Final Project Report - 12/9/19
CS256

Team Members:
Kyle Dumont [kyd562@g.harvard.edu]
David Li [david_li@college.harvard.edu]

Project Title: Value Prediction on x86 Architecture

Summary

In this paper we will revisit the concept of value locality presented in *Exceeding the Dataflow Limit via Value Prediction [Mikko H. Lipasti et. all, 1996]*.

Data in registers often often exhibit Value Locality, which states that a previously-seen value is likely to appear again. This stems from the fact that in practice, many programs are general-by-design and hence, in a computation, may involve trivial computation (eg. when computing a 3d expression when only 2 dimensions are used and the third value is 0), or redundant computation. These cannot be easily optimized out in compile time, as even aggressively optimized code tends to exhibit value locality, but there is some promise to caching values generated by computations during runtime in hardware. The paper, *Exceeding the Dataflow Limit via Value Prediction* has demonstrated up to 80% Value Locality with history depth of 4 entries. By taking careful advantage of this phenomenon, computer cores can achieve higher throughput than non-speculative systems. Since the order of instruction execution is not necessarily bounded by data dependencies anymore, this can open the door to a much higher ILP. The primary purpose of this project is to demonstrate what value speculation accuracy can be expected for various computation type and entry length. We can also test the degree of value locality depending on different degrees of compiler optimizations, to see whether or not certain optimizations may increase or decrease value locality.

Instruction Locality

Instruction Type Classification

We write boolean expressions to classify each instruction into one of the following categories: [Int, Float]x[Pure Load, Arith, Reg Move, Load Arith] in addition to [Int Arithmetic with 1 or 2 register input arguments]

Due to the complexity of the x86 instruction set, it is impractical to manually separate every instruction into these categories. Hence, to determine the general type of an instruction, we made use of the `INS_Category` function included in the Intel PIN library, which given an instruction, maps it to one of over a hundred broad categories. For example, the `XED_CATEGORY_DATAXFER` category is used to indicate `mov`-type instructions.

To classify the datatype of each instruction we use the datatype of the output register, which was classified using the functions `REG_is_fr, REG_is_gr8...REG_is_gr64`, corresponding to a floating-point output register, and integer registers of different sizes, respectively.

For arithmetic instructions, we also classified them by the number of input operands, as in the Lipasi paper. This categorization can be determined by the `INS_MaxNumRRegs` function.

Below are the different categories we classify instructions by, and examples of instructions in those categories taken from our test cases.

Instruction Category	**Description**
I_PURE_LOAD	Integer Pure Load Instruction (e.g. `mov esi, dword ptr [rsi]`)
I_LOAD_ARITH	Integer Load + Arithmetic Instruction (e.g. `add r8, qword ptr [rsi+0x10]`)
I_ARITH_1OP	Integer Pure Arithmetic Instruction, 1 operand (e.g `add rcx, 0x40`)
I_ARITH_2OP	Integer Pure Arithmetic Instruction, 2 operand (e.g `add rcx, rax`)
I_REG_MOV	Integer Register Move Instruction (e.g. `mov rax, rdi`)
F_PURE_LOAD	Floating Point Pure Load Instruction (e.g. `movdqa xmm5, xmmword ptr [rdi]`)
F_LOAD_ARITH	Floating Point Load with Arithmetic Instruction (e.g. `pminub xmm4, xmmword ptr [rdi+0x30]`)
F_PURE_ARITH	Floating Point Pure Arithmetic Instruction (e.g. `paddd xmm0, xmm6`)
F_REG_MOVE	Floating Point Register Move Instructions (e.g. `movd xmm0, esi`)

Table 1. Instruction Classifications

Considerations For x86 Architecture

With the above classification we cover about 99% of the instructions encountered in our testing that write to a register. There are more specialized operations that we are not included in this analysis, such as 3 and 4 operand instructions. Certain instructions may write to more than 1 register, in particular, the exchange register instruction. It's not supported for now.
Unlike the fixed-width registers in the Lipasti paper, the x86 architecture has registers for multiple data-types, which needed to be supported.

