Register Sharing for Equality Prediction

Arthur Perais, Fernando A. Endo, André Seznec

Register Sharing for Equality Prediction

Arthur Perais
INRIA/IRISA
arthur.perais@inria.fr

Fernando A. Endo
INRIA/IRISA
fernando.endo@inria.fr

André Seznec
INRIA/IRISA
andre.seznec@inria.fr

***Abstract*—Recently, Value Prediction (VP) has been gaining renewed traction in the research community. VP speculates on the result of instructions to increase Instruction Level Parallelism (ILP). In most embodiments, VP requires large tables to track predictions for many static instructions.**

However, in many cases, it is possible to detect that the result of an instruction is produced by an older inflight instruction, but not to predict the result itself. Consequently it is possible to rely on predicting register equality and handle speculation through the renamer.

To do so, we propose to use Distance Prediction, a technique that was previously used to perform Speculative Memory Bypassing (short-circuiting def-store-load-use chains). Distance Prediction attempts to determine how many instructions separate the instruction of interest and the most recent older instruction that produced the same result. With this information, the physical register identifier of the older instruction can be retrieved from the ROB and provided to the renamer.

In this paper, we first quantify the performance gains brought by renaming-based register equality prediction and regular VP on SPEC benchmarks. Second, we study the overlap between the two different schemes and show that those mechanisms often capture different behavior.

I. Introduction

High-performance processors are expected to provide meaningful performance improvements across generations. To that extent, one trend has been to pack more cores on a single die, increasing computation throughput. Unfortunately, many programs do not benefit from this additional throughput because they are intrinsically sequential. Therefore, there is a renewed demand for increasing sequential performance.

Increasing the superscalar width as well as the instruction window size is notoriously hard because the power and area of related structures increase exponentially. As a result, sequential performance improvements have been obtained by less "direct" optimizations, such as better branch prediction, better prefetching, move elimination, zero-idiom elimination, micro- and macro-op fusion, and many other – likely undisclosed – features [1], [2]. In most cases, these optimizations do not require widening the superscalar width or instruction window.

Another mechanism aiming at increasing sequential performance is Value Prediction (VP) [3], [4], [5], where the result of instructions is predicted to break true data dependencies, increasing Instruction-Level Parallelism (ILP). VP was proposed in the late 90's and never implemented due to complexity concerns. However, recent contributions suggest that said complexity can be mostly eliminated [6], [7], [8]. Nonetheless, a – large – structure is required to provide predictions.

Indeed, while Perais and Seznec obtained good performance with a TAGE-like value predictor (D-VTAGE) requiring around 16-32KB of storage [6], such a large structure is not trivial to implement, and it is likely to have a noticeable power draw if it must handle multiple accesses per cycle. A more storage-efficient alternative VP scheme was proposed by Tullsen and Seng [9]. In this embodiment, predictions are located in physical registers, and speculation is done by allowing dependents of a predicted instruction to see the previous mapping of the predicted architectural register. As a result, while the paper is entitled "Storageless *Value Prediction* using Prior Register Values", Tullsen and Seng really proposed register equality prediction. To perform well, this **S**torageless VP (SVP) scheme relies on the compiler to create additional opportunities for reusing the physical register previously mapped to the architectural register written by the instruction to predict. In other words, recompiling is desirable.

In this work, we propose to revisit SVP and address its overheads. First, instead of relying on last mapping reuse and on the compiler to transform code to fit this reuse pattern, we rely on the *Instruction Distance* (IDist) to identify which physical registers contain values that can be used as predictions. The IDist of a given instruction tells us, within a trace of committed instructions, how far is the most recent[1] older instruction that produced the same result as the instruction of interest. Given the IDist of an instruction at Rename, a FIFO-like structure can be used to retrieve the identifier of the physical register expected to contain the result of the instruction. IDist has previously been considered for *Speculative Memory Bypassing* (SMB) [10].

With this scheme, inserting instructions to transform certain reuse patterns into last mapping reuse is not necessary, hence the overhead of SVP is reduced. Additionally, since IDist can be expressed using 8-10 bits depending on the instruction window size, we are able to get performance improvements with smaller structures than regular VP.

[1]Finding the **most recent** older instruction is actually not a requirement.

Lastly, SVP still allocates physical registers to predicted instructions, even though its dependents see the previous mapping [9]. In other words, SVP does not share physical registers between instructions. While this specific limitation will not impact performance compared to a baseline without SVP (because all instructions would get their own physical register anyway), SVP does not exploit the potential reduction in register pressure. As a result, we propose **R**egister **S**haring for **E**quality **P**rediction (RSEP). RSEP leverages recent propositions regarding physical register-sharing schemes [11], [12], [13] to *actually share* the physical register between the predicted instruction and the older instruction.

Overall, the paper makes the following contributions. First, we present RSEP, a register equality prediction implementation that requires neither ISA changes nor recompiling programs nor very large hardware structures. Second, we quantify the potential performance improvements brought by RSEP and VP through a state-of-the-art predictor both in isolation and in combination and study how those mechanisms overlap. Third, we study various parameters affecting RSEP.

The paper is organized as follows: Section II provides background on VP and SVP as well as physical register-sharing and IDist. Sections III and IV present two flavours of RSEP. Section V depicts the experimental framework, while Section VI describes experimental results. Finally, Section VII provides concluding remarks.

II. Related Work

A. Value Prediction

Value Prediction (VP) was first introduced by Lipasti et al. [3], [4] and Mendelson and Gabbay [5]. VP speculates on the values produced by instructions, allowing dependents to issue earlier, increasing ILP.

VP has been covered extensively in the late 90's [9], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], but fell out of fashion as implementing it appeared extremely complex and intrusive to existing microarchitectures.

Recently, VP has been regaining some traction in the community with several work aiming at simplifying its implementation. Perais and Seznec [7] first showed that mispredictions do not have to be detected at execute time as long as the predictor is highly accurate. Rather, validation can be done in-order, at Commit, and the pipeline fully squashed on a misprediction. This greatly reduces implementation complexity as instructions do not have to be replayed directly from the scheduler (a.k.a., Instruction Queue or IQ) when a value misprediction is detected.

Subsequently, the same authors proposed EOLE [8], a microarchitecture where instructions with ready operands flowing from the predictor are executed in-order in parallel with Rename, while predicted instructions are executed as late as possible in-order, at Commit. As a result, fewer instructions enter the scheduler, and its aggressiveness can be reduced. In [8], the issue-width is reduced from 6 to 4 with marginal performance degradation, but significant complexity reduction in the scheduler.

Lastly, Perais and Seznec [6] also provided a way to predict several instructions per cycle using only single-ported structures. The same work introduces the D-VTAGE value predictor that is inspired from the ITTAGE indirect target predictor [25] and the D-FCM value predictor [18]. D-VTAGE is currently considered state-of-the-art.

