EOLE: Combining Static and Dynamic Scheduling through Value Prediction to Reduce Complexity and Increase Performance

Arthur Perais, André Seznec

EOLE: Combining Static and Dynamic Scheduling through Value Prediction to Reduce Complexity and Increase Performance

Arthur Perais, INRIA
André Seznec, INRIA

Recent work in the field of VP has shown that given an efficient confidence estimation mechanism, prediction validation could be removed from the out-of-order engine and delayed until commit time. As a result, a simple recovery mechanism – pipeline squashing – can be used, while the out-of-order engine remains mostly unmodified.

Yet, VP and validation at commit time require additional ports on the Physical Register File, potentially rendering the overall number of ports unbearable. Fortunately, VP also implies that many single-cycle ALU instructions have their operands predicted in the front-end and can be executed in-place, in-order. Similarly, the execution of single-cycle instructions whose result has been predicted can be delayed until commit time since predictions are validated at commit time.

Consequently, a significant number of instructions – 10% to 70% in our experiments – can bypass the out-of-order engine, allowing for a reduction of the issue width. This reduction paves the way for a truly practical implementation of Value Prediction. Furthermore, since Value Prediction in itself usually increases performance, our resulting {Early — Out-of-Order — Late} Execution architecture, EOLE, is often more efficient than a baseline VP-augmented 6-issue superscalar while having a significantly narrower 4-issue out-of-order engine.

CCS Concepts: •**Computer systems organization** → **Superscalar architectures;** *Complex instruction set computing; Pipeline computing;*

General Terms: Microarchitecture, Performance

Additional Key Words and Phrases: Value prediction, speculative execution, out-of-order execution, EOLE, VTAGE

ACM Reference Format:
Arthur Perais, André Seznec, 2015. EOLE: Combining Static and Dynamic Scheduling through Value Prediction to Reduce Complexity and Increase Performance. *ACM Trans. Comput. Syst.* ?, ?, Article XXXX (XXXX 2015), 34 pages.
DOI: 0000001.0000001

1. INTRODUCTION & MOTIVATIONS

Even in the multicore era, the need for higher single thread performance is driving the definition of new high-performance cores. Although the usual superscalar design does not scale, increasing the ability of the processor to extract *Instruction Level Parallelism* (ILP) by increasing the window size as well as the issue width has generally been the favored way to enhance sequential performance. For instance, consider the recently introduced Intel Haswell micro-architecture that has 33% more issue capacity than Intel Nehalem.[1] To accommodate this increase, both the *Reorder Buffer* (ROB)

[1]State-of-the-art in 2009

This work was partially supported by the European Research Council Advanced Grant DAL No. 267175.
 0734-2071/2015/XXXX-ARTXXXX $15.00
DOI: 0000001.0000001

and scheduler size were substantially increased.[2] On top of this, modern schedulers must support complex mechanisms such as *speculative scheduling* to enable back-to-back execution and thus *replay* or *selective replay* to efficiently recover from schedule mispredictions [Kim and Lipasti 2004].

In addition, the issue width impacts other structures: The *Physical Register File* (PRF) must provision more read/write ports as the width grows, while the number of physical registers must also increase to accommodate the ROB size. Because of this, both latency and power consumption increase and using a monolithic register file quickly becomes complexity-ineffective. Similarly, a wide-issue processor should provide enough functional units to limit resource contention. Yet, the complexity of the bypass network grows quadratically with the number of functional units and quickly becomes critical regarding cycle time [Palacharla et al. 1997]. In other words, the out-of-order engine impact on power consumption and cycle time is ever increasing [Ernst and Austin 2002].

In this paper, we propose a modified superscalar design, the {*Early — Out-of-Order — Late*} *Execution* microarchitecture, *EOLE*.[3] It is built on top of a Value Prediction (VP) pipeline. VP allows dependents to issue earlier than previously possible by using predicted operands, artificially increasing ILP. Yet, predictions must be verified to ensure correctness. Fortunately, Perais and Seznec observed that the cost of validating the predicted results (and recovering from mispredictions) at retirement can be absorbed provided an enhanced confidence estimation mechanism that yields a very high prediction accuracy [Perais and Seznec 2014b]. In other words, VP does not need to intervene in the execution engine, save for the PRF.

With EOLE, we leverage this observation to further reduce both the complexity of the out-of-order execution engine and the number of ports required on the PRF when VP is implemented. We achieve this reduction without significantly impacting overall performance. Our contribution is therefore twofold: First, EOLE paves the way to truly practical implementations of VP. Second, it reduces complexity in arguably the most complex and power-hungry part of a modern superscalar core.

EOLE relies on the fact that swhen using VP, a significant number of single-cycle instructions have their operands ready in the front-end thanks to the value predictor. As such, we introduce *Early Execution* to execute single-cycle ALU instructions in-order in parallel with Rename by using predicted and/or immediate operands. Early-executed instructions are **not** sent to the out-of-order scheduler. Moreover, delaying VP validation until commit time removes the need for *selective replay* and enforces a complete pipeline squash on a value misprediction. This guarantees that the operands of committed early executed instructions were the correct operands. Early Execution requires simple hardware and reduces pressure on the out-of-order instruction window.

Similarly, since predicted results can be validated outside the out-of-order engine at commit time [Perais and Seznec 2014b], we can offload the execution of predicted single-cycle ALU instructions to some dedicated in-order *Late Execution* pre-commit stage, where no *Select & Wakeup* has to take place. This does not hurt performance since instructions dependent on predicted instructions will simply use the predicted results rather than wait in the out-of-order scheduler. Similarly, the resolution of high confidence branches can be offloaded to the Late Execution stage since they are very rarely mispredicted.

Overall, a total of 10% to 70% of the retired instructions can be offloaded from the out-of-order engine. As a result, EOLE benefits from both the aggressiveness of modern

[2]From respectively 128 and 36 entries to 192 and 60 entries.
[3]*Eole* is the french name of *Aeolus*, the ruler of the winds in Greek mythology.

out-of-order designs and the higher energy-efficiency of more conservative in-order designs.

We evaluate EOLE against a baseline out-of-order model featuring VP and show that it achieves similar levels of performance having only 66% of the baseline issue capacity and a significantly less complex physical register file. This is especially interesting since it provides architects extra design headroom in the out-of-order engine to implement new architectural features.

The remainder of this paper is organized as follows. Section 2 discusses related work and provides some background on Value Prediction. Section 3 details the EOLE microarchitecture, which implements both Early and Late Execution by leveraging Value Prediction. Section 4 describes our simulation framework while Section 5 presents experimental results. Section 6 focuses on the qualitative gains in complexity and power consumption permitted by EOLE. Finally, Section 7 provides concluding remarks and directions for future research.

2. RELATED WORK

Many propositions aim at reducing complexity in modern superscalar designs. In particular, it has been shown that most of the complexity and power consumption reside in the out-of-order engine, including the PRF [Wallace and Bagherzadeh 1996], scheduler and bypass network [Palacharla et al. 1997]. As such, previous studies focused either in devising new pipeline organizations or reducing the complexity of existing structures.

2.1. Alternative Pipeline Organizations

[Farkas et al. 1997] propose the *Multicluster* architecture in which execution is distributed among several execution clusters, each of them having its own register file. Since each cluster is simpler, cycle time can be decreased even though some inefficiencies are introduced due to inter-cluster dependencies. In [Farkas et al. 1997], inter-cluster data dependencies are handled by dispatching the same instruction to several clusters and having all instances but one serve as inter-cluster data-transfer instructions while a single instance actually computes the result. To enforce correctness, this instance is data-dependent on all others. The Alpha 21264 [Kessler et al. 1998] is an example of real-world clustered architecture and shares many traits with the *Multicluster* architecture.

[Palacharla et al. 1997] introduce a *dependence-based* microarchitecture where the centralized instruction window is replaced by several parallel FIFOs. FIFOs are filled with independent instructions (i.e., all instructions in a single FIFO are dependent) to allow ILP to be extracted: since instructions at the head of the FIFOs are generally independent, they can issue in parallel. This greatly reduces complexity since only the head of each FIFO has to be scanned by the *Select* logic. They also study a *clustered dependence-based* architecture to reduce the amount of bypass and window logic by using clustering. That is, a few FIFOs are grouped together and assigned their own copy of the register file and bypass network, mimicking a wide issue window by having several smaller ones. Inter-cluster bypassing is naturally slower than intra-cluster bypassing.

[Tseng and Patt 2008] propose the *Braid* architecture, which shares many similarities with the *clustered dependence-based* architecture with three major differences: 1) Instruction partitioning – steering – is done at compile time via dataflow-graph coloring 2) Each FIFO is a narrow-issue (dual-issue in [Tseng and Patt 2008]) in-order cluster called a *Braid Execution Unit* with its own local register file, execution units, and bypass network 3) A global register file handles inter-unit dependencies instead of an inter-cluster bypass network. As such, they obtain performance on par with an accord-

ingly sized out-of-order machine without using complicated scheduling logic, register file and bypass logic.

[Austin 1999] proposes *Dynamic Implementation VAlidation* (DIVA) to check instruction results just before commit time, allowing the core to be faulty. That is, DIVA can greatly reduce the complexity of *functionally validating* the core since only the checker has to be functionally correct. The core complexity itself can remain very high, however. An interesting observation is that the latency of the checker has very limited impact on performance. This hints that adding pipeline stages between Writeback and Commit (such as value prediction validation) should not impact performance much.

[Fahs et al. 2005] study *Continuous Optimization* where common compile-time optimizations are applied dynamically in the Rename stage. This allows to *early execute* some instructions in the front-end instead of the out-of-order engine. Similarly, [Petric et al. 2005] propose RENO which also dynamically applies optimizations at rename-time.

2.2. Decreasing the Complexity of Implemented Mechanisms

Instead of studying new organizations of the pipeline, [Kim and Lipasti 2003] present the *Half-Price Architecture*. They argue that many instructions are single-operand and that both operands of dual-operands instructions rarely become ready at the same time. Thus, the load capacitance on the tag broadcast bus can be greatly reduced by *sequentially waking-up* operands. In particular, the left operand is woken-up as usual, while the right one is woken-up one cycle later by inserting a latch on the broadcast bus. This scheme relies on an operand criticality predictor to place the critical tag in the *left operand* field.

Similarly, [Ernst and Austin 2002] propose *Tag Elimination* to limit the number of comparators used for *Wakeup*. In particular, only the tag of the last arriving operand (predicted as such) is put in the scheduler. This allows to only use one comparator per entry instead of several (one per operand). However, this also implies that the instruction must be replayed on a wrong last operand prediction.

Regarding the Physical Register File (PRF), [Kim and Lipasti 2003] also observe that many issuing instructions do not need to read both their operands in the PRF since one or both will be available on the bypass network. Thus, provisioning two read ports per issue slot is generally over-provisioning. In particular, for the small proportion of instructions that indeed need to read two operands in the PRF, they propose to do two consecutive accesses using a single port. Reducing the number of ports drastically reduces the complexity of the register file as ports are much more expensive than registers.

Lastly, [Lukefahr et al. 2012] propose to implement two back-ends – in-order and out-of-order – in a single core and to dynamically dispatch instructions to the most adapted one. In most cases, this saves power at a slight cost in performance. In a sense, EOLE has similarities with such a design since instructions can be executed in different locations. However, no decision has to be made regarding the location where an instruction will be executed, since this only depends only on the instruction type and status (e.g., predicted or not).

Note that our proposal is orthogonal to all these contributions since it only impacts the number of instructions that enters the out-of-order execution engine.

Value Prediction. EOLE builds upon the broad spectrum of research on Value Prediction independently initiated by [Gabbay and Mendelson 1998; Lipasti and Shen 1996].

[Sazeides and Smith 1997] refine the taxonomy of VP by categorizing predictors. They define two classes of value predictors: *Computational* and *Context-based*. The

former generate a prediction by applying a function to the value(s) produced by the previous instance(s) of the instruction. For example, the Stride predictor [Mendelson and Gabbay 1997] and the *2-Delta* Stride predictor [Eickemeyer and Vassiliadis 1993] use the addition of a constant (stride).

On the other hand, the latter – *Context-Based* predictors – rely on patterns in the value history of a given static instruction to generate predictions, e.g., the Finite Context Method (FCM) predictors [Sazeides and Smith 1997].

Most of the initial studies on Value Prediction either assume that recovering from a value misprediction induces almost no penalty [Lipasti and Shen 1996; Lipasti et al. 1996; Zhou et al. 2003], or simply focus on accuracy and coverage rather than speedup [Goeman et al. 2001; Nakra et al. 1999; Rychlik et al. 1998; Sazeides and Smith 1997; Thomas and Franklin 2001; Wang and Franklin 1997]. The latter studies were essentially ignoring the performance loss associated with misprediction recovery.

In a recent study, [Perais and Seznec 2014b] show that all value predictors are amenable to very high accuracy at a reasonable cost in both prediction coverage and hardware storage. This allows to delay prediction validation until commit time, removing the burden of implementing a complex replay mechanism that is tightly coupled to the out-of-order engine [Kim and Lipasti 2004]. As such, the out-of-order engine remains mostly untouched by VP. This proposition is crucial as Value Prediction was usually considered very hard to implement in part due to the need for a very fast recovery mechanism.

In the same paper, the VTAGE context-based predictor is introduced. As the ITTAGE indirect branch predictor [Seznec and Michaud 2006], VTAGE uses the global branch history to select predictions, meaning that it does not require the previous value to predict the current one. This is a strong advantage since conventional value predictors usually need to track inflight predictions as they require the last value to predict.

