EOLE: Paving the Way for an Effective Implementation of Value Prediction

Arthur Perais André Seznec
IRISA/INRIA
{arthur.perais, Andre.Seznec}@inria.fr

Abstract

Even in the multicore era, there is a continuous demand to increase the performance of single-threaded applications. However, the conventional path of increasing both issue width and instruction window size inevitably leads to the power wall. Value prediction (VP) was proposed in the mid 90's as an alternative path to further enhance the performance of wide-issue superscalar processors. Still, it was considered up to recently that a performance-effective implementation of Value Prediction would add tremendous complexity and power consumption in almost every stage of the pipeline.

Nonetheless, 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, recovering from mispredictions via selective replay can be avoided and a much simpler mechanism – pipeline squashing – can be used, while the out-of-order engine remains mostly unmodified.

Yet, VP and validation at commit time entails strong constraints on the Physical Register File. Write ports are needed to write predicted results and read ports are needed in order to validate them at commit time, 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 60% in our experiments – can bypass the out-of-order engine, allowing the reduction of the issue width, which is a major contributor to both out-of-order engine complexity and register file port requirement. 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.

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) 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 *selective replay* to efficiently recover from schedule mispredictions [17].

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 rapidly 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 [24]. In other words, the out-of-order engine impact on power consumption and cycle time is ever increasing [6].

In this paper, we propose a modified superscalar design, the *{Early | Out-of-Order | Late} Execution* microarchitecture, *EOLE*. It is built on top of a Value Prediction (VP) pipeline. VP allows dependents to issue earlier than previously possible by using predicted operands, and thus uncovers more ILP. Yet, predictions must be verified to ensure correctness. Fortunately, Perais and Seznec observed that one can validate the predicted results outside the out-of-order engine – at retirement – provided an enhanced confidence estimation mechanism [25].

With EOLE, we leverage this observation to further reduce both the complexity of the out-of-order (OoO) 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 the most complicated and power-hungry part of a modern OoO core.

In particular, when 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

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

[2] From respectively 128 and 36 entries to 192 and 60 entries.

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

in parallel with Rename by using predicted and/or immediate operands. Early-executed instructions are **not** sent to the OoO 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 OoO instruction window.

Similarly, since predicted results can be validated outside the OoO engine at commit time [25], 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 OoO 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 60% of the retired instructions can be offloaded from the OoO core. As a result, EOLE benefits from both the aggressiveness of modern OoO designs and the higher energy-efficiency of more conservative in-order designs.

We evaluate EOLE against a baseline OoO 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 OoO 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

Complexity-Effective Architectures 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 OoO engine, including the PRF [38], Scheduler and bypass network [24]. As such, previous studies focused on either reducing the complexity of existing structures, or in devising new pipeline organizations.

Farkas et al. propose the *Multicluster* architecture in which execution is distributed among several execution clusters, each of them having its own register file [8]. The Alpha 21264 [15] is an example of real-world clustered architecture and shares many traits with the *Multicluster* architecture.

Palacharla et al. introduce a *dependence-based* microarchitecture where the centralized instruction window is replaced by several parallel FIFOs [24]. This greatly reduces complexity since only the head of each FIFO has to be selected by the *Select* logic. They also study a *clustered dependence-based* architecture to reduce the amount of bypass and window logic by using clustering.

Tseng and Patt propose the *Braid* architecture [36], which shares many similarities with the *clustered dependence-based* architecture except that instruction steering is done at compile time.

Austin proposes *Dynamic Implementation VAlidation* (DIVA) to check instruction results just before commit time, allowing the core to be faulty [2]. 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* does not actually impact performance much.

Fahs et al. study *Continuous Optimization* where common compile-time optimizations are applied dynamically in the Rename stage [7]. This allows to *early execute* some instructions in the front-end instead of the OoO core. Similarly, Petric et al. propose RENO which also dynamically applies optimizations at rename-time [27].

Instead of studying new organizations of the pipeline, Kim and Lipasti present the *Half-Price Architecture* [16]. 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. Similarly, Ernst and Austin propose *Tag Elimination* to limit the number of comparators used for *Wakeup* [6].

Regarding the register file, Kim and Lipasti also observe that many issuing instructions do not need to read both their operands in the register file since one or both will be available on the bypass network [16]. Thus, provisioning two read ports per issue slot is generally over-provisioning. 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. propose to implement two back-ends – in-order and OoO – in a single core [20] 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 as the location where an instruction will be executed depends only on its type and status (e.g. predicted).

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

Value Prediction *EOLE* builds upon the broad spectrum of research on Value Prediction independently initiated by Lipasti et al. and Gabbay et al. [10, 18].

Sazeides et al. refine the taxonomy of VP by categorizing predictors [29]. 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 [22] and the *2-Delta* Stride predictor [5] 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 [29]. Most of the initial studies on Value Prediction either assumed that the recovery on a value misprediction induces almost no penalty [18, 19, 40], or simply focused on accuracy and coverage rather than speedup [12, 23, 28, 29, 35, 39]. The latter studies were essentially ignoring the performance loss associated with misprediction recovery, i.e. assuming a perfect 0-/1-cycle selective replay. Such a mechanism is known to be unrealistic [17].

