Value Speculation Scheduling for High Performance Processors

Chao-ying Fu Matthew D. Jennings Sergei Y. Larin Thomas M. Conte

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7911
{cfu, mdjennin, sylarin, conte}@eos.ncsu.edu

ABSTRACT

Recent research in value prediction shows a surprising amount of predictability for the values produced by register-writing instructions. Several hardware based value predictor designs have been proposed to exploit this predictability by eliminating flow dependencies for highly predictable values. This paper proposed a hardware and software based scheme for value speculation scheduling (VSS). Static VLIW scheduling techniques are used to speculate value dependent instructions by scheduling them above the instructions whose results they are dependent on. Prediction hardware is used to provide value predictions for allowing the execution of speculated instructions to continue. In the case of miss-predicted values, control flow is redirected to patch-up code so that execution can proceed with the correct results. In this paper, experiments in VSS for load operations in the SPECint95 benchmarks are performed. Speedup of up to 17% has been shown for using VSS. Empirical results on the value predictability of loads, based on value profiling data, are also provided.

Keywords

Value speculation, value prediction, VLIW instruction scheduling, instruction level parallelism

1. INTRODUCTION

Modern microprocessors extract instruction level parallelism (ILP) by using branch prediction to break control dependencies and by using dynamic memory disambiguation to resolve memory dependencies [1]. However, current techniques for extracting ILP are still insufficient. Recent research has focused on value prediction hardware for dynamically eliminating flow dependencies (also called true dependencies) [2], [3], [4], [6], [7], [8], [9]. Results have shown that values produced by register-writing instructions are potentially highly predictable using various value predictors: last-value, stride, context-based, two-level, or hybrid predictors. This work illustrates that value speculation in future high performance processors will be useful for breaking flow dependencies, thereby exposing more ILP. This paper examines ISA, hardware and compiler synergies for exploiting value speculation. Results indicate that this synergy enhances performance on difficult, integer benchmarks.

Prior work in value speculation utilizes hardware-only schemes (e.g. [2], [3]). In these schemes, the instruction address (PC) of a register-writing instruction is sent to a value predictor to index a prediction table at the beginning of the fetch stage. The prediction is generated during the fetch and dispatch stages, then forwarded to dependent instructions prior to their execution stages. A value-speculative dependent instruction must remain in a reservation station (even while its own execution continues), and be prevented from retiring, until verification of its predicted value. The predicted value is compared with the actual result at the state-update stage. If the prediction is correct, dependent instructions can then release reservation stations, update system states, and retire. If the predicted value is incorrect, dependent instructions need to re-execute with the correct value. Figure 1

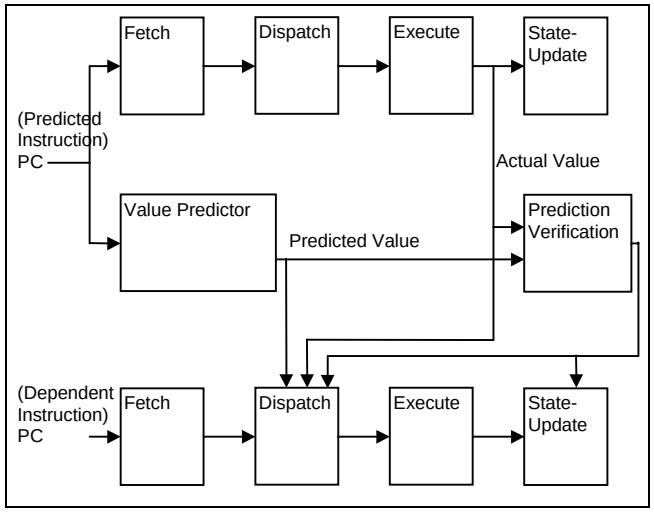

Figure 1. Pipeline Stages of Hardware Value Speculation Mechanism for Flow Dependent Instructions. The dependent instruction executes with the predicted value in the same cycle as the predicted instruction.

illustrates the pipeline stages for value speculation utilizing a hardware scheme.

Little work has been done on software-based schemes to perform value prediction and value speculation of dependent instructions. In a related approach to a different problem, the memory conflict buffer [1] was presented to dynamically disambiguate memory dependencies. This allows the compiler to speculatively schedule memory references above other, possibly dependent, memory instructions. Patch-up code, generated by the compiler, ensures correct program execution even when the memory dependencies actually occur. Speculatively scheduled memory references improves performance by aggressively scheduling references that are highly likely to be independent of each other. Likewise, value-speculative scheduling attempts to improve performance by aggressively scheduling flow dependencies that are highly likely to be eliminated through value prediction. Patch-up code is used when values are miss-predicted. We apply this scheme to value speculation and propose a combined hardware and software solution, which we call *value speculation scheduling (VSS)*.

Hardware pipeline stages for the VSS scheme are shown in Figure 2. Two new instructions, *LDPRED* and *UDPRED*, are introduced to interface with the value predictor during the execution stage. LDPRED loads the predicted value generated by the predictor into a specified general-purpose register. UDPRED updates the value predictor with the actual result, resetting the device for future predictions after a miss-prediction. Figure 3 shows an example of using LDPRED and UDPRED to perform VSS.

