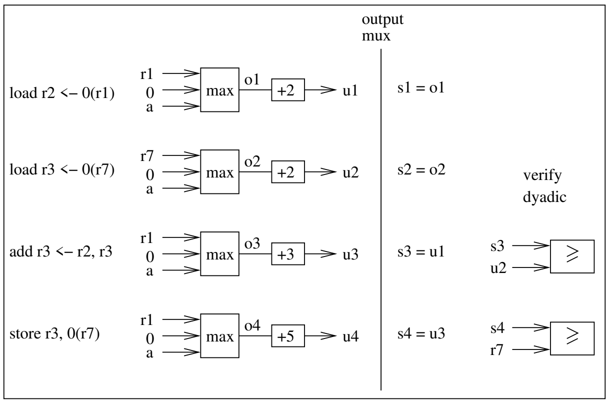

Figure 5. *Example of parallel prescheduling. Loads and stores have a latency of 2 cycles. We neglect source r3 of the add and source r7 of the store.*

schedule line computation).

Parallel prescheduling. The preschedule stage is principally constituted of N $max(a, b, c) + x$ operators, N being the pipeline width.

Previous pipeline stages participate in the preschedule task, performing intra-group dependencies analysis and determining instruction latencies in order to set the inputs of all $max(a, b, c) + x$ operators. However this preliminary work is not the core of the problem (the analysis of intra-group dependencies is necessary also for register renaming).

The main issue is to perform in the same cycle several chained $max(a, b, c) + x$ operations. We present here a possible solution to break dependency chains and allow the implementation of data-flow prescheduling.

Figure 4 shows the circuit for computing the schedule line and use line numbers $\{s_n\}$ and $\{u_n\}$ of a group of instructions, based on the operation $max(a, b, c) + x$. One entry, a, is the active line number. The two other entries i_n and i'_n are the source operand use cycles. The settings for the i_n and i'_n sources and the command for the output multiplexor depend on intra-group dependencies determined in previous pipeline stages.

If instruction n does not depend on previous instructions in the group, then i_n and i'_n are taken from the RULT or the LFST, and the increment x_n is equal to the instruction latency l_n. The schedule line number s_n, in this case, is read at the output o_n of the *max* operator.

Now let use suppose that instruction n depends on a previous instruction in the group. If instruction n is monadic and depends on instruction m, then the n^{th} operator is configured as follows : $i_n = i_m$, $i'_n = i'_m$, $x_n = x_m + l_n$, and $s_n = u_m$.

The difficulty comes from dyadic instructions dependent on previous instructions in the group. We propose to treat dyadic instructions like monadic instructions by neglecting one source operand, that is, predicting which of the two source use line numbers is not the maximum of the three input use line numbers.

The *not-the-max* predictor we simulated is a 2-bit saturating counter stored along with the instruction. The most significant bit of the counter indicates which source operand to neglect.

To check the prediction, we verify that the schedule line is greater than or equal to the neglected source use line : we compare the s_n value with the source use line j_n we neglect in the computation of s_n. If the prediction is correct, the 2-bit counter is strengthened, else it is weakened.

Upon a misprediction, the group is split : the mispredicted dyadic instruction and following instructions will be prescheduled in the next cycle. An example is given on Figure 5.

From our experimentations, we found an average of one *not-the-max* misprediction every 30 instructions. When fetching 8 instructions per cycle, *not-the-max* mispredictions decrease the fetch rate by 5-10%.

5.3. Deadlocks

Keeping the RULT coherent and stalling upon a preschedule window overflow ensures that if an instruction B is register-dependent on an instruction A, then B cannot enter in the issue buffer before A. So the data-flow prescheduler described previously cannot experience deadlocks because of register dependencies.

Load-store dependencies cannot generate deadlocks. A load is always scheduled on a line after the store it is predicted to depend on. Note that if the prescheduler failed to detect a load-store dependency, the load cannot be blocked in the issue buffer by the store.