In a recent study, Perais and Seznec show that all value predictors are amenable to very high accuracy at a reasonable cost in coverage [25]. This allows to delay prediction validation until commit time, removing the burden of implementing a complex replay mechanism. As such, the OoO 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.

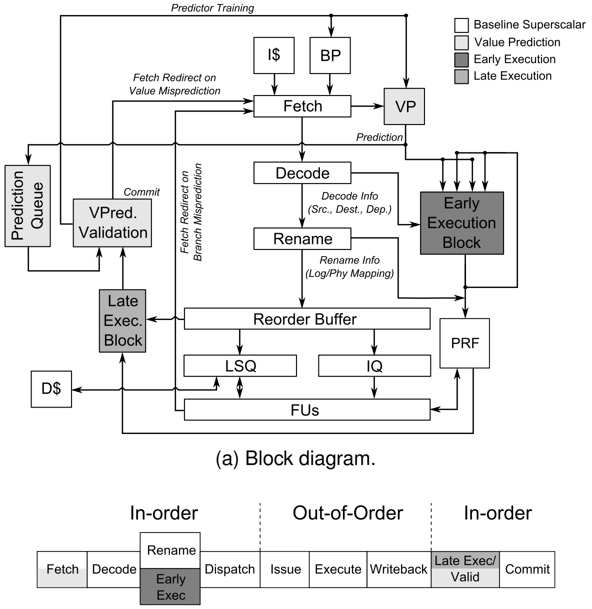
In the same paper, they introduce the VTAGE context-based predictor. As the ITTAGE indirect branch predictor [31], VTAGE uses the global branch history to select predictions, meaning that it does not require the previous value to predict the current one. This is a strong advantage since conventional value predictors usually need to track inflight predictions as they require the last value to predict.

3. EOLE

3.1. Enabling EOLE Using 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. Yet, to be implemented, EOLE requires prediction validation to be done at commit since validating at Execute mechanically forbids Late Execution. Furthermore, 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 OoO Scheduler in case they need to be replayed [17]. Hence, *squashing* must be used to recover from a misprediction so that early/late-executed instructions can safely bypass the Scheduler.



(a) Block diagram.

(b) Pipeline diagram.

Figure 1: The EOLE μ-architecture.

Fortunately, Perais and Seznec have proposed a confidence estimation mechanism greatly limiting the number of value mispredictions, *Forward Probabilistic Counters* (FPC) [25]. 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 OoO engine, EOLE substantially reduces the pressure on complex and power-hungry structures. Thus, those structures can be scaled down, yielding a less complex architecture whose performance is on par with a more aggressive design. Moreover, doing so is orthogonal to previously proposed mechanisms such as *clustering* [8, 15, 24, 32] and does not *require* a centralized instruction window, even though this is the model we use in this paper. Fig. 1 depicts the EOLE architecture, implementing both Early Execution (darkest), Late Execution (darker) and Value Prediction (lighter). 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,

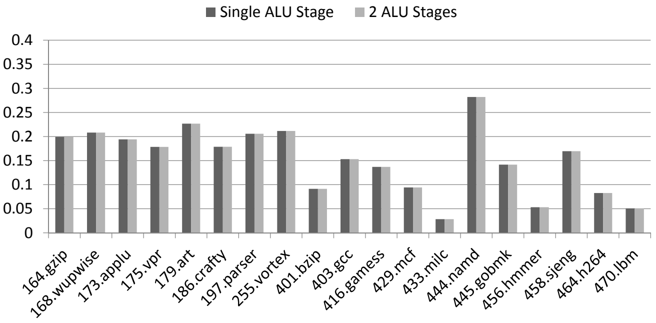


Figure 2: Proportion of committed instructions that can be early-executed, using one or two ALU stages and a VTAGE-2DStride hybrid predictor (later described in Section 4).

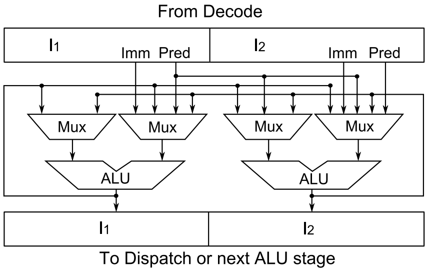


Figure 3: Early Execution Block. The logic controlling the ALUs and muxes is not shown for clarity.

memory instructions are not early-executed. Early Execution is done in-order, hence, it does not require renamed registers and can take place in parallel with Rename. For instance, Fig. 3 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[3] bypass network (i.e. from instructions early-executed in the previous cycle) or the value predictor. Operands are **never** read from the PRF.

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 Fig. 2. This figure shows the proportion of committed instructions eligible for Early Execution for a baseline 8-wide rename, 6-issue model (see Table 1 in Section 4), using the VTAGE/2D-Stride hybrid predictor (later described in Table 2, Section 4). As a result, in further experiments, we consider a 1-deep Early Execution Block.

