Published in the Proceedings of the 26th International Symposium on Computer Architecture, May 1999.

Selective Value Prediction

Brad Calder Glenn Reinman Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
{calder,greinman,tullsen}@cs.ucsd.edu

Abstract

Value Prediction is a relatively new technique to increase instruction-level parallelism by breaking true data dependence chains. A value prediction architecture produces *values, which may be later* consumed *by instructions that execute speculatively using the predicted value.*

This paper examines selective techniques for using value prediction in the presence of predictor capacity constraints and reasonable misprediction penalties. We examine prediction and confidence mechanisms in light of these constraints, and we minimize capacity conflicts through instruction filtering. The latter technique filters which instructions put values into the value prediction table. We examine filtering techniques based on instruction type, as well as giving priority to instructions belonging to the longest data dependence path in the processor's active instruction window. We apply filtering both to the producers of predicted values and the consumers. In addition, we examine the benefit of using different confidence levels for instructions using predicted values on the longest dependence path.

1 Introduction

True data dependencies can greatly impede instruction level parallelism (ILP). Data dependencies decrease ILP when long latency instructions flow through the pipeline, and there are not enough independent instructions available to keep the processor busy. We classify long latency instructions as either load instructions, which have a variable length latency, or fixed length long latency instructions (e.g., DIV, MUL, SQRT). Data dependent instructions will stall behind these long latency instructions, potentially creating one or several critical paths through a portion of the program.

These long latency instructions and critical paths can eventually cause the fetch unit to stall because the buffers will fill up and prevent additional instructions from flowing into the pipeline. Therefore, breaking these true dependency chains can increase ILP by (1) reducing the lengths of critical paths through a program, and (2) preventing the fetch unit from stalling.

Value prediction is an approach that breaks true data dependency chains by predicting the resulting value for an instruction, and by allowing dependent instructions to use this predicted value as a source value. This allows the dependent instructions to execute in parallel with the long latency instructions, reducing the lengths of the critical paths through a program. In this paper, we call an instruction that produces a value prediction for its result register a *producer* of value prediction, and a dependent instruction that uses a predicted value as an input operand will be called a *consumer* of value prediction.

Prior studies on value prediction have applied value prediction to all instruction types or just to load instructions. A few studies have even broken the prediction up based on instruction type. What is missing from this prior research is a mechanism to indicate the *importance* of predicting an instruction. For best performance, the processor should concentrate on using value prediction for important instructions on the critical path. The goal is to speculate on operations with large gains and small losses even when confidence in that prediction is low, and only speculate on operations with lower gains and larger losses when the confidence is high. Moreover, accurate confidence prediction is needed to guide when and where to use value prediction. Value prediction should not degrade a processors performance, even in the presence of expensive misprediction penalties, if the appropriate confidence level is used.

The paper is organized as follows. Section 2 provides a description of the baseline value prediction architecture we used in this study. Section 3 describes the methodology used to gather the results for this paper and the baseline architecture configuration. Section 4 presents our selective value prediction techniques and evaluates their performance. Finally, Section 5 summarizes the contributions of this work.

2 A Value Prediction Architecture

This section describes the value prediction architecture used in this paper, and how value prediction interacts with the processor pipeline.

2.1 Value Prediction Table

Several architectures have been proposed for value prediction including last value prediction [19, 20], stride prediction [11, 14], context predictors [26], and hybrid approaches [30, 22].

This study uses a hybrid value predictor to evaluate the effects of selective value prediction. The hybrid value predictor is modeled after those proposed in [30, 2, 22, 25]. It is a hybrid between Stride and Context predictors. When simulating the value predictors, we update the values and strides speculatively, and repair an incorrect update in the commit stage. In addition, confidence counters are used to guide when to use the prediction information. These are updated in the write-back stage once the outcome of the prediction is known.

2.1.1 Stride

A *stride* predictor [6, 10, 26] keeps track of not only the last value brought in by an instruction, but also the difference between that value and the previous value. This difference is called the stride. The predictor speculates that the new value seen by the instruction will be the sum of the last value seen and the stride. We chose to use the two-delta stride predictor [10, 26], which only replaces the predicted stride with a new stride if that new stride has been seen twice in a row. Each entry contains a tag, the predicted value, the predicted stride, the last stride seen, and a form of confidence counter.

2.1.2 Context

A *context* predictor [26, 27, 30] bases its prediction on the last several values seen. We chose to look at the last 4 values seen by an instruction. A table called the VHT contains the last 4 values seen for each entry. Another cache, called the VPT, contains the actual values to be predicted. An instruction's PC is used to index into the VHT, which holds the past history of the instruction. The 4 history values in this entry are combined (or folded) using an xor hash into a single index into the VPT. This entry in the VPT contains the value to be predicted. The context predictor is able to keep track of a finite number of reference patterns that are not necessarily constrained by a fixed stride. This is the predictor used in [27], but is much smaller in size. Confidence counters are also used to guide prediction.

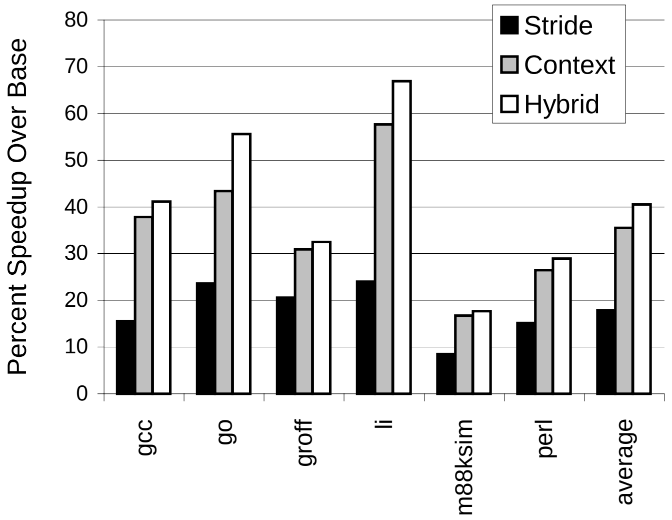

Figure 1: Percent IPC speedup over base architecture with perfect confidence prediction.

