Appears in MICRO-33

Compiler Controlled Value Prediction using Branch Predictor Based Confidence

Eric Larson and Todd Austin
Electrical Engineering and Computer Science
University of Michigan
{larsone,austin}@eecs.umich.edu

Abstract

Value prediction breaks data dependencies in a program thereby creating instruction level parallelism that can increase program performance. Hardware based value prediction techniques have been shown to increase speed, but at great cost as designs include prediction tables, selection logic, and a confidence mechanism. This paper proposes compiler-controlled value prediction optimizations that obtain good speedups while keeping hardware costs low. The branch predictor is used to estimate the confidence of the value predictor for speculated instructions. This technique obtains 4.6% speedup when completely implemented in software and 15.2% speedup when minimal hardware support (a 1 KB predictor table) is added. We also explore the use of critical path information to aid in the selection of value prediction candidates. The key result of our study is that programs with long dynamic dependence chains benefit with this technique while programs with shorter chains benefit more so from simple selection methods that favor optimization frequency. A new branch instruction that ignores innocuous value mispredictions is shown to eliminate unnecessary mispredictions when program semantics aren't violated by confidence branch mispredictions.

1. Introduction

High performance computing requires high throughput instruction execution. Control and data hazards impede this goal, preventing programs from using all of the available resources. Research has shown that the outcomes of many instructions are highly predictable leading to a growing body of work in value prediction. Values are predicted for selected instructions, breaking output dependencies that allow dependent instructions to execute concurrently. This technique breaks data hazards, extracting additional instruction level parallelism (ILP) and increasing the number of instructions executed per cycle (IPC). There are several different types of value predictors: last value predicted [10], stride [3], context-based [19], or hybrid [3].

To date, most value prediction research has focused on hardware-based schemes. In these predictors, the prediction table is indexed very early in the pipeline using the PC. Typically, the prediction is known in the fetch and/or dispatch stages, such that predicted values can be forwarded immediately to dependent instructions stalled waiting for inputs. Speculative instructions must delay retirement until the value prediction is verified. If the prediction is correct, any dependent instructions that have executed can retire immediately. If the prediction is wrong, the errant instruction and all dependent instructions must be re-executed. The high cost of value mispredictions (tens of cycles in proposed microarchitectures) limits the application of value prediction to only the most highly predictable instructions.

To increase the scope of value prediction, it is possible to employ confidence mechanisms to reduce the probability that a value is mispredicted [7]. A confidence mechanism is a meta-predictor, it predicts if the prediction of the value predictor is correct. If the confidence mechanism is not very confident that a prediction will be correct, it will not predict the value, thereby avoiding an expensive misprediction. Many instructions with low overall prediction accuracies do have highly predictable confidence, making it possible to leverage more predictions without incurring more mispredictions.

One technique to measure the confidence of value prediction is to use a saturating counter [3] where the counter is incremented on every correct prediction and subtracted or reset to zero when an incorrect prediction occurs. If the value of the counter is greater than some threshold, it will be considered a high-confidence prediction and will be subject to value prediction. Another scheme to measure confidence is to base predictions on the past *n* predictions [3]. This implementation allows for highly confident predictions on instructions that exhibit a specific pattern of correct and incorrect predictions. This scheme is very similar to pattern-based branch predictors.

Hardware-based value predictions, while efficient, have significant hardware costs. Value predictors are often on the order of cache sizes before they attain respectable accuracies and coverage. In addition, misprediction recovery mechanisms, especially partial re-execution techniques [20], are quite complex. Efforts have been made to address these costs through software-based value prediction optimizations. For these techniques, the compiler is responsible for locating prediction candidates, implementing the prediction optimization, verifying the prediction, and providing fixup code in case of a misprediction [5]. Typically, the compiler chooses value prediction candidates based on profiling data.

To date, software-based value prediction techniques have only rendered small speedups compared to hardware-based techniques. The primary reason for this disparity is because software-based techniques have not employed confidence mechanisms that can increase the coverage and accuracy of value prediction. Without a confidence mechanism, software-based techniques must choose between either low accuracy or few candidates - neither choice provides much opportunity for performance gains. Moreover, candidates are selected using profiles from a particular input set; choices made for a particular input set may hinder performance for another.

Another factor that limits software-based speedups is the cost of applying the technique. Instructions must be added to implement predictions, update predictor tables, and fix up mispredictions. To limit these costs, most previous work has employed simple predictors such as last-value predicted or static stride. More powerful methods require too many instructions and consumes too much code and data space to be efficiently handled by software. Moreover, the additional instructions required to maintain software-based predictors increase register pressure and consume valuable memory bandwidth. One technique for solving both problems is to add explicit predict and update instructions to the instruction set architecture (ISA) [6]. The prediction state is completely stored in hardware and allows for more sophisticated context-based predictors. Candidate selection and misprediction recovery is still implemented by the compiler.

In this paper, we present an optimized approach to compiler-controlled value prediction. Our approach improves the performance of compiler-controlled value prediction while keeping hardware costs significantly lower than hardware-only based techniques. It also addresses many of the drawbacks endemic to compiler-controlled value prediction.

Low coverage and accuracy can be addressed by adding confidence. We implemented a confidence prediction mechanism using the underlying branch predictor. The branch predictor has a choice of executing a section of code using the predicted value or the actual value. Based on previous history, the branch predictor will make an informed decision on which section of code should be executed. If a selected candidate turns out to be unpredictable, the branch predictor will predict that the non-speculative value should be used, resulting in performance comparable to the original code. Assuming a fixed accuracy, a scheme with confidence will have higher coverage and a higher potential for speedup.

To address overhead concerns, we propose improvements to the optimized instruction selection process. By selecting more gainful sites for selection, we can select fewer of them, thus limiting the overheads associated with software-based value prediction. Typically, the basis for choosing candidates is based upon the prediction accuracy and the number of times it is executed. While these are good characteristics, it does not tell the complete story. Research has shown [3, 12] that selecting candidates on the critical path is important to realize the maximum potential of value prediction.