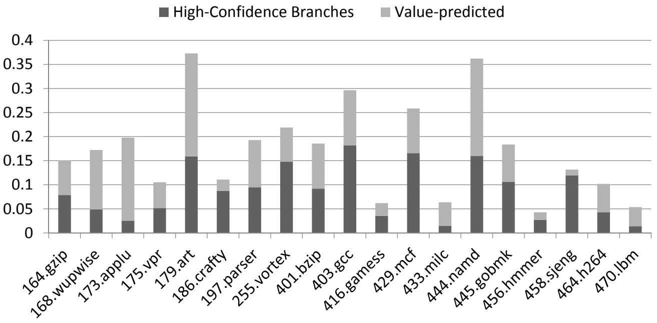


Figure 4: Proportion of committed instructions that can be late-executed using a VTAGE-2DStride (see Section 4) hybrid predictor. Late-executable instructions that can also be early-executed are not counted since instructions are executed once at most.

To summarize, Early Execution only requires a single new computing block, which is shown in dark grey in Fig. 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.

3.3. Late Execution Hardware

Late Execution (LE) targets instructions whose result has been predicted[4]. It is done just before 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 OoO engine, but validated at commit.

Interestingly, Seznec showed in [30] 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 has 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.

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.

In any case, Late Execution further reduces pressure on the OoO engine in terms of instructions dispatched to the Sched-

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

[4] Instructions eligible for prediction are μ-ops producing a 64-bit or less result that can be read by a subsequent μ-op, as defined by the ISA implementation.

[5] Predictions whose confidence counter is saturated [30].

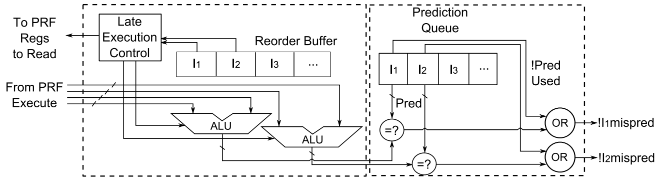


Figure 5: Late Execution Block for a 2-wide processor. The left part can late-execute two instructions while the right part validates two results against their respective predictions. Buses are *register-width*-bit wide.

uler. As such, it also removes the need for predicting only critical instructions [9, 28, 37] since minimizing the number of instructions flowing through the OoO engine requires maximizing the number of predicted instructions. Hence, usually useless predictions from a performance standpoint become useful in EOLE. In particular, Fig. 4 shows the proportion of committed instructions that can be late-executed using a baseline 6-issue processor with a VTAGE-2DStride hybrid predictor (respectively described in Tables 1 and 2 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 and 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, as in DIVA [2]. 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.

As a result, the hardware needed for LE is fairly simple, as suggested by the high-level view of a 2-wide LE Block shown in Fig. 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 and the pipeline fill delay. 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.

3.4. Potential OoO Engine Offload

We obtain the ratio of retired instructions that can be offloaded from the OoO engine for each benchmark by summing the columns in Fig. 2 and 4 (both sets are disjoint as we only count late executable instructions that cannot also be early executed). This ratio is very dependent on the application, ranging from less than 10% for *milc, hmmer* and *lbm* to more than 50% for *art* and up to 60% for *namd*. Nonetheless, it represents a significant part of the retired instructions in most cases.

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. We use a modified[6] version of the *gem5* cycle-accurate simulator [3]. Unfortunately, contrarily to modern x86 implementations, *gem5* does not support *move elimination* [7, 14, 27], *μ-op fusion* [11] and does not implement a *stack-engine* [11].

We consider a relatively aggressive 4GHz, 6-wide issue superscalar[7] baseline with a fetch-to-commit latency of 19 cycles. Since we focus on the OoO 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 1 describes the characteristics of the baseline pipeline we use in more details. In particular, the OoO scheduler is dimensioned with a unified centralized 64-entry IQ and a 192-entry ROB on par with Haswell's, the latest commercially available Intel microarchitecture. We refer to this baseline as the *Baseline_6_64* configuration (6-issue, 64-entry IQ).

As μ-ops are known at Fetch in *gem5*, all the widths given in Table 1 are in μ-ops, even for the fetch stage. Independent memory instructions (as predicted by the Store Sets predictor [4]) are allowed to issue out-of-order. Entries in the IQ are released upon issue.

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

4.2. Value Predictor Operation

The predictor makes a prediction at fetch time for every eligible μ-op (i.e. producing a 64-bit or less register that can be read by a subsequent μ-op, as defined by the ISA implementation). To index the predictor, we XOR the PC of the x86_64 instruction left-shifted by two with the μ-op number inside

[6] 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.

[7] On our benchmark set and with our baseline simulator, an 8-issue machine achieves only marginal speedup over this baseline.