2.1.3 Hybrid

The hybrid predictor is identical to the ones used in [22, 25]. It is composed of one context predictor and one stride predictor, which are of the sizes described above. Prediction is guided by confidence counters. If both predictors hit (the confidence is above their predict threshold), then the value to be speculated is chosen from the predictor with the higher confidence. If both have the same confidence, a global *mediator* counter of correct predictions is consulted. Whichever predictor has the greater history of correct predictions is declared the winner. Preference is given to stride prediction in the case of a tie. The mediator counter is cleared every 100,000 cycles. The hybrid predictor combines the ability of the context predictor to recognize repeated values without a fixed stride, and the ability of the stride predictor to predict values that have not been seen, but that are a fixed stride apart.

2.1.4 Perfect Confidence

We also simulated the hybrid predictor with perfect confidence prediction. The Perfect predictor is the same as the hybrid predictor, except it only chooses to use the value prediction when the prediction is correct, and it chooses not to predict when the prediction is going to be incorrect.

Figure 1 shows the percent speedup achievable when using an infinitely sized hybrid value predictor with perfect confidence for the architecture we modeled. The context predictor achieves most of the performance, but also takes more physical area to implement. Figure 2 shows the percent prediction breakdown between the context and stride predictors that compose the hybrid predictor.

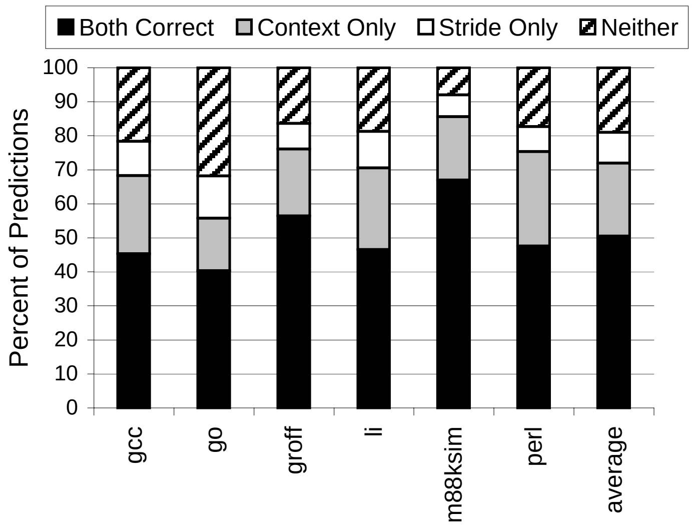

Figure 2: The percent of predictions that were accurately predicted by both the context and stride predictor, by only the the context predictor, followed by only the stride predictor, and the percent of predictions both predictors missed.

2.2 Value Prediction Processor Pipeline

When an instruction produces a value prediction, the value is inserted into the instruction's allocated physical register. This value will then be seen and consumed (used) by subsequent instructions. The predicted instruction still takes its normal path of execution for a non-speculative instruction. When the predicted instruction's real value becomes available, it is checked against the predicted value for misspeculation.

In the fetch stage of the processor, the value prediction table is accessed for the range of PCs being fetched. The value table lookup could potentially take multiple cycles, and needs to complete by the time the instruction enters the register rename stage. Efficient techniques, like those proposed in [13], are needed to handle multiple value predictions, but modeling this was beyond the scope of this paper.

After an instruction is decoded, it then enters the register renaming stage, where the instruction is allocated a physical register. The allocated physical register is normally used to hold the result value for an instruction, but we also use it to temporarily hold the predicted value for an instruction. If a value prediction was found in the table, the predicted value is stored into the physical register, to be potentially consumed by other instructions. Once the instruction finishes executing and the real result value is available, it will overwrite any predicted value in its physical register. In this design, we modified the physical register to contain 4 additional bits. The first bit indicates whether the physical register contains a real value or a predicted value. The remaining three bits (low, medium, and high) indicate the value prediction confidence the processor has assigned to the value stored in the physical register. The use of these three confidence bits will be described in Section 4.

During the issue stage of the processor, instructions are issued to reservation stations when a free reservation station for a functional unit becomes available. Instructions are then executed from the reservation stations when their operands become available. When an instruction is inserted into a reservation station, the instruction reads its available inputs from the register file, storing the four additional bits – whether the value is real or predicted and the 3 confidence bits – along with each value in the reservation station.

Instructions are scheduled to execute from the reservation station, when there is an idle functional unit. The input operands of an instruction in the reservation station can be in one of the following states (1) ready, (2) pending, (3) pending with value prediction. When choosing which instruction to execute, priority is given to those instructions whose operands are ready. If no instruction in the reservation station is ready to execute and there is an idle functional unit, we try to find an instruction to schedule using a predicted value. An instruction is a candidate for executing with value predicted operands if all input operands have either resolved or have predicted values. In addition, all of the predicted values have to have the correct confidence (see Section 4). Otherwise, the instruction is not a candidate for execution.

When an instruction consumes a predicted value, this means that the instruction producing this value has not completed. We keep a *use bit* in the reorder buffer entry of the producer instruction. In the reservation station, the consumer instruction already keeps track of the reorder buffer entry of the instruction producing the result value. When an instruction uses a predicted value it sets the use bit in the reorder buffer entry for the producer of the value to indicate that an instruction has used it. Once the producer finishes execution, it checks the use bit to see if its prediction has been consumed. If so, it will need to compare the predicted value to the real value to see if misprediction recovery is necessary. This means that when an instruction is mispredicted, recovery will only be necessary if that prediction is actually consumed by another instruction.

When an instruction that has consumed a value prediction finishes executing, it stores its computed value in its physical register. The value is also broadcast to all reservation stations. This allows results from speculative values to propagate through the pipeline, potentially allowing the execution of instructions well down the dependency chain to execute in parallel with the stalled instruction(s) at the head of the chain.

A value misprediction is discovered when an instruction finishes execution, and its reorder buffer entry indicates that it is a producer of a consumed value prediction, and the predicted value differs from the real value.

