Software-Only Value Speculation Scheduling[*]

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
(919) 513-2013
{cfu, mdjennin, sylarin, conte}@eos.ncsu.edu

Abstract

Recent research in value prediction, including several recent publications for MICRO, 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. A hardware and software based technique, value speculation scheduling (VSS), combines static instruction level parallelism (ILP) scheduling techniques with dynamic value prediction hardware. This paper extends the previous work on VSS to a software-only based scheme for VSS, which we will call *software value speculation scheduling (SVSS).* The advantages of SVSS are that no instruction set architecture (ISA) extensions and no value prediction hardware is required. Therefore, SVSS is applicable to existing microarchitectures such as Intel's P6, Digital's Alpha, Sun's UltraSparc and IBM/Motorola's PowerPC. As with VSS, static ILP scheduling techniques are used for SVSS to speculate value dependent instructions by scheduling them above the instructions whose results they are dependent on. For SVSS, compiler generated instructions are used to produce value predictions in place of the prediction hardware of

[*] Review copy. Do not distribute

VSS. These predictions allow the execution of speculated instructions to continue. Again, 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 for applying SVSS to select load instructions in the SPECint95 benchmarks are performed. Speedup of up to 8% has been shown for using SVSS. Empirical results on the software-only value predictability of loads, based on value profiling data, are also provided.

Keywords: Value speculation, value prediction, VLIW instruction scheduling, instruction level parallelism, ILP

1. Introduction

Modern microprocessors extract instruction level parallelism (ILP) by using branch prediction to break control dependencies [14], dynamic memory disambiguation to resolve memory dependencies [1] and register renaming to eliminate anti and output data dependencies [15]. However, current techniques for extracting ILP are still insufficient, especially for integer benchmarks. Recent research has demonstrated the viability of value prediction as a technique for hiding flow dependencies (also called true dependencies) [2], [3], [4], [6], [7], [8], [9] [16]. Results have shown that values produced by many register-writing instructions can be highly predictable using various value predictors: last-value, stride, context-based, two-level, or hybrid predictors. This prior work illustrates that value speculation in future high performance processors will be useful for breaking flow dependencies, thereby exposing more ILP.

There is prior work in value speculation that focuses on hardware-only schemes [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 illustrates the pipeline stages for value speculation utilizing a hardware scheme.

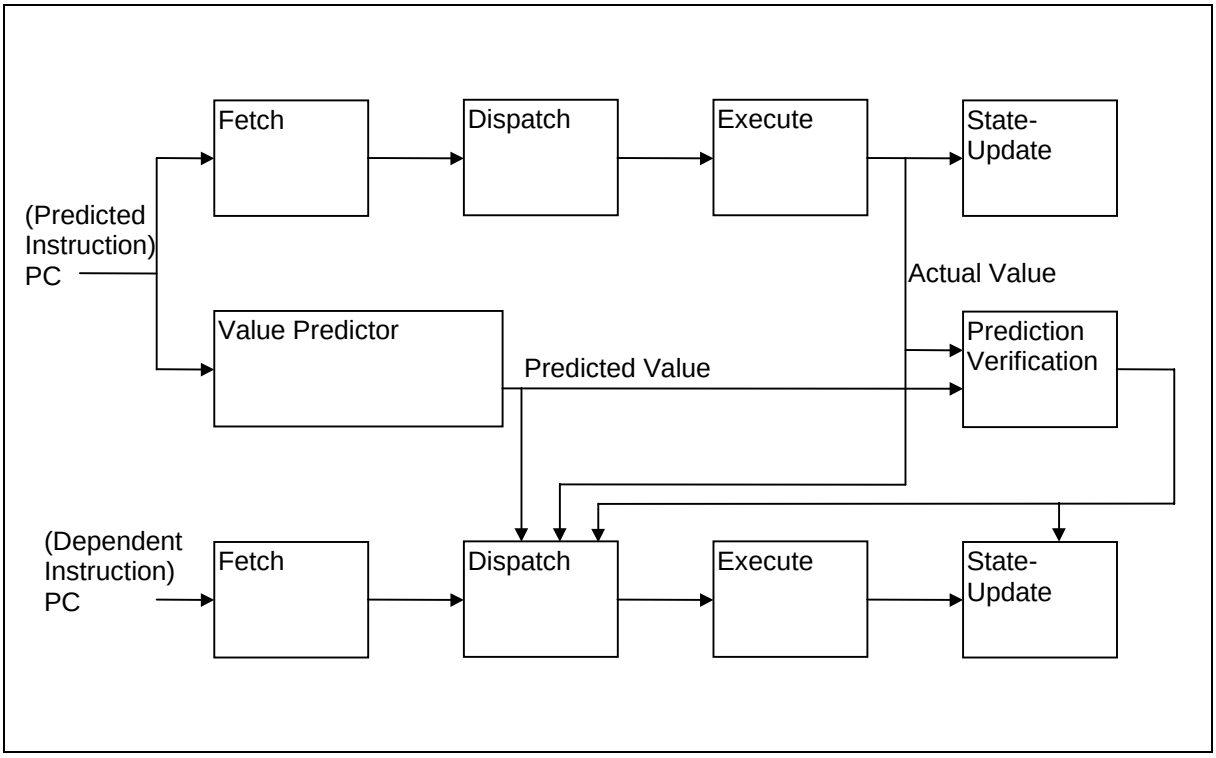

Figure 1. Pipeline Stages of Hardware Value Speculation Mechanism for Flow Dependent Instructions. The dependent instruction executes with the predicted value in the same cycle as the predicted instruction.