Front End	L1I 4-way 32KB, Perfect TLB; 8-wide fetch (2 taken branch/cycle), decode, rename; TAGE 1+12 components [31] 15K-entry total, 20 cycles min. mis. penalty; 2-way 4K-entry BTB, 32-entry RAS;
Execution	192-entry ROB, 64-entry IQ unified, 48/48-entry LQ/SQ, 256/256 INT/FP registers; 1K-SSID/LFST Store Sets [4]; 6-issue, 6ALU(1c), 4MulDiv(3c/25c*), 6FP(3c), 4FPMul-Div(5c/10c*), 4Ld/Str; Full bypass; 8-wide retire;
Caches	L1D 4-way 32KB, 2 cycles, 64 MSHRs, 4 load ports; Unified L2 16-way 2MB, 12 cycles, 64 MSHRs, no port constraints, Stride prefetcher, degree 8, distance 1; 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; Across a 64B bus; Min. Read Lat.: 75 cycles, Max. 185 cycles.

Table 1: Simulator configuration overview. *not pipelined.

the x86_64 instruction. This avoids all μ-ops mapping to the same entry. 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 OoO engine [25]. 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.

Finally, since we focus on single core, the impact of VP on memory consistency [21] in EOLE is left for future work.

x86 Flags In the x86_64 ISA, some instructions write flags based on their results while some need them to execute (e.g. branches) [13]. 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 [13].

Predictors Considered in this Study In this study, we focus on the hybrid predictor VTAGE-2DStride recently introduced by Perais and Seznec [25]. It combines a simple and cost-effective *2-Delta* Stride predictor [5] as a representative of the *computational* family – as defined by Sazeides et al. [29] – and a state-of-the-art VTAGE predictor [25] as a representative of the *context-based* family. For confidence estimation, we use Forward Probabilistic Counters as described by Perais and Seznec in [25]. In particular, we use 3-bit confidence counters whose forward transitions are controlled by the vector $v = \{1, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{64}, \frac{1}{64}\}$ as we found it to perform best with VTAGE-2DStride. Table 2 summarizes the configuration of each predictor component.

Predictor	#Entries	Tag	Size (KB)
2D-Stride [5]	8192	Full (51)	251.9
VTAGE [25]	8192 (Base) 6×1024	- 12 + *rank*	68.6 64.1

Table 2: Layout Summary. For VTAGE, *rank* is the position of the tagged component and varies from 1 to 6, 1 being the component using the shortest history length.

4.3. Benchmarks

We use a subset of the the SPEC'00 [33] and SPEC'06 [34] suites to evaluate our contribution as we focus on single-thread performance. Specifically, we use 12 integer benchmarks and 7 floating-point programs[8]. Table 3 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 [26]. 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.

5. Experimental Results

Program	Input	IPC
164.gzip (INT)	input.source 60	0.984
168.wupwise (FP)	wupwise.in	1.553
173.applu (FP)	applu.in	1.591
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	1.326
179.art (FP)	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	1.211
186.crafty (INT)	crafty.in	1.769
197.parser (INT)	ref.in 2.1.dict -batch	0.544
255.vortex (INT)	lendian1.raw	1.781
401.bzip2 (INT)	input.source 280	0.888
403.gcc (INT)	166.i	1.055
416.gamess (FP)	cytosine.2.config	1.929
429.mcf (INT)	inp.in	0.105
433.milc (FP)	su3imp.in	0.459
444.namd (FP)	namd.input	1.860
445.gobmk (INT)	13x13.tst	0.766
456.hmmer (INT)	nph3.hmm	2.477
458.sjeng (INT)	ref.txt	1.321
464.h264ref (INT)	foreman_ref_encoder_baseline.cfg	1.312
470.lbm (FP)	reference.dat	0.748

Table 3: Benchmarks used for evaluation. Top: CPU2000, Bottom: CPU2006. INT: 12, FP: 7, Total: 19.

In our experiments, we first use *Baseline_6_64* as the baseline to gauge the impact of adding a value predictor only. Then,

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

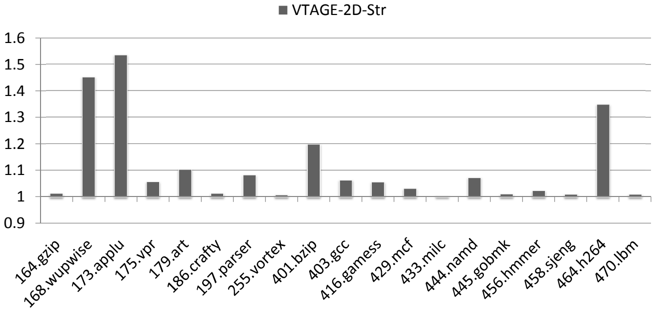


Figure 6: Speedup over *Baseline_6_64* brought by Value Prediction using VTAGE-2DStride as the predictor.

