Focused Value Prediction*

Sumeet Bandishte
Processor Architecture Research Lab
Intel Labs
Bengaluru, India
sumeet.bandishte@intel.com

Jayesh Gaur
Processor Architecture Research Lab
Intel Labs
Bengaluru, India
jayesh.gaur@intel.com

Zeev Sperber
Intel Corporation
Haifa, Israel
zeev.sperber@intel.com

Lihu Rappoport
Intel Corporation
Haifa, Israel
lihu.rappoport@intel.com

Adi Yoaz
Intel Corporation
Haifa, Israel
adi.yoaz@intel.com

Sreenivas Subramoney
Processor Architecture Research Lab
Intel Labs
Bengaluru, India
sreenivas.subramoney@intel.com

***Abstract*—Value Prediction was proposed to speculatively break true data dependencies, thereby allowing Out of Order (OOO) processors to achieve higher instruction level parallelism (ILP) and gain performance. State-of-the-art value predictors try to maximize the number of instructions that can be value predicted, with the belief that a higher coverage will unlock more ILP and increase performance. Unfortunately, this comes at increased complexity with implementations that require multiple different types of value predictors working in tandem, incurring substantial area and power cost.**

In this paper we motivate towards lower coverage, but focused, value prediction. Instead of aggressively increasing the coverage of value prediction, at the cost of higher area and power, we motivate refocusing value prediction as a mechanism to achieve an early execution of instructions that frequently create performance bottlenecks in the OOO processor. Since we do not aim for high coverage, our implementation is light-weight, needing just 1.2 KB of storage. Simulation results on 60 diverse workloads show that we deliver 3.3% performance gain over a baseline similar to the Intel Skylake processor. This performance gain increases substantially to 8.6% when we simulate a futuristic up-scaled version of Skylake. In contrast, for the same storage, state-of-the-art value predictors deliver a much lower speedup of 1.7% and 4.7% respectively. Notably, our proposal is similar to these predictors in performance, even when they are given nearly eight times the storage and have 60% more prediction coverage than our solution.

I. Introduction

Modern Out-of-Order (OOO) processors rely on large instruction windows to extract instruction level parallelism (ILP). However, inherent data dependencies limit ILP. Value prediction [16] was proposed as a mechanism to break these data dependencies and unlock ILP. By predicting the value of a producer instruction, the consumer can speculatively execute even before the producer has executed. If the prediction was incorrect, a pipeline flush is typically needed [20].

Early value predictors were based on last value [16], stride values [13] or program context [23]. State-of-the-art value predictors like VTAGE [20], DVTAGE [21] and EVES [25] employ a combination of last value, stride and context predictors using multiple tagged prediction tables. The DLVP predictor [27] used address prediction to fetch load values in advance from the data cache and used them for value prediction. A recent work [28] combined DLVP and EVES into a Composite Predictor to further increase the coverage of value prediction.

As is evident, the current approaches towards value prediction are motivated solely towards maximizing the number of instructions that are value predicted, with the belief that higher coverage will unlock more ILP and improve performance. This approach, however, comes at a substantial cost as the predictors become large and complex to implement. Most state-of-the-art value predictors assume a large, 8-16 KB predictor table that adds significant complexity and power. Additionally, value prediction requires writing the predicted value to the register files and validating the prediction later for correctness [25], [27]. A higher coverage of value prediction hence adds significant pressure to the highly power and area intensive OOO pipeline structures. The high area cost coupled with the additional power cost of validating the value prediction, makes it extremely challenging to arrive at a practical solution for value prediction.

The above problems motivate the investigation of light-weight value predictors that focus only on a small set of instructions that matter the most for performance. To better understand this problem, we performed a detailed analysis of current value prediction techniques over a wide variety of workloads, and made the following observations.

1) We observed that even though high coverage value prediction can successfully break many chains of data-dependent instructions, accelerating a majority of such dependence chains does not help performance, since their execution latency is already hidden by the large instruction window of modern OOO processors. However, modern processors still suffer from frequent bottlenecks created by the execution of instructions like delinquent loads that miss the last level cache (LLC) or branches that are wrongly speculated. An early execution of such

**Concepts, techniques and implementations presented in this paper are subject matter of pending patent applications, which have been filed by Intel Corporation*

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00018

instructions can help mitigate, though not eliminate, the processor stalls, and yield significant performance improvement. This early execution can be accomplished if the data dependencies to these *critical* instructions can be successfully value predicted. Targeting only this subset of instructions, that are on the data-dependence chain of the critical instruction, forms the basis of our proposal Focused Value Prediction.

2) We also observed that value prediction has traditionally targeted only data-dependencies arising through registers. However data dependencies also emerge from memory because of stores and loads to the same memory address. For instance, the address generation of a delinquent load that misses the LLC, may depend on an earlier store, and value prediction will not target this memory dependency. However, if the store to load forwarding can be accurately predicted for this load, then there is no need to predict any other register dependency that may be needed to calculate the address of the load. Predicting such memory dependencies can hence significantly reduce the number of register dependencies that need to be predicted. Moreover, many load instructions do not always access constant data, and hence value predictors rely on large, expensive tables based on program context to predict them. We observed that a big fraction of such loads are actually forwarded by prior stores, necessitating storing and predicting different data for each program context. By incorporating memory dependence learning, our proposal eliminates the need of using context value prediction for such store forwarded loads, and thereby reduces the size of context tables that are needed for value prediction. A large body of research already exists for accurately predicting memory dependencies [10], [32], and we can leverage those works in our value predictor.

Based on the previous observations, we propose Focused Value Prediction (FVP), a light-weight and practical solution to the problem of value prediction. FVP learns instructions that frequently create performance bottlenecks in the OOO processor, and leverages value prediction of register and memory dependencies to accomplish an early execution of such *critical* instructions. By focusing on a small subset of register or memory dependencies, FVP significantly reduces the area, power and design complexity of implementing value prediction, while still delivering large performance gains. Detailed simulation results on 60 diverse, single threaded workloads, show that we value predict about 25% of all load instructions and deliver 3.3% performance gain over a baseline similar to the Intel Skylake processor. This performance grows substantially to 8.6% when we simulate a futuristic processor that has double the OOO resources of Skylake. In contrast, the state-of-the-art Composite value predictor proposed by Sheikh et al. [28], with eight times the storage of our predictor and covering 39% of all load instructions, delivers 3.9% and 8.7% performance improvement, respectively. More importantly, for similar storage as our proposal, the speedup of this predictor drops to 1.7% and 4.7%, over the two respective baselines.

II. Background and Motivation

Value predictors are based on the concept of value locality [16], [20], according to which a given dynamic instruction is likely to produce a result that it had produced in the past. By predicting the value of a producer instruction, the consumer can execute speculatively even before the producer has executed. Wrong prediction causes pipeline flushes, and hence a high confidence is established before applying value prediction.

Several kinds of value predictors have been proposed in the past. The Last Value Predictor (LVP) [16], predicts the result of an instruction to be the same as what was produced in the past. The Stride Value Predictor [13] learns the stride between values and predicts future instructions by adding the stride to the current output. Context value predictors use program state in addition to an instruction's Program Counter (PC) to provide better predictions. For example, the FCM [23] predictor identifies patterns in the output of a given static instruction and makes a prediction when the pattern repeats. The VTAGE predictor [20] relies on global branch history to provide more accurate predictions. The D-VTAGE [21] predictor incorporates stride prediction into VTAGE and the EVES predictor [25] uses smart allocation schemes to optimize the coverage and area of D-VTAGE. The DLVP predictor [27] uses context-based address prediction to fetch load values in advance from the data cache. These values are then used for value prediction of corresponding load instructions. The Composite predictor [28] combines EVES and DLVP to further enhance the coverage of value prediction.