This paper extends Value Speculation Scheduling (VSS). Value Speculation Scheduling [16] combines ISA, hardware and compiler synergies for exploiting value predictions using static speculation of instructions dependent on predictable values. VSS improves performance by aggressively scheduling flow dependencies that are highly likely to be eliminated through correct value prediction. Value profiling is used to statically select predictable values for applying VSS. Patch-up code generated by the compiler ensures correct program execution in the event of value miss-predictions. Value prediction hardware dynamically provides values that allow for the speculative execution of value dependent instructions. ISA extensions are necessary, in the form of two new instructions. A load prediction (LDPRED) instruction is required to load predictions from the value prediction hardware to registers. An update prediction (UDPRED) instruction is necessary to update predictor state in the case of miss-predicted values. Hardware pipeline stages for the VSS scheme are shown in Figure 2.

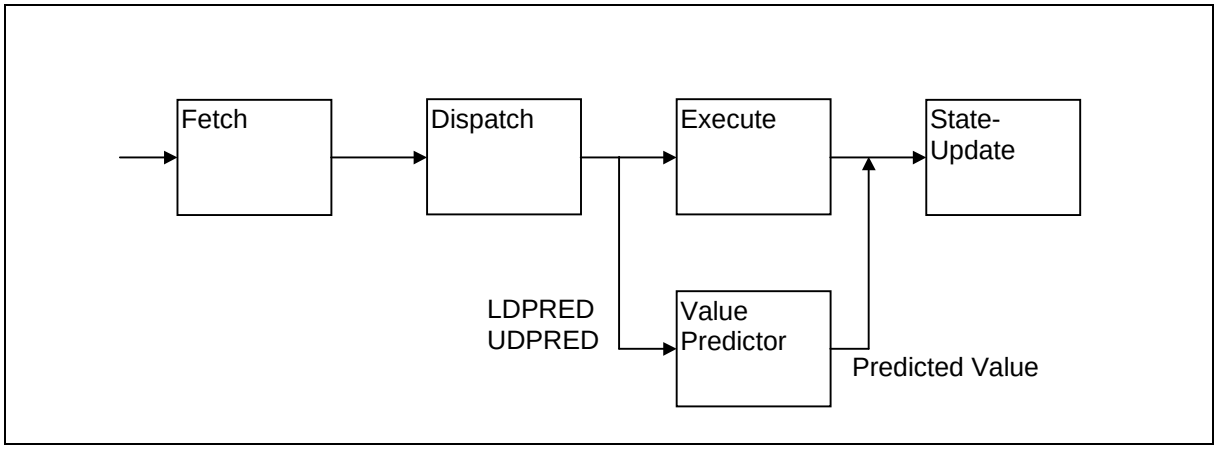

Figure 2. Pipeline Stages of Value Speculation Scheduling Scheme. LDPRED and UDPRED interface with the value predictor during the execution stage.

In this paper, the VSS approach to value speculation is extended so that it can be used as a software-only tool for increasing ILP. The main advantage software value

speculation scheduling (SVSS) is that it may be applied to existing microarchitectures because ISA extensions and value speculation hardware are not required. Emulating hardware branch predictors in software requires more code instrumentation than in the original VSS approach, thereby affecting speedup performance. Due to code expansion and overhead constraints, static stride prediction is used in SVSS because of simplicity when compared to dynamic stride, context-based, two-level, or hybrid predictors. While last-value predictors are the simplest to implement, they do not provide enough prediction accuracy to provide sufficient speedup. To fully emulate the performance of hardware value predictors in software, load and store operations are needed to allow spilling the state for software versions of the value predictors upon procedure entry and exit. We also look at the performance of value predictors that do not spill to memory, resulting in less code instrumentation overhead but un-initialized value predictor state each time the procedure containing the emulated value predictor is entered.

The remainder of this paper is organized as follows: Section 2 presents an overview of VSS while at the same time introduces SVSS via an example. Section 3 provides more details about the SVSS scheme and compiler implementation details. Section 4 presents experimental results for SVSS, including value profiling data and speedup. Section 5 concludes the paper and mentions future work.

2. Value Speculation Scheduling (SVSS/VSS) example

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 instruction I3 to I4 is broken, via VSS or SVSS, 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, 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.

(a) Original code			(b) Code after VSS applied			(c) Code after SVSS (static stride) applied		
I1:	ADD	R1 ← R2, 5	I1:	ADD	R1 ← R2, 5	I1:	ADD	R1 ← R2, 5
I2:	SHL	R3 ← R1, 2	I2:	SHL	R3 ← R1, 2	I2:	SHL	R3 ← R1, 2
I3:	LW	R4 ← 0(R3)	I3:	LW	R4 ← 0(R3)	I3:	LW	R4 ← 0(R3)
I4:	ADD	R5 ← R4, 1	**// load prediction from hardware**			**// calculate software static stride prediction**		
I5:	OR	R6 ← R5, R7	**I7:**	**LDPRED**	**R8 ← index**	**I7:**	**ADD**	**R8 ← R8 + stride**
I6:	SW	0(R3) ← R6	**I4':**	**ADD**	**R5 ← R8, 1**	**I4':**	**ADD**	**R5 ← R8, 1**
Next:	.....		**I5':**	**OR**	**R6 ← R5, R7**	**I5':**	**OR**	**R6 ← R5, R7**
			I6':	**SW**	**0(R3) ← R6**	**I6':**	**SW**	**0(R3) ← R6**
			// verify prediction			**// verify prediction**		
			I8:	**BNE Patchup**	**R8, R4**	**I8:**	**BNE Patchup**	**R8, R4**
			Next:	.....		Next:	.....	
			Patchup:			Patchup:		
			// update prediction to hardware			**// update software static stride predictor**		
			I9:	**UDPRED R4, index**		**I9:**	**MOVE**	**R8 ← R4**
			I4:	ADD	R5 ← R4, 1	I4:	ADD	R5 ← R4, 1
			I5:	OR	R6 ← R5, R7	I5:	OR	R6 ← R5, R7
			I6:	SW	0(R3) ← R6	I6:	SW	0(R3) ← R6
			I10:	**JMP Next**		**I10:**	**JMP Next**	

