Value Speculation Mechanisms for EPIC Architectures

Chao-ying Fu Thomas M. Conte[1]
Department of Electrical and Computer Engineering
North Carolina State University
conte@ncsu.edu

Value speculation has the potential of extending instruction level parallelism by breaking the barrier caused by chains of pure dependencies. To date, the literature has focused primarily on implicitly parallel architectures with superscalar cores. This paper presents a scheme for integrating value speculation into an explictly parallel instruction computer ISA, such as IA-64 or HPL-PD. It presents a novel approach to recovering from incorrect speculation via the CHECKPRED instruction semantics. This technique can be coupled with interrupt handling mechanisms or register renaming techniques. The total effect is to eliminate the need for the patchup code. In addition, the paper addresses the selection of operations to value speculate. Using an optimizing compiler of the authors' construction, the critical paths of the SPECint95 benchmarks are analyzed. The interesting and non-intuitive result is that, although slicing a dependence chain in half should yield the highest speedup due to overlapped execution, in practice it doesn't. Results for the maximal speedup resulting from various operation selection alternatives show that the predictability of an operation is also an important criterion for selection. The paper closes with recommendations for compiler-guided value selection decisions in EPIC architectures.

1. Introduction

Chains of pure dependencies are a significant limit to higher levels of instruction level parallelism (ILP). Value speculation has been shown to be a promising approach for breaking this data dependence barrier to higher ILP [1],[2],[3],[4]. To date, the literature has focused primarily on implicitly parallel architectures with superscalar cores. This paper discusses

[1] Because this paper builds on our prior publications, we found it very awkward to write anonymously. We are therefore electing the non-anonymous review option.

techniques to deploy value prediction in a processor that uses an explicitly parallel instruction computing (*EPIC*) ISA. One notable such processor is the Intel/HP IA-64 Merced processor [5],[6]. Value speculation uses prediction mechanisms of register-writing instructions to break long dependence chains. Instructions that consume the predicted instruction's results can then be executed much earlier in the code. When the speculation is incorrect, it must be undone. In [4], a mechanism was proposed that required compiler-generated patchup code to undo speculation. A new mechanism is presented that eliminates this drawback.

The effectiveness of a value speculation technique relies not only on the predictability of operations, but also the potential speedup from predicting operations. Various value predictor designs have been proposed that provide high prediction accuracy [1],[2],[3]. However, work remains to be been done on selecting appropriate operations for value speculation. Previous research either tried to predict every operation [2] or focused only on load operations [1], [3], [4]. This paper presents an analysis of the benefits of various operation selection alternatives. Operations from the top, middle, and bottom of a dependence chain are tested for their predictability and overall post-speculation speedup. In addition, the first load in a dependence chain is also tested. Results suggest a set of guidelines for the compiler to use when applying the proposed EPIC value speculation instruction semantics.

The remainder of this paper is organized as follows. Section 2 describes the proposed EPIC instruction semantics for value speculation scheduling. Section 3 presents the selection of operations, and shows the prediction accuracy and the speedup of an EPIC architecture under different selections. Section 4 concludes the paper and mentions future work.

2. Proposed EPIC Instruction Semantics for Value Speculation Scheduling

Value speculation scheduling (VSS) was introduced in [4], which is summarized briefly here. Previous value speculation research presents an implicit interface between the instructions to be predicted and the prediction hardware. For example, a given load instruction is selected as a candidate for prediction, predicted, and speculated without any input from the instruction format (i.e., no input from the programmer or the compiler). VSS makes this selection explicit by introducing two new instructions:

- **LDPRED R*x*,** *index* - Loads a predicted value from the value prediction hardware into a general-purpose register (R*x*). The *index* specifier is used to select which entry in the value predictor should be used to make the prediction. (Such a decision is made implicitly in most hardware schemes by using, for example, the PC value of the instruction to hash into the prediction table.) To save instruction overhead, LDPRED speculatively updates the value predictor hardware with the predicted value. This speculative update is undone by the patchup code in the case of an incorrect prediction [4].
- **UDPRED R*y*,** *index* - This instruction updates the value predictor with the actual value stored in R*y*. This actual value is the result from the predicted instruction. In [4], UDPRED is used only in the patchup code.

