Leveraging Targeted Value Prediction to Unlock New Hardware Strength Reduction Potential

Arthur Perais

Leveraging Targeted Value Prediction to Unlock New Hardware Strength Reduction Potential

Arthur Perais
arthur.perais@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA
38000 Grenoble, France

ABSTRACT

Value Prediction (VP) is a microarchitectural technique that speculatively breaks data dependencies to increase the available Instruction Level Parallelism (ILP) in general purpose processors. Despite recent proposals, VP remains expensive and has intricate interactions with several stages of the classical superscalar pipeline. In this paper, we revisit and simplify VP by leveraging the irregular distribution of the values produced during the execution of common programs.

First, we demonstrate that a reasonable fraction of the performance uplift brought by a full VP infrastructure can be obtained by predicting only a few "usual suspects" values. Furthermore, we show that doing so allows to greatly simplify VP operation as well as reduce the value predictor footprint. Lastly, we show that these Minimal and Targeted VP infrastructures conceptually enable *Speculative Strength Reduction* (SpSR), a rename-time optimization whereby instructions can disappear at rename in the presence of specific operand values.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**.

KEYWORDS

Microarchitecture, speculation, value prediction, performance

ACM Reference Format:
Arthur Perais. 2021. Leveraging Targeted Value Prediction to Unlock New Hardware Strength Reduction Potential. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480050

1 INTRODUCTION

In modern general purpose processors, high sequential performance is achieved by building on three axioms : 1) Provide high instruction fetch throughput at low latency 2) Provide high data fetch throughput at low latency and 3) Minimize structural hazards that prevent executing otherwise ready instructions.

Arguably, the first item is the most impactful on performance, as even the most aggressive out-of-order execution engine is not able to do much if there are no instructions in the pipeline. To address the two other items, bigger instruction windows are built to be able to perform work under long latency data misses, and more intricate data prefetching schemes are tied to bigger and deeper memory hierarchies to minimize the number of data cache misses.

However, data dependencies between instructions – which express program semantics – oftentimes lead to instruction window stalls where, for instance, one long latency miss is stuck at the head of the Reorder Buffer and a single dependency chain resides in the scheduler. While increasing the scheduler size improves performance in this case, it is unlikely to completely solve the problem. Indeed, allowing a second dependency chain to enter the scheduler may require a scheduler that is order of magnitudes bigger than the ones that are currently implemented. Moreover, the scheduler is notorious for its inability to scale [31].

An alternative is therefore to predict instructions results in the frontend to allow dependents to execute, even though the actual result is not available. This technique, known as *Value Prediction* (VP) has potential to increase performance in this specific example but also in general, by virtue of increasing the available Instruction Level Parallelism (ILP) beyond what exists in the program.

Value Prediction is not a new proposal : there have been many contributions in that area [4, 7, 8, 13, 14, 23, 26, 30, 33–35, 42, 43, 46, 47, 49, 51, 53]. Common wisdom suggests that while there is performance to be gained from VP, the hardware and complexity overhead of the technique is not yet worth the hassle. This is especially true as other less complex techniques may still be used to improve performance, for instance improving branch prediction, prefetching, or increasing the size of microarchitectural buffers.

In this work, we first demonstrate that focusing on predicting only a handful of distinct values permits to significantly reduce the value predictor footprint, but more importantly, to remove the additional write ports on the register file that are usually required to insert predicted values into the execution engine. This narrowing of scope further simplifies prediction validation by permitting to validate in-place at the functional unit, with one possible design not requiring reading an additional operand from the physical register file (or dedicated storage) just to validate the prediction and train the predictor.

Second, we build on this simpler VP infrastructure and introduce the concept of *Speculative Strength Reduction* (SpSR). The concept is similar to strength reduction as used in compilers whereby a complex operation (e.g., division by 2) is replaced by a "weaker" but semantically equivalent operation (e.g., right shift by 1). However, SpSR is implemented at the microarchitectural level and dynamically swaps a complex instruction for a simple one, based on predicted operands. For instance, the ARMv8 instruction *add x0,*

*Institute of Engineering Univ. Grenoble Alpes

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480050

x0, x1 can be speculatively strength reduced to a *nop* if x1 is predicted to be 0x0. In addition to appearing as having 0-cycle latency, speculatively strength reduced instruction do not require execution resources (physical register, scheduler entry, issue slot) in the execution engine. This may further improve performance by allowing other instructions to consume those resources rather than block on a structural stall.

2 GENERIC VALUE PREDICTION

Value prediction is usually performed through the addition of a hardware value predictor that associates a prediction to a given instruction. Arbitrary state can be leveraged to generate a prediction. For instance, the VTAGE value predictor [34] will hash the PC of the instruction with the global branch history to generate the index at which the candidate prediction resides in the predictor tables. Other predictors, such as FCM [43] rely on the value history (i.e., the last *n* values produced by a given PC) to identify correlations.

Value prediction operates in three steps. First, a prediction is generated by the value predictor. Second, the prediction is validated by comparing it against the computed result, and if incorrect, corrective steps are taken. Third, the value predictor is updated using the computed value and the outcome – correct or incorrect. Conditional branch prediction and indirect branch target prediction operate similarly. In spirit, indirect branch target prediction is closely related to value prediction.

In this context, one may ask why processors are able to predict a 64-bit target for an indirect branch, compare it against the value of the branch source register and update the predictor, yet do not implement value prediction, despite the mechanisms being quite similar at first glance.

Observation I : *The need for branch prediction is a fact of life in pipelined microarchitectures implementing von Neumann architectures, but the need for value prediction is not.* In other words, the hardware cost of branch prediction **has** to be paid even to attain medium-range performance, while value prediction is not always useful and its benefits appear highly workload dependent. Yet, area investment to implement VP is significant and paid even if VP is not beneficial.

Observation II : *Value predictions have multiple consumers.* This is a key difference with branch prediction in which only the predicted branch consumes the prediction when it executes. Conversely, in VP, many dependent instructions may need to consume the prediction so they can execute earlier. This entails that value predictions cannot just be carried down the pipeline by the predicted instructions themselves. Rather, they need to be written to storage accessible by the functional units, be it the physical register file itself, or dedicated storage [46].

Given those two initial high-level observations, we argue that a VP infrastructure should be constructed as a fairly inexpensive microarchitectural add-on that provide noticeable performance improvements in some workloads. In the next paragraphs, we highlight other source of VP complexities to strengthen this argument.

2.1 Value Predictors

Algorithms. Some value prediction algorithms rely on the value of instance $n-1$ to generate a prediction for instance n. For instance, in Stride, D-FCM [14] or D-VTAGE [35], the prediction is the previous result plus a certain stride. In deep pipelines, many instances of the same instruction can be live at any given time. In this case, using the most recent *retired* previous result will lead to incorrect predictions. Consequently, this type of predictors need to track speculative state, whether in a window of previous results or a per-entry counter of live instruction (to multiply the stride with). In spirit, this is similar to the speculative update of local branch histories in branch predictors, which is not straightforward [48]. For Stride-based value predictors, a fully associative, priority-encoded structure can be used to implement the speculative window [35], although the overhead of such a structure will increase with the instruction window size. Predictors that use any form of value history to correlate are also subject to this shortcoming.

Storage. While there exist proposals that suggest predicting only narrow values [42] or partition predictor structures based on actual prediction width [27], many recent proposals attempt to predict arbitrary general purpose register data, i.e., 64-bit worth of data per instruction. This mechanically increases the bitcount of the value predictor as compared to the conditional branch predictor, if we assume both use the same structure and number of entries. This issue compounds with the fact that there are in general more instructions candidate for value prediction than conditional branches.

2.2 Predictions in the Pipeline

Generation. Predictions that are generated in the frontend of the pipeline should be made available to dependent instructions so they can execute early. This entails writing predictions to either the Physical Register File (PRF) itself[1] or to a dedicated storage area that can be read by issuing instructions [46]. The former requires additional write ports on the PRF, while the latter adds a third source of operand data for execution (the bypass network being the second). Both require that predicted instructions carry their predictions down the pipeline until such time the prediction can be written to the staging area (usually, after the Rename stage). If *Virtual Registers* are used, [15], predicted instructions cannot easily write their predictions in the PRF at Rename since virtual registers are bound to physical registers at Issue or Writeback.

An alternative, if the microarchitecture supports it, is to inject *move immediate* μ-ops[2] in the instruction stream to perform the register write in the backend through the existing datapath. However, this implies that all value predicted instructions will *a minima* consume a second scheduler entry and issue slot.