2.2.1 Misprediction Recovery

When a data misspeculation occurs, our recovery mechanism flushes all the instructions out of the reorder buffer after the misspeculated instruction, and *refetches* the instructions from the cache starting at the next instruction after the misspeculated instruction. This is identical to the miss recovery approach used for branches. This form of recovery should be efficient to implement since it uses the same mechanism and data paths as branch misprediction recovery. This is a relatively conservative model for misprediction recovery, but ensures that our techniques work even in the presence of a high recovery cost.

2.3 Choosing Instructions to Value Predict

In this paper, we will examine results using a reservation station with 256 entries. We actually model having three separate reservation stations, one for branches, the other for loads, and the final one for all the other instruction types. The buffers are searched in this order looking for instructions to execute. When performing instruction scheduling, the stations are searched from oldest instruction to newest looking for instructions to allocate to a free functional unit.

While performing this search, pointers are kept to instructions that are prime candidates for value prediction. After all ready instructions are issued, if there are still free functional units available, these prediction candidates will start executing if their input operands have the correct confidence threshold bit set (low, med, or high).

For an architecture which uses selective reexecution for its misprediction recovery, it may be advantageous to propagate the confidences during speculative execution. Selective reexecution, would only reexecute the instructions directly and indirectly dependent upon the mispredicted instruction. To propagate the confidences, when an instruction uses a predicted value its own register entry is also marked as speculative, and its confidence is assigned the lowest of the confidences used for the speculative values used to execute the instruction. During instruction scheduling this information can be used to give priority to instructions with non-speculative input values. Propagating these confidences does not provide benefits for the misprediction model we use in this paper, because if the value is mispredicted then the whole pipeline has to be squashed.

3 Evaluation Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [3], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, branch misprediction, or load misspeculation.

Program	Input	# instr exec (M)	# instr fastfwd (M)	256 Entry RS Base IPC
gcc	1cp-decl	1041	500	1.7
go	5stone21	32699	500	1.9
groff	man page	52	0	2.8
li	ref	18089	500	2.6
m88ksim	ref	76271	500	3.5
perl	scrabbl	28243	500	2.5

Table 1: Program statistics for the baseline architecture. Base IPCs are shown when using 256 reservation stations.

To perform our evaluation, we collected results for the SPEC95 benchmarks and one C++ program: `groff` (a troff text formatter). The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers. We compiled the SPEC benchmark suite under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

Table 1 shows the data set we used in gathering results for each program, the number of instructions executed to complete the program (in millions), the number of instructions fast forwarded through before starting our simulations (in millions), and the baseline architecture IPC using 256 entry reservation stations. We used the `-fastfwd` option in SimpleScalar/Alpha 3.0 to skip over the initial part of execution. Results are then reported for simulating each program for 100 million committed instructions.

3.1 Baseline Architecture

Our baseline simulation configuration models a future generation microarchitecture. We've selected the parameters to capture three underlying trends in microarchitecture design.

First, the model has an aggressive fetch stage, employing a variant of the collapsing buffer[7]. The fetch unit can deliver two basic blocks from the I-cache per fetch cycle, but no more than 8 instructions total. If future generation microarchitectures wish to exploit more ILP, they will have to employ aggressive fetch designs like this or one that is comparable, such as the trace cache [24].

Second, we've given the processor a large window of execution, by modeling large reservation stations, reorder buffers, and load/store queues. Large windows of execution expose the ILP necessary to reach future generation performance targets. Our out-of-order processor can fetch up to 8 instructions per cycle, issue 16 operations per cycle from a unified reservation station, and has a 256 entry reservation station. Loads in the baseline architecture can only execute when all prior store addresses are known. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 3 cycles.

Third, processor designs are including larger on-chip and off-chip caches. Larger caches create longer load latencies for hits in the L1 data cache. The Alpha 21264 processor has a 3 to 4 cycle first level data cache latency [17]. The processor we simulated has a 32K 2-way associative instruction cache and a 32K 2-way associative data cache. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and non-blocking with four ports. The latency of the data cache is 3 cycles, and the cache is pipelined to allow up to 3 new requests each cycle. There is a unified 2nd level 512K 4-way associative cache with 64 byte blocks and a 12 cycle cache hit latency. A 2nd level cache miss has a 108 cycle miss penalty, making the round trip access to main memory 120 cycles. We model the bus latency to main memory with a 10 cycle bus occupancy per request. There is a 32 entry 8-way associative instruction TLB and a 32 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

The branch predictor is a hybrid predictor with an 8-bit gshare that indexes into 16k predictors + 16k bimodal predictors [21]. There is an 8 cycle minimum branch and value misprediction penalty. The processor has 10 integer ALU units, 4-FP adders, 2-integer MULT/DIV, and 2-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are pipelined to allow a new instruction to initiate execution each cycle.

4 Selective Value Prediction

This paper examines the performance of value prediction in light of realistically sized value prediction tables. For many applications, these tables will experience significant capacity conflicts, lowering overall prediction coverage and prediction accuracy. We first examine changes to the prediction and confidence mechanisms that work well under these constraints, and then examine techniques to filter the number of instructions that either produce values into or consume values from the value tables, thus significantly reducing pressure on those tables. Throughout this section we will use the hybrid predictor described in section 2.1.3 for our value table.

4.1 Size and Sharing of 2nd Level Context Table

We begin by investigating how the size of the 2nd-level table (VPT) of the context predictor affects performance. Each entry of the context table stores the four most recent values seen. We examined several different hashing functions to index into the VPT, and present results for the best one we found. To compute the hashing function, each value is folded onto itself using an XOR to include all of its bits. The result is a value Id that is equal in size (in terms of bits) to the VPT table index. These four value Id's are then combined by shifting each value by twice its position in the value stream and XORing these values together. Figure 3 shows the results for this hashing function for VPT tables with 256, 1024, and 65536 entries with an infinite sized first level table (VHT). Results are shown with and without using the PC as part of the hash function into the context value table. We found that hashing too many of the PC bits resulted in a significant degradation in performance. The reason for this is that the value stream encountered by one instruction can be very similar to the values encountered by another instruction. An example of this occurs between two different load instructions that are traversing the same pointer list. One load instruction will initialize the VPT with its values, and the other load instruction can achieve 100% value prediction accuracy when traversing that same list.