Finally, [Perais and Seznec 2015] propose a tightly-coupled hybrid of a VTAGE predictor and a Stride-based predictor. Similarly to the *Differential* FCM predictor of [Goeman et al. 2001], the *Differential* VTAGE predictor (D-VTAGE) implements a *Last Value Table* (LVT) containing the last outcome of each static instruction, to which a stride is added to compute the prediction. The stride is retrieved using the TAGE/VTAGE indexing scheme. D-VTAGE has been shown to outperform a similarly sized D-FCM predictor as well as a more simple hybrid selecting between VTAGE and 2-delta Stride based on confidence. A practical implementation of the speculative window required to track inflight last outcomes is also provided in [Perais and Seznec 2015].

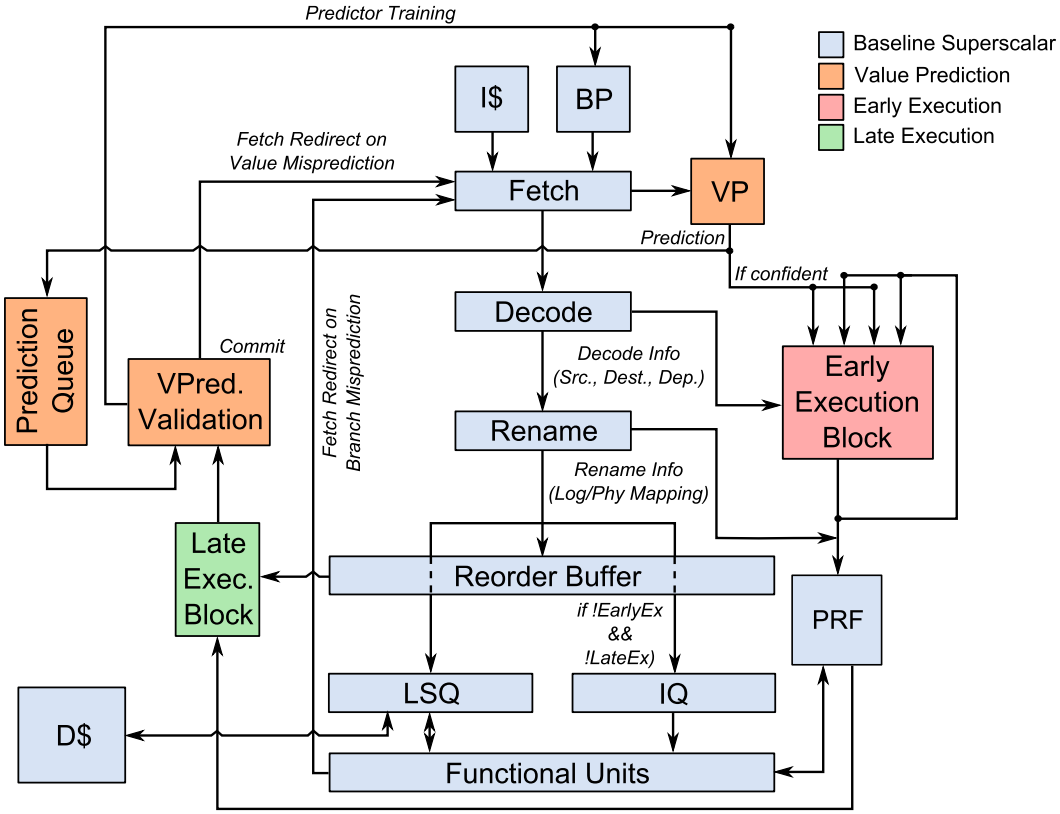
3. EOLE

3.1. Enabling EOLE through Value Prediction

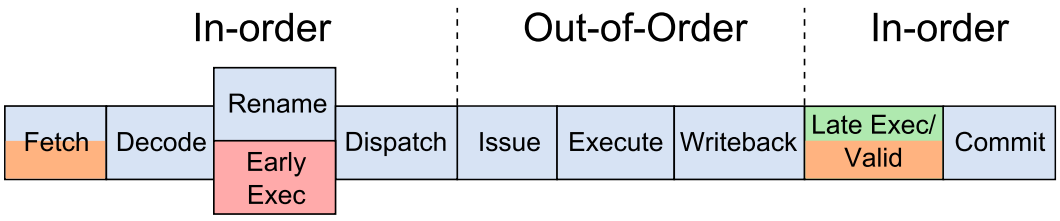
As previously described, EOLE consists of a set of simple ALUs in the in-order front-end to early-execute instructions in parallel with Rename, and a second set in the in-order back-end to late-execute instructions just before they are committed.

While EOLE is heavily dependent on Value Prediction, they are in fact complementary features. Indeed, the former needs a value predictor to predict operands for Early Execution and provide temporal slack for Late Execution, while Value Prediction needs EOLE to reduce PRF complexity and thus become truly practical.

Moreover, to be implemented, EOLE requires prediction validation to be done at commit since validating at Execute mechanically forbids Late Execution. In addition, using *selective replay* to recover from a value misprediction nullifies the interest of both Early and Late Execution as all instructions must flow through the out-of-order scheduler in case they need to be replayed [Kim and Lipasti 2004]. Hence, *squashing*



(a) Block diagram.



(b) Pipeline diagram.

Fig. 1: The EOLE μ-architecture.

must be used to recover from a misprediction so that early/late executed instructions can safely bypass the scheduler.

Fortunately, [Perais and Seznec 2014b] have proposed a confidence estimation mechanism greatly limiting the number of value mispredictions, *Forward Probabilistic Counters* (FPC). With FPC, the cost of a single misprediction can be high since mispredicting is very rare. Thus, validation can be done late – at commit time – and squashing can be used as the recovery mechanism. This enables the implementation of both Early and Late Execution, hence EOLE.

By eliminating the need to dispatch and execute many instructions in the out-of-order engine, EOLE substantially reduces the pressure put on complex and power-hungry structures. Thus, those structures may be scaled down, yielding a less complex

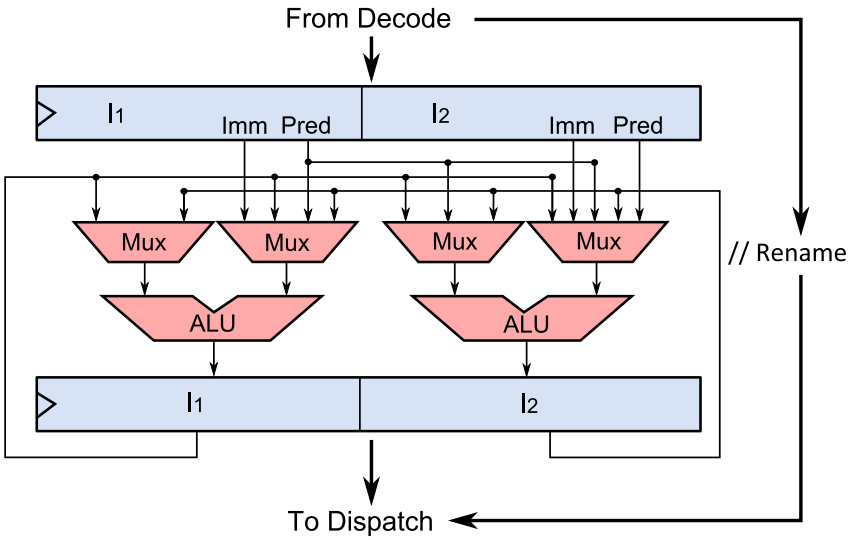


Fig. 2: Early Execution Block. The logic controlling the ALUs and muxes is not shown for clarity.

architecture whose performance is on par with a more aggressive design. Moreover, doing so is orthogonal to previously proposed mechanisms such as *clustering* [Farkas et al. 1997; Kessler et al. 1998; Palacharla et al. 1997; Seznec et al.] and does not *require* a centralized instruction window, even though this is the model we use in this paper. Figure 1 depicts the EOLE architecture, implementing both Early Execution (red), Late Execution (green) and Value Prediction (orange). In the following paragraphs, we detail the two additional blocks required to implement EOLE and their interactions with the rest of the pipeline.

3.2. Early Execution Hardware

The core idea of Early Execution (EE) is to position one or more ALU stages in the front-end in which instructions with available operands will be executed. For complexity concerns, however, it seems necessary to limit Early Execution to single-cycle ALU instructions. Indeed, implementing complex functional units in the front-end to execute multi-cycle instructions does not appear as a worthy tradeoff. In particular, memory instructions are not early executed. EE is done in-order, hence, it does not require renamed registers and can take place in parallel with Rename. For instance, Figure 2 depicts the Early Execution Block adapted to a 2-wide Rename stage.

Renaming is often pipelined over several cycles. Consequently, we can use several ALU stages and simply insert pipeline registers between each stage. The actual execution of an instruction can then happen in any of the ALU stages, depending on the readiness of its operands coming from *Decode* (i.e., immediate), the local[4] bypass network (i.e., from instructions early executed in the previous cycle) or the value predictor. Operands are **never** read from the PRF.

[4]For complexity concerns, we consider that bypass does not span several ALU stages. Consequently, if an instruction depends on a result computed by an instruction located two rename-groups ahead, it will not be early executed.

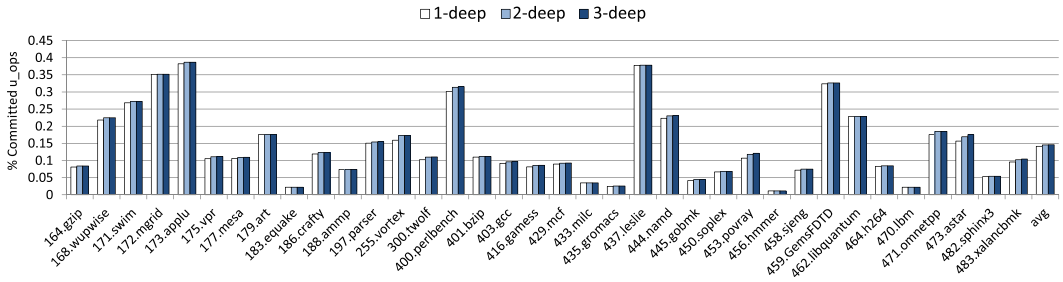


Fig. 3: Proportion of committed instructions that can be early executed, using one or two ALU stages and a D-VTAGE hybrid predictor (later described in Section 4).

In a nutshell, all eligible instructions flow through the ALU stages, propagating their results in each bypass network accordingly once they have executed. Finally, after the last stage, results as well as predictions are written into the PRF.

An interesting design concern lies with the number of stages required to capture a reasonable proportion of instructions. We actually found that using more than a single stage was highly inefficient, as illustrated in Figure 3. This Figure shows the proportion of committed instructions eligible for Early Execution for a baseline 8-wide rename, 6-issue model (see Table I in Section 4), using the D-VTAGE value predictor (later described in Table II, Section 4). As a result, in further experiments, we consider a 1-deep Early Execution Block only.

To summarize, Early Execution only requires a single new block, which is shown in red in Figure 1. The mechanism we propose does not require any storage area for temporaries as all values are living inside the pipeline registers or the bypass network(s). Finally, since we execute in-order, each instruction is mapped to a single ALU and scheduling is straightforward (as long as there are as many ALUs as the rename width).

3.3. Late Execution Hardware

Late Execution (LE) targets instructions whose result has been predicted. Instructions eligible for prediction are μ-ops producing a 64-bit or less result that can be read by a subsequent μ-op, including non-architectural temporary registers, as defined by the ISA implementation. Unless mentioned otherwise, vector and scalar SIMD instructions are not predicted.

Late Execution intervenes just before prediction validation time, that is, out of the execution engine. As for Early Execution, we limit ourselves to single-cycle ALU instructions to minimize complexity. That is, predicted loads are executed in the out-of-order engine, but validated at commit.

Interestingly, [Seznec 2011] showed that conditional branch predictions flowing from TAGE can be categorized such that very high confidence predictions are known. Since high confidence branches exhibit a misprediction rate generally lower than 0.5%, resolving them in the Late Execution block will have a marginal impact on overall performance. Thus, we consider both single-cycle predicted ALU instructions and very high confidence branches[5] for Late Execution. In this study, we did not try to set confidence on the other branches (indirect jumps, returns). Yet, provided a similar high confidence estimator for these categories of branches, one could postpone the resolution of high confidence ones until the LE stage.

[5]Predictions whose confidence counter is saturated [Seznec 2011].

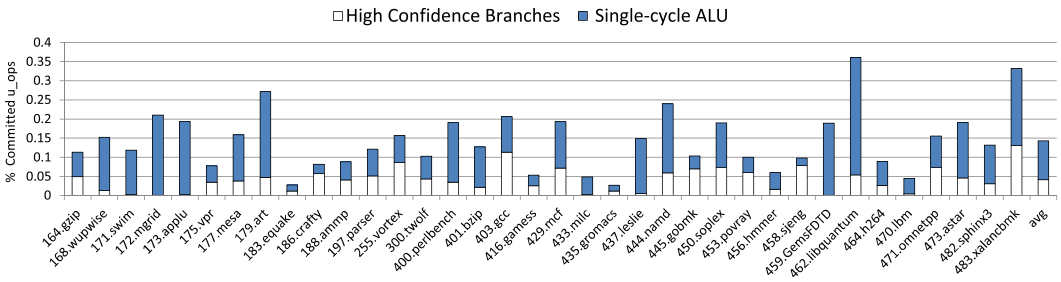


Fig. 4: Proportion of committed instructions that can be late executed using a D-VTAGE (see Section 4) hybrid predictor. Late executable instructions that can also be early executed are not counted since instructions are executed at most once.

Furthermore, note that predicted instructions can also be early executed. In that event, they only need to be validated in case another early executed instruction from the same rename-group used the prediction as an operand. That is, instructions are executed in a single location.