As we will show shortly, memory dependences are very important in the context of value prediction. Prior works like [17], [26] and [22] have studied store-load associations and proposed techniques to bypass memory speculatively and improve performance. Tyson et.al. [32] had proposed Memory Renaming (MR) which learns memory dependence between store-load PC pairs. For a confident store-load association, it allows the consumers of the load to take data directly from the store. Essentially, MR acts as a value predictor, where a store-load association is used to predict the load's data.

Current state-of-the-art value predictors inherently require large amount of storage for recording multiple PC and context combinations. This makes it very costly to increase the coverage of these predictors. There have been multiple proposals which attempt to reduce the storage of value predictors. Calder et al. [8] optimized the value prediction table management by prioritizing the allocation of all instructions belonging to the longest data dependence path. The EVES predictor [25] introduces smart data management by giving preference to instructions based on their latency. The Composite predictor [28], which combines DLVP and EVES, uses multiple policies for optimizing table management while increasing the coverage. These approaches are mainly focused towards better table utilization and therefore have limited benefits. In contrast,

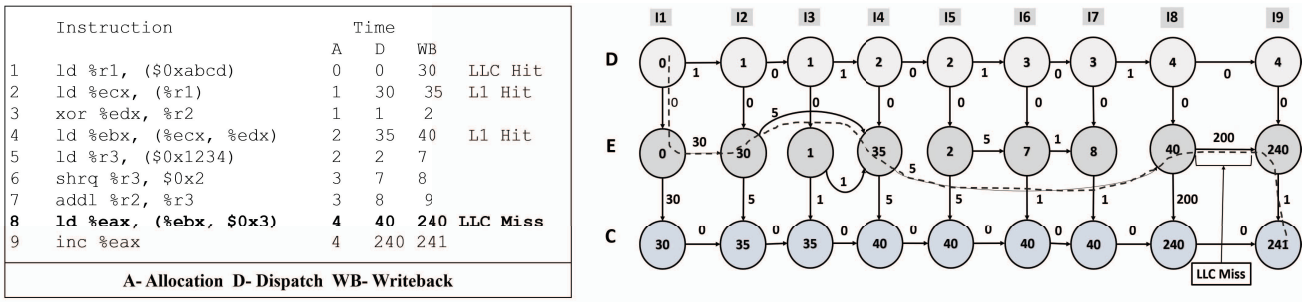

Fig. 1. (a) shows the example program and (b) shows it represented as a DDG

our approach tackles this issue fundamentally by identifying and value predicting only the minimal set of instructions that are needed to alleviate processor bottlenecks, resulting in significant area savings.

A. Implementation Challenges

Value predictors are typically implemented in the Frontend of the pipeline [25], [27], [28]. They are looked up for all instructions and hence need significant area for large tables. For example, the EVES [25] predictor uses a large 8 KB storage split into multiple different logical tables and each instruction needs to perform simultaneous lookups to each of these tables.

Another issue with value prediction is its interaction with the OOO pipeline. When an instruction is predicted, the predicted value is written into the register file for use by its consumers. After the instruction executes, the predicted value is read from the register file to be validated against the actual value. As more instructions are predicted, the number of read and write operations increase proportionately. This adds significant power to the register-file which is inherently a very power intensive structure [33].

To simplify value prediction implementation, EOLE architecture [19] proposed implementation of value prediction without adding extra read ports to the register file for validation of prediction. It accomplished this by adding an extra pipeline stage called the Late execution/Validation and training phase, and used this pipeline stage for validating the prediction. However, EOLE still relies on a large value predictor, which needs substantial storage and incurs power costs for validating the high number of predictions that it performs. Our solution, FVP, can be built on top of architectures like EOLE to further simplify the implementation of value prediction.

The discussions above show that the current approach towards high coverage value prediction is highly area and power intensive. This motivates the need to investigate high performance but light-weight predictors that focus only on a small set of instructions that matter for performance, instead of trying to maximize overall coverage. Such predictors will have low storage requirements and reduced overheads on power hungry and timing critical OOO pipeline resources.

B. Program Criticality

The performance of an OOO processor is bound by the critical path, and instructions whose execution lies on the critical path are called critical instructions. Clearly, one of the ways to reduce the overheads of value prediction is to predict only the instructions that lie on the critical path. Critical instructions are typically a small fraction of the overall dynamic instructions [12], [18]. Several works have hence described heuristics that can be used to enumerate critical instructions [12], [29]–[31]. A recent work proposed a graph buffering approach [18] to detect the critical path. Once critical path is detected, value prediction may be selectively applied to these instructions that lie on the critical path.

Unfortunately, accurately learning the entire critical path is a very challenging and expensive task [12], [18]. More importantly, as we will reason in Section III, not all instructions that lie on the critical path are important for value prediction, rendering the approach of enumerating the critical path unwieldy and sub-optimal. Our proposal is unique as unlike criticality based proposals like the one by Tunes et al. [30], [31], it does not predict all instructions on the critical path. Rather our proposal identifies dynamically a minimum set of instructions that can effectively *cut away* the root of the critical path and reduce processor bottlenecks. By early execution of instructions that create processor stalls, we show that our policy yields substantial performance gains, at low costs. Finding such instructions in an efficient manner and designing a high performance, light-weight predictor for them, forms the motivation for our proposal. In Section VI, we will contrast quantitatively our proposal with more complex solutions that warrant an accurate detection of critical paths.

III. Value Prediction and Early Execution

Instead of the traditional approach of applying value prediction to increase ILP, we envision value prediction as a mechanism to perform an early execution of instructions that may create bottlenecks in the processor. Typically, such instructions are long latency loads missing in the LLC or branch mis-predictions, but they can also sometimes be part of a long dependence chain, whose execution cannot be hidden by the OOO instruction window. In this section, we examine such potential target instructions for value prediction.

Consider the example in Figure 1. The load instruction $I8$ is an LLC miss, which takes 200 cycles to execute. This execution latency will typically not be hidden by the instruction window of the OOO processor. Therefore, instruction $I8$ will stall the processor. We should note that when $I8$ is allocated at

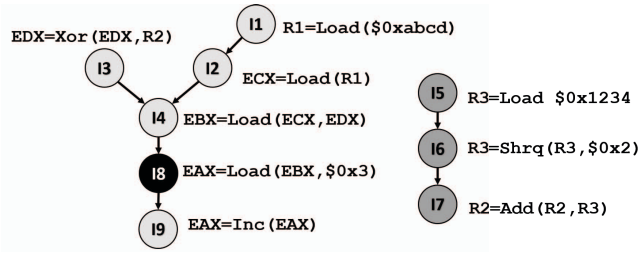

Fig. 2. Dependence chains for Figure 1

cycle $t = 4$, it cannot execute since it is waiting for its address to be generated through the execution of instructions $I1$, $I2$ and $I4$. Once $I4$ is executed at cycle $t = 35$, $I8$ can execute. That is why the critical path also contains the dependence chain which needs to be executed before $I8$ can be launched.

We have a re-look at this example through a Data Dependency Graph (DDG) [12]. We use the notation used by Fields et al. [12] to create the DDG. The path through the graph with the maximum weighted length is called the critical path and any instruction whose execution (E node of the DDG) appears on this path is critical. Figure 1 shows the critical path using a dotted line. As we see, the critical path is constituted of instruction $I8$, its dependence chain comprising of $I1$, $I2$, $I4$ and its forward dependent $I9$. The total critical path latency is hence $30(I1)+5(I2)+5(I4)+200(I8)+1(I9) = 241cycles$.