Figure 3: Example of Code Transformations for VSS and SVSS assuming a static stride predictor. Instruction I3 of the original code sequence is value predicted.

For the VSS schedule 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. A source 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 the 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 the patch-up code, the program jumps to the next instruction after I8 and execution proceeds as normal.

The SVSS schedule in Figure 3(c) is analogous to the VSS schedule except that instruction I7 adds a constant stride to register R8 (which again holds the value prediction). The constant stride value is determined through value profiling and then is permanently fixed by the compiler. This ADD instruction is inserted in place of the LDPRED instruction of the original VSS scheme and emulates in software a static stride value predictor. It is interesting to note that this I7 ADD instruction could be combined with the I4' ADD instruction for this example. This opportunity for instruction combining is not true in general and we are not taking advantage of it at this time. Also, in the SVSS schedule, a MOVE instruction updates the software prediction in the case of an incorrect prediction. This replaces the UDPRED of the original VSS schedule. In this example, the potential speedup due to exposed ILP and the penalty for miss-predicted

values are the same for both the original VSS and the new SVSS scheme for static stride. Actual performance speedup depends on the prediction accuracy of the hardware prediction mechanism and the accuracy of the static stride software predictor.

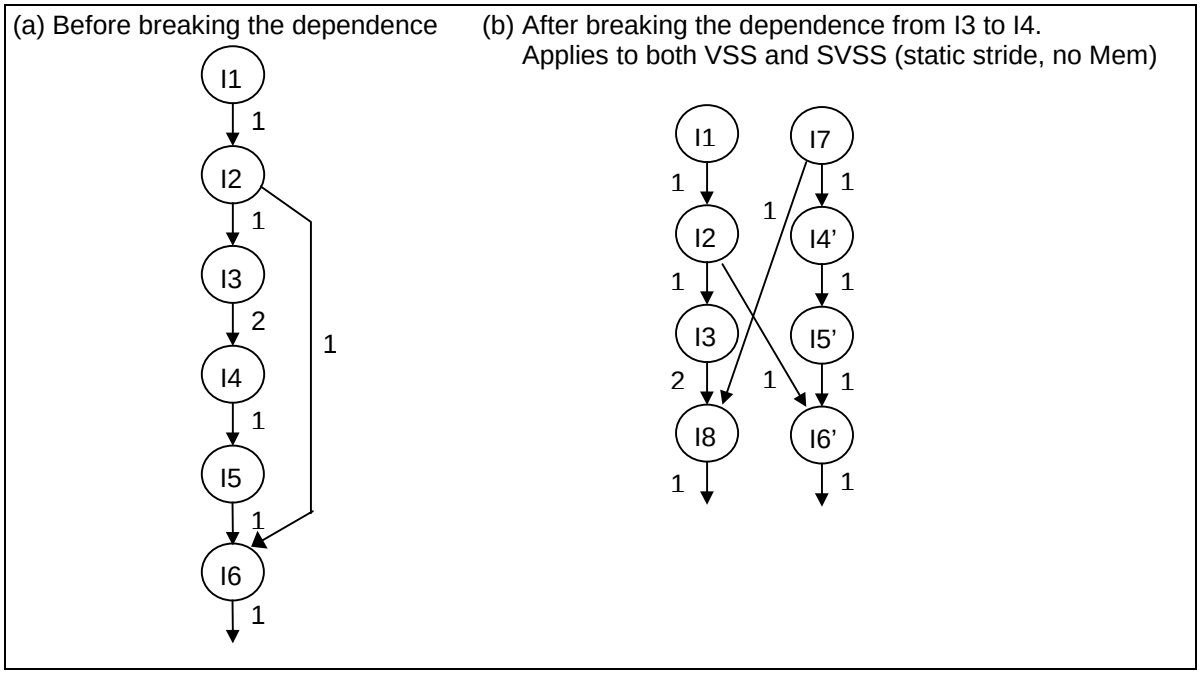

Figure 4. Data Dependence Graphs for Codes of Figure 3. The numbers along each edge represent the latency of each instruction. In 4(a), the schedule length is seven cycles. In 4(b), because of exposed ILP and dependence height reduction, the schedule length is reduced to five cycles.

We see several possible advantages to both the VSS and SVSS schemes, including:

1. Value speculative dependent instructions can execute as early as possible before the predicted instruction that they depend on. This is possible because the value predictor is no longer in the critical path, as it is in the hardware-only approaches.
2. The compiler determines good candidates for value prediction based on value profiling. Only instructions that the compiler deems are good candidates for predictions are then predicted, minimizing any penalty for a miss-prediction.
3. 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.

4. Patch-up code is automatically generated, reducing the need for elaborate hardware recovery techniques.

The software-only approach, SVSS, offers the following additional advantage that as a software-only technique, no additional hardware or extensions to existing ISAs are required. Therefore, SVSS can be applied to existing microarchitectures.

