By-Passing the Out-of-Order Execution Pipeline to Increase Energy-Efficiency

Hans Vandierendonck
Ghent University
Dept. ELIS / HiPEAC
St.-Pietersnieuwstraat 41
B-9000 Gent, Belgium
hvdieren@elis.ugent.be

Philippe Manet, Thibault Delavallée,
Igor Loiselle, Jean-Didier Legat
Université catholique de Louvain
Microelectronics Laboratory
Place du Levant, 3
B-1348 Louvain-la-Neuve, Belgium
{manet,delavallee,loiselle,legat}@dice.ucl.ac.be

ABSTRACT

Out-of-order execution significantly increases the performance of superscalar processors. The out-of-order execution mechanism is, however, energy-inefficient, which inhibits scaling superscalar processors to high issue widths and large instruction windows.

In this paper, we build on the observation that between 19% and 36% of the instructions are immediately ready for execution, even before entering the issue queue. Yet, these instructions proceed to the energy-consuming steps of instruction wake-up and select and they needlessly occupy space in the issue queue. To save energy, we propose for these instructions to by-pass the out-of-order execution core. Instead, we execute them on an energy-efficient single-issue in-order by-pass pipeline.

The by-pass pipeline executes a significant fraction of all instructions, allowing performance-energy trade-offs with respect to the issue width of the out-of-order pipeline and to the issue queue size. By making these trade-offs, we show energy reductions of 53% for the issue queue, 33% for the register file and 31% in the write-back and wake-up logic. Performance remains almost unaffected.

Categories and Subject Descriptors: C.1.1 [Processor Architectures]: Single Data Stream Architectures; C.5.3 [Microcomputers]: Microprocessors

General Terms: Performance, Design, Experimentation

Keywords: Out-of-order execution, instruction scheduling, instruction wake-up, energy-efficiency

1. INTRODUCTION

Superscalar processors employ out-of-order execution (also known as dynamic instruction scheduling) to increase the issue rate and to overcome long instruction latencies [13, 25]. However, out-of-order execution consumes a significant amount of power by design. The issue queue together with the wake-up and select logic stands for a significant fraction of the total energy consumed by a processor. Reported numbers are in the range from 18% for the Alpha 21264 [7], 19% for the Pentium Pro [14], and up to 40% for the Pentium 4 [8]. Furthermore, the issue queue is a hot spot, making it more important to reduce its power consumption than other components.

The issue queue controls out-of-order execution [19, 25]. Instructions are first placed in the issue queue where they wait for their operands to become available. Once their operands are available, instructions can be selected for execution. After execution, the instructions broadcast their result together with a destination tag to the issue queue to wake-up dependent instructions. Both the wake-up and select steps require associative searches through the issue queue which disallows large issue queues [19,20] and is detrimental for energy-efficiency [2, 4, 5, 11].

Although all instructions are placed in the issue queue and all of them have to be selected for execution dynamically, it is not uncommon that one or all of the operands of an instruction are available before the instruction enters the issue queue [4,11]. The reason for this is inherent to the nature of programs [6]: register values can remain live during long periods of time, so instructions dependent on such registers are immediately ready to execute. In our baseline 4-issue processor model, described in Section 4, between 19% and 36% of all dispatched instructions have all of their operands ready when they enter the issue queue (Figure 1). Yet, these instructions are actively selected for execution, they needlessly

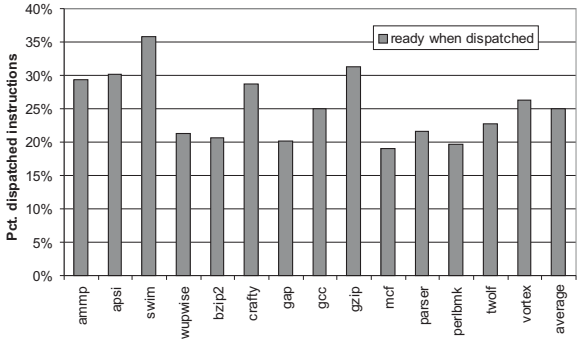

Figure 1: Percentage of dispatched instructions that are immediately ready.

CF'07, May 7–9, 2007, Ischia, Italy.

consume issue queue space and they needlessly participate in the wake-up process.

This paper proposes a mechanism to reduce energy consumption by not executing all instructions on the out-of-order execution pipeline. Instead of dispatching instructions that are ready to execute to the issue queue, we propose to send these instructions to a separate in-order pipeline that by-passes the out-of-order execution pipeline. The by-pass pipeline is a single-issue in-order pipeline, making it highly energy-efficient. The by-pass pipeline consists of three stages: register file read, execute and write-back. There is no issue queue, select or wake-up logic as instructions execute immediately when they are dispatched to the by-pass pipeline.

Some architectural modifications are necessary to allow correct execution of instructions on the by-pass pipeline. The main hurdle is access to the register file to fetch the operands. The register file is strongly integrated in the out-of-order core. Adding read and write ports for the by-pass pipeline would increase latency and power consumption. Therefore, we add a second register file holding a subset of the registers. This register file holds only those registers whose values remain live for a long period of time. These registers are typically ready when the instruction dispatches. These long-living registers are copied to the new register file selectively to ensure correct operation.