We now look at how value prediction can shorten this critical path. Figure 2 shows the example program split into two data dependence chains. As is evident, the dependence chain on the left is critical as it leads to the long latency Load $I8$ whereas the other chain comprising $I5$, $I6$ and $I7$ can be ignored. Predicting the load miss $I8$ only removes $I9$ from the critical path, and hence saves just 1 cycle from the critical path. Note that value prediction does not eliminate the need to execute the load $I8$. We still need to execute it to validate the value prediction, which implies the machine will still suffer the stall from $I8$.

Value predicting instruction $I1$, which is a load served from the last level cache (hence a high latency of 30 cycles), will allow issuing the load $I8$ earlier and the critical path latency would reduce to $1(I1)$(D-D)$+5(I4) + 5(I2) + 200(I8) + 1(I9) = 212$ cycles. This is a performance gain of 13%. However if we could predict the L1 hit $I4$, the critical path latency would have dropped to 205 cycles, giving us a speedup of 24%. Even if we value predict all instructions, the execution time would only be 204 cycles. Clearly, value predicting $I4$ is most important for performance. In fact, if we predict $I4$, there is no need to predict any of the other instructions, since the critical path has already been substantially shortened. Essentially, only 1 instruction out of the 9 instructions in the example, needs to be predicted.

The above example clearly demonstrates that in the presence of value prediction, not all instructions on the baseline critical path matter equally for performance. Instead of naively targeting all critical instructions, value prediction should predict only the instructions closest to the root instruction in the critical path, that creates the processor bottleneck. Hence the focus of our work is on finding out and value predicting the minimum set of instructions that directly impact performance.

A. Leveraging Memory Dependencies

Data dependence can also originate from memory. If a store writes data to an address that is fetched by a future load, the store and load have a memory dependence. We now analyze this property in the context of value prediction.

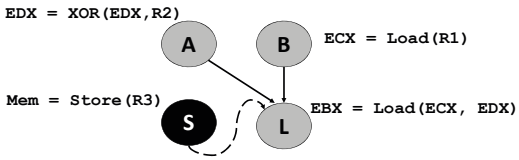

Fig. 3. Example of a Memory Dependence

Figure 3 shows a dependence chain with a load L which we want to predict. Its address is computed using outputs of A and B and its data is produced by Store S. Therefore, L has a memory dependence with the Store S. Suppose that L is not value predictable. We can predict either A and/or B. Though predicting A and/or B will compute the address of L faster, its data will only be available after S completes. Therefore, predicting these instructions may not offer any speedup if S is yet to execute. However, once the data of S is available, it can be used as a prediction for L if their addresses are expected to be the same. Essentially, we make a prediction on the address correlation between S and L and use that to value predict L. We can also extend this scheme further to predict the dependent chain of S and avail its data faster. Moreover, since we have effectively predicted L, we have no need to predict A or B, which means less instructions are predicted. Therefore leveraging memory dependencies can have significant benefits for performance as well as storage.

We apply these learnings in our work, Focused Value Prediction, which is discussed next in Section IV.

IV. Focused Value Prediction

Focused Value Prediction (FVP) first identifies the root of the critical path in a given instruction window and then picks the minimum set of instructions closest to this root which are predictable. We now discuss the sequence of steps to implement this in the OOO pipeline.

A. Identifying the Root of the Critical Path

OOO processors use deep instruction windows to hide the latency of a series of dependent instructions. For example, the Skylake processors from Intel have a reorder buffer (ROB) depth of 224 outstanding instructions [11] which is expected to grow in future processors. Despite such deep instruction windows, if the cumulative latency of a series of dependent instructions (referred to as a dependence chain) cannot be hidden by the processor's depth, it will produce stalls and hurt performance. The most common case when this happens

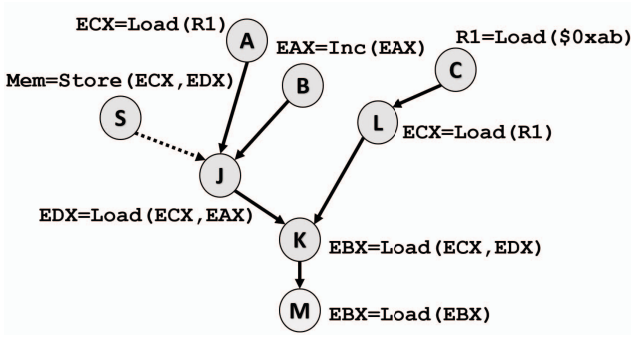

Fig. 4. Dependence chain of long latency load M for focused training

is if there is at least one long latency load (LLC miss) in this chain of dependent instructions.

A well known heuristic to identify such scenarios is the retirement stall heuristic [14]. This heuristic defines a critical instruction as an instruction that stalls the retirement of the OOO processor. This occurs when the OOO depth cannot hide the combined latency of the instruction and the dependence chain leading upto it. In such a case, the stalling instruction and the dependence chain leading upto it forms the critical path and the stalling instruction is the root of the critical path.

Value predicting the stalling instruction will not remove the retirement bottleneck because value prediction does not alter the execution of an instruction. Instead, if we predict the instructions that create the sources of the stalling instruction, we can dispatch this instruction earlier and potentially remove the bottleneck. This minimum set of value predictable instructions forms the target of our proposal. We should note that there may be additional instructions dependent on this root critical instruction, that may also stall future retirement. Therefore, value predicting the root of the critical path may also be beneficial.

1) Critical Instruction Table: Critical instructions are recorded in a 32-entry, Direct-mapped table called the Critical Instruction Table (CIT) which resides in the Front-End of the processor. Each entry holds a 11 bit Tag, 2 bit confidence and 2 bit utility. When an instruction executes, we check its distance from the ROB retirement pointer (the oldest entry in ROB). If the distance from the retirement pointer is smaller than the commit width of the machine, the instruction may stall retirement, and is likely to be the root of a potential critical path. Hence, we record such an instruction in the CIT. Every time this happens the confidence and utility are incremented by 1 until saturation. When the confidence saturates, the instruction is considered a critical root and becomes the target of acceleration for FVP.

When a new instruction is unable to occupy a CIT entry because of another resident instruction, the utility of that entry is decreased by 1. The resident instruction gets evicted and replaced when the utility becomes 0. To adjust to phase changes in the program, all CIT entries are reset completely after a fixed interval, called Criticality Epoch. Through simulations we found a good value of the Criticality Epoch to be 400,000 instruction retirements.

2) The case of Bad Speculation: Control mis-speculations frequently lie on the critical path, and should also be targeted for value prediction. However, since modern branch predictors use global branch history for prediction, a frequently mispredicting branch signifies that the output of the dependence chain of the branch has poor correlation with the global branch history. Current state-of-the-art value predictors (including FVP) also use similar branch history information for value prediction. This makes them ill-equipped to predict instructions on the dependence chain of such mis-predicting branches. Hence we ignore branch mis-speculations and focus on other critical instructions more amenable to value prediction.

B. Focused Training

Figure 4 shows a dependence chain in which M, a long latency memory access, is the root of the critical path. From Section III, we know that predicting K is highly rewarding since it dispatches M earlier. However, predicting M can also provide some speedup by accelerating downstream dependents of M. Hence, we try to predict K as well as M.

Identifying Dependencies: We first describe how to find the parent sources of an instruction i.e. the instructions that produce a source of a given instruction. We augment the Rename Alias Table (RAT) to also record the PC of the instruction that last wrote to a given architectural register. An instruction allocating in the OOO updates its PC in the RAT entry corresponding to its destination architectural register. This information is also read in parallel with the RAT during renaming allowing the pipeline timing to be unaffected.

In the example in Figure 4, when M allocates into the OOO, the RAT provides the PC(s) of its parent source i.e. K. The PC(s) of the sources are added to a small, 2-entry buffer called the Learning Table (LT). When K executes, it hits in the LT and is added to the value predictor and the LT entry is released. In case the parent source has executed already, the next instance will hit the LT, update the predictor and release the LT entry. Having the LT avoids extra write ports in the value predictor by delaying the update of parent source PC(s) till execution.