In any case, Late Execution further reduces pressure on the out-of-order engine in terms of instructions dispatched to the scheduler. As such, it also removes the need for value predicting only critical instructions [Fields et al. 2001; Rychlik et al. 1998; Tune et al. 2002] since minimizing the number of instructions flowing through the out-of-order engine requires maximizing the number of predicted instructions. Hence, predictions considered as useless from a performance standpoint become useful in EOLE. Figure 4 shows the proportion of committed single-cycle ALU instructions that can be late executed using a baseline 6-issue processor with a D-VTAGE predictor (respectively described in Tables I and II in Section 4).

Late Execution needs to implement *commit width* ALUs and the associated read ports in the PRF. If an instruction I_1 to be late executed depends on the result of instruction I_0 of the same commit group that will also be late executed, it does not need to wait as it can use the predicted result of I_0. In other words, all non executed instructions reaching the Late Execution stage have all their operands ready in the PRF, as in DIVA [Austin 1999]. Due to the need to validate predictions (including reading results to train the value predictor) as well as late-execute some instructions, at least one extra pipeline stage after Writeback is likely to be required in EOLE. In the remainder of this paper, we refer to this stage as the Late Execution/Validation and Training (LE/VT) stage.

Overall, the hardware needed for LE is fairly simple, as suggested by the high-level view of a 2-wide LE Block shown in Figure 5. It does not even require a bypass network. In further experiments, we consider that LE and prediction validation can be done in the same cycle, before the Commit stage. EOLE is therefore only one cycle longer than the baseline superscalar it is compared to. While this may be optimistic due to the need to read from the PRF, this only impacts the value misprediction penalty, the pipeline fill delay, and ROB occupancy. In particular, since low confidence branches are resolved in the same cycle as for the baseline, the average branch misprediction penalty will remain very similar. Lastly, as a first step, we also consider that enough ALUs are implemented (i.e., as many as the commit-width). As a second step, we shall consider reduced-width Late Execution.

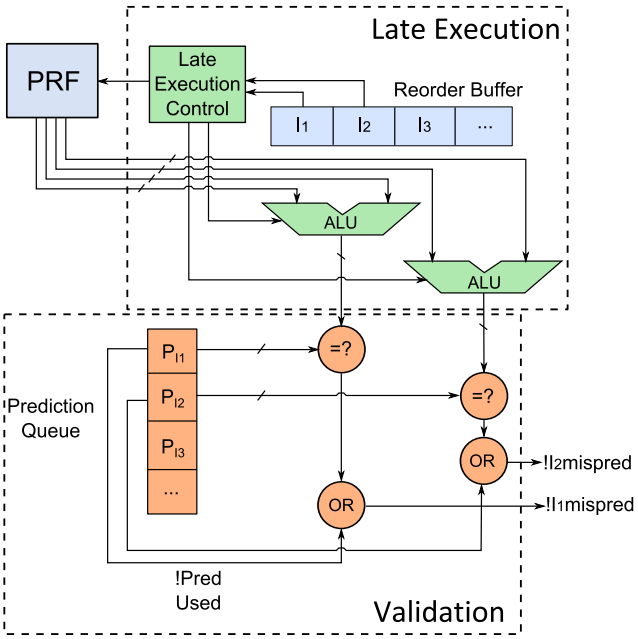


Fig. 5: Late Execution Block for a 2-wide processor. The top part can late-execute two instructions while the bottom part validates two results against their respective predictions. Buses are *general purpose register-width*-bit wide.

3.4. Potential out-of-order Engine Offload

3.4.1. Baseline. We obtain the ratio of retired instructions that can be offloaded from the out-of-order engine for each benchmark by summing the columns in Figure 3 and 4 (both sets are disjoint as we only count late executable instructions that cannot also be early executed), and adding *load immediate* instructions. Those numbers are reported in Figure 6 where we distinguish four types of instructions:

(1) Early executed because it is a *load immediate*, i.e., its operand is necessarily ready at Rename.[6] In a regular Value Prediction pipeline, these instructions can also bypass the out-of-order engine as the immediate can be considered as an always correct value prediction. This includes loading an immediate to a floating point register.
(2) Early executed because it is a ready single-cycle ALU instruction.
(3) Late executed because it is a high-confidence branch, as predicted by the TAGE branch predictor.
(4) Late executed because it is a value predicted single-cycle ALU instruction.

We observe that the ratio is very dependent on the application, ranging from less than 10% for *equake, milc* and *lbm* to more than 50% for *swim, mgrid, applu, perlbench, leslie, GemsFDTD, xalancbmk* and up to 60% for *namd* and 70% for *libquantum*. It generally represents a significant part of the retired instructions in most cases (around 35% on average).

[6]Except for load immediate to a partial register.

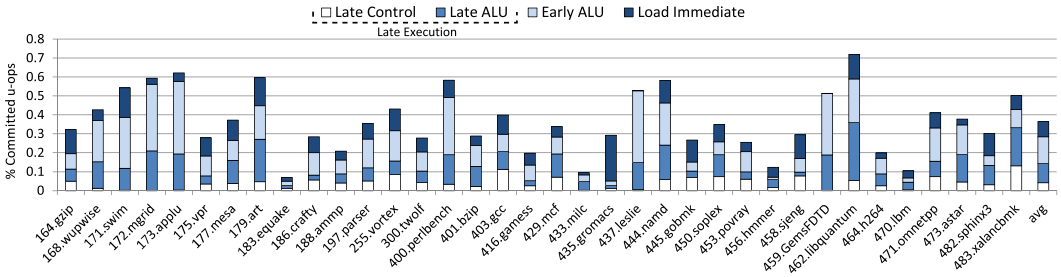


Fig. 6: Proportion of dynamic instructions that can bypass the execution engine by being a high-confidence branch, a value predicted ALU instruction, an ALU instruction with its operands ready in the frontend or a load immediate instruction.

3.4.2. Limit Study. We also consider numbers when we do not specify any constraint regarding the type of instructions that can be early or late executed. In particular, all load and complex arithmetic (e.g., multiply, divide) instructions are considered for bypassing the execution engine. We also consider floating-point (FP) operations and packed vector operations (the shortcomings of actually predicting those types of instructions are discussed in the next Section).

Figure 7 shows the overall potential either without FP/SIMD prediction (but still allowing loads and complex integer instructions to be early or late executed, second set of bars) and with FP/SIMD prediction (third set of bars). The first set of bars recalls the numbers of Figure 6.

The second set of bars (no FP/SIMD prediction, but all instructions are eligible for early/late execution) shows that in most case, allowing all instructions to be early/late executed significantly increases the proportion of instructions that can bypass the execution engine. However, this increase is generally due to an increase in late-executable instructions (from 14.3% to 22.3% on average) rather than early-executable ones (from 14.1% to 14.5% on average). Moreover, in most cases, the bulk of the additional candidates for late execution consists of value predicted loads. Indeed, on average, loads represent 26.2% of the predicted instructions versus 1.56% for other types of instructions (multiplications, divisions and conversions from float to int) that are now allowed to be early/late executed. As a result, allowing loads to be late executed would be of great help to further reduce pressure on the out-of-order engine.

The third set of bars (FP/SIMD prediction, all instructions are eligible for early/late execution) suggests that although the value predictor is able to predict a significant amount of instructions writing to xmm registers for a few benchmarks (e.g., *equake, milc* and *soplex*), the average increase in early/late-executable instructions is much lower than the one we observed by allowing loads and complex integer instructions to be early/late executed.

4. EVALUATION METHODOLOGY

4.1. Simulator

We use the x86_64 ISA to validate EOLE, even though EOLE can be adapted to any general-purpose ISA without any more restrictions than the ones already existing for Value Prediction. We use a modified[7] version of the *gem5* cycle-level simulator [Binkert et al. 2011].

[7]Our modifications mostly lie with the ISA implementation. In particular, we implemented branches with a single μ-op instead of three and we removed some false dependencies existing between instructions due to the way flags are renamed/written.

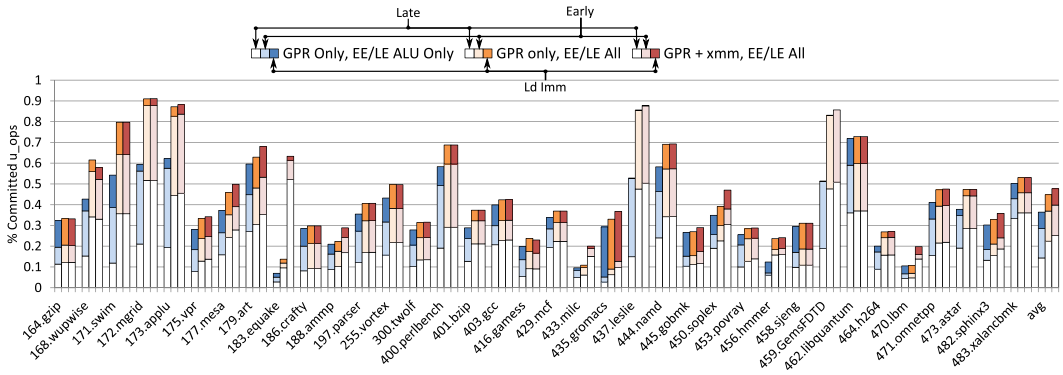


Fig. 7: Proportion of dynamic instructions that can bypass the execution engine, depending on what kind of instructions are value predicted and what kind of instructions are allowed to be early/late executed.

We consider a relatively aggressive 4GHz, 6-wide issue superscalar[8] baseline with a fetch-to-commit latency of 19 cycles. Since we focus on the out-of-order engine complexity, both in-order front-end and in-order back-end are overdimensioned to treat up to 8 μ-ops per cycle. We model a deep front-end (15 cycles) coupled to a shallow back-end (3 cycles) to obtain realistic branch/value misprediction penalties.

Table I describes the characteristics of the baseline pipeline we use in more details. In particular, the out-of-order engine is dimensioned with a unified centralized 60-entry scheduler (a.k.a. instruction queue or IQ) and a 192-entry Reorder Buffer (ROB) on par with Haswell's, the latest commercially available Intel microarchitecture. We refer to this baseline as the *Baseline_6_60* configuration (6-issue, 60-entry IQ). Functional units are grouped by stacks, to mimick the behavior of Haswell's issue ports as best as possible. Note that there are 8 stacks, but since we consider a 6-issue processor, at most 6 stacks will be assigned an instruction each cycle.

In that context, reducing the issue width can be achieved in two fashions. First, by simply removing some issue ports. The associated FUs can then be redistributed to other ports, or be totally discarded from the design. Second, by keeping the same FU stacks, but by only allowing new_issue_width instructions to be issued per cycle. An actual implementation will likely do the former, however, since finding the optimal mix of FUs per issue port is beyond the scope of this paper, we keep the same FU stacks for all the different issue widths we consider in our experiments.

As μ-ops are known at Fetch in *gem5*, all the widths given in Table I are in μ-ops, even for the Fetch stage. Independent memory instructions (as predicted by the Store Sets predictor [Chrysos and Emer 1998]) are allowed to issue out-of-order. Entries in the scheduler are released upon issue, except for load instructions, which release their entry at Writeback.

In the case where Value Prediction is used, we add a pre-commit stage responsible for validation/training and Late Execution when relevant : the LE/VT stage. This accounts for an additional pipeline cycle (20 cycles) and an increased value misprediction penalty (21 cycles min.). Minimum branch misprediction latency remains unchanged except for mispredicted very high confidence branches when EOLE is used. Note that

[8] On our benchmark set and with our baseline simulator, an 8-issue machine achieves only marginal speedup over this baseline. Hence we consider an issue width of 6 to ensure that reducing it will noticeably decrease performance.

Table I: Simulator configuration overview. *not pipelined. Additional hardware for EOLE is shown in bold.

Front End	L1I 8-way 32KB, 1 cycle, perfect ITLB; 8-wide Fetch within two 16-byte blocks each cycle, potentially over one taken branch TAGE 1+12 components 15k-entry total ($\simeq$ 32KB) [Seznec and Michaud 2006], 20 cycles min. branch mis. penalty, 2-way 8K-entry BTB, 32-entry RAS 8-wide Decode 8-wide Rename **+ 8-wide Early Execution**
Execution	8-wide Dispatch to: 192-entry Reorder Buffer, 60-entry unified scheduler (IQ), 72-entry Load Queue, 48-entry Store Queue 4K-entry SSID/LFST Store Sets [Chrysos and Emer 1998] memory dependency predictor 256/256 INT/FP – 64/128-bit SIMD registers are allocated from the pool of 64-bit FP registers Each cycle, issue one instruction to 6 ports out of the following 8: Port0: ALU(1c) $\otimes$ Mul(3c) $\otimes$ Div(25c*) $\otimes$ FP Mul(5c) $\otimes$ FP Div(10c*) $\otimes$ 128-bit SIMD FP Mul(5c)Div(25c*) $\otimes$ 128-bit SIMD INT Mul(3c) Port1: ALU(1c) $\otimes$ FP(3c) $\otimes$ 128-bit SIMD ALU (1c INT, 3c FP) Port2: Ld/Str Port3: Ld/Str Port4: ALU(1c) Port6: ALU(1c) Port7: Str 8-wide Writeback 8-wide Validation (only when VP is used) **+ 8-wide Late Execution** 8-wide Commit
Caches	L1D 8-way 32KB, 4 cycles, 64 MSHRs, 2 reads and 2 writes/cycle, perfect D-TLB Unified L2 16-way 1MB, 12 cycles, 64 MSHRs, no port constraints L2 Stride prefetcher, degree 8 All caches have 64B lines and LRU replacement
Memory	Single channel DDR3-1600 (11-11-11), 2 ranks, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Min. Read Lat.: 75 cycles, Max. 185 cycles.

the value predictor is really trained after Commit, but the value is read from the PRF in the LE/VT stage.

Shortcomings of the Model. Contrarily to modern x86 implementations, *gem5* does not support *move elimination* [Fahs et al. 2005; Jourdan et al. 1998; Petric et al. 2005], μ*-op fusion* [Gochman et al. 2003] and does not implement a *stack-engine* [Gochman et al. 2003]. It also lacks *macro-op fusion*, *zero-idioms elimination* (e.g., *xor rax, rax*), as well as a μ*-op cache* [Intel 2014].