The by-pass pipeline provides additional issue width at a low energy cost so it improves performance in an energy-efficient way. However, we show that the by-pass pipeline enables further energy-related optimizations in the out-of-order execution pipeline while keeping the same performance as the baseline processor. In particular, we show that both issue width and issue queue size can be reduced, yielding an 8.4% reduction of energy and a 9.7% reduction of the *total* power consumption. Overall, these trade-offs reduce issue queue energy by 53%.

In the remainder of this paper we discuss related work in Section 2. Then, we describe an architecture that bypasses the out-of-order execution core for instructions whose dependencies are satisfied (Section 3) and evaluate its performance and energy-efficiency (Section 4). The paper concludes in Section 5.

2. RELATED WORK

Some researchers have proposed techniques to speedup execution by letting instructions by-pass the functional units. However, these techniques are not likely to increase energy-efficiency. Instruction reuse [16,26] was proposed to speedup execution by caching instruction results. When the operands to an instruction are available, the result is looked up in the reuse table. If the table hits, the instruction is not executed. The reuse table can be large, e.g. 221 KB [16].

Some instructions perform trivial computations, e.g., multiplications by zero or one, adding zero, etc. [21, 29]. These instructions need not be executed but can instead by-pass the functional units. However, instructions remain in the issue queue until the trivializing operand is available [29]. Thus, trivialization has less potential to improve energy-efficiency than the by-pass pipeline proposed in this work.

A number of works have previously noted the energy-inefficiency of the issue queue and the operation of select and wake-up logic. Kim and Lipasti [11] and Ernst and Austin [4] reduce the complexity of wake-up logic by reducing the number of tag comparators. This is possible since many instructions have one or both operands ready when they enter the issue queue.

Instruction select logic can be speculatively pipelined to reconcile the complex select logic with a fast clock [28]. Select-free scheduling is also possible by speculatively marking all woken-up instructions as selected. An instruction scheduler is still necessary to detect when more instructions have been woken up than issue bandwidth limitations allow [1]. Yet another attempt to simplify instruction select is to combine instructions into macro-ops [12]. Scheduling macro-ops allow multi-cycle scheduling delays as the operations themselves take multiple cycles to execute.

Önder and Gupta [17] and Michaud and Seznec [15] investigate techniques that reschedule the instruction stream based on data-flow information and then dispatch this rescheduled stream to a small issue queue. Canal and González [3] present issue schemes based on in-order issue queues. Instructions enter the queue only when they are ready (first-use scheme) or they are placed in a position in the queue depending on their dataflow distance (distance scheme). Both schemes require additional out-of-order issue queues in order not to loose significant amounts of performance.

Folegnani and González [5] remove some of the inefficiencies of the issue queue by gating-off wake-up for empty issue queue entries and for ready operands. They also propose to dynamically resize the issue queue, a technique that has been refined by others [2, 9].

A large instruction window may be combined with energy-efficient issue logic when the window is partitioned. Palacharla, Jouppi and Smith [19] propose to cluster the issue queue based on instruction dependencies. The instruction scheduler operates on each cluster of the issue queue independently while inter-cluster wake-up suffers a delay due to the larger distance.

Raasch, Binkert and Reinhardt [20] divide the instruction window over time, where instructions are promoted from one slice of the window to the next as they come nearer to execution.

The by-pass pipeline proposed in this work may be viewed as a particular incarnation of Palacharla's clustered architecture with non-uniform clusters. Note however that such architectures are very sensitive to the quality of the instruction steering across clusters [22]. Sub-optimal instruction steering causes an important reduction of IPC which must be offset by a higher overall issue width. This architecture retains the out-of-order execution core because of its high performance efficiency. We show that our architecture achieves the same IPC for the same total issue width as a baseline out-of-order architecture.

3. DESCRIPTION OF THE ARCHITECTURE

A significant number of instructions have their operands ready when they are dispatched to the issue queue (Figure 1). Having these instructions go through the steps of select and wake-up wastes energy. Also, storing them in the issue queue causes them to participate in the wake-up process of other instructions and waste energy there. To increase energy-efficiency, we propose to execute such instructions using a separate low-power pipeline.

Figure 2 shows a pipeline diagram of the architecture. Besides the out-of-order execution pipeline (which may itself consist of separate integer, floating-point and memory pipelines), we add the by-pass pipeline. The by-pass pipeline

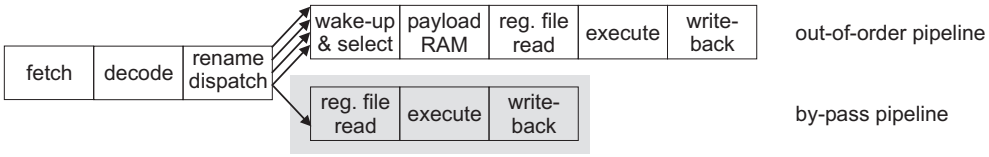

Figure 2: Pipelines of the architecture showing the out-of-order execution pipeline and the by-pass pipeline.

has low complexity and it is a short (fast) pipeline. The block diagram of our architecture is shown in Figure 3.