Nevertheless, artificial dependencies between stores in the same store set can cause deadlocks, because they are not taken into account in the schedule line computation of a store. However, such deadlocks are very rare. Most of our simulations experienced no deadlock at all. Only a few simulations experienced deadlocks, but never with less than 1 million instructions per deadlock.

Deadlocks can be detected and solved easily : when no instructions have been issued for a certain number of cycles, we release all stores in the issue buffer by clearing their artificial dependencies. These dependencies are not necessary for a correct execution, they were introduced only for reducing the number of memory order traps.

6. Experimental evaluation

6.1. Line size trade-off

The line size is an important parameter of the data-flow prescheduler. If the line is chosen too small, prescheduling will stall too often, limiting the effective instruction window. On the other hand, if the line is chosen too large, many wrong-path instructions will enter the issue buffer before correct-path instructions and may delay the correct path.

The effective instruction window grows proportional to the square of the line size. As the line size is increased, prescheduling stalls less frequently, and more instructions can enter the prescheduler. So there is a direct relation between the line size and the effective instruction window.

We simulated an "ideal" configuration, replacing the e-gskew branch predictor with a perfect branch predictor. In these conditions, the instruction fetch rate is limited only by line overflows. In this experiment and all subsequent ones, the preschedule window is fixed to 128 physical lines so that it is not a performance bottleneck.

Figure 6 shows the IPC with and without a prescheduler. For the configuration with a prescheduler, we keep the issue buffer size fixed to 32, and we vary the line size. For the configuration with no prescheduler, we vary the issue buffer size. The issue width is kept equal to the issue buffer size.

This experiment shows the relation between the line size and the effective instruction window size. For example, a prescheduler with a line size of 16 instructions gives the same IPC as an an issue buffer of 128 instructions (for the instruction latencies simulated, and with unlimited execution resources).

We observed that the average number of instructions waiting in the preschedule buffer is roughly proportional to the square of the line size, which is coherent with the square-root law observed in [9].

It should be noted that, on Figure 6, the issue width is larger than the line size. However, when execution resources are limited, a data-flow prescheduler is not exactly equivalent to a large issue buffer, because the data-flow order differs from the optimal issue order. In case of a resource conflict, a large issue buffer should give priority to older instructions. A data-flow prescheduler does not have this degree of freedom. In particular, it is possible for wrong-path instructions to delay the execution of correct-path instructions if the line size is larger than the issue width.

Sampling method. Our simulator is trace driven, it is not able to simulate instructions on the wrong path. However, we have simulated the impact of wrong-path instructions from the observation that, from the point of view of the data flow structure, it is very hard to distinguish the wrong path from the correct path (otherwise, this would provide a way to detect mispredicted branches). This observation led us to a sampling method, using correct-path instructions to simulate the wrong path. A similar technique was used in [2].

The whole instruction trace is injected in the simulator, as usual, so that its internal structures (branch predictor, cache, store sets, ...) are kept "warm". However, we collect statistics only for one slice every 100 on average. We define a slice as a piece of instruction trace delimited by two consecutive branch mispredictions. For simulating the wrong path, we inject in the simulator the correct-path instructions which follow the slice currently sampled. The time counter starts counting when the first instruction in the slice is fetched, and the counting stops when the mispredicted branch ending the slice is executed. The "sample" IPC is the total number of instructions in all slices divided by the time cumulated on all samples.

To verify the validity of the method, we have simulated a large issue buffer giving priority to older instructions, so that instructions on the wrong path have no effect. We also ran simulations without sampling so as to obtain the "oracle" IPC, that is, the IPC obtained when the instruction fetching stalls after each mispredicted branch. The difference between the "sample" IPC and the "oracle" IPC measured on the IBS benchmarks are within $\pm 2\%$ for 5 of the 8 IBS benchmarks, the three others being -2.2% (gs), $+2.7\%$ (sdet) and -3.8% (nroff). It should be noted that our sampler uses a random number generator which is always initialized with the same seed. Hence the sequence of slices that are sampled is fixed for a given benchmark, which makes comparisons safer. In the remaining, the "sample" IPC is used as the performance metric.