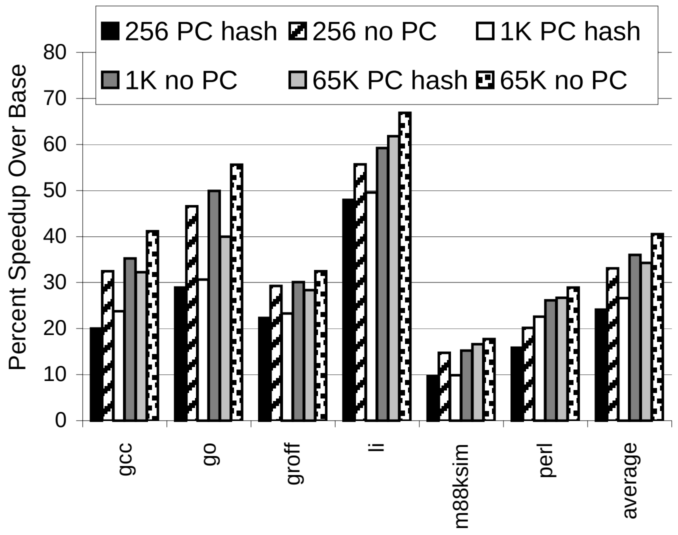

Figure 3: Performance for using a 256, 1024, and 65536 entry context value table. Results are shown when the PC is used as part of the hashing function (PC hash), and when it is not used at all (no PC).

The results in Figure 3 show that even a very small 256 entry context table can provide reasonable results for a hybrid predictor. In the rest of these results we used a 1024 entry context table for our hybrid predictor.

4.2 Value Confidence Prediction

A predicted value should only be used if the confidence associated with that value is above a given threshold. We examined several forms of confidence prediction and based our predictors on the confidence estimation techniques developed for branch prediction and multiple path execution [16, 15]. The relatively high misprediction cost of our recovery scheme will necessitate a confidence scheme that reacts quickly to mispredictions or disruptions in the table.

4.2.1 Confidence Saturating Counter

A confidence counter can be described using a set of six numbers (saturation threshold, low threshold, mid threshold, high threshold, miss penalty, increment bonus). The saturation threshold is the maximum value the counter can contain. When a value prediction is made, if the confidence of the prediction is equal to or above each of the three prediction thresholds (low, med, high), the corresponding threshold bit will be set in the physical register file as described in Section 2.2. When a misprediction occurs, the confidence counter is decremented by the miss penalty. Conversely, when a correct prediction occurs, the counter is incremented by the increment bonus.

We examined many different values for these four parameters, and found that for our conservative mispredict recovery mechanism, the counter needs to quickly turn off once it starts mispredicting before it can cause too much damage. The best counter configuration for this recovery scheme is (15,3,7,15,7,1), which has a saturation value of 15. When a predicted value is read from the prediction table and inserted into the physical register, the low confidence bit will be set when the counter is 3 or above, the medium confidence bit will be set when 7 or above, and the high confidence bit will be set with the counter equals 15. If an incorrect prediction occurs, the saturating counter is decremented by 7. If the prediction is correct, the predictor is incremented by 1.

Figures 4 and 5 show the performance when using an infinite sized first level stride and context tables for the hybrid predictor when different confidence thresholds are used for prediction. The results show that on average, a threshold of 15 correct predictions in a row yields the best performance for all instructions. If the prediction was incorrect one out of every 15 tries, this would result in 94% prediction accuracy, which is needed to tolerate the misprediction penalty. The results also show that each program on average does best with a different degree of confidence. The optimal threshold also varies by instruction type, as evidenced by the differences between the two graphs. The results show that for `gcc`, load instructions only need a threshold of 7, but when predicting all instruction types, better performance comes from a confidence threshold of 15.

4.2.2 Confidence History Counter

The use of history information, similar to local branch histories [18, 31], can be used to provide history confidence for value prediction, and thus increase the reliability of the predictors.

To model this architecture, each value prediction entry in the value table contains an N-bit history register, keeping track of the accuracy of the last N value predictions. A 0 is shifted into the history register if the last prediction was incorrect, and a 1 if it was correct. The history register is then used to access a table of saturating counters to produce the confidence. This allows instructions with the same prediction history to share confidence counters. The history predictor performed best when using the N-bit history register to index the table of counters directly. As with the context VPT index, XORing this N-bit counter with too many PC bits resulted in worse performance, because it did not allow instructions to share their history effectively.

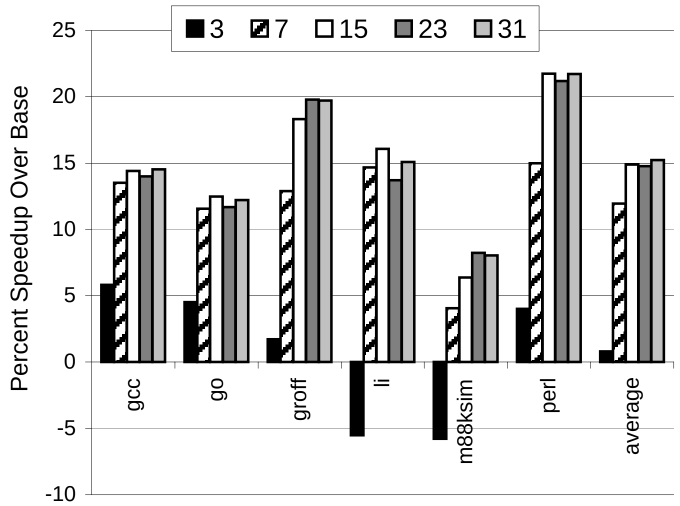

Figure 4: The effect of using different confidence thresholds (3, 7, 15, 23, and 31) to guide when to use values predicted by all register defining instructions.