Nonetheless, 16-32KB of storage are still required by the predictor to obtain good performance. This is in contrast with the proposition of Tullsen and Seng [9] where predictions are stored in the Physical Register File (PRF). In that case, a predicted instruction is attributed a new physical register at Rename, but dependent instructions actually see the *old* mapping. This implementation allows last mapping reuse and relies on the compiler to transform patterns that could benefit from PRF-based VP into last-mapping reuse patterns. We argue that while extremely storage-efficient, this scheme has two main drawbacks. First, it requires recompiling programs to uncover all the potential for reuse. Second, it does not leverage physical register sharing to reduce register pressure.

B. Register Sharing

Physical register sharing is a powerful technique that enables *move elimination* [26], SMB [11], *register integration* [27] as well as other various rename-based optimizations [28], [29]. Nonetheless, sharing physical registers between instructions is not trivial because ownership of a register can change during execution as a result of a branch (or other) misprediction.

Many previous works assume the presence of per-register reference counters [26], [27], [28], [29], but those are problematic in the presence of checkpointing as reference counters do not straightforwardly lend themselves to checkpointing [11], [12], [13].

Roth [13] proposed a matrix-style way of tracking register sharing, which is checkpointable but hardly scalable due to its matrix nature. Battle et al. improved on this by greatly diminishing the size of the matrix [12]. In the former case, a roughly $ROB \times \#preg$ bit matrix is required [13], while in the latter case, $\#preg \times max\_sharers\_per\_reg$ bit matrix is required [12]. However, even in the latter case, a single checkpoint of the shared state requires 512 bits for 256 physical registers and two possible sharers per register. Consequently, this scheme is adapted to mechanisms generally requiring a low number of sharers.

Perais and Seznec [11] proposed a dual-counter scheme. The first counter tracks the number of references to a physical register (speculative), while the second counter tracks the number of committed references (architectural). This scheme is compatible with a checkpointing processor (only the first

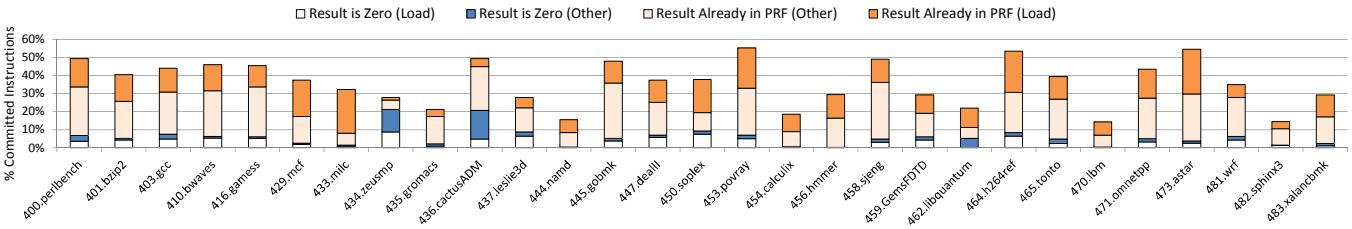

Fig. 1: Ratio of committed instructions for which the result is already in the PRF or is 0 (SPEC CPU'06 compiled for Aarch64 with gcc 4.9.3 -O3).

counter has to be checkpointed). They also argue that not all registers are shared at any given time. Therefore, instead of implementing two counters per physical register, entries in a small fully-associative buffer (the *Inflight Shared Registers Buffer*, or ISRB) can be allocated and freed on demand.

C. Instruction Distance

Sha et al. [10] proposed a microarchitecture that does not implement a Store Queue. Rather, the distance between a load and its producing store is predicted and Speculative Memory Bypassing is performed. That is, the data is forwarded to the consumer load without an associative lookup. The distance is computed by indexing in the *Data Dependency Table* (DDT) using the effective address at Commit. Each DDT entry contains the sequence number of the last committed store that wrote to this particular address. Subtracting the load's sequence number to the one contained in the DDT yields the distance, which is then used to train a *Distance Predictor*.

Recently, [11] applied the same scheme but considered bypassing from both stores and loads using the IDist. They also considered a TAGE-like IDist predictor, which outperformed the *gshare*-like predictor of Sha et al. [10].

III. Zero Prediction

We begin by pointing out that it is possible to implement a limited form of equality prediction through the PRF without going through the hassle of physical register sharing.

a) Hardwiring Registers to Specific Values: If a physical register is hardwired to a specific value (e.g., 0x0 or 0x1), then it does not have to be allocated or freed. This is leveraged explicitly by certain ISAs such as MIPS or ARM where an architectural register is defined as the *zero register*. This is – most likely– leveraged implicitly in recent x86 implementations by *zero-idiom elimination* [2]. In that case, instructions that put 0 in a register (e.g., *xor eax, eax*; *mov eax, 0*; *and eax, 0*; etc.) can be detected in the frontend, and their destination register renamed to the hardwired *zero register*. The same could be done for any value [12].

b) Zero Prediction: Zero-idiom elimination is non-speculative: Decode recognizes instructions that put 0 in a register. However, we can imagine a *zero* predictor that would allow us to rename many more destinations registers to the *zero register*. Although those instructions would still have to be executed to validate the prediction, register sharing would be trivial (no specific management required).

Figure 1 gives the ratio of instructions that are not zero-idioms yet write 0 to their destination register. The ratio is computed by cumulating across ten 100M instruction checkpoints per SPEC'06 benchmark [30]. The Figure differentiates between *load* instructions and other register-producing instructions. In a significant number of benchmarks, around 5% of the committed instruction actually produce or load 0, with some benchmarks reaching almost 20% (e.g., *zeusmp* and *cactusADM*). Therefore, there is potential for Zero Prediction even in the presence of zero-idiom elimination.

IV. Distance Prediction

As a second step, we consider a more general scheme that allows any result to be predicted, as long as said result was produced by a previous instruction, and the result is still in a *live* physical register. Figure 1 also depicts the potential for such a scheme by showing the ratio of committed instructions for which the produced value is already in the PRF (resolved at commit-time), cumulated across ten 100M checkpoints per SPEC'06 benchmark. In most cases, the ratio is around or greater than 5%, with many benchmarks featuring a ratio greater than 30%. As a result, there is high potential for reusing values already present in physical registers.

A. Identifying Pairs through their Result

As we attempt to identify pairs of instructions that produce the same result, the most straightforward is to compare couples of instruction results. However, full 64-bit comparison is not required since misprediction is allowed. Consequently, we propose to hash 64-bit results into a smaller value that will be used in comparisons, trading off accuracy for implementation complexity and power consumption.

To do so, we need a hash function to map a 64-bit value into a much smaller word. To minimize delay, the hash function should be simple enough, e.g., use shift and XOR operators only, but it should also minimize the number of false positives.