Impact of wrong path instructions. Figure 7 shows the "sample" and "oracle" IPC measured on an "ideal" configuration as a function of the line size, for an issue width of 4 and 8. To gain place, we show only the harmonic mean on all benchmarks.

The issue buffer size was fixed to 32, so that it is not a performance bottleneck. The difference between the "oracle" and "sample" IPC values quantifies the performance loss associated with potentially issuing wrong-path instructions before correct-path instructions.

We observe that when the line size is equal to the issue width, the instructions on the wrong path have no impact on performance, which is coherent.

As the line size increases, so does the effective instruction window, and this increases the "sample" IPC. However, after a certain line size, wrong-path instructions begin to

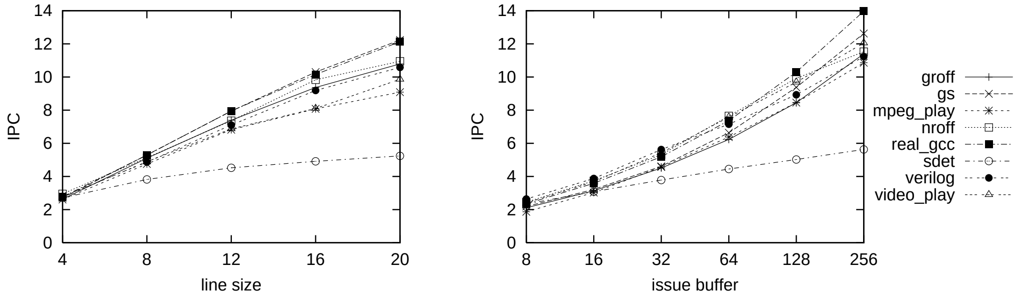

Figure 6. *IPC of an "ideal" configuration with a* **perfect branch prediction**. *On the left graph, there is a prescheduler. and we vary the line size. On the right graph, there is no prescheduler and we vary the issue buffer size.*

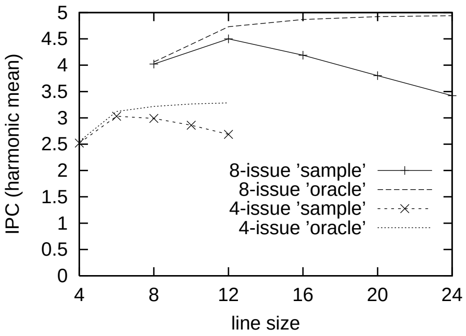

Figure 7. *Data-flow prescheduling on an "ideal" configuration. Harmonic mean on all benchmarks of the "sample" and "oracle" IPC as a function of the line size, for an issue width of 4 and 8.*

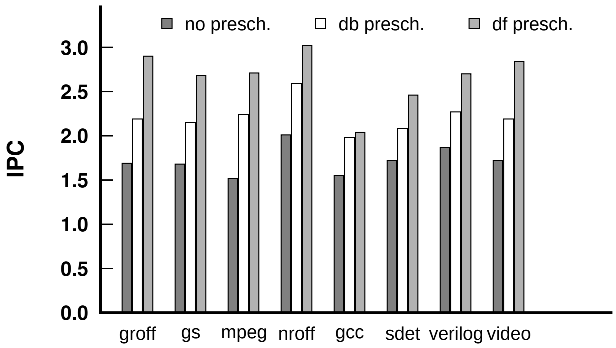

Figure 8. *IPC on a "8-way" configuration with a* **8-entry issue buffer** *with a data-flow prescheduler (12-instruction lines), a dependence-based prescheduler (8 FIFOs), and with no prescheduler.*

consume too much issue bandwidth, and the "sample" IPC falls.

With the instruction latencies simulated, the optimal line size is approximately 50 % larger than the issue width. For example, for an issue width of 8, we should take a line size of 12. In this case, on our simulations, wrong-path instructions generate a 5% performance loss on average.

6.2. Data-flow prescheduling effectiveness

In this section, we compare three "8-way" configurations with the same issue buffer size: one uses a data-flow prescheduler, another uses a dependence-based prescheduler (the issue buffer size is the number of FIFOs), and the last has no prescheduler.