Validation. The least involved way to validate predictions is to inject dedicated *compare* μ-ops in the pipeline. When combined to the injection of *move immediate* μ-ops to write value predictions in the physical register file, we can simply have the *compare* μ-op compare the result of the predicted instruction against the result of

[1] Or the ROB if the microarchitecture uses the ROB to store speculative register values.
[2] To keep the discussion ISA-agnostic, we consider that architectural instructions are transformed into one or multiple microarchitecture-specific operations (μ-ops) at Decode, before entering the processor backend. μ-op injection would therefore happens at Decode.

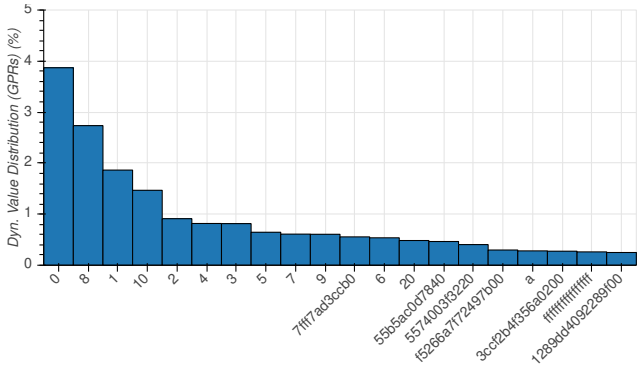


Figure 1: Distribution of values produced by instructions writing to general purpose registers (SPEC2k17 [6] *speed*, train inputs, x86 ISA). From [32].

the *move immediate* μ-op, and resteer the pipeline on a mismatch. All consumers of the predicted instructions will have been made dependent on the *move immediate* destination physical register at Rename. This scheme does not increase port requirements on the PRF and does not require additional hardware at the functional units to validate predictions. However, it may require a predicted instruction to occupy three entries in the scheduler (instruction itself, *move immediate* μ-op and *compare* μ-op), and to consume an additional physical register (one for the instruction itself, one for the *move immediate* μ-op). Scheduler entries and physical registers are critical resources and increasing pressure on these resources just to save a single cycle (e.g., if an addition is value predicted) is most likely not a good tradeoff. In other words, *validating arbitrary value predictions without specific changes to the pipeline will incur overheads on-par with the performance gains expected from these (correct) predictions.*

Consequently, hardware support is commonly assumed to validate predictions in place at the functional unit, during execution. In this case, the predicted value needs to be read by the predicted instruction, in addition to its regular operands. To avoid reading the prediction from the PRF at issue time and therefore increase PRF read port requirements, the prediction may also be read from the FIFO that tracks inflight VP state and is used to perform predictor updates after retirement. Alternatively, the prediction may be carried by the instruction into dedicated scheduler storage.

Finally, predictions may also be validated at retirement [33, 34]: predictions are read from the VP state tracking FIFO at retirement and compared against the actual value that has been written back to the PRF, incurring an additional PRF read during the lifetime of the predicted instruction.

Recovery Scheme. Much like memory dependency prediction, value prediction speculates on the dataflow. As a result, on a misprediction, the instructions in the pipeline are still on the correct path, even though they are using incorrect data. The bare minimum needed to recover is therefore to re-execute direct and indirect consumers of the mispredicted instruction. This technique, known as *replay* can be selective (replay only direct/indirect consumers) or coarse grain (replay all younger instructions), but is generally less expensive performance-wise than flushing the pipeline. However, *replay* is known to be complex to implement efficiently and can lead to "replay tornadoes" if the replay wavefront cannot catch up to the speculative wavefront of executing instructions [24]. As a result, flushing the pipeline may be preferred as a simpler – if slower – alternative. Perais and Seznec go as far as suggest to perform prediction validation and a full pipeline flush at retire time to prevent significant modifications to the out-of-order execution engine [33, 34].

2.3 Summary

While many proposals go to great length to reduce hardware overhead, many aspect of "generic" value prediction introduce complexity in the processor, either by requiring large prediction structures [35], structures to identify critical instructions that should be predicted in priority [4] or adding read/write ports to the PRF [33]. In this paper, we argue that a more limited form of value prediction still has potential to improve performance while leveraging existing mechanisms to handle predictions in the pipeline. This proposal requires a value predictor with a much lower storage footprint than *Generic VP* (GVP), and is notably oblivious to the use of Virtual Registers [15]. We also demonstrate that this *targeted* implementation of value prediction enables a novel rename-time optimization : Speculative Strength Reduction (SpSR).

3 TARGETED VALUE PREDICTION

A key difficulty of value prediction is that a prediction has to be made available to multiple consumers, for instance by writing it directly into the physical register file. In this section, we will demonstrate that limiting ourselves to few distinct values simplifies this process.

3.1 Minimal Value Prediction (MVP)

The distribution of values manipulated by general purpose programs is biased towards a few specific values [52], as shown for instructions of SPEC CPU 2k17 in Fig. 1. While some of those values are related to the algorithm itself, the most heavily produced value is 0x0, and 0x1 is third. Evidently, the frequency at which a value is produced does not say anything about how often it can be correctly predicted, or how beneficial predicting it will be. However, 0x0 and 0x1 will often participate in boolean computations, which are often predictable since branch directions are generally predictable. Therefore, those values appear as reasonable candidates if the goal is to limit the number of distinct values that can be predicted.

Focusing on 0x0 and 0x1 allows us to "write" predictions to physical registers implicitly through hardwired physical registers. Indeed, by implementing a physical register whose value is always 0x0 (resp. 0x1), we can simply rename the destination register of an instruction predicted to produce 0x0 (resp. 0x1) to the relevant hardwired register. In fact, several ISAs already define a zero architectural register whose value is always 0x0, and any microarchitecture implementing such ISA must therefore implement a hardwired zero register. Even in x86, modern microarchitectures implement a hardwired zero register (resp. one register) to perform zero-idiom (resp. one-idiom) elimination [18].

3.2 Targeted Value Prediction (TVP)

Focusing only on two values may be a reasonable tradeoff, but Fig. 1 clearly depicts that many narrow values are also produced often. As a result, we propose to leverage physical register names to either encode a physical register name, or a small constant. This idea, *register inlining*, was already proposed in another context by Lipasti et al. [25], but was never considered in the context of Value Prediction.

Specifically, to implement out-of-order execution, instructions are allocated an entry in the scheduler. There, they track the readiness of their source operands by matching their source physical register identifiers or *names* against the physical register names being produced each cycle. A scheduler entry already tracks several source physical register names as well as one or multiple destination register names (depending on the ISA and microarchitecture). Modern high-performance processors feature between 256 and 512 physical registers, therefore, each physical register name is 9-bit (assuming no virtual register [15] scheme is implemented). By increasing the size of physical register names by one bit, we can overload physical register names so they can be interpreted as actual values, alleviating the need for providing write ports to the PRF to write predictions. Note that the larger names are only used in Rename structures. Notably, the wakeup logic of the scheduler does not need to use the additional bit in the CAM hardware that determines if a broadcast name matches operands stored in entries.

3.2.1 Renaming Hardware. To demonstrate that targeted Value Prediction requires only that we widen physical register names by one bit, we assume a baseline implementation of renaming that uses a Committed Register Alias Table (CRAT) in addition to the speculative Register Alias Table (RAT), without loss of generality[3].

Register Allocation. Overloading register names to represent small values essentially causes the RAT/CRAT to act as physical register files. For instance, consider an instruction that is value predicted to produce 0x42. The associated physical register "name" will be 0x242, as physical register names are now 10-bit (from 9-bit) and the most significant bit is always set if the physical name represents a value, and unset for physical register names.

This instruction then associates its architectural destination register (e.g., $x0$) to physical "name" 0x242 in the RAT, instead of fetching a new physical register name from the Free List. Dependents then read the RAT entry for $x0$ and retrieve 0x242, which they will know to interpret as the value 0x42. Consequently, when a dependent is dispatched to the scheduler, it will not mark itself waiting on physical register 0x42. When the dependent is eventually selected for issue, the value can be muxed to the functional unit, with bit 9 of the overloaded name controlling the mux.

Lastly, when the predicted instruction executes, the produced value is checked against the physical destination register name (which is the predicted value). Assuming the prediction is correct, no specific action has to be taken. Mispredictions are discussed in 3.4

[3]Correct RAT state can be still recovered without a CRAT by "undoing" wrong-path mappings stored in the Active List FIFO from the youngest mapping to the mispredicted instruction. Flash copy of RAT snapshots without iterative walks is also possible [1].

Register Reclamation. Physical register reclamation usually takes place when an instruction retires. At that point, the instruction moves the physical register currently mapped to its architectural destination register in the CRAT into the Free List, and overwrites the CRAT entry with its own physical destination register.

