A Case for Speculative Strength Reduction

Arthur Perais

Abstract—Most high performance general purpose processors leverage register renaming to implement optimizations such as move elimination or zero-idiom elimination. Those optimizations can be seen as forms of strength reduction whereby a faster but semantically equivalent operation is substituted to a slower operation. In this letter, we argue that other reductions can be performed dynamically if input values of instructions are known in time, i.e., prior to renaming. We study the potential for leveraging Value Prediction to achieve that goal and show that in SPEC2k17, an average of 3.3 percent (up to 6.8 percent) of the dynamic instructions could dynamically be strength reduced. Our experiments suggest that a state-of-the-art value predictor allows to capture 59.7 percent of that potential on average (up to 99.6 percent).

Index Terms—General purpose microprocessors, microarchitecture, speculation

1 INTRODUCTION

Strength reduction is a term notably used in compiler design that consists in picking the least costly operation that is able to implement the semantics specified by the programmer. A famous example is to replace a multiplication (resp. division) by $2^n$ by a left (resp. right) shift of $n$.

Strength reduction is therefore commonly thought of as a compile-time optimization, hence we could refer to it as static strength reduction. However, modern general purpose processors already employ several forms of dynamic strength reduction, such as move elimination [1] and zero-idiom elimination [2]. The former consists in eliding a “move register to register” instruction by renaming the architectural destination register of the instruction to its source physical register. The latter will rename the destination register of an instruction that statically produces 0x0 (e.g. xor rax, rax) to a physical register that is hardwired to 0x0. Generally speaking, any instruction whose result can be known statically can be eliminated using renaming to implement optimizations such as move elimination or zero-idiom elimination. Those optimizations can be seen as forms of strength reduction whereby a faster but semantically equivalent operation is substituted to a slower operation. In this letter, we argue that other reductions can be performed dynamically if input values of instructions are known in time, i.e., prior to renaming.

In this letter, we observe that many instructions not initially thought to be strength reducible become so during execution. For instance, and overlooking side effects such as status flag updates in X86_64 ISA. Other values such as 0x0 or 0x1 are produced significantly more often than most other values, as depicted in Fig. 1 for SPEC2k17 (int/fpspeed) ran under Intel Pin [4]. Nearly 6.0 percent of the dynamically generated general purpose register values are either 0x0 or 0x1.

Incidentally, 0x0 and 0x1 are the values that will generally enable instructions to be dynamically strength reduced. Some – but not all – possible reductions are shown in Table 1 assuming the x86_64 ISA. Other values such as – 1 (null) may unlock more potential (notably for bitwise logical instructions), but this letter focuses on 0x0 and 0x1 as those values can be trivially encoded in a value predictor and already benefit from special treatment (either architecturally [5], [6] or microarchitecturally [2]) in the pipeline. Note that the proposed reductions ignore any side effects the reduced instructions may have, such as x86_64 status flags modifications (which are particularly relevant for test). We address side effects in Section 3.3.

3 BUILDING AN SSR MICROARCHITECTURE

The SSR microarchitecture is built on a limited value prediction infrastructure that focuses on predicting a few key values. Therefore, we first need to design a robust VP flow that will provide and validate predictions as well as train the predictor.

3.1 A Limited Value Prediction Infrastructure

3.1.1 Predicting Values

SSR requires predicted source operands to determine that an instruction can be speculatively strength reduced in the frontend. Any state-of-the-art predictor can be used for that purpose (e.g., VTAGE [7]), although we note that since we only consider 0x0 and 0x1 for prediction, its footprint will be much smaller than in a VP infrastructure where any value can be predicted [8]. The predictor should attempt to predict the destination of all register-producing instructions. In this fashion, VP enables SSR but also permits consumers of the predicted result that cannot be SSR’d to execute earlier by virtue of breaking Read-after-Write dependencies.

In this design, the frontend keeps track of which architectural names are currently predicted to be 0x0 or 0x1, and is responsible for replacing SSR candidates by their strength reduced counterpart when relevant. Instructions that are SSR’d will be eliminated during renaming through the existing move/zero-idiom elimination machinery.

Since a prediction can only be 0x0 or 0x1, and since there generally already exists a hardwired zero register, this VP implementation need not write predictions to the Physical Register File (PRF).

Unfortunately, hardware generally does not know the source operands values by the time the decision to strength reduce the instruction has to be made (Decode or Rename). Consequently, we propose speculative strength reduction (SSR), in which source operands of instructions are predicted to enable opportunistic dynamic strength reduction.