As a first step, we consider a simple folding function that iteratively XORs n-bit chunks of the result into a n-bit

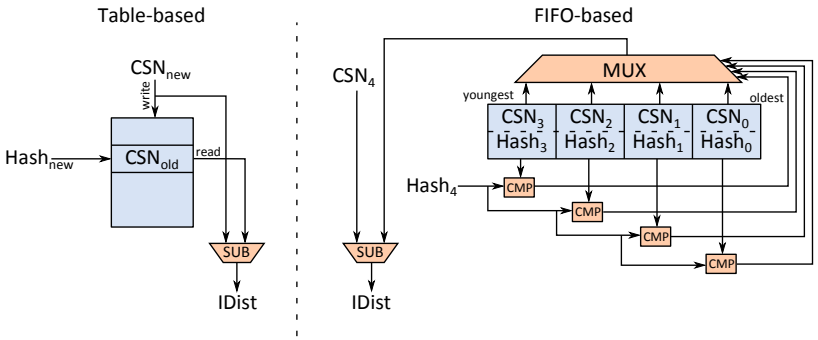

Fig. 2: Example showing how the Instruction Distance is computed after Commit by using Commit Sequence Numbers (CSNs) and either a table-based scheme (left) or a FIFO-based scheme (right).

hash. To minimize false positives, n should be different from a power of two, to avoid many collisions for common value (e.g., -1 in 2's complement and 0x0 would both hash to 0x0 with an 8- or 16-bit fold). For instance, with $n = 14$, the function is defined as follows:

$$Hash_{[13..0]} = val_{[13..0]} \oplus val_{[27..14]} \oplus val_{[41..28]} \oplus val_{[55..42]} \oplus val_{[63..56]}$$

Hashes are computed at execute time, at the output of functional units (FUs). This does not impact the execution critical path since dependents may get the full result on the bypass network while the hash is being computed. Hashes are written in a dedicated register file that mirrors the physical register file (i.e., management is trivial). This *Hash Register File* (HRF) is written at Writeback and read at Commit.

B. Linking Pairs – Design Space

1) Previous Proposition – Pairing through The Data Dependency Table: Sha et al. [10] relied on a Data Dependency Table (DDT) to identify pairs of instructions that access the same address to perform Speculative Memory Bypassing. An entry in the DDT contains a *Commit Sequence Number* (CSN). At Commit, a load accesses the DDT with its effective address and gets a CSN. It then subtracts its own CSN to the CSN read from the DDT to determine its IDist. Conversely, a store simply puts its own CSN in the entry. All these steps are out of the critical path. In [11], loads also write their CSNs in the DDT. This latter scheme is depicted in the left part of Figure 2.

Adapting the DDT to RSEP appears straightforward: only indexing changes, as it should be done with the hash present in the HRF rather than the effective address.

However, if such a structure may be adapted to the case of SMB because only loads and stores have to access it, it may in fact not be adapted to RSEP. The reason is that superscalar processors may commit several instructions per cycle, hence the DDT will have to support multiple accesses per cycle. If the DDT were indexed using the PC of instructions, banking could be envisioned to support multiple accesses per cycle without implementing several ports. Unfortunately, since hashes of register values are used to index it, banking is of little help, and the DDT is simply impractical for our purpose.

2) Simple Pairing through a FIFO Structure: An alternative way to compute the IDist is to keep a history of the n last retired instructions in a FIFO. Each cycle, the hashes of c retired instructions are compared to the n older hashes. They are also compared with each other to make sure that no pair goes unnoticed. Finally, the c retired instructions are pushed in the FIFO. This structure is depicted in the right part of Figure 2.

While this scheme alleviates the need for a highly ported structure such as the DDT, a significant number of comparisons must be performed each cycle. In particular, for a commit-width of 8, 28 $hash$-bit comparisons must be performed just within the commit group. Then, each instruction of the commit group requires another n $hash$-bit comparisons. Assuming that the last 256 instructions are kept in the FIFO and considering 14-bit hashes, this amounts to 2076 14-bit comparisons[2] each cycle, for a buffer size of 768 bytes (10-bit CSN).

Consequently, this scheme should most likely be limited to small buffer sizes to minimize power consumption. This would entail that some pairs fitting within the processor window would not be captured (i.e., IDist $<$ ROB_size with IDist $>$ FIFO_size).

3) Cost-Effective Pairing through Sampling: To limit the number of comparisons performed each cycle, sampling can be applied. That is, for a commit group containing c instructions, a single one can be picked randomly and attempt to identify a matching hash in the FIFO history.

However, this is likely to hinder potential by greatly increasing training time. In particular, instead of being pre-

[2]2048 comparators to check 256 hashes for 8 committed instructions, and 28 comparators to check hashes within the group of 8 committed instructions.

dicted after o occurrences (at best), an instruction will now begin to be predicted after $o \times commit\_width$ occurrences (assuming it is picked once every $commit\_width$ time it is retired). Given that in our experiments, o is 255 (confidence counters saturate at 255 and we predict only when the counter is saturated), training time will increase *significantly*.

a) Sample to Identify Likely Candidates: To mitigate this phenomenon, we propose to perform sampling only to identify *likely* candidates for RSEP. In other words, we define two confidence threshold in the distance predictor. The first, *use_pred*, is the value that the confidence counter must exceed for the instruction to be predicted. We found that on average, $use\_pred = 255$ greatly limits the number of mispredictions, maximizing potential. The second, *start_train*, is the value that the confidence counter must exceed for the instruction to be considered a *likely candidate* for RSEP.

b) Finalize Training With the Validation Mechanism: *Likely* candidates for RSEP will flow through the prediction validation mechanism described in Section IV-F and explicitly compare their destination register with the one they would have shared if they had been predicted. They will not check the FIFO at Commit.

Assuming validation has low performance and complexity overheads, this allows to continue training likely candidates by performing a single 64-bit comparison per instruction. In addition, several instructions can be trained each cycle, while Commit can only compute one IDist each cycle with the sampling approach.

As a first order approximation, an instruction can begin to be predicted after $(start\_train \times commit\_width) + (use\_pred - start\_train)$ occurrences. For instance, for a *training* threshold of 7 and a commit width of 8, this amounts to 304 occurrences respectively (at best), instead of 255 occurrences without sampling.

C. The Distance Predictor

Once the IDist has been computed for a particular dynamic instruction, the distance predictor must be trained so that an IDist can be provided to subsequent instances of the corresponding static instruction. Sha et al. used a *gshare*-like predictor with two tables to predict the distance and perform Speculative Memory Bypassing. The first table is direct-mapped, while the second one uses a hash of the PC and the global branch history [10]. Recently Perais and Seznec used a TAGE-like structure and showed that it outperformed a *gshare*-like predictor when considering Speculative Memory Bypassing [11].

Consequently, we use a TAGE-like predictor to perform IDist prediction. This predictor features several partially tagged tables indexed by a hash of the PC and some bits of the global branch history (as well as some bits of the global path history). Those partially tagged tables are backed up by a PC-indexed, untagged base table. Each entry contains a distance (8-9 bits for a 256- 512-entry ROB), as well as a confidence counter. Indeed, since mispredicting is very expensive (full pipeline squash as in regular VP), speculation must take place only if the predictor is very confident that the prediction is correct [7]. Moreover, each partially tagged entry contains a partial tag as well as a single *useful* bit used by the replacement policy. We refer the reader to [31] for more details on how TAGE operates.