VSS generates patchup code to undo the effects of speculation. Control is transferred to the patchup code based on a comparison between the predicted value (R*x*) and the actual value once it has been calculated by the code. Patchup code has several drawbacks, including code bloat and

the interruption of region formation for global scheduling by the compiler. The latter has the potential to reduce the benefits obtained from VSS.

In the original algorithm of VSS, code transformation and instruction scheduling are done in separate phases. The scheduler inserts LDPRED for selected operation to break flow dependencies, a branch operation to verify the prediction, and a patchup block to re-execute those dependent operations in the case of miss-prediction. The operations between LDPRED and the verifying branch are candidates for value-speculative execution. The number of the candidate operations affects the actual saving cycles and reflects to the penalty cycles of patchup block. However, not all of the candidate dependent operations are executed before the predicted operation. This is due to limited actual machine resources and the remaining data dependencies, both of which restrict parallel execution. Therefore, the value speculation scheduler needs to consider the machine resources and the data dependencies when deciding the location of the verification branch in order to maximize the saving cycles and minimize the penalty cycles. The original algorithm used a location of the verification branch, which results in non-maximum saved cycles and not-minimum penalty cycles for some cases.

In this study, VSS is modified to use the existing hardware branch recovery mechanism to undo incorrect speculation. Thus, the patchup block is not needed in value speculation scheduling. Instead, the recovery from a miss-prediction is performed by hardware. Since EPIC processors support interrupts (see [7], [8]), such hardware is already present. In order to support hardware recovery of miss-prediction, a new operation, CHECKPRED, is introduced:

- **CHECKPRED R*y*, R*x*,** *pc-offset*: The first source operand (R*y*) is the destination register from the predicted operation. The second source operand (R*x*) is the

destination register from the LDPRED operation. The third source operand (*pc-offset*) is a PC-relative pointer to the location of the original LDPRED.

The action of the CHECKPRED operation is in essence an enhanced compare-and-branch instruction. It operates as follows: First, CHECKPRED compares the actual value (R*y*) and the predicted value (R*x*). If they are the same, the prediction is correct and the program proceeds as normal. Otherwise, the miss-prediction occurs and the processor enters recovery mode. CHECKPRED uses the pointer to LDPRED (*pc-offset*) branch to the LDPRED operation. The actual value (R*y*) is copied to the destination register of LDPRED (R*x*). The LDPRED behaves like the (now not needed) UDPRED operation in the recovery mode: it uses the *index* to the prediction table and the actual value (R*y*) to update the value predictors. Then, the processor re-executes those operations after the LDPRED operation, until it meets the CHECKPRED operation. It then exits recovery mode.

The CHECKPRED idea is an adaptation of the interrupt handling and speculation recovery schemes presented in [7], [8]. In [7], the current state buffer for fast interrupt handling allows only the dependent operations are re-executed. In [8], the inline recovery scheme for the IMPACT EPIC architecture achieves a similar result using extra tags in the register file. Applying these techniques to the recovery scheme of value speculation, only dependent opeartions needed to be re-executed. When the CHECKPRED operation is met again, the processor finished recovery mode and return to the normal mode again.

For value-speculative execution, the source operands of the value-speculative executed operations between LDPRED and CHECKPRED need to be protected. This is because when a miss-prediction occurs, the operations must receive their original source operands and be re-

executed. In [7], treating unsafe anti-dependencies as flow dependencies satisfies the requirement, so that no modification of the source operands can be performed within the scope of potentially excepting operations. In [8], using register renaming to split the operation into two parts and to place a copy operation after the check operation ensures this requirement. This guarantees that the modification of the source operations does not occur among value-speculative execution operations. We use the register renaming technique to ensure this requirement in our value speculation scheduling.

Having the new operation, CHECKPRED, allows the value speculation scheduler to perform better scheduling. The new value speculation scheduler inserts the LDPRED and the CHECKPRED operations immediately after the predicted operation. Then, instruction scheduling is performed based on the machine resource and the remaining data dependencies. The scheduler automatically schedules the dependent operations into the location between LDPRED and CHECKPRED. The number of the dependent operations is flexible and reflects to the actual machine resources. Thus, the saved cycles are the maximum and the penalty cycles are reduced to the minimum. Figure 1 shows the revised algorithm of value speculation scheduling.