With Targeted Value Prediction, either of the CRAT physical register name or the physical destination register name of the committing instruction may be values. If the former, then the name in the CRAT is not put on the Free List. If the latter, then the algorithm is unchanged.

Pipeline Flush. To restore correct mappings on a pipeline flush, the RAT has to be repaired. This can be achieved through checkpointing [1] or by copying the CRAT to the RAT and iteratively re-applying mappings from an in-order queue (the Active List) to the RAT until the misprediction point is reached. In both cases, the use of small values instead of physical register names has no bearing on the recovery algorithm. It simply widens the names by a bit, which will increase the size of checkpoints or Active List entries accordingly.

3.2.2 Other Possible Optimization. The ability to use physical register names as values lets us eliminate *move immediate* instructions whose immediate fits in a signed 9-bit integer (in this paper) [25]. Essentially, this is *9-bit signed integer-idiom elimination*.

3.3 Impact on the Value Predictor

Predictor Design and Sizing. Minimal and Targeted Value Prediction have two effects on value predictor design. First, specific algorithms such as stride-based prediction become mostly irrelevant. This alleviates the need for speculative management of the predictor entries to account for the presence of several instances of the same instruction in the pipeline. Second, the bit count of each entry is reduced significantly. Indeed, the content of a predictor entry is generally comprised of a tag, a prediction and a confidence counter. The prediction size will greatly decrease, from 64-bit to 1-bit and 9-bit respectively. Considering the aggressive VTAGE predictor depicted in Table 2, this decreases storage from 55.2KB (64-bit) to 13.9KB (9-bit) and 7.9KB (1-bit), respectively.

Predictor Training. Abstractly, the Reorder Buffer may be used to track inflight value prediction state and update the predictor based on the outcome at retirement. In practice, since not all instructions are VP-eligible, a dedicated FIFO structure may be used [34]. Alternatively, the Active List can be augmented with storage to store value prediction information.

It should be noted that when using a predicted value in the pipeline, the physical destination register name *is* the predicted value, therefore, we simply have to compare it against the result of the functional unit, and update the the FIFO accordingly. While this assumes a dedicated comparator – potentially pipelined after the functional unit (FU) – is present in all execution lanes that can execute predicted instructions, this may allow the predictor to be more aggressive compared to [34], in which the cost of validating predictions at retire forces the predictor to be conservative.

However, during the training phase of a predictor entry, when predictions are generated but not yet used, the candidate instruction is still given a physical destination register. Therefore, we cannot

generally compare the physical destination register name against the output of the FU to determine if the prediction was correct. Fortunately, during training, the predicted instruction is the only consumer of the prediction, therefore, we assume a mechanism similar to indirect branch prediction where the prediction is either read from the FIFO at issue or carried with the instruction and placed in the scheduler payload. It is then compared against the result at execute, and the outcome is written to the corresponding FIFO entry. While this puts pressure on the FIFO, reads and writes are guaranteed never to collide and since we only predict arithmetic and load instructions, port requirements remain smaller than on the actual PRF (i.e. branch, store and INT to FP/SIMD conversion pipelines do not require ports on that FIFO). In MVP, an alternative "brute force" approach can be used during training : the result is always compared against both 0x0 and 0x1, and the outcome is used to update the corresponding FIFO entry.

3.4 Handling Mispredictions

MVP/TVP will detect mispredictions as early as possible, at execute time. To repair the microarchitectural state, *replay* (selective or not) is a possibility. However, in this paper, we focus on pipeline flush as it is the simplest scheme, and it is sufficient to demonstrate the idea. Microarchitectures that already implement replay for other reasons (e.g., speculative scheduling) may build VP recovery on top of it.

The key difference with Generic VP is that on a value misprediction, we must include the mispredicted instruction in the set of flushed instructions. The reason is that to predict an instruction, its destination register was either renamed to a hardwired register (MVP) or not allocated a physical register at all (TVP). Therefore, if the prediction is incorrect, the correct result cannot overwrite the hardwired value (MVP), or does not have a physical space to write the correct result (TVP) to. Consequently, the mispredicted instruction has to be renamed again. This is most easily achieved by flushing and refetching the mispredicted instruction.

3.4.1 Preventing Livelock. In MVP/TVP, the mispredicted instruction is included in the pipeline flush, and it is therefore the first instruction that will be fetched when the frontend restarts. If the value predictor provides the same – incorrect but confident – prediction to the new instruction, then the processor will livelock. Therefore, on a misprediction, we need to ensure that the mispredicting instruction does not immediately get value predicted again. Although this can be achieved through several means, we found that *silencing* the predictor for a small number of cycles is sufficient. *Silencing* means that the predictor may provide predictions for the purpose of training, but those will not be used by the pipeline, even if they are confident. In our experiments, we found that MVP/TVP/GVP perform similarly well by using a very small number of 15 silencing cycles, except in *roms* with TVP. In this particular case, TVP causes the L1D Stride prefetcher to issue additional prefetches that are detrimental to performance (-1.09%). Indeed, the gem5 [28] implementation does not currently throttle the Stride prefetcher if it does not perform well and does not dynamically adapt distance and degree, so occasional performance drops are not unexpected. GVP sees the same prefetcher behavior but compensates slowdown through increased prediction coverage. Regardless, to curb that effect, we use 250 silencing cycles in all cases, as we found it did not impact performance in MVP and GVP. We expect that the optimal silencing amount varies with pipeline geometry and benchmark, and a dynamic scheme would likely be beneficial. If needed, this scheme could be combined to a commit watchdog to guarantee liveness.

3.4.2 Silent Value Mispredictions. Existing VP proposals can prevent a value misprediction from causing corrective actions if the prediction has not yet been used by dependents when the mismatch is detected. In this case, the correct value can just overwrite the wrong prediction in the physical register. In MVP/TVP, this optimization is not possible since either writes to hardwired registers are nullified, and the correct value will therefore not be able to replace the incorrect prediction in the physical register (MVP), or there is no storage for the correct value (TVP). Therefore, incorrect predictions that have not yet been used by dependents at validation time still cause a pipeline flush.

3.5 Potential Gains

The main interest of VP is to speculatively break data dependencies, thus increasing the available ILP. However, in both MVP and TVP, Value Prediction also provides hardware resource savings as predicted instructions do not require a destination physical register.

Moreover, while a value predictor needs to be implemented and physical register names need to be widened by a bit in TVP, predicted instructions reduce PRF read and write pressure as predictions are stored in the RAT and inserted directly into consumers' scheduler entries "for free" in both MVP and TVP.

3.6 Other

In this work, we focus on the ARMv8 ISA. The ISA defines a relaxed memory model, which can theoretically lead to incorrect execution when VP is used in a multithreaded context [29]. To address this issue, one solution is to attach *acquire* semantics to all value predicted loads.

ARMv8 already features *load acquire* instructions, therefore, pre-existing hardware support can be expected. Moreover, loads younger than a *load-acquire* may still speculate past the barrier as long as they get squashed when receiving a relevant invalidation message. As a result, the cost of marking value predicted loads as *acquire* remains limited in a high-performance setting.

4 SPECULATIVE STRENGTH REDUCTION

Focusing on a few values to predict is of notable interest because many instructions have specific behavior when one of their source operands is 0x0 or 0x1.

As an example, we consider the ARMv8 instruction *add x0, x0, x1*. If *x1* is predicted to be 0x0, then the instruction becomes a *nop* (no changes to the architectural state). If *x0* is predicted to be 0x0, then the instruction becomes a *move*. Many instructions exhibit the same behavior.

Therefore, as a side effect of targeting 0x0 and 0x1 in MVP/TVP, we can uncover additional strength reducible instructions in the frontend. Such instructions are handled by existing register name

manipulation mechanisms (*move elimination*, or ME, and *zero/one-idiom elimination*). Since those instructions are "reduced" speculatively as a result of a value prediction, we refer to this technique as "Speculative Strength Reduction" (SpSR). This is in contrast with "Static Strength Reduction" (StSR) which is performed by the code generator (e.g., using a left shift instead of a multiply by a power of 2) and "Dynamic Strength Reduction" (DSR) which is performed by the microarchitecture but is not speculative (e.g., *move elimination* and *zero/one-idiom elimination*).

4.1 Potential Gains

Much like move eliminated and zero/one-idiom eliminated instructions, a speculative strength reduced instruction appears as having an execution latency of 0 cycle.

Moreover, eliminated instructions do not consume specific pipeline resources, which may help newer instructions enter the execution engine earlier. Indeed, an eliminated instructions requires neither a new physical destination register from the Free List nor a scheduler entry, nor an execution unit. This means that similarly to value predicted instructions, eliminated instructions will not dissipate power writing the PRF. In addition, eliminated instructions will not *read* the PRF and dissipate power being scheduled and executed.