Ideally, a long chain of dependent instructions should be split in half resulting in two chains that can be executed in parallel. Past work [4] has employed static analysis to identify instructions in the middle of these output dependence chains. This approach does not provide complete information regarding true critical paths, since it is difficult to tell where the performance degrading (e.g., long latency) instructions lie and how much of a particular chain actually resides in the instruction window. As a result, we analyze the dependence chains at runtime in the instruction window of a detailed processor simulator. Instructions are given scores based on how far they are to either end of a dependence chain. The final candidates are selected as a function of this critical path metric, optimization frequency, and optimization accuracy.

In all software-based value prediction techniques, a branch is used to validate the prediction. If a prediction is incorrect, the branch will re-direct the program to the fixup code where the computation proceeds with the non-speculative value. We use a similar approach when mapping confidence to the branch predictor. This approach introduces unnecessary branch misprediction when the branch predictor directs instruction fetch down the fixup path and the value prediction is correct. When the branch direction is verified to be incorrect (i.e., the value prediction was correct), the pipeline is flushed and instruction fetch is redirected to the value speculative code. Since the fixup path is always correct, regardless of the value prediction, this branch misprediction is superfluous. We create a misprediction tolerant branch that eliminates the misprediction recovery when this situation occurs.

The rest of this paper is organized as follows: Section 2 describes our three value prediction optimizations in more detail: compiler-controlled value prediction with confidence, critical path based selection, and a special branch that tolerates certain mispredictions. Section 3 describes our results. Section 4 describes related work in value prediction and Section 5 gives a conclusion and some ideas for future work.

2. Value prediction optimizations

This section details techniques used to improve compiler-controlled value prediction while keeping implementation costs low. The first technique described employs the existing branch predictor as a confidence mechanism. This allows for additional optimization candidates to be selected while keeping the accuracy high. The second technique is a critical path based selection technique which selects instructions based on their position in the critical path. The approach reveals to the compiler the best candidates to select for optimization. The final optimization is a misprediction tolerant branch that will remove unnecessary mispredictions when program correctness remains intact.

2.1 Branch based confidence

In order to reap the full benefits of value prediction, it is necessary to have a confidence scheme. This is difficult to implement directly in software since the data used to keep track of the previous history must be stored in data memory and additional instructions will be needed to extract and update this confidence data. To avoid adding this complexity, we propose using the branch predictor to measure the confidence of a value prediction. The primary benefit is that the information needed to make the confidence estimate and the misprediction recovery scheme is already built into the processor. In addition, a good branch predictor will provide a better measure of confidence than any reasonable software implementation could attain because the predictor implements powerful algorithms directly in hardware. For value predictions, it is necessary to record if a value was correctly predicted or not. To use the branch predictor for value prediction confidence, we map “value prediction was correct” to “not taken” and “value prediction was incorrect” to “taken” by bracketing value prediction sites with a controlling confidence branch.

BEFORE:	AFTER:
`r3 <- load X`	`r3 <- predict index`
`r4 <- add r3, r1`	`r9 <- load X`
`r5 <- sub r4, r8`	`r10 <- cmpeq r9, r3`
	`beq r10, fixup`
	`A: update index, r3`
	`r4 <- add r3, r1`
	`r5 <- sub r4, r8`
	`fixup: r3 <- mov r9`
	`br A`

Figure 1: Software Value Prediction with Confidence Example. Value prediction is applied to the initial load instruction. The branch predictor will try to predict if the predicted value (r3) matches the actual value (r9).

Our technique is illustrated by the example in Figure 1. The code before the optimization shows a chain of two instructions that are dependent on a load instruction. In this example, the optimization is applied to the load instruction. The destination register of the load is replaced by a register that is free during this segment (r9 in this case). The prediction is written into the original destination register of the load (details on the predict and update instructions are given later; assume for now that the destination register automatically contains a prediction for the load instruction). After the load, compare and branch instructions are added to validate the prediction. If the prediction is incorrect, the branch is taken where fixup code will place the proper value into the destination register. If correct, no fixup code is executed, and the program proceeds with the speculated value. The dependent instructions that follow the prediction remain unchanged.

The branch instruction gauges the confidence of the prediction. If the branch predictor predicts the branch not taken, it is predicting with high confidence that the value will be predicted correctly. In this case, execution of the dependent instructions can begin as soon as the predicting instruction has completed (and, if necessary, data from other sources is also ready). If the branch predictor predicts the branch taken, it predicts low confidence and speculates down the unoptimized path. The instruction on the unoptimized path is dependent on the actual value of the load instruction (or whatever instruction was being predicted) so progress will not be made until this instruction has completed. If the branch predictor was wrong, the penalty is the cost of a branch misprediction.

Our approach assumes the underlying machine contains a dynamic scheduling mechanism. Using this mechanism, we can dynamically select speculative or non-speculative code sequences knowing that the scheduler will extract all available instruction level parallelism. In a statically scheduled machine, extracting instruction level parallelism from the value speculative code sequence requires that the compiler commit to applying the optimization at compile time, otherwise, no benefits are found. For instance, the compare instruction in Figure 1 will stall on a statically scheduled machine, eliminating the benefit of predicting the load.

The example in Figure 1 uses the predict and update instructions that were proposed in [6]. These instructions are additions to the ISA that contain an index which refers to a specific entry in a hardware value prediction table. Predict instructions employ different indices to distinguish between different prediction sites. Each predict instruction also has a corresponding update instruction which informs the predictor of the last correct value. The predictor used by these instructions is transparent to the programmer, permitting the hardware predictor mechanism to be changed without impacting correctness of programs.

A static stride predictor [5] is used for all experiments. The static stride is determined by profiling. This predictor works well and is fairly straightforward to implement in software. Implementing more powerful predictors in software incur significantly more overhead in terms of instructions and memory operations.