Although breaking true data dependencies through VP can overlap with some of these features (e.g., zero-idioms elimination and stack engine), most of them are orthogonal, therefore, the inherent increase in ILP brought by VP should not generally be shadowed by said optimizations. In other words, the pipeline we model is sufficiently resembling a modern superscalar processor to illustrate how VP can push performance higher and how it can help reduce the aggressiveness of the execution engine.

Another limitation regards prefetching. Indeed, one could argue that implementing a better prefetcher (we use a simple stride-based prefetcher in the LLC) could greatly reduce the gains brought by VP, through transforming a predicted load that misses in the LLC into a predicted load that hits in the L1.

While this is true and a better prefetcher may overlap with VP, on our benchmark set, we found that on average, only 5% of the loads that miss in the LLC are value predicted. This suggests that even if prefetching could capture all value predicted loads missing in the LLC, VP would still be able to bring significant performance improvements.

In addition, we point out that correctly predicting a L1 hit would still potentially save 4 cycles in Haswell, but would more importantly allow more freedom regarding speculative scheduling [Kim and Lipasti 2004]. That is, since load dependents can use the prediction to execute, there is no need to schedule them speculatively assuming the load will hit and no bank conflict will take place in the L1 (if the L1 is banked). As a result, scheduling replays due to L1 misses and L1 bank conflicts (hit in L1 but the cache bank is busy) can be avoided if the aforementioned load is value-predicted.

4.2. Value Predictor Operation

The predictor makes a prediction at fetch time for every eligible μ-op (we define eligible in the next paragraph). To index the predictor, we XOR the PC of the x86_64 instruction with the μ-op number inside the x86_64 instruction. This avoids all μ-ops mapping to the same entry for x86 instructions generating more than one μ-op. We assume that the predictor can deliver as many predictions as requested by the Fetch stage.

In previous work, a prediction is written into the PRF and replaced by its non-speculative counterpart when it is computed in the out-of-order engine [Perais and Seznec 2014b]. In parallel, predictions are put in a FIFO queue to be able to validate them – in-order – at commit time. In EOLE, we also use a queue for validation. However, instead of directly writing predictions to the PRF, we place predictions in the Early Execution units, which will in turn write the predictions to the PRF at Dispatch. By doing so, we can use predictions as operands in the EE units.

Eligible μ-ops. Since the predictor is able to produce 64-bit values, all μ-ops writing to a 64-bit (or less) General Purpose Register (GPR) are predicted, including instructions that convert a floating-point value to an integer value (e.g., *Convert Single Scalar FP to Signed D/Qword, CVTSS2SI*).

In addition, *gem5-x86* splits 128-bit packed instructions into two 64-bit (32-bit packed or 64-bit scalar) μ-ops.[9] As a result, it appears possible to predict floating-point results (both scalar and vector), as well as packed integer results, and in our first experiments, we consider doing so. However, cracking packed instructions into several μ-ops is not representative of a high-performance implementation. Thus, predicting packed 128-bit (or more) registers would in fact involve looking-up several predictor entries for the same μ-op, or provision 128 bits (or more) in each predictor entry, which may not be practical. Consequently, our final experiments do not consider packed instructions as a target for value prediction.

Nonetheless, scalar floating-point operations produce a 64-bit (or 32-bit for single-precision FP) value, therefore, it should be possible to predict them. Unfortunately, both encodings (legacy and VEX since AVX) of the x86 FPU, SSE, require that a scalar operation on a SIMD register copies the upper part of the first source xmm register to

[9]Both can be executed in a single cycle in our model, hence execution throughput is still 128 bits per cycle for packed instructions.

the destination xmm register.[10] For instance, a scalar double-precision floating-point addition can be expressed by the following algorithm, assuming 256-bit wide ymm SIMD registers[11] (as implemented in Haswell [Intel 2013]):

Legacy:
dest[63:0] = dest[63:0] + src[63:0]
dest[255:64] = dest[255:64] (unmodified)

VEX-encoded:
dest[63:0] = src1[63:0] + src2[63:0]
dest[127:64] = src1[127:64]
dest[255:128] = 0

Consequently, although the functional unit computes a 64-bit value, the need to merge the physical destination register with one physical source register yields an effective result width of 256 bits (Legacy) and 128 bits (VEX-encoded). Hence, except for scalar loads (which zero the upper part of both xmm and ymm), it is not useful to predict scalar floating point results,[12] because the RAW dependency is not broken by the prediction.

Note however that this limitation is purely inherent to the ISA, and that it would be possible to overcome it at the microarchitectural level, for instance by allowing different parts of a single ymm/xmm register to be renamed to different physical registers, and injecting a *merge* μ-op where appropriate.[13] Since this would allow to eliminate many copies of the upper part of xmm registers, this might in fact be an optimization that is already implemented. In that event, predicting scalar FP instructions would be possible.

To summarize, while we consider predicting all results (i.e., scalar/vector integer and scalar/vector floating-point) as a first step, to gauge the potential for additional performance and the predictability of FP results, we will ultimately consider predicting values that are written to a general purpose register only, as predicting scalar FP operations requires microarchitectural support due to ISA limitations, and FP load coverage tends to be very low, on our benchmark set.

x86 Flags. In the x86_64 ISA, some instructions write flags based on their results while some need them to execute (e.g., conditional branches) [Intel 2013]. We assume that flags are computed as the last step of Value Prediction, based on the predicted value. In particular, the *Zero Flag* (ZF), *Sign Flag* (SF) and *Parity Flag* (PF) can easily be inferred from the predicted result. Remaining flags – *Carry Flag* (CF), *Adjust Flag* (AF) and *Overflow Flag* (OF) – depend on the operands and cannot be inferred from the predicted result only. We found that always setting the *Overflow Flag* to 0 did not cause many mispredictions and that setting CF if SF was set was a reasonable approximation. The *Adjust Flag*, however, cannot be set to 0 or 1 in the general case. This is a major impediment to the value predictor coverage since we consider a prediction as incorrect if one of the derived flags – thus the flag register – is wrong. Fortunately, x86_64 forbids the use of decimal arithmetic instructions. As such, AF is not used and we can simply ignore its correctness when checking for a misprediction [Intel 2013].

[10] In the legacy encoding, the first source is also the destination register, but the copy of the upper part of the old physical register to the new one is still required.
[11] 128-bit xmm registers correspond to the lower 128 bits of 256-bit ymm registers.
[12] At least for SSE/AVX, which is the current FPU ISA for x86, x87 being deprecated [Intel 2013].
[13] This is also a way to handle x86 general purpose partial registers.

Predictor	#Entries	Tag	Size (KB)
D-VTAGE [Perais and Seznec 2015]	8192 (Base) 6×1024 (Tagged)	- 12 + $rank$	131.0 62.6

Table II: Layout Summary. $rank$ is the position of the tagged component and varies from 1 to 6, 1 being the component using the shortest history length.

Predictor Considered in this Study. In this study, we focus on the tightly-coupled hybrid of VTAGE and a simple stride-based Stride predictor, D-VTAGE [Perais and Seznec 2015]. For confidence estimation, we use Forward Probabilistic Counters [Perais and Seznec 2014b]. In particular, we use 3-bit confidence counters whose forward transitions are controlled by the vector $v = \{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$ as we found it to perform best with D-VTAGE. Those counters allow to push accuracy very high while only requiring 3-bit per counter (The Linear Feedback Shift Register used to provide randomness is amortized on the whole predictor, or a large number of entries). This is required to absorb the cost of validating predictions at Commit and squashing to recover.

We consider an 8K-entry base predictor and 6 1K-entry partially tagged components. We do not try to optimize the size of the predictor (by using partial strides, for instance), but it has been shown that 16 to 32KB of storage are sufficient to get good performance with D-VTAGE [Perais and Seznec 2015].

4.3. Benchmarks

We use a subset of the the SPEC'00 [Standard Performance Evaluation Corporation 2000] and SPEC'06 [Standard Performance Evaluation Corporation 2006] suites to evaluate our contribution as we focus on single-thread performance. Specifically, we use 18 integer benchmarks and 18 floating-point programs.[14] Table III summarizes the benchmarks we use as well as their input, which are part of the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in the benchmark using Simpoint 3.2 [Perelman et al. 2003]. We simulate the resulting slice in two steps: First, warm up all structures (caches, branch predictor and value predictor) for 50M instructions, then collect statistics for 100M instructions.

Note that the baseline simulator configuration we use in this work is significantly different from the configuration used in [Perais and Seznec 2014a]: Fetch is less aggressive, the last level cache is smaller, the first level data cache is slower, and much less complex functional units and loads ports are implemented. Thus, a more realistic pipeline configuration is simulated, but the IPC in some benchmarks is much lower (e.g., *vpr* has 0.664 vs. 1.326 and *art* has 0.441 vs. 1.211). On the contrary, since we use a bigger LQ (72-entry vs 48-entry in [Perais and Seznec 2014a]) to better fit the Intel Haswell pipeline configuration, and since we use a bigger and better implementation of the Store Sets memory dependency predictor, IPC is higher in other benchmarks (e.g., *gamess* has 2.196 vs. 1.929 and *parser* has 0.872 vs. 0.544). In almost all cases, the gains come from the better memory dependency predictor, and we actually found that in the version of *gem5* used in [Perais and Seznec 2014a], the PC of memory instructions is right-shifted by 2 before accessing the table, accommodating 4-byte aligned RISC but not x86's CISC.

[14] We do not use the whole suites due to some currently missing system calls or instructions in *gem5-x86*.

Table III: Benchmarks used for evaluation. Top: CPU2000, Bottom: CPU2006. INT: 18, FP: 18, Total: 36.

Program	Input	IPC
164.gzip (INT)	input.source 60	0.835
168.wupwise (FP)	wupwise.in	1.337
171.swim (FP)	swim.in	2.206
172.mgrid (FP)	mgrid.in	2.356
173.applu (FP)	applu.in	1.481
175.vpr (INT)	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	0.664
177.mesa (FP)	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm	1.268
179.art (FP)	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	0.441
183.equake (FP)	inp.in	0.655
186.crafty (INT)	crafty.in	1.551
188.ammp (FP)	ammp.in	1.212
197.parser (INT)	ref.in 2.1.dict -batch	0.872
255.vortex (INT)	lendian1.raw	1.795
300.twolf (INT)	ref	0.468
400.perlbench (INT)	-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1	1.370
401.bzip2 (INT)	input.source 280	0.780
403.gcc (INT)	166.i	1.028
416.gamess (FP)	cytosine.2.config	2.196
429.mcf (INT)	inp.in	0.116
433.milc (FP)	su3imp.in	0.499
435.gromacs (FP)	-silent -deffnm gromacs -nice 0	0.792
437.leslie3d (FP)	leslie3d.in	2.139
444.namd (FP)	namd.input	2.347
445.gobmk (INT)	13x13.tst	0.845
450.soplex (FP)	-s1 -e -m45000 pds-50.mps	0.271
453.povray (FP)	SPEC-benchmark-ref.ini	1.519
456.hmmer (INT)	nph3.hmm	2.016
458.sjeng (INT)	ref.txt	1.302
459.GemsFDTD (FP)	/	2.146
462.libquantum (INT)	1397 8	0.460
464.h264ref (INT)	foreman_ref_encoder_baseline.cfg	1.127
470.lbm (FP)	reference.dat	0.373
471.omnetpp (INT)	omnetpp.ini	0.309
473.astar (INT)	BigLakes2048.cfg	1.166
482.sphinx3 (FP)	ctlfile . args.an4	0.787
483.xalancbmk (INT)	-v t5.xml xalanc.xsl	1.934

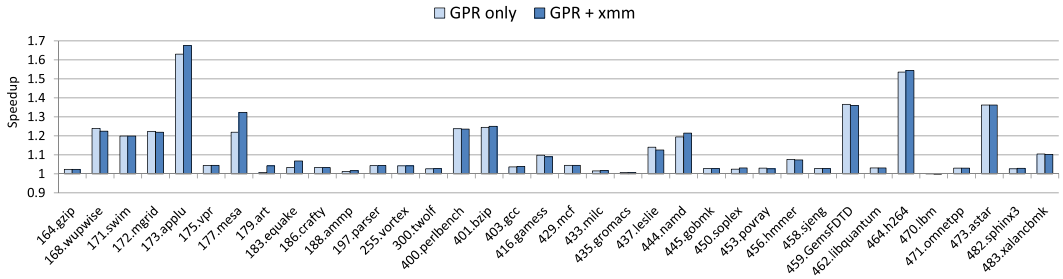
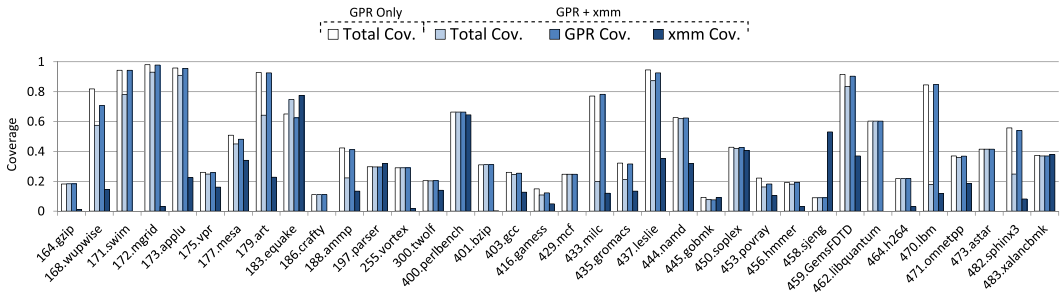
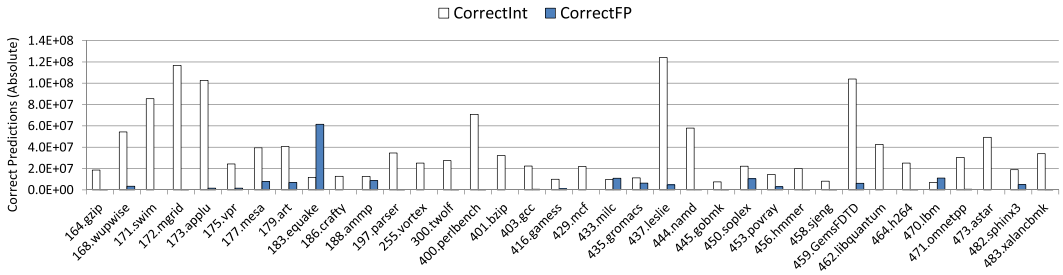


Fig. 8: Speedup over *Baseline_6_60* brought by Value Prediction using D-VTAGE. Instructions eligible for VP are either those writing a GPR register (GPR), or all instructions writing a register (GPR + xmm).