As a first step, we consider a very large predictor, with 6 components of 1K entries[3] in addition to a 16K-entry[4] base component, amounting to 42.6KB. However, we show in our experiments that good results can be obtained with a predictor requiring only around 10KB of storage.

D. Complexity of the Proposed Structures

1) Hash Register File: The HRF is a register file-like structure that requires at most 8 write ports and 8 read ports assuming an 8-wide pipeline. However, if writes are random-access because writeback is done out-of-order, reads are done at Commit and are therefore in-order. Consequently, the HRF can be banked, each bank featuring only one read port. Note that the banking of the HRF must mirror the banking of the PRF. In addition, similarly to the PRF, all writes are guaranteed to be mutually exclusive, and all writes are guaranteed to be exclusive with reads.

Moreover, HRF registers are only n-bit wide with n the width of hashes. In our experiments, we found that good results were obtained with 14-bit hashes.

Thus, the HRF has to be contrasted with the PRF that implements 64-bit registers and should handle up to 16 random reads and 8 random writes per cycle assuming the same 8-wide pipeline. Given that PRF area and power grow quadratically with port count and quasi linearly with the register width [33], we expect the HRF to represent less than 5% of the PRF area.

2) FIFO History: Because it is managed as a FIFO, the history does not require many random access ports. Rather, each cycle Commit retires instructions, those instructions are inserted at the tail of history while the head instructions are removed. Note that the history can be implemented as a circular structure and managed through pointers, i.e., there is no need to explicitly shift the content of entries.

To handle superscalar behavior without sampling, each entry must provision $commit\_width$ comparators. This amounts to 2076 comparators for a 256-entry structure able to handle a *commit_width* of 8. Note that all comparators are required only if Commit retires 8 instructions per cycle at a steady rate. If less instructions are committed each cycle, then some comparators will not be used. In particular, as

[3]One 8-bit distance, one 3-bit probabilistic confidence counter [7], [32], one useful bit and a partial tag (respectively 13,14,15,16,17 and 18-bit wide) per entry.

[4]One 8-bit distance and one 3-bit probabilistic confidence counter per entry.

retiring *commit_width* instructions that all produce a register is unlikely, it is possible to implement less comparators than the theoretical maximum at the cost of skipped comparisons if Commit retires more instructions than expected. In our experiments, we found that on average, 6 (respectively 4) comparators are sufficient in more than 95% (respectively 70%) of the cases in most SPEC'06 benchmarks. On our framework, only *lbm* and *gamess* frequently retire 8 instructions eligible for distance prediction ($> 25\%$ and $> 5\%$ of the commit groups, respectively).

The remainder of the complexity lies with how the IDist is actually computed, as this depends on what information is stored in the history.

a) Explicit IDist Computation: One possibility is for the IDist to be explicitly computed by subtracting the instruction's CSN to the older CSN matching the hash. This implementation requires that 1) Each FIFO entry provision space for a CSN and 2) $Commit\_width$ CSN-bit adders be implemented to compute the IDist. Those adders should be able to handle CSN wrap-around. In this case, a 256-entry history requires 768 bytes of storage, but since only instructions that produce a register are pushed in the buffer at retirement, less entries may suffice to capture a distance of 256.

b) Implicit IDist Computation: A second implementation would push all instructions in the buffer, even those that do not produce a result and therefore cannot be paired with other instructions. This allows the instruction distance to be computed trivially. That is, since the instruction distance is respected in the buffer, a hash match for a given pair of instructions corresponds to a statically defined instruction distance. Consequently, the adders are not required, and neither are the CSNs. In that case, a 256-entry history requires 448 bytes of storage.

c) Tradeoff: In a nutshell, the main benefit of the first implementation is that instructions that do not produce a result are not pushed in the buffer, virtually increasing its size. However, it requires more storage in each buffer entry, which will increase power consumption. In addition, more logic is required as adders must be implemented.

On the contrary, the second implementation may be more power efficient as adders are not required and entries are smaller. However, because "useless" instructions occupy entries in the buffer, more entries may be required to capture pairs that were captured with less entries in the first implementation.

Therefore, depending on the number of additional entries required for the second implementation to capture as many pairs as the first one, one implementation may be more power efficient than the other. In our experiments, we only consider the first implementation, but we stress that there exists a tradeoff regarding the buffer implementation.

d) Distance Predictor: The distance predictor only stores distances that are reachable within the processor window. That is, for a 256-entry ROB, 8-bit fields are sufficient. As a result, at a similar number of entries, a TAGE-based distance predictor will require significantly less storage than a TAGE-based value predictor.

In addition, contrary to the Data Dependency Table that is indexed using register value hashes, the distance predictor is indexed using the instruction PC. As a result, it can be banked to handle multiple accesses per cycle without requiring multiporting [6], [34]. That is, while the distance predictor will still take up several KB of storage, its complexity will remain comparable to that of the branch predictor.

E. Sharing Registers

1) Reorder Buffer: Our scheme relies on the fact that inflight instructions are tracked in a FIFO structure, the Reorder Buffer (ROB). In particular, with the IDist of an instruction, we can index into the ROB and hit the instruction that is expected to produce the same result as the predicted instruction. It is then straightforward to retrieve the physical register identifier of the destination register. Note that for clarity's sake, we consider that the ROB contains the destination identifier and can be accessed in a cost effective fashion. In practice, a dedicated FIFO structure managed with the ROB head and tail pointers may be implemented, alleviating the need for additional ROB reads.

2) The Inflight Shared Registers Buffer: RSEP requires physical register reference counting. To accommodate processors recovering from branch – or other – mispredictions by restoring a checkpointed state, we use the Inflight Share Register Buffer (ISRB) [11].

The ISRB is a small fully-associative structure where entries are allocated when a reuse opportunity is detected. Each entry features two counters, *referenced*, that tracks the number of references to the register (including speculative ones) and *committed*, that tracks the number of committed references. Each time a register is reused, *referenced* is increased, and each time a register is de-referenced, *committed* is increased. If after this update, *committed* is strictly greater than *referenced* (or *committed* overflows), the entry and register are freed. Similarly, checkpoint recovery consists in restoring the checkpointed value of *referenced* and freeing the entry and register if *committed* is strictly greater than *referenced*. We refer the reader to [11] for a more detailed description of the ISRB. If no ISRB entry is free, no sharing takes place.

F. Validating Predictions

Equality prediction – in the form of a predicted distance – must be validated. Validating equality predictions can be done in several manners, some of which may significantly increase complexity and power consumption. In the next paragraph, we depict a possible implementation that attempts to minimize both, nonetheless stressing that final complexity is strongly dependent on the overall design.