In the original code sequence of Figure 3(a), instructions I1 to I6 form a long flow dependence chain, which must execute sequentially. If the flow dependence from

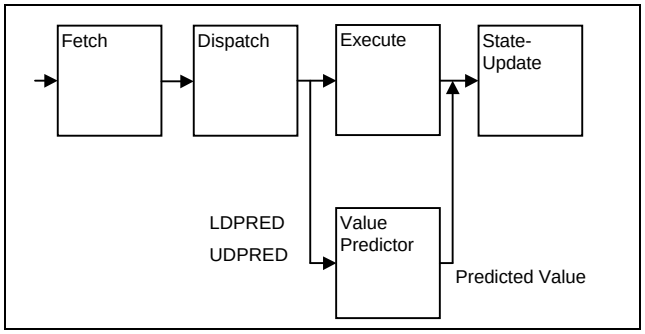

Figure 2. Pipeline Stages of Value Speculation Scheduling Scheme. Two new instructions, LDPRED and UDPRED, interface with the value predictor during the execution stage.

(a) Original code

```
I1:     ADD     R1 ← R2, 5
I2:     SHL     R3 ← R1, 2
I3:     LW      R4 ← 0(R3)
I4:     ADD     R5 ← R4, 1
I5:     OR      R6 ← R5, R7
I6:     SW      0(R3) ← R6
Next:   .....
```

(b) New code after value speculation of R4 (predicted instruction I3)

```
I1:     ADD                 R1 ← R2, 5
I2:     SHL                 R3 ← R1, 2
I3:     LW                  R4 ← 0(R3)
I7:     LDPRED              R8 ← index  // load prediction into R8
I4':    ADD                 R5 ← R8, 1
I5':    OR                  R6 ← R5, R7
I6':    SW                  0(R3) ← R6
I8:     BNE Patchup         R8, R4      // verify prediction
Next:   .....

Patchup:
I9:     UDPRED              R4, index   // update predictor with R4
I4:     ADD                 R5 ← R4, 1
I5:     OR                  R6 ← R5, R7
I6:     SW                  0(R3) ← R6
I10:    JMP Next
```

Figure 3: Example of Value Speculation Scheduling.

instruction I3 to I4 is broken, via VSS, the dependence height of the resulting dependence chain is shortened. Furthermore, ILP is exposed by the resulting data dependence graph. Figure 4 shows the data dependence graphs for the code sequence of Figure 3 before and after breaking the flow dependence from instruction I3 to I4. Assume that the latencies of arithmetic, logical, branch, store, LDPRED and UDPRED instructions are 1 cycle, and that the latency of load instructions is 2 cycles. Then, the schedule length of the original code sequence of Figure 4(a), instructions I1 to I6, is seven cycles. By breaking the flow dependence from instruction I3 to I4, VSS results in a schedule length of five cycles. Figure 4(b) illustrates the schedule now possible due to reduced overall dependence height and ILP exposed in the new data dependence graph.

This improved schedule length, from seven cycles to five cycles, does not consider the penalty associated with miss-prediction due to the required execution of patch-up code. The impact of patch-up code on performance will be discussed in section 3.

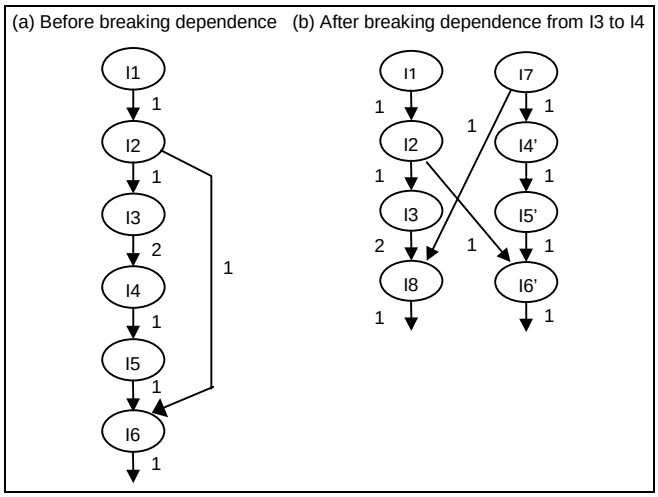

Figure 4. Data Dependence Graphs for Codes of Figure 3. The numbers along each edge represent the latency of each instruction. In 4(a), the schedule length is seven cycles. In 4(b), because of exposed ILP and dependence height reduction, the schedule length is reduced to five cycles.

In Figure 3(b), the value speculation scheduler breaks the flow dependence from instruction I3 to I4. Instructions I4, I5 and I6 now form a separate dependence chain, allowing their execution to be speculated during scheduling. They become instructions I4' I5' and I6', respectively. An operand of instruction I4' is modified from R4 to R8. Register R8 contains the value prediction for destination register R4 of the predicted instruction I3.

Instruction I7, LDPRED, loads the value prediction for instruction I3 into register R8. When the prediction is incorrect (R8≠R4), instruction I9, UDPRED, updates the value predictor with the actual result of the predicted instruction, from register R4. Note that the resulting UDPRED instruction is part of patch-up code and its execution is only required when a value is miss-predicted. To ensure correct program execution, the compiler inserts the branch instruction, I8, after the store instruction, I6', to branch to the patch-up code when the predicted value does not equal the actual value. The patch-up code contains UDPRED and the original dependent instructions, I4, I5 and I6. After executing patch-up code, the program jumps to the next instruction after I8 and execution proceeds as normal.