Suppose if instruction K is *not predictable* (as will be described in Section IV-C), we then look for its parent sources, which are J and L in our example. We add J and L to the value predictor for training. If J and L are highly predictable, we need not predict any more instructions on this chain.

Supposing load J is *not predictable*, we look for its parent sources i.e A and B. Interestingly, J also has a memory dependence with the older store instruction S (shown by the hashed arrow). As reasoned in Section III-A, store S should be prioritized over A or B for prediction. Therefore, if we determine a constant memory dependence between J and S, we can predict J's value using S. This technique, called Memory Renaming (MR) [32], is explained further in Section IV-D. We can now, if needed, also accelerate the dependence chain of the store S. If no memory dependence is found for J, we look at the parent sources i.e. A and B.

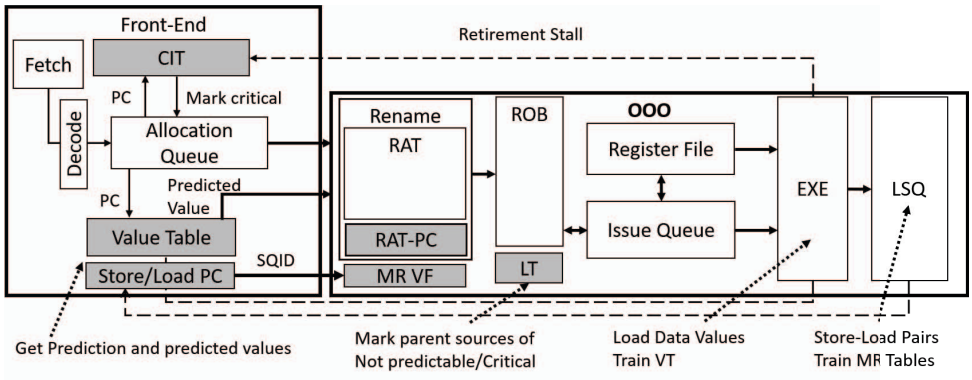

Fig. 5. Implementation of Focused Value Prediction in an OOO processor pipeline

Likewise, if L is found *not predictable*, its parent source C is added to the value predictor for training. This repeats until we find a highly predictable instruction on this chain too.

Load vs All Instruction prediction: Our studies showed little benefit of predicting non-loads on top of predicting load instructions (shown in Section VI-A2). Hence, we modify FVP to perform only load value prediction and discard all non-load instructions during training. While allocating in the value predictor, we check an instruction's type. If it is not a load, it is marked as *not predictable*. This triggers a search for its parent sources which undergo the same test. This process repeats until a load is found. Eventually, all non-load instructions are evicted (overwritten) without getting predicted. Similarly, during learning of the CIT table, we only learn for loads that are stalling retirement.

In essence, the FVP algorithm picks the minimum number of value predictable loads which can shorten the critical path of the program in a hardware-friendly implementation. We now discuss the predictor implementation in Section IV-C.

C. Predicting Dependencies through Registers

We use a hybrid value predictor which performs Last value (LV) and Context based value (CV) prediction. In Last value prediction, if the value remains constant for a given PC, the same is predicted for future occurrences. For Context value prediction, if the value remains constant for a given PC and branch history (context), the same is predicted when the PC and branch history repeat. For our work, the branch history is the outcome of the last 32 branches in the program.

The value predictor consists of a 48-entry, 2-way set-associative table, called the Value Table (VT) which holds all data needed for prediction. The VT is updated on instruction execution. Since the VT is updated on instruction execution, speculative instructions may also update the VT. Each VT entry holds a 11-bit tag, 64-bit data, 3-bit confidence, 2-bit no-predict, and 2-bit utility counters. The confidence is incremented when the data repeats with a probability of 1/16 and utility is also incremented. If the data changes, the confidence and utility are reset. Once the confidence saturates, FVP can start predicting the instruction. This provides FVP an accuracy above 99%. This is on par with state-of-the-art predictors like VTAGE [20], EVES [25] and Composite predictor [28].

The 2-bit no-predict counter tracks the overall accuracy of an entry and signals if this entry is *not predictable*. It is initialized to 0 on a new allocation. It is incremented by 1 on a mis-prediction (data change) and reset to 0 when the confidence saturates. If the no-predict counter saturates, it essentially denotes a highly fluctuating data and we term the entry as *not predictable*. Also, as described in Section IV-B, for load value prediction, all non-load instructions are allocated to the VT with the no-predict counter initialized to maximum. This allows only loads to be predicted.

A VT entry can be replaced when its utility becomes 0. Generally, entries which are *not predictable* have low utility and hence, are likely to get evicted faster.

Interestingly, a single Value Table supports both Last Value and Context value prediction. This is accomplished through the lookup function. If only PC is used for searching the Value Table, the data pertains to Last value prediction. If the PC as well as branch history are used for lookup, we are doing Context value prediction. All instructions are first predicted using Last value prediction. If they are *not predictable* by this, they are marked for context value prediction to be re-recorded in the Value Table post execution using the PC and branch history. For such instructions, we only re-record those instances whose distance from the oldest entry in the ROB is smaller than the commit width of the processor when they execute. This reduces the number of branch histories that we track thereby reducing the size of the Value Table.

The branch history for a load can be obtained from the history of the last allocated branch (before it). Branches typically record their history with them for repair on a mis-prediction and we can use this to deliver the history to a load.

Each cycle, we predict upto 2 loads as our baseline can execute upto two loads per cycle (refer Section V). Every load requires two lookups in the Value Table (Last value and Context value). Hence, the VT requires a total of 4 read ports. Also, since the processor has 2 Load Execution Units, the Value Table supports 2 write ports for updates. We will study the sensitivity to the Value Table size in Section VI-D.

D. Predicting dependencies through memory

Tyson et.al [32] proposed Memory Renaming (MR) which learns if a store instruction forwards data to a load instruction repeatedly. MR further says that, when this happens, the load data can be predicted using the store's output. We implement MR as described by Tyson et al. [32] which we briefly elaborate hereafter. The MR taps the Load-Store Queue (LSQ) forwarding network to identify store-load PC pairs where store forwards data to the load. The PC pairs are stored in a Store/Load PC cache in the Front-End as shown in Figure 5. If a store allocating in the OOO has acquired sufficient confidence in this cache, it records its Store-Queue ID (SQID) in a structure called Value File (VF). The VF records SQIDs of associated stores for each of the loads in the cache. The consumers of the load use this SQID to retrieve data once the store finishes and execute speculatively before the load. If the store-load association was wrong, a pipeline flush occurs [32]. Similar to the Value Table, the MR structures have 2 read/write ports for loads and 1 more read/write port for store since we have 1 store execution Unit (Section V).

As is evident from the above description, MR is effectively a value predictor which uses the associated store to predict the load even before the address of the load is computed. When a load is *not predictable* by Last value, it is added to Context value prediction as well as the MR. While allocating to the OOO, a load always checks the MR first, before checking the Value Table with Context. Moreover, a memory renamed load does not train the Value Table. This ensures a higher priority to MR compared to context value prediction. Lastly, if the load is *not predictable* by either Context value predictor or MR, we look for its parent sources as depicted in Figure 4.

E. Value Prediction Pipeline

Figure 5 illustrates all the components of FVP and their interactions in a modern OOO pipeline. The Value Table/MR reside in the Front-end. They are looked up after an instruction is decoded. As mentioned in Section IV-D, loads preemptively lookup the MR before the Value Table, whereas stores only access the MR. Either the Value Table or MR will signal when a decoded instruction is predictable. The Value Table provides a predicted value whereas the MR provides the SQID of the associated store to be used for value prediction in the OOO.