3.1 Baseline Architecture

We assume a baseline out-of-order processor as depicted in Figure 3. Load and store instructions are split in two operations: an effective address computation part and a cache access part. The first is dispatched to the general issue queue and the latter is dispatched to a specific load/store queue that also tracks memory dependencies. The register file is a merged architectural and rename register file [24], similar to the Alpha 21264 [10]. It is accessed after instructions are selected for execution.

3.2 The By-pass Pipeline

The by-pass pipeline is a simple 3-stage pipeline containing one pipeline stage to access the long-living register file, one execution stage and one write-back stage (Figure 2). There is no issue queue: instructions are sent to this pipeline only when they can execute immediately. The by-pass pipeline executes instructions in-order and has limited issue width (1 instruction per cycle in this paper).

The execution unit in the by-pass pipeline performs only simple 1-cycle operations on integer registers. Memory accesses cannot be performed from the by-pass pipeline, but memory address computations (effectively an addition) can execute in the by-pass pipeline. The by-pass pipeline can resolve branch targets and signal branch mispredictions.

3.3 The Long-Living Register File

All register values are stored in the main register file, but long-living registers have a copy in the long-living register file. The long-living register file is indexed by the architectural register name. Only a subset of the registers are stored in the long-living register file. A vector of valid bits indicates which registers are present.

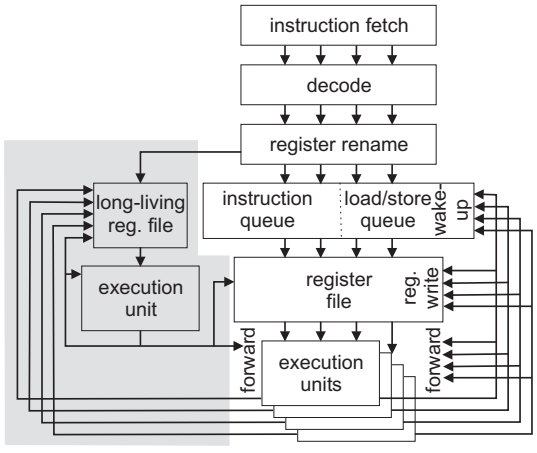

Figure 3: The proposed architectural modifications. The shaded area contains a simple pipeline whereby instructions by-pass the out-of-order execution core.

When an instruction is dispatched to the out-of-order execution pipeline, its destination register's valid bit is cleared. This signals that instructions dependent on that register cannot be dispatched to the by-pass pipeline. The valid bit may be reset when the register is recomputed under the conditions explained in the next section.

The long-living register file is small in comparison to the physical register file because (i) its size is proportional to the number of architectural registers and (ii) it only holds the integer registers and (iii) it has fewer read/write ports. In our experiments, the long-living register file holds 32 registers (Alpha ISA) while the physical register file holds 96 registers.

Note that the long-living register file only holds copies of register values, so it is not strictly necessary for correct execution. This implies that its contents may always be discarded by setting all valid bits to zero. This may be beneficial when performing context switches or when recovering from complicated misprediction scenarios.

3.4 Copying Long-Living Registers

We copy registers to the long-living register file as often as possible to promote by-passing the out-of-order execution core. Newly computed register values are stored in the long-living register file when the executing instruction is the most recent in-flight instruction that produces the architectural register. Applying this condition ensures that dependent instructions will obtain the correct value from the long-living register file when they are dispatched to the by-pass pipeline.

The long-living register file is updated when the most recent version of an architectural register is produced. To decide if an instruction produces the most recent version of an architectural register, we add the *most recent renaming* bit vector to the rename stage. This bit vector adds one bit of information to every physical register. The bit is 1 if the physical register is the most recent renaming for an architectural register. It is 0 if the renaming has already been superseded by another renaming.

When an instruction executes and produces a value for its destination register, the most recent renaming bit vector is checked. If the bit corresponding to the physical destination register is 1, the corresponding architectural register is updated in the long-living register file. Otherwise, no update is performed on the long-living register file. These conditions are applied to all instructions, independent of the pipeline they execute on.

Note that register rename map tables are implemented either as a RAM table or as a CAM table [18]. The CAM table implementation already contains exactly the most recent renaming bit vector as part of the rename map table.

3.5 Instruction Steering

The number of instructions steered to the by-pass pipeline per cycle is limited to the by-pass pipeline's issue width as there is no issue queue to temporarily buffer instructions. In this paper, at most one instruction is steered to the by-pass pipeline per cycle. If multiple instructions in a single fetch group are applicable to execute on the by-pass pipeline, then only the first instruction in the group is steered to the by-pass pipeline.

An instruction is steered to the by-pass pipeline when all of its operands are available in the long-living register file (as determined by the vector of valid bits) and its operation can be executed on the simple 1-cycle integer ALU in the by-pass pipeline.