(a) Coverage



(b) Absolute number of correctly predicted instructions

Fig. 9: Relative and absolute coverage of the D-VTAGE predictor.

5. EXPERIMENTAL RESULTS

In our experiments, we first use *Baseline_6_60* as the baseline to gauge the impact of adding a value predictor only. Then, in all subsequent experiments, we use *Baseline_6_60* augmented with the predictor presented in Table II as our performance baseline. We refer to it as the *Baseline_VP_6_60* configuration. Our objective is to characterize the potential of EOLE at decreasing the complexity of the out-of-order engine. We assume that the Early and Late Execution stages are able to treat any group of up to 8 consecutive μ-ops every cycle. In Section 6, we will consider tradeoffs to enable realistic implementations.

5.1. Performance of Value Prediction

Figure 8 illustrates the performance benefit of augmenting the baseline processor with D-VTAGE. The first set of bars consider a predictor that only speculates on the value of general purpose registers. A few benchmarks present interesting potential e.g., *applu, GemsFDTD, h264* and *astar*, some a more moderate potential e.g., *wupwise, swim, mgrid, mesa, perlbench, bzip, namd* and *xalancbmk* and a few others low potential. No slowdown is observed.

In the second set of bars, the predictor is allowed to speculate on the results of instructions writing to 128-bit *xmm* registers, including integer and FP vector instructions. As previously mentioned, *gem5-x86* splits a 128-bit instruction into two 64-bit ones, therefore, from the point of view of the predictor, a single packed instruction is in fact two scalar (or 64-bit packed) μ-ops. In general, performance is comparable to the previous case, yet, in *applu, mesa, art, equake, namd, soplex* and *h264*, a slight performance increase can be observed, while a slight decrease can be seen in *wupwise, gamess* and *leslie*.

To gain some insight on why we observe such behavior, we consider the overall coverage of the value predictor, as well as the coverage for integer and FP instructions in Figure 9 (a). The first observation is that in general, FP coverage[15] is much lower than integer coverage. This is expected since a stride-based prediction scheme cannot naturally predict FP values other than constants. In some cases, however, FP coverage is higher (e.g., *equake, gobmk, sjeng* and *xalancbmk*). Yet, since coverage is a relative metric, it is not representative of how many instructions are actually predicted. As a result, Figure 9 (b) shows the absolute number of predicted instructions. Except in one benchmark (*equake*), the number of predicted FP instructions is often quite low, and in particular, lower than the number of integer predictions. As a result, in two out of the four previously mentioned cases where FP coverage is higher than INT coverage (*sjeng* and *xalancbmk*), the absolute number of predicted FP instructions is actually very low.

Regardless, benchmarks where performance increases slightly have a similar level of integer coverage as in the "integer prediction only" case, but they are also able to predict a small (*applu, art, h264, namd*) to moderate (*mesa, equake, soplex*) amount of FP instructions. On the contrary, when performance decreases, integer coverage is lower than in the "integer prediction only" case, and the small FP coverage does not make up for this reduction. This phenomenon can be explained by the fact that more static instructions have to share the PC-indexed Last Value Table (LVT) of D-VTAGE, hence potential decreases because more instructions collide in the LVT.

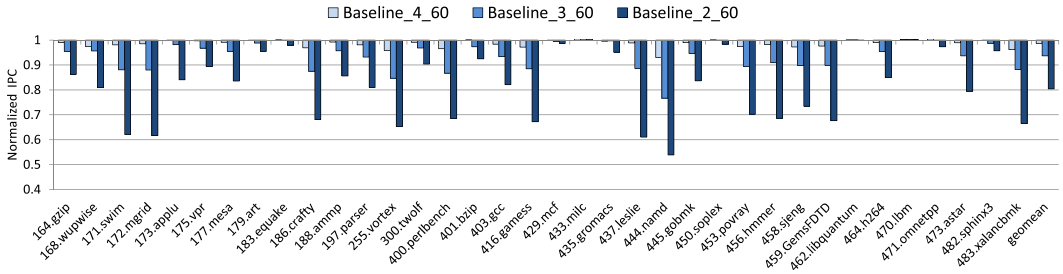
Nonetheless, it is interesting to note that in *equake*, many FP instructions are actually predictable (the majority being scalar single-precision additions and multiplications, but not loads from memory), hinting that although stride-based prediction does not match the floating-point representation, there are still some FP computations that show enough redundancies to benefit from having a value predictor.

5.2. Issue Width Impact on Processor Performance

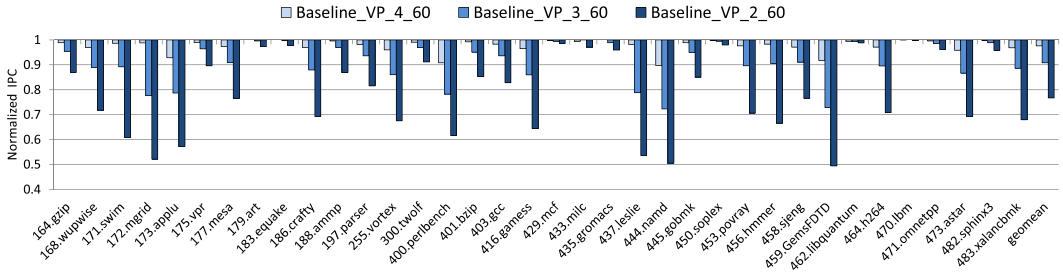
Baseline VP Model. We first depict the performance loss implied by reducing the issue-width on the baseline processor, *Baseline_6_60*, in Figure 10 (a). As the issue width is reduced, performance often decreases, but *Baseline_4_60* generally performs within 5% of *Baseline_6_60*, with a maximum slowdown of 7% in *namd*.

We also experimented with VP (*Baseline_VP_6/4/3/2_60*) in Figure 10 (b). It is quite clear that although the average performance of the 4-issue pipeline is close to

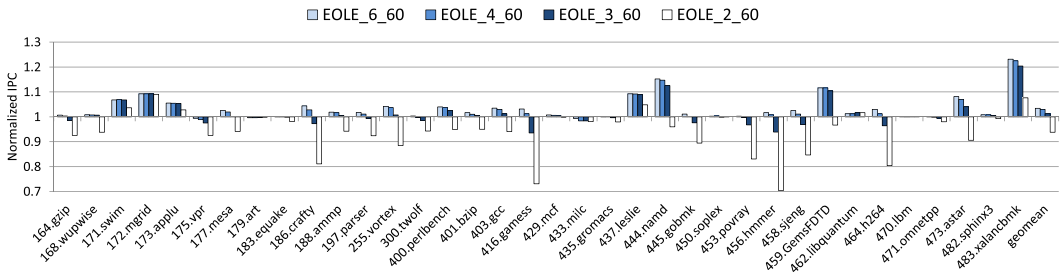
[15] FP coverage considers all instructions writing to xmm, including integer packed instructions.



(a) Baseline no-VP



(b) Baseline VP



(c) EOLE

Fig. 10: Impact of a reduced issue width on performance, normalized to *Baseline_6_60* in (a) and *Baseline_VP_6_60* in (b,c).

that of the 6-issue one, up to 10% performance is lost in *applu, perlbench, namd* and *GemsFDTD*. Moreover, reducing the issue width beyond 4 is clearly detrimental, as average performance is 90% that of the 6-issue pipeline for a 3-issue processor, and less than 80% for a 2-issue processor (with a maximum slowdown of 50% in *namd* and *GemsFDTD*).

As a result, the reduction in execution engine complexity comes at a noticeable cost in performance when VP is implemented. Without VP, less ILP can be extracted by the processor, and diminishing the issue width is less detrimental, although some benchmarks are still affected as soon as the issue width is decreased to 4.

EOLE. The performance of EOLE is illustrated in Figure 10 (c), using different out-of-order issue widths.

On top of the 6-issue VP pipeline, EOLE slightly increases performance over the baseline, with a few benchmarks achieving 5% speedup or higher (first bar). The particular case of *namd* is worth to be noted as with VP only, it would have benefited from an 8-issue core by more than 10%. Through EOLE, we actually increase the number of

instructions that can be executed each cycle, hence performance goes up in this benchmark.

For EOLE, reducing the issue width to 4 only decreases performance by a few percent compared with *EOLE_6_60*. Furthermore, *EOLE_4_60* still performs slightly higher than *Baseline_VP_6_60* in several benchmarks e.g., *swim, mgrid, applu, leslie, namd, GemsFDTD, astar* and *xalancmbk*. Slowdowns can be observed in *vpr* and *milc*, but remain modest, respectively 1.2% and 1.7%.

Further reduction of the issue width, however, begins to have noticeable impact on performance, with slowdowns of around 5% being observed in *gamess, povray, hmmer* and *h264* on the 3-issue pipeline. As a result, EOLE can be considered as a means to reduce issue width without significantly impacting performance on a processor featuring VP, as long as the execution engine remains wide enough (e.g., 4-issue).

5.3. Impact of Instruction Queue Size on Processor Performance

Baseline VP Model. In Figure 11 (a), we illustrate the performance loss inherent to the reduction in the number of instruction queue entries for the baseline model. Since VP is not present, ILP is often lower, hence instructions stay longer in the scheduler. This leads to performance decreasing noticeably even when only 6 IQ entries are removed (e.g., *gamess, leslie, namd, hmmer* and *GemsFDTD*), which is not desirable.

Figure 11 (b) shows results for the baseline VP pipeline. Since ILP is higher, instructions stay less longer in the IQ, and the performance drop is generally less significant than without VP. However, although the maximum slowdown is less than when the issue width is reduced, almost all benchmarks are slowed down even when only six entries are removed (first bar). On average (gmean), 2% performance is lost, with a maximum of 5.4% in *hmmer*. Further reducing the instruction queue size has even more detrimental effects, with an average performance loss of 4% and 8% for a 48-entry and a 42-entry IQ, respectively.

EOLE. With EOLE, the performance loss is much less pronounced in general, as shown in Figure 11 (c). Yet, even when only six entries are removed from the IQ, a 5% performance degradation is observed in *hmmer*, which is still problematic as our goal is to keep at least the same level of performance as the baseline VP model.

In practice, the benefit of EOLE is greatly influenced by the proportion of instructions that are not sent to the out-of-order engine. For instance *namd* needs a 60-entry IQ in the baseline case, but since it is an application for which many instructions are early or late executed, it can deal with a smaller IQ in EOLE.

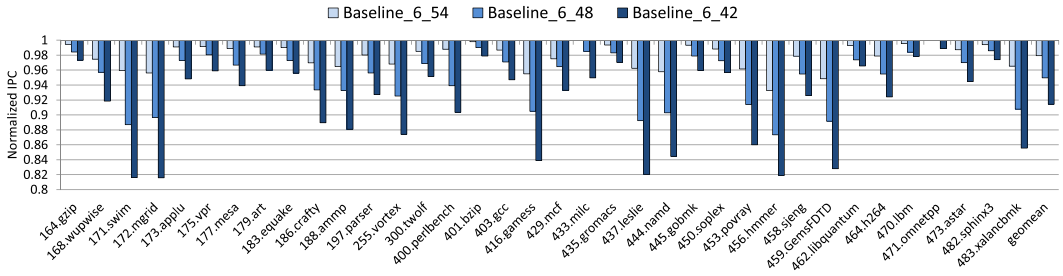
On the other hand, *hmmer*, the application that suffers the most from reducing the instruction queue size with EOLE, exhibits a relatively low coverage of predicted or early executed instructions.

5.4. Summary

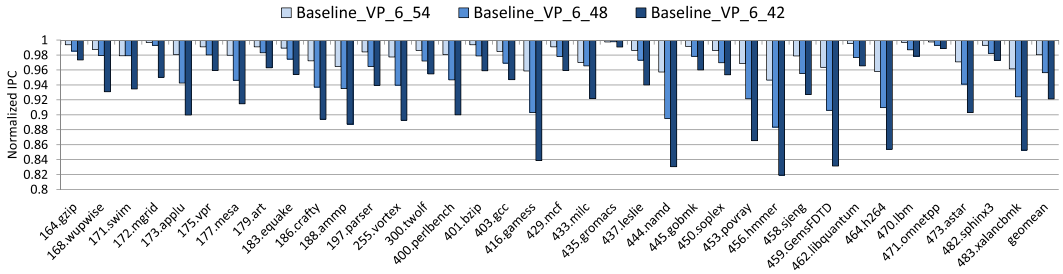
EOLE provides opportunities for either slightly improving the performance over a VP-augmented processor without increasing the complexity of the execution engine, or reaching the same level of performance with a significantly reduced execution engine complexity. Said complexity can be reduced via two means: reducing the issue width, and reducing the IQ size.