Each LDPRED and UDPRED instruction pair that corresponds to the same value prediction uses the same table entry index into the value predictor. Each index is assigned by the compiler to avoid unnecessary conflicts inside the value predictor. While the number of table entries is limited, possible conflicts are deterministic and can be factored into choosing which values to predict in a compiler approach. A value predictor design, featuring the new LDPRED and UDPRED instructions, will be described in section 2.

By combining hardware and compiler techniques, the strengths of both dynamic and static techniques for exploiting ILP can be leveraged. We see several possible advantages to VSS:

- Static scheduling provides a larger scheduling scope for exploiting ILP transformations, identifying long dependence chains suitable for value prediction and then re-ordering code aggressively.
- Value-speculative dependent instructions can execute as early as possible before the predicted instruction that they depend.
- The compiler controls the number of predicted values and assigns different indices to them for accessing the prediction table. Only instructions that the compiler deems are good candidates for predictions are then predicted, reducing conflicts for the hardware.
- Patch-up code is automatically generated, reducing the need for elaborate hardware recovery techniques.
- Instead of relying on statically predicted values (e.g., from profile data), LDPRED and UDPRED access dynamic prediction hardware for enhanced prediction accuracy.
- VSS can be applied to dynamically-scheduled (superscalar) processors, statically-scheduled (VLIW) processors, or EPIC (explicitly parallel instruction computing) processors [14].

There is a drawback to VSS. Because static scheduling techniques are employed, value-speculative instructions are committed to be speculative and therefore always require predicted values. Hardware only schemes can dynamically decide when it is appropriate to speculatively execute instructions. The dynamic decision is based on the value predictor's confidence in the predicted value, avoiding miss-prediction penalty for low confidence predictions.

The remainder of this paper is organized as follows: Section 2 examines the value predictor design for value speculation scheduling. Section 3 introduces the VSS algorithm. Section 4 presents experimental results of VSS. Section 5 concludes the paper and mentions future work.

2. VALUE PREDICTOR DESIGN

Microarchitectural support for value speculation scheduling (VSS) is in the form of special-purpose value predictor hardware. Value prediction accuracy directly relates to performance improvements for VSS. Various value predictors, such as last-value, stride, context-based, two-level, and hybrid predictors [2], [3], [4], [6], [7], [9],

provide different prediction accuracy. Value predictors with the most design complexity, in general, provide for the highest prediction accuracy. In order to feature LDPRED and UDPRED instructions for VSS, previously proposed value predictors must be re-designed slightly.

Figure 5 shows the block diagram of a value predictor that includes LDPRED and UDPRED instructions. In this value predictor, there are three fundamental units, the *current state* block, the *old state* block and the *prediction hardware* block. The current state block may contain register values, finite state machines, history information, or machine flags, depending on the prediction method employed. The old state block hardware is a duplicate of the current state block hardware. Predictions are generated by the prediction hardware with input from the current state block. Various prediction mechanisms can be used. For example, generating the prediction as the last value (last value predictors [2], [3]). Or, generating the prediction as the sum of the last value and the stride, which is the difference between the most recent last values (stride predictors [4], [6], [7], [9]). Also, two-level predictors [7] allow for the prediction of recently computed values. For two-level predictors, a value history pattern indexes a pattern history table, which in turn is used to index a value prediction from recently computed values. Two-level value prediction hardware is based on two-level branch prediction hardware.

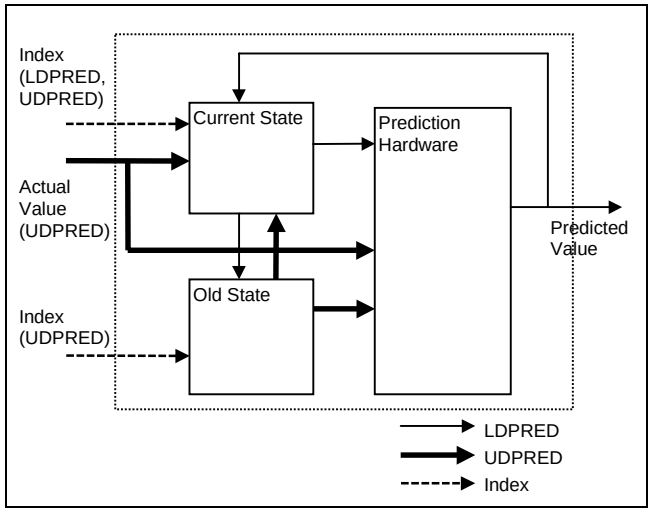

Figure 5. Block Diagram of Value Predictor featuring LDPRED and UDPRED.

Both the LDPRED and UDPRED instructions contain an immediate operand that specifies the value predictor table index. In general (independent of the prediction hardware chosen) the LDPRED instruction performs three actions. The compiler assigned number indexes each action. First, the prediction hardware generates the predicted value by using input from the current state block. Second, current state information is shifted to the old state block. Last, the current state block is updated based on the predicted value from the prediction hardware. Information used by the prediction hardware is updated simultaneously with the current state block update. Note that for the LDPRED instruction, the predicted value is used to update the current state block speculatively.