1. Select the operation to value predict.
2. Insert LDPRED after the predicted operation. The destination register number of LDPRED is the original destination register of the predicted operation.
3. Change the destination register number of the predicted operation to a free register.
4. Insert the CHECKPRED operation after the LDPRED operation. The CHECKPRED operation has two source registers, the first one is the destination register of the predicted operation, the second one is the destination register of the LDPRED operation. In addition, CHECKPRED specifies the location of the LDPRED operation (as a PC offset).
5. Perform the region scheduling to let the scheduler to move operations based on the machine resources and the remaining data dependencies.
6. After scheduling, the pointer to the LDPRED operation in the CHECKPRED operation is updated to the actual location of the LDPRED.

Figure 1: The revised value speculation scheduling algorithm.

Figure 2 shows an example of the new value speculation scheduling algorithm. In Figure 3(a), the original code contains a long dependence chain from I1 to I5. The load instruction I3 is selected because it is in the middle of the dependence chain and has long latency. Figure 3(b) shows code transformation performed by the value speculation scheduler. In Step I, LDPRED and CHECKPRED are inserted right after I6. In Step II, the LDPRED operation, I6, and the dependent operations, I4 and I5, are hoisted above the predicted instruction, I3, by the scheduler. Because the scheduler performs code motion, the machine resources are considered, so maximum benefit is gained.

(a) Original code	(b) Value speculation of R4 (for I3)
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	Step I. After inserting LDPRED and CHECKPRED I1: ADD R1, R2, 5 I2: SHL R3, R1, 2 I3: LW R8, 0(R3) **I6: LDPRED R4, index** // load prediction into R4 **I7:CHECKPRED R8, R4, @I6** // check prediction I4: ADD R5, R4, 1 I5: OR R6, R5, R7 Step II. After scheduling, I6, I4 and I5 are hoisted. I1: ADD R1, R2, 5 I2: SHL R3, R1, 2 **I6: LDPRED R4, index** // load prediction into R4 **I4: ADD R5, R4, 1** **I5: OR R6, R5, R7** I3: LW R8, 0(R3) **I7:CHECKPRED R8, R4, @I6** // check prediction

Figure 2: Example of the new value speculation scheduling algorithm.

3. The Selection of Operations

The selection of operations plays an important role in realizing performance gain from value speculation. It is intuitive that predicting those operations that reside in the beginning or at the end of the dependence chains will not exhibit benefit, since the resulting parallelism is not

enhanced significantly. Most hardware schemes and the VSS scheme as presented in [4] selected load operations. However load operations tend to start dependence chains. Intuition also suggests that a better choice would be selecting the operations in the middle of the dependence chains and breaking the flow dependencies by predicting these operations. This section presents experimental data to confirm, and in some cases refute these intuitive conclusions.

A static analysis of the data dependence graph (DDG) identifies the location of operations in the dependence chains. In this work, treegion scheduling was used for global scheduling [9]. A treegion is a region of acyclic control flow that has one entrance and multiple exits. It is more general than a superblock because it encompasses multiple paths of control (hence, it is a tree shaped region, or a *tree-gion*). One other interesting property of treegions is that their formation is profile data independent. Profile data is used later in the global scheduling process by the list scheduler's priority function [9].

The data dependence graph (DDG) of each treegion in each program in the benchmark set was constructed using the techniques from [9]. The dependence types between operations are flow-, anti-, output-, control- and memory dependencies. The control dependencies exist between branch operations. The memory dependencies are used for load and store operations that cannot be statically disambiguated. After constructing the data dependence graph, the bottom-up depth-first-search (DFS) height assignment algorithm is performed on the DDG. All operations that do not have succeeding dependent operations in a treegion are assigned as height 0. The height of the operation is the sum of the maximum height of their succeeding operations and its own operation latency. After height assignment, the critical paths (the longest dependence chain)

are identified for each treegion as the paths from the operations with the maximum height to the operations with height 0 (the same as the reverted paths in a depth-first-search).

For the purposes of this study, the SPECint95 benchmarks were used. All programs were 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 Nortah Carolina State University, was used to insert profiling code, form treegions, construct data dependence graphs, analyze critical paths, and schedule instructions [9].

Table 1 shows the statistics of the critical paths in treegions. The average length of critical paths in treegions is about 10 cycles, except 147.vortex whihc has the biggest average length, 14.62 cycles. Comparing the maximum length in tregions, 126.gcc, 134.perl, and 147.vortex have big number, 295, 274, and 214 cycles. 129.compress has the smallest maximum length, 28 cycles.