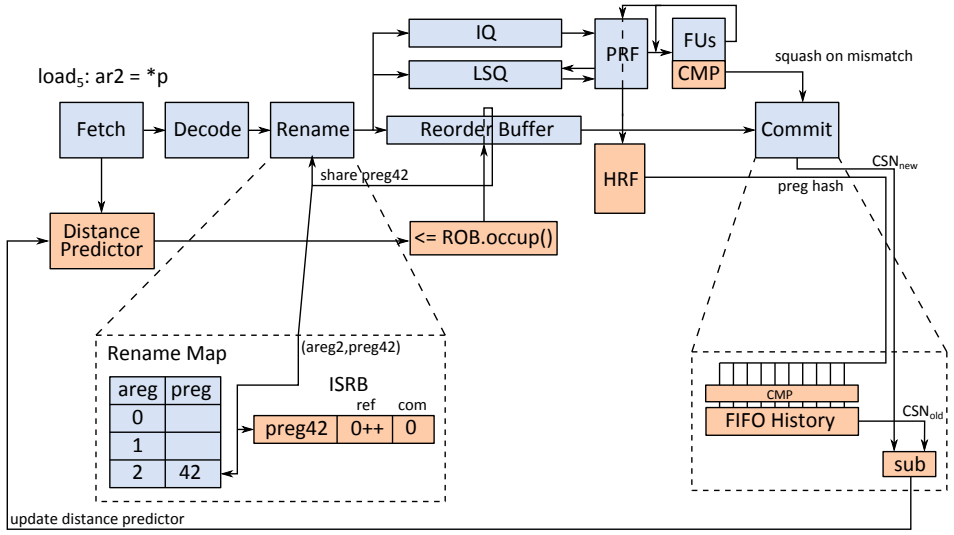

Fig. 3: RSEP pipeline diagram. Blocks belonging to the baseline superscalar design are in blue, while new blocks added for RSEP are in orange.

1) A Possible Validation Flow: An acceptable solution to perform validation should not increase the number of registers read from the PRF each cycle, and it should not force a physical register to be allocated to predicted instructions.

To respect the first requirement, we propose to inject a *compare* μ-op to perform validation. Through this, the distance-predicted instruction does not have to read an additional register at issue time. However, to respect the second requirement, we must guarantee that the injected μ-op executes back-to-back with its corresponding distance-predicted instruction. This allows one operand to be caught on the bypass network, and avoids the need to buffer results of predicted instructions in the PRF or a dedicated structure.

To enforce this behavior, the predicted instruction is made dependent on the initial instruction it is sharing a register with. This ensures that when the validation μ-op issues, the shared register (or bypass network) has the value to compare to the actual result of the predicted instruction. Adding this additional dependency is less of a hurdle if one considers *matrix* schedulers [35], but may have a significant cost for broadcast-based schedulers if the additional dependency is not already provisioned (e.g., to handle 3/4-input μ-ops).

Then, the picker has to be modified to always prioritize validation μ-ops so that they are always issued in the cycle following their corresponding predicted instruction.

Finally, validation throughput should be high enough w.r.t. the maximum number of predicted instructions that issue each cycle. The most straightforward way is to issue a validation μ-op to the same functional unit as the predicted instruction and to put a 64-bit comparator at the output of every relevant FU.

a) Multi-cycle instructions: This mechanism is straightforward for single-cycle instructions. However, it is not clear how multi-cycle (e.g., multiplication) and variable-latency (e.g., division, load) instructions can be handled.

To handle multi-cycle but *fixed* latency instructions, a counter can be put in the IQ entry such that the validation μ-op becomes ready only after $inst\_latency$ cycles (the counter is decremented each cycle). In this fashion, the validation μ-op will see the instruction result on the bypass network when it enters Execute.

To handle variable latency instructions, a possibility is to wake up the validation μ-op only when the instruction finishes, and to buffer the instruction result within the bypass network the time it takes for the validation μ-op to read the shared physical register (e.g., 2-4 cycles).

It should also be noted that in theory, because of multi-cycle ans variable latency instructions, it is still possible that more validation μ-ops than there are issue ports must be issued in a given cycle. However, we believe that by carefully allocating FUs to issue ports at design time, this can be made rare enough to squash and inhibit prediction for a few hundred cycles when detected.

b) Reducing Validation Footprint: At this point, the cost of validation appears significant since 1) Predicted instructions occupy two entries in the ROB and IQ and 2) Predicted instructions consume two issue slots of their instruction type (i.e., when the validation μ-op of a load issues, it prevents an actual load from issuing.)

1) can be addressed as injecting distinct μ-ops is not necessary if validation is performed in this fashion. That is, instead of dedicating ROB and scheduler entries to validation μ-ops, we can force the picker to issue predicted instructions

twice. First, the instruction is issued normally. Second, the instruction is issued again, but with the semantics of a comparison reading a single operand in the PRF: the destination register of the predicted instruction. The second operand will be available directly at the FU. Thanks to this, pressure on the ROB and IQ is reduced, although predicted instructions must retain their scheduler entry for at least an additional cycle.

Unfortunately, we observed that issuing the validation μ-op to the same functional unit as the predicted instruction significantly degraded performance in several benchmarks. The reason is that when prediction coverage for loads is high, load throughput cannot be sustained because regular loads compete with load validation μ-ops for the two load issue ports.

Sustaining load throughput while being able to issue validation μ-ops for predicted loads requires that both a validation μ-op and a load be steered to a single load issue port in a single cycle. This is likely to entail significant change in the picker and datapaths.

As a result, we propose to leverage the global bypass network and allow a validation μ-op to be issued to a different port than the one where the instruction it is validating issued. Then, non-load ports can be given priority when issuing validation μ-ops, such that load issue ports will be used for validation purpose only if many validation μ-ops are issued in a single cycle. This will also mitigate the performance loss incurred by using load units to perform simple comparisons.

c) Summary: This possible implementation of equality prediction validation issues predicted instructions two times, the second instance being in charge of performing a simple 64-bit comparison without even writing its result in the PRF and bypass network. The overhead is kept minimal and stems from delaying the issue of actual instructions when there are validation μ-ops to issue, as well as the additional dependency in the scheduler. No complex functional unit is locked just to perform the comparison, which allows to sustain the maximum load throughput.

In addition, because we rely on the scheduler to make a predicted instruction depend on the instruction producing the shared register, we are able to detect instructions for which the reused register is the last arriving operand (i.e., the dependency chain is *lengthened* by RSEP). However, we found that it is generally more interesting to allow such instructions to train the predictor because this behavior is often transient. Only in a couple of benchmarks is performance higher when those dynamic instructions are forbidden from training the distance predictor.

G. Overview

Figure 3 shows a high-level overview of the RSEP pipeline. First, as instructions flow through the frontend, they access the distance predictor, which provides them with an IDist if it is confident enough. The distance is used to retrieve a register index from the ROB, which is then used as the renamed destination register at Rename. The ISRB is also notified that one more instruction references this register. Then, at Execute, the prediction is validated by issuing predicted instructions a second time. Finally, at Commit, retired instructions enter the FIFO history and one is selected randomly to check its hashes against the hashes of older instructions. The IDist predictor is also trained at Commit. In the event of a misprediction, the pipeline is flushed once the mispredicted instruction reaches the head of the ROB.