3.6 Eliminating Wake-up for By-passing Instructions

Instructions that execute on the by-pass pipeline already avoid accessing the issue queue, instruction selection and physical register file read. These instructions are also allowed to skip a wake-up operation in the issue queue and, depending on architectural details, it may also be possible to eliminate forwarding the result to the execution units in the out-of-order core.

In the general case, whenever an instruction is dispatched to the out-of-order execution pipeline, all of its dependencies that are produced in the by-pass pipeline are marked ready. Thus, by-passing instructions need not wake-up their dependents. The timing between dependent instructions is correct, as the by-passing instruction skips the instruction select stage and gains the dispatch-execute delay, which is at least one cycle.

When the dispatch-execute delay is more than 1 cycle, operands are available in time for every instruction, even when assuming the worst case: When the producing instruction is dispatched in the same cycle as the dependent instruction and the dependent instruction is selected immediately after dispatch, then write-back of the producing instruction's result occurs at least in the same cycle as the dependent instruction reads the register file.

When the dispatch-execute delay is 1 cycle, then a forwarding path from the by-pass pipeline to the out-of-order pipeline's execution units must be added.

3.7 Recovering from Misspeculation

Wrong speculation (e.g. control-flow mispredictions) may corrupt the state of the long-living register file by copying wrong-path register contents into the long-living register file. These values must be removed when recovering from the misspeculation.

One can distinguish two schemes to recover the long-living register file: (i) all registers produced on the wrong execution path are invalidated or (ii) all registers produced on the wrong path are copied back from the main register file. In either case, the valid bits must be set appropriately.

3.8 Discussion

The by-pass pipeline increases hardware complexity. The area overhead is relatively small. We have already pointed out that the long-living register file is much smaller than the physical register file. The one integer ALU is small compared to the complex ALUs in the out-of-order pipeline (single-cycle integer operations vs. multi-cycle integer operations and floating-point operations). By this, the overall static power consumption will not be affected much. In the remainder of this paper, we consider only dynamic power.

Energy reduction is obtained mostly by reducing the activity factors in components with high energy consumption: the issue queue and write-back stages (including wake-up and select logic). As the issue queue is a hot spot, saving power here is even more important for temperature reasons.

Energy is increased in a few components. The physical register file needs one additional write port to write-back results computed in the by-pass pipeline. The result buses must be extended to send results to the long-living register file where they are conditionally written back. This increases the load on the result buses. However, both cases are mitigated when we trade-off performance for energy by reducing the complexity of the out-of-order pipeline. The number of read/write ports to the physical register file decreases by 3 when the issue width is decreased by 1 (net gain of 2 ports). The load on the result buses is strongly reduced when shrinking the issue queue. Both optimizations increase overall energy-efficiency.

Note that it is possible to reduce the number of write ports of the long-living register file. We find that only about 10% of the write-backs in the out-of-order pipeline are copied to the long-living register file, so the write bandwidth in the latter may be substantially less than the write-back bandwidth. If reducing the write bandwidth implies loosing some copy opportunities, this would reduce the energy efficiency of the processor, but not impact its correctness.

Table 1: Baseline Processor Model

Processor core	
Issue width	4 instructions
Reorder buffer	96
Issue queue	32
Load-store queue	48
Dispatch-execute delay	5 cycles
ALUs	3 simple int, 1 int mul/div 2 simple fp, 1 fp mul/div
Fetch Unit	
Fetch width	4 instructions, 2 branches/cycle
Instruction fetch queue	8 instructions
Fetch-dispatch delay	8 cycles
Cond. branch predictor	64 Kbits O-GEHL
Return address stack	16 entries, checkpoint top 2
Branch target buffer	256 sets, 4 ways
Cascaded branch target predictor	64 sets, 4 ways 8-branch path history
Memory Hierarchy	
L1 I/D caches	64 KB, 4-way, 64B blocks
L2 unified cache	256 KB, 8-way, 64B blocks
L3 unified cache	4 MB, 8-way, 64B blocks
Cache latencies	1 (L1), 6 (L2), 20 (L3)
Memory latency	150 cycles average

4. EVALUATION

We use a mix of benchmarks from the SPEC CPU2000 integer benchmarks and floating-point benchmarks. The

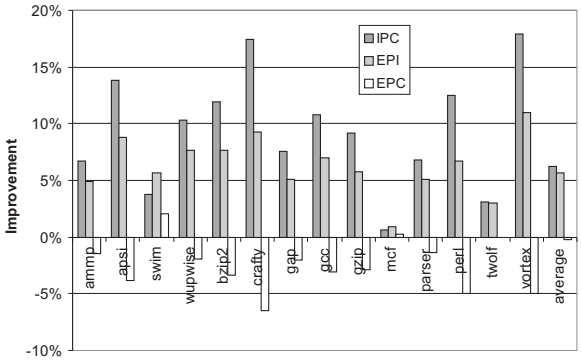

Figure 4: Reduction of performance (IPC), energy per instruction (EPI) and energy per cycle (EPC) obtained by adding a 1-issue by-pass pipeline to the baseline architecture.

benchmarks are compiled for the Alpha ISA using the native cc compiler with optimization flags "-fast" and static linking. Representative simulation intervals of 500M instructions are determined using SimPoint [23].