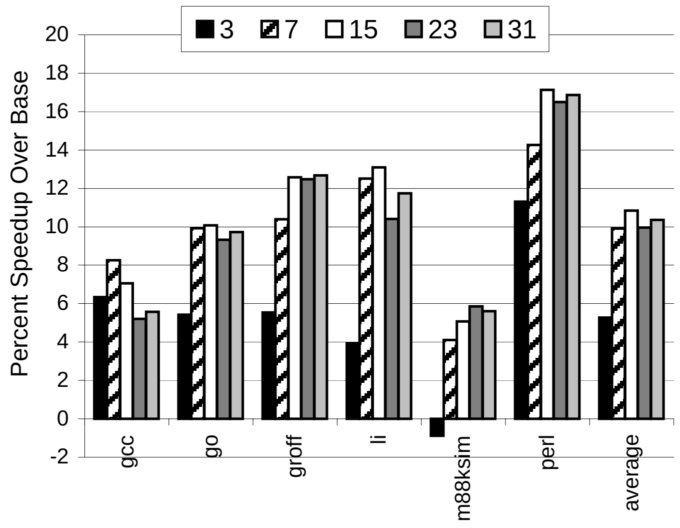

Figure 5: The effect of using different confidence thresholds (3, 7, 15, 23, and 31) to guide when to use values predicted by load instructions.

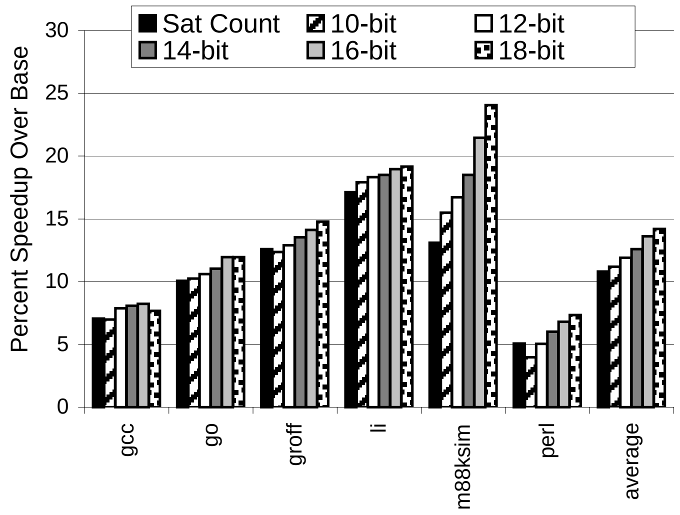

Figure 6: Performance for different history table sizes (1024 to 256K entry), compared to giving each instruction its own saturating counter with a threshold of 15.

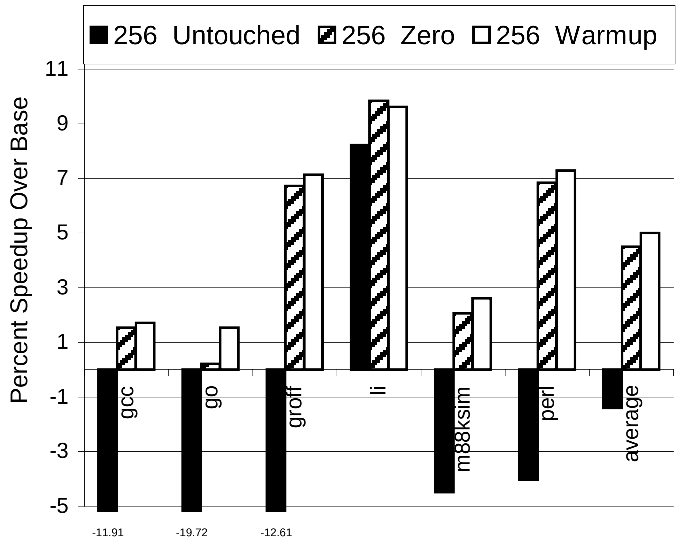

Figure 7: Initializing confidence on value table replacement for a value table with 256 entries.

Figure 6 shows the performance for predicting only load instructions when storing an N-bit (where N is 10, 12, 14, 16, and 18) prediction history. The first bar shows the performance obtained using a saturating counter with a threshold of 15, as a comparison. The results show that history confidence can provide a large increase in speedups, especially for *Li*, but at a much higher cost than the saturating counter. Since the goal of this study was to examine the effects of value prediction for realizable sizes, the remaining results are shown using the saturating counter described in Section 4.2.1 with a high-bit prediction threshold of 15.

4.2.3 Coping with Value Table Capacity Misses

To provide high performance with smaller value prediction tables, we want to minimize unnecessary replacements in the table, and deal appropriately with those replacements when they occur. In the value prediction table we modeled, we used 2 additional small counters to help fight against the loss of prediction due to capacity thrashing.

The first counter is called the replacement counter and is used to to provide hysteresis. The replacement counter is incremented on a correct prediction, and decremented on an incorrect prediction. The counter is also decremented when another instruction attempts to use that entry in the prediction table. This counter will allow highly predictable instructions to stay in the table.

We model a 4-way associative table, with LRU + replacement counter replacement. An instruction that misses in the value prediction table will be inserted into the value table during its write-back stage. When inserting an instruction, the LRU element from the indexed set is examined for replacement. If the replacement counter for that entry is below the replace threshold, then the new instruction is inserted into the table. Otherwise, it is not - but the replacement counter the the LRU entry is decremented as mentioned above to promote fairness.

The other counter we added delays updates to the confidence counter and updates into the 2nd level context table (VPT) until after the instruction has warmed up its prediction information. If this is not used, then the first few predictions an instruction makes will most likely be incorrect, and the confidence counter will end up set to a strong *not predict* state. To counteract this we added a *warm-up* counter to each entry. This counter does not allow a prediction to be made and does not allow modification of the confidence counter and VPT until after the instruction has hit in the value table a certain number of times. Our stride predictor will have its stride initialized after two hits in a row in the table, and the context predictor will have its history initialized after four hits in a row. The warm-up counter is set to 0 on replacement, and is incremented every time an instruction hits in the value table. After the warm-up counter has reached the threshold value of 3 for stride and 5 for context prediction the instructions can now start providing predictions, and the context predictor can now start updating its VPT. When an instruction is inserted into the value table its confidence value is initialized to 14, where 15 is the high predict threshold.