Table 1: Statistics for the critical path length across benchmarks.

SPECint95	# of treegions	Average length	Maximum length
099.go	2317	10.25	78
124.m88ksim	883	10.59	78
126.gcc	12106	10.64	295
129.compress	67	8.01	28
130.li	671	9.04	107
132.ijpeg	1549	10.15	96
134.perl	2383	10.34	276
147.vortex	3164	14.62	214

3.1 Critical path characteristics

In order to show the effect of the selection of operations on value speculation, we selected operations at five locations in the critical path of each treegion: **top** (the first register-writing operations in the chain), **middle** (the register-writing operations in the middle of the chain), **bottom** (the last register-writing operations in the chain), **first_load** (the first load operations in the chain), and **first_load+middle** (the union of the operations in the **middle** and **first_load** categories). The selection criterion for **top** is those operations whose height is the same as the maximum height in the treegion. Those operations whose height is between floor(*max_height*/2) and floor(*max_height*/2)+1 belong to the **middle** category. The **bottom** category contains the operations which are in the last three levels of the dependence graph.

Table 2: Statistics for the number of selected operations in each location.

SPECint95	Top	middle	bottom	first_load	first_load+middle
099.go	1548	1215	96	1751	2788
124.m88ksim	233	92	3	158	235
126.gcc	4893	2415	314	4133	6319
129.compress	37	20	2	25	43
130.li	177	91	7	90	170
132.ijpeg	443	273	21	369	618
134.perl	542	294	29	298	584
147.vortex	1973	395	25	1781	2142

Table 2 shows the statistics for the number of selected operations in each category. Most dependence graphs were inverted trees, so that there were more operations at the top of the chains than at the bottom. This explains why the number of operations in **top** is larger than in **middle** and **bottom**; and, why the number of operations in **middle** is also larger than **bottom**. Note that, operations in the **first_load** category may reside in any location of the dependence

chain, **top**, **middle, bottom**, or elsewhere. Because of this potential for overlap, the number of operations in **first_load+middle** does not equal the sum of the number of **first_load** and of **middle**.

3.2 The compiler value profiling method

The code was value profiled to determine the effectiveness of selecting operations in each category. A hybrid value predictor was simulated during value profiling (see below). After every execution of the selected operations, the simulated prediction was compared with the actual value to determine prediction accuracy. The value predictor simulators were then updated with actual values (as they would be in hardware) in order to prepare for the prediction at the next use.

The hybrid predictor used by the profiler consists of the stride and two-level value predictors [10]. Each entry for the stride predictor used has two fields, the *current value* and the *stride*. The stride equals the difference between the most recent two current values. The prediction is always *current value*+*stride*. The two-level value predictor design is from [10], with four data values and six outcome value history patterns in the value history table of the first level. The value history pattern indexes the second-level pattern history table. The pattern history table employs four saturating counters, used to select the most likely prediction among 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. In the hybrid predictor 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. Since the goal of value profiling is to

measure the potential benefit of value speculation rather than the required capacities of the hardware buffers, no index conflicts between operations are modeled. If this were actual hardware instead of a software profiling technique for compiler feedback, it would experience conflicts (although these would be minimized due to the compiler-assigned *index* field of the LDPRED operation).

Figure 3 shows the prediction accuracies of operations belonging to the five locations using the hybrid predictor. In four programs, 129.compress, 130.li, 132.ijpeg, and 147.vortex, the prediction accuracy of **top** is higher than the prediction accuracy of **middle**, which is higher than that of **bottom**. Examination of the code shows that operations on the top of the dependence chains usually perform initialization from global data or input parameters, or address calculation for load operations. Thus, they are highly predictable. In the middle of the dependence chains, the operations usually calculate the internal data, so they are not as predictable as the operations in **top**. However, 124.m88ksim, 126.gcc and 134.perl are exception. The operations in **middle** have better prediction accuracy than those in **top**. The small of selected operations in **bottom** may affect the prediction accuracy, that is the lowest among all. The prediction accuracy of **first_load** operations is very high, largely due to these loads loading the same global values. The order of harmonic mean of prediction accuracies across all benchmarks from high to low is **top**, **first_load**, **first_load+middle**, **middle**, and **bottom**.