There are several methods to implement value prediction in the OOO. Prior predictors, like LVP and EVES [25] assumed additional write ports in the register file using which the predicted value is written in the registers to be used by the load's consumers. The DLVP predictor by Sheikh et al. [27] proposed a separate, small register file where the predicted values are written. Additional logic was introduced which selected the appropriate register file for the consumers. Our proposal, FVP is agnostic of the underlying OOO micro-architecture for value prediction and works well with either of them. Moreover, since FVP predicts a much lower fraction of instructions compared to previous works, the number of register file read and write operations is much smaller giving FVP higher energy efficiency compared to other value predictors.

Structure	Field(bits)	Storage
Critical Instruction Table (32 entries)	Tag (11b), Confidence (2b), Utility (2b)	60
Value Table (48 entries)	Tag (11b), Confidence (3b), Utility (2b), Data (64b), No-Predict (2b)	492
MR Store/Load Table (136 entries)	Tag (11b), Confidence (3b), LRU(2b)	272
MR VF (40 entries)	Data(64b), Store ID(6b)	350
RAT-PC (16 entries)	PC(11b)	22

TABLE I
STORAGE REQUIREMENTS FOR FVP

F. Storage Calculations

Table I lists the storage requirement for each component of FVP (including CIT). As we see, FVP adds only about 1.2 KB of additional area, most of which comes from storing the predicted data. Optimizations, such as described by Seznec et al. [25], to use a common data storage, can reduce this area even more, but we do not explore them in this work.

V. EVALUATION METHODOLOGY

We simulate a dynamically scheduled x86 core using a cycle-accurate simulator that accurately models a 4-wide OOO core clocked at 3.2 GHz. The core parameters are similar to the Intel Skylake processor [11]. The main parameters are summarized in Table II. We also simulate a future scaled up version of Skylake, which is 8-wide and where all the execution resources and bandwidths are doubled, compared to Skylake. We call this as the Skylake-2X baseline. We will show results on both Skylake and Skylake-2X baselines. In all our studies, the penalty of a value prediction is 20 cycles.

Front End	4 wide fetch and decode , TAGE/ITTAGE branch predictors [24], 20 cycles mis-prediction penalty, 64KB, 8-way L1 instruction cache, 4 wide rename into OOO with macro and micro fusion
Execution	224 ROB entries, 64 Load Queue entries, 60 Store Queue entries and 97 Issue Queue entries. 8 Execution units (ports) including 2 load ports, 3 store address ports (2 shared with load ports), 1 store-data port, 4 ALU ports, 3 FP/AVX ports, 2 branch ports. 8 wide retire and full support for bypass. Aggressive memory disambiguation predictor. Out of order load scheduling to L1
Caches	32 KB, 8-way L1 data caches with latency of 5 cycles, 256 KB 16-way L2 cache (private) with a round-trip latency of 15 cycles. 8 MB, 16 way shared LLC with data round-trip latency of 40 cycles. Aggressive multi-stream prefetching into the L2 and LLC. PC based stride prefetcher at L1
Memory	Two DDR4-2133 channels, two ranks per channel, eight banks per rank, and a data bus width per channel of 64 bits. 2 KB row buffer per bank with 15-15-15-39 (tCAS-tRCD-tRP-tRAS) timing parameters

TABLE II
CORE PARAMETERS FOR SIMULATION

We selected 60 diverse, single-threaded applications from various category of workloads, that are summarized in Table III. All benchmarks are compiled for the x86 ISA and

carefully selected based on representation. The performance is measured in Instructions Per Cycle (IPC).

Benchmarks	Category
perlbench, bzip2, gcc, mcf, h264ref, gobmk,hmmer, sjeng, libquantum, omnetpp, astar, xalancbmk	SPEC INT 2006 (ISPEC06)
bwaves, gamess, milc, zeusmp, soplex, povray, calculix, gemsfdtd, tonto, wrf, sphinx3 gromacs, cactusADM, leslie3D, namd, deall	SPEC FP 2006 (FSPEC06)
nab, cam4, pop2, roms, leela, cactubssn, xz, gcc, mcf, xalanc, exchange2, omnetpp, perlbench, bwaves, lbm, fotonik3d	SPEC17
lammps [4], hplinpack [3], tpce, spark, cassandra [1], specjbb [5], specjenterprise, hadoop [2], specpower [6]	Server

TABLE III
APPLICATIONS USED IN THIS STUDY

VI. Simulation Results

We first present the performance of FVP on our workloads in Section VI-A. We compare our proposal to state-of-the-art value predictors in Section VI-B and contrast with various criticality based heuristics in VI-C. Finally, we perform sensitivity analysis in Section VI-D and conclude with a qualitative power analysis in Section VI-F.

A. Summary of FVP Results

Figure 6 summarizes the performance delivered by FVP on different category of workloads for the Skylake baseline. We also show the coverage of FVP which is defined as the fraction of predicted loads out of total load instructions.

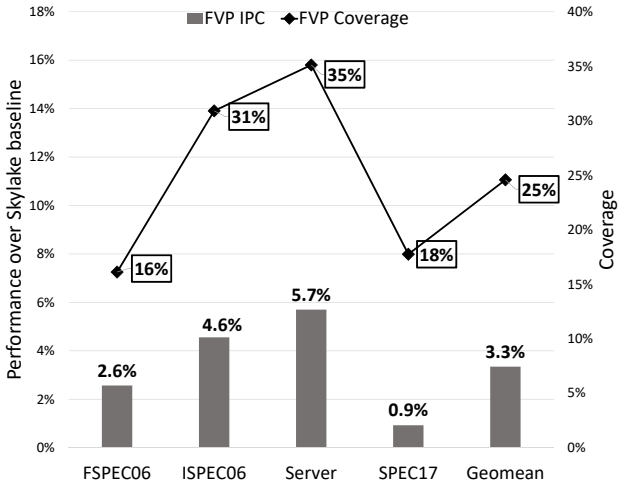

Fig. 6. Performance and Coverage of FVP on Skylake

From the results we see that FVP delivers 3.3% performance gain (geometric mean) over the Skylake baseline while predicting 25% of all dynamic loads (coverage). All categories of workloads, except SPEC17 workloads, show good sensitivity to performance improvement. We found that for many SPEC17 workloads the critical path is dominated by branch mispredictions. In Section IV-A2 we reasoned that value prediction cannot accelerate such mis-predicting branches since value prediction and branch prediction use similar branch histories for their predictions. Hence, the speedup in SPEC17 for FVP is relatively low.

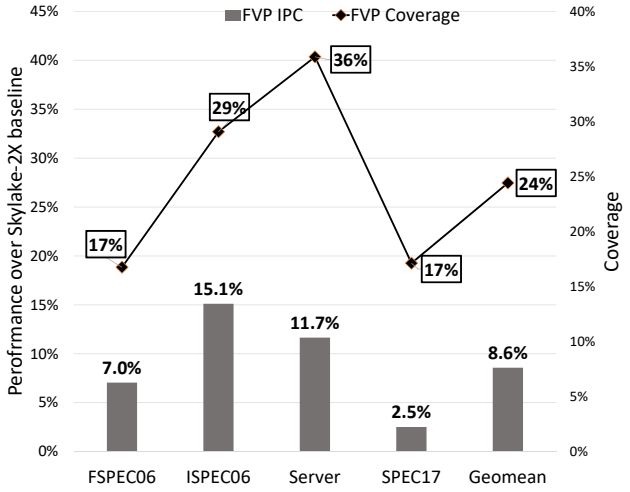

Fig. 7. Performance and Coverage of FVP on Skylake-2X

Figure 7 shows similar results for the Skylake-2X (future core) baseline. The performance gain of FVP on this scaled up core is significantly higher at 8.6% with almost the same coverage as Skylake. Since Skylake-2X has lot more resources and execution bandwidth, it shows better performance sensitivity than Skylake. All categories of workloads show similar trends, where FVP gains on Skylake-2X are higher than Skylake. Interestingly, the performance scaling of ISPEC06 category on Skylake-2X is better than the Server category. For Skylake baseline, Server category was the highest gainer, but for scaled up Skylake-2X baseline, ISPEC06 shows higher sensitivity. We found that even though execution resources have been scaled up, some of the server category workloads get limited by front-end bottlenecks in Skylake-2X configuration, primarily instruction cache misses (because of higher code footprint) and branch prediction bandwidth. Scaling the front-end of the processor should help further increase the sensitivity of the server category.

