Performance and Energy Impact of Instruction-Level Value Predictor Filtering

Ravi Bhargava and Lizy K. John
Laboratory for Computer Architecture
Electrical and Computer Engineering Department
The University of Texas at Austin
{*ravib,ljohn*}*@ece.utexas.edu*

Abstract

This work evaluates value predictor access filtering and its effects on performance and dynamic energy consumption in a wide-issue, high-frequency processor. New and previously proposed filtering strategies are analyzed with realistic predictor constraints, such as port restrictions and table access latency. Filters restrict access to the value predictor for instructions with unconsumed predictions, poorly predicted instructions, and quickly executing instructions. Read access filtering improves speedup due to value prediction from 16.1% to 23.6%, while reducing dynamic value predictor reads by 31.3%. Adding write filtering decreases update activity by 78.6%, while still providing 14.8% speedup. The overall reduction in activity leads to a value predictor energy consummption decrease of 52.6%.

1 Introduction

Value prediction has the potential to be a high performance mechanism in future high-frequency, wide-issue environments [2, 11]. With a high instruction fetch bandwidth, data consumers arrive very quickly, usually within one cycle. This scenario gives value prediction ample opportunity to break important data dependencies.

This same push toward high frequencies and wider issue widths creates a wire-delay constrained processor, causing a nontrivial latency for large centralized structures [1, 4]. An excessive value prediction latency is a detriment to overall instruction throughput, reducing the effectiveness of successful predictions by prolonging the resolution of data dependencies. In the presence of a lengthy latency for computing a predicted value, an instruction can produce its actual result before the predicted result is available.

Restricting Access Ports One way to maintain a reasonable value predictor latency is to restrict the number of access ports. Limiting ports greatly reduces the size of the value prediction tables and therefore the delay due to data traveling on wires. However, by limiting ports, it is possible that not all eligible instructions receive a value prediction.

In a wide-issue environment, simultaneously fetched instructions compete for value prediction resources. In the SPEC CPU2000 integer benchmarks, 61% to 78% of all instructions are eligible for value prediction. When limiting the number of ports, the instructions which access the value predictor must be chosen carefully to achieve high performance.

Filtering Value predictor read access filtering is the process of determining whether an instruction should request a prediction. One goal of filtering is to limit access based on which instructions benefit the most (or least) from value prediction. This enables the read access ports to be utilized in a more judicious manner. The filtering decisions are made on an per-instruction basis. They can be based on static information such as instruction type, or dynamic information such as past prediction history.

Filtering is not limited to just read accesses. Write updates can also be filtered. Updates are performed at retire-time and generally update bandwidth is not as critical as read bandwidth [14]. However, if done well, the filtering of instructions at update can lead to more efficient use of table entries by reducing the number of updates and subsequent replacements.

Energy Considerations Another design issue in a high-performance processor is energy consumption. More transistors, higher clock rates, incorrect speculations, and wider microarchitectures all contribute to this growing problem. Previous work shows that a traditional at-fetch hybrid value predictor can consume 10 times as much energy as all of the on-chip caches combined because of high predictor activity and high performance table design [2].

Access filtering has an obvious energy benefit. By

reducing the number of instructions that read and write the predictor, the dynamic activity and, therefore, the dynamic energy consumption decrease. The key is to achieve this energy reduction without losing the desired level of performance.

Previous Filtering Work Calder et al. present techniques for choosing which instructions to value predict, performing replacement in the tables, and filtering value predictor updates [7]. They concentrate on improving table efficiency and predictor accuracy to minimize the misprediction recovery time. The strategies presented include entirely eliminating non-load instructions and filtering updates from instructions with unconsumed predictions. They also investigate altering value prediction confidence thresholds based on the length of an instruction's current path.

Tune et al. continue this work with Critical Path Predictions which they apply to value prediction [22]. Their architecture allows only one value prediction per cycle (one read port). Critical Path Prediction is used to determine which instructions will use this resource. Fields et al. also propose a mechanism to isolate critical instructions, but they apply it only to value prediction update [9].

Rychlik et al. discuss the "usefulness" of value predictions [19]. For SPEC CPU95, they observe that 31% (integer programs) and 62% (floating point programs) of predicted values are never read before they are overwritten by the final result. The authors introduce a mechanism which only updates the value predictor on useful predictions. However, this mechanism did not improve prediction rates, while instruction throughput increased very slightly.