Reducing the issue width reduces the number of broadcast buses used for Wakeup as well as the complexity of the Select operation. Moreover, the maximum number of ports required on the PRF is reduced, and finally, the number of values co-existing in the bypass network each cycle is also reduced.

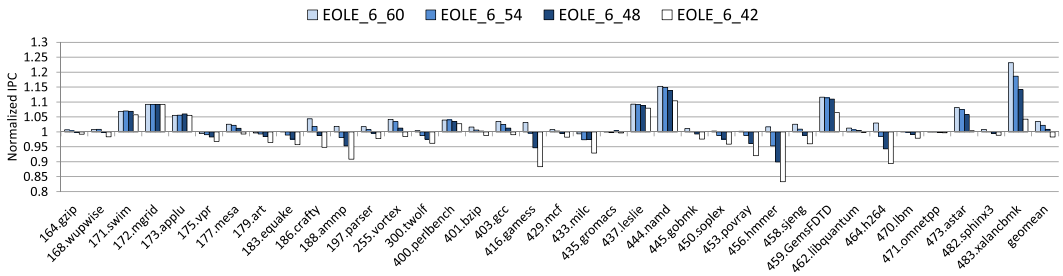
Reducing the number of IQ entries, on the contrary, does not reduce the number of broadcast buses, PRF ports and inflight values inside the bypass network. It only



(a) Baseline no-VP



(b) Baseline VP



(c) EOLE

Fig. 11: Impact of a reduced instruction queue on performance, normalized to *Baseline_6_60* in (a) and *Baseline_VP_6_60* in (b,c).

reduces the delay and the power spent in Wakeup by having shorter broadcast buses, as well as the Select delay since less entries have to be scanned [Palacharla et al. 1997]. Regardless, EOLE does not appear as efficient at mitigating the performance loss inherent to a smaller IQ, hence reducing the issue width appears as the better choice overall.

In the next section, we provide directions to limit the global hardware complexity and power consumption induced by the EOLE design and the overall integration of VP in a superscalar processor.

6. HARDWARE COMPLEXITY

In the previous section, we have shown that, provided that the processor already implements Value Prediction, adopting the EOLE design may allow to use a reduced-issue execution engine without impairing performance. Yet, extra complexity and power consumption are added in the Early Execution engine as well as the Late Execution engine.

In this section, we first describe the potential hardware simplifications on the execution engine enabled by EOLE. Then, we describe the extra hardware cost associated with the Early Execution and Late Execution engines. Finally, we provide directions to mitigate this extra cost. Note however that a precise evaluation would require a complete processor design and is beyond the scope of this paper.

6.1. Shrinking the Out-of-Order Engine

Out-of-Order Scheduler. Our experiments have shown that with EOLE, the out-of-order issue width can be reduced from 6 to 4 without significant performance loss, on our benchmark set. This would greatly impact *Wakeup* since the complexity of each IQ entry would be lower. Similarly, a narrower issue width mechanically simplifies *Select*. As such, both steps of the *Wakeup & Select* critical loop could be made faster and/or less power hungry.

Providing a way to reduce complexity with no impact on performance is also crucial because modern schedulers must support complex features such as *speculative scheduling* and thus *selective replay* to recover from scheduling mispredictions [Kim and Lipasti 2004; Perais et al. 2015].

Lastly, to our knowledge, most scheduler optimizations proposed in the literature can be added on top of EOLE. This includes the *Sequential Wakeup* of [Kim and Lipasti 2003] or the *Tag Elimination* of [Ernst and Austin 2002]. As a result, power consumption and cycle time could be further decreased.

Functional Units & Bypass Network. As the number of cycles required to read a register from the PRF increases, the bypass network becomes more crucial. It allows an instruction to "catch" its operands as they are produced and thus execute back-to-back with its producer(s). However, a full bypass network is very expensive, especially as the issue width – hence the number of functional units – increases. [Ahuja et al. 1995] showed that partial bypassing could greatly impede performance, even for a simple in-order single-issue pipeline. Consequently, in the context of a wide-issue out-of-order superscalar with a multi-cycle register read, missing bypass paths may cripple performance even more.

EOLE allows to reduce the issue width in the out-of-order engine. Therefore, it reduces the design complexity of a full bypass by reducing the number of simultaneous writers on the network.

A Limited Number of Register File Ports on the out-of-order Engine. Through reducing the issue width on the out-of-order engine, EOLE mechanically reduces the maximum number of read and write ports required on the PRF for regular out-of-order execution.

6.2. Extra Hardware Complexity Associated with Late/Early Execution

Cost of the Late Execution Block. The extra hardware complexity associated with Late Execution consists of three main components. First, for validation at commit time, a prediction queue (FIFO) is required to store predicted results. This component is needed anyway as soon as VP associated with validation at commit time is implemented, since the prediction must be stored until it can be compared against the actual result at Commit.

Second, ALUs are needed for Late Execution. Lastly, the operands for the late executed instructions must be read from the PRF. Similarly, the result of VP-eligible instructions must be read from the PRF for validation (predicted instructions only) and predictor training (all VP-eligible instructions).

In the simulations presented in Section 5, we have assumed that up to 8 μ-ops (i.e. *commit-width*) could be late executed per cycle. This would necessitate 8 ALUs

and up to 16 read ports on the PRF (including ports required for validation and predictor training).

Cost of the Early Execution Block. A single stage of simple ALUs is sufficient to capture most of the potential benefits of Early Execution. The main hardware cost associated with Early Execution is this stage of ALUs and the associated full bypass. Additionally, the predicted and early computed results must be written on the register file.

Therefore, in our case, a complete 8-wide Early Execution stage necessitates 8 ALUs, a full 8-to-8 bypass network and 8 write ports on the PRF.

The Physical Register File. From the above analysis, an EOLE-enhanced core featuring a 4-issue out-of-order engine (*EOLE_4_60*) would have to implement a PRF with a total of 12 write ports (resp. 8 for Early Execution and 4 for regular execution) and 24 read ports (resp. 8 for regular execution and 16 for late execution, validation and training).

The area cost of a register file is approximately proportional to $(R + W) * (R + 2W)$, R and W respectively being the number of read and write ports [Zyuban and Kogge 1998]. That is, at equal number of registers, the area cost of the EOLE PRF would be 4 times the initial area cost of the 6-issue baseline without value prediction (*Baseline_6_60*) PRF. Moreover, this would also translate in largely increased power consumption and access time, thus impairing cycle time and/or lengthening the register file access pipeline.

Without any optimization, *Baseline_VP_6_60* would necessitate 14 write ports (resp. 8 to write predictions and 6 for the out-of-order engine) and 20 read ports (resp. 8 for validation/training and 12 for the out-of-order engine), i.e., slightly less than *EOLE_4_60*. In both cases, this overhead might be considered as prohibitive in terms of silicon area, power consumption and access time.

Fortunately, simple solutions can be devised to reduce the overall cost of the PRF and the global hardware cost of Early/Late Execution without significantly impacting global performance. These solutions apply for EOLE as well as for a baseline implementation of VP. We describe said solutions below.

6.3. Mitigating the Hardware Cost of Early/Late Execution

6.3.1. Mitigating the Early-Execution Hardware Cost. Because Early Executed instructions are processed in-order and are therefore consecutive, one can use a banked PRF and force the allocation of physical registers for the same dispatch group to different register banks. For instance, considering a 4-bank PRF, out of a group of 8 consecutive μ-ops, 2 could be allocated to each bank. In this fashion, a dispatch group of 8 consecutive μ-ops would at most write 2 registers in a single bank after Early Execution. Thus, Early Execution would necessitate only two extra write ports on each PRF bank, as illustrated in Figure 12 for an 8-wide Rename/Early Execute, 4-issue out-of-order core. Interestingly, this would add-up to the number of write ports required by a baseline 6-issue out-of-order core.

In Figure 13, we illustrate simulation results with a banked PRF. In particular, registers from distinct banks are allocated to consecutive μ-ops and Rename is stalled if the current bank does not have any free register. We consider respectively 2 banks of 128 registers, 4 banks of 64 registers and 8 banks of 32 registers. We observe that the performance loss associated with load unbalancing is quite limited for our benchmark set for the 2- and 4-bank configuration. Therefore, using 4 banks of 64 registers instead of a single bank of 256 registers appears as a reasonable tradeoff. However, using 8 banks of 32 registers begins to impact performance as Rename has to stall more often because there are no free registers in a given bank.

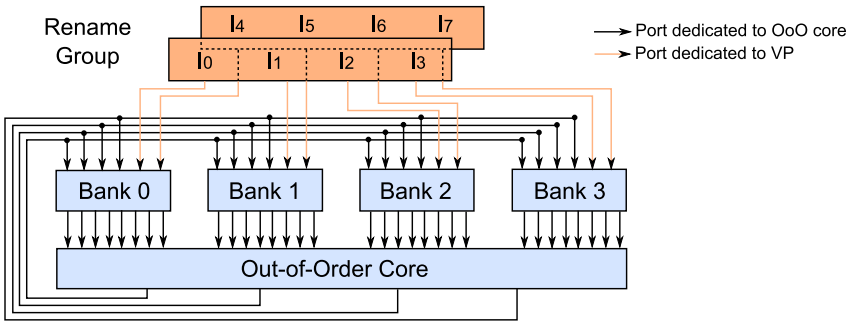


Fig. 12: Organization of a 4-bank PRF supporting 8-wide Early Execution/prediction and 4-wide out-of-order issue. Read ports dedicated to prediction validation and Late Execution are not shown.

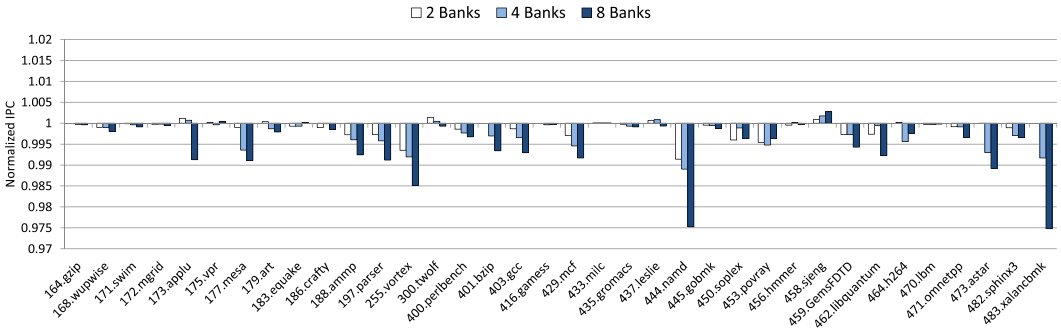


Fig. 13: Performance of *EOLE_4_60* using a different number of banks in the PRF, normalized to *EOLE_4_60* with a single bank.

Note that register file banking is also a solution for a practical implementation of a core featuring Value Prediction without EOLE, since validation is also done in-order. Therefore, only two read ports are necessary (assuming 4 banks) to validate 8 μ-ops per cycle.

6.3.2. Limited Late Execution and Port Sharing. Not all instructions are predicted or late-executable (i.e., predicted and simple ALU or high confidence branches). Moreover, entire groups of 8 μ-ops are rarely ready to commit. Therefore, one can limit the number of potentially late executed instructions and/or predicted instructions per cycle. For instance, the maximum commit-width can be kept to 8 with the extra constraint of using only 6 or 8 PRF read ports for Late Execution and Validation/Training.

Moreover, one can also leverage the register file banking proposed above to limit the number of read ports on each individual register file bank at Late Execution/Validation and Training. To only validate the prediction for 8 μ-ops and train the predictor, and assuming a 4-bank PRF, 2 read ports per bank would be sufficient. However, not all instructions need validation/training (e.g., branches and stores). Hence, some read ports may be available for LE, although extra read ports might be necessary to ensure smooth LE.

Our experiments showed that limiting the number of read ports on each register file bank dedicated to LE/VT to only 4 results in a marginal performance loss. Figure 14 illustrates the performance of *EOLE_4_60* with a 4-bank PRF and respectively 2, 3 and 4 ports provisioned for the LE/VT stage (per bank). As expected, having only two ad-

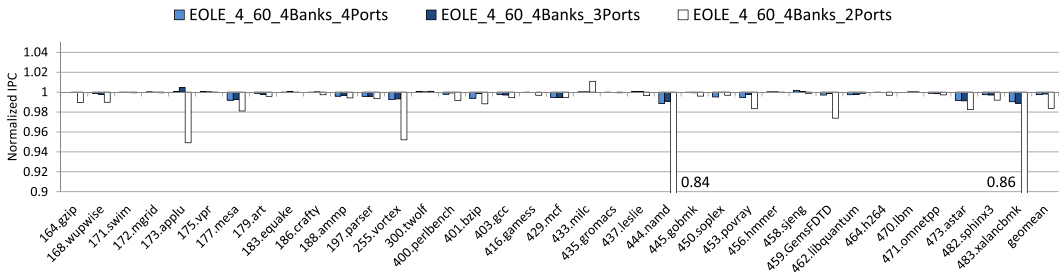


Fig. 14: Performance of *EOLE_4_60* (4-bank PRF) when the number of read ports dedicated to LE/VT is limited, normalized to *EOLE_4_60* (1-bank PRF) with enough ports for full width LE/VT.

ditional read ports per bank is not sufficient. Having 4 additional read ports per bank, however, yields an IPC very similar to that of *EOLE_4_60*. Interestingly, adding 4 read ports adds up to a total of 12 read ports per bank (8 for out-of-order execution and 4 for LE/VT), that is, the same amount of read ports as the baseline 6-issue configuration. Note that provisioning only 3 read ports per bank is also a possibility as performance is also very close to the ideal *EOLE_4_60*.

It should be emphasized that the logic needed to select the group of μ-ops to be Late Executed/Validated on each cycle does not not require complex control and is not on the critical path of the processor. This could be implemented either by an extra pipeline cycle or speculatively after Dispatch.