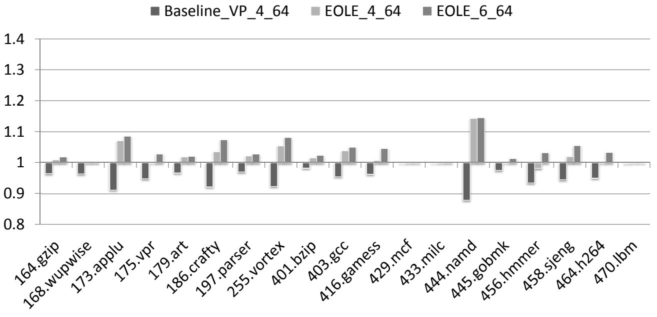


Figure 7: Performance of EOLE and the baseline with regard to issue width, normalized to *Baseline_VP_6_64*.

in all subsequent experiments, we use said baseline augmented with the predictor presented in Table 2. We refer to it as the *Baseline_VP_6_64* configuration. Our objective is to characterize the potential of EOLE at decreasing the complexity of the OoO engine. We assume that 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

Fig. 6 illustrates the performance benefit of augmenting the baseline processor with the VTAGE-2DStride predictor. A few benchmarks present interesting potential e.g. *wupwise, applu, bzip, h264*, some a more moderate potential e.g. *vpr, art, gamess, gcc, namd* and a few others low potential. No slowdown is observed.

In further experiments, illustration in performance figures will be speedups over the baseline described in Table 1, featuring the VTAGE-2DStride value predictor of Table 2: *Baseline_VP_6_64*.

5.2. Issue Width Impact on EOLE

Applying EOLE without modifying the OoO core (*EOLE_6_64*) is illustrated in Fig. 7. In this figure, we also illustrate the baseline, but with a 4-issue OoO engine (*Baseline_VP_4_64*) and EOLE using a 4-issue OoO engine (*EOLE_4_64*).

By itself, EOLE slightly increases performance over the baseline, with a few benchmarks achieving 5% speedup or higher. The particular case of *namd* is worth to be noted as with VP, 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.

Shrinking the issue width to 4 reduces the performance of the baseline by a significant factor on many applications, e.g. *applu, crafty, vortex, namd, hmmer* and *sjeng* for which slowdown is more than 5% (up to 12% for *namd*). For EOLE, such a shrink only reduces performance by a few percent compared with *EOLE_6_64*. Furthermore, *EOLE_4_64* still performs slightly higher than *Baseline_VP_6_64* in several benchmarks e.g. *applu, vortex* and *namd*. A single slowdown of 1.8% is reported for *hmmer*.

As a result, EOLE can be considered as a mean to reduce issue width without significantly impacting performance on a processor featuring VP.

5.3. Impact of Instruction queue size

In Fig. 8, we illustrate the respective performance of shrinking the instruction queue size from 64 to 48 entries for the baseline and EOLE.

A few benchmarks are quite sensitive to such a shrink for *Baseline_VP_6_48* e.g. *wupwise, applu, crafty, vortex, namd, hmmer* and *h264*. On the other hand, *EOLE_VP_6_48* does not always exhibit the same behavior. Most applications encounter only minor losses with *EOLE_6_48* (less than 5% except for *hmmer* and *h264*) and higher losses with *Baseline_6_48*, e.g. *applu* with 4% speedup against 9% slowdown, or *namd* with around 13% speedup against 9% slowdown.

In practice, the benefit of EOLE is greatly influenced by the proportion of instructions that are not sent to the OoO engine. For instance *namd* needs a 64-entry IQ in the baseline case, but since it is an application for which many instructions are predicted or early-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. Nonetheless, with *EOLE_6_48*, slowdown is limited to 5% at most for all but one benchmark, *hmmer*, for which slowdown is around 10%.

5.4. Summary

EOLE provides opportunities for either slightly improving the performance over a VP-augmented processor without increasing the complexity of the OoO engine, or reaching the same level of performance with a significantly reduced OoO engine complexity. In the latter case, reducing the issue width is our favored direction as it addresses scheduler complexity, PRF complexity and bypass complexity. EOLE also mitigates the performance loss associated with a reduction of the instruction queue size.

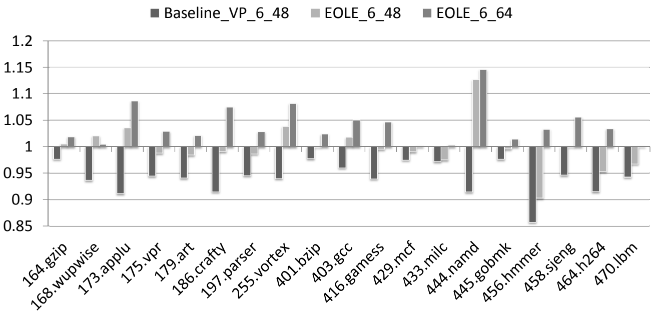


Figure 8: Performance of EOLE and the baseline with regard to the number of entries in the IQ, normalized to *Baseline_VP_6_64*.

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 OoO engine without impairing performance. On the other hand, 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 OoO 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 OoO 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 issue width 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 [17].