Using a software approach, Burtscher et al. demonstrate that a compiler can be an effective tool for determining which load instructions are most suitable for value prediction [6]. The static filtering approach eliminates the need for some dynamic hardware.

The rest of this paper is organized as follows. In Section 2, the evaluated value predictors are presented along with background information. Section 3 describes the value predictor filtering strategies and an implementation. Next, a performance analysis for the different strategies and the impact on energy consumption are presented in Section 4. Finally, Section 5 summarizes this work.

2 Value Predictor Design

This section discusses the design of the evaluated value predictors. There are many proposed strategies for value prediction. The primary ones include last value prediction [15, 16], stride prediction [10, 12], context prediction [20, 23], and hybrid prediction [18, 23]. More recently, hybrid predictors with the ability to dynamically classify instructions have been evaluated [14, 19]. In this work, we look primarily at a hybrid predictor with a last value predictor, stride predictor and a context predictor, similar to predictors used previously [14, 19], but without the dynamic classification schemes.

2.1 At-Fetch Prediction

In this work, value prediction is performed at instruction fetch, which is a commonly assumed implementation [7, 11, 15, 18]. In a typical processor, an instruction address can be sent to the fetch logic each cycle. This same fetch address is used to access the value predictor. Based on the fetch address, the value predictor generates predictions for all of the instructions being fetched in that cycle.

At-fetch prediction hides value predictor latency. There is no need for a predicted value until the instruction has been decoded and renamed. Therefore, some or all of the value predictor's table access latency is hidden by the instruction fetch latency and decode stages. This is an advantage over post-decode prediction [2], which waits until more instruction information is available before accessing the value predictor.

At-fetch prediction has some restrictions as well. In a trace cache processor that fetches past branches in a single cycle, such as the one presented in this work, the instruction address can be non-contiguous. Determining the address for each fetched instruction requires more information than is typically available at fetch. The second problem is the lack of instruction type information. During fetch, the instructions are indistinguishable, so value prediction resources are being consumed by instructions that are not eligible for predictions (e.g. branches, stores). These conditions are costly for a port-constrained predictor, and are modeled in our baseline at-fetch predictor.

2.2 Evaluated Predictors

Three basic value predictors are studied. The primary predictor is accessed at fetch time with realistic port, instruction type, and latency constraints (*At-Fetch*). For comparison, another at-fetch predictor is presented with unlimited ports, all instruction type information, and no access latency (*Optimistic*). Finally, a post-decode predictor is also presented for comparison (*Post-Decode*).

All three are hybrid value predictors that use a last value sub-predictor, a stride sub-predictor, and a context sub-predictor. The two-level context value sub-predictor is similar to the one discussed by Wang and Franklin [23]. The studied stride sub-predictor uses matched (two-delta) stride prediction [8].

The predictor table configurations are chosen based on a performance and energy analysis that incorporated table access latency [2]. The last value sub-predictor and stride sub-predictor both use a four-ported, 8192-entry, direct-mapped table. Both levels of the context sub-predictor use a four-ported, 1024-entry, direct-mapped table. This particular size is chosen such that the total access latency of the two context tables matches that of the stride and last value sub-predictor tables.

The value predictor read access latency is derived from Cacti, an analytical cache modeling tool [17]. Targeting a 3.5 GHz processor at 1.1 Volts in a 100nm technology [21], the reported latency to access the value predictor is eight cycles. For at-fetch prediction, four of those cycles are hidden by early stages of the pipeline. However, for post-decode prediction, value predicted instructions are exposed to the entire value predictor latency.

For the baseline predictors, each table is tagged and updated by every result-producing instruction. When a value mis-speculation is encountered, the microarchitecture reissues all instructions younger than the mispredicted instruction. All value predictors are assumed to be fully pipelined and capable of serving new requests every cycle.

3 Access Filtering

This section discusses the applied filtering techniques. There are four read access filtering techniques and one write access filtering technique. In addition to describing the techniques, an implementation for collecting the filtering information and accessing the value predictor is discussed.

3.1 Filters

The **instruction type filter** allows instructions to access an at-fetch predictor with information that is normally not available until after instruction decode. For instance, knowledge of instruction types and instruction addresses beyond the first branch are not typically known (although this knowledge is typically assumed). This filter is essentially detecting instructions that do not produce results and allowing only result-producing instructions to access the value predictor.