4.2 Strength Reducibility

Regardless of the ISA, instructions eliminated through move elimination and zero/one-idiom elimination can only be eliminated if they have no side-effects, or because the side effects can be trivially performed at elimination time.

For instance, if we consider x86 *xor regd, regs*. The result of this instruction is always 0x0 if *regd* and *regs* are the same architectural register. However, the x86 *xor* instruction has side effects in that it must update the condition flags based on the outcome of the instruction [19]. Fortunately, the result is always 0x0, so the update to the condition flags can be performed directly at elimination time.

Another example is x86 *mov regd, regs* instruction. In this case, the outcome of the instruction is not known at elimination time, and can be any 64-bit value. If *mov* had had side effects similar to *xor*, then the elimination could not have been performed : the instruction would still have to read the source operand, compute the condition flags and then update them, although dependents on the destination register of the *mov* would still see a 0-cycle latency for the moved operand. Fortunately, x86's *mov* does not modify the condition flags, and it can therefore be *fully eliminated* [19]. In this context a *partially eliminated* (non fully eliminated) instruction does make its destination register available faster, as per the strength reduction, but an operation is still dispatched to the scheduler to perform the side effects, such as updating the condition flags. The notion of side effects actually plays a major role in the context of Speculative Strength Reduction, as ISAs are not created equal.

x86: Most arithmetic and logic instructions have side effects in x86 in that they write all or some of the condition flags [19]. Therefore, SpSR'd instructions that become *move* instructions can only be partially eliminated. While this may still improve performance by short-circuiting the production of the destination register, the resource savings will be limited : only the physical destination register is saved, but the instruction still has to dispatch to the scheduler, read its operand(s), execute, and write the condition flags.

ARMv8: contrarily to x86, most arithmetic and logic instructions do not modify the condition flags [3]. As a result, the vast majority of SpSR'd instructions can be fully-eliminated, including instructions that reduce to *move*. It should however be noted that in some cases, instructions with side effects can also be fully eliminated if the microarchitecture implements hardwired condition flags registers. For instance, *ands* is guaranteed to produce 0x0 if one source operand is 0x0. Therefore, the condition flags can be trivially computed and *ands* can be fully eliminated if a condition flags register hardwired to {N=0,Z=1,C=0,V=0} is implemented. The same is possible in x86, as long as the produced value is always known (e.g. *test* if one of the sources is 0x0).

RISC-V: Instructions that are candidate for SpSR do not have side effects [40]. There are no condition flags as branch instructions embed the comparison operation. As a result, all instructions of interest to SpSR may be fully eliminated.

In this work, we focus on the ARMv8 ISA. Table 1 depicts the SpSR idioms we implement in our experiments. Note that *csel/csneg/csinc* as well as conditional branches are considered to be SpSR candidates. This may seem counter-intuitive since they depend on the condition flags. However, we assume that we can keep track of the *NZCV* ARMv8 condition flags in the frontend when they are generated by an SpSR'd instruction. For instance, if an *ands x0, x1* sees that x1 is predicted to be 0x0, then it can be SpSR'd, and we can write {N=0,Z=1,C=0,V=0} into the NZCV register that is local to the frontend.[4] This register is invalidated as soon as the next condition flag writer is renamed (if it is not itself SpSR'd), but it allows instructions that depend on the condition flags to be SpSR'd as well. Specifically, *csinc/csneg/csel* can be reduced to a simple *move* if the condition is known, and conditional branches can be resolved early.

4.3 Hardware Overhead

The design of a fast and wide state-of-the-art renamer is out of the scope of this paper. However, we point out that SpSR does introduce additional complexity in the renamer.

Baseline superscalar renaming suffers from *intra-group* dependencies. That is, the physical source register of a younger instruction may be the non-yet renamed physical destination register of an older instruction in the same rename group. Consequently, the renaming circuit is expected to rename all sources via the RAT, and stitch any stale mapping combinationally if needed. Pipelining may be used to widen the circuit [41], but intra-group dependencies become even more likely as the superscalar degree increases.

In baseline superscalar renaming without any form of strength reduction, we can assume that the destination physical registers of all instructions in the rename group become available at the same time as all instructions speculatively retrieve a new physical register from the Free List concurrently.

[4] We also assume the presence of hardwired NZCV physical registers in the backend to allow the *ands* instruction to be fully reduced.

Table 1: Strength Reductions Considered in this Work

Instruction	src0 value		src1 value	
	0x0	0x1	0x0	0x1
sub dst, src0, #1	–	zero-idiom	–	–
sub dst, src0, src1	–	zero-idiom if src1 == 0x1	move-idiom	zero-idiom if src0 == 0x1
add/orr/xor dst, src0, #1	one-idiom	–	–	–
add/orr/xor dst, src0, src1	move-idiom	–	move-idiom	–
and dst, src0, #1	zero-idiom	one-idiom	–	–
and dst, src0, (src1 | #imm)	zero-idiom	–	zero-idiom	–
shr/shl dst, src0, #imm	zero-idiom	–	–	–
shr/shl dst, src0, src1	zero-idiom	–	move-idiom	–
ubfm dst, src0, ...	zero-idiom	–	–	–
bic dst, src0, (src1 | #imm)	zero-idiom	–	move-idiom	–
rbit dst, src0	zero-idiom	–	–	–
ands src0, src1	nop+NCZV	nop+NCZV if src1 == 0x1	nop+NCZV	nop+NCZV if src0 == 0x1
ands src0, #imm	nop+NCZV	–	–	–
subs/adds src0, src1	nop+NCZV if src1 == 0x0 or 0x1	nop+NCZV if src1 == 0x0 or 0x1	nop+NCZV if src0 == 0x0 or 0x1	nop+NCZV if src0 == 0x0 or 0x1
cbz/tbz src0	nop	nop	–	–
b.*cond*	nop if NCZV avail.			
csel src0, src1, *cond*	move-idiom if NCZV avail.			
csinc/csneg src0, src1, *cond*	move-idiom if NCZV avail and cond is true.			

With rename optimizations such as SpSR (or even plain ME), this is no longer true. Indeed, for an eliminated *move*, the physical destination register is its physical source register. Thus, a *move* may need to wait for its source physical operand to be updated by an older instruction in the same rename group before it can propagate its own physical destination register to newer instructions within the group. A pathological case would be a rename group full of dependent *move* instructions. Before the correct physical source register of the last *move* can be resolved, the physical destination register of the second to last *move* must be resolved, and so on.

SpSR furthers this issue because the strength reduction decision depends on the physical source registers. That is, with plain *move elimination*, the decision to strength reduce can be made prior to inspecting the physical source register of the instruction : the renamer knows that the instruction can be strength reduced *a priori* thanks to its opcode. Conversely, in SpSR, the decision to strength reduce is delayed until the physical source registers are known, which may be late if the source register has to propagate through a *move* chain. Therefore, the design of the renamer will likely need modifications to accommodate SpSR.

5 EVALUATION FRAMEWORK

We evaluate MVP/TVP and SpSR using the full-system gem5 simulator [28]. We compile SPEC2k17 CPU [6] speed source code to Aarch64 binaries using gcc 8.3 -O3,[5] and simulate ten 100M instruction Simpoints [17] representative of the first 100B instructions of each benchmark (*reference* inputs). The system is warmed-up for 50M instructions.

We initially attempted to model a pipeline resembling Apple's M1 processor core using publicly available reverse-engineered numbers from microbenchmarks [11] (630-entry ROB, 148-entry Load Queue, 106-entry Store Queue). However, using such a large instruction window led to performance inversions when implementing non-speculative techniques that can only improve instruction flow in the pipeline (e.g. *0/1-idiom* and *move elimination*). This is because faster execution on a highly speculative path can yield a net performance loss. Aragón et al. [2] showed through simulation that even for a Power 4-class processor (128-entry ROB), an oracle fetcher that does not fetch wrong-path instruction (but still mispredicts branches) provides 5% speedup in SPECint95 and SPECint2000. The use of bigger speculation windows than what was considered in [2] will likely amplify the trend. Aragón et al. provide heuristics aimed at reducing the power consumed on the wrong path, but do not manage to reclaim lost performance. To the best of our knowledge, there is no proposal explicitly tackling the performance impact of wrong-path on deep machines in the literature. As as result we chose to keep the functional units and caches unchanged, but reduced the size of the ROB/IQ/LSQ to mitigate wrong-path effects. The final machine configuration, which is depicted in Table 2, remains aggressive.