Implementation

The logic for categorizing instructions can be found in function definition `INST_CAT set_instr_cat(INS ins, REG write_reg)`.

In order to support the different register types used in the x86 architecture, we defined a class `regval` of a union type, whose goal is to abstract away the different register types of x86 into one data type to be used within our value predictor data structures. The `regval` can wrap registers of up to 512 bits—the largest-size registers used in x86. As those registers cannot be stored in one machine word, their underlying implementation is a byte array rather than an unsigned 64-bit integer. Of course, these differences were all ultimately abstracted away such that through this class, the value of any register type within x86 could be constructed, represented, and compared.

To simulate a value predictor, our pintool is invoked *after* each instruction so as to record the value that was written to the output register of the instruction. To each callback, we pass the instruction pointer (used to uniquely identify each instruction), a pointer to a structure containing general metadata about the instruction, and a pointer to a memory-mapped view of the destination register of the instruction which is being instrumented. The memory-mapped view of the destination register is used to populate a `regval` instance.

We define the class, `inst_data` to represent general metadata pertaining to each instruction, identified by its instruction pointer. This class records static information such as opcode, category, datatype write register and the instruction's string representation (for debugging purposes). It also records dynamic information about the instruction, which includes statistics about the instruction, as well as an instance of the `regval` class, indicating the last value written by that instruction. The `inst_data` instance for a particular instruction is initialized during the first encounter of that instruction and does not change for the lifetime of the program, with the exception of the statistics.

The data contained in `inst_data` alone is sufficient to deduce the degree of value locality for every instruction, as defined by the Lipasi paper. We compute value locality by recording the number of times that the `regval` most recently written by the instruction equals exactly the past `regval` written by the instruction. This is then divided by the number of times the particular instruction was executed. We can then aggregate these results over all instructions of a particular type by iterating through the `inst_data` objects and filtering for the desired instruction class.

Results

We tracked the locality of each operation category using a history depth of 1. The following chart shows the percent locality for each instruction (bars) as well as the number of occurrences of each instruction type (line).