2 MOTIVATION

Fully featured Value Prediction (VP) [3] is expensive in terms of silicon, simply due to the fact that the value predictor may have to store 64-bit worth of prediction for each candidate instruction. Moreover, the value distribution in programs is not uniform. Indeed, combined, the values 0x0 and 0x1 are produced significantly more often than most other values, as depicted in Fig. 1 for SPEC2k17 (int/fpspeed) ran under Intel Pin [4]. Nearly 6.0 percent of the dynamically generated general purpose register values are either 0x0 or 0x1.

Note that eliminated, in this context, means that instructions do not need to be sent to the backend for execution, hence, they do not occupy a slot in the scheduler and they do not require an execution unit. However, those instructions may still consume tracking resources to enforce sequential semantics and enable microarchitectural state to be rolled-back on a pipeline flush.

In this letter, we observe that many instructions not initially thought to be strength reducible become so during execution. For instance, and overlooking side effects such as status flag updates in X86_64 ISA. Other values such as 0x0 or 0x1 are produced significantly more often than most other values, as depicted in Fig. 1 for SPEC2k17 (int/fpspeed) ran under Intel Pin [4]. Nearly 6.0 percent of the dynamically generated general purpose register values are either 0x0 or 0x1.

Note that eliminated, in this context, means that instructions do not need to be sent to the backend for execution, hence, they do not occupy a slot in the scheduler and they do not require an execution unit. However, those instructions may still consume tracking resources to enforce sequential semantics and enable microarchitectural state to be rolled-back on a pipeline flush.

In this letter, we observe that many instructions not initially thought to be strength reducible become so during execution. For instance, and overlooking side effects such as status flag updates in X86_64 ISA. Other values such as 0x0 or 0x1 are produced significantly more often than most other values, as depicted in Fig. 1 for SPEC2k17 (int/fpspeed) ran under Intel Pin [4]. Nearly 6.0 percent of the dynamically generated general purpose register values are either 0x0 or 0x1.

Incidentally, 0x0 and 0x1 are the values that will generally enable instructions to be dynamically strength reduced. Some – but not all – possible reductions are shown in Table 1 assuming the x86_64 ISA. Other values such as – 1 (null) may unlock more potential (notably for bitwise logical instructions), but this letter focuses on 0x0 and 0x1 as those values can be trivially encoded in a value predictor and already benefit from special treatment (either architecturally [5], [6] or microarchitecturally [2]) in the pipeline. Note that the proposed reductions ignore any side effects the reduced instructions may have, such as x86_64 status flags modifications (which are particularly relevant for test). We address side effects in Section 3.3.

3 BUILDING AN SSR MICROARCHITECTURE

The SSR microarchitecture is built on a limited value prediction infrastructure that focuses on predicting a few key values. Therefore, we first need to design a robust VP flow that will provide and validate predictions as well as train the predictor.

3.1 A Limited Value Prediction Infrastructure

3.1.1 Predicting Values

SSR requires predicted source operands to determine that an instruction can be speculatively strength reduced in the frontend. Any state-of-the-art predictor can be used for that purpose (e.g., VTAGE [7]), although we note that since we only consider 0x0 and 0x1 for prediction, its footprint will be much smaller than in a VP infrastructure where any value can be predicted [8]. The predictor should attempt to predict the destination of all register-producing instructions. In this fashion, VP enables SSR but also permits consumers of the predicted result that cannot be SSR’d to execute earlier by virtue of breaking Read-after-Write dependencies.

In this design, thefrontend keeps track of which architectural names are currently predicted to be 0x0 or 0x1, and is responsible for replacing SSR candidates by their strength reduced counterpart when relevant. Instructions that are SSR’d will be eliminated during renaming through the existing move/zero-idiom elimination machinery.

Since a prediction can only be 0x0 or 0x1, and since there generally already exists a hardwired zero register, this VP implementation need not write predictions to the Physical Register File (PRF).
to allow dependents to use predicted values. Rather, the destination register of predicted instructions is renamed to the hardwired 0x0 physical register, and a hardwired physical 0x1 register is implemented. This has significant implications on PRF complexity as additional write ports – which are needed in some state-of-the-art VP proposals [9], [10] – are costly in term of PRF area and energy [11].