The compiler assigned number also indexes the operation of the UDPRED instruction. When the value prediction is incorrect, the patch-up basic block of Figure 3(b) must be executed. The execution of UDPRED instructions only occurs in patch-up code, or only when values are miss-predicted. The UDPRED instruction causes the update of both the current state block and the prediction hardware with the actual computed value and the old state block.

If the compiler can ensure that each LDPRED/UDPRED instruction pair is executed in turn (each prediction is verified and value predictions are not nested) the old state block requires only one table entry. The same table entry in the old state block is updated by every LDPRED instruction, and used by every UDPRED instruction, in the case of miss-prediction.

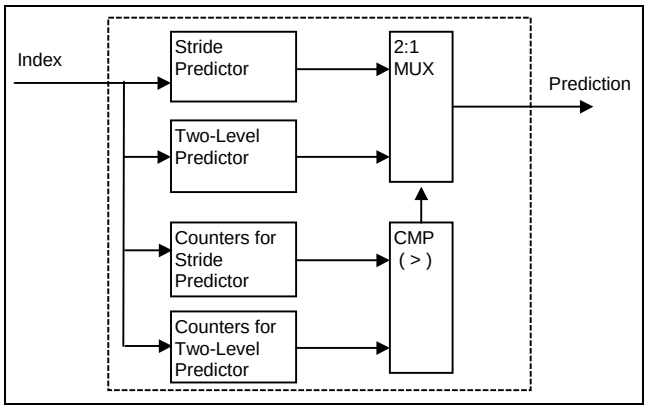

Figure 6. Hybrid Predictor (Stride and Two-Level). Saturating counters are compared to select between the prediction techniques.

In the VSS scheme, a prediction needs to be generated for each LDPRED instruction. There is no flag in the value predictor to indicate if a value prediction is valid or not. The goal of the value predictor is to generate as many correct predictions as possible. In this paper, stride, two-level and hybrid value predictors [7] are implemented to find the design which provides the highest prediction accuracy for use in the VSS scheme. Stride predictors predict arrays and loop induction variables well. Two-level predictors capture the recurrence of recently used values and generate predictions based on previous patterns of values. However, neither of them alone can obtain high prediction accuracy for all programs, which exhibit different characteristics. Therefore, hybrid value predictors, consisting of both stride and two-level prediction are designed to cover both of these situations.

1. **Perform Value Profiling**
2. **Perform Region Formation**
3. **Build Data Dependence Graph for Region**
4. **Select Instruction with Prediction Accuracy (based on Value Profiling) greater than a Threshold**
5. **Insert LDPRED after Predicted Instruction (selected instruction of step 4)**
6. **Change Source Operand of Dependent Instruction(s) to Destination Register of LDPRED**
7. **Insert Branch to Patch-up Code**
8. **Generate Patch-up Code (which contains UDPRED)**
9. **Repeat Steps 4 – 8 until no more Candidates Found**
10. **Update Data Dependence Graph for Region**
11. **Perform Region Scheduling**
12. **Repeat Steps 2 – 11 for each Region**

Figure 7. Algorithm of Value Speculation Scheduling.

Figure 6 shows such a hybrid predictor that obtains high prediction accuracy. The selection between the stride predictor and the two-level predictor is different from that in [7]. Every table entry has a saturating counter in the stride predictor and in the two-level predictor. The saturating counter increments when its corresponding prediction is correct, and decrements when its prediction is incorrect. Both saturating counters and predictors are updated for each prediction, regardless of which prediction is actually selected. The hybrid predictor selects the predictor with the maximum saturating counter value. In the event of a tie, the hybrid predictor favors the prediction from the two-level predictor. Prediction accuracy results for the three value predictors will be presented in section 4.

3. VALUE SPECULATION SCHEDULING

Performance improvement for value speculation scheduling (VSS) is affected by prediction accuracy, the number of saved cycles (from schedule length reduction) and the number of penalty cycles (from execution of patch-up code). Suppose that after breaking a flow dependence, value-speculative dependent instructions are speculated, saving S cycles in overall schedule length when the prediction is correct. Patch-up code is also generated and requires P cycles. Prediction accuracy for the speculated value is X. In this case, speedup will be positive if $S > (1-X) * P$ holds. For the example of Figure 3(b) VSS saves 2 cycles (from 7 cycles to 5 cycles) and the resulting patch-up code contains 5 instructions, requiring 3 cycles in an ILP processor. Therefore, for positive speedup, the prediction accuracy must be greater than 33%. If the actual prediction accuracy is less, performance will be degraded by VSS.

With these performance considerations in mind, an algorithm for VSS is proposed in Figure 7.

The first step is to perform value profiling. The scheduler must select highly predictable instructions to improve performance through VSS. Results from value profiling under different inputs and parameters have been shown to be strongly correlated [5], [6]. Therefore, value profiling can be used to select highly predictable instructions on which to perform value speculation.

Value profiling can be performed for all register-writing instructions. If profiling overhead is a concern, a filter may be used to perform value profiling only on select instructions. Select instructions may be those that reside in critical paths (long dependence height) or those that have long latency (e.g., load instructions). In [5], estimating and convergent profiling are proposed to reduce profiling overhead for determining the invariance of instructions. Similar techniques could be applied for determining the value predictability of instructions.