The **quick execution filter** identifies quickly executing instructions, defined as instructions that execute before a prediction becomes available, and does not allow them to read the value predictor. This is unique to processors which incur a high value prediction latency. A two-bit saturating counter is initialized to zero, increments when a quick instruction is detected, decremented otherwise, and filters occur when the counter value is above or equal to two.

The **useless prediction filter** is based on the observation of *useless* predictions [7, 18]. It prevents instructions whose predictions are not consumed during the life of the instruction from doing any more predictions. Once again, a two-bit saturating counter is used. It is initialized to zero, incremented on a useless detection, decremented otherwise, and filters accesses when the counter value is above or equal to two.

The **mispredict filter** identifies instructions that commonly mispredict in the value predictor. These instructions are then forbidden from reading the value predictor. This filter uses a three-bit counter, initialized to zero, incremented on mispredicts, and decremented otherwise. It only restricts read access when it is fully saturated at seven. This is different from previous confidence schemes because once an instruction is designated as mispredicting, it cannot read the value predictor again until the counter is reset (discussed in Section 3.2).

The **write filter** is the only presented strategy for filtering predictor updates. It prevents instructions that did not access the predictor at fetch-time from updating the predictor. Therefore, it is really based on the read access filters. This strategy is not always beneficial. If one particular instance of an instruction does not read the predictor, it does not mean that future dynamic instances of the same instruction won't be able to read the predictor. However, for energy consumption, table efficiency, and port contention purposes, it may be beneficial to reduce the update traffic by filtering.

3.2 Collecting Filter History

The value predictor filters all require per-instruction execution history. To help achieve this, filter information is stored in the trace cache. The trace cache framework provides a unique instruction feedback loop. Instructions are fetched from the trace cache, executed, and then compiled into new traces. By adding a few small fields for each instruction slot in the trace cache, a low-latency, low-energy, per-instruction profiling mechanism is established.

Dynamically updated information, such as counters and state information, are stored in the trace cache between dynamic invocations of an instruction. This profiled data is fetched along with the instructions and remains associated with the instruction as it travels through the pipeline. This trace cache based method of instruction history collection and distribution has been leveraged successfully in other recent work [2, 3, 14].

Filtering data is now subject to history loss on trace cache line replacements. When a trace cache line is evicted from the trace cache, all of the corresponding filter data for each instruction in the trace is lost. The overall

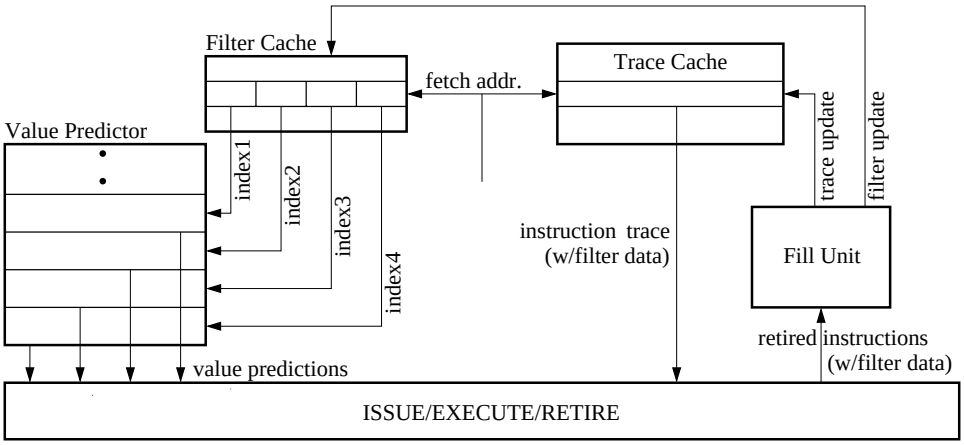

Figure 1: Value Prediction With A Filter Cache

performance impact is not great (except perhaps under extreme thrashing circumstances). However, the re-entry of instructions into the trace cache serves as a method for resetting filter counters that have become "stuck" at the threshold levels. This provides another opportunity for instructions with varying behaviors to avoid filtering.