1) Coverage vs. IPC Gain: Figure 8 depicts a line graph for all workloads in our study list, showing the correlation between IPC gain and the value prediction coverage. As we see, applications like *namd*, *gobmk*, *cassandra* and *sphinx3* have low coverage but still gain significant performance, whereas applications like *mcf* and *gcc* do not gain performance even by predicting a large fraction of loads. The latter are workloads that suffer from a large number of cache misses. We should recall that value prediction cannot reduce the latency of a load but only dispatches its consumers faster. Specifically, these workloads suffer from limited memory resources like load-store queue, that are needed to serve the long latency load miss, and hence value prediction cannot help these workloads

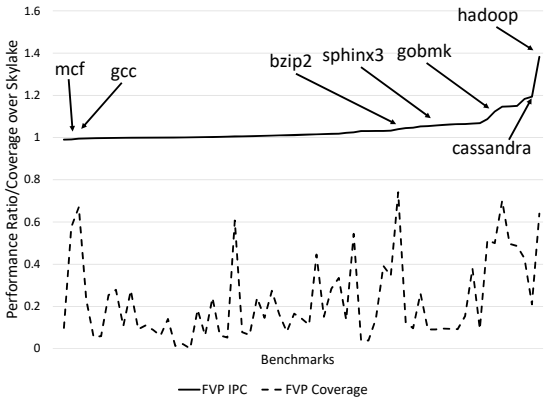

Fig. 8. IPC and Coverage of FVP for every workload on Skylake

unless other OOO resources are scaled.

Figure 9 contrasts the performance gain of each workload in our study list for Skylake and Skylake-2X baselines. Skylake-2X, being a wider core with more execution resources, shows much higher sensitivity of speedup from FVP. Applications like *gcc* that showed almost no performance sensitivity on Skylake, even with high coverage of value prediction, show significant speedup by applying FVP on the Skylake-2X baseline. Also some of the server category workloads, do not show much increase in performance sensitivity with FVP (marked in Figure 9). We found that these workloads had front-end bottlenecks (instruction cache misses and branch prediction bandwidth) on the Skylake-2X baseline. Some of these front-end bottlenecks would need to be alleviated to increase the performance sensitivity of these workloads.

In general, as processors scale in resources and execution bandwidth, they will be more exposed to bottlenecks created by memory accesses, bad speculation and true data dependencies. Performance features like FVP that can alleviate some of these bottlenecks will hence become increasingly more important for such processors and would be needed to increase the performance scaling of future OOO processors.

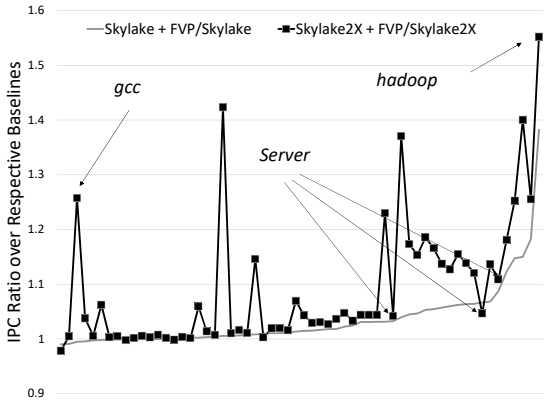

Fig. 9. Comparing FVP performance on Skylake and Skylake-2X baselines

2) Predicting all types of instructions: In FVP, we predict only load instructions. Our experiments did not show any significant speedup from predicting non-load instructions which is similar to the observations made in previous works [27], [28]. In fact, predicting all instructions degrades performance slightly because of increased conflict misses in the small FVP tables.

3) Value prediction for Branch Mis-prediction: We attempted to predict loads which lie on the dependence chain of a branch mis-prediction. However, this scheme only gives 0.5% more coverage and 0.05% more speedup over default FVP. This confirms the reasoning in Section IV-A2 that such loads cannot be predicted using current value predictors (including FVP) and hence, we ignore them.

B. Comparison with Prior Art

We compare FVP with state-of-the-art predictors namely Memory Renaming (MR) and the Composite predictor recently proposed by Sheikh et al. [28]. Memory Renaming is implemented using the mechanism described in Section IV-D but with a large Store/Load cache and Value File, resulting in an area of 8KB. Any store-load pair, irrespective of criticality, that has seen a data exchange in the LSQ can be Memory Renamed if the MR has attained confidence on the pair.

The Composite value predictor [28] has four-components, Last Value Predictor (LVP), Context Value Predictor (CVP), Stride Address Predictor (SAP) and Context Address Predictor (CAP). Essentially it is an intelligent fusion of EVES [25] and DLVP predictors [27], thereby outperforming both of them. We derive the individual value predictors from EVES [25] and add the address predictors (SAP, CAP) at instruction fetch, similar to the micro-architecture proposed by Sheikh et al. [28]. We corroborated the results of Sheikh et al. [28] with our simulations and saw that the Composite predictor outperforms EVES as well as the DLVP predictor. Hence, we do not compare with EVES and DLVP separately, since Composite predictor picks the best of the two predictors.

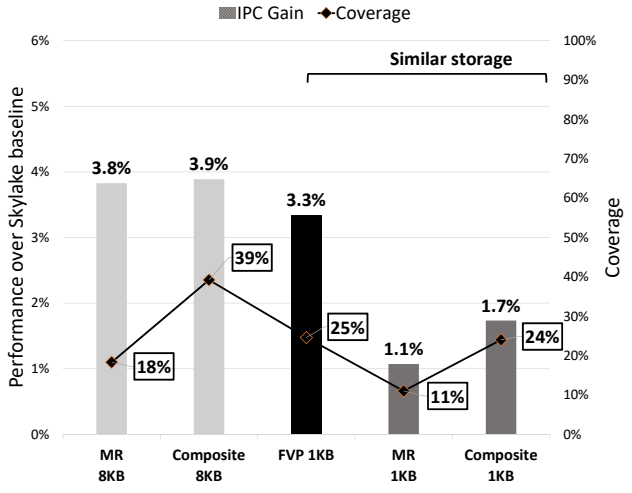

Fig. 10. Comparison between Memory Renaming(MR) by Tyson et.al [32], Composite Value Predictor by Sheikh et.al. [28] and FVP on Skylake

Figure 10 compares the speedup between FVP and the above mentioned predictors. We show results on Skylake baseline for large sized predictors (8KB) as well as a scaled down version of the predictors that have similar area (1KB) as

our proposal FVP. Figure 11 repeats the same results on the Skylake-2X baseline. For the Skylake baseline, MR gives 3.8% performance gain at 19% coverage whereas the composite predictor delivers 3.9% performance with 39% coverage. The gains and coverage for the composite predictor on Skylake are close to the gains reported by Sheikh et al. [28], which was 4.6% with about 40% coverage over a Skylake like baseline.