As a side note, we point out that while reminiscent of *general reuse* through *register integration* as proposed by Petric et al. [27], RSEP is in fact more general. In particular, it allows unrelated instructions to share the same physical register while [27] allows reuse only for instructions with the same opcode and the same renamed operands.

H. Overlap With Existing Techniques

1) Move Elimination: Move elimination is a known feature relying on physical register sharing [26]. The idea is to implement circuitry to detect register-to-register moves (or equivalent idioms such as *orr x0, zero_reg, x1* or *add x0, zero_reg, x1* in Aarch64), and execute them at Rename by renaming the destination register to the source register. Move elimination is of particular interest in x86 since one of the sources also acts as the destination in most instructions, i.e., registers often have to be copied if the variable they contain has to be reused later. Recent Intel processors implement move elimination [2].

Ideal RSEP encompasses move elimination. Indeed, the move to eliminate will be predicted to depend on the instruction that produces its source register, and its destination register renamed to the destination register of the oldest instruction. This is equivalent to move elimination. However, move elimination is *non-speculative*, which first guarantees that no misprediction will take place, and second allows to not execute the move.

As a result, when considering RSEP, we also implement move elimination and do not perform distance prediction for 64-bit register-to-register moves.

2) Speculative Memory Bypassing: Speculative Memory Bypassing attempts to identify *def-store-load-use* chains to collapse them into *def-use* pairs by renaming the destination register of the load to the source register of the store. In particular, in Sha et al., Distance Prediction is used to identify store-load pairs to the same address [10]. The main difference with RSEP is that the Data Dependency Table is indexed using the effective address of memory instructions. In RSEP, values are used, which has an advantage.

Indeed, since RSEP links instructions through their *result*, loads can use registers from instructions on a different dependency chain as long as the values match. Therefore,

Front End	L1I 8-way 32KB, 1 cycle, 128-entry ITLB 32B fetch buffer, 8-wide fetch over 1 taken branch TAGE 1+12 components [31] 15K entry total, 17 cycles min. mis. penalty; 2-way 4K-entry BTB, 32-entry RAS 8-wide decode 8-wide rename with zero-idiom elimination [2]
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ (STLF lat. 4 cycles), 235/235 INT/FP registers 2K-SSID/1-K LFST Store Sets, not rolled-back on squash [36] 8-issue, 4ALU(1c) including 1Mul(3c) and 1Div(25c*), 3FP(3c) including 1FPMul(3c) and 1FPDiv(11c*), 2Ld/Str, 1Str Full bypass 8-wide retire
Caches	L1D 8-way 32KB, 4 cycles load-to-use, 64 MSHRs, 2 load ports, 1 store port, 64-entry DTLB, Stride prefetcher (degree 1) Unified private L2 16-way 256KB, 12 cycles, 64 MSHRs, no port constraints, Stream prefetcher (degree 1) Unified shared L3 24-way 6MB, 21 cycles, 64 MSHRs, no port constraints, Stream prefetcher (degree 1) All caches have 64B lines and LRU replacement
Memory	Dual channel DDR4-2400 (17-17-17), 2 ranks/channel, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Min. Read Lat.: 36 ns. Average: 75 ns.

TABLE I: Simulator configuration overview. *not pipelined.

there is more potential for the latency of loads to be hidden and ideal RSEP encompasses SMB.

In fact, since SMB is also speculative, there is no obvious performance advantage in considering SMB only versus full blown RSEP.

V. Evaluation Methodology

We run experiments using the cycle-level simulator gem5 [37]. We use the 64-bit ARMv8 ISA (Aarch64). We model an aggressive 8-wide microarchitecture for which the parameters are summarized in Table I. The parameters are on par with Intel Haswell's. When considering Value Prediction, we use the parameters given in [6] (amounting to a roughly 256KB D-VTAGE predictor).

a) Benchmarks: We consider single-thread workloads (SPEC'06 [30] compiled with *gcc4.9 -O3 -mtune=armv8*). We use the whole benchmark suite and consider *ref* inputs. We uniformly collected 10 checkpoints per benchmark. We warm-up the processor structures for 50M instructions then collect statistics for 100M instructions. We then report IPC for each benchmark as the harmonic mean of the 10 individual IPC.

VI. Experimental Results

A. Performance

1) Comparison with Other Mechanisms: Figure 4 reports performance improvements obtained with different mechanisms: zero prediction, move elimination, equality prediction, value prediction and finally, a combination of equality prediction and value prediction. We point out that both equality and value prediction cover zero prediction, and recall that as a side-effect of the ability to share physical registers when doing distance prediction, we also implement move elimination when equality prediction is present. Finally, for this first experiment, we consider an ideal validation mechanism for zero prediction and RSEP as well as a large FIFO history for RSEP ($>>$ ROB).

Figure 5 reports the percentage of committed instructions that is processed by each mechanism when equality prediction is used, as well as when the combination of equality prediction and value prediction is used.

Let us begin by instructions that can be zero predicted. While Figure 1 in Section III suggests that many instructions produce 0, Figure 5 suggests that few of those instructions do so in a regular fashion, as few zero predictions take place in general. This translates in small speedups in two benchmarks, *gamess* and *libquantum*.

Second, even though we are considering Aarch64 and not x86, move elimination is able to capture more than 5% of the committed instructions in around a third of the considered benchmarks. We recall that in this experiment, only 64-bit moves are considered. Once again, this translates in marginal speedups in only two benchmarks, *dealII* and *xalancbmk*.

Moving on to equality prediction, we can observe in Figure 5 that more than 10% of the committed instructions are captured in 7 benchmarks out of 29. This generally translates in marginal speedups, except in *mcf*, *dealII*, *hmmer*, *libquantum*, *omnetpp* and *xalancbmk*.

There is no particular correlation between the category of instructions that are predicted and the speedup that is observed. In particular, in *mcf*, almost only loads are predicted, but in *dealII*, most predicted instructions are not loads.

Value prediction is able to capture many more instructions in a significant amount of benchmarks, although most are non-load instructions, as depicted in Figure 5. However, this does not always translate in higher speedups than with distance prediction. For instance, in *mcf*, *dealII*, *hmmer*, *libquantum* and *omnetpp*, equality prediction performs better, even though far fewer instructions are captured. On the contrary, value prediction performs better in *perlbench*, *wrf*, *xalancbmk* and several other benchmarks where speedup is less pronounced (e.g., *zeusmp*, *gromacs*).

Finally, we combine equality and value prediction and observe that in a single case, *perlbench*, RSEP is redundant with VP. For this benchmark, we can see in Figure 4 that speedups are similar when using value prediction only and when using the combination of both mechanisms. In Figure 5, this translates in value prediction covering close to all distance-predicted instructions. However, in virtually all other cases, VP does not fully overlap with equality prediction. As such, there is benefit in combining both mechanisms (e.g., *mcf*, *dealII*, *hmmer*, *libquantum*, *omnetpp*, *xalancbmk*) even though **both** distance and value predictors are TAGE-based.