3.3 Filter Hardware Design

The per-instruction filter data needs to be transformed into value predictor table indexes. At retire-time, the fill unit analyzes the available profile information in the trace that it is currently constructing. Based on the dynamically collected filter information and instruction types, up to four instructions are chosen for value prediction. If more than four instructions are eligible and unfiltered, then the first (or oldest) instructions are chosen.

The indexes must be stored and then read prior to accessing the value predictor tables. The addresses of the instructions selected for prediction are condensed into a small trace of value predictor indexes, and stored in a separate cache. This cache, referred to as the *filter cache*, is accessed by the same address that indexes the trace cache.

Figure 1 illustrates the basic organization of a value predictor with a filter cache. The filter cache is designed with the same configuration as the trace cache and therefore can be indexed with the same fetch addresses. The filter cache provides indexes that are consumed by the value predictor, which in turn provides value predictions. The fill unit updates the filter cache with traces of indexes that correspond to a instruction trace in the trace cache.

There are several ways to implement the filter cache. A straightforward implementation is to obtain indexes for the instructions being fetched. However, in this case, the filter cache access would be serial to the value predictor access, compounding the latency. This option is investigated in the analysis.

Another option is to have a lookahead filter cache. In this case, the current trace cache fetch address is still used to index the filter cache. However, the trace of indexes in the entry would be for a trace cache access in the future. This allows the filter cache access latency to be tolerated without adding to the overall value prediction latency.

There are variations on how a lookahead predictor can be implemented to reduce the likelihood of a lookahead miss, including storing multiple traces of indexes and choosing among them as more fetch information becomes available. Further discussion of this style of design is beyond the scope of this paper. However, an ideal lookahead filter cache is also studied.

4 Results and Analysis

4.1 Simulation Methodology

Six of the integer benchmarks from the SPEC CPU2000 suite are presented. Programs that improve the most from value prediction are chosen. The benchmark executables are the precompiled Alpha binaries available with the SimpleScalar 3.0 simulator [5]. The MinneSPEC reduced input set [13] is used when applicable. Otherwise, the SPEC *test* input is used. The benchmarks and their respective inputs are presented in Table 1. All benchmarks are run for 100 million instructions after skipping the first 100 million instructions.

Table 1: SPEC CINT2000 Benchmarks

Benchmark	Input Source	Inputs
crafty	SPEC test	crafty.in
gap	SPEC test	-q -m 64M test.in
mcf	MinneSPEC	lgred.in
perlbmk	MinneSPEC	mdred.makerand.pl
twolf	MinneSPEC	mdred
vortex	MinneSPEC	mdred.raw

To perform the simulations, a detailed, cycle-accurate microprocessor simulator is interfaced to the functional simulator from the SimpleScalar 3.0 simulator suite (*sim-fast*) [5]. The basic pipeline consists of eight stages: three stages of fetch plus decode/merge, rename, issue, execute, and writeback. Memory operations require additional pipeline stages, including TLB access and cache access. The parameters for the simulated base microarchitecture are in Table 2.

Table 2: Baseline Microarchitecture Configuration

Data memory	
· L1 Data Cache:	4-way, 32KB, 2-cycle access
· L2 Unified cache:	4-way, 1MB, 10 cycle
· Non-blocking	12 MSHRs and 2 ports
· D-TLB	512-entry, 4-way 1-cycle hit, 30-cycle miss
· Store buffer:	32-entry w/load forwarding
· Load queue:	32-entry, no spec. disambiguation
· Main Memory	Infinite, 75 cycle

Fetch Engine	
· Trace cache:	4-way, 1K entry, 3-cycle access 16 instr./entry, max. 3 blocks partial matching, no path assoc.
· L1 Instr cache:	4-way, 4KB, 1-cycle access one basic block per access
· Branch Predictor:	16k entry gshare/bimodal predictor
· BTB	512 entries, 4-way

Execution Core			
· Functional unit	#	Exec. lat.	Issue lat.
Load/store	6	1 cycle	1 cycle
Simple Integer	8	1	1
Int. Mul/Div	2	3/20	1/19
Simple FP	4	3	1
FP Mul/Div/Sqrt	1	3/12/24	1/12/24
Branch	4	1	1
· Data Forwarding Latency: 1 cycle			
· Register File Latency: 2 cycle			
· 128-entry ROB			
· 8 reservation station entries per func. unit			
· Fetch width: 16			
· Decode width: 16			
· Issue width: 16			
· Execute width: 16			
· Retire width: 16			

4.2 Performance Analysis

In addition to the baseline at-fetch, post-decode, and optimistic value predictors, the performance impact of two variations of a filtered at-fetch predictor is studied. These results are shown in Figure 2. On average, at-fetch value prediction outperforms post-decode value prediction. The additional information available after decode does not help performance enough to overcome the extra cycles of unhidden value predictor latency. Therefore, we concentrate on the at-fetch value predictor for the filtering analysis.