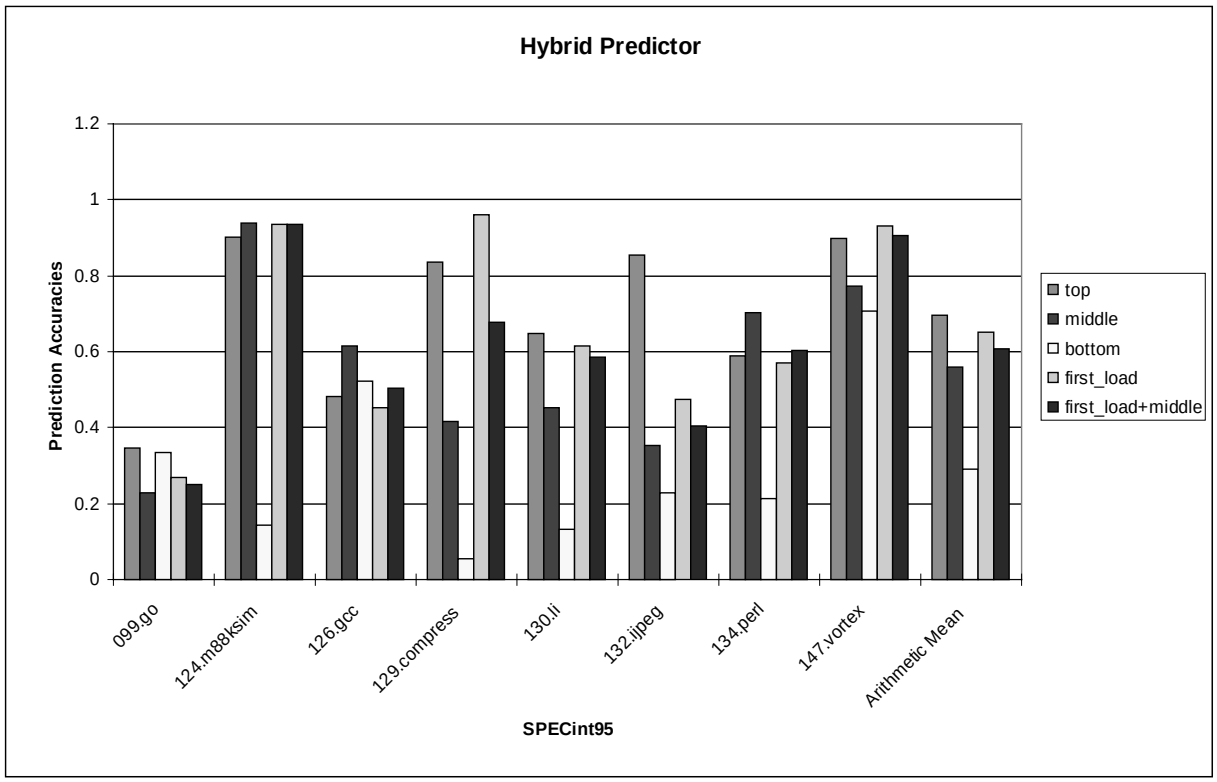

Figure 3: Prediction accuracies for top, middle, bottom, first_load, and first_load+middle operations (see text for definitions of these selection categories).

3.3 The benefits of the selection alternatives

Although the middle operations do not have the highest prediction accuracy, they have the potential of providing the highest benefit. To test this, the overall speedup after applying the modified VSS was measured. For evaluation of the speedup, the TINKER EPIC architecture was used. TINKER incorporates an expanded version of the HPL-PD (PlayDoh) architectural framework [14]. This architecture has many properties in common with IA-64:

- Operations are grouped into packets, where each represents a collection of guaranteed-independent operations.
- All operations support predicated execution.
- All operations support non-faulting speculation using Sentinel Scheduling [13].
- The recovery scheme for value miss-prediction allows only dependent operations to be re-executed [7], [8].

- Branches are decomposed into prepare to branch (PBR), compare-to-predicate (CMPP) and the control flow transfer point (BR). This results in a programmatic forward slot that can hide most misprediction penalty.

To find the speedup potential of the five operation selection alternatives, an eight universal unit (8-U) machine model was used. All functional units are fully pipelined, with an integer latency of 1 cycle and a load latency of 2 cycles. Program execution time was measured by using the schedule length of each region and its execution profile weight. Because the goal here is to determine the relative ranking between schemes, not the absolute benefit of any one scheme, instruction and data caches are not modeled. If this were a study to determine if VSS is appropriate for a given EPIC implementation, then these resources should be modeled. That is not the goal of this study, but it is planned for a future study of a specific TINKER implementation.