The software-only version is applied in a similar manner. The major difference is where the predictions are stored. Without hardware support to store the predictions in a table, predictions must be stored in memory and/or registers. We chose to keep the predictions in memory until we enter the function where the instruction resides. At that time, the prediction is transferred into a register. Throughout the function, all updates are done by updating the register. At the end of the function, the value is stored back into memory for future use. This approach cannot always be applied as it requires that the prediction consume a register throughout the entire function. The optimization was applied at link-time when sufficient free registers were available (no spills). As a result, fewer sites were optimized in the software-only implementation. For some benchmarks, we were unable to apply the optimization to any site. We also explored loading and storing the prediction to and from memory instead of using a register but this resulted in consistent slows down due to the high number of additional memory operations.

Any instruction can be selected as a value prediction candidate. This includes floating-point instructions as well as integer instructions. Floating-point predictions are restricted to last-value only since there is no floating point immediate instruction to add a static stride. This is not a significant concern since there are very few floating point instructions that are predictable with nonzero static strides.

2.2 Value prediction candidate selection

Profiling data is used to select candidates for value prediction. We examine two techniques to determine which instructions are the best candidates for applying the optimizations. It it best to apply the optimization when the following equation holds true:

$$OPT_{accuracy} \times OPT_{benefit} >> OPT_{inaccuracy} \times OPT_{penalty}$$

Optimization accuracy and inaccuracy is only gauged at sites where the branch predictor indicates a high-confidence prediction. The optimization accuracy ($OPT_{accuracy}$) is computed (using profiling) as the number of times a value was correctly predicted divided by the number of high-confidence predictions. Similarly, the optimization inaccuracy ($OPT_{inaccuracy}$) is the percentage of incorrect high-confidence predictions (or 1 - $OPT_{accuracy}$). When a misprediction tolerant branch is implemented (described in the next section), the low-confidence case is equivalent to no optimization being applied at all. Therefore, we do not consider low-confidence predictions when computing the optimization inaccuracy. There is a small penalty due to the instructions added as a result of the optimization and increased pressure of the branch predictor. We ignore this effect in order to simplify the selection process.

The optimization penalty ($OPT_{penalty}$) is equivalent to a branch misprediction plus the overhead of executing the fixup code. In addition, there are issues such as increasing code size and branch predictor pressure that could adversely affect performance. Since the number of optimized candidates is small, the effect is a second order consideration and is not considered further.

The optimization benefit ($OPT_{benefit}$) is much more difficult to measure in an out-of-order microarchitecture. Instead of trying to derive an equation, we estimate the benefit using two techniques: a naive approach that assumes a fixed benefit each time the optimization is executed and a critical-path based approach which looks at how often the instruction is on critical paths in the processor instruction window.

In the naive approach, we assume the benefit is fixed with a value of one, making the overall benefit equal to the number of times the optimization was successfully applied. This fixed benefit model does not hold true in modern out-of-order machines as reducing the latency of different instructions will have varying levels of benefit. For example, a load instruction that misses the cache and heads a long chain of dependent instructions will have a higher benefit than a load instruction that hits in the cache and has no dependencies.

Previous research [3] has shown that applying value prediction to instructions on the critical path will have a higher benefit than applying it to instructions that are not on the critical path. In order to maximize the benefit of value prediction, it is desirable to split long chains of dependent instructions in half so the two halves can execute in parallel, improving performance. Our critical-path based selection technique estimates the benefit by looking at how often it is in the middle of a dependence chain.

An example of a dependence chain is given in Figure 2. From each instruction in the dependence chain, a "middle metric" is computed by taking the minimum distance from the endpoints of the chain. The profiled latencies of the instructions are used in calculating the distance to properly weight long latency instructions. This middle metric is an estimate of the parallelism gained by splitting the chain at that particular instruction. In the example, the subtract instruction has the best middle metric, and thus is the best candidate to apply value prediction to in this dependence chain. When the chain is split, each chain requires only four cycles to execute. Since both chains can be executed in parallel, the savings is three cycles over the original execution.

Obtaining the required data for profiling in this step requires the use of a detailed microarchitectural simulator that captures the state of the instruction window for each cycle of execution. Examples of tools that could perform this analysis include the SimpleScalar tool set [1] or Intel's VTune [24]. Addressing the performance of these analyses is beyond the scope of this paper, however, we believe techniques such as microarchitectural memoization [21] or sampling could be used to limit the cost of dependence chain profiling.

2.3 Misprediction tolerant branches

An important optimization scenario occurs when the value is predicted correctly but the branch predictor indicated it would be predicted incorrectly. With a normal branch instruction, the processor will speculate down the path using the actual value of the instruction assuming the prediction is incorrect. When the branch

Initial chain of dependent instructions	Lat.	Dist. Top	Dist. Bottom	Middle Metric
lda r0 <- 40(r29)	1	1	7	1
load r3 <- 0(r0)	2	3	6	3
sub r9 <- r5, r3	1	4	4	4
sll r13 <- r9, 8	1	5	3	3
add r8 <- r6, r13	1	6	2	2
cmpeq r11 <- r8, 3	1	7	1	1

Resulting chain 1	Resulting chain 2
lda r0 <- 40(r29)	predict r9
load r3 <- 0(r0)	sll r13 <- r9, 8
sub r9 <- r5, r3	add r8 <- r6, r13
	cmpeq r11 <- r8, 3
Latency: 4 cycles	Latency: 4 cycles

Figure 2: Example of computing the middle metric and the resulting chains. The left diagram shows the computation of the middle metric and indicates that the subtract instruction is the best choice. The right diagram shows the resulting chains when the initial chain is split at the sub instruction. This results in a potential savings of three cycles over the initial chain.

executes, a misprediction takes place and control is transferred down the path which uses the predicted value. However, using the actual value of the predicted instruction does not violate the semantics of the program. Useful instructions are thrown away when recovering from the misprediction. To solve this particular problem, we add a special branch instruction that will not recover from such a misprediction. We call this branch BEQIT, which stands for "Branch if EQual to zero Ignoring mispredictions down the Taken path". If the branch predictor speculates down the taken path, no misprediction will take place if the branch was mispredicted. The branch predictor is still updated with the correct choice so it can potentially make a better choice in the future. If the predictor speculates the branch is not taken but should be, a branch misprediction will be declared as normal (to fix the incorrect value). While it is still preferable to go down the optimized code path, there is no overall benefit to reaching it by taking a misprediction.