Performance and energy consumption are estimated using the sim-flex[1] simulator. The simulator is modified and configured to model a future deeply pipelined processor (Table 1). Important configuration settings for this work are the minimum dispatch-execute delay of 5 cycles and the reasonable issue width of four instructions per cycle. The issue queue can hold 32 instructions while the instruction window totals 96 instructions.

Power and energy numbers are listed for the conditional clock gating style ("cc3") with the default sim-flex technological parameters. The power model is extended to take all modifications related to the by-pass pipeline into account. We have modeled the long-living register file as well as the addition of a write-port to the physical register file. Other energy savings result from a reduction of activity factors.

Metrics are averaged across benchmarks as if all benchmark traces are concatenated and the metrics are computed over the combined trace. Thus, per-instruction metrics (EPI) are averaged using the arithmetic mean and per-cycle metrics (IPC, EPC) are averaged using the weighted harmonic mean where the weights are proportional to the cycle count of each benchmark.

4.1 Impact on Performance, Energy and Power

Figure 4 shows the improvement of IPC, EPI and EPC when adding the by-pass pipeline to our baseline processor configuration. The issue width in the by-pass pipeline is 1. The by-pass pipeline increases IPC by 6.3% on average, with peaks up to 17.9%. IPC increases due to the additional issue width and execution unit available in the by-pass pipeline.

The mcf benchmark is waiting for memory most of the time, so it does not react to our optimizations. If we exclude mcf, then the average speedup becomes 8.9%.

We measure energy consumption by the EPI (energy per committed instruction) metric. The by-pass pipeline reduces EPI by 5.7%.

Power is measured by the EPC metric (energy per cycle). Adding the by-pass pipeline to the baseline architecture slightly increases average EPC. Note that the average EPC is mainly dominated by the swim and mcf benchmarks, as these have a significantly higher cycle count than the other benchmarks.

Although the general trend is to increase EPC, it is important to note that the real merit of the by-pass pipeline is the possibility to simplify the out-of-order pipeline and save energy and power while retaining performance. This trade-off is explored in Section 4.4.

[1] `http://www.ece.cmu.edu/~simflex`

Table 2: Analysis of by-passing instructions by operation type.

Benchmark	%insn bypass	alu	branch	load addr.	store addr.
ammp	17.2%	5.3%	0.5%	11.2%	0.2%
apsi	24.4%	11.4%	1.6%	10.8%	0.6%
swim	37.0%	16.5%	1.0%	18.4%	1.1%
wupwise	26.0%	14.1%	4.1%	7.6%	0.2%
bzip2	23.6%	13.0%	2.6%	7.4%	0.7%
crafty	26.9%	12.5%	2.9%	11.2%	0.2%
gap	18.2%	9.7%	2.1%	5.6%	0.8%
gcc	25.3%	11.7%	3.3%	8.5%	1.9%
gzip	25.2%	12.3%	4.0%	7.6%	1.3%
mcf	18.0%	5.7%	8.7%	2.8%	0.8%
parser	21.2%	9.7%	4.7%	6.1%	0.6%
perl	19.0%	10.1%	2.1%	6.3%	0.5%
twolf	17.6%	10.8%	2.7%	4.0%	0.2%
vortex	29.9%	15.6%	4.0%	7.7%	2.5%
average	23.4%	11.2%	3.3%	8.1%	0.8%

Table 3: Reduction of activity factor in critical pipeline stages.

Benchmark	fetch	dispatch	reg. file read	write-back	IQ fill
ammp	0.6%	17.3%	6.6%	12.8%	2.1%
apsi	0.9%	24.4%	6.3%	17.9%	14.8%
swim	0.2%	37.1%	5.8%	27.6%	26.0%
wupwise	2.3%	26.0%	1.9%	20.4%	14.6%
bzip2	2.4%	22.7%	1.2%	17.4%	17.0%
crafty	1.8%	26.2%	7.0%	19.9%	17.5%
gap	3.1%	18.0%	1.2%	13.5%	5.6%
gcc	3.5%	24.9%	-0.6%	18.0%	20.0%
gzip	-1.4%	23.0%	4.6%	18.5%	15.3%
mcf	0.3%	17.8%	6.6%	14.1%	1.9%
parser	1.3%	20.2%	-0.4%	15.5%	7.9%
perl	1.4%	18.2%	-0.4%	13.5%	6.2%
twolf	1.0%	16.9%	2.6%	13.2%	6.0%
vortex	9.0%	29.9%	-5.9%	21.8%	30.0%
average	1.8%	22.9%	3.0%	17.4%	9.3%

4.2 Speedup: Analysis

Table 2 shows the percentage of dispatched instructions that are sent to the by-pass pipeline. On average, 23.4% of the dispatched instructions execute in the by-pass pipeline, which is very close to the maximum achievable of 25% reported in Figure 1.