6.3.3. The Overall Complexity of the Register File. Interestingly, on *EOLE_4_60*, the register file banking proposed above leads to equivalent performance as a non-constrained register file. However, the 4-bank file has only 2 extra write ports per bank for Early Execution and prediction and 4 extra read ports for Late Execution/Validation/Training. That is a total of 12 read ports (8 for the out-of-order engine and 4 for LE/VT) and 6 write ports (4 for the out-of-order engine and 2 for EE/Prediction), just as the baseline 6-issue configuration without VP.

As a result, if the additional complexity induced on the PRF by VP is noticeable (as issue width must remain 6), EOLE allows to virtually nullify this complexity by diminishing the number of ports required by the out-of-order engine. The only remaining difficulty comes from banking the PRF. Nonetheless, according to the previously mentioned area cost formula [Zyuban and Kogge 1998], the total area and power consumption of the PRF of a 4-issue EOLE core is similar to that of a baseline 6-issue core **without** Value Prediction. Yet, performance is substantially higher since VP is present, as summarized in Figure 15.

It should also be mentioned that the EOLE structure naturally leads to a distributed register file organization with one file servicing reads from the out-of-order engine and the other servicing reads from the LE/VT stage. The PRF could be naturally built with a 4-bank, 6 write/8 read ports file (or two copies of a 6 write/4 read ports) and a 4-bank, 6 write/4 read ports one. As a result, the register file in the out-of-order engine would be less likely to become a temperature hotspot than in a conventional design.

6.3.4. Limiting the width of Early and Late Execution. So far, we have considered that *rename_width* and *commit_width* ALUs were implemented in the Early Execution stage and the Late Execution stage, respectively. However, for the same reason that the number of read ports dedicated to Late Execution can be limited, the width of EE and LE can be reduced. Indeed, since many instructions are not predictable or early/late exe-

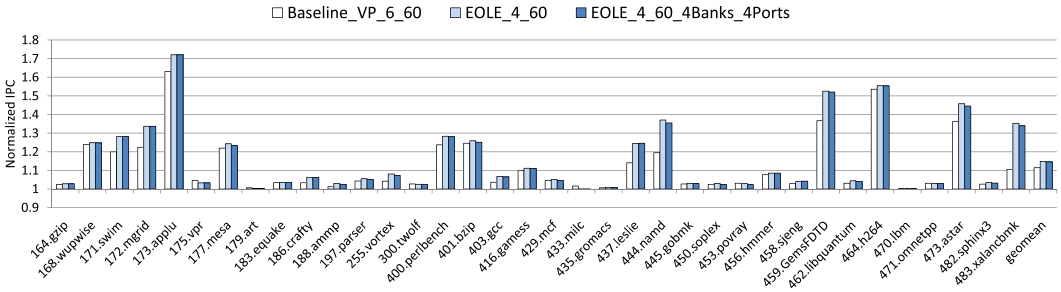


Fig. 15: Performance of *Baseline_VP_6_60*, *EOLE_4_60* with 16 ports for LE/VT and a single bank and *EOLE_4_60* using 4 ports for LE/VT and having 4 64-register banks, normalized to *Baseline_6_60*.

cutable, good enough performance can be attained with only a few additional simple ALUs overall.

Reducing the width of EE is especially interesting since EE requires a full bypass network to perform best. For instance, reducing the number of ALUs from 8 to 4 reduces the bypass paths from 8-to-8 to 4-to-4. There is a caveat, however, in the fact that predicted results of instructions that are not early executed should also be made available to the next rename group. As a result, bypass is rather 8-to-4 than 4-to-4, since there still might be 8 results available for an 8-wide frontend.

Regardless, reducing the width of EE will not cause the pipeline to stall if too many instructions are eligible for Early Execution. Indeed, those instructions will simply have to be executed in the out-of-order engine (unless they are value predicted). On the contrary, reducing the width of Late Execution may stall the pipeline by limiting the number of instructions that can be committed, causing the ROB to become full. For instance, if eight instructions are late-executable but only two ALUs are present, then it will take four cycles to commit them, while the Commit stage could have retired them in a single cycle if 8 ALUs had been present.

Note that since we perform prediction validation at Late Execution, we consider that each LE ALU has the comparator required to do so. As a result, by reducing the number of LE ALUs, we also reduce the number of instructions whose prediction can be validated in order to train the predictor and squash on a misprediction. The maximum commit width of 8 remains attainable if the commit group contains enough instructions that do not produce a register (e.g., low confidence branches that were executed out-of-order, stores, etc.).

Figure 16 depicts the performance impact of decreasing the width of Early/Late Execution. The Figure shows IPC normalized to *EOLE_4_60* with a 4-bank PRF and 4 read ports dedicated to LE/VT per PRF bank, and having 8-wide EE/LE stages. The first observation is that for an 8-wide Rename and an 8-wide Commit, 6-wide EE/LE is sufficient, as performance is comparable to having 8-wide EE/LE, except in *namd* (13.3% performance loss). Note however that performance in *namd* is still higher than the baseline 6-issue without VP (17.5% speedup).

However, as the EE/LE widths are decreased, performance degrades significantly, to attain 70% of the baseline when EE and LE are only 2-wide. Interestingly, using a 2-wide EE stage but a 6-wide LE stage has similar impact as using 6-wide EE/LE stages. This suggests that most of the performance loss is caused by the reduced-width LE stage, since it often limits the actual commit width, while the reduced-width EE stage only increases pressure on the execution engine. Consequently, while decreasing the EE width from 8 to only 2 does not impact performance significantly, keeping the

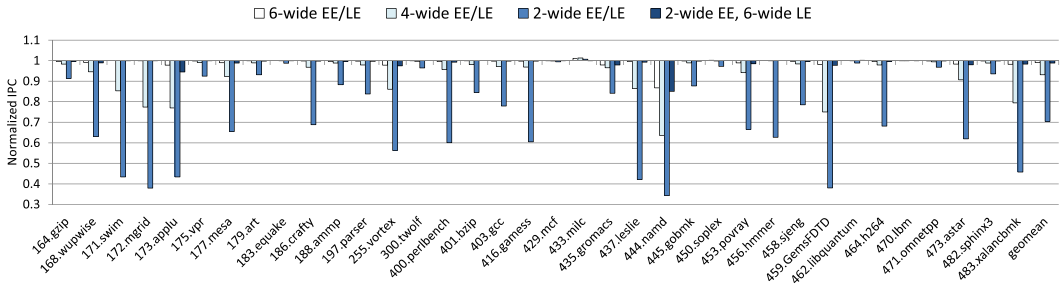


Fig. 16: Performance impact of reducing the width of Early/Late Execution. Normalized to *EOLE_4_60* with a 4-bank PRF and 4 read ports dedicated to LE/VT, per PRF bank, and having 8-wide EE/LE stages.

LE width close to the commit width is necessary to limit pipeline stalls due to the ROB becoming full.

Furthermore, as mentioned in 3.4.2, a significant amount of value predicted instructions are loads, which require validation but not late execution. Therefore, it is possible that the LE width could be decreased further if LE and prediction validation/training were to be implemented as two separate pipeline stages, and the validation/training width kept similar to the commit width.

6.4. Late Executing Predicted Loads

Most processors do not allow more than two loads to issue in the same cycle. The main reason is that adding ports to the data cache to handle more accesses is very expensive. Consequently, banking the data cache is generally preferred, but arbitration increases access latency and the number of bank conflicts will necessary increase with the number of loads issued each cycle. As a result, it is not desirable to add a datapath between the data cache and the Late Execution stage as long as the execution engine already has the ability to issue two loads per cycle.

Nonetheless, by only allowing one load per cycle to be issued by the execution engine, we can move the second load port to the Late Execution stage. If the performance level of the baseline processor can be maintained, doing so may enable further reduction of the out-of-order engine aggressiveness.

Specifically, the Load Queue (LQ) is implemented to provide correctness in the presence of out-of-order execution of memory operations. It can also be snooped by remote writes to enforce the memory model in some processors (e.g., x86_64 [Intel 2007]). Since late executed loads are executed in-order, they will not alias with an older store whose address is unknown, since all older stores have executed, by construction. As a result, it may not be necessary to add late executed loads to the LQ, depending on the memory model implemented by the processor.

6.4.1. Late Executing Loads. As a first step, and since we focus on single-core, we can ignore the fact that x86_64 is strongly ordered and consider that loads can safely bypass the LQ while allowing late-execution not to wait for older loads to return before moving on to younger instructions.[16] This essentially gives us the best of both world, by potentially allowing to reduce the LQ size, while avoiding to stall the pipeline because the ROB became full waiting for a late executed load to complete while subsequent instructions could have been late executed in the meantime.

[16]This can actually lead to a violation of the memory model for x86_64 [Intel 2007].

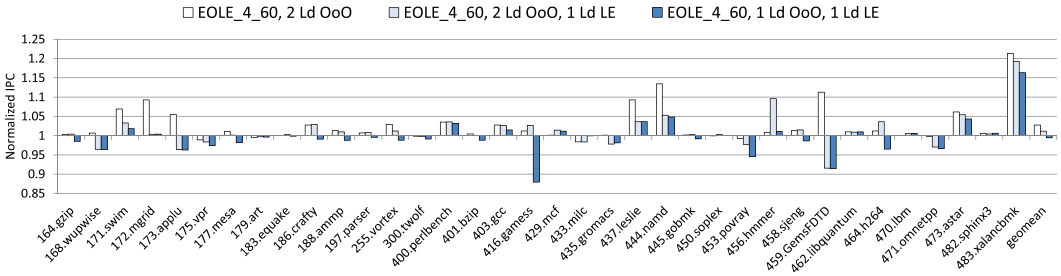


Fig. 17: Performance impact of allowing predicted loads to be late executed. IPC is normalized to *Baseline_VP_6_60*. All configurations have 4 PRF banks and 4 read ports dedicated to LE/VT.

Figure 17 shows IPC normalized to *Baseline_VP_6_60* for different EOLE configurations. The first bar shows performance for *EOLE_4_60* with 4 PRF banks and 4 read ports dedicated to LE/VT (all the considered configurations have the same number of PRF banks and read ports). The second bar adds the ability to late execute loads, but can still issue *two* loads out-of-order each cycle, contrarily to the third bar, which can only issue *one*. In the two latter cases, validation is able to process up to 8 μ-ops even if they are not contiguous in the ROB. For instance, when a late executed load returns, it can be validated in parallel with 7 younger μ-ops that may be further down the ROB.[17]

Overall, a noticeable performance drop can be observed in several benchmarks when predicted loads are late executed, even if the 2 loads/cycle throughput is preserved in the out-of-order engine. For the second bar (2 loads OoO, 1 load LE), the reason is that if many loads are predicted, then the maximum load throughput will tend towards one, and since the completion of a predicted load happens later than if it had been executed out-of-order, there is more chance of the ROB becoming full than before. This is the case in *wupwise, applu, GemsFDTD*, and this leads to performance dropping even though 3 loads can theoretically proceed each cycle.

For the third bar (1 load OoO, 1 load LE), if few loads are predicted, then once again, the maximum load throughput will tend towards one, and performance may be reduced (e.g., *gamess* and *h264*).

Consequently, in both low load predictability and high load predictability cases, having a single load port in the out-of-order engine can lead to performance degradation. In the high load predictability case, it would be beneficial to have a steering mechanism deciding if a predicted load is to be late executed or not. This would allow a more efficient use of the two load ports (one in OoO, one in LE). However, in the second case (too few loads are predicted), the only solution to improve the load throughput is to allow non-predicted loads to be late executed, which is unlikely to be beneficial since it might significantly increase the length of the program critical path.

Moreover, we found that although much fewer instructions enter the scheduler when predicted loads are late executed, neither the scheduler size nor the issue width can be reduced further without impacting performance noticeably. In that context, the only interest of allowing loads to be late executed would be to reduce the LQ size.

Load Queue Size. Figure 18 considers the case where the size of the LQ is varied from the baseline (72-entry) down to 36 entries. Until 48 entries, this has virtually no impact on performance. Even then, while a slowdown is observed in *equake*, 48

[17]This could be achieved by keeping two ROB pointers: one to the oldest pending load, and one to the oldest non-validated non-load instruction.

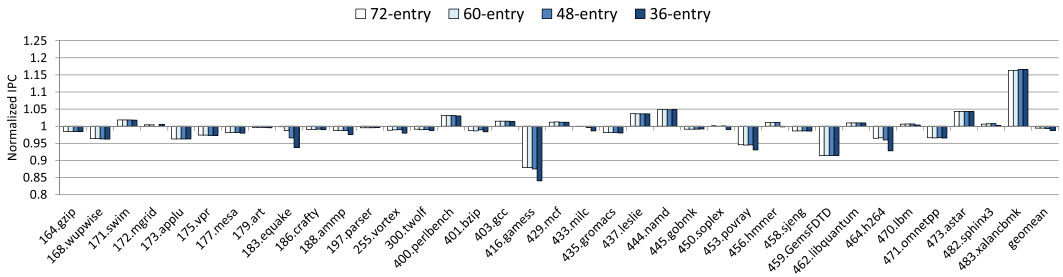


Fig. 18: Impact of reducing the LQ size on performance when predicted loads are late executed and a single load port is provisioned in the execution engine. IPC is normalized to *Baseline_VP_6_60*.

entries also provide performance on par with the 72-entry LQ. A 36-entry LQ, however, impedes performance in *equake, gamess* and *h264*. As a result, the LQ should remain large enough to handle cases where few loads are predicted (i.e., the majority of loads must be inserted in the LQ).

In addition, we found that even in the EOLE model that cannot late execute loads, the LQ can be reduced to 60-entry without significant performance loss. This suggests that if not adding late executed loads to the LQ clearly reduces the pressure put on the structure, the LQ is generally not a bottleneck on our benchmark set.