One advantage to the misprediction tolerant branch is that it is inexpensive to implement. It only requires some additional decoding logic and a few gates to prevent the processor from entering a speculative state when the branch is mispredicted in the taken direction.

3. Results

In this section, the results of our experimental evaluation are detailed. The initial experiment looks at the coverage and accuracies of a value predictor with and without confidence. This is followed up with simulation-based performance analysis. Next, we look at the effectiveness of software-only value prediction. The final experiments evaluate the utility of various selection mechanisms, and analyzes the performance impact of the misprediction tolerant branch.

3.1 Experimental framework

We obtained our results using a mix of SPEC95 and SPEC2000 benchmarks including integer and FP benchmarks. The benchmarks were compiled using the Compaq C (version 5.9) and Fortran (version 5.3) compilers under using full compiler optimization (-O4). The benchmarks used are listed in Table 1. The train input set was used during all profiling runs.

Table 1: Benchmarks used in simulation

Name	Fastfwd Cycles	Sim. Cycles	Base IPC (train)	Base IPC (ref)
art (2000 FP)	100 M	250 M	0.8988	0.9106
compress (95 INT)	none	51M	2.1864	1.4796
crafty (2000 INT)	100 M	250 M	1.7819	1.7524
equake (2000 FP)	100 M	250 M	2.6225	2.5963
go (95 INT)	100 M	250 M	1.5356	1.5478
m88ksim (95 INT)	none	150 M	2.3362	2.2220
mcf (2000 INT)	100 M	250 M	2.5588	2.5630
tomcatv (95 FP)[1]	none	85 M	2.3905	2.1885

1. For tomcatv, the reference set was used for profiling and will be referred to train throughout the paper and vice versa.

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [1], 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 4-way 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, or branch misprediction.

Initially, the benchmarks were run with SimpleScalar twice to obtain profiling information. In the first pass, the best static stride and average latencies were determined for each instruction that wrote to an output register. In this phase, all instruction latencies are assumed to be one cycle except loads and branches. The latency for each load is the average latency given an L1 cache hit is one cycle and an L1 miss is eight cycles. The average latency for branches is computed given that a correct prediction executes in one cycle and an incorrect prediction recovers in six cycles. In the second profiling pass, the predictability, optimization accuracy, and the middle metric was computed for each instruction. The predictability was determined by using an infinite sized static stride predictor to eliminate conflicts. Optimization accuracies were determined using a 16k gshare branch predictor to estimate confidence.

Statistics gathered from profiling were then used to select the candidates to apply the value prediction optimization. The optimization was applied using ALTO [16] - a link time optimizer that provides a set of classic compiler optimizations that can be applied to Alpha COFF object files. To normalize results, we disabled all ALTO optimizations, only utilizing its intermediate representation construction and analysis features.

Finally, the optimized program was simulated using SimpleScalar. Our baseline simulation configuration models a modern out-of-order processor microarchitecture. The processor has a large window of execution; it can fetch and issue up to 4 instructions per cycle. It has a 32 entry re-order buffer with a 16 entry load/store buffer. Loads can only execute when all prior store addresses are known. In addition, all stores are issued in program order with respect to prior stores. A 4k entry gshare branch predictor was used and there is an six cycle minimum branch misprediction penalty. The processor has 4 integer ALU units, 2-load/store units, 2-FP adders, 1-integer MULT/DIV, and 1-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 fully pipeline allowing a new instruction to initiate execution each cycle.

The processor we simulated has 32k 2-way set-associative instruction and data caches. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with 2 ports. The data cache access latency is one cycle (for a total load latency of two cycles). There is a unified second-level 512k 4-way set-associative cache with 32 byte blocks, with a six cycle cache hit latency. If there is a second-level cache miss it takes a total of 60 cycles to make the round trip access to main memory. We model the bus latency to main memory with a six cycle bus occupancy per request. There is a 16 entry 4-way associative instruction TLB and a 32 entry 4-way associative data TLB, each with a 30 cycle miss penalty.

The value prediction table used by the PREDICT and UPDATE instructions contains enough entries so that each optimization site gets a unique entry. The number of entries in the table ranges from 22 (*m88ksim*) to 136 (*mcf*) when using the best set of candidates. The best set of value prediction candidates is determined in Section 3.5. This set is used for all other experiments (except for the software-only section - see Section 3.4).

3.2 Coverage and accuracy

Value predictors can be measured by their coverage and accuracy, giving a clear indication how well the predictor will perform. In profile based candidate selection, a threshold is used to filter out bad candidates. An instruction is only a candidate if its accuracy exceeds this threshold. Using this approach, there is a trade-off between coverage and accuracy. When the threshold is increased, the accuracy increases by only allowing the most accurate candidates; and coverage decreases since the higher threshold will eliminate many candidates. Conversely, decreasing the threshold increases the coverages but lowers the accuracy.

Figure 3 looks at the confidence and accuracy for all value prediction candidates within each of the benchmarks. The baseline statistics are updated each time the instruction is executed while statistics in the branch based confidence scheme are only updated when the branch predictor indicates a high confidence prediction. A static stride predictor is used in both instances. The accuracy for the static predictor is equivalent to the value prediction accuracy, and the coverage is the percentage of instructions (using a dynamic instruction count) that meet or exceed the threshold. For the predictor with confidence, the accuracy is equal to the number of times a prediction is correct when the branch predictor indicates a high confidence prediction and the coverage is the percentage of instructions that were predicted to have high confidence.