We break-down the by-passing instructions as branch instructions, load address computations, store address computations and other ALU operations (Table 2). Half of all by-passing instructions are common ALU operations.

Table 4: Percentage energy reduction of by-passing. Column EPI shows overall EPI reduction. The other columns show the energy reduction for particular units.

Bench-mark	EPI	branch pred.	inst. cache	register rename	issue queue	ld/st queue	register file	data cache	alu	write-back	clock
ammp	4.9%	4.2%	8.3%	0.2%	15.9%	2.0%	6.7%	2.0%	1.2%	13.5%	5.0%
apsi	8.8%	8.6%	15.4%	-0.3%	21.1%	2.4%	6.4%	3.3%	1.6%	18.6%	10.4%
swim	5.6%	3.5%	11.0%	-0.2%	27.1%	1.3%	1.7%	0.9%	0.3%	23.2%	5.8%
wupwise	7.6%	4.8%	13.9%	-0.2%	22.3%	2.8%	1.9%	3.8%	1.4%	19.0%	7.9%
bzip2	7.7%	5.3%	12.6%	-0.9%	19.9%	0.7%	2.2%	6.1%	0.5%	17.9%	7.7%
crafty	9.3%	7.6%	16.2%	-0.6%	23.2%	2.8%	7.7%	3.2%	0.9%	19.7%	9.6%
gap	5.1%	3.7%	11.2%	-0.5%	15.5%	2.3%	0.1%	2.2%	1.5%	13.5%	4.3%
gcc	7.0%	4.4%	11.0%	0.0%	20.8%	0.8%	0.3%	4.4%	0.0%	18.7%	7.1%
gzip	5.8%	5.4%	11.1%	-2.1%	21.1%	1.0%	3.8%	2.0%	-0.3%	16.8%	5.0%
mcf	0.9%	0.5%	5.2%	-0.5%	9.1%	0.0%	-5.1%	-0.2%	0.2%	4.1%	0.9%
parser	5.1%	3.1%	7.6%	-0.7%	17.6%	1.0%	0.6%	2.5%	-0.1%	13.9%	5.6%
perl	6.7%	4.6%	11.4%	-0.8%	16.5%	1.4%	1.5%	4.4%	1.3%	14.8%	6.7%
twolf	3.0%	1.8%	5.7%	-0.7%	14.5%	0.5%	0.7%	0.6%	-1.1%	11.5%	3.1%
vortex	11.0%	6.2%	17.0%	0.0%	25.9%	2.3%	-2.8%	7.7%	3.1%	23.1%	11.3%
average	5.7%	3.6%	10.5%	-0.6%	18.9%	1.3%	1.1%	2.7%	0.6%	15.7%	5.8%

Branch instructions executing on the by-pass pipeline contribute to just 3.3% of all dispatched instructions. This low number is not unexpected: many branches are data-dependent on load instructions [27]. These branches cannot execute on the by-pass pipeline because the load is typically still executing when the branch dispatches.

Branch mispredictions may be detected in the by-pass pipeline, but this happens very infrequently. In fact, disallowing branch instructions to execute on the by-pass pipeline has a very negligible impact on performance and power.

Load address computations appear frequently in the by-pass pipeline. On average, 8.1% of all dispatched instructions are by-passed load address computations. This corresponds to 32% of the load address computations. In contrast, store address computations are rarely executed early: only 0.8% of all instructions are by-passed store address computations. A possible explanation for this trend is the saving and restoring of registers on the stack. When saving registers on the stack, the store address computations are dependent on the instruction that previously reserved stack space. Hence, they may not be eligible for execution on the by-pass pipeline. When the registers are restored, the stack pointer was probably not recently modified, so it is much more likely that the load address computations execute on the by-pass pipeline.

Note that scheduling, wake-up and forwarding energy related to load/store instructions depends on architectural details. The simulated architecture splits every load/store instruction into an address generation micro-op and a cache access micro-op. Each of these micro-ops must be scheduled independently. Other architectures may implement a dedicated address generation unit that is tightly coupled to the cache. Load/store instructions are now scheduled as a whole. It is not necessary to explicitly wake-up the cache access micro-op or to forward the computed address outside of the address generation unit. On the other hand, when the by-pass pipeline is present, the difference in energy consumption between the two architectures decreases, allowing the architect to select the implementation to optimize other constraints.

4.3 Energy: Analysis

Energy-efficiency is improved by executing an important fraction of all instructions in the by-pass pipeline, which is more energy-efficient than the out-of-order execution pipeline. Table 3 shows a reduction of activity factor in the critical pipeline stages. We observe a small reduction of instruction fetch activity, which is the consequence of detecting branch mispredictions early in the by-pass pipeline.