Figure 7 shows the effect of (1) not changing the confidence counter at all on a replacement, (2) setting the confidence to 0 on replacement, and (3) using the warm-up counter with an initialization of 14 for the confidence on replacement. Results show that programs like go, which have a lot of capacity constraints, can benefit from using a warm-up counter. Larger improvements were seen when we modeled smaller tables where there were more capacity prob-

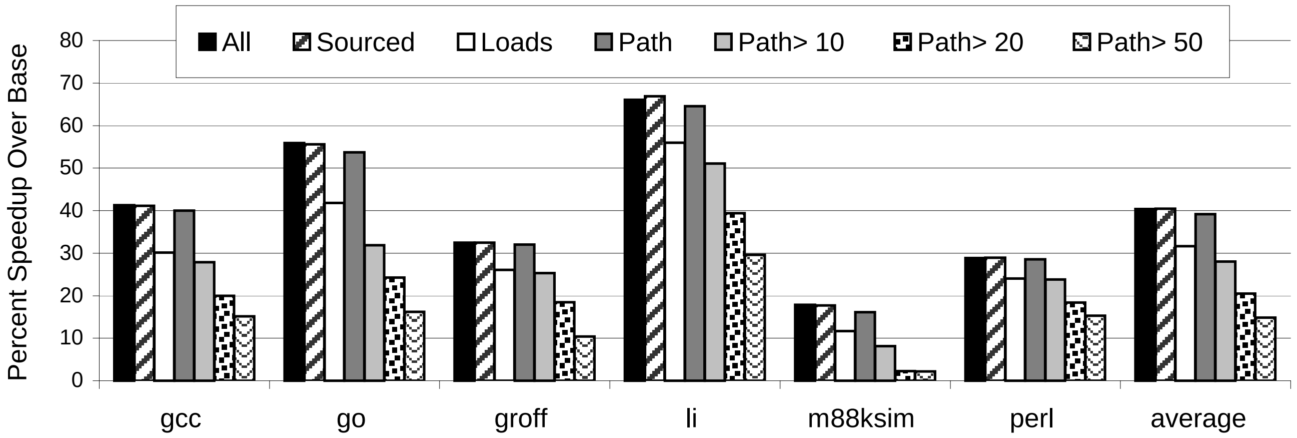

Figure 8: Performance achieved from filtering which instructions to put into the value prediction table.

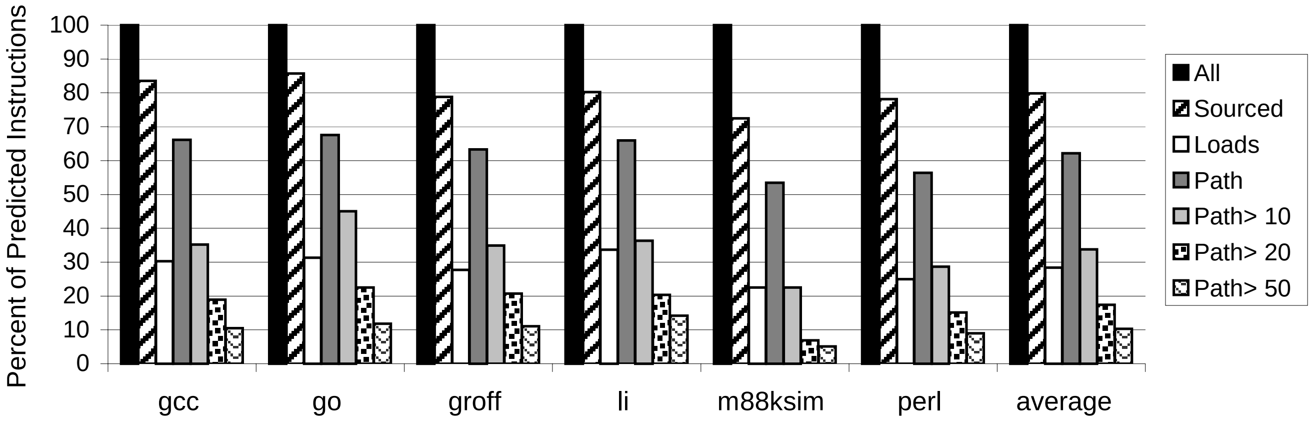

Figure 9: The percent of register defining instructions that were actually predicted using a given filtering technique.

lems. In addition to the replacement and warm-up counter, additional filtering techniques used in branch prediction may also be beneficial for value prediction and warrant further investigation [8, 9].

4.3 Filtering the Producers of Predicted Values

A significant number of instructions define register values which potentially can be used by other instructions. One mechanism used to reduce pressure on the prediction tables is to predict fewer instructions (more specifically, to allow fewer instructions to write into the tables). For a reasonably sized value prediction table to obtain performance, the number of instructions using the table to store their values for later use must be filtered. If we filter the right instructions we can still predict the instructions that impact performance, while ignoring those that do not.

Figure 8 shows the speedups obtained when we filter the instructions that are put into an infinitely sized table using perfect confidence prediction. In addition, Figure 9 shows the reduction in the number of predicted values made, based on the filtering techniques from Figure 8. An infinite sized table will not have any capacity constraints, so these two graphs show the trade-offs that each filter is making in terms of performance and pressure on the prediction table.

The first bar in Figure 8 shows the speedups when all register defining instructions are allowed to compete for entries in the value prediction table. The next bar (Sourced) shows the speedup achievable when we only allow entries to be allocated to those instructions that define registers which are actually used by another instruction in the current instruction window [25]. The *Loads* filter only allows the storage of load instructions into the table. To evaluate the full potential of selectively inserting the most important instructions, we also examined the performance of limiting the instructions that are inserted into the value table to only those instructions on the critical path.