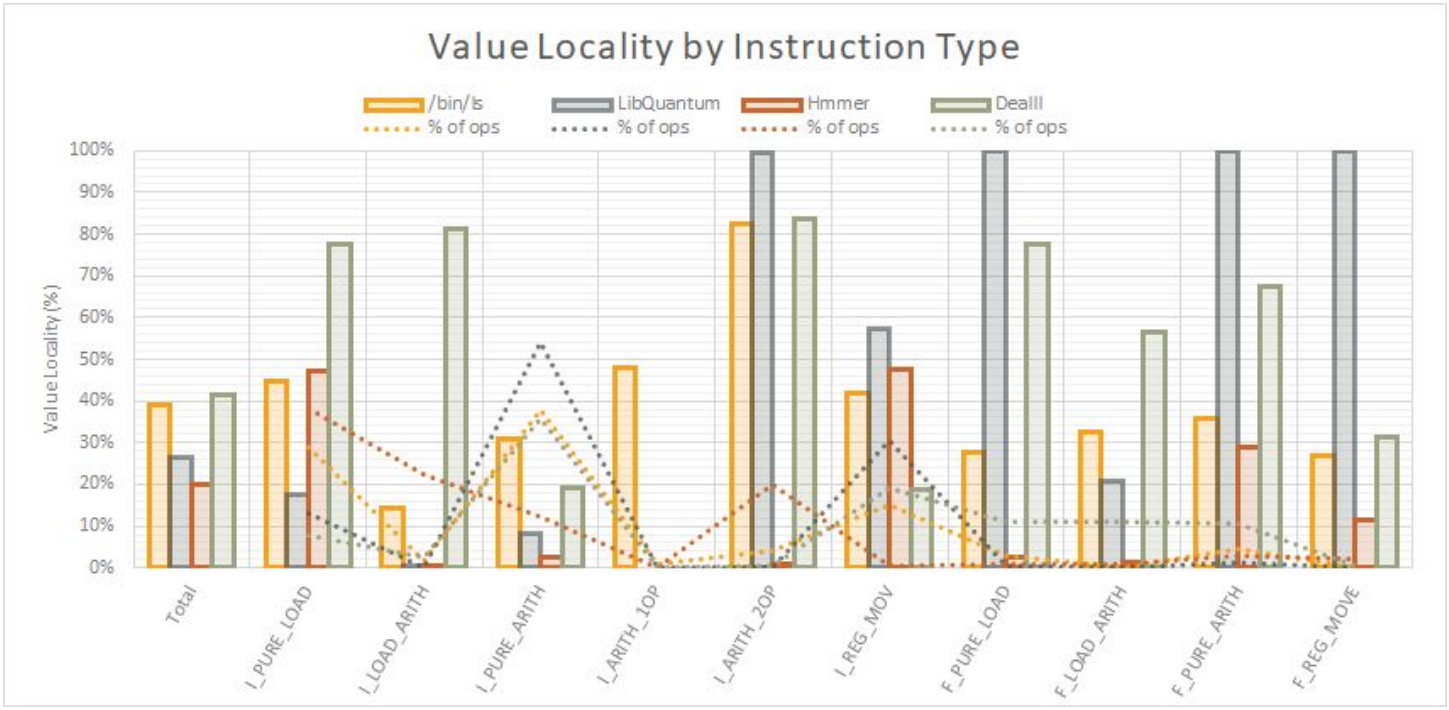


Figure 1. Value Locality by Instruction Type
For each of the operation categories we calculate the value locality with history depth of 1

On average, we find ~30% of executed operations exhibit value locality at a history depth of 1.

However, there is a large variance in how instructions behave between each benchmark. The Hmmer benchmark executes significantly more 2OP Integer Arithmetic (I_ARITH_2OP) operations, but these exhibit extremely low locality.

Pure Integer Load (I_PURE_LOAD) and Integer Register Move (I_REG_MOV) operations both have strong locality for all benchmarks (>20%), and occur quite frequently. These could be good instructions to use for value prediction. Conversely, I_PURE_ARITH operations occur very frequently, but the locality is low (~10%), so it could be difficult to predict values for these operations.

Value Prediction Table (VPT)

Implementation

The Value prediction table

To implement the value prediction table, we save a list of values keyed by the lowest n bits of the Instruction Pointer address (PC). In this simple implementation we do not yet use a classification table or prediction history. Instead, the code always attempts to make a prediction. The value history is always updated for each instruction using a depth of 1 (i.e. the value history is replaced at every invocation of the instruction with no hysteresis).