The results in Figure 3 show that the value predictor with confidence has better coverage - accuracy pairs than with no confidence. Even with a threshold of zero, accuracy is high for almost all of the benchmarks. Six of the eight benchmarks have accuracies greater than 94% with *crafty* (87.6%) and *go* (79.8%) being the two exceptions. As threshold increases the predictor with no confidence obtain high accuracies, but coverage decreases. This result is expected since using confidence will add candidates that have a pattern of predictability but do not necessarily have overall high predictability.

3.3 Value prediction with confidence

Now, we measure the actual performance benefit of the value prediction technique using confidence. We compare each program to a baseline with no value prediction and to a model of the value prediction scheme

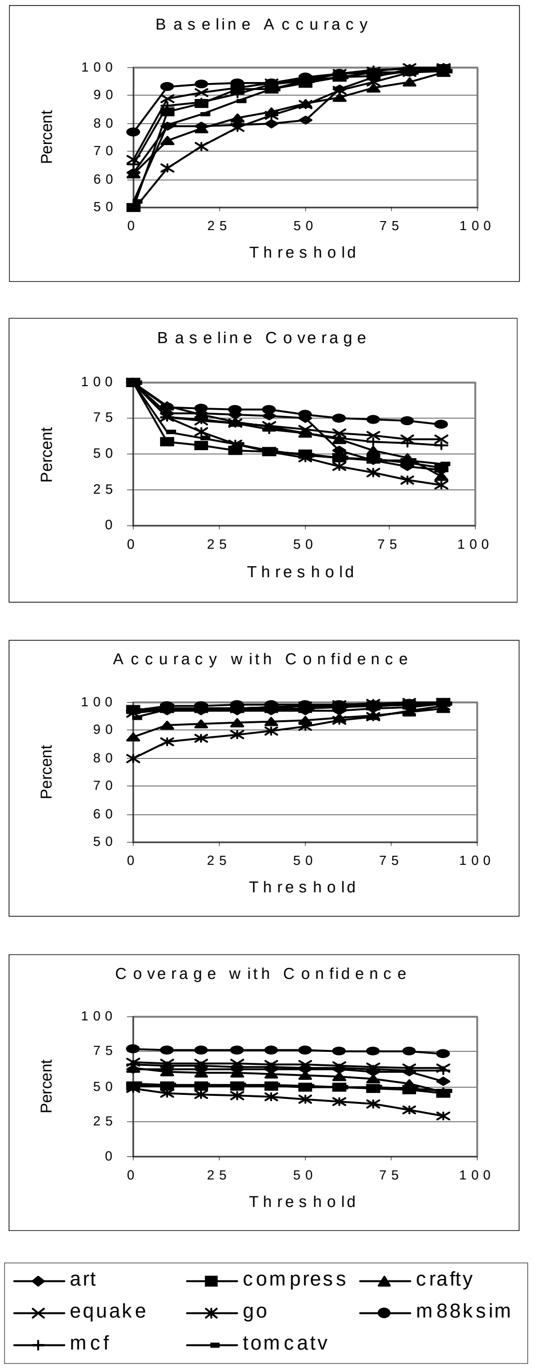

Figure 3: Coverage and accuracy results. The top plots show baseline value prediction coverage and accuracy with no confidence and the bottom plots show coverage and accuracy when the branch predictor is used to estimate confidence. Given a fixed accuracy, the coverage is higher when confidence is used resulting in more candidates for value prediction.

implemented in [6]. This scheme is also compiler-controlled and uses the PREDICT and UPDATE instructions. It is applied using a statically scheduled compiler where speculative code is executed before a branch that will verify if the prediction was correct. The fixup code re-executes all of the instructions using the correct value. This is modeled in our implementation by forcing execution down the “not taken” or “use value prediction” path. If the value shouldn’t be used, control will be transferred to the fixup code. However, a misprediction penalty is only assessed if the branch predictor predicted not taken.

The results of this experiment are shown in Figure 4. An average speedup of 15% is obtained over the baseline with no value prediction and up to 38.1% for *m88ksim* with reference input. The value predictor with confidence was, on average, 3.3% faster than the model, but only three benchmarks (*compress*, *equake*, and *mcf*) exhibited any significant speedup. The static predictor even did slightly better for the *crafty* benchmark.

We also ran experiments using an identically-sized bimodal and a hybrid branch predictor (profiling was still done using a gshare predictor). Not surprisingly, the gshare predictor outperformed the bimodal predictor (average gain of 5.4%) but was not as good as the hybrid predictor (average gain of 2.9%).

3.4 Software value prediction

The results of software-only value prediction are shown in Figure 5. Speedups were a mixed bag, ranging from 1% for *art* and *mcf* to 13% for *m88ksim*. The average speedup for the five benchmarks is 4.6%. Compiler-controlled value prediction with ISA support did much better. The slower performance can be attributed to the fact that the predictor state is stored in memory, resulting in larger overheads compared to hardware. Another problem can be attributed to the fact we do not allow register spills in our implementation. Candidates were rejected if there wasn’t a free register to hold the prediction. This means far fewer candidates were optimized (a subset of the best set as determined in section 3.5) compared to the experiments with ISA support. In fact, no instructions were optimized for this reason in three of the benchmarks (*compress*, *crafty*, and *tomcatv*). A possible improvement is to include more sophisticated register allocation techniques but this require balancing register files spills in to memory and the benefit of applying the value prediction optimization.