The reason for this behavior is that equality prediction captures equality between – potentially unrelated – instructions. It does not attempt to predict the particular value. As

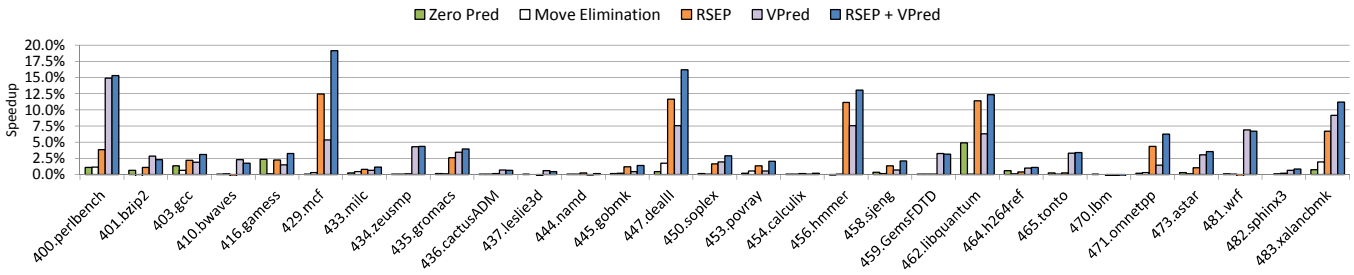

Fig. 4: Speedup over baseline using different performance-optimizing mechanisms.

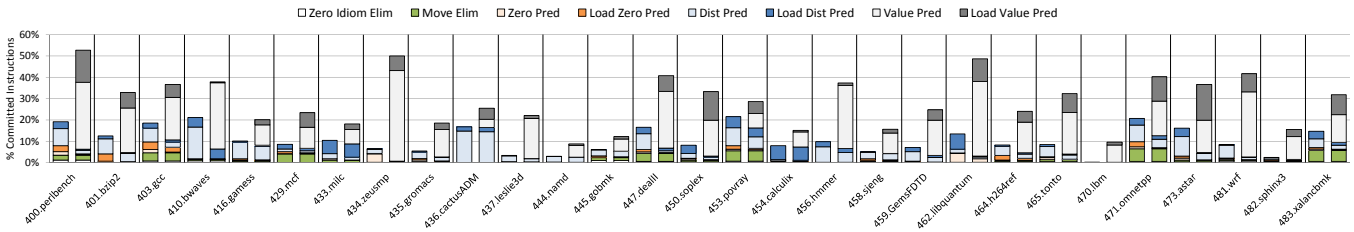

Fig. 5: Percentage of committed instructions covered by each mechanism. Two bars are shown for each benchmark, the first considers RSEP only, while the second considers VP on top of RSEP.

a result, the value may not be predictable by conventional means, yet performance gain can still be obtained through sharing. On the contrary, value prediction focuses on capturing the exact result.

2) Impact of the History Depth: We experimented with different FIFO history depth for RSEP (only instructions producing a result are pushed in the history). We found that in general, the performance obtained with a FIFO history is higher (from 0 to 2.5 additional speedup percentage points) than the performance obtained with an unrealistic 16KB DDT. The reason is that using a FIFO allows to match with several older instructions and select the matching distance that corresponds to the predicted distance (assuming the predicted distance is propagated with instructions, e.g., in a dedicated FIFO). This decreases noise due to some instructions matching by chance rather than because they always produce the same result. Using a DDT forces to match with the most recent older instruction, and therefore suffers from noise due to "per chance" matches.

Regardless, on the SPEC'06 benchmarks and for a maximum of 224 instructions in flight between Rename and Commit, a history with only 128-entry appears sufficient. In its ideal implementation (i.e., without sampling), such a structure requires 1024 14-bit comparators and 384 bytes of storage (10-bit CSNs and 14-bit hashes). We note that except in a handful of cases (*hmmer* and *xalancbmk*), keeping only 32 older instructions is sufficient to achieve most of the potential, meaning that matching pairs are often close.

3) Impact of ISRB Size: We ran experiments varying the number of entries in the ISRB and found that implementing only 24 entries of two 6-bit counters (tagged by the physical register identifier) was not detrimental to performance. As a result, the storage cost of the sharing mechanism itself remains limited.

4) Impact of Validation and Sampling: Figure 6 depicts the impact of the validation mechanism on the potential performance improvement of RSEP. We consider an ideal (free) mechanism against a mechanism where predicted instructions are issued a second time either to the FU where the predicted instruction issued (blue bar) or to any available FU (white bar). As mentioned in Section IV-F1, issuing a second time has very limited impact, but issuing to the same FU is problematic in many cases as maximum load throughput cannot be sustained. Consequently, the solution relying on the bypass network to allow validation μ-ops to be issued to any functional unit should be privileged.

The two last bars of Figure 6 show performance when sampling is applied at Commit. That is, only one committing instruction – chosen randomly – accesses the FIFO history to attempt to find a match. However, once a certain confidence threshold (15 and 63 in Figure 6) is reached in the distance predictor, instructions flow through the validation mechanism of RSEP to continue training the predictor. Sampling has moderate impact in *dealII* and *hmmer*.

However, in *bzip*, a slowdown of around 2% is observed when the threshold is 15. This is due to the fact that equality prediction occasionally lengthens the critical path, and that because of sampling, this lengthening also happens during the training phase, i.e., when RSEP does not provide any gain. Using a higher threshold limits this effect, therefore, we choose a threshold of 63 in further experiments to minimize slowdown on the overall suite. *bzip2* is one of the two benchmarks – *perlbench* is the other – that would benefit from forbidding instructions that occasionally increase the

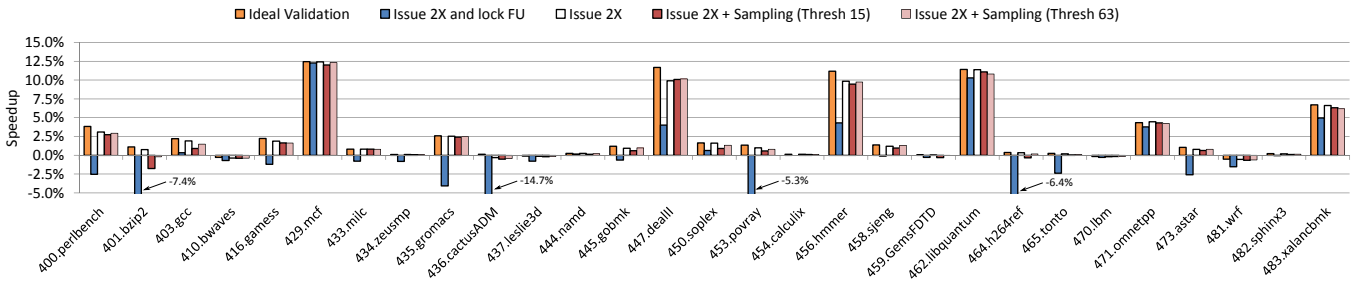

Fig. 6: Impact of equality prediction validation and sampling on performance.