The first filtered value prediction scheme (*AF w/read filt. & serial FC*) represents a filter cache and value predictor that are accessed serially. In this case, the value prediction incurs three extra cycles of latency to account for the filter cache. By making better use of the limited read ports, this value prediction scheme leads to a significant improvement in speedup (22.2%) when compared the unfiltered at-fetch value predictor (16.1%).

The second filtered scheme (*AF w/read filt. & LA FC*)

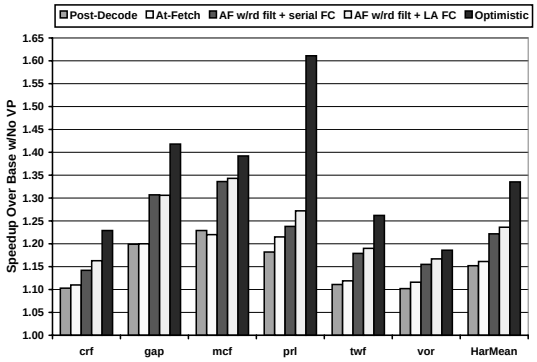

Figure 2: Value Predictor Speedup Comparison

assumes a perfect lookahead filter cache. Therefore, the value predictor latency consists just of the value predictor table access latency. In this scheme, the value predictor benefits from a reduced latency and improved read access efficiency. These advantages are reflected by the further improved speedup (23.6%). However, the filtering of the lookahead filter cache is not enough to improve performance to the level of the optimistic value predictor (33.5%).

The speedup provided using the filters is due to improved efficiency and accuracy. Table 3a presents the port utilization during value predictor read. This value is the average number of reads during cycles with read activity. It includes reads from wrong path instructions and instructions not eligible for prediction (in the case of *At-Fetch*).

In the optimistic scenario, around six ports are desired on average for value predictor reads. However, *At-Fetch* and *Post-Decode* only average around half of this because of their restricted ports. This illustrates the need for a filtering mechanism. When the read filters are applied to an at-fetch predictor, the read port usage is reduced to 2.89 reads per active cycle.

Not every eligible instruction receives a prediction due to port restrictions, confidence mechanisms, and read filters. The percentage of eligible retired instructions that receive a prediction are shown in Table 3b. *Post-Decode* predicts for a higher percentage of instructions than *At-Fetch* because it does not waste ports on ineligible instructions. *AF rd filt* does not have this problem due to the instruction type filter but still predicts fewer instructions than *Post-Decode* because of the other filters. *Optimistic* performs almost two times the number of predictions as *At-Fetch*, which is one reason for its superior performance.

Value prediction accuracies are presented in Table 3c. The accuracy of value prediction is not significantly affected by the stage in which it is accessed or the presence of filters. However, overall, the filtered at-fetch predictor has the best prediction accuracy, better than even the

Table 3: Value Prediction Run-Time Characteristics

a. Value Predictor Port Utilization at Read

	AF	PD	AF rd filt	Opt.
crafty	3.39	3.34	3.04	6.94
gap	3.29	3.22	2.96	6.35
mcf	3.16	3.11	2.84	5.95
perlbmk	3.26	3.31	3.14	6.84
twolf	3.03	2.97	2.84	6.19
vortex	3.23	2.92	2.53	4.91
average	3.23	3.14	2.89	6.20

b. Predicted Eligible Instruction Percentage

	AF	PD	AF rd filt	Opt.
crafty	29.76%	34.13%	32.03%	62.07%
gap	25.99%	29.51%	29.04%	50.52%
mcf	24.12%	28.32%	27.06%	42.01%
perlbmk	36.96%	45.21%	41.15%	78.95%
twolf	15.15%	17.03%	17.64%	31.33%
vortex	26.30%	34.23%	29.59%	58.17%
average	26.38%	31.41%	29.42%	53.84%

c. Value Predictor Accuracy

	AF	PD	AF rd filt	Opt.
crafty	94.51%	94.47%	95.21%	94.14%
gap	94.37%	93.77%	96.76%	95.87%
mcf	97.51%	97.64%	97.64%	97.19%
perlbmk	99.95%	99.91%	99.94%	99.96%
twolf	85.84%	86.56%	92.57%	91.44%
vortex	96.12%	96.26%	97.72%	97.29%
average	94.72%	94.70%	96.60%	95.98%