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 [16] or the *Tag Elimination* of Ernst and Austin [6]. 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 instructions to "catch" their operands as they are produced and thus execute back-to-back. However, a full bypass network is very expensive, especially as the issue width – hence the number of functional units – increases. Ahuja et al. showed that partial bypassing could greatly impediment performance, even for a simple in-order single-issue pipeline [1]. Consequently, in the context of a wide-issue OoO superscalar with a multi-cycle register read, missing bypass paths may cripple performance even more.

EOLE allows to reduce the issue width in the OoO engine. Therefore, it reduces the design complexity of a full bypass by reducing the number of ALUs and thus the number of simultaneous writers on the network.

A Limited Number of Register File Ports on the OoO Engine Through reducing the issue width on the OoO engine, EOLE mechanically reduces the number of read and write ports required on the PRF for regular OoO 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. Second, ALUs are needed for late execution. Last, 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 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 OoO engine (*EOLE_4_64*) would have to implement a PRF with a total of 12 write ports (resp. 8 for Early Execution and 4 for OoO

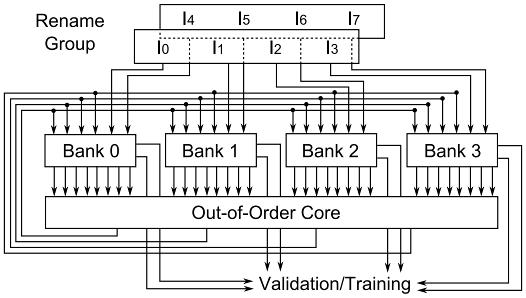


Figure 9: Organization of a 4-bank PRF supporting 8-wide Early Execution and 4-wide OoO issue. The additional read ports per-bank required for late execution and validation/training are not shown for clarity.

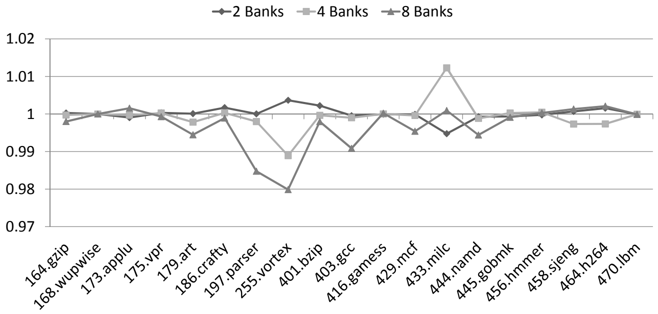


Figure 10: Performance of *EOLE_4_64* using a different number of banks in the PRF, normalized to *EOLE_4_64* with a single bank.

execution) and 24 read ports (resp. 8 for OoO 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 [41]. 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 (*Baseline_6_64*) PRF. Moreover, this would also translate in largely increased power consumption and longer access time, thus impairing cycle time and/or lengthening the register file access pipeline.

Without any optimization, *Baseline_VP_6_64* would necessitate 14 write ports (resp. 8 to write predictions and 6 for the OoO engine) and 20 read ports (resp. 8 for validation/training and 12 for the OoO engine), i.e. slightly less than *EOLE_4_64*. In both cases, this overhead might be considered as prohibitive in terms of silicon area, power consumption and access time.

However, 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

Mitigating Early-Execution Hardware Cost Because Early Executed instructions are treated 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. 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 Fig. 9 for an 8-wide Rename/Early Execute, 4-issue OoO core. Interestingly, this would add-up to the number of write ports required by a baseline 6-issue OoO Core.

In Fig. 10, 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[9]. We observe that the performance loss associated with load unbalancing is quite limited for our benchmark set, and using 4 banks of 64 registers instead of a single bank of 256 registers appears as a reasonable tradeoff.

Note that register file banking is also a solution for a practical implementation of a core featuring Value Prediction without EOLE.

Narrow 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 LE/VT read ports on each register file bank to 4 results in a marginal performance loss. Fig. 11 illustrates the performance of *EOLE_4_64* 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 additional 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_64*. Interestingly, adding 4 read ports adds up to a total of 12 read ports per bank (8 for OoO execution and 4 for LE/VT), that is, the

[9] 8 banks were simulated. However, there could be rare situations where the whole set of architectural registers would be allocated to a single bank, leading to major functionality issues.

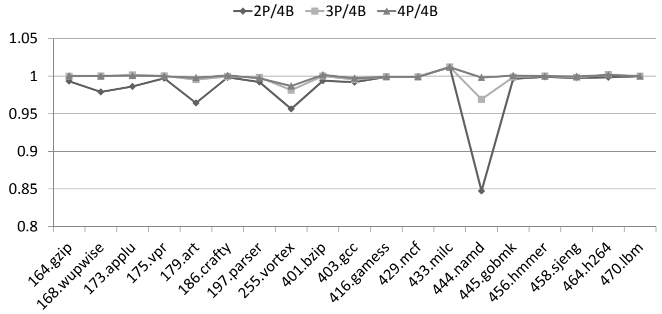


Figure 11: Performance of *EOLE_4_64* (4-bank PRF) when the number of read ports dedicated to late execution and validation/training is limited, normalized to *EOLE_4_64* (1-bank PRF) with enough ports for full width LE/validation.