We implement *0/1-idiom* and *move elimination* in gem5 to prevent MVP/TVP and SpSR from artificially increasing performance. Move idioms include *add/eor/orr* with *xzr* as a source operand. Zero idioms include *eor* of a register with itself, *movz* with 0x0 as the immediate, and *and* with the zero register. We only consider *movz* with 0x1 as the immediate for one idioms. We assume unlimited reference counting for *move elimination*, as existing proposals for realistic reference counting [5, 36] achieve potential that is close to ideal. However, we do not support a 64-bit register being moved into a 32-bit register to avoid cases such as *add x0, x0, x1* followed by *eor w12, w0, wzr*, as a subsequent instruction such as *add x15, x12, x14* could read the full-width definition of $x0$ which would be incorrect. In practice, this elimination would be possible if the RAT provided up to date width information to consumers, allowing them to mask half the physical register when relevant. In our baseline, 10.5% of the move idioms are not eliminated due to a width mismatch, on average.

[5] aarch64-linux-gnu-gcc-8 (Debian 8.3.0-2) 8.3.0, 4.19.0-10-amd64 #1 SMP Debian 4.19.132-1 (2020-07-24) x86_64 GNU/Linux.

[6] This is aggressive but required by the gem5 infrastructure to obtain the 0-cycle taken branch penalty provided by decoupled fetching [38] and L0BTBs in modern designs [16].

[7] L1 TLB latency is accounted for in the L1 caches load to use.

Table 2: Gem5 Processor configuration (11-stages pipeline, 3Ghz)

Branch Prediction	32KB, 1+15-table TAGE predictor [44] Min/Max Hist. length : 5/640, 8192-entry BTB 1k-entry Indirect Branch Target Cache 32-entry Return Address Stack
Value Prediction	**Optional** 1+7-table VTAGE predictor [34] Min/Max Hist. length : 2/128 – Log2 sizes: 12, 9, 9, 8, 8, 8, 7 7 – Tag width: 4, 9, 9, 10, 10, 11, 11, 12 3-bit FPC confidence [34, 39], 2-bit *useful* field [44] Gen. VP: **55.2KB** ($\frac{1}{16}$ FPC proba, 250 silenced cycles on misp.) Tar. VP: **13.9KB** ($\frac{1}{16}$ FPC proba, 250 silenced cycles on misp.) Min. VP: **7.9KB** ($\frac{1}{16}$ FPC proba, 250 silenced cycles on misp.)
Fetch	16-wide fetch from 64B Line Buffer 32-instruction fetch queue, 1-cycle taken branch penalty, 3-cycle Fetch to Decode
Decode	8-wide, Mistarget detection (BTB miss) 1-cycle Decode to Rename
Rename	8-wide, 2-cycle Rename to Dispatch 0/1-idiom elimination, move-elimination **Optional** support for 9-bit-idiom elimination, SpSR
Dispatch/ Commit	8-wide, 315-entry Reorder Buffer, 92-entry Instruction Queue, 74-entry Load Queue, 53-entry Store Queue, 292 INT Regs, 292 FP/SIMD Regs
Issue	Up to 15 instructions per cycle into 4 simple ALU, 2 (simple ALU + IntMul(3c)), 1 IntDiv(20c, not pipelined) 3 (simple FP/SIMD(3c) + FP/SIMD Mul(4c mul/5c mac)) 1 (simple FP/SIMD(3c) + FP/SIMD Mul(4c mul/5c mac) + FP/SIMD Div (12c, not pipelined)), 2 Loads, 2 Stores Store Sets [9] mem. dep. pred. (2k-entry LFST, 2k-entry SSIT)
Caches	128KB 8-way L1D, 64B line size, 4c load-to-use, 56 MSHRs, LRU 128KB 8-way L1I, 64B line size, 1c[6] load-to-use, 8 MSHRs, LRU 1MB 8-way L2, 64B line size, 12c load-to-use, 64 MSHRs, LRU 8MB 16-way L3, 64B line size, 37c load-to-use, 64 MSHRs, LRU
TLBs	256-entry 1-way L1I (0c)[7] + 256-entry 1-way L1D (0c)[4] TLBs 3072-entry 12-way L2TLB, 4 cycles, LRU
Prefetchers	L1D : Stride Prefetcher, degree 4 [12], L2 : AMPM Prefetcher [20]

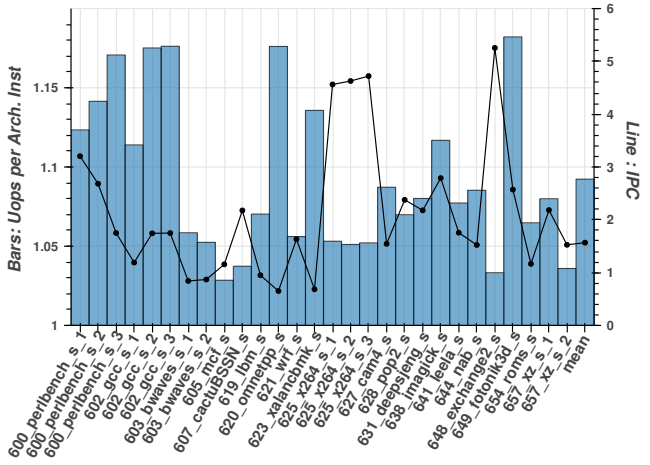


Figure 2: Bar: Retired μ-ops over retired architectural instructions (left y-axis). Line: Baseline IPC (right y-axis). Average is amean for μ-ops per instruction and hmean for IPC. SPEC2k17 CPU *refspeed*.

6 ANALYSIS

Prior to analysis, we point out that *move elimination*, *0/1/9-bit signed integer-idiom elimination* and SpSR focus on architectural instructions. Therefore, it is consistent to reason about the fraction of *architectural* instructions that can be eliminated at Rename through these instances of strength reduction. However, in *gem5*, several simple architectural instructions that would flow as a single μ-op in a high-performance pipeline may in fact generate multiple μ-ops (one notable example is load/stores that use pre/post increment addressing mode). This "expansion ratio" x may limit the performance uplift we can obtain as we reduce one instruction out of $x \times 100M$ (with $x > 1$) rather than out of $100M$. Fig. 2 reports the number of μ-ops per retired architectural instructions as well as the baseline IPC (still reported as architectural instructions per cycle).

6.1 Value Prediction

We run simulations using the VTAGE [34] value predictor depicted in Table 2. In the case where only 0x0 and 0x1 are eligible for VP (Minimal VP, or MVP), writing the prediction to the PRF is done through renaming the destination to the hardwired 0x0/0x1 physical register. In the case where all 9-bit signed integer values are eligible (Targeted VP, or TVP), the physical register names are overloaded to represent the predicted value. In TVP, *9-bit integer idiom elimination* is also enabled as it can use the same inlining mechanism [25] as TVP itself. Finally, when all values are eligible (Generic VP, or GVP), predictions that fit on a 9-bit signed integer (including 0x0 and 0x1) behave as in TVP, however, larger predictions are given a physical register at rename and we assume that the PRF has enough ports to write the prediction. Validation is still performed at execute time, but we assume that the prediction is read from the FIFO used to perform predictor updates, and therefore no additional PRF read is incurred. Predictions are used in the pipeline only when the associated Forward Probabilistic (confidence) Counter (FPC) is saturated [34, 39], and only instructions that produce one (or more) general purpose register are eligible for being value predicted. Fig. 3 depicts the speedup we obtain with the different VP flavors from a baseline with *move elimination* and *0/1-idiom elimination*.

General Trend. While generic VP is able to lift performance significantly (+4.67% geomean), MVP and TVP are still able to provide a modest fraction of the performance uplift using a much smaller predictor, respectively +0.54% (11.5% of the GVP speedup using 14.4% of the storage) and +1.11% (23.7% of the GVP speedup using 25.1% of the storage). The predictor footprint has to be contrasted against past and currently implemented branch predictors and BTBs which may occupy tens to hundreds of KBs [16, 45]. Average coverage ($\frac{\#correct\_used}{\#VP-eligible}$) is respectively 5.3%/12.6%/32.7% for MVP/TVP/GVP, with average accuracy ($\frac{\#correct\_used}{(\#correct\_used+\#incorrect\_used)}$) being above 99.9% in all cases, thanks to the stringent confidence requirements that must be met before a prediction is used.