At nearly one-eighth of the storage, FVP comes close to the best performing DLVP-EVES Composite predictor. More importantly, at the same storage, FVP delivers nearly twice the performance of the Composite predictor (1.7 to 3.3%). These differences become even more significant when evaluated on the more sensitive Skylake-2X baseline as shown in Figure 11. Here again, with just one-eighth the area, FVP delivers similar performance as the Composite predictor and a very significant performance upside at the same area (8.6% vs 4.7%). As can be observed from Figure 10, FVP has significantly lower coverage (39% to 25%) than the Composite predictor. This reinforces the importance of targeted value prediction as compared to naively increasing the coverage. By smartly focusing on the root of the critical path of execution, FVP delivers higher performance at a significantly low storage making it very attractive for implementation in future OOO machines.

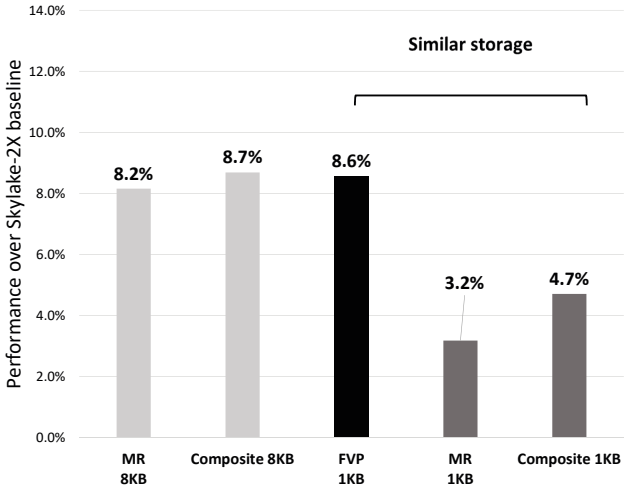

Fig. 11. Comparison between Memory Renaming(MR) by Tyson et.al [32], Composite Value Predictor by Sheikh et.al. [28] and FVP on Skylake-2X

We also experimented with combining the MR and Composite predictor and comparing it with FVP. However for small 1 KB tables, this causes significant thrashing and performs poorly. Also, in the predictor studies, we did not evaluate the Stride predictor [13]. Our simulations showed that the stride predictor gives a very small overall gain and helps only some workloads. Similar observations have also been reported by Sheikh et al. [28]. Nonetheless, it can be added on top of all the existing predictors, including FVP.

C. Quality of Criticality Detection

FVP shows that targeted application of value prediction can achieve high performance at very low area. In this section, we evaluate alternate criticality heuristics to achieve the same benefits.

One of the simplest heuristics could be to value predict only L1 cache misses since L1 cache misses usually have a long latency which may stall ROB retirement. Figure 12 shows results for this implementation which we call as *FVP-L1-Miss-Only*. This scheme shows negligible IPC improvement. As we had reasoned in Section III, value prediction does not reduce the latency of a load. Therefore, predicting only L1 misses will not be very beneficial. However, if we use value prediction to accelerate the address generation of such L1 cache misses, we can reduce the processor stalls by dispatching them earlier.

Figure 12 also shows such an implementation, labeled as *FVP-L1-Miss*, where instead of using the retirement stall heuristic, we simply assume that any L1 miss is a root of the critical path. As expected, *FVP-L1-Miss* achieves significant speedup over *FVP-L1-Miss-Only* because it predicts the dependence chain instructions and dispatches cache misses faster. Still, this performance is only 70% of the speedup compared to when the retirement stall heuristic is used for the training(*FVP*). The reason for this is that there are other instructions, apart from L1 misses, which also lie on the critical path of execution. For example, a long chain of dependent L1 hits could be on the critical path if the cumulative latency of the chain cannot be hidden by the OOO instruction window. Hence, just using L1 miss as a criticality criteria is insufficient.

Finally we compare our proposal against a very accurate *Oracle Criticality* detection of critical path shown in Figure 12. For this, we implemented and used the graph buffering approach described in [18]. A detailed data dependence graph [12] is created on-the-fly and is used to identify the critical instructions (loads) which are then predicted by FVP. This Oracle Criticality, if implemented in hardware, would need 3-6 KB of multi-ported storage, which is fairly impractical. Nevertheless it serves as an oracle upper bound to test the efficacy of our FVP algorithms.

This oracle policy has slightly higher performance than FVP but its coverage is lower. On deeper analysis, we see that some workloads like *dealii, perlbench and gamess* have lower speedup from the graph buffering approach because it marks all loads on the critical path for value prediction and thereby thrashes the small predictor tables. The thrashing, consequently, lowers the coverage for these benchmarks. In contrast, the superior criticality detection of *Oracle Criticality* works favorably in workloads like *omnetpp, zeus and gobmk* which see a higher speedup at a lower coverage. In summary, the overall result shows that FVP's approach is quite close to an oracular criticality detection approach in performance.

1) Sensitivity to Criticality Epoch: We also experiment with different values of the Criticality Epoch. A very small epoch lowers performance because the CIT gets less time to learn criticality before getting reset. Moreover, a very large epoch also reduces performance because the CIT entries are not evicted fast enough and become stale when the program phase changes.

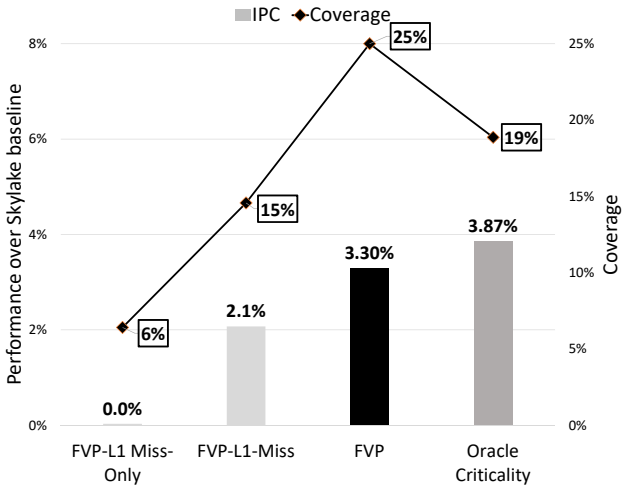

Fig. 12. Sensitivity to criticality criteria

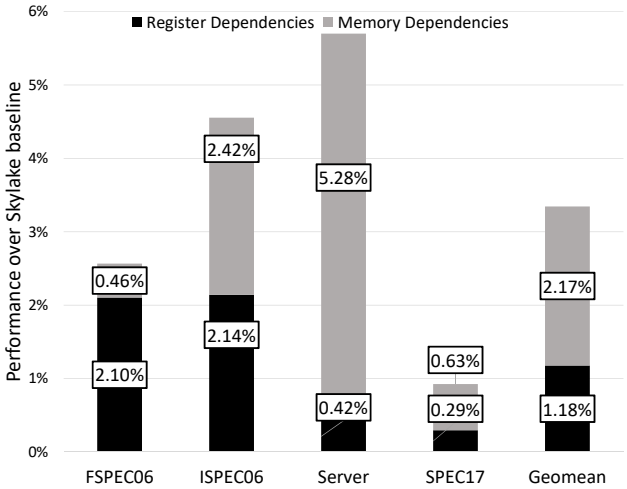

Fig. 13. Performance Contribution of Individual FVP components

D. Sensitivity to Table sizes

Our default FVP implementation uses a 48 entry Value Table and a 40 entry MR value file. Growing the structure sizes showed only a small improvement in speedup. In fact, when we grow the value table to 96 entries, and the MR value file to 128 entries, the performance increases by just 1%. Increasing table sizes beyond this has no visible improvement in performance, which is in line with our argument that targeting more instructions may not necessarily result in increased performance.