AF: Baseline at-fetch predictor. *PD:* Baseline post-decode predictor. *AF rd filt:* At-fetch predictor with read filtering. *Opt.:* Optimistic predictor.

optimistic predictor.

Figure 3 separates the contributions from the different filters. All of the filters reduce the number of instruction eligible to read from the value predictor. However, as this figure illustrates, that is not always enough to improve performance significantly. No single filter improve performance significantly, and the *useless* filter actually leads to lower average performance in isolation (also seen by Rychlik et al. [19]).

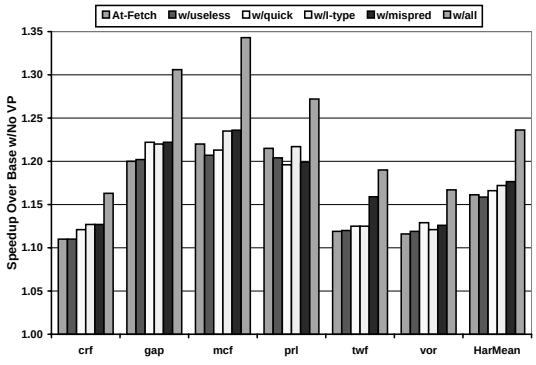

Figure 3: Individual Read Filter Isolated Contributions

The only filter that provides an improvement for every benchmark is the instruction type filter (*i-type*). While other filters may remove potentially useful instructions, the *i-type* filter only isolates instructions that are not eligible for value prediction. Overall, the best isolated filter scheme is the *mispred* filter, which prevents commonly mispredicted instructions from proving the value predictor for a prediction. Finally, the figure also shows that combining all of the filters provides a synergistic speedup improvement.

Table 4 presents the percentage of all prediction-eligible instructions that are filtered at read due to the useless, mispredict,and quick filters (*% filtered*). This value ranges widely from about one-third of the eligible instructions in *perlbmk* to more than two-thirds in *twolf*. The remaining rows breakdown these filtered instructions into groups based on which read filter (not including the instruction type filter) is responsible for restricting instruction port access at read. By far, the mispredict filter is the best filter at identifying instructions that other filters do not (*% mispred filt*). The quick filter is also very adept at finding unique instructions to filter in most benchmarks. However, the useless filter does not find many unique instructions to filter. Around one-fourth of the instructions are filtered by more than one filter (*% two filts* and *% three filts*).

Table 4: Breakdown of Dynamic Value Predictor Filtering

	crf	gap	mcf	prl	twf	vor
% filtered	44.67	54.79	63.46	37.19	67.93	55.43
% useless filt	0.59	0.75	0.44	2.79	0.25	0.82
% mispred filt	57.27	65.45	58.63	46.25	80.40	45.84
% quick filt	28.31	18.34	15.81	44.65	8.50	27.81
% two filts	12.78	13.19	23.92	6.21	9.86	23.13
% three filts	1.05	2.27	1.20	0.10	0.99	2.40

% filtered is the percentage of value prediction eligible instructions that are filtered at read (including wrong-path instructions). The remaining rows are a percentage of all filtered instructions.

4.3 Energy Analysis

This section evaluates the dynamic energy consumption for the presented value predictors. The read and write filters reduce the amount of port activity, which is directly related to value predictor energy consumption.

The read and write activity of two value predictors are presented in Figure 4. The at-fetch value predictors performs both read and write filtering. A load-only post-decode value predictor is also analyzed. Load instructions provide a large percentage of value prediction speedup and a natural means of filtering. Therefore, a load-only predictor is believed to be an energy-efficient value predictor, and is presented for comparison purposes. The load-only value predictor is configured like the baseline post-decode value predictor.