### 3.1.2 Validating Predictions

Predicted values need to be validated for correctness. There are multiple possible designs for general VP, but in SSR, a prediction can only be 0x0 or 0x1. Therefore, it may be reasonable for predicted instructions to carry the predicted value while they flow down the pipeline to permit in-place validation directly at the functional unit. Indeed, the mere fact that the destination register number is the hardwired 0x0 or 0x1 register can be used to indicate what value the result of the execution unit should be compared against.

This contrasts with state-of-the-art where validation is performed at commit by reading the actual value from the PRF and comparing it to the prediction that was stored in a FIFO structure [7]. This impacts complexity as either additional PRF read ports are required on the PRF, or read port arbitration is required between issuing instructions reading their operands from the PRF and predicted instructions reading the actual result from the PRF to compare against the predicted value. The former has implications on PRF area and energy per access [11], whereas the latter may introduce delay if an arbiter is not implemented in the baseline (i.e., there are enough read ports for all issuing instructions in the baseline to simplify scheduling). Conversely, limited VP does not touch the PRF at all. Limited VP also does not require the FIFO of predictions as the predicted value is carried with the instruction.

In addition, validating at commit can significantly increase the cost of a misprediction. In our limited VP infrastructure, mispredictions can be discovered at execution. This makes the cost of a value misprediction comparable to the cost of a branch misprediction.

Note that since SSR’s instructions are always – directly or indirectly – dependent on a predicted value, validating the prediction is sufficient to validate the reduction. However, this requires that a value misprediction trigger a pipeline flush and not a replay. Indeed, SSR’s instructions cannot be replayed since they were never sent to the scheduler in the first place.

### 3.1.3 Training the Predictor

Training in itself can be done at execution or at retirement and predictor management may be arbitrarily complex, knowing that we would like to allocate predictor entries only to instructions that are known to at least occasionally produce 0x0 and 0x1. There have been several proposals to filter the instructions that are allowed to enter the predictor tables [10], however, filtering and generally tailoring the value predictor to SSR is left for future work. In this letter, we consider a large state-of-the-art predictor to highlight the potential of SSR.

### 3.2 Potential Interactions With Conditional Branch Prediction

While conditional branches are predicted and therefore do not stall instruction retrieval, said branches still have to be executed in the backend to validate the predicted direction. With SSR, instructions that are directly feeding branches may be value predicted or even eliminated. In that context, the branch itself may be executed in the frontend using the generated status flags (x86, ARM) or register value (RISC-V) of a predicted/eliminated instruction, because if the value prediction is ultimately correct, the early branch resolution is also correct.

While this can further increase the number of eliminated instructions, this possibility also opens the door to mechanisms that improve the timeliness of branch – or even value – corrections, as was initially suggested by Aragon et al. [12], such as overriding a branch prediction with a value prediction, or vice versa.

### 3.3 Enforcing Program Semantics in an SSR Microarchitecture

In general, an instruction can be eliminated only if it does not have any side effects, or if those side-effects can be applied as part of the elimination. For instance, in x86_64, mov reg, reg instructions can be move eliminated because they have no side effects, and notably, they do not modify the status flags register [13]. Similarly, in x86_64, xor rax, rax can be zero-idiom eliminated because although the xor instruction modifies the status flags, it does so in a way that is known at elimination time because the output of the instruction is known to be 0x0 [13].

Conversely, some of the x86_64 examples shown in Table 1 do have side effects that cannot generally be handled at elimination time. For instance, add rax, rbx where rbx is predicted to be 0x0 cannot be replaced by a nop because the status flags have to be updated based on the output of the instruction [13].

Those limitations stem from the semantics defined at the architectural level, and are therefore ISA-dependent. For instance, in ARMv8, dedicated arithmetic and logic instructions are used to modify the status flags (e.g., add; and subs [5]), and the instruction add x1, x1, x2 (which is similar to add rax, rbx in x86_64) can be strengthened reduced to a nop if x2 is predicted to be 0x0. As a result, the potential for a performant SSR implementation is tied to the ISA.