Results

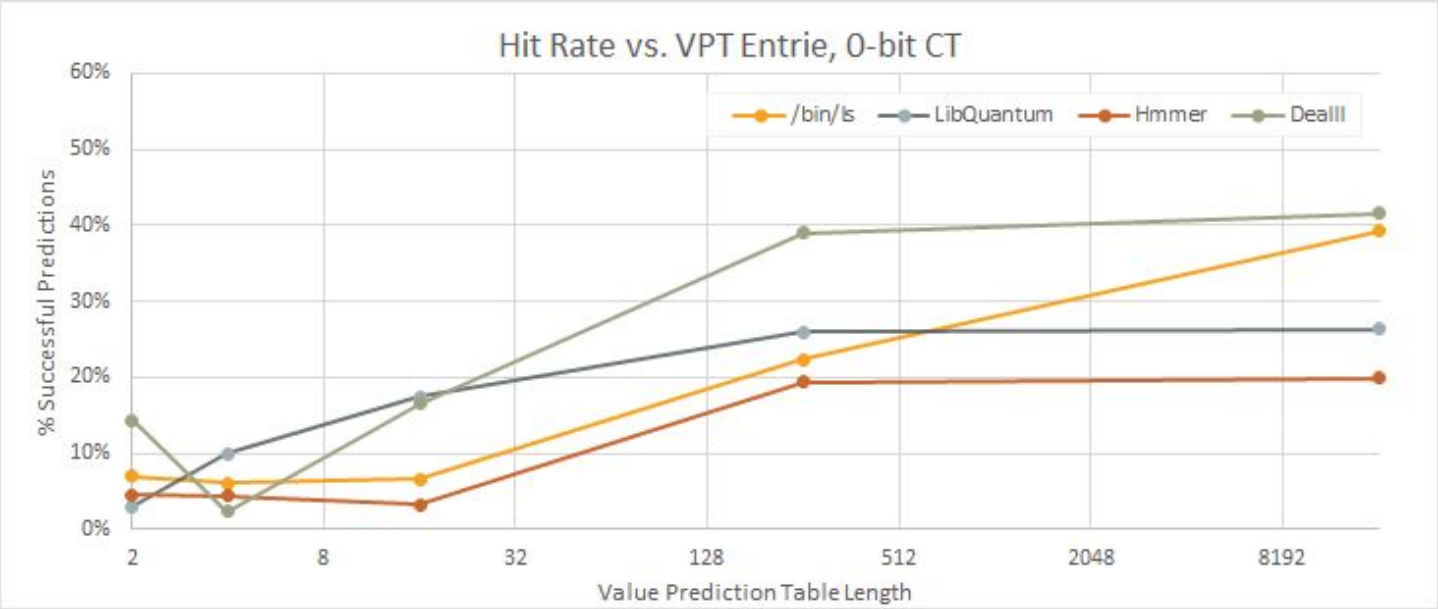


Figure 2. Prediction Success with no prediction history
For each operation we check the VPT table for a match

We find that each of the benchmarks approaches the value locality we saw in Figure 1 (Total). VPT size below 32 entries has very few successful entries, but we find that around 256 entries most of the benchmarks have achieved almost maximum locality. The exception is /bin/ls.

Classification Table

Implementation

Because mispredictions can be costly and an update to the VPT could evict a value which will later be used, the Lipasti study includes a classification table which will increment or decrement a saturating counter corresponding to the instruction pointer address. This adds hysteresis to the prediction.

The following table is used to set the appropriate threshold for prediction and replacement:

Configuration (entries/bits)	State Descriptions
0-bit	Always Predict
1-bit	{0=no pred, 1-pred & no repl}
2-bit	{0,1=no pred, 2,3=pred, 3=no repl}
3-bit	{0,1=no pred, 2-7=pred, 5-7=no repl}

Table 1. Classification Table Configurations

Results

The following are the results of adding a 1-bit and 2-bit saturating counter Classification Table. In each case the CT is the same number of entries as the VPT. The dotted line for each entry indicates the percent of times a prediction was taken.