The original hardware-software approach, VSS, uniquely provides the ability to use more elaborate prediction schemes that can take advantage of changes in the values to be predicted for enhanced prediction accuracy. Also in VSS, the compiler controls the number of predicted values and assigns different indices to them for accessing the prediction table. This reduces index table conflicts when compared to the hardware-only schemes that index using a portion of the PC.

There is a drawback to both VSS and SVSS. 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 SVSS technique is the most sensitive to dynamic changes in values, as emulated software prediction schemes that are able to adjust to changes in value profiles are more expensive to implement in terms of code instrumentation. The sensitivity of SVSS for benchmark inputs other than training inputs used for value profiling is shown in section 4.

For SVSS, the insertion of code that emulates a value predictor is analogous to VSS, as indicated in the example for a static stride predictor. Otherwise, the SVSS

algorithm is the same as the VSS algorithm. Figure 5 summarizes the SVSS/VSS algorithm. Due to space constraints, the reader is referred to our original VSS paper for the details on this algorithm, a discussion of the penalty incurred for executing patch-up code and the requirements of hardware value predictors necessary to support VSS [16].

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 software value predictor code (SVSS) or LDPRED (VSS) after Predicted Instruction (selected instruction of step 4)**
6. **Change Source Operand of Dependent Instruction(s) to Destination Register of software value predictor (SVSS) or LDPRED (VSS)**
7. **Insert Branch to Patch-up Code**
8. **Generate Patch-up Code (which also updates software value predictor (SVSS) or contains UDPRED (VSS))**
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 in program**

Figure 5. Algorithm for Software Value Speculation Scheduling (SVSS) and Value Speculation Scheduling (VSS).

3. Software Value Speculation Scheduling (SVSS)

For our implementation of SVSS, a static stride value predictor is emulated in software. A static stride predictor was chosen both because of its simplicity to implement and because of its performance on SPEC95 CINT benchmarks. Adding a constant value to a register holding the previous value requires one ADD instruction to generate static stride predictions. The compiler statically assigns the constant value based on profiling data. A last-value predictor was also analyzed because of its ease of implementation: the

register value is assumed to be correct, requiring no instructions to be inserted for generating predictions. Prediction accuracy for a last-value predictor was high enough to obtain non-negligible speedup on only one CINT benchmark, speedup of up to 6% on 124.m88ksim. Dynamic stride predictors, while still relatively simple, require two register values (current value and previous value), a subtraction for calculating the stride, a move for updating the state of the previous value and an addition using the stride to generate a new prediction for the current value. We have not yet analyzed dynamic stride predictors because of this overhead. Other value prediction methods (context-based, two-level and hybrid) incur too much overhead to implement in software effectively.

To fully emulate the performance of a hardware static stride predictor in software, load and store operations are needed to allow spilling of the current value to memory upon procedure exit and loading of the current value to a register upon procedure entrance. If spill code is not inserted, prediction accuracy suffers as the static stride predictor will be uninitialized each time control transfers to the procedure. Prediction accuracy may still be good though for value predictions of data contained in intra-procedure loop code. In this case, assuming that the correct static stride is predicted, only the first uninitialized iteration of the loop results in a miss-predicted value. Therefore, we also look at the performance of value predictors that do not spill to memory, resulting in less code instrumentation overhead. In the experimental results section, the version that fully emulates static stride predictor hardware is called *with-memory* while the version that allows uninitialized values is called *without-memory*. Allowing the without-memory version introduces an interesting issue that is generally regarded as taboo for a compiler:

generation of uninitialized registers. In our case, for SVSS, uninitialized references will result in a branch to patch-up code, which will guarantee correct program semantics.

Using a register to store the current value state for a software static value predictor will result in increased register lifetimes and increased register pressure. For the with-memory version that uses spill code, the lifetime for the register spans the entire procedure. For the without-memory version, the lifetime may be confined to a loop. For each version, the additional register pressure is proportional to the number of data values that are predicted within a procedure. Our experiments assume memory virtual registers so that an upper-bound on performance can be provided. While our experiments assume memory virtual registers, we will also provide statistics for the number of data value predictions on a procedural level to provide insights into register requirements.

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, 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 results 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 load executed multiplied by its execution frequency. The results of Table 1 are for the training inputs for these benchmarks. The number of dynamic load instructions combined with the prediction accuracy of each benchmark gives a strong indication of the opportunity for speedup using SVSS.

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

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	35,847	17,127	132,056,116
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

Profiling is also used to determine prediction accuracy for the different types of value prediction schemes. Static stride predictors, the with-memory version and the without-memory version, are profiled for the SVSS scheme. Both versions are simulated during value profiling to evaluate prediction accuracy for each load instruction. During value profiling, after every execution of a load instruction, the simulated prediction is compared with the actual value to determine prediction accuracy. Value predictor state is accurately maintained to prepare for the prediction of the next use.

Figure 6 shows the results of the value prediction profiling experiments. Results are presented for each predictor for SPEC95 CINT training and test inputs. In the legend descriptions, (**train**) indicates that the static stride constant was determined using the training inputs and (**test**) indicates that this constant was determined using the test inputs. Both results are included to give an indication if the "test" inputs can successfully work as training inputs when determining speedup for executing with the "train" inputs. Benchmark 130.li shows the most sensitivity to determining the static stride constant with one set of inputs and executing the program using the other. The other benchmarks show little prediction accuracy sensitivity to input shifts.