To estimate the critical path, we keep track of the longest dependency chain in terms of cycles during execution. The dependency chain starts from an instruction currently being executed and includes instructions all the way up through the decode stage. The longest path, when tracked dynamically in this way, can be very different than what one normally thinks of as the critical path through the program. There are actually many critical paths, and correctly value predicting an instruction on one of these paths may leave another path as the critical path in the next cycle. To make matters more complicated, value prediction may not help in cases where the critical path is broken by other factors in the execution of the program. This can happen when dependent instructions in the critical path are scheduled too far apart by the compiler and therefore are not in the active instruction window at the same time. Also, the fetch unit may stall due to an I-cache miss or to insufficient available buffers. If the fetch unit stalls for long enough, the critical path to an instruction currently not in the pipeline can be broken, since the long latency instruction may find enough time to complete its execution. In the programs we examined, the longest critical paths often lasted only a few hundred cycles because of the above factors.

The last four bars in Figure 8 show the results when value predictor table storage is allocated only to instructions that *start* a longest path in the current instruction window. Each cycle, the instruction being executed that starts the current longest path is marked. These longest path instructions are then inserted into the value table in the write-back stage, if they were not originally found in the value table. The first of these bars shows results when allocating all instructions that start a path, while the other three bars show results when marking only those instructions that start a path when the longest path length is greater than 10, 20, and 50 cycles respectively. Figure 9 shows the reduction in the number of predicted values made, based on the filtering techniques from Figure 8. The results show that only predicting load instructions can provide 75% of the potential value prediction performance while only dealing with 30% of the possibly predicted instructions. Concentrating on the longest path can achieve about the same performance as predicting all instruction types, with a nearly 40% reduction in the number of predicted values.

The prior results are for an infinite size value table where conflicts do not occur, and the best we can hope for in filtering is to approach ideal performance; however, on a finite table, filtering reduces conflicts and thus can increase overall performance. Figure 10 shows the performance for filtering table inserts based on load instructions, instructions on the longest path, and all register defining instructions that were sourced. Inserting all sourced instructions creates high capacity constraints, and thus concentrating on the potential critical path is an effective solution. Srinivasan and Lebeck [28] and Bahar *et. al.,* [1] recently focused on trying to identify important long latency load instructions, and their approach could be used to help guide which load instructions to insert into the value prediction table.

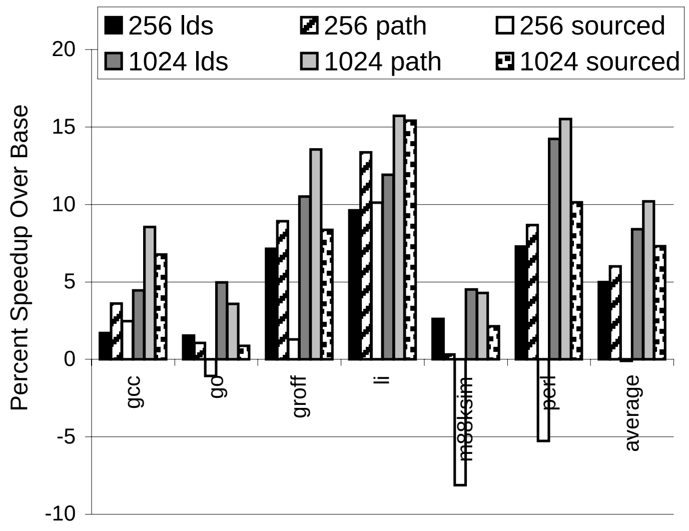

Figure 10: Performance for filtering table inserts based on load instructions, instruction on the longest path, and all register defining instructions that were sourced.

4.4 Finding the Important Consumers

The prior section focused on filtering which instructions to *insert* into the value prediction table. Filtering which instructions *use* a predicted value can also be advantageous. To provide filtering of uses, we use the three confidence bits (low, med, and high) described in section 2.2. These confidence bits provided along with the value to be predicted, can be used to guide whether an instruction should use the value for speculative execution.

We examined several heuristics for guiding the appropriate confidence level to use for consuming predicted values. Two heuristics that worked well were based on the longest path analysis from the prior section. The heuristic chooses to use values with lower degrees of confidence (low and med bit set) for instructions that are on the longest path. When the scheduler chooses an instruction to execute using this heuristic, it searches the longest path dependency chain for an instruction with predictable inputs that can issue. Value predictions are made in this order, and if there is still issue bandwidth left over, the remaining instructions become candidates for value prediction, proceeding in order, starting from oldest first. Figure 11 shows the performance of giving priority to longest path instructions for speculative execution, and the benefit of using a lower threshold for those instructions. Results are shown when only load instructions are inserted into the value table, as described in the prior section. *Fifo* gives the performance when instructions are chosen to execute speculatively with predicted values from the oldest instruction to newest using a confidence threshold of 15. For the *Path* heuristic, an instruction on the longest path will consume a predicted value if the value's medium

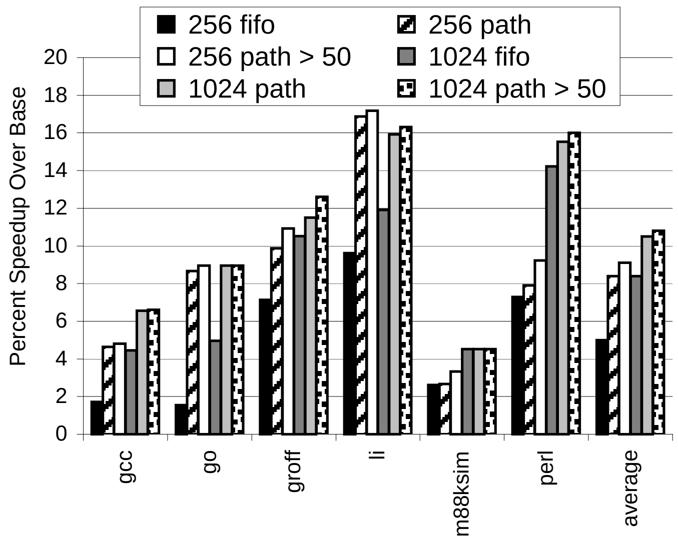

Figure 11: Performance for using the low, med, and high confidence bits to guide when to use a predicted value. Results are shown for using lower confidence for instructions on the longest path. Only load instructions are inserted into the value table.

confidence bit is set (prediction threshold of 7), and all other instructions use a threshold of 15. For the *Path* $>$ *50* heuristic, we use a threshold of 3 (low-bit) for all instructions on a path over 50 cycles long.