Next, the value speculation scheduler performs region formation. Treegion formation [10] is the region type chosen for our experiments. A treegion is a non-linear region that includes multiple execution paths in the form of a tree of basic blocks. The larger scheduling scope of treegions allows the scheduler to perform aggressive control and value speculation. A data dependence graph is then constructed for each treegion. In step four, a threshold of prediction accuracy is used to determine whether or not to perform value speculation on each instruction. For each instruction, the scheduler queries the value profiling information to get the estimate of its predictability. If the predictability estimate is greater than the threshold, value prediction is performed. For aggressive scheduling, more instructions can be speculated by choosing a low threshold. Suggested values for the threshold are derived from experimental results in section 4.

When an instruction is selected for value prediction, a LDPRED instruction is inserted directly after it. The LDPRED instruction has an immediate value that is assigned by the scheduler to be its chosen index into the value predictor. A new register is also assigned as the destination of the LDPRED instruction. Once the new destination register has been chosen for the LDPRED instruction, any dependent instruction(s) need to update their source register(s) to reflect the new dependence on the LDPRED instruction. Only the first dependent instruction in a chain of dependent instructions needs to update its register source, the remaining dependencies in the chain are

unaffected. Even though more than one chain of dependent instructions may result from just one value prediction, only one LDPRED instruction is needed for each value prediction.

In step seven, a branch to patch-up code is inserted for repairing miss-predictions. Only one branch per data value prediction is required and the scheduler determines where this branch is inserted. Once the location of the branch is set, all instructions in all dependence chains between the predicted instruction and the branch to patch-up code are candidates for value-speculative execution. It is therefore desirable to schedule any of these instructions above the predicted instruction. Actual hardware resources will restrict the ability to speculatively execute these candidates for value speculation. Also, as all candidates for value speculation are duplicated in patch-up code, their number directly affects the penalty for miss-prediction. These factors affect the scheduler's decision on where to place the branch to patch-up code.

In step eight, patch-up code is created for repairing miss-predictions. The patch-up code contains the UDPRED instruction, a copy of each candidate for value-speculative execution, and an unconditional jump back to the instruction following the branch to patch-up code. The UDPRED instruction uses the same immediate value, assigned by the scheduler, as its corresponding LDPRED instruction for indexing the value predictor. The other source operand for the UDPRED instruction is the destination register of the predicted instruction (the actual result of the predicted instruction). The UDPRED instruction index and the actual result are used to update the value predictor.

Finally, in steps ten and eleven, the data dependence graph is updated to reflect the changes and treegion scheduling is performed. Because of the machine resource restrictions and dependencies, not all candidates for value speculation are speculated above the predicted instruction. Section 4 shows the results of using different threshold values for determining when to do value speculation.

4. EXPERIMENTAL RESULTS

The SPECint95 benchmark suite is used in the experiments. All programs are compiled with classic optimizations by the IMPACT compiler from the University of Illinois [11] and converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories [12]. Then, the LEGO compiler, a research compiler developed at North Carolina State University, is used to insert profiling code, form treegions, and schedule instructions [10]. After instrumentation for value profiling, intermediate code from the LEGO compiler is converted to C code. Executing the resultant C code generates value profiling data.

For the experiments in value speculation scheduling (VSS), load instructions are filtered as targets for value speculation. Load instructions are selected because they are usually in critical paths and have long latencies. Value profiling for load instructions is performed on all programs. Table 1 shows the statistics from these profiling runs. The number of total profiled load instructions represents the total number of load instructions in each benchmark, as all load instructions are instrumented (profiled). The number of static load instructions represents the number of load instructions that are actually executed. The difference between total profiled and static load instructions is the number of load instructions that are not visited. The number of dynamic load instructions is the total of each