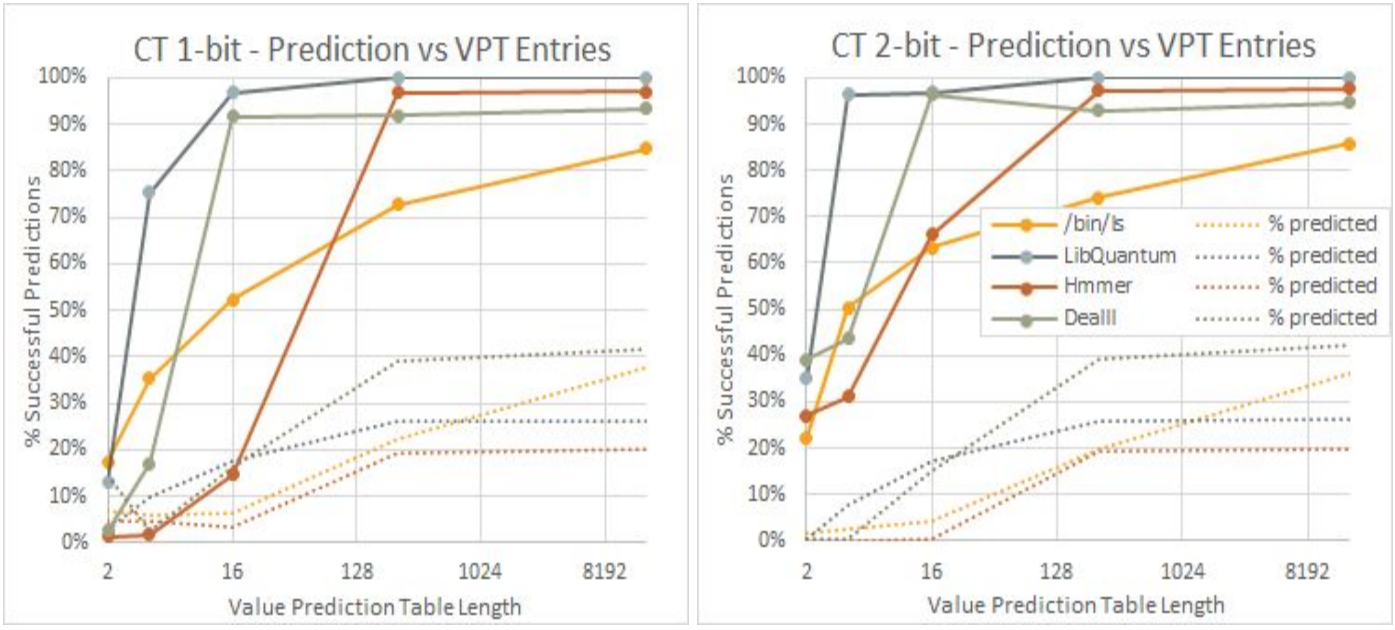


Figure 3 & 4. Success Rate for Predictions Classification Table

We see the classification table does a great job of getting the predictions close to 100%, even with few entries.

Optimization

We also varied the Classification table entries and bits for the saturating counter. The following table recreates the results from Figure 6. In the Lipasti paper. The results are an aggregate across the four benchmarks.