For the data-flow prescheduler, the line size is set to 12 instructions (following the conclusion of Section 6.1) and we take into account specific implementation constraints : virtual line numbers are coded on 12 bits, the preschedule stage uses *not-the-max* predictions, and the pipeline front-

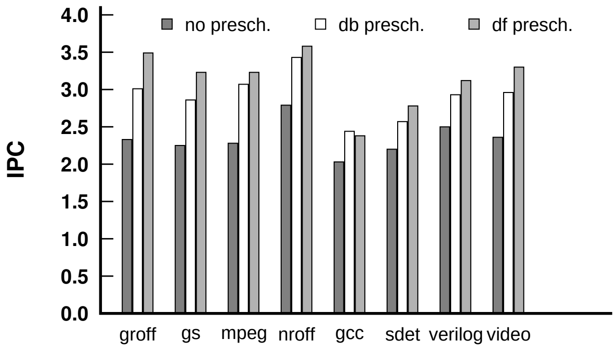

Figure 9. *IPC on a "8-way" configuration with a* **16-entry issue buffer** *with a data-flow prescheduler (12-instruction lines), a dependence-based prescheduler (16 FIFOs), and with no prescheduler.*

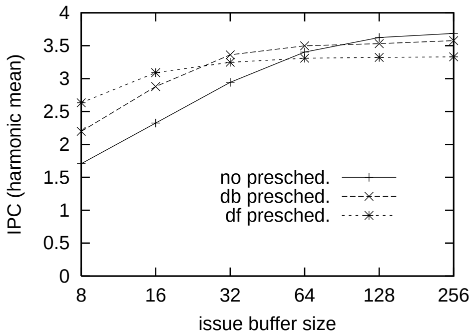

Figure 10. *Harmonic mean of the IPC on all benchmarks as we vary the issue buffer size.*

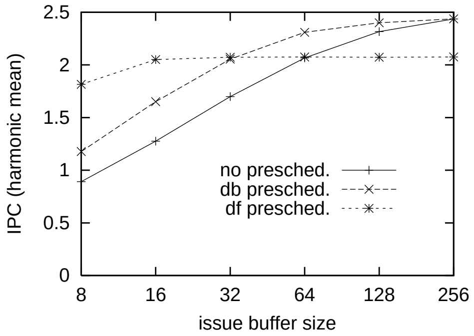

Figure 11. *Harmonic mean of the IPC when the L1 cache is removed and predicted load latencies correspond to a L2 cache access.*

end features 13 stages instead of 10 for the other two configurations (i.e., assuming data-flow prescheduling requires 3 extra pipeline stages).

Figures 8 and 9 show the IPC for issue buffer sizes 8 and 16 respectively.

First, we observe that the data-flow prescheduler, on average, outperforms the dependence-based prescheduler for these issue buffer sizes. With a 8-entry issue buffer, a data-flow prescheduler is on average 20% more performant than a dependence-based prescheduler and 54% more performant than with no prescheduler. With a 16-entry issue buffer, a data-flow prescheduler is still 7% more performant than a dependence-based prescheduler and 33% more performant than with no prescheduler.

Analysis. Figure 10 shows the harmonic mean of the IPC on all benchmarks as we vary the issue buffer size. With a data-flow prescheduler, it is beneficial for the issue buffer to be larger than the line size : the IPC with an issue buffer of 16 is higher than with an issue buffer of 8. The main reason is that the data-flow order is not the optimal issue order because of the limited issue width. An issue buffer larger than the line size gives more opportunities to the issue logic for correcting the preschedule order and get closer to an optimal issue order.

We can observe that increasing the issue buffer size from 16 to 32 brings on average a slight performance gain with a data-flow prescheduler. Actually, looking at benchmarks individually, it is correlated with the frequency of data cache misses. Benchmarks with a high data cache miss rate (e.g., *verilog*, *video_play*) benefit from an issue buffer of 32, whereas benchmarks with few cache misses (e.g., *nroff*) do not. This is because cache misses degrade the accuracy of the predicted data-flow order.