3.5 Selecting value prediction candidates

Selecting proper candidates is an important aspect to obtaining the maximum speed up for value prediction. In this experiment, we look at three different selection criteria. In the first case, we assume that each time the optimization is executed, it has a constant or *fixed* benefit. As a result, this selection criteria is tied to the number of times the instruction will successfully execute the optimization. In the second case, the benefit is tied to its estimated position and frequency in the *critical path*. This middle metric is computed for each instruction by determining the distance from the closest end-

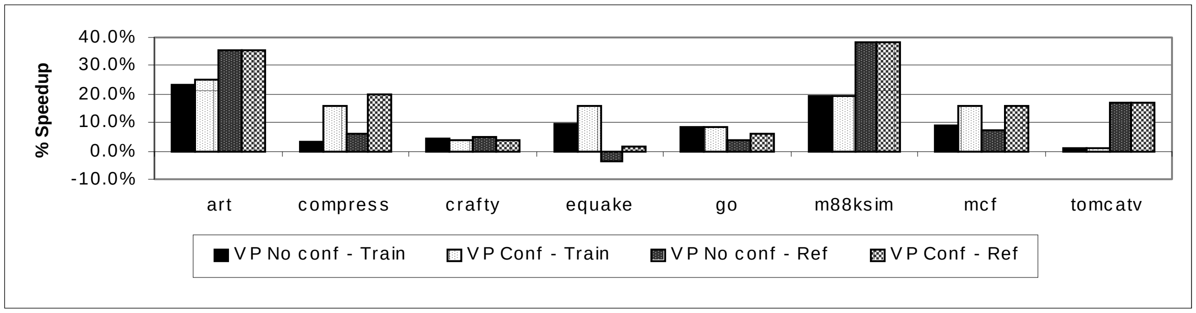

Figure 4: Measuring value prediction with confidence. Using the branch predictor with confidence obtains an average speedup of 15.2% compared to a program with no value prediction and an average speedup of 3.3% speedup over programs with value prediction but no confidence.

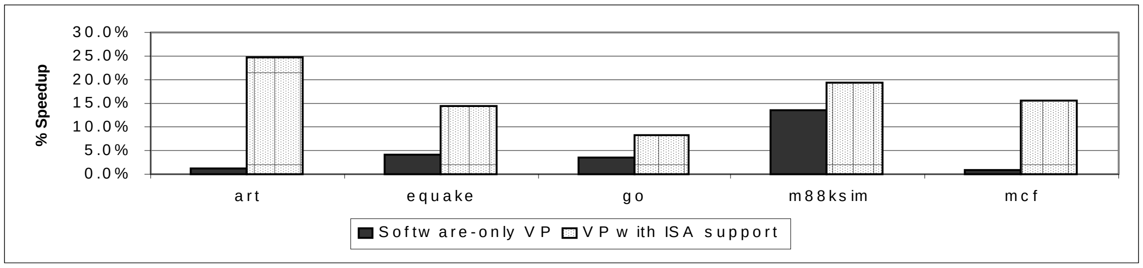

Figure 5: Comparing software-only value prediction to value prediction with ISA support. Software-only value prediction leads to a average speedup of 4.6% Speedups of at least 1% for each of the five benchmarks including a 13% speedup for *m88ksim* but value prediction with ISA support does significantly better. Insufficient free registers were available to apply the optimization to the three other benchmarks (*compress*, *crafty*, and *tomcatv*).

point in the dynamic instruction window dependence chain. This result is averaged over the duration of the simulation and then multiplied by the number of times the optimization is executed. The third selection criterion is a *union* of the two previous techniques in an attempt to gather the best candidates from both selection criteria. The final selection is based on multiplying the benefit by the optimization accuracy.

When selecting candidates, only the best instruction was selected for each basic block since adjacent instructions tended to have similar benefit scores and applying optimizations too close together results in overheads that reduce the effectiveness of value prediction. This heuristic is simple but may be wrong in situations where the basic blocks are small. We plan to address this problem in future work.

For each benchmark, four to five sets of candidates were analyzed for each selection method. The sets were obtained by taking the best *n* candidates where *n* was a different number for each set. The numbers used to determine the sets were the same for each criterion within a benchmark except for the union which contained the union of the two other selection methods. The numbers used varied from benchmark to benchmark due to different static instruction counts.

The results of this experiment are shown in Figure 6. The graph compares the best performing set of each selection technique. For most benchmarks, there isn't significant differentiation between the three selection methods. There are three benchmarks (*crafty*, *go*, and *tomcatv*) where the critical path based techniques did poorly. This can be explained using the data in Table 2. This table shows the average number of unique dependence chains present in the instruction window during execution, the average size of each chain, and the average length of the longest size chain present in the instruction window each cycle. Benchmarks *crafty*, *go*, and *tomcatv* all had an average long chain of 4.2 or less while the other five benchmarks had an average longest chain greater than 5.0. Since the longest chain is short, the benefit of splitting up the chains is reduced since less instructions are executed in parallel. For benchmarks that did well, such as *m88ksim* and *equake*, there are a large number of long chains. The other benchmarks saw little difference between the two techniques despite long chain sizes. This is due to the selection methods picking similar lists of candidates.

Table 2: Chain statistics for benchmarks

Benchmark	Avg. Num. of Chains	Avg. Size of Chain	Avg. Longest Sized Chain
art	1.7572	5.2454	5.5716
compress	6.6651	3.3440	5.3126
crafty	4.7575	2.6762	3.0922
equake	6.4199	2.9883	5.2122
go	3.8074	3.1829	3.7823
m88ksim	7.4921	4.3198	5.0041
mcf	7.1499	3.1283	5.7780
tomcatv	4.8718	2.8814	4.1790

Table 3: Number of optimized instructions. This table indicates the number of instructions that were optimized by the compiler.

Benchmark	Optimized Instructions	Benchmark	Optimized Instructions
art	26	go	78
compress	59	m88ksim	22
crafty	87	mcf	136
equake	134	tomcatv	61

Table 3 shows the number of instructions that are optimized in the best case and gives a rough estimate of the size of the value predictor table needed. In the worst case, *mcf* optimized 136 instructions, assuming 8 bytes (standard Alpha data size) for each entry, the predictor will have an overall size of just over 1 KB. This is significantly smaller than hardware-only solutions that require tables in the 16 - 32 KB range [19]. In addition, hardware cannot do selection so it must indiscriminately apply value prediction to highly predictable instructions while the compiler can selectively apply the optimization to only the most gainful sites requiring significantly fewer resources.