Table 3: The statistics for the number of predicted operations in each location.

SPECint95	**top**	**middle**	**bottom**	**first_load**	**first_load+middle**
099.go	366	144	18	311	433
124.m88ksim	126	45	0	88	125
126.gcc	1854	1121	107	1488	2532
129.compress	29	7	1	22	28
130.li	77	34	1	37	68
132.ijpeg	352	150	9	272	405
134.perl	145	70	7	103	169
147.vortex	1212	171	7	1291	1435

Any operation with a prediction accuracy less than 70%, based on the results of the profile run, was not selected for value speculation. This filter threshold is from the conclusions of [4]. Table 3 shows the statistics for the number of predicted operations in each category. The filter threshold causes the largest number of operations selected to be **first_load+middle** for all but a

few benchmarks, and it is a very close second for those two (129.compress, 130.li). The **top, first_load** and **middle** categories are then selected with the next highest frequency. Finally, the **bottom** category is rarely selected. This is consistent with the low prediction accuracy for this category, as witnessed in Figure 3.

Figure 4 shows the execution time speedup of programs scheduled with VSS over those without VSS. The five alternative selections were used, **top**, **middle**, **bottom**, **first_load** and **first_load+middle**. These results take the miss-prediction penalty into account. It is clear from the results that **middle**, **first_load** and **first_load+middle** are better alternatives than **top** or **bottom**. This serves to confirm intuition. What is counter-intuitive is that **middle** achieves lower speedup than **first_load** for 099.go, 126.gcc, 129.compress, 134.perl and (most significantly) 147.vortex. In the other cases, the advantage that **middle** gives over other alternatives is either very small or nonexistent. The hybrid alternative, **first_load+middle**, places either first or second for all but 132.ijpeg (for which **first_load+middle** had very low prediction accuracy).

The results from this section suggest a selection heuristic for compiler-driven VSS for EPIC architectures:

1. Select operations that are the first load of the critical path, or are in the middle of the dependence chain.
2. Perform value profiling of these operations using a hybrid predictor.
3. Value speculate only those selected operations that achieved high accuracy according to the value profiling result.