In any case, in x86_64, the only actual way to *not* add late executable loads to the LQ is to enforce that they *complete* in-order in Late Execution. Consequently, to allow some of the latency to be hidden, late executable loads would have to issue speculatively if an older load were in flight, and be replayed if a younger late executed load returned before an older one. Due to the numerous events that can lead to such a case (cache misses, cache bank conflicts, bus contention), this may plainly be too impractical for x86_64. Consequently, we argue that the baseline EOLE model that only allows single-cycle ALU instructions to be early/late executed is a more interesting design point, at least for the window size that is considered.

6.5. Summary on the Hardware Complexity of EOLE

Apart from the prediction tables and the update logic, the major hardware overhead associated with implementing VP and validation at commit time comes from the extra read and write ports on the physical register file. We have shown above that EOLE allows to get rid of this overhead on the PRF as long as enough banks can be implemented.

Specifically, EOLE allows to use a 4-issue out-of-order engine instead of a 6-issue engine. This implies a much smaller instruction scheduler, a much simpler bypass network and a reduced number of PRF read and write ports in the out-of-order engine. As a result, one can expect many advantages in the design of the out-of-order execution core: Significant silicon area savings, significant power savings in the scheduler and the register file and savings on the access time of the register file. Power consumption savings are crucial since the scheduler has been shown to consume almost 20% of the power of a modern superscalar core [Ernst and Austin 2002], and is often a temperature hotspot in modern designs. As such, even if global power savings were not to be achieved due to the extra hardware required in EOLE, the power consumption is likely to be more distributed across the core.

On the other hand, EOLE requires some extra but relatively simple hardware for Early/Late Execution. Apart from some control logic, this extra hardware consists of a set of ALUs and a bypass network in the Early Execution stage and a set of ALUs

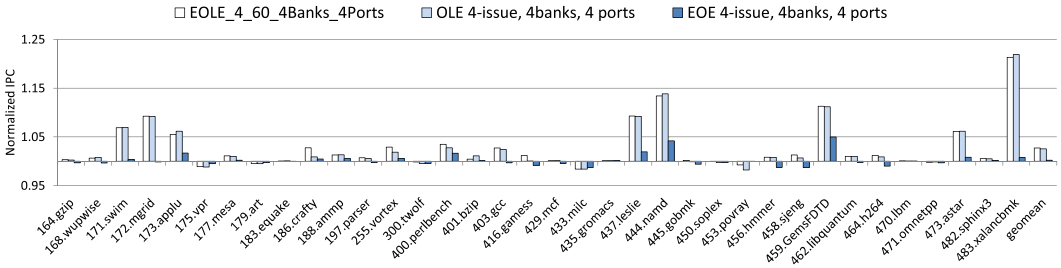


Fig. 19: Performance of *EOLE_4_60*, *OLE_4_60* and *EOE_4_60* using 4 ports for LE/VT and having 4 64-register banks, normalized to *Baseline_VP_6_60*.

in the Late Execution stage. A full rank of ALUs is actually unlikely to be needed. In fact, from Figure 16 we consider that a rank of 2/4 ALUs would be sufficient for EE, while 6/8 ALUs would be required in LE.

In addition, while it would potentially be possible to move one cache port to Late Execution to allow predicted loads to be late executed, we found that this has little practical interest as performance may be reduced due to the uneven number of predicted/non predicted loads as well as the additional pressure put on the ROB by late executed loads.

Lastly, implementing EOLE is unlikely to impair cycle time. Indeed, Early Execution requires only one stage of simple ALUs and can be done in parallel with Rename. Late Execution and validation may require more than one additional pipeline stage compared to a conventional superscalar processor, but this should have a fairly small impact since low-confidence branch resolution is not delayed. In fact, since EOLE simplifies the out-of-order engine, it is possible that the core could actually be clocked higher, yielding even more sequential performance.

Therefore, our claim is that EOLE makes a clear case for implementing VP on wide-issue superscalar processors. Higher performance is enabled thanks to VP (see Figure 15) while EOLE enables a much simpler and far less power hungry out-of-order engine. The extra hardware blocks required for EOLE are relatively simple: Sets of ALUs in Early Execution and Late Execution stages, and storage tables and update logic for the value predictor itself.

6.6. A Note on the Modularity of EOLE: Introducing OLE and EOE

EOLE need not be implemented as a whole. In particular, either Early Execution *or* Late Execution can be implemented, if the performance vs. complexity tradeoff is deemed worthy. Removing Late Execution can further reduce the number of read ports required on the PRF. Removing Early Execution saves on complexity since there is no need for an additional bypass network anymore.

Figure 19 shows the respective speedups of *EOLE_4_60*, *OLE_4_60* (Late Execution only) and *EOE_4_60* (Early Execution only) over *Baseline_VP_6_60*. As in the previous paragraph, only 4 read ports are dedicated to Late Execution/Validation and Training, and the PRF is 4-banked (64 registers in each bank). The baseline has a single 256-register bank and enough ports to avoid contention.

We observe that some benchmarks are more sensitive to the absence of Late Execution (e.g., *applu, bzip, gcc, namd, hmmer* and *h264*) while one is more sensitive to the absence of Early Execution (*povray*). Nonetheless, removing Late Execution appears as more detrimental in the general case.

However, slowdown over *Baseline_VP_6_60* remains under 5% in all cases. This suggests that when considering an effective implementation of VP using EOLE, an ad-

ditional degree of freedom exists as either only Early or Late Execution may be implemented. Indeed, while Early Execution only does not generally allow to increase performance over *Baseline_VP_6_60*, it is still able to maintain the same performance level while the issue width is reduced.

7. CONCLUSION AND FUTURE WORK

Single thread performance remains the driving force for the design of high-performance cores. However, hardware complexity and power consumption remain major obstacles to the implementation of new architectural features.

Value Prediction (VP) is one of such features that has still not been implemented in real-world products due to those obstacles. Fortunately, a recent advance in research on VP partially addressed these issues [Perais and Seznec 2014b]. In particular, it was shown that validation can be performed at commit time without sacrificing performance. This greatly simplifies design, as the burdens of validation at execution-time and *selective replay* for VP in the out-of-order engine are eliminated.

Building on this previous work, we have proposed EOLE, an {*Early — Out-of-Order — Late*} Execution microarchitecture aiming at further reducing the hardware complexity and the power consumption of a VP-augmented superscalar processor.

With Early Execution, single-cycle instructions whose operands are immediate or predicted are computed in-order in the front-end and do not have to flow through the out-of-order engine. With Late Execution, predicted single-cycle instructions as well as very high confidence branches are computed in-order in a pre-commit stage. They also do not flow through the out-of-order engine. As a result, EOLE significantly reduces the number of instructions dispatched to the out-of-order engine.

Considering a 6-wide, 60-entry IQ processor augmented with VP and validation at commit time as the baseline, EOLE allows to drastically reduce the overall complexity and power consumption of both the out-of-order engine and the PRF. EOLE achieves similar or higher performance when using a 4-issue, 60-entry IQ engine, with a few – three on our benchmark set – exceptions being modestly slowed down (1.7% at worst in *milc*).

With EOLE, the overhead over a 6-wide, 60-entry IQ processor (without VP) essentially consists of relatively simple hardware components, the two set of ALUs in the Early and Late Execution stages, a bypass network and the value predictor tables and update logic. The need for additional ports on the PRF is also substantially lowered by the reduction in issue width and some PRF optimizations (e.g., banking). Lastly, the PRF could be distributed into a copy in the out-of-order engine and a copy only read by the Late Execution/Validation and Training stage. Consequently, EOLE results in a much less complex and power hungry out-of-order engine, while generally benefiting from higher performance thanks to Value Prediction. Moreover, we hinted that Late Execution and Early Execution can be implemented separately, with Late Execution appearing as more cost-effective.

Further studies to evaluate the possible variations of EOLE designs may include the full range of hardware complexity mitigation techniques that were discussed in Section 6.3 and 6.4 for both Early and Late execution, and the exploration of other possible sources of Late Execution, e.g., indirect jumps, returns, but also store address computations. One can also explore the interactions between EOLE and previous propositions aiming at reducing the complexity of the out-of-order engine such as the *Multicluster* architecture [Farkas et al. 1997] or register file-oriented optimizations [Wallace and Bagherzadeh 1996]. Finally, future research includes the need to look for even more accurate predictors.

REFERENCES

P.S. Ahuja, D.W. Clark, and A. Rogers. 1995. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the International Symposium on Microarchitecture*.

T. M. Austin. 1999. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the International Symposium on Microarchitecture*.

N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

G. Z. Chrysos and J. S. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*.

R.J. Eickemeyer and S. Vassiliadis. 1993. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development* 37, 4 (1993), 547–564.

D. Ernst and T. Austin. 2002. Efficient dynamic scheduling through tag elimination. In *Proceedings of the International Symposium on Computer Architecture*.

B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. 2005. Continuous Optimization. In *Proceedings of the International Symposium on Computer Architecture*.

K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. 1997. The Multicluster Architecture: reducing cycle time through partitioning. In *Proceedings of the International Symposium on Microarchitecture*. 11.

B. Fields, S. Rubin, and R. Bodík. 2001. Focusing processor policies via critical-path prediction. In *Proceedings of the International Symposium on Computer Architecture*.

F. Gabbay and A. Mendelson. 1998. Using value prediction to increase the power of speculative execution hardware. *ACM Trans. Comput. Syst.* 16, 3 (Aug. 1998), 234–270.

S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. 2003. The Intel Pentium M Processor: MicroArchitecture and Performance. *Intel Technology Journal* 7 (May 2003).

B. Goeman, H. Vandierendonck, and K. De Bosschere. 2001. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the International Conference on High-Performance Computer Architecture*.

Intel. 2007. *Intel 64 Architecture Memory Ordering White Paper*. http://www.cs.cmu.edu/~410/doc/Intel_Reordering_318147.pdf

Intel. 2013. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

Intel. 2014. Software Optimization Manual. (September 2014). http://www..fr/content/www/fr/fr/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. 1998. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *Proceedings of the International Symposium on Microarchitecture*.

R. E. Kessler, E. J. Mclellan, and D. A. Webb. 1998. The Alpha 21264 microprocessor Architecture. In *Proceedings of the International Conference on Computer Design*.

I. Kim and M. H. Lipasti. 2003. Half-price Architecture. In *Proceedings of the International Symposium on Computer Architecture*.

I. Kim and M. H. Lipasti. 2004. Understanding Scheduling Replay Schemes. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. 1996. Value locality and load value prediction. *Proceedings of the International conference on Architectural Support for Programming Languages and Operating Systems* (1996).

M. H. Lipasti and J. P. Shen. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the Annual International Symposium on Microarchitecture*.

A. Lukefahr, S. Padmanabha, R. Das, F.M. Sleiman, R. Dreslinski, T.F. Wenisch, and S. Mahlke. 2012. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the International Symposium on Microarchitecture*.

A. Mendelson and F. Gabbay. 1997. *Speculative execution based on value prediction*. Technical Report TR1080. Technion-Israel Institute of Technology.

T. Nakra, R. Gupta, and M.L. Soffa. 1999. Global context-based value prediction. In *Proceedings of the International Symposium On High-Performance Computer Architecture*. 4–12.

S. Palacharla, N.P. Jouppi, and J.E. Smith. 1997. Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture*.

A. Perais and A. Seznec. 2014a. EOLE: Paving the Way for an Effective Implementation of Value Prediction. In *Proceedings of the International Symposium on Computer Architecture*.

A. Perais and A. Seznec. 2014b. Practical Data Value Speculation for Future High-end Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

A. Perais and A. Seznec. 2015. BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. Hagersten. 2015. Cost-Effective Speculative Scheduling in High Performance Processors. In *Proceedings of the International Symposium on Computer Architecture*.

E. Perelman, G. Hamerly, and B. Calder. 2003. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

V. Petric, T. Sha, and A. Roth. 2005. RENO: a rename-based instruction optimizer. In *Proceedings of the International Symposium on Computer Architecture*.

B. Rychlik, J.W. Faistl, B.P. Krug, A.Y. Kurland, J.J. Sung, M.N. Velev, and J.P. Shen. 1998. Efficient and accurate value prediction using dynamic classification. *Carnegie Mellon University, CMμART-1998-01* (1998).

Y. Sazeides and J.E. Smith. 1997. The predictability of data values. In *Proceedings ofthe International Symposium on Microarchitecture*.

A. Seznec. 2011. Storage Free Confidence Estimation for the TAGE branch predictor. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

A. Seznec and P. Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction Level Parallelism* 8 (2006).

A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors. In *Proceedings of the International Symposium on Microarchitecture*.

Standard Performance Evaluation Corporation. 2000. CPU. (2000). http://www.spec.org/cpu2000/

Standard Performance Evaluation Corporation. 2006. CPU. (2006). http://www.spec.org/cpu2006/

R. Thomas and M. Franklin. 2001. Using dataflow based context for accurate value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

F. Tseng and Y. N. Patt. 2008. Achieving Out-of-Order Performance with Almost In-Order Complexity. In *Proceedings of the International Symposium on Computer Architecture*.

E. S. Tune, D. M. Tullsen, and B. Calder. 2002. Quantifying instruction criticality. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*.

S. Wallace and N. Bagherzadeh. 1996. A scalable register file Architecture for dynamically scheduled processors. In *Proceedings of the International Conference on Parallel architectures and Compilation Techniques*.

K. Wang and M. Franklin. 1997. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the International Symposium on Microrchitecture*.

H. Zhou, J. Flanagan, and T. M. Conte. 2003. Detecting Global Stride Locality in Value Streams. In *Proceedings of the International Symposium on Computer Architecture*.

V. Zyuban and P. Kogge. 1998. The energy complexity of register files. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 305–310.

Received November 2015; revised ; accepted