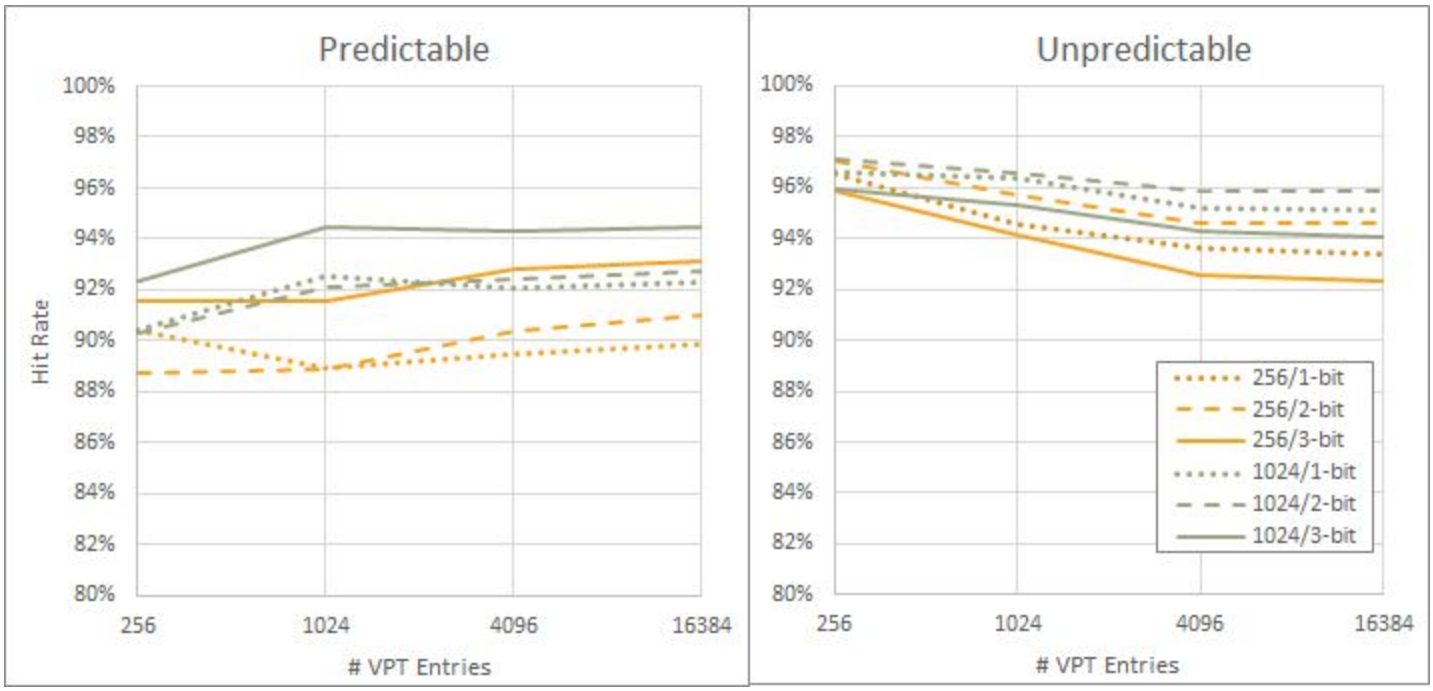


Figure 5 & 6. Hit Rates for predictable and unpredictable Historical Values

We find that the general trends in the data match expected results. The predictable rate generally increases as we add more entries to the Value Prediction Table and/or classification table.

[Deviations from Lipasti Paper] We find generally a higher hit rate of our tests. The two most likely causes of this are the underlying architecture we are running on and the benchmarks we are using. The /bin/ls results actually match much more closely to the results from the Lipasti paper than any of the 3 benchmarks. It might suggest that these benchmarks don't have a varied or common set of instructions.

Speedup Calculations

Implementation

We developed a simple model for classifying speedup:

$$\text{speedup} = \frac{CPI * \#\ Instructions}{CPI * \#\ Instructions - STALL\_CYCLES * \%\ Successful\ Predictions + MIS\_PENALTY * \%\ False\ Predictions}$$

Where CPI = 2, STALL_CYCLES = 1, MIS_PENALTY = 1

Effectively, we start with a base 2 cycles per instruction, decrease one cycle for each correct prediction, and increase for each false prediction.

This is a large simplification because it generally assumes that all instructions are stalled for one cycle. Although this is a good aggregate metric, a better solution would analyze each instruction type for statistical stalls due to true data dependencies. This would involve a significant simulation and knowledge about the underlying architecture we do not have access to.

Results

For each of the Value Prediction Unit Configurations from Table 6 in the Lipasti paper (shown below) we found the following speedup results.