**TABLE 1 Examples of Possible Strength Reductions for Some x86_64 Instructions**

| Op     | src1 | src2 | mov src1,src2 | mov src1,0x0 | mov src1,0x1 | mov src1,0x0 | mov src1,0x1 |
|--------|------|------|---------------|--------------|--------------|--------------|--------------|
| imul   | nop  | mov src1,src2 | mov src1,0x0 | mov src1,0x0 | mov src1,0x1 | mov src1,0x0 | mov src1,0x1 |
| add    | mov src1,src2 | – | nop | – | nop | – | – |
| sub    | – | nop | – | nop | – | nop | – |
| shr    | nop | – | nop | – | nop | – | – |
| test   | nop | – | nop | – | nop | – | – |
| xor    | mov src1,src2 | – | nop | – | nop | – | – |
| or     | mov src1,src2 | – | nop | – | nop | – | – |

Side effects such as x86 flags updates are ignored in this Table.
others (e.g., instructions that reduce to mov reg, reg in Table 1) can only be partially eliminated.

Partial elimination will not eliminate the instruction from the pipeline, meaning that the instruction will consume a scheduler entry and be given a new physical destination flag register. However, the destination result register of the instruction will behave as if the instruction had been eliminated, meaning that consumers of the result register may still execute earlier thanks to the speculative strength reduction.

ARMv8, which is mainly used in embedded and mobile devices but is poised to take over some desktop and datacenter offerings, is generally advantaged due to the fact that most examples shown in Table 1 do not have side effects. Indeed, dedicated arithmetic and logical instructions must be used to modify the status flags [5].

RISC-V has recently been getting traction and is also generally advantaged in the context of SSR since there are no status flags to be updated as side effects of arithmetic and logical instructions [6]. In particular, conditional branches will directly read register values and perform the compare operation as part of the branch instruction.

4 SUMMARY OF SSR POTENTIAL BENEFITS

Assuming SSR’d instructions are fully eliminated most of the time (i.e., ARMv8 or RISC-V case), SSR will help reduce the backend energy spent executing those instructions. SSR also has potential to reduce execution time if the SSR’d instructions are on the critical path.

Moreover, SSR has potential to reduce stalls caused by oversubscribed resources. First, at the PRF level as SSR’d instructions will neither read/write nor be allocated a regular physical register, reducing PRF activity and pressure. Second, at the scheduler level as SSR’d instructions do not occupy an entry, which can allow younger instructions to be dispatched to the scheduler and improve ILP/MLP. Lastly, at the functional unit (FU) level as SSR’d instructions do not use one, which has potential to let a younger instruction use the FU earlier than it would have if a particular FU type is oversubscribed.

5 EXPERIMENTAL RESULTS

Fig. 2 reports the percentage of dynamic instructions that can be strength reduced according to Table 1. The Figure also categorizes conditional branches whose inputs become available as part of value prediction or strength reduction as strength reducible, but ignores zero- and one-idioms (e.g., xor rax, rax) as well as regular mov instructions as those can already be eliminated in the baseline. The Figure also depicts the ratio of candidates that can be captured by using a state-of-the-art 13KB VTAGE predictor as well as candidates that can be non-speculatively strength reduced by tracking 0x0 and 0x1 immediates through mov reg, imm and move reg, reg instructions. The numbers are reported for SPEC2k17 (int/fpspeed, train inputs) runs using Intel Pin [4].

Fig. 3 further depicts how reduction types are distributed within the overall instructions that are found to be strength reducible at runtime. Clearly, the ability to value predict one or both inputs of test instructions is quite beneficial as it allows to eliminate branches relying on the outcome as well as the test instructions themselves. Nonetheless, several workloads show specific modes, e.g., bwaves in which add instructions with the second input being 0x1 make up for 60 percent of the DSR candidates, and imagick where sub instructions with the second input being 0x0 represent almost all the DSR candidates (and interestingly, 49.9 percent of the dynamic sub instructions).

Overall, the number of instructions that can be SSR’d according to the rules of Table 1 is modest, but not negligible: 3.3 percent on average, with a maximum of 6.8 percent in 625.x264. Those numbers are however upper bounds, since only a fraction of those cases will be sufficiently predictable, as shown in Fig. 2. Indeed, on average, the VTAGE value predictor we consider (13KB storage) allows 59.7 percent of the DSR potential to be captured, which amounts to 1.9 percent of the dynamic instructions. We also found that a simple 8KB-entry gshare-like value predictor (6KB storage) captures 38.6 percent of the DSR potential (1.1 percent of the dynamic instructions) on average, although average value MPKI was higher (0.02 instead of 0.002 for VTAGE). We do not plot results for this predictor due to space constraints, but the results suggest that even a simple value predictor can achieve a significant fraction of the DSR potential although tuning may be required to minimize value mispredictions.

6 RELATED WORK

Move elimination was initially proposed [1] 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, therefore values have to be saved in other registers more often than in other ISAs.