The other columns in Table 3 relate to the out-of-order pipeline. The number of instructions dispatched into this pipeline is reduced by 22.9% (note that no-ops are never dispatched in this architecture), register file reads are reduced by 3.0% and write-back (and wake-up) traffic is reduced by 17.4%. Finally, the average occupation of the issue queue is reduced between 2% and 30%, which may be further exploited by dynamic issue queue scaling techniques [2, 5, 9].

The strong reduction of activity factors directly translates into a reduction of energy consumption as shown in Table 4. The pipeline stages showing strong reduction in activity consume less energy.

Instruction cache energy is reduced by 10.5% on average. This energy savings is due in part to a decrease in the instruction cache activity factor (Table 3) and in part to an increase in the average size of fetch groups. The fetch groups become larger because the issue rate is increased and because dispatch stalls less frequently for a full issue queue. Hereby, the instruction groups leaving the decode pipeline become larger, so the instruction groups entering the decode pipeline can be larger too. When fetch groups are larger, fewer instruction cache accesses are necessary to fetch the same total number of instructions. Consequently, the instruction cache is idle more often and energy is saved.

The long-living register file is an additional source of energy consumption: it consumes one third the power of the physical register file. Note, however, that this constitutes less than 0.5% of the total processor power.

We have measured that on average just 10% (between 6% and 16%) of all write-backs are copied to the long-living register file, so there is an opportunity for saving write ports on this register file. A register copied to the long-living register

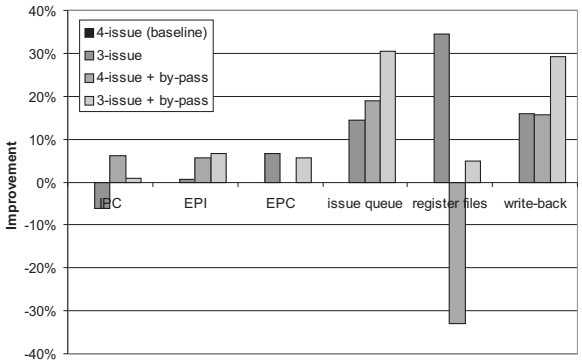

Figure 5: Improvement of IPC, EPI, EPC and energy consumption in critical components with respect to the baseline 4-issue processor when trading-off issue width in the out-of-order pipeline for issue width in the by-pass pipeline.

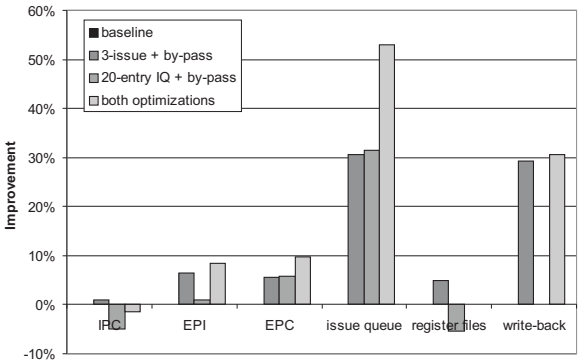

Figure 6: Improvement of IPC, EPI, EPC and energy consumption in critical components with respect to the baseline 4-issue processor when trading-off the issue width and the issue queue size.

file is used 0.93 times on average, so there is opportunity to filter write bandwidth further. The majority of values stored in the long-living register file are produced inside the by-pass pipeline itself (53% on average).

4.4 Trading-off Performance vs. Power

Adding the by-pass pipeline to the processor allows to further trade-off power against performance. The by-pass pipeline reduces the execution time by 6.3% and the EPI by 5.7%. Overall, EPC remains largely the same (Figure 5, bar "4-issue+by-pass"). However, as most power is consumed in the out-of-order pipeline, reducing the issue width of that pipeline will reduce power consumption strongly, while a possible performance loss is covered by the by-pass pipeline.

We evaluate reducing the issue width of the out-of-order pipeline from 4 to 3. We assume that a 3-issue out-of-order pipeline also saves 1 cycle of the dispatch-schedule delay, as the wake-up and select logic is less complex and the physical register file has fewer ports. Assuming this trade-off, the 3-issue processor is 6.2% slower than the baseline 4-issue processor, its EPI is the same and its EPC is 6.7% less (Figure 5, bar "3-issue"). However, the 3-issue processor with by-pass pipeline is as fast as the 4-issue processor (1% faster), its EPI is reduced by 6.6% and its EPC is reduced by 5.6%. Thus, it is possible to improve all of IPC, EPI and EPC by adding a simple in-order by-pass pipeline and reducing out-of-order issue width. The energy savings are realized mostly in the issue queue (lower select complexity), register file (fewer access ports) and in the write-back stage (lower wake-up complexity).

We observed above that the issue queue occupation is drastically reduced up to 30% (Table 3). Thus it is possible to reduce the issue queue size without sacrificing performance. Figure 6 shows the result of this trade-off. The bars "3-issue + by-pass" recapitulate the previous trade-off of the issue width. The bars "20-entry IQ + by-pass" show the result of reducing the issue queue from 32 to 20 entries and adding the by-pass pipeline. This trade-off is beneficial for IPC, EPI and EPC and it is more beneficial than the issue-width trade-off. Energy is saved in the issue queue and the write-back stage (wake-up logic).