The read port activity decreases by an average of 31.3% when filters are applied to the baseline at-fetch predictor. The decrease is due to the filtering of instructions, but each filtered instruction does not necessarily lead to one less access. Sometimes when an instruction is filtered,

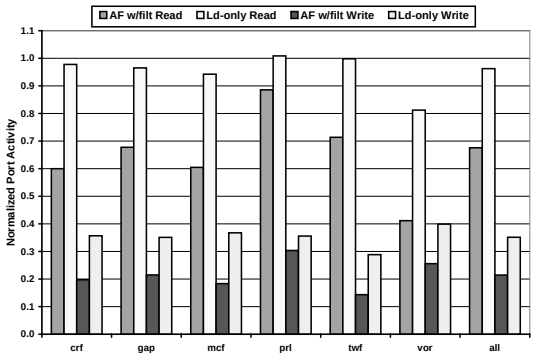

Figure 4: Normalized Port Activity

The activity is normalized to that of the baseline at-fetch predictor.

another instruction that did not previously get read access, takes its place.

The write port activity decreases by an average of 78.6% for the baseline at-fetch predictor. The program *twolf* has the largest decrease in write activity. This is attributed to the fact that it has the largest percentage of eligible instructions that are not predicted as well as the highest mispredction rate. Recall that the write filter prevents instructions that did not read the value predictor from writing the value predictor. Therefore, an instruction can be denied access to the value predictor because of its instruction type, dynamic prediction characteristic (e.g. mispredictions), or port utilization.

The load-only value predictor has a small decrease in read activity. In one case, the activity actually increases versus the baseline at-fetch predictor. This value is dependent upon the percentage of load instructions in the programs. However, the load-only predictor reduces write activity significantly since only one instruction type is updating.

To model value predictor energy consumption, an analytical cache modeling tool [17] provides the energy consumed per read access (writes are assumed to consume as much energy as reads). The energy comparisons are then based on combining these energy per access values with the table activity. Relevant data structures, their configuration, and the energy per port access are presented in Table 5. One important thing to note is the energy efficiency of the filter cache and trace cache compared to the hybrid value predictor tables.

The energy consumed by the hybrid value predictor tables is simply the product of the read and write activity and the energy per access. The filter cache is read each time the value predictor is read, and written only on trace cache writes.

Without embedded filtering data, the trace cache is only written when a unique instruction trace is constructed. With filter data, the trace cache entries are larger and the trace cache is written on on each trace construction. This extra energy is included in the analysis.

Table 5: Energy Per Port Access

Structure	Entries	Ports	Assoc	Data	Energy
LastVP/StrideVP	8192	4	1	8 B	5.80 nJ
Context VP	1024	4	1	8 B	2.26 nJ
Filter Cache	1024	1	2	8 B	0.69 nJ
Basic TC	1024	1	2	64 B	1.00 nJ
TC Filt. Data	1024	1	2	16 B	0.09 nJ

The energy per port access (*Energy*) is obtained using Cacti 2.0 [17]. Cacti allows two read/write ports to be modeled. We model the first two ports this way, and the other ports as one read port and one write port. *Data* is the number of bytes of data per entry. The energy per access for *TC Filt. Data* is the additional energy per trace cache access due to the filter data stored in the trace cache.

The results of the energy evaluation are shown in Figure 5. The reduction in energy consumption due to just the read filters (*AF rd filt*) is 13.7% on average. When the write filtering technique is applied, then the energy consumption is reduced by 52.6%. This more impressive reduction occurs because the value predictor updates are not filtered by port restrictions, like predictor reads. Therefore, every instruction that is identified for write filtering contributes to a reduction in energy. The load-only value predictor falls in between with a 29.4% energy reduction.

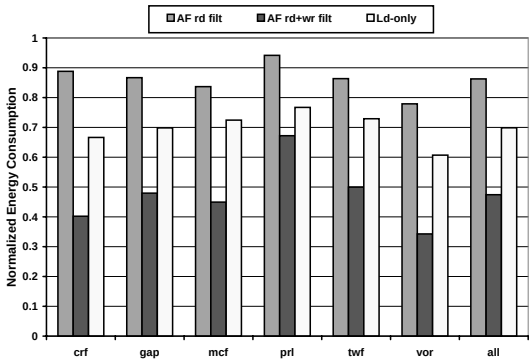

Figure 5: Normalized Value Predictor Energy Consumption

The energy is normalized to that of the baseline at-fetch predictor.