From the arithmetic mean of these results, the accuracy of static stride without-memory is about one-half of the accuracy of static stride with-memory. Individually for 132.ijpeg, the accuracy of the without-memory and with-memory versions are similar. This indicates that most of the load values of 132.ijpeg that can be predicted accurately with stride predictors are found in **intra**-procedural loops. Conversely for 147.vortex, the high prediction accuracy of static stride with-memory indicates that static stride is appropriate for these loads, but the much lower prediction accuracy for the without-memory version indicates that a lot of uninitialized references result from **inter**-procedural loops.

The relatively low prediction accuracy for static stride without-memory indicates limited opportunities for speedup using SVSS, with the possible exception of 124.m88ksim. The higher prediction accuracy for static stride with-memory suggests some opportunity for speedup using SVSS although the implied presence of inter-procedural loops for 147.vortex is worrisome.

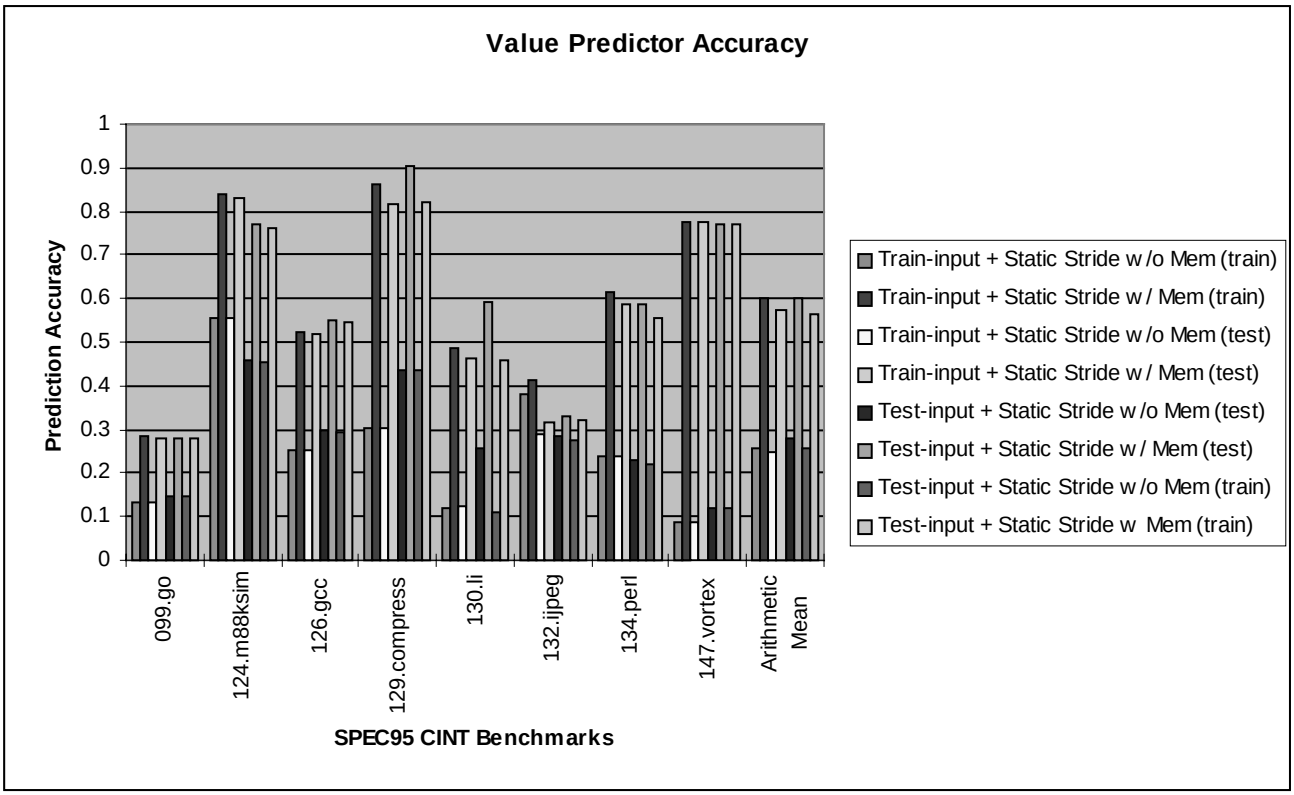

Figure 6. Prediction Accuracy of Load Instructions using Software Static-Stride with-Memory (SVSS) and Software Static-Stride without-Memory (SVSS).

Figures 7 and 8 show prediction accuracy distribution for dynamic load instructions using the static stride without-memory and static stride with-memory predictors. The vertical axis in this chart shows the percentage of dynamic loads that have prediction accuracy greater than or equal to the prediction accuracy threshold of the horizontal axis. The static stride values were determined using the training inputs and the prediction accuracy is found by executing with test inputs. The results of hardware-software VSS in [16], show that a prediction accuracy of about 70% is desired for peak speedup performance. Based on these results, selecting load operations as candidates for