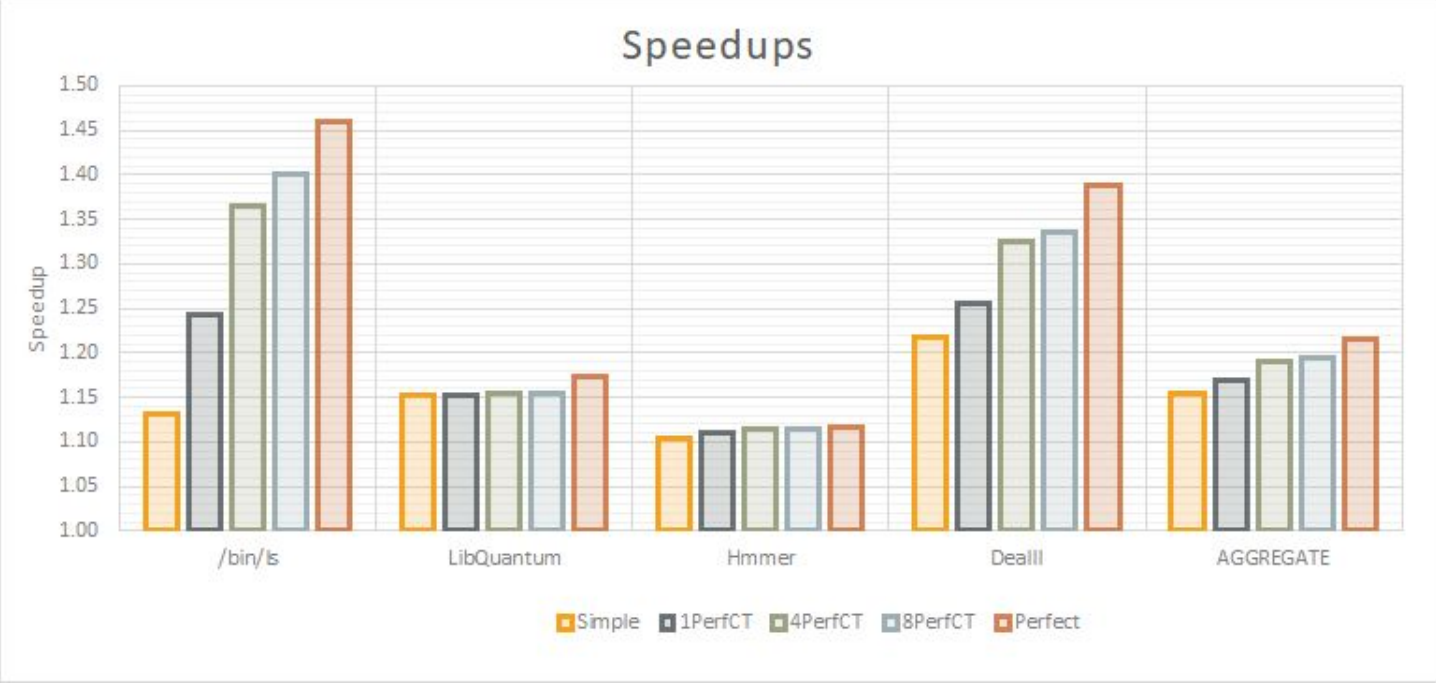


Figure 7. Speedup results for common Value Prediction Unit Configurations

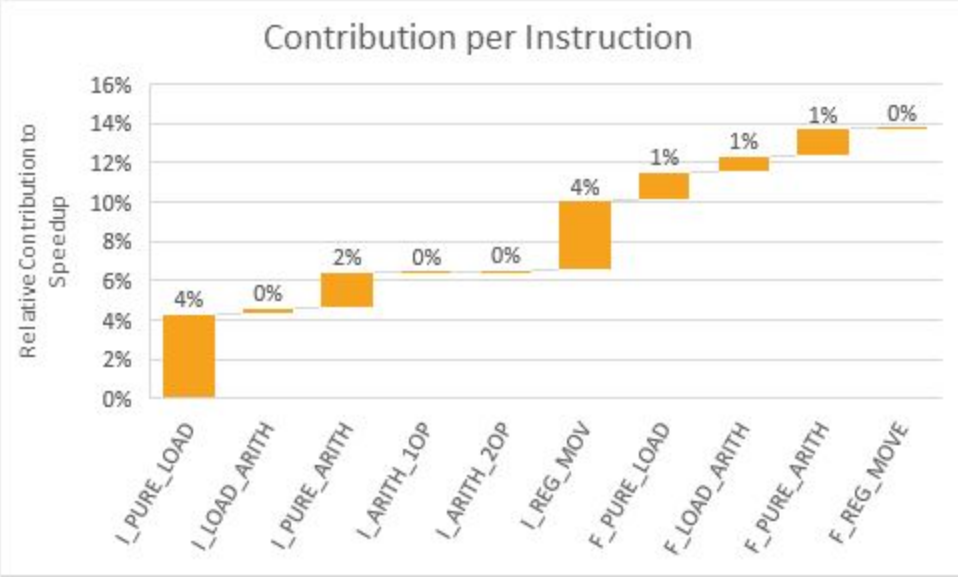
TABLE 6. VP Unit Configurations.