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 speedup is 0.97 at worst (in *namd*).

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.

The Overall Complexity of the Register File Interestingly, on *EOLE_4_64*, 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 OoO engine and 4 for LE/VT) and 6 write ports (4 for the OoO 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 OoO engine. The only remaining difficulty comes from banking the PRF. Nonetheless, according to the previously mentioned area cost formula [41], 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.

It should also be mentioned that the EOLE structure naturally leads to a distributed register file organization with one file servicing reads from the OoO 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 OoO engine would be less likely to become a temperature hotspot than in a conventional design.

Further Possible Hardware Optimizations It might be possible to further limit the number of effective write ports on the PRF required by Early-Execution and Value Prediction as many μ-ops are not predicted or early executed. Hence, they do not generate any writes on the PRF and one could therefore limit the number of μ-ops that write to the PRF at the exit of the EE stage. The number of ALUs in said EE stage could also be limited. Specifically, μ-ops and their predicted/computed results could be buffered after the Early Execution/Rename stage. Dispatch groups of up to 8 μ-ops would be built, with the extra constraint of a limited number of at most 4 Early Execute or prediction writes on the PRF per dispatch group.

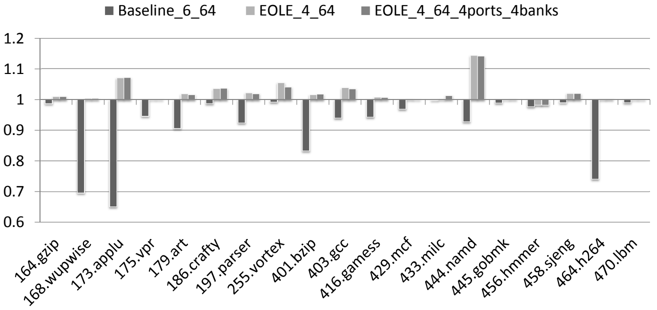


Figure 12: Performance of *EOLE_4_64* using 4 ports for late execution and validation/training and having 4 64-register banks, *EOLE_4_64* with 16 ports for LE/validation and a single bank and *Baseline_6_64*, normalized to *Baseline_VP_6_64*.

As already mentioned, it is also possible to limit the number of read ports on each register file bank to 3 with only marginal performance loss (with the exception of 3% on *namd*).

6.4. Summary

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 register file [25]. 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 OoO 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 OoO engine. As a result, one can expect many advantages in the design of the OoO 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 [6], 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 relatively simple 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 in the Late Execution stage. A full rank of ALUs is actually unlikely to be needed. From Fig. 2,

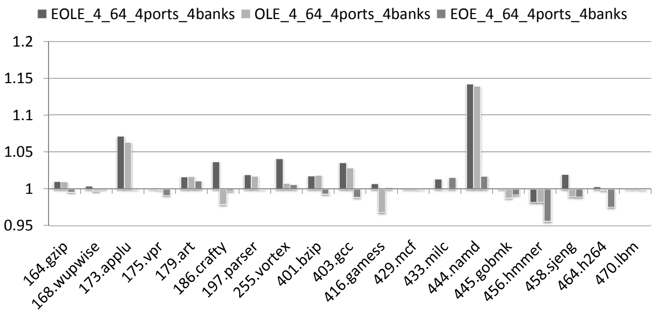


Figure 13: Performance of *EOLE_4_64*, *OLE_4_64* and *EOE_4_64* using 4 ports for LE/VT and having 4 64-register banks, normalized to *Baseline_VP_6_64*.

we presume that a rank of 4 ALUs would be sufficient.

Furthermore, implementing EOLE will not 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 OoO 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 Fig. 12) while EOLE enables a much simpler and far less power hungry OoO 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.

6.5. 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 8-to-8 bypass network anymore.

Fig. 13 shows the respective speedups of *EOLE_4_64*, *OLE_4_64* (Late Execution only) and *EOE_4_64* (Early Execution only) over *Baseline_VP_6_64*. 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 some are more sensitive to the absence of Early Execution (e.g. *crafty* and *gamess*). Nonetheless, the performance impact of removing Late Execution appears as more important in the general case.

Moreover, slowdown over *Baseline_VP_6_64* remains under 5% in all cases. This suggests that when considering an effective implementation of VP using EOLE, an additional degree of freedom exists as either only Early or Late Execution may be implemented.

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 [25]. 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 OoO 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 OoO 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 OoO engine. As a result, EOLE significantly reduces the number of instructions dispatched to the OoO engine.

Considering a 6-wide, 64-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 OoO engine and the PRF. EOLE achieves performance very close to the baseline using only a 4-issue, 48-entry IQ OoO engine. It achieves similar or higher performance when using a 4-issue, 64-entry IQ engine, with the exception of one benchmark, *hmmer* (1.8% slowdown).