To determine the sensitivity of MVP/TVP/GVP to predictor storage, we also run each VP flavor using different storage budgets. The geomean speedups are reported in Table 3. The numbers suggest that generic VP is generally a better choice if implementation complexity is not an issue. Indeed, the advantage of MVP and TVP lie in the fact that they require very limited changes to the renamer

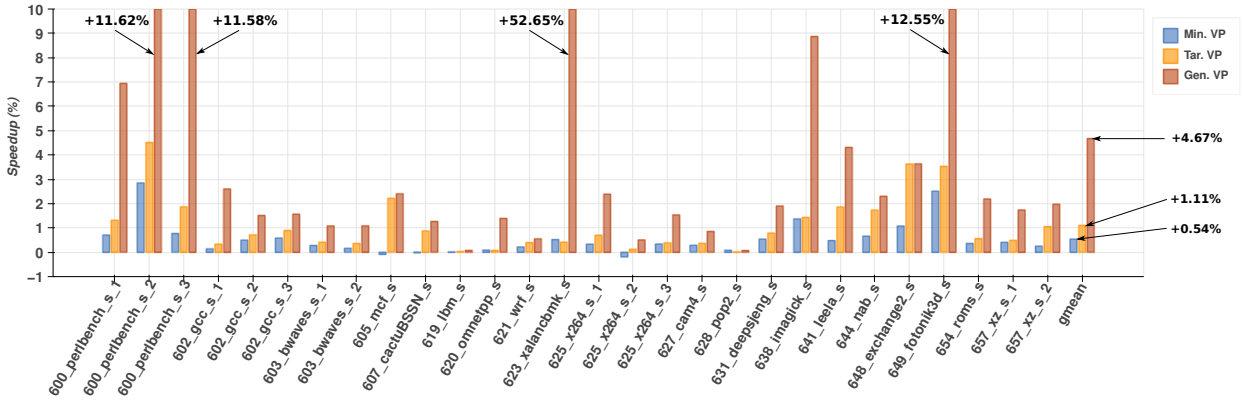


Figure 3: Performance uplift brought by MVP/TVP/GVP on top of a baseline featuring *move* and *0/1-idiom elimination*. SPEC2k17 refspeed.

structure and the physical register file. As a result, we believe that both MVP and TVP achieve the goal of providing noticeable performance uplift in some cases while having minimal impact on the existing microarchitecture, save for adding the value predictor and FIFO update queue. This is especially true for MVP since Table 3 shows that most of the potential (of the VTAGE algorithm) is already attained even with a 4KB predictor.

Table 3: geomean of speedups for each of MVP/TVP/GVP over baseline when ran with various budgets (same number of tables/history bits, only table size is modified.)

Size	MVP	TVP	GVP
0.5 × MVP budget (≈ 4KB)	+0.50%	+0.74%	+2.54%
MVP budget (≈ 8KB)	+0.54%	+0.96%	+2.86%
TVP budget (≈ 14KB)	+0.60%	+1.11%	+3.51%
GVP budget (≈ 55KB)	+0.66%	+1.24%	+4.67%

Outlier. In *xalancbmk*, GVP speeds performance up by 52.65%. The fact that TVP behaves similarly to MVP suggests that *9-bit-signed-idiom elimination* is not responsible for the performance uplift. Moreover, the coverage of the predictors are respectively 7.30%, 55.97% and 72.32% for MVP/TVP/GVP, while accuracy is similar (over 99.99%) in all cases. Speedups are +0.52%/+0.41%/+52.65% respectively. Therefore, the higher VP coverage of TVP as opposed to MVP does not translate to improved performance, and the difference in performance cannot be explained by pipeline flushes due to mispredictions.

We found the limiting factor to be three predictable (with GVP) yet dependent loads within a loop, that are used to retrieve the base address of a structure through multiple indirections. The address is then fed to a fourth load that retrieves a 2B element with a displacement. It should be noted that in this case, Memory Renaming [50] would likely be beneficial as the first predicted load always depends on a previous – silent – store to the same address, and both use the stack pointer and the same offset to compute their address. Those instructions are located in *ValueStore::contains()* but may actually belong to functions inlined into it. Since virtual addresses usually need more than 9 bits, MVP and TVP cannot capture those critical predictions.

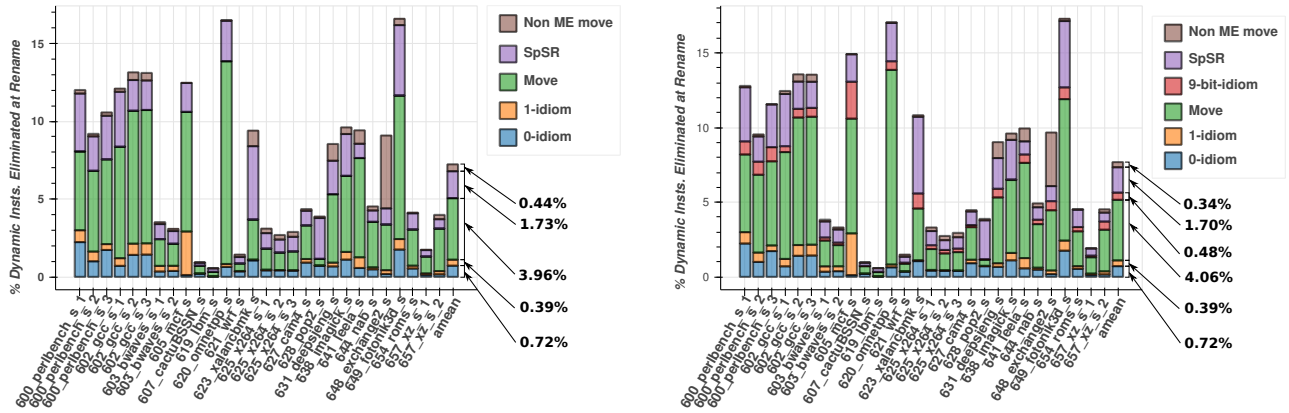
6.2 Speculative Strength Reduction

Fraction of Optimized Instructions. Figure 4 (a) and (b) report the fraction of rename-optimized instructions when SpSR is enabled for MVP and TVP (1.73% and 1.70% respectively).[8] TVP also brings an additional 0.48% eliminated instructions thanks to *9-bit-signed-idiom elimination*.

One can also note a difference in the fraction of eliminated *move* instructions. This stems from the fact that as stated in Section 5, we do not allow a 64-bit register to be moved into a 32-bit register in our experiments. However, when using 9-bit signed physical register identifiers for value predictions and eligible *move immediate* instructions, we are sometimes able to move a 64-bit register into a 32-bit register if the 64-bit register is predicted or 9-bit-signed-idiom eliminated, as we can guarantee that the 32 upper bit are 0 when the value is not sign-extended. Therefore TVP has slightly more potential for ME in our framework, though this would not be the case with a more aggressive ME implementation.

Performance. The additional fraction of instructions (+1.73% for MVP and 1.70 + 0.48% for TVP) unfortunately does not often translate to significantly improved performance, as shown in Figure 5. Even worse, in several cases, performance slightly decreases : *perlbench_2/3* (MVP, TVP), *x264_2/3* (MVP and TVP) and *cam4* (TVP). This is counter-intuitive since SpSR does not introduce new pipeline stalls or flushes. Our experiments suggest that similarly to the behavior in *roms* described in Section 3.4.1, this is caused by interactions with the L1D Stride prefetcher. On a configuration without the Stride prefetcher, SpSR provides +0.00%/+0.00%/+0.57%/+0.00% IPC over TVP for *perlbench_2/3*, *x264_2* and *cam4*, respectively (geomean improvement is +0.11% compared to +0.06% in Fig. 5). The same effect was observed for MVP in *perlbench_2/3* and *x264_3*.

[8]The difference comes from using 250 silencing cycles. This impacts TVP and MVP differently because they do not always predict/mispredict the same instructions.



(a) Fraction of rename-optimized instructions for MVP.

(b) Fraction of rename-optimized instructions for TVP.

Figure 4: Additional fraction of dynamic instructions that can be optimized away at Rename with MVP (DSR+SpSR : 1.73% avg.) and TVP (DSR+SpSR/9-bit idiom elim. : 1.70%/0.48% avg.). SPEC2k17 *refspeed*.