EPI and EPC are further reduced when combining both trade-offs (reducing issue width and issue queue size). IPC suffers slightly by 1.5% compared to the baseline processor. However, EPI is reduced by 8.4% and EPC is reduced by 9.7%. Strong energy savings are obtained in the issue queue (53%), the register file (33%) and the write-back stage (31%).

4.5 An Alternative Implementation

We have illustrated the energy-efficiency of the by-pass pipeline by copying registers to the long-living register file. Using the by-pass pipeline, the issue width of the out-of-order core can be reduced by one.

An alternative implementation is possible where the long-living register file is left out and the by-pass pipeline is integrated into the out-of-order core. In this implementation, the by-pass pipeline can be seen as one of four lanes in the 4-issue out-of-order core.

The major modifications of this implementation with respect to the baseline architecture are summarized as follows. One issue slot is reserved to instructions that are ready at dispatch, which would be otherwise dispatched to the by-pass pipeline. These instructions can execute on just one execution unit, which is reserved exclusively for this type of instructions. The execution unit takes its operands either from the register file, as an immediate, or from its own forwarding data path. The result of this execution unit is forwarded to the other functional units only if the dispatch-execute delay of the architecture is one cycle. Instructions executing on this lane do not wake up dependent instructions in the issue queue; they are marked ready at dispatch.

There are no fundamental reasons why performance (IPC) of the two possible implementations should be different. Only energy consumption differs between the two implementations for several reasons. Leaving out the long living register file is beneficial for static power. On the other hand, accessing the larger physical register file may consume more dynamic power than accessing the smaller long-living register file. Also, integrating both pipelines may increase average wire length compared to having two separate pipelines. Note also that the write bandwidth to the long-living register file may be reduced (Section 4.3), resulting in less energy consumption for write-backs and for reads from the long-living register file than reported in this paper. Accurately evaluating these concerns is out of the scope of our tools and should be performed in the context of a specific architecture.

5. CONCLUSION

This paper considers the addition of a simple energy-efficient by-pass pipeline to an out-of-order superscalar processor to increase energy-efficiency. The by-pass pipeline relieves the out-of-order execution pipeline of 23% of its work, resulting in reduced activity factors in the out-of-order pipeline. We discussed the architectural implications of the by-pass pipeline and we evaluated its impact on performance and energy. In particular, we have shown a 6.3% speedup, while reducing the energy consumed per instruction (EPI) by 5.7%. The issue queue energy is reduced by 18.9% and write-back and wake-up energy by 15.7%.

The by-pass pipeline provides additional execution resources. The equivalent of these resources may be taken away from the out-of-order pipeline to further increase its energy-efficiency. Reducing both the issue width of the out-of-order pipeline and the issue queue size allows for the same performance level as the baseline processor, but issue queue energy is reduced by 53%, physical register file energy by 33% and write-back energy by 31%. This results in an overall energy savings of 8.4% for executing the same work and a reduction of energy consumed per cycle by 9.7%.

Acknowledgment

This research is sponsored in part by the Fund for Scientific Research-Flanders (FWO), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and Ghent University. Philippe Manet, Thibault Delavallée and Igor Loiselle are funded by the Walloon region.

6. REFERENCES

[1] M. D. Brown, J. Stark, and Y. N. Patt. Select-free instruction scheduling logic. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 204–213, 2001.

[2] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 147–156, June 2003.

[3] R. Canal and A. González. A low-complexity issue logic. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 327–335, 2000.

[4] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 37–46, 2002.

[5] D. Folegnani and A. González. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, June 2001.

[6] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 236–245, 1992.

[7] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 726–731, 1998.

[8] K.-S. Hsiao and C.-H. Chen. An efficient wakeup design for energy reduction in high-performance superscalar processors. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 353–360, 2005.

[9] T. Karkhanis, J. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *Intl. Symposium on Low Power Electronics and Design*, pages 178–183, Aug. 2002.

[10] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *The International Conference on Computer Design*, 1998.

[11] I. Kim and M. H. Lipasti. Half-price architecture. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 28–38, 2003.

[12] I. Kim and M. H. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 277, 2003.

[13] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, June 2002.

[14] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 132–141, June 1998.

[15] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 27–38, 2001.

[16] C. Molina, A. González, and J. Tubella. Dynamic removal of redundant computations. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 474–481, 1999.

[17] S. Önder and R. Gupta. Superscalar execution with direct data forwarding. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, 1998.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-1996-1328, Computer Sciences Department, University of Wisconsin-Madison, 1996.

[19] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, Denver, Colorado, June 1997.

[20] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 318–329, 2002.

[21] S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, Sun Microsystems, 1992.

[22] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, 2005.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[24] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20(05):70–83, 2000.

[25] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 36–44, 1985.

[26] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 194–205, 1997.

[27] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Conference on Microprogramming and Microarchitecture*, 1999.

[28] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 57–66, 2000.

[29] J. J. Yi and D. J. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, pages 462–465, 2002.