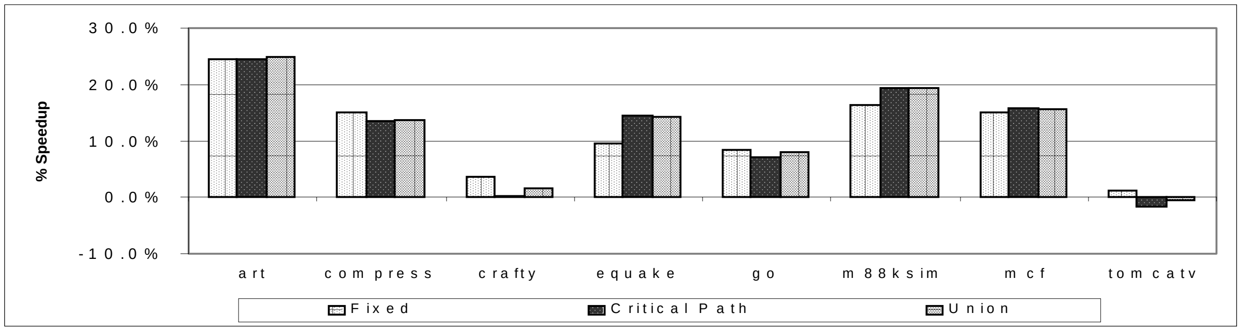

Figure 6: Analyzing different techniques for selecting value prediction candidates. The critical path based technique works best for benchmarks that have long chains such as *equake* and *m88ksim*. Using a fixed benefit model worked best for benchmarks with short chains such as *crafty* and *go*. In other cases, the two techniques were about equal.

3.6 Misprediction tolerant branches

The special branch BEQIT is used to ignore branch mispredictions when the branch predictor mistakenly uses the actual value when the predicted value is indeed correct. Using the actual value does not cause incorrect program behavior so a misprediction is unnecessary in this case. The results of applying the BEQIT branch are shown in Figure 7. The misprediction tolerant branch improved the performance of many of the benchmarks, especially for *compress* where a speedup of 13% was realized for the reference input set. A few programs, such as *m88ksim* and *tomcatv*, saw little benefit as these programs have few mispredicted value speculation branches.

4. Related work

Several studies have looked at the predictability of the instructions and the overall performance potential of value prediction [2, 9, 10, 11, 13, 19]. Lepak and Lipasti [9] show that the data used in store instructions is predictable. Marcuello and Gonzalez [13] look at how various microarchitectural parameters affect value prediction performance. They show that small instruction windows only render moderate speedups, making the criticality of instructions important. Calder *et. al.* [2] found that loads in general are predictable and have invariant values. They also use this knowledge to specialize code sequences.

Fu *et. al.* [4, 5, 6] first examined compiler controlled value prediction. They applied value prediction optimizations to selected candidates based on profiling data. In [5], the value prediction is completely software based. Speedups were obtained for a few benchmarks. In [6], PREDICT and UPDATE instructions are added to the instruction set, resulting in better speedups since predictions could be stored more efficiently in hardware. In [4], the design is modified to incorporate a CHECKPRED instruction which will rewind the architectural state in the event of a value misprediction. The main advantage of this approach is that it eliminates the need of explicit fixup code since the original code is re-executed with the proper value. They also look at the predictability of instructions based on their location in the static dependence graph and found that instructions in the middle of dependence chains do not necessarily lead to the best speedup.

Many researchers have developed techniques for more accurate value prediction [3, 8, 17, 19, 23]. Calder *et. al.* [3] adds the notion of confidence to value predictors so the hardware can identify when it should predict values. Unlike our scheme, the confidence is completely implemented in hardware.

Recent research has provided a better understanding of the behavior of long latency instructions. Zilles and Sohi [25] looked at the backward slices of performance degrading instructions. The slice is formed by looking at the instructions that lead up to the performance degrading event. The slices can be pre-executed to hide the latency of the event improving performance. Srinivasan and Lebeck [22] found that many load instructions can tolerate long latencies without hindering performance. Value prediction tries to remove these long latencies and this research implies that effort should be concentrated on predicting load instructions where the long latency cannot be tolerated without hindering performance.

Other value prediction research has focused on separating the value prediction hardware from the rest of the machine or applying it to different microarchitectures [12, 14, 15, 18]. Lee *et. al.* [12] describes an implementation where the value prediction hardware is decoupled from the instruction fetch stage. Nakra *et. al.* [18] applies value prediction to VLIW machine that uses two execution engines. One engine is for VLIW code that uses value prediction to remove data dependencies and the other executes compensation code when there is a misprediction. Marcuello *et. al.* [14, 15] added value prediction to multithreaded architectures and found it is important to eliminate serialization caused by inter-thread dependencies.

5. Conclusions

In this paper, we have shown that using a branch predictor as a confidence mechanism for value prediction was effective in improving performance while minimizing hardware costs. The speedup on average is 15.2% over a program with no value prediction and 3.3% over a program with a static value predictor without confidence. Hardware can be further reduced by implementing value prediction completely in software. Moderate speedups were obtained for some benchmarks (average of 4.6%); others were excluded due to a lack of free registers to implement the optimization.