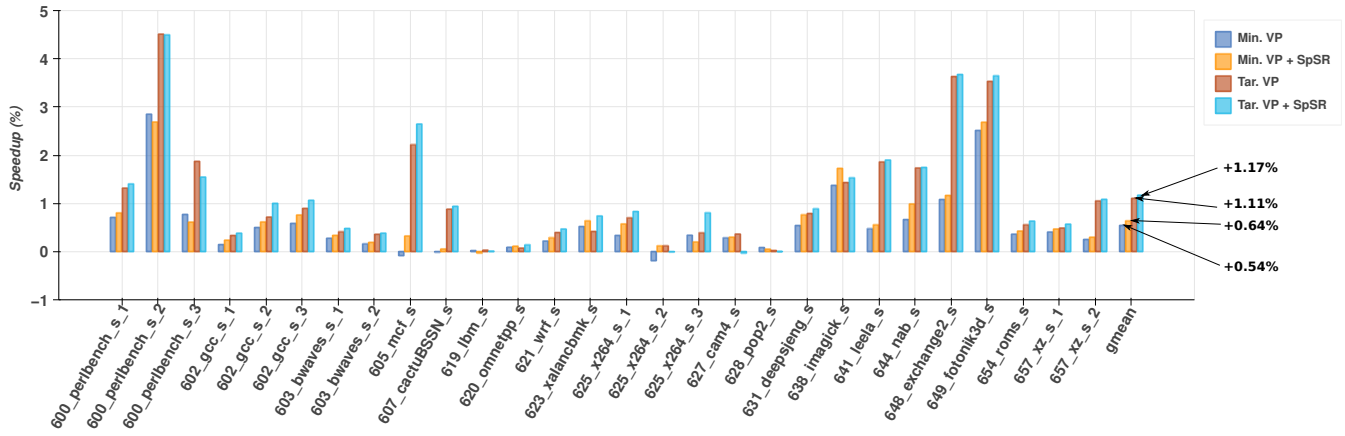


Figure 5: Performance uplift brought by MVP/TVP with and without SpSR on top of a baseline featuring *move* and *0/1-idiom elimination*. GVP is not shown as it exhibits the same trend with SpSR as TVP. SPEC2k17 refspeed.

Impact on Activity. In this paper, we do not provide a full estimate of the power dissipation and energy consumption of the pipeline as the impact of some modifications (e.g., renamer, interactions between VP-tracking FIFO and issue/execute) on power would require a high-end design that is out of our reach. However, we provide proxy metrics such as the overall number of integer physical register reads and writes as well as the number of dispatched and issued instructions in Figure 6. We only report average activity for configurations of interest for readability.

Through using hardwired registers and physical register inlining, MVP and TVP are able to significantly reduce integer PRF activity (-2.41% Rd.[6]/-4.17% Wr. and -9.51% Rd/-11.32% Wr., respectively). IQ activity remains similar since value predicted instructions still need to be dispatched and issued. GVP sees many more integer PRF writes as we assume that predictions that cannot leverage physical register inlining need to be explicitly written to the PRF. Furthermore, we consider that in GVP, training and validation do not require an additional PRF read as the prediction is read from the FIFO used to perform predictor updates and compared against the result computed by the functional unit. Yet, state-of-the-art proposals such as EOLE would incur an additional PRF read since the correct value would need to be read from the PRF at validation/update time, and compared against the prediction sitting in the VP-tracking FIFO [33, 34]. In our experiments, this would account for an additional 22% PRF reads over baseline.

SpSR provides instructions that do not need to dispatch to the IQ at all. As a result, when applying SpSR on top of MVP/TVP, we observe a decrease in IQ activity : -1.64%/-1.53% and -2.41%/-2.04% dispatched/issued instructions, respectively. TVP + SpSR reduce the number of dispatched and issued instructions slightly more than MVP + SpSR due to the presence of *9-bit-signed-idiom*

[6]In MVP, we assume that the zero and one registers are not read by consumers since the physical name in the scheduler entry can serve as the value (Physical Register Number – PRN – 0 is 0x0, PRN 1 is 0x1), much like in full-blown physical register inlining, but without the additional bit.

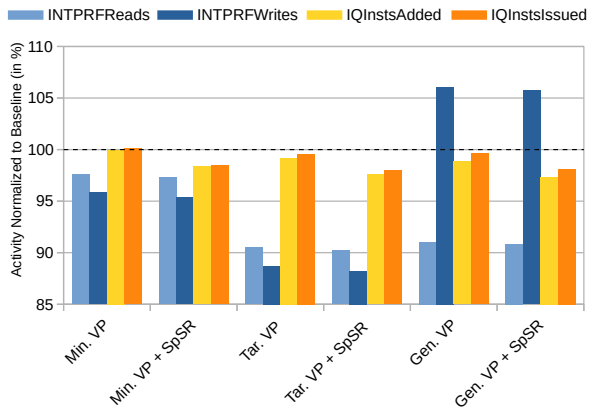


Figure 6: Average INT PRF Read/Writes and IQ dispatched/issued instructions normalized to Baseline. SPEC2k17 refspeed.

elimination. Moreover, GVP + SpSR provides similar reduction, although the mere fact that it is significantly faster (+4.67% IPC) already provides a reduction in dispatched instructions. GVP + SpSR reduces dispatched instructions by 2.66% and issued instructions by 1.90%. The number of issued instructions is higher than in MVP/TVP due to GVP being much better at breaking data dependencies.

7 RELATED WORK

Value Prediction. Early/OoO/Late Execution (EOLE) [33] leverages VP to early execute some instructions in the frontend, using dedicated execution units. Those instructions are not sent to the scheduler thereby achieving a similar effect as SpSR. However SpSR does not require execution units and is able to perform eliminations even when a single input is predicted, whereas EOLE needs all inputs to be available.

Sheikh et al. introduce DLVP, in which only load addresses are predicted, and unused D-Cache access slots are used to prefetch data into a dedicated buffer [46]. Focused Value Prediction attempts to identify candidates that maximize performance gains and train the predictor only for those candidates [4], minimizing the predictor footprint. However, in the context of SpSR, we want to maximize the number of eliminated instructions and therefore, considering all register-producing instructions is preferable.

Finally, there exist many variations of value predictors that could be swapped in to implement MVP/TVP [7, 13, 14, 23, 26, 30, 43, 49, 51, 53]. MVP is especially interesting as it can also leverage branch prediction algorithms such as perceptron [21].

Strength Reduction. Move elimination [22] was initially proposed to execute move instructions at rename. This technique is especially beneficial in the x86 ISA because for most instructions, one of the sources is the destination.

Speculative memory bypassing leverages renaming to transform a def-store-load-use chain into a def-use chain by renaming the destination register of the load to the source register of the store [50]. Validating speculation implies checking that addresses match but also that the load value is consistent with what the memory model allows.

REName Optimizer (RENO) [37] and Continuous Optimization [10] suggest to monitor instruction sequences to remove redundant computations through the renamer. For instance, two loads from the same address but to different registers, in which case the second load can map its destination register to the destination register of the first load. Different optimizations may be performed including constant propagation (CP), redundant load elimination (RLE), and store forwarding. Some of those optimizations require additional hardware in the backend (CP) and/or actual execution of the "eliminated" instruction (RLE).

SpSR is orthogonal to those techniques as it is inherently dynamic, that is, inspecting instruction bits is not sufficient to perform SpSR. However, SpSR does build on the availability of move elimination and zero/one-idiom elimination.

8 CONCLUSION

In this paper, we present Minimal and Targeted Value Prediction (MVP and TVP), two implementations of Value Prediction that leverage register renaming to provide predictions to dependent instructions at lowered hardware complexity. While those implementations still require a value predictor to provide values, the fact that only a very limited number of values are candidate for prediction greatly limits the footprint of the value predictor. MVP and TVP provide 0.54% and 1.11% speedup while a generic implementation provides 4.67% speedup, on average, on SPEC2k17 CPU speed benchmarks. This demonstrates that a limited value prediction infrastructure can still provide modest speedup even if few distinct values can be predicted.

We further build on MVP and TVP with Speculative Strength Reduction, a novel rename-time optimization that makes specific instructions disappear at Rename if their operands are predicted to be 0x0 or 0x1. We find that 1.73% of the dynamic instructions can disappear from Rename in MVP. When adding *9-bit signed idiom elimination* through TVP, this fraction increases to 2.18%. While SpSR does not improve performance significantly in our experiments, it decreases the number of instructions dispatched to and issued from the IQ. Therefore, SpSR appears as a natural add-on to MVP/TVP to further reduce pressure on the backend structures such as the PRF and the IQ.

REFERENCES

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*. IEEE, 423–434.
[2] Juan L Aragón, José González, and Antonio González. 2003. Power-aware control speculation through selective throttling. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture*. IEEE, 103–112.
[3] Arm Ltd. [n. d.]. Armv8 Reference Manual. https://documentation-service.arm.com/static/5f20515cbb903e39c84dc459?token=.
[4] S. Bandishte, J. Gaur, Z. Sperber, L. Rappoport, A. Yoaz, and S. Subramoney. 2020. Focused Value Prediction. In *Proc. of the ACM/IEEE Intl. Symp. on Computer Architecture*. 79–91. https://doi.org/10.1109/ISCA45697.2020.00018
[5] Steven Battle, Andrew D Hilton, Mark Hempstead, and Amir Roth. 2012. Flexible register management using reference counting. In *Proc. of the IEEE Intl. Symp. on High-Performance Comp Architecture*. IEEE, 1–12.
[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC Intl. Conf. on Performance Engineering*. 41–42.