The final piece to this analysis is the performance impact of write filtering (shown in Figure 6). On average, write filtering reduces performance by eliminating possibly useful predictor updates. In this case, the speedup due to read-filtered at-fetch prediction drops from 23.6% to 14.8%. However, this is still superior to the 13.1% speedup from load-only value prediction.

5 Summary

In this work, the impact of value predictor filtering is examined. Value prediction is studied in a high-frequency, wide-issue environment where a high-performance at-

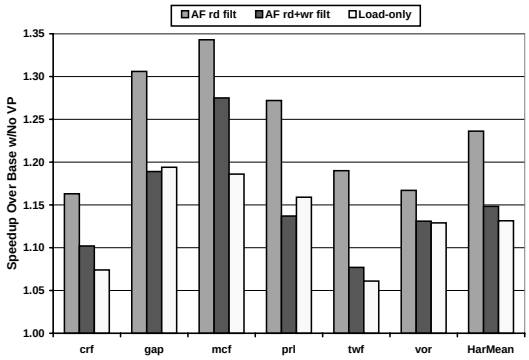

Figure 6: Performance Impact of Write Filtering

The At-Fetch predictor uses the ideal lookahead filter cache.

fetch value predictor is modeled with an extended access latency as well as other realistic design constraints. Various filtering techniques are applied to increase the effectiveness and performance of the value predictor while reducing both the dynamic read and write activity.

Value predictor reads are filtered based on identifying instructions that are eligible for value prediction, execute quickly, do not produce a consumed value, and mispredict frequently. Incorporating the read filters into an at-fetch value predictor improves execution speedup from 16.1% to 23.6%. At the same time, read activity is reduced by 31.3%. When value predictor writes are filtered based on their read status, the write activity is reduced by 78.6% on average. This large decrease in activity leads to a 52.6% reduction in overall dynamic energy consumed by a value predictor, but is accompanied by a degradation in performance.

These results encourage the notion that high-performance and energy-conscious at-fetch value prediction is possible in a high-frequency, wide-issue environment.

Acknowledgments Thanks to the reviewers for their insight and helpful suggestions. This research is partially supported by the National Science Foundation under grant number 0113105, and by AMD, Intel, IBM, Tivoli and Microsoft Corporations.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture*, pages 248–259, Jun 2000.

[2] R. Bhargava and L. K. John. Latency and energy aware value prediction for high-frequency processors. In *16th International Conference on Supercomputing*, pages 45–56, June 2002.

[3] R. Bhargava and L. K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *30th International Symposium on Computer Architecture*, June 2003.

[4] M. Bohr. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[5] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical report, University of Wisconsin, Madison, WI, 1997.

[6] M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data cache misses. In *Conference on Programming Language Design and Implementation*, pages 222–233, June 2002.

[7] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *25th International Symposium on Computer Architecture*, pages 64–74, May 1999.

[8] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Researh and Development*, 1993.

[9] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *28th International Symposium on Computer Architecture*, pages 74–85, Jul 2001.

[10] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report 1080, Technion - Israel Institute of Technology, Nov 1996.

[11] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th International Symposium on Computer Architecture*, pages 272–281, June 1998.

[12] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ILP. In *International Conference on Supercomputing*, pages 21–28, Jul 1998.

[13] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[14] S. Lee, Y. Wang, and P. Yew. Decoupled value prediction on trace processors. In *6th International Symposium on High Performance Computer Architecture*, pages 231–240, Jan 2000.

[15] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitectures*, pages 226–237, Dec 1996.

[16] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct 1996.

[17] G. Reinman and N. Jouppi. An integrated cache timing and power model, 1999. COMPAQ Western Research Lab.

[18] B. Rychlik, J. Faistl, B. Krug, and J. P.Shen. Efficacy and performance impact of value prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 148–154, Oct 1998.

[19] B. Rychlik, J. W. Faistl, B. P. Krug, A. Y. Kurland, J. J. Sung, M. N. Velev, and J. P. Shen. Efficient and accurate value prediction using dynamic classification. Technical report, Carnegie Mellon University, 1998.

[20] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, Dec 1997.

[21] Semiconductor Industry Association. The national technology roadmap for semiconductors, 1999.

[22] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *7th International Symposium on High Performance Computer Architecture*, Jan 2001.

[23] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th International Symposium on Microarchitecture*, pages 281–290, Dec 1997.