With EOLE, the overhead over a 6-wide, 64-entry IQ processor (without VP) essentially consists of relatively simple hardware components, the two set of ALUs in the Early and Late Execution, 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 OoO 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 OoO 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 slightly 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 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 OoO engine such as the *Multicluster* architecture [8] or register file-oriented optimizations [38]. Finally, future research includes the need to look for more storage-effective value prediction schemes as well as even more accurate predictors.

Acknowledgments

This work was partially supported by the European Research Council Advanced Grant DAL No. 267175.

References

[1] P. Ahuja, D. Clark, and A. Rogers, "The performance impact of incomplete bypassing in processor pipelines," in *the International Symposium on Microarchitecture*, 1995.

[2] T. M. Austin, "DIVA: a reliable substrate for deep submicron microArchitecture design," in *the International Symposium on Microarchitecture*, 1999.

[3] 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, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[4] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *the International Symposium on Computer Architecture*, 1998.

[5] R. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 37, no. 4, pp. 547–564, 1993.

[6] D. Ernst and T. Austin, "Efficient dynamic scheduling through tag elimination," in *the International Symposium on Computer Architecture*, 2002.

[7] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, "Continuous optimization," in *the International Symposium on Computer Architecture*, 2005.

[8] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The Multicluster Architecture: reducing cycle time through partitioning," in *the International Symposium on Microarchitecture*, 1997.

[9] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *the International Symposium on Computer Architecture*, 2001.

[10] F. Gabbay and A. Mendelson, "Using value prediction to increase the power of speculative execution hardware," *ACM Trans. Comput. Syst.*, vol. 16, no. 3, pp. 234–270, Aug. 1998.

[11] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine, "The Intel Pentium M processor: MicroArchitecture and performance," *Intel Technology Journal*, vol. 7, May 2003.

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

[13] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2012.

[14] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *the International Symposium on Microarchitecture*, 1998.

[15] R. E. Kessler, E. J. Mclellan, and D. A. Webb, "The Alpha 21264 microprocessor Architecture," in *the International Conference on Computer Design*, 1998.

[16] I. Kim and M. H. Lipasti, "Half-price Architecture," in *the International Symposium on Computer Architecture*, 2003.

[17] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *the International Symposium on High Performance Computer Architecture*, 2004.

[18] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *the Annual International Symposium on Microarchitecture*, 1996.

[19] M. Lipasti, C. Wilkerson, and J. Shen, "Value locality and load value prediction," *the International conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[20] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *the International Symposium on Microarchitecture*, 2012.

[21] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti, "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing," in *the International Symposium on Microarchitecture*, 2001.

[22] A. Mendelson and F. Gabbay, "Speculative execution based on value prediction," Technion-Israel Institute of Technology, Tech. Rep. TR1080, 1997.

[23] T. Nakra, R. Gupta, and M. Soffa, "Global context-based value prediction," in *the International Symposium On High-Performance Computer Architecture*, 1999, pp. 4–12.

[24] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *the International Symposium on Computer Architecture*, 1997.

[25] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2014.

[26] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *the International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[27] V. Petric, T. Sha, and A. Roth, "Reno: a rename-based instruction optimizer," in *the International Symposium on Computer Architecture*, 2005.

[28] B. Rychlik, J. Faistl, B. Krug, A. Kurland, J. Sung, M. Velev, and J. Shen, "Efficient and accurate value prediction using dynamic classification," *Carnegie Mellon University, CMμART-1998-01*, 1998.

[29] Y. Sazeides and J. Smith, "The predictability of data values," in *the International Symposium on Microarchitecture*, 1997.

[30] A. Seznec, "Storage free confidence estimation for the TAGE branch predictor," in *the International Symposium on High Performance Computer Architecture*, 2011.

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

[32] A. Seznec, E. Toullec, and O. Rochecouste, "Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors," in *the International Symposium on Microarchitecture*, 2002.

[33] Standard Performance Evaluation Corporation. CPU2000. Available: http://www.spec.org/cpu2000/

[34] Standard Performance Evaluation Corporation. CPU2006. Available: http://www.spec.org/cpu2006/

[35] R. Thomas and M. Franklin, "Using dataflow based context for accurate value prediction," in *the International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[36] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *the International Symposium on Computer Architecture*, 2008.

[37] E. S. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *the Conference on Parallel Architectures and Compilation Techniques*, 2002.

[38] S. Wallace and N. Bagherzadeh, "A scalable register file Architecture for dynamically scheduled processors," in *the International Conference on Parallel architectures and Compilation Techniques*, 1996.

[39] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *the International Symposium on Microrchitecture*, 1997.

[40] H. Zhou, J. Flanagan, and T. M. Conte, "Detecting global stride locality in value streams," in *the International Symposium on Computer Architecture*, 2003.

[41] V. Zyuban and P. Kogge, "The energy complexity of register files," in *the International Symposium on Low Power Electronics and Design*, 1998, pp. 305–310.