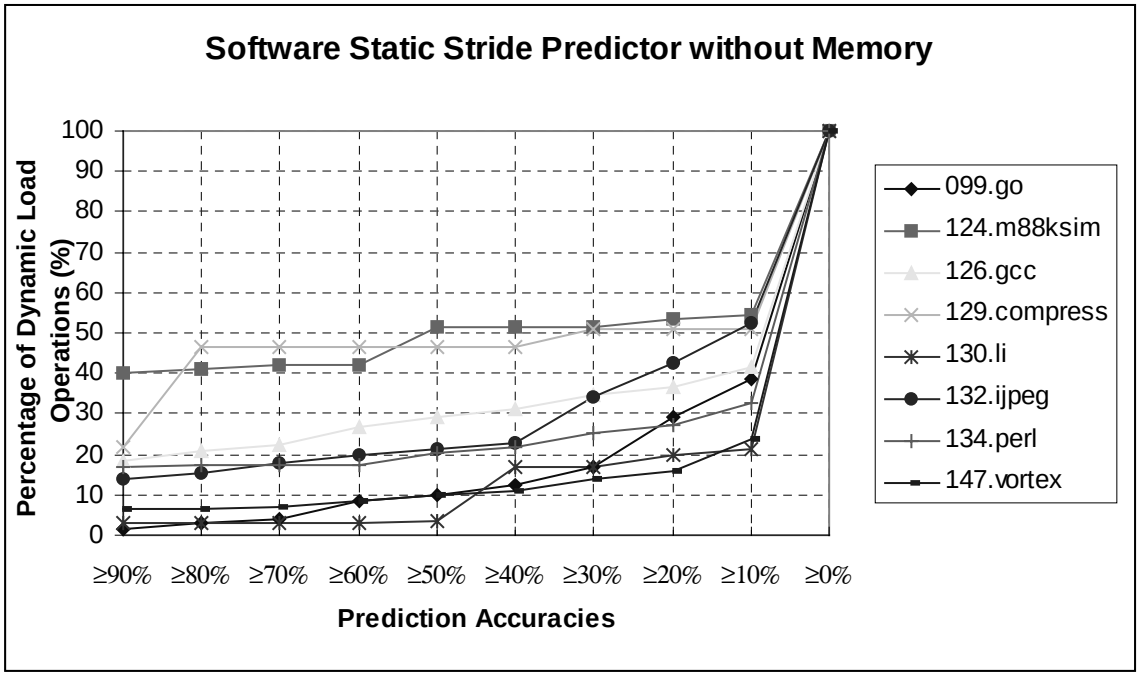

Figure 7. . Prediction Accuracy Distribution for Dynamic Load Instructions using Software Static Stride Predictor without-Memory (SVSS)

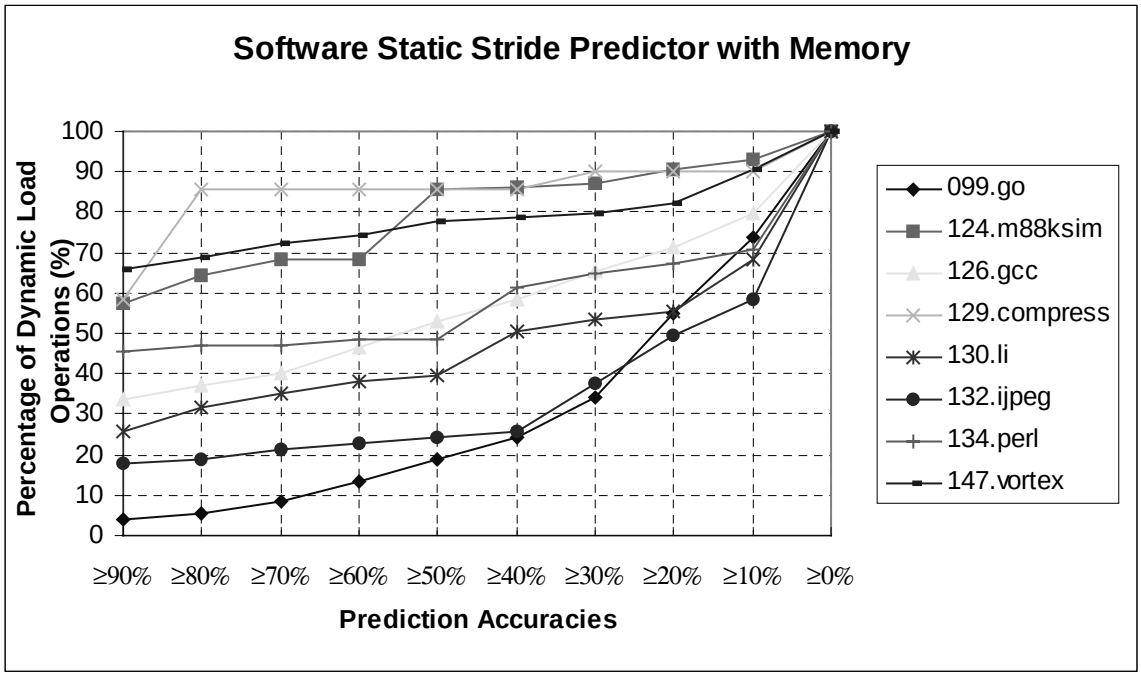

Figure 8. Prediction Accuracy Distribution for Dynamic Load Instructions using Software Static Stride Predictor with-Memory (SVSS)

SVSS with a lower than 70% will likely result in to much overhead for executing patch-up code. This will especially be the case in SVSS for static stride with-memory because of the additional requirement of load and store operations upon procedure entrance and exit, respectively.

For static stride without-memory, the prediction accuracy distribution indicates limited opportunities to find SVSS candidates using a prediction accuracy of 70%. The benchmarks 124.m88ksim and 129.compress have the highest percentage of dynamic loads with a prediction accuracy of greater than or equal to 70%, in the range of 42%-47%. For 129.compress the number of dynamic loads drops off dramatically above 80% and absolute number of dynamic loads, from Table 1 is small. The absolute number of dynamic loads for 124.m88ksim is also small relative to the other benchmark programs.