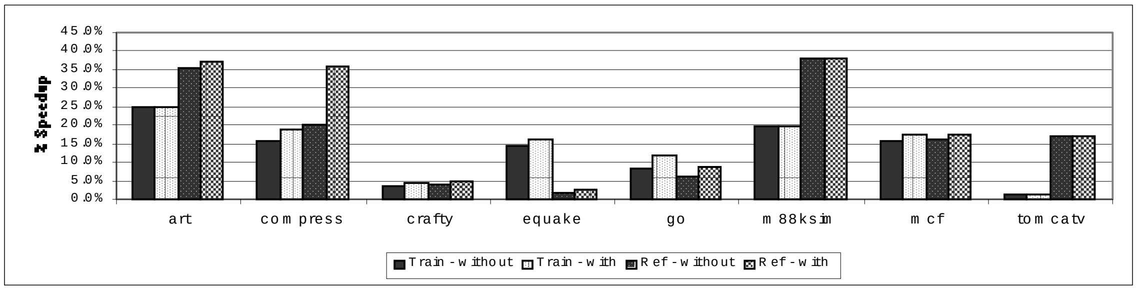

Figure 7: Speedup with and without a misprediction tolerant branch. There were speedups of greater than 1% in four of the eight benchmarks. The branch performed well in *compress* with the reference data set with a speedup of 13%.

The best method for selecting candidates differs upon the size and number of dependence chains in a program. Using a constant fixed benefit was better for programs with fewer and shorter chains while the critical path technique was better for programs with more and longer chains. The largest number of optimized instructions in any one benchmark was 136 for *mcf*, requiring a value prediction table of only slightly larger than 1 KB, much smaller than required by similar performing hardware techniques. A low-cost misprediction tolerant branch was used to avoid misprediction, resulting in speedups for most benchmarks, including a 13% speedup in *compress*.

There are several opportunities for future work. One possibility is to further understand the best set of value prediction candidates. A factor not looked at in this paper is the spacing of the optimized instructions. If optimized instructions are too close together, there will be too much splitting of dependence chains and the overhead of the optimization will dominate. On the other hand, if the optimized instructions are too far apart, the maximum benefit of value prediction is not obtained. There may be other uses for the optimizations outlined in this paper. The branch predictor could be used to estimate confidence for other forms of speculation, or the misprediction tolerant branch can be used in optimizations that contain a programmatically correct path regardless of the actual outcome of the branch.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This material is based upon work supported under a National Science Foundation Graduate Fellowship. Equipment support was provided by Intel Corporation.

References

[1] D.C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", *University of Wisconsin Computer Sciences Technical Report #1342*, June 1997.

[2] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", in J*ournal of Instruction-Level Parallelism*, March 1999.

[3] B. Calder, G. Reinman, and D. M. Tullsen, "Selective Value Prediction", in *26th International Symposium of Computer Architecture*, May 1999.

[4] C. Fu. and T. M. Conte, "Value Speculation Mechanisms for EPIC Architectures", *Technical Report. Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911*, October 1998.

[5] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Software-Only Value Speculation Scheduling", *Technical Report. Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911*, June 1998

[6] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value Speculation Scheduling for High Performance Processors", in *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[7] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun, "Confidence Estimation for Speculation Control", in *25th International Symposium of Computer Architecture*, June 1998.

[8] J. Huang, Y. Choi, D. J. Lilja, "Improving Value Prediction by Exploiting Both Operand and Output Value Locality", *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 99-06*, July 1999.

[9] K. M. Lepak and M. H. Lipasti, "On the Value Locality of Store Instructions", in *27th International Symposium of Computer Architecture*, June 2000.

[10] M. H. Lipasti and J. P. Shen, "Exploiting Value Locality to Exceed the Dataflow Limit", in *29th International Symposium on Microarchitecture*, December 1996.

[11] M. H. Lipasti and J. P. Shen, "The Performance Potential of Value and Dependence Prediction", in *EUROPAR-97*, August 1997.

[12] S. Lee, Y. Wang, P.Yew, "Decoupled Value Prediction on Trace Processors", in *6th International Symposium on High Performance Computer Architecture*, January 2000.

[13] P. Marcuello and A. Gonzalez, "The Potential of Data Value Speculation to Boost ILP", in *12th International Conference on Supercomputing*, July 1998.

[14] P. Marcuello and A. Gonzalez, "A Quantitative Assessment of Thread-Level Speculation Techniques", in *1st International Parallel and Distributed Processing Symposium*, May 2000.

[15] P. Marcuello, J. Tubella, and A. Gonzalez, "Value Prediction for Speculative Multithreaded Architectures" in *32th International Symposium on Microarchitecture*, November 1999.

[16] R. Muth, S. Debray, S. Watterson, and K. De Bosschere, "alto: A Link-Time Optimizer for the Compaq Alpha", *University of Arizona Computer Sciences Technical Report 98-14*, December 1998.

[17] T. Nakra, R. Gupta, and M. L. Soffa, "Global Context-Based Value Prediction", in *5th International Symposium on High Performance Computer Architecture*, January 1999.

[18] T. Nakra, R. Gupta, and M. L. Soffa, "Value Prediction in VLIW Machines", in *26th International Symposium on Computer Architecture*, May 1999.

[19] Y. Sazeides and J. E. Smith, "The Predictability of Data Values", in *30th International Symposium on Microarchitecture*, December 1997.

[20] A. Sodani and G. Sohi, "Dynamic Instruction Reuse", in *24th International Symposium on Computer Architecture*, June 1997.

[21] E. Schnarr and J. Larus, "Fast Out-Of-Order Processor Simulation Using Memoization", in *8th International Conference on Architectural Support for Programming Languages and Operating System*s, October 1998.

[22] S. T. Srinivasan and A. R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors", in *31st International Symposium on Microarchitecture*, December 1998.

[23] D. M. Tullsen, J. S. Seng, "Storageless Value Prediction Using Prior Register Values", in *26th International Symposium on Computer Architecture*, May 1999.

[24] VTune^{TM} Performance Analyzer Home Page, http://developer.intel.com/vtune/analyzer/index.htm

[25] C. B. Zilles and G. Sohi, "Understanding the Backwards Slices of Performance Degrading Instructions", in *27th International Symposium of Computer Architecture*, June 2000.