The CIT size has a low impact on IPC. The difference between 16-entry CIT and 8-entry CIT is just 0.15% higher speedup. Critical instructions keep getting copied from CIT into the VT/MR and hence the lifetime of an instruction in the CIT is much smaller compared to VT/MR. Hence, the effect of conflict misses is less evident in the CIT as compared to the VT/MR.

E. Contribution of Individual FVP Components

FVP uses value prediction to break register data dependencies as well as eliminate memory dependencies. Figure 13 breaks down the performance of FVP from these two components for the Skylake baseline. Server category benefits more from breaking memory dependencies whereas FSPEC06 favors value predicting register dependencies. ISPEC06 favors both of them equally.

Interestingly, most of the prior works on value prediction do not try to take advantage of breaking memory dependencies. We observed in our analysis that for many loads, the data changes because of older stores writing to the same address as the load. Such loads can be easily predicted using store-load associations. This is more area efficient compared to large context predictors which have to track multiple different paths to predict such a load. Our studies showed significant overlap between loads that are amenable for Context prediction and those that are predictable using store-load dependences. This implies that many scenarios where Context predictors work well can be covered by simple memory dependences with a much lower storage.

F. Qualitative Power Analysis

The FVP predictor, by virtue of being highly selective, is a very small predictor. It takes about 1.2KB of storage which is about one-eighth of the area compared to the other predictors evaluated (8KB). We should note that all instructions need to lookup this table during fetch to check if they can be value predicted or not. Hence a smaller predictor implies less power for lookup. Additionally, smaller area helps with static power.

Moreover, as seen in Figure 10, FVP predicts only 25% of all loads (6% of all instructions) as compared to 39% (9% of all instructions) by the Composite predictor. We should note that all predicted instructions need to write the predicted value in the register-file to be used by their dependents. We also need to validate the prediction. Consequently, FVP performs less register-file read-write operations owing to its small coverage which helps in reducing the dynamic power consumption.

In summary, because of targeted application, FVP has significantly lower power and area requirements compared to larger state-of-the-art value predictors.

VII. Other Related Work

We discussed the various value prediction proposals in Section II. Similar to value prediction, predictors have also been proposed for addresses [7], [15]. DLVP [27] used address prediction to fetch load values in advance from the data cache for value prediction. We performed a detailed quantitative comparison with the state-of-the-art techniques in Sections VI-B.

The learning of target instructions for focused value prediction, has some similarity with the IBDA approach [9] which finds out address generating instructions for a load or store. However unlike IBDA, FVP learns candidates only one at a time and stops as soon as it finds out a value prediction candidate, whereas IBDA tries to extract the full program

slice. Also IBDA does not learn for memory dependencies, whereas FVP also accounts for memory dependencies, along with register dependencies, to find out the minimum set of target instructions for value prediction.

Using program criticality to improve the performance of processors has been explored extensively in the past. The data dependence graph described in this work was described by Fields et al. [12]. Several other works have described heuristics that can be used to enumerate all instructions that lie on the critical path [12], [29]–[31]. Our approach towards program criticality is unique and very different from past proposals. In previous approaches, the goal was to enumerate all instructions on the critical path. However detecting and enumerating the full critical path is difficult and expensive in hardware. Our proposal does not try to find all critical instructions but rather tries to break the critical path closest to the root, using a simple detection mechanism in the OOO. We presented a detailed comparison of our policy with an accurate criticality detection mechanism in Section VI-C.

VIII. Summary

In this paper we have presented Focused Value Prediction (FVP), a fundamentally new approach towards value prediction. Unlike current approaches in value prediction which try to maximize the number of instructions being predicted, FVP focuses on a small subset of critical instructions, that are closest to the root of the program critical path and hence matter the most for performance. This allows FVP to have small hardware structures, making it area and power efficient. Our results show that FVP outperforms the state-of-the art value predictors, while needing just one-eighth of the area of those predictors. More importantly, we show that as the OOO machines scale in resources, FVP gains even higher performance, thereby confirming the robustness of FVP's implementation and its signficance as an important direction for improving the performance of future OOO processors.

References

[1] "Apache cassandra nosql performance benchmarks." [Online]. Available: https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks

[2] "Hadoop benchmarking." [Online]. Available: https://hadoop.apache.org/docs/r2.8.0/hadoop-project-dist/hadoop-common/Benchmarking.html

[3] "Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers." [Online]. Available: http://www.netlib.org/benchmark/hpl/

[4] "A quick tour of lammps." [Online]. Available: https://lammps.sandia.gov/workshops/Aug15/PDF/tutorial_Plimpton.pdf

[5] "Standard performance evaluation corporation." [Online]. Available: https://www.spec.org/jbb2015/

[6] "Standard performance evaluation corporation." [Online]. Available: https://www.spec.org/power_ssj2008/

[7] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 54–63. [Online]. Available: http://dx.doi.org/10.1145/300979.300984

[8] B. Calder, G. Reinman, and D. M. Tullsen, "Selective value prediction," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 64–74. [Online]. Available: http://dx.doi.org/10.1145/300979.300985

[9] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 272–284.

[10] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, July 1998, pp. 142–153.

[11] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, Mar 2017.

[12] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proceedings 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 74–85.

[13] F. Gabbay and F. Gabbay, "Speculative execution based on value prediction," EE Department TR 1080, Technion - Israel Institue of Technology, Tech. Rep., 1996.

[14] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 84–95. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485930

[15] J. González and A. González, "Speculative execution via address prediction and data prefetching," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 196–203. [Online]. Available: http://doi.acm.org/10.1145/263580.263631

[16] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII. New York, NY, USA: ACM, 1996, pp. 138–147. [Online]. Available: http://doi.acm.org/10.1145/237090.237173

[17] A. Moshovos and G. S. Sohi, "Speculative memory cloaking and bypassing," *Int. J. Parallel Program.*, vol. 27, no. 6, p. 427–456, Dec. 1999. [Online]. Available: https://doi.org/10.1023/A:1018776132598

[18] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 96–109.

[19] A. Perais and A. Seznec, "Eole: Paving the way for an effective implementation of value prediction," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 481–492.

[20] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 428–439.

[21] A. Perais and A. Seznec, "Bebop: A cost effective predictor infrastructure for superscalar value prediction," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 13–25.

[22] A. Perais and A. Seznec, "Cost effective physical register sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 694–706.

[23] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, Dec 1997, pp. 248–258.

[24] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-level Parallelism - JILP*, vol. 8, 01 2006.

[25] A. Seznec, "Exploring value prediction with the EVES predictor," in *CVP-1 2018 - 1st Championship Value Prediction*, Los Angeles, United States, Jun. 2018, pp. 1–6. [Online]. Available: https://hal.inria.fr/hal-01888864

[26] T. Sha, M. M. K. Martin, and A. Roth, "Nosq: Store-load communication without a store queue," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 285–296.

[27] R. Sheikh, H. W. Cain, and R. Damodaran, "Load value prediction via path-based address prediction: Avoiding mispredictions due to

conflicting stores," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 423–435. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123951

[28] R. Sheikh and D. Hower, "Efficient load value prediction using multiple predictors and filters," in *Proceedings of the 25th International Symposium on High-Performance Computer Architecture*, ser. HPCA '19, 02 2019.

[29] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 419–430.

[30] E. Tune, Dongning Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001, pp. 185–195.

[31] E. S. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2002, pp. 104–113.

[32] G. S. Tyson and T. M. Austin, "Memory renaming: Fast, early and accurate processing of memory communication," *Int. J. Parallel Program.*, vol. 27, no. 5, pp. 357–380, Oct. 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1018734923512

[33] L. Wehmeyer, M. K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan, "Analysis of the influence of register file size on energy consumption, code size, and execution time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, pp. 1329–1337, Nov 2001.