From these curves, it appears that an effective instruction window of 128 instructions is sufficient (actually, *real_gcc* requires a window of only 64 instructions because the distance between branch mispredictions is twice smaller than for other benchmarks). Without a prescheduler, we would need a very large issue buffer to implement such a large window. With a dependence-based prescheduler, the difficulty is “halved” : 16 FIFOs emulate an effective window of about 32 instructions.

On the other hand, according to Figure 6, a data-flow prescheduler emulates a window of about 64 instructions with a line size of 12. In practice, a part of this potential is consumed by the impact of wrong path instructions, by the extra pipeline stages, and by cache misses degrading the accuracy of the predicted data-flow order.

To better demonstrate the potential of data-flow prescheduling and give hints for future works, we have performed a simple experiment which results are shown on Figure 11. In this experiment, we remove the L1 data cache so that all loads access directly to the L2 cache, and the prescheduler predicts that the load latency corresponds to a L2 cache access. We can observe that this emphasizes the importance of a large issue buffer : a larger instruction window is needed to saturate the execution units. We can also observe that we get the full potential of the data-flow prescheduler with an issue buffer of 16. This is because the data-flow order is now very accurate, as there are no longer L1 cache misses. More interesting, the *no-prescheduler* curve now crosses the *data-flow-prescheduler* curve at an issue buffer size of 64, despite the impact of wrong-path instructions and extra pipeline stages. By predicting longer load latencies, we decrease the frequency of line overflows and we allow more instructions to enter in the prescheduler. In other words, predicting longer latencies enlarges the effective instruction window. Now, with the same 12-instruction line size, we are emulating an effective window larger than 64 instructions.

7. Conclusion and future works

The issue buffer is one of the critical pipeline stages in modern out-of-order processors. The traversal time of the issue stage increases with the issue buffer size. This may prevent the implementation of large issue buffers.

Data-flow prescheduling allows to reorder instructions dynamically. The goal is to push instructions in the issue buffer in the data-flow order rather than in sequential order. This allows to reach the same IPC using a smaller issue buffer.

The implementation proposed in this study is only a point in the design space. In particular, we did not explore the possibility of introducing associativity in the preschedule buffer. Associativity might be useful for “smoothing” the precheduler behavior. This concerns both the utilization of the preschedule buffer space and the definition of line overflows.

Prescheduling (or other techniques tackling the same problem) should be viewed as a way to tolerate long instruction latencies. The main situation requiring a large instruction window is when there is not enough instruction parallelism to saturate the execution units. This is often the case on code sections experiencing frequent data cache misses.

The data-flow prescheduler we simulated predicts that all load latencies correspond to a L1 data cache hit. However, for applications with frequent cache misses, we would like the prescheduler to predict longer load latencies.

As part of future work, it would be interesting to study how the memory hierarchy design could take advantage of the latency tolerance afforded by a prescheduler. In particular, hit-miss prediction techniques [17, 7] should be considered as part of the problem.

Future work should also focus on the problem of bypass latencies. In this study, we assumed a centralized instruction window feeding a compact pool of execution units. However, clustered architectures are appearing, with restricted bypass networks. Data-flow prescheduling might also be interesting for those architectures.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[2] M. Butler and Y. Patt. An investigation of the performance of various dynamic scheduling techniques. In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[3] R. Canal and A. González. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing*, 2000.

[4] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[5] D.S. Henry, B.C. Kuszmaul, G.H. Loh, and R. Sami. Circuits for wide-window superscalar processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[6] N.P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. Workshop on Mixing Logic and DRAM (ISCA'97) . http://iram.cs.berkeley.edu/isca97-workshop/.

[7] R.E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, March 1999.

[8] D. Leibholz and R. Razdan. The Alpha 21264: a 500 MHz out-of-order execution microprocessor. In *Proceedings of IEEE COMPCOM*, 1997.

[9] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[10] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[11] S. Palacharla, N. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[12] N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[13] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[14] J.E. Smith and A.R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 1985.

[15] S.T. Srinivasan and A.R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31th Annual International Sympoisum on Microarchitecture*, 1998.

[16] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[17] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.