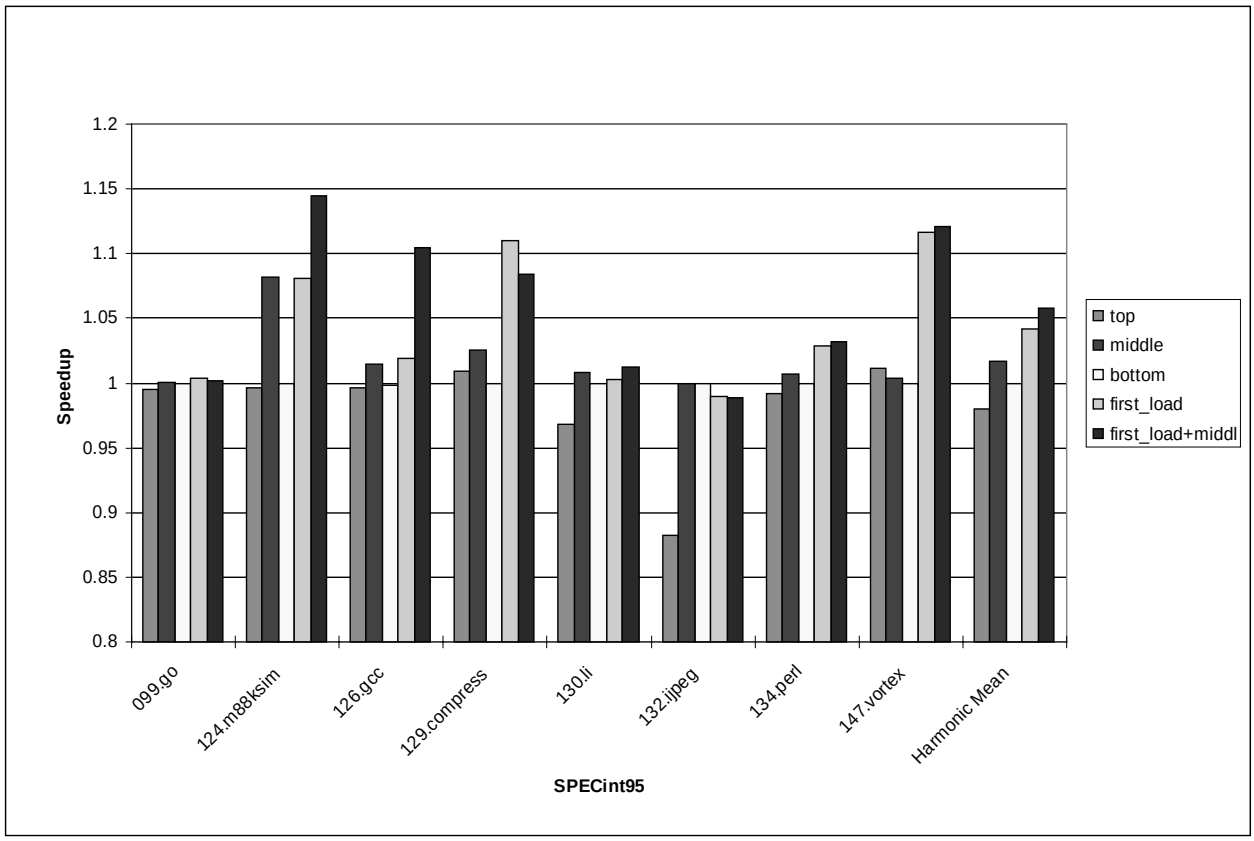

Figure 4. Execution Time Speedup for VSS over no VSS using five different selection alternatives.

4. Conclusions

Value speculation has the potential of extending instruction level parallelism by breaking the barrier caused by chains of pure dependencies. This paper presents a scheme for integrating value speculation into an explictly parallel instruction computer ISA, such as IA-64 or HPL-PD. It presents a novel approach to recovering from incorrect speculation via the CHECKPRED instruction semantics. This technique can be coupled with interrupt handling mechanisms or register renaming techniques. The total effect is to eliminate the need for the patchup code described in [4].

Even if value speculation has high potential benefit, reaping the benefit requires careful application of the technique. Results showed that there is a tradeoff between the potential benefit of speculating a value in the middle of a dependence chain and the accuracy of the prediction of that value. If the prediction has very low accuracy, the high payoff of slicing a critical path in half cannot be realized. This paper presented analysis of several options for selecting the operation. The interesting and non-intuitive result was that breaking the chain at the first load was often more profitable than breaking it in the middle. Taken together, the results suggest that the compiler must make its speculation decision with profile results in-hand. Work needs to be done to either make such profiles reasonable to collect (following the anology to [15]) or to develop static estimates to supplant or augment the value profile information (following the anology to [16]).

References

[1] 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*, pp. 138-147, October 1996.

[2] 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.

[3] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin, "Profile Guided Load Marking for Memory Renaming," University of California, San Diego, Technical Report UCSD-CS98-T93, July 1998.

[4] C. Fu, M. D. Jennings, S. Y. Larin and T. M. Conte, " Value Speculation Scheduling for High Performance Processors," *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, (San Jose, CA), Oct. 4-7, 1998.

[5] C. Dulong, "The IA-64 Architecture at Work," *COMPUTER*, vol. 31 no. 7, pp. 24- 32, July 1998.

[6] L. Gwennap, “Intel, HP Make EPIC Disclosure,” *Microprocessor Report*, vol. 11, no. 14, pp. 1-9, October 1997.

[7] E. Ozer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings and T. M. Conte, "A fast interrupt handling scheme for VLIW processors," *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, (Paris, France), Oct. 1998.

[8] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, “Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture,” *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[9] 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)*, (Las Vegas), Feb. 1998.

[10] 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.

[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 structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 229-248, Jan. 1993.

[12] R. Johnson and M. Schlansker, “Analysis Techniques for Predictaed Code,” *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, (Paris, France), pp. 100-113, Dec. 1996.

[13] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," ACM Trans. Computer Systems, vol. 11, pp. 376-408, Nov. 1993.

[14] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[15] K. N. P. Menezes, ``Hardware-based profiling for program optimization," Ph.D. thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, 1997.

[16] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, (Albuquerque, NM), pp. 300-313, June 1993.