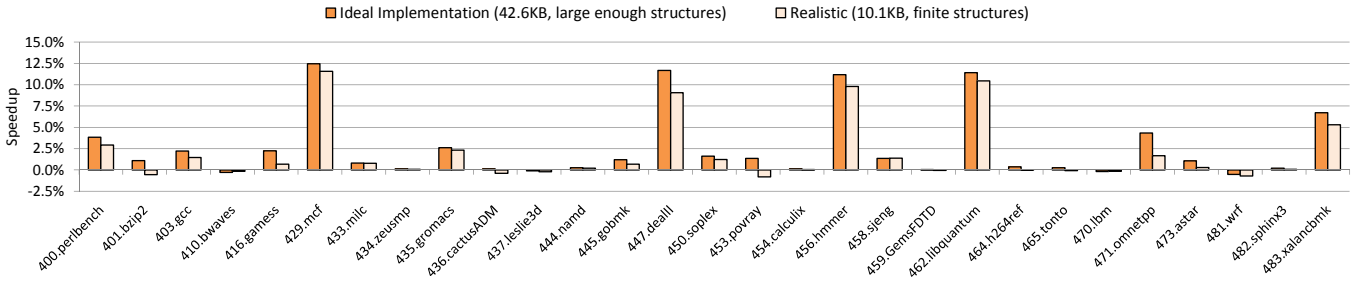

Fig. 7: Performance of ideal RSEP and realistic RSEP using much smaller structures.

length of the critical path to update the distance predictor.

B. Putting It All Together

Finally, we run experiments considering all the results obtained so far. In particular, we consider a 128-entry FIFO history, a 24-entry ISRB with 6-bit counters, and we perform sampling to train the distance predictor, using a threshold of 63. We also reduce the size of the predictor to 10.1KB by using a 2K-entry base predictor, six 512-entry tagged components, and using smaller partial tags (from 5 to 10 bits respectively).

Figure 7 gives speedup over the baseline for this configuration and the ideal configuration. Overall, using more realistic sizes and training/validation mechanisms has a noticeable impact on the performance improvement brought by RSEP. However, speedup is still present and close to 10% in 4 benchmarks, while remaining small for most of the suite. In those experiments, prediction accuracy is always greater than 99.5% while average coverage is 28.5% of the eligible dynamic instructions (e.g., stores and branches are not eligible). Moreover, we note that the overall storage required by this particular implementation of RSEP is around 10.8KB: 10.1KB for the predictor, 384B for the FIFO history, 224B to propagate distances in a dedicated FIFO so matching in the FIFO history can privilege the predicted distance rather than the shortest one, and 63B for the ISRB (not counting checkpoint space). This has to be contrasted with the 16-32KB dedicated to Value Prediction in [6].

VII. Conclusion

Amdahl's Law calls for new architectural features to improve sequential performance. In this paper, we presented such a feature: equality prediction leveraging register renaming, RSEP. RSEP identifies pairs of instructions producing the same result and let the younger instruction use the physical destination register of the older instruction as its own destination register. This study illustrates the large redundancy present in the physical register file. Our proposition, RSEP, shows that this potential could be exploited: it allows to increase performance by 5% to 11% in five SPEC'06 benchmarks, while requiring around 10.8KB of storage and no multi-ported structures.

While many instructions covered by RSEP can also be value predicted, we found that RSEP is able to significantly improve performance over value prediction in benchmarks where it performs well on its own. Consequently, RSEP can be seen as yet another opportunity to perform speculation that has moderate storage and complexity requirements.

Acknowledgment

This work was partially supported by an Intel research gift.

References

[1] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2013.

[2] Intel. (2014, September) Software optimization manual. [Online]. Available: http://www..fr/content/www/fr/fr/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

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

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

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

[6] A. Perais and A. Seznec, "BeBoP: A cost effective predictor infrastructure for superscalar value prediction," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

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

[8] ——, "EOLE: Paving the way for an effective implementation of value prediction," in *Proceedings of the International Symposium on Computer Architecture*, 2014.

[9] D. Tullsen and J. Seng, "Storageless value prediction using prior register values," in *Proceedings of the International Symposium on Computer Architecture*, 1999, pp. 270–279.

[10] T. Sha, M. M. K. Martin, and A. Roth, "NoSQ: Store-load communication without a store queue," in *Proceedings of the International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 285–296.

[11] A. Perais and A. Seznec, "Cost-effective physical register sharing," in *Proceedings of the International Symposium on High-performance Computer Architecture*. IEEE Computer Society, 2016, p. TBD.

[12] S. Battle, A. Hilton, M. Hempstead, and A. Roth, "Flexible register management using reference counting," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2012, pp. 1–12.

[13] A. Roth, "Physical register reference counting," *Computer Architecture Letters*, vol. 7, no. 1, pp. 9–12, Jan 2008.

[14] M. Burtscher and B. G. Zorn, "Exploring last-n value prediction," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[15] B. Calder, G. Reinman, and D. Tullsen, "Selective value prediction," in *Proceedings ofthe International Symposium on Computer Architecture.*, 1999.

[16] F. Gabbay and A. Mendelson, "The effect of instruction fetch bandwidth on value prediction," in *Proceedings of the International Symposium on Computer Architecture*, 1998.

[17] ——, "Using value prediction to increase the power of speculative execution hardware," *ACM Trans. Comput. Syst.*, vol. 16, no. 3, pp. 234–270, Aug. 1998.

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

[19] B. Rychlik, J. Faistl, B. Krug, and J. Shen, "Efficacy and performance impact of value prediction," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1998.

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

[21] ——, "Implementations of context based value predictors," Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Tech. Rep. ECE97-8, 1998.

[22] A. Sodani and G. Sohi, "Understanding the differences between value prediction and instruction reuse," in *Proceedings of the International Symposium on Microarchitecture*, 1998.

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

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

[25] A. Seznec, "A 64-Kbytes ITTAGE indirect branch predictor," *Journal of Instruction-Level Parallelism*, 2011.

[26] 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 *Proceedings of the International Symposium on Microarchitecture*, 1998.

[27] V. Petric, A. Bracy, and A. Roth, "Three extensions to register integration," in *Proceedings of the International Symposium on Microarchitecture*, 2002, pp. 37–47.

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

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

[30] Standard Performance Evaluation Corporation. (2006) CPU. [Online]. Available: http://www.spec.org/cpu2006/

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

[32] N. Riley and C. B. Zilles, "Probabilistic counter updates for predictor hysteresis and stratification," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

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

[34] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor," in *Proceedings of the International Symposium on Computer Architecture*, 2002, pp. 295–306.

[35] P. G. Sassone, J. Rupley, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," in *Proceedings of the International Symposium on Computer Architecture*, 2007, pp. 335–346.

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

[37] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.