Config-uration	VPT		CT	
	Entries	History Depth	Entries	Bits/ Entry
Simple	4096	1	1024	2
1PerfCT	4096	1	∞	Perfect
4PerfCT	4096	4/Perfect	∞	Perfect
8PerfCT	4096	8/Perfect	∞	Perfect
Perfect	∞	Perfect	∞	Perfect

We find an aggregate 1.15x speedup due to the value prediction units, with performance increasing steadily with increased classification.

Instruction Types Revisited for Speedup

We also took another look at how each instruction type contributes to the speedup. The following plot shows the contribution of each instruction category toward the total speedup. We could make the case for omitting instruction types with low contribution, like most integer arithmetic operations and floating point register moves. However, none of the instruction types contribute negatively to the speedups with a reasonably sized VPT, even in the 'Simple' case above.



Further Improvements to VPU

Victim Cache

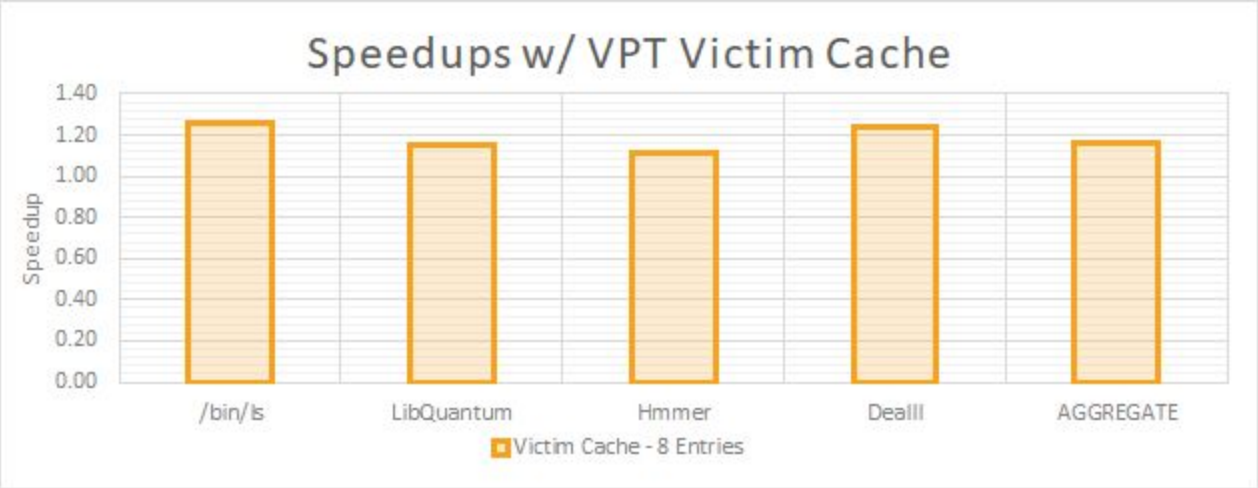
One improvement to the Value Prediction Unit could be the addition associativity in the VPU. One implementation is a Victim Cache which can buffer the LRU values ejected from the VPT. Similar to victim cache in memory, this victim cache will contain the PC tag and value history. When the PC value isn't found in the VPU, check the victim cache.

This will help the cases we see in the data where a value is repeatedly ejected from the VPT and replaced with another.

Results

The following test was done using:

- 256 VPT entries
- 1024 CT entries
- 2-bits CT
- 1 Value History Depth
- 8 Victim Buffer entries



The initial results from the Victim cache implementation look promising. It appears very effective in allowing us to reduce the number of VPT entries and still getting hits. We are able to achieve the speedup of the '8PerfCT' above using significantly fewer resources with a small Victim Cache. (Note: this implementation would require more testing before declaring it a valid solution).

Delta

Another improvement that could be made involves looking for values that are incrementing or decrementing. This can be done by calculating if the data on a misprediction is off by a value of 1 and applying that delta to subsequent predictions. Though a promising idea, we were unable to test this.