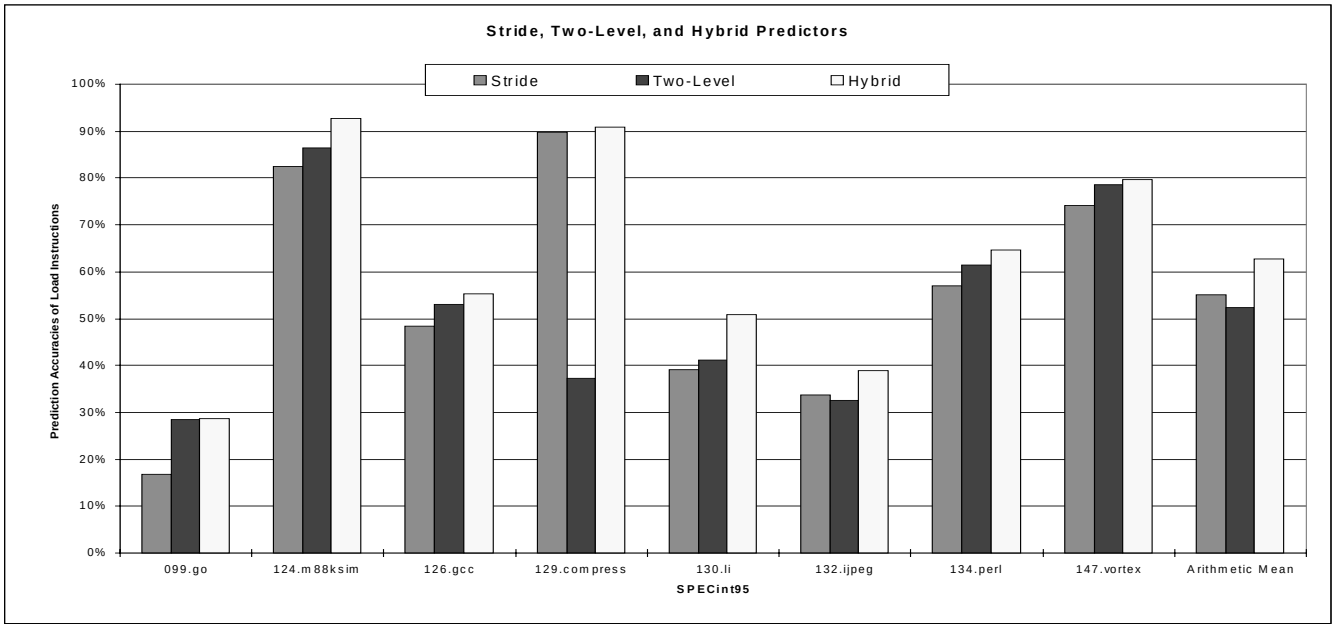

Figure 8. Prediction Accuracy of Load Instructions under Stride, Two-Level, and Hybrid Predictors.

load executed multiplied by its execution frequency.

Stride, two-level, and hybrid value predictors are simulated during value profiling to evaluate prediction accuracy for each load instruction. Since the goal of this paper is to measure the performance of VSS rather than the required capacities of the hardware buffers, no indices conflicts between loads are modeled. An intelligent index assignment algorithm likely will produce results similar to this, but development of such an algorithm is outside the scope of this paper and left for future work. During value profiling, after every execution of a load instruction, the simulated prediction is compared with the actual value to determine prediction accuracy. The value predictor simulators are updated with actual values, as they would be in hardware, to prepare for the prediction of the next use.

SPECint95	Total Profiled Load Instructions	Static Load Instructions	Dynamic Load Instructions
099.go	7,702	6,370	86,613,967
124.m88ksim	2,954	747	15,765,232
126.gcc	35,948	17,418	132,178,579
129.compress	96	72	4,070,431
130.li	1,202	414	24,325,835
132.ijpeg	5,104	1,543	118,560,271
134.perl	6,029	1,429	4,177,141
147.vortex	16,587	10,395	527,037,054

Table 1. Statistics of Total Profiled, Static and Dynamic Load Instructions.

Each entry for the stride value predictor used has two fields, the *stride*, the *current value*. The prediction is always the *current value* plus the *stride*. The *stride* equals the difference between the most recent current values. The stride value predictor always generates a prediction. No finite state machine hardware is required to determine if a prediction should be used.

The two-level value predictor design is as in [7], with four data values and six outcome value history patterns in the value history table of the first level. The value history patterns index the pattern history table of the second level. The pattern history table employs four saturating counters, used to select the most likely prediction amongst the four data values. The saturating counters in the pattern history table increment by three, up to twelve, and decrement by one, down to zero. Selecting the data value with the maximum saturating counter value always generates a prediction.

The hybrid value predictor of stride and two-level value predictors utilizes the previous description illustrated earlier in Figure 6 of section 2. In the hybrid design, the saturating counters, used to select between stride and two-level prediction, also increment by three, up to twelve, and decrement by one, down to zero.

Figure 8 shows the prediction accuracy of load instructions under stride, two-level, and hybrid predictors. The prediction accuracy of the two-level predictor is higher than that of the stride predictor for all benchmarks except 129.compress and 132.ijpeg. However, the average prediction accuracy for the stride predictor is higher than that for the two-level predictor because of the large performance difference in 129.compress. Examining the value trace for 129.compress shows many long stride sequences that are not predicted correctly by the history-based two-level predictor. The hybrid predictor, capable of leveraging the advantages of each prediction method, has the highest prediction accuracy, at 63% on average across all benchmarks.

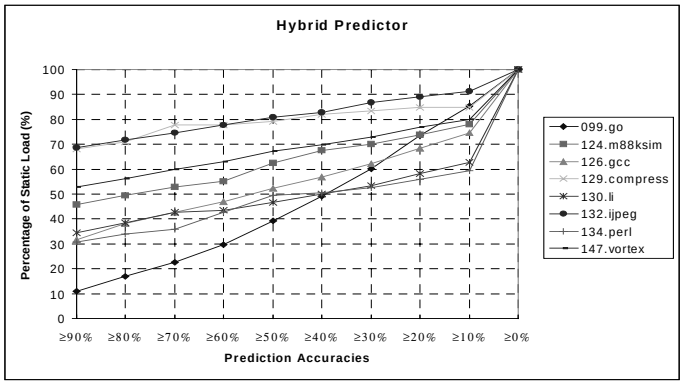

Figure 9. Prediction Accuracy Distribution for Static Load Instructions Using Hybrid Predictor.