[7] Martin Burtscher and Benjamin G Zorn. 1999. Exploring last n value prediction. In *Proc. of the IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*. IEEE, 66–76.

[8] Brad Calder, Glenn Reinman, and Dean M Tullsen. 1999. Selective value prediction. In *Proc. of the IEEE/ACM Intl. Symp. on Computer Architecture*. 64–74.

[9] G Chrysos and J Emer. 1998. Memory Dependence Prediction using Store Sets. In *Proc. of the ACM/IEEE Intl. Symp. on Computer Architecture*. IEEE, 0142–0142.

[10] Brian Fahs, Todd Rafacz, Sanjay J Patel, and Steven S Lumetta. 2005. Continuous optimization. In *Proc. of the Intl. Symp. on Computer Architecture*. IEEE, 86–97.

[11] Andrei Frumusanu. [n. d.]. Apple's Humongous CPU Microarchitecture. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

[12] John WC Fu, Janak H Patel, and Bob L Janssens. 1992. Stride directed prefetching in scalar processors. In *Pro. of the IEEE/ACM Intl. Symp. on Microarchitecture*. 102–110.

[13] Freddy Gabbay and Avi Mendelson. 1998. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems* 16, 3 (1998), 234–270.

[14] Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere. 2001. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proc. of the IEEE Intl. Symp. on High-Performance Computer Architecture*. IEEE, 207–216.

[15] Antonio Gonzalez, Jose Gonzalez, and Mateo Valero. 1998. Virtual-physical registers. In *Proc. of the IEEE Intl. Symp. on High-Performance Computer Architecture*. IEEE, 175–184.

[16] Brian Grayson, Jeff Rupley, Gerald Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. 2020. Evolution of the samsung exynos CPU microarchitecture. In *Proc. of the ACM/IEEE Intl. Symp. on Computer Architecture*. IEEE, 40–51.

[17] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[18] Intel Corporation. [n. d.]. Intel 64 and IA-32 Arch. Optim. Reference Manual. software.intel.com/content/dam/develop/public/us/en/ documents/64-ia-32-architectures-optimization-manual.pdf.

[19] Intel Corporation. [n. d.]. Intel 64 and IA-32 Arch. Soft. Dev. Manuals. software.intel.com/content/www/us/en/develop/download/ intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[20] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd international conference on Supercomputing*. 499–500.

[21] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *Proc. of the IEEE Intl. Symp. on High Performance Computer Architecture*. IEEE, 197–206.

[22] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. 1998. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proc. of the Intl. Symp. on Microarchitecture*. IEEE, 216–225.

[23] Kleovoulos Kalaitzidis and André Seznec. 2020. Leveraging Value Equality Prediction for Value Speculation. *ACM Transactions on Architecture and Code Optimization* 18, 1 (2020), 1–20.

[24] Ilhyun Kim and Mikko H Lipasti. 2004. Understanding scheduling replay schemes. In *Proc. of the IEEE Intl. Symp. on High Performance Computer Architecture*. IEEE, 198–209.

[25] Mikko H Lipasti, Brian R Mestan, and Erika Gunadi. 2004. Physical register inlining. In *Proc. of the IEEE/ACM Intl. Symp. on Computer Architecture*. IEEE, 325–335.

[26] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *Proc. of the ACM Tntl Conf. on Architectural Support for Programming Languages and Operating Systems*. 138–147.

[27] Gabriel Loh. 2003. Width prediction for reducing value predictor size and power. In *First Value Prediction Workshop, at IEEE/ACM ISCA*. Citeseer.

[28] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:cs.AR/2007.03152

[29] Milo MK Martin, Daniel J Sorin, Harold W Cain, Mark D Hill, and Mikko H Lipasti. 2001. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proc. of the ACM/IEEE Intl. Symp. on Microarchitecture. MICRO-34*. IEEE, 328–337.

[30] Tarun Nakra, Rajiv Gupta, and Mary Lou Soffa. 1999. Global context-based value prediction. In *Proc. ot the IEEE Intl. Symp. on High-Performance Computer Architecture*. IEEE, 4–12.

[31] Subbarao Palacharla, Norman P Jouppi, and James E Smith. 1997. Complexity-effective superscalar processors. In *Proc. of the IEEE/ACM Intl. Symp. on Computer architecture*. 206–218.

[32] Arthur Perais. 2021. A Case for Speculative Strength Reduction. *IEEE Computer Architecture Letters* 20, 1 (2021), 22–25.

[33] Arthur Perais and André Seznec. 2014. EOLE: Paving the way for an effective implementation of value prediction. In *Proc. of the ACM/IEEE Intl. Symp. on Computer Architecture*. IEEE, 481–492.

[34] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *Proc. of the IEEE Intl. Symp. on High Performance Computer Architecture*. IEEE, 428–439.

[35] Arthur Perais and André Seznec. 2015. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *Proc. of the IEEE Intl. Symp. on High Performance Computer Architecture*. IEEE, 13–25.

[36] Arthur Perais and André Seznec. 2016. Cost effective physical register sharing. In *Proc. of the IEEE Intl. Symp. on High Performance Computer Architecture (HPCA)*. IEEE, 694–706.

[37] Vlad Petric, Tingting Sha, and Amir Roth. 2005. Reno: a rename-based instruction optimizer. In *Proc. of the Intl. Symp. on Computer Architecture*. IEEE, 98–109.

[38] G Reinman, T Anstin, and B Calder. 1999. A scalable front-end architecture for fast instruction delivery. In *Proc. of the ACM/IEEE Intl. Symp. on Computer Architecture*. IEEE, 234–245.

[39] Nicholas Riley and Craig Zilles. 2006. Probabilistic counter updates for predictor hysteresis and stratification. In *Proc. of the IEEE Intl. Symp. on High-Performance Computer Architecture, 2006*. IEEE, 110–120.

[40] RISC-V Foundation. [n. d.]. RISC-V Unprivileged Spec. https://github.com/riscv/riscv-isa-manual/releases/latest.

[41] Elham Safi, Andreas Moshovos, and Andreas Veneris. 2010. Two-stage, pipelined register renaming. *IEEE Transactions on Very Large Scale Integration systems* 19, 10 (2010), 1926–1931.

[42] Toshinori Sato and Itsujiro Arita. 2000. Table size reduction for data value predictors by exploiting narrow width values. In *Proc. of the Intl. Conf. on Supercomputing*. 196–205.

[43] Yiannakis Sazeides and James E Smith. 1997. The predictability of data values. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*. IEEE, 248–258.

[44] André Seznec. 2011. A new case for the tage branch predictor. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*. 117–127.

[45] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. 2002. Design tradeoffs for the Alpha EV8 conditional branch predictor. *Proc. of the ACM/IEEE Intl. Symp. on Computer Architecture* 30, 2 (2002), 295–306.

[46] Rami Sheikh, Harold W Cain, and Raguram Damodaran. 2017. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*. 423–435.

[47] Rami Sheikh and Derek Hower. 2019. Efficient load value prediction using multiple predictors and filters. In *Proc. of the IEEE Intl. Symp. on High Performance Computer Architecture*. IEEE, 454–465.

[48] Niranjan Soundararajan, Saurabh Gupta, Ragavendra Natarajan, Jared Stark, Rahul Pal, Franck Sala, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney. 2019. Towards the adoption of local branch predictors in modern out-of-order superscalar processors. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*. 519–530.

[49] Renju Thomas and Manoj Franklin. 2001. Using dataflow based context for accurate value prediction. In *Proc. of the IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*. IEEE, 107–117.

[50] Gary S Tyson and Todd M Austin. 1997. Improving the accuracy and performance of memory communication through renaming. In *Proc. of the Intl. Symp. on Microarchitecture*. IEEE, 218–227.

[51] Kai Wang and Manoj Franklin. 1997. Highly accurate data value prediction using hybrid predictors. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*. IEEE, 281–290.

[52] Jun Yang and Rajiv Gupta. 2002. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems (TECS)* 1, 1 (2002), 79–105.

[53] Huiyang Zhou, Jill Flanagan, and Thomas M Conte. 2003. Detecting global stride locality in value streams. In *Proc. of the IEEE/ACM Intl. Symp. on Computer architecture*. 324–335.