For static stride with-memory, the prediction accuracy distributions for three of the programs are much improved. Based on these results, there may be opportunities for using SVSS for 129.compress, 147.vortex and 124.m88ksim.

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 eight SPECint95 benchmarks are statically scheduled according to the VSS model by the LEGO compiler. 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.

Figure 9 illustrates the speedup achieved by applying SVSS to the SPEC95 CINT benchmarks. Training set inputs were used to select the stride for both static stride with-memory and static stride without-memory.

A speedup of nearly 8% was achieve for 124.m88ksim using the training inputs while no speedup was achieved using the test inputs. This is a surprising result, considering there was little shift in predictability for 124.m88ksim from the results of Figure 6 using both training and test inputs. We suspect that the training and test inputs exercise orthogonal portions of the program in this case so that the test inputs do not benefit from the speculated loads selected through profiling using the training inputs.

Limited speedup of about 4-5% was achieved for 129.compress using the test inputs. In this case, performance with the test inputs was somewhat better than with the training inputs. Based on the prediction accuracy distribution of dynamic loads for static stride without-memory of Figure 7, a little bit of speedup was expected and achieved. The prediction accuracy distribution of Figure 8 indicated that the with-memory version may receive a little bit more speedup, but the additional load store overhead negates the increased predictability.

Negligible speedup was obtained for 147.vortex. This result was disappointing considering the prediction accuracy obtained in Figure 6 for the static stride with-memory version. But, the sharp decline in predictability for the without-memory version indicates

that most of the speculated load are contained in inter-procedural loops. The penalty for the additional loads and stores required for the with-memory version negates the speedup.

Based on the prediction accuracy distribution of Figures 7 and 8, not much speedup is expected for the rest of the benchmarks. Some speedup was obtained for 134.perl when training inputs were used for the without-memory version.

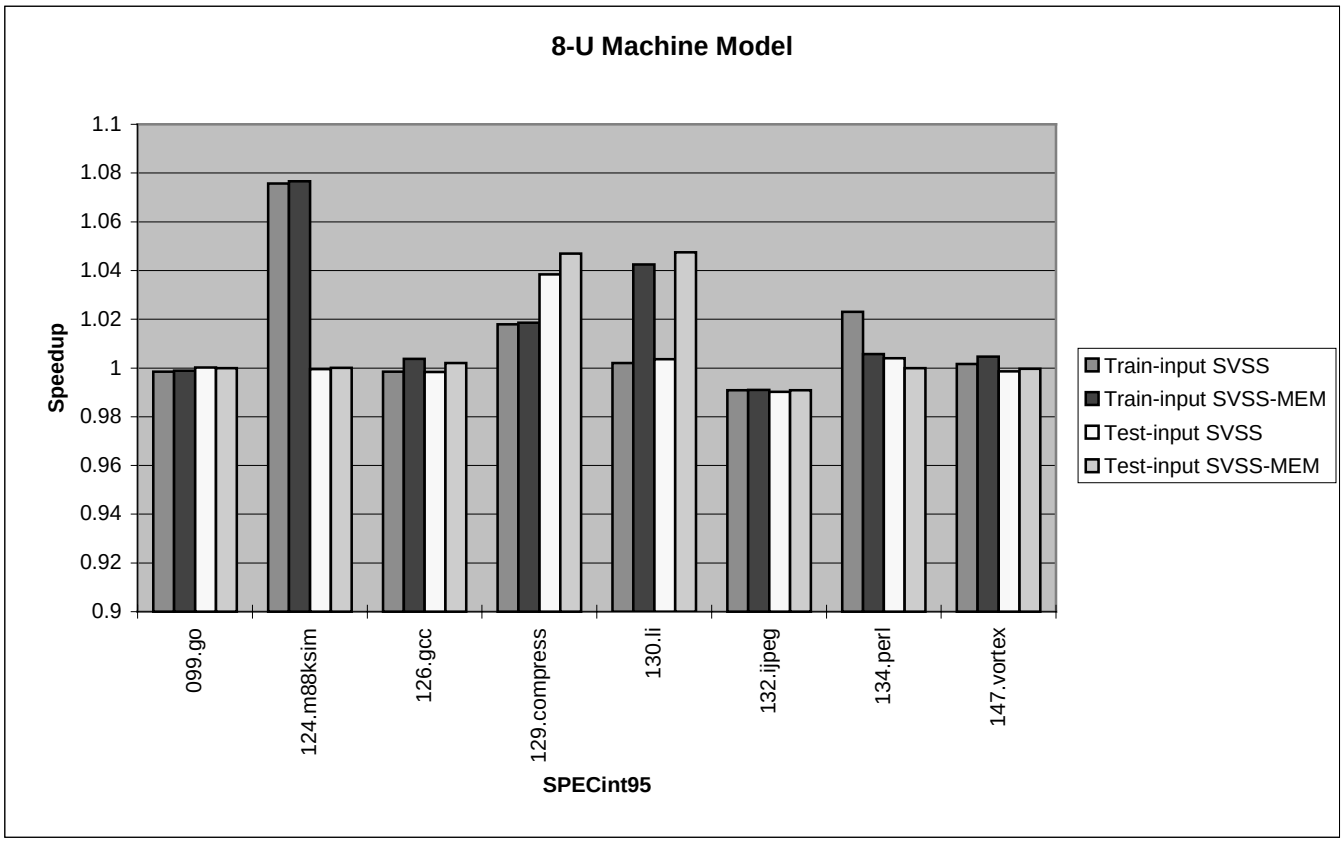

Figure 9. Speedup for Applying SVSS to SPEC95 CINT

Figure 10 shows the number of data values predicted for each procedure in each benchmark. This gives an indication of the number of additional registers needed to support SVSS. For static stride, each data value prediction requires one register. Most benchmarks would use a modest number of registers. 126.gcc uses the most but its performance does not warrant applying SVSS at this time.