Figures 9 and 10 show prediction accuracy distribution for load instructions using the hybrid predictor. Figure 9 is the distribution for static loads and Figure 10 is the distribution for dynamic loads. For 124.m88ksim, 90% of dynamic load instructions have prediction accuracy of 90%. For 129.compress, 80% of dynamic load instructions have prediction accuracy of 90%. For 124.m88ksim, 45% of the static loads have prediction accuracy 90%, representing most of the dynamic load instructions. For 129.compress, 70% of the static loads have prediction accuracy of 90%. These loads are excellent candidates for VSS. Such high prediction accuracy results in low overhead due to the execution of patch-up code. However, for benchmarks 099.go and 132.ijpeg respectively, only 15% and 25% of

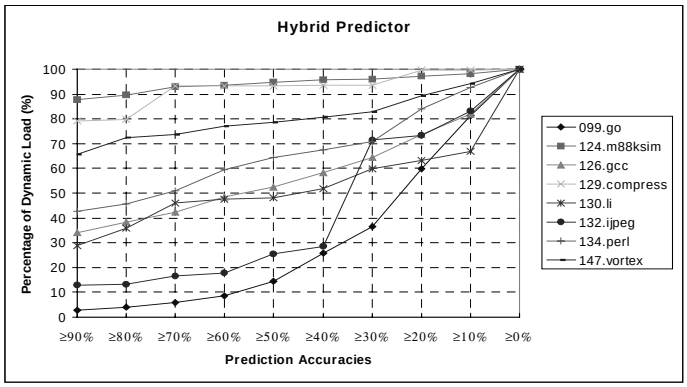

Figure 10. Prediction Accuracy Distribution for Dynamic Load Instructions Using Hybrid Predictor.

dynamic load instructions have prediction accuracy above 50%. Therefore, they will not gain much performance benefit from VSS.

The VSS algorithm of Figure 7 is performed on the programs of SPECint95. Prediction accuracy threshold values of 90%, 80%, 70%, 60% and 50% are evaluated. The number of candidates for value-speculative execution is limited to three for each value prediction. This parameter was varied in our evaluation, with the value of three providing good results.

For the evaluation of speedup, a very long instruction word (VLIW) architecture machine model based on the Hewlett-Packard Laboratories PlayDoh architecture [13] is chosen. One cycle latencies are assumed for all operations (including LDPRED and UDPRED) except for load (two cycles), floating-point add (two cycles), floating-point subtract (two cycles), floating-point multiply (three cycles) and floating-point divide (three cycles). The LEGO compiler statically schedules the programs of SPECint95. The scheduler uses treegion formation [10] to increase the scheduling scope by including a tree-like structure of basic blocks in a single, non-linear region. The compiler performs control speculation, which allows operations to be scheduled above branches. Universal functional units that execute all operation types are assumed. An eight universal unit (8-U) machine model is used. All functional units are fully pipelined, with an integer latency of 1 cycle and a load latency of 2 cycles. Program execution time is measured by using the schedule length of each region and its execution profile weight. The effects of instruction and data cache are ignored, and perfect branch prediction is assumed in an effort to determine the maximum potential benefits of VSS.

Figure 11 shows the execution time speedup of programs scheduled with VSS over without VSS. Five different prediction accuracy thresholds are used to select which load operations are value speculated.

The maximum speedup for all benchmarks is 17% for 147.vortex. As illustrated in Figure 10, 147.vortex has many dynamic load operations that are highly predictable. While 147.vortex does not have the highest predictability for load operations, the sheer number, as illustrated in Table 1, results in the best performance. Benchmarks 124.m88ksim and 129.compress also show impressive speedups, 10% and 11.5% respectively, using a threshold of 50%. Speedup for 124.m88ksim actually goes up, even as the prediction threshold goes down, from 90% to 50%. This result can be deduced from the distribution of dynamic loads. For 124.m88ksim, there is a steady increase in the number of dynamic loads available as the threshold decreases from 90% to 50%. There is a tapering off in speedup though, as more miss-predictions are seen near a threshold of 50%. For 129.compress, the step in the distribution of dynamic loads from 80% to 70% is reflected in a corresponding step in speedup. Performance gains for 126.gcc are more reflective of the large number of dynamic load operations than of their predictability. Penalties for miss-prediction at the lower thresholds reduce speedup for 126.gcc. Benchmark 130.li, with a distribution of dynamics loads similar to 126.gcc, has lower performance due to fewer dynamic loads. Benchmark 134.perl clearly suffers