Speculative memory bypassing leverages renaming to transform a def-store-load-use chain into a def-use chain by renaming 2. We refer the reader to [14] for details on the predictor used in the experiments. Average MPKI of the predictor is 0.002 with a maximum of 0.010 in 631. deepsjeng. The average coverage (correct predictions that would be used by the pipeline) (confident) over dynamic instructions that actually produce 0x0 or 0x1) is 71.4 percent (98.4 percent in 644.nab).
the destination register of the load to the source register of the store [15]. Validating speculation implies checking that addresses match but also that the load value is consistent with what the memory model allows.

REName Optimizer (RENO) [16] and Continuous Optimization [17] 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).

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

Lastly, EOLE [9] leverages value prediction 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 SSR. However SSR does not require execution units and is able to perform eliminations even when a single input is predicted, whereas EOLE can only early execute an instruction if all inputs are available. Moreover, SSR does not suffer from the usual shortcomings of value prediction infrastructures (notably PRF requirements for using and validating predictions).

7 Conclusion

Using simple rules and ignoring ISA-specific aspects, this letter shows that a non negligible percentage (3.3 percent on average) of instructions can be dynamically strength reduced (59.7 percent of the dynamically strength reducible instructions with negligible MPKI. SSR only requires a limited value prediction framework that has much lower storage and complexity overhead than a fully featured VP infrastructure.

Overall, this letter proposes a technique with the potential to further increase single-thread performance, although performance improvement and specifically the validity of the power/performance tradeoff remain to be demonstrated and are left for future work.

References

[1] 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 Proc. IEEE Int. Symp. Microarchit., 1998.

[2] Intel Corporation, “Intel 64 and ia-32 arch. optim. reference manual,” 2020. [Online]. Available: software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf

[3] M. H. Lipasti and F. P. Shen, “Exceeding the dataflow limit via value prediction,” in Proc. IEEE Int. Symp. Microarchit., 1996, pp. 226–237.

[4] Intel Corporation, “Pin: A dynamic binary instrumentation tool,” 2020. [Online]. Available: software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[5] Arm Ltd., “Armv8 reference manual,” 2020. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ff/

[6] RISC-V Foundation, “RISC-V unprivileged spec,” 2020. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/latest

[7] A. Perais and A. Seznec, “Practical data value speculation for future high-end processors,” in Proc. IEEE Int. Symp. High Perform. Comput. Archit., 2014, pp. 428–439.

[8] T. Sato and I. Arita, “Low-cost value predictors using frequent value locality,” in Proc. Int. Symp. High Perform. Comput., 2002, pp. 106–119.

[9] A. Perais and A. Seznec, “Eole: Paving the way for an effective implementation of value prediction,” in Proc. Int. Symp. Comput. Archit., 2014, pp. 481–492.

[10] S. Bandishke, J. Gaur, Z. Sperber, L. Rappoport, A. Yoaz, and S. Subramoney, “Focused value prediction,” in Proc. Int. Symp. Comput. Archit., 2020, pp. 79–91.

[11] V. Zyuban and P. Kogge, “The energy complexity of register files,” in Proc. Int. Conf. Low Power Electronics Des., 1998, pp. 305–310.

[12] J. L. Aragón, J. González, J. M. García, and A. González, “Selective branch prediction reversal by correlating with data values and control flow,” in Proc. Int. Conf. Comput. Des., 2001, pp. 228–233.

[13] Intel Corporation, “Intel 64 and IA-32 Arch. Soft. Dev. Manuals,” 2020. [Online]. Available: software.intel.com/content/www/us/en/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html

[14] A. Seznec and K. Kalaitzidis, “Exploring value prediction limits,” in Championshp Value Prediction (Online contest), 2020. [Online]. Available: https://www.microarch.org/cvp1/papers/Seznec_Unlimited_2020.pdf

[15] G. S. Tyson and T. M. Austin, “Improving the accuracy and performance of memory communication through renaming,” in Proc. IEEE Int. Symp. Microarchit., 1997, pp. 218–227.

[16] V. Petric, T. Sha, and A. Roth, “Reno: A rename-based instruction optimizer,” in Proc. IEEE Int. Symp. Comput. Archit., 2005, pp. 98–109.

[17] B. Falb, T. Rafacz, S. J. Patel, and S. S. Lumetta, “Continuous optimization,” in Proc. IEEE Int. Symp. Comput. Archit., 2005, pp. 86–97.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.