The speedups increase by an absolute 11% over using only high confidence for all instructions when concentrating on the longest path for both squash and reexecute. The longest/critical path scheduling algorithm is ideal, but would be difficult to build. For future work we are concentrating on using the compiler/profiler to find critical path instructions, which will benefit from value prediction and other optimization [29].

In Figure 11, the performance benefit from using the path heuristic is smaller for the 1024 entry table than the 256 entry table. Prior research has shown that 60 to 80% of the instructions always produce very predictable or unpredictable values [5]. Once there are no more capacity problems, the confidence for these instructions will converge quickly, and provide a very accurate confidence for the value. But for smaller tables, which will have capacity problems, the confidence needs to be reestablished when an entry is inserted into the table. This is one of the reasons why better performance benefit is seen for a 256 entry table than the 1024 entry table.

Multiple levels of prediction can also be of benefit during instruction cache misses. When this occurs, the cost of a value misprediction is much less, since the fetch unit is the bottleneck instead of the program's critical path. If the fetch unit is stalled, a value misprediction may not accrue any additional penalty, even with a large misprediction penalty. If the I-cache is allowed to process requests from a mispredicted path while waiting for data to come back from a higher level of memory, a misprediction may not cost anything since refetching the instructions would result in using the fetch unit that would have otherwise been sitting idle. In this case, it is very beneficial to be overly aggressive about consuming predicted values. A heuristic can be implemented to keep track of when the I-cache is stalled due to a miss, and during this time use the low-bit confidence threshold. Examining this and other techniques for guiding which confidence threshold should be used when consuming instructions is part of future research.

Note that value prediction is limited by the in-order committal of instructions. Our results show that we can provide benefits by concentrating on the longest path, but the path must stay in the processor, which uses reorder buffer entries. Breaking a chain with correct value prediction will not clear instructions out of the reorder buffer until the instruction at the start of the chain commits. Therefore, to obtain large gains from a value prediction architecture, the reorder buffer should be as large as possible, even if the number of reservation stations is small.

5 Summary

This paper presents techniques to intelligently choose when to use value prediction, and which instructions to value predict.

The performance gap between perfect confidence and ordinary confidence prediction shows that accurate confidence prediction is key to obtaining speedups for value prediction. We showed that even in the presence of high misprediction penalties, confidence counters can be used to provide speedups of 10%. In addition, we showed that history confidence can reduce this performance gap. When gathering the history and 2nd level context results we found that confidence and values produced by one load instruction can be used by another load instruction, providing very accurate predictions.

A fixed sized storage device like a value prediction table will have capacity problems, especially for large commercial applications. In order to benefit from value prediction for these workloads, the value prediction table needs to use a replacement counter to keep highly predictable instructions in the value table. In addition, it can be beneficial to use a warm-up counter, to warm up an instruction's value prediction information before updating its confidence and using it for prediction.

To reduce some of these capacity constraints it is useful to heavily filter which instructions are put into the value prediction table. We showed that concentrating on instructions from the longest path in the instruction window increases performance. The results also show that only concentrating on loads, for the workload examined, is a reasonable filtering approach since load latencies will be responsible for most of the critical paths in integer programs, as pointed out in [27]. Prior research has shown that value profiling techniques [4, 12, 23] can be used to accurately find predictable instructions, and can be used for filtering table entry allocation. What is harder, and probably more important, is the need to concentrate on storing instructions that can potentially provide significant gains, even if those instructions are hard to predict. We are currently looking at using critical path profiling to find such instructions [29].

Determining which instructions should consume a predicted value is just as important as determining which instructions to store in the value table. An incorrectly predicted value does not cause any misspeculation unless that value is used by a dependent instruction. We showed that it is beneficial for instructions on the longest path to use predicted values with a lower degree of confidence than those instructions not on the longest path. Prior value profiling work concentrated on identifying which instructions should *produce* predicted values. Future profiling work should in addition concentrate on identifying which instructions should *consume* predicted values and the degree of confidence to use for those values.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was funded in part by NSF CAREER grants No. CCR-9733278 and No. MIP-9701708, NSF grant No. CCR-980869, and a grant from Compaq Computer Corporation.

References

[1] R.I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design*, August 1998.

[2] B. Black, B. Mueller, S. Postal, R. Rakvie, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. In *12th International Conference on Supercomputing*, June 1998.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling. In *30th International Symposium on Microarchitecture*, 1997.

[5] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1999.

[6] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[7] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[8] K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *31st International Symposium on Microarchitecture*, December 1998.

[9] A. N. Eden and T. Mudge. The yags branch prediction scheme. In *31st International Symposium on Microarchitecture*, December 1998.

[10] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, July 1993.

[11] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[12] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *30th International Symposium on Microarchitecture*, 1997.

[13] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th Annual International Symposium on Computer Architecture*, 1998.

[14] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.

[15] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[16] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th International Symposium on Microarchitecture*, December 1996.

[17] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprosessor architecture. In *International Conference on Computer Design*, December 1998.

[18] K.L. Lick. *Hybrid Branch Prediction Using Limited Dual Path Execution*. PhD thesis, University of California, Riverside, December 1996.

[19] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pages 138–147, October 1996.

[20] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[21] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[22] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, 1998.

[23] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Profile guided load marking for memory renaming. Technical Report UCSD-CS98-593, University of California, San Diego, July 1998.

[24] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture*, December 1996.

[25] 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*, 1998.

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

[27] Y. Sazeides and J. E. Smith. Modeling program predictability. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[28] S.T. Srinivasan and A.R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *31st International Symposium on Microarchitecture*, December 1998.

[29] D. Tullsen and B. Calder. Computing along the critical path. Technical report, University of California, San Diego, 1998.

[30] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[31] T.Y. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.