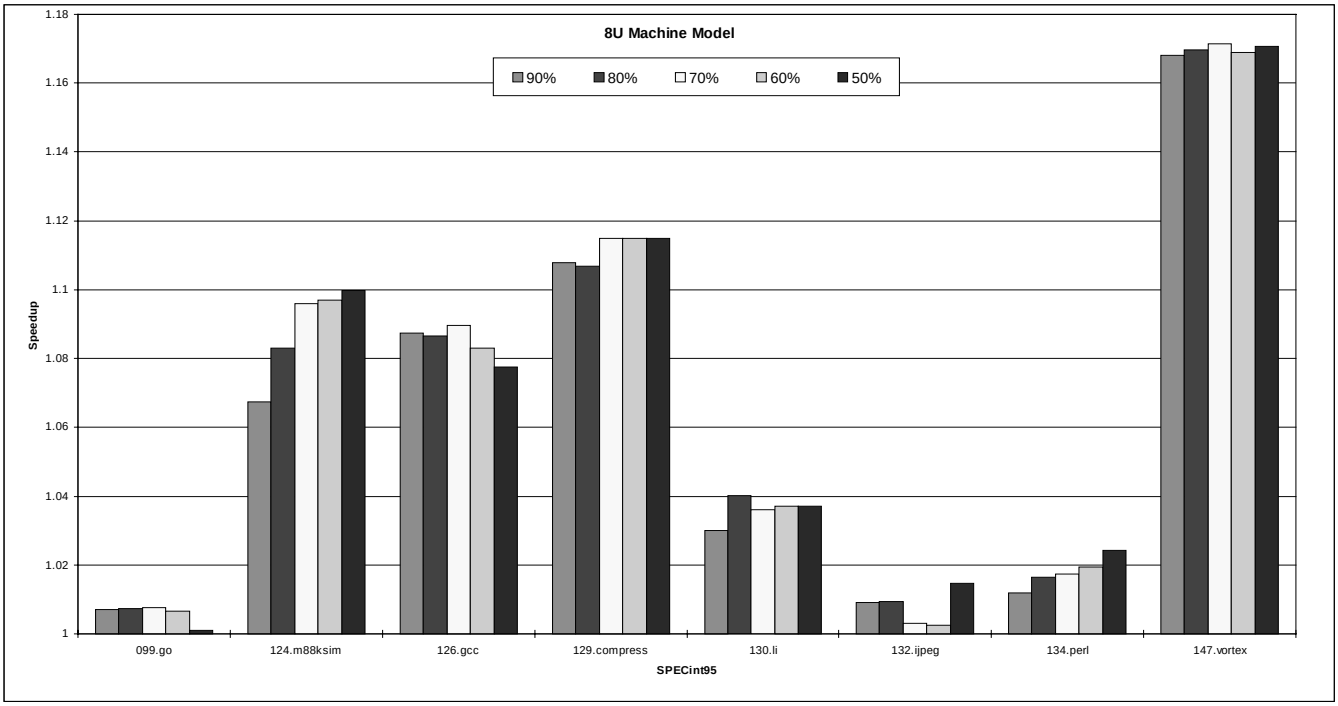

Figure 11. Execution Time Speedup for VSS over no VSS.
Prediction accuracy threshold values of 90%, 80%, 70%, 60% and 50% are used.

from not having many dynamic loads. Benchmarks 099.go and 132.ijpeg do not have good predictability for load operations.

Based on these performance results, a predictability threshold of 70% appears to be a good selection. From the distribution of predictability for dynamic loads in Figure 10, a threshold 70% includes a large majority of the predictable dynamic loads. Choosing a threshold of predictability lower than 70% results in a tapering off in performance for some benchmarks. This is due to both a higher penalty for miss-prediction and saturation of functional unit resources, resulting in fewer saved execution cycles.

5. CONCLUSIONS AND FUTURE WORK

This paper presents value speculation scheduling (VSS), a new technique for exploiting the high predictability of register-writing instructions. This technique leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing ILP. Dynamic value prediction is used to enable aggressive static schedules in which value dependent instructions are speculated. In this way, VSS can be thought of as a static ILP transformation that relies on dynamic value prediction hardware. The results for VSS presents in this paper are impressive, especially when considering that only load operations were considered for value speculation. Future work will include the study of heuristics for selecting register-writing operations in critical paths. Available functional unit resources and remaining data dependencies affect the ability to improve the static schedule and the penalty for patch-up code. VSS should also be applied to operations other than loads based on their predictability and potential benefit to speedup. How many candidates for value-speculative execution (dependent instructions between the predicted instruction and the branch to patch-up code) to allow is also an important parameter. In general, better heuristics for deciding when to speculate values and how many VSS candidates to allow (directly affecting the amount of patch-up code) will be studied.

6. ACKNOWLEDGMENTS

This work was funded by grants from Hewlett-Packard, IBM, Intel and the National Science Foundation under MIP-9625007.

We would like to thank Bill Havanki, Sumedh Sathaye, Sanjeev Banerjia, and other members in the Tinker group. We also thank the anonymous reviewers for their valuable comments.

7. REFERENCES

[1] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhall, W. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 183-195, October 1994.

[2] M. H. Lipasti, C. B. Wilkerson, J. P. Shen, "Value Locality and Load Value Prediction," *Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 138-147, October 1996.

[3] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pp. 226-237, December 1996.

[4] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pp. 248-258, December 1997.

[5] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pp. 259-269, December 1997.

[6] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?," *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pp. 270-280, December 1997.

[7] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pp. 281-290, December 1997.

[8] F. Gabbay and A. Mendelson, "The Effect of Instruction Fetch Bandwidth on Value Prediction," EE Department TR #1127, Technion, November 1997.

[9] F. Gabbay, "Speculative Execution based on Value Prediction," EE Department TR #1080, Technion, November 1996.

[10] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion Scheduling for Wide-Issue Processors," *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, vol. 7, pp. 229-248, January 1993.

[12] R. Johnson and M. Schlansker, "Analysis Techniques for Predicated Code," *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pp. 100-113, December 1996.

[13] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," Hewlett-Packard Laboratories Technical Report HPL-93-80, Computer Systems Laboratory, February 1994.

[14] L. Gwennap, "Intel, HP Make EPIC Disclosure," *Microprocessor Report*, 11(14): 1-9, October 1997.