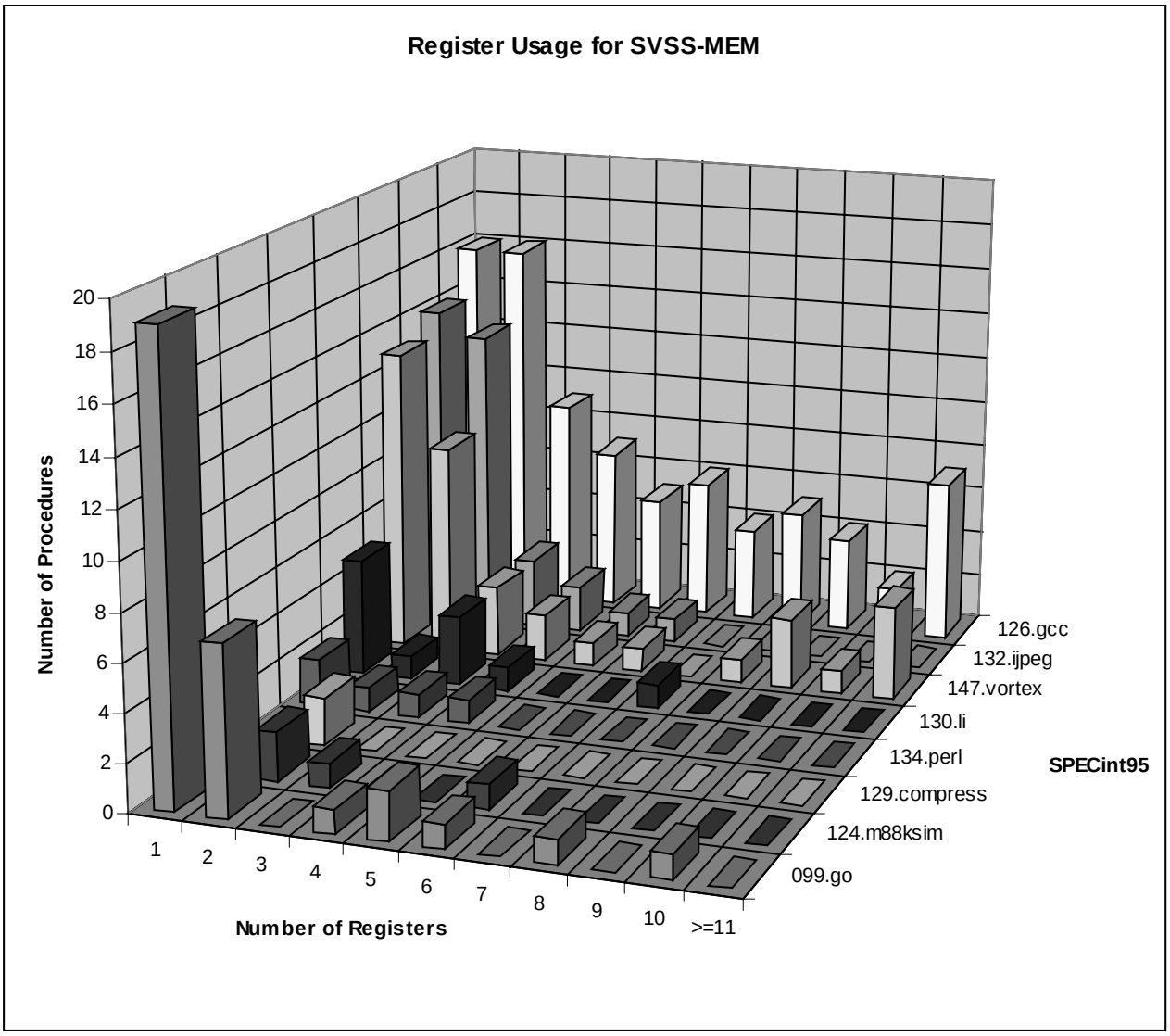

Figure 10. Histograph for Number of Data Values that SVSS was Applied to per Procedure (Number of Load Operations Selected per Procedure).

5. Conclusions and Future Work

Applying SVSS results in speedup of up to 8% for 124.m88ksim and modest speedup for a few of the other benchmarks. Considering the prediction accuracy obtained through value profiling, these results were somewhat disappointing. We still have confidence in the technique of SVSS but belief that better heuristics are required for selecting appropriate data values to apply speculation to. In the work the heuristic

consisted of selecting loads using a predictability threshold of 70%. We see a lot of opportunity for selecting data values in found in intra-procedural loops so that static stride without-memory can be used with the least overhead. We also see many opportunities for instruction combining to hide the overhead of emulating static stride in software. The are several instances of strides of zero that can be predicted without any instrumentation. In general, we need to be more selective in choosing values that are highly predictable, incur little instrumentation overhead, and are good candidates for ILP transformations.

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*, 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*, 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] T. Y. Yeh and Y. N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction,” *Proceedings of the 19th International Symposium on Computer Architecture,* pp. 124-134, May 1992.

[15] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arighmetic Units,” *IBM Journal of Research and Development 11*, pp. 25-33, January 1967.

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