Using Value Prediction to Increase the Power of Speculative Execution Hardware

Freddy Gabbay
Department of Electrical Engineering
Technion - Israel Institute of Technology,
Haifa 32000, Israel.
fredg@psl.technion.ac.il

Avi Mendelson
National Semiconductor Israel
Avi.Mendelson@nsc.com

Abstract

This paper presents an experimental and analytical study of value prediction and its impact on speculative execution in superscalar microprocessors. Value prediction is a new paradigm that suggests predicting outcome values of operations (at run-time) and using these predicted values to trigger the execution of true-data dependent operations speculatively. As a result, stalls to memory locations can be reduced and the amount of instruction-level parallelism can be extended beyond the limits of the program's dataflow graph. This paper examines the characteristics of the value prediction concept from two perspectives: 1. the related phenomena that are reflected in the nature of computer programs, and 2. the significance of these phenomena to boosting instruction-level parallelism of super-scalar microprocessors that support speculative execution. In order to better understand these characteristics, our work combines both analytical and experimental studies.

1. Introduction

The growing density of gates on a silicon die allows modern microprocessors to employ increasing number of execution units. Current microprocessor architectures assume sequential programs as an input and a parallel execution model. Thus, the hardware is expected to extract the parallelism at run time out of the instruction stream without violating the sequential correctness of the execution. The efficiency of such architectures is highly dependent on both the hardware mechanisms and the application characteristic; i.e., the instruction-level parallelism (ILP) the program exhibits.

Instructions within the sequential program cannot always be ready for parallel execution due to several constraints. These are traditionally classified as: *true-data dependencies*, *name dependencies (false dependencies)* and *control dependencies* ([21], [35]). Neither control dependencies nor name dependencies are considered an upper bound on the extractable ILP since they can be handled (or even eliminated in several cases) by various hardware and software techniques ([1], [2], [3], [4], [5], [6], [8], [9], [10], [12], [13], [14], [21], [22], [23], [24], [27], [29], [32], [33], [36], [37], [38], [39]). As opposed to name dependencies and control dependencies, only true-data dependencies are considered to be a fundamental limit on the extractable ILP, since they reflect the serial nature of a program by dictating in which sequence data should be passed between instructions. This kind of extractable parallelism is represented by the *dataflow graph of the program* ([21]).

In this paper we deal with a superscalar processor execution model ([9], [11], [20], [21], [34]). This machine model is divided into two major subsystems: instruction-fetch and instruction-execution, which are separated by a buffer, termed the *instruction window*. The instruction fetch subsystem acts as a producer of instructions. It fetches multiple instructions from a sequential instruction stream, decodes them simultaneously and places them in the program appearance order in the instruction window. The instruction execution subsystem acts as a consumer, since it attempts to execute instructions placed in the instruction window. If this subsystem executed instructions according to their appearance order in the program (*in-order execution*), it would have to stall each time an instruction proves unready to be executed. In order to overcome this limitation and better utilize the machine's available resources, most existing modern superscalar architectures are capable of searching and sending the ready instructions for execution beyond the stalling instruction (*out-of-order execution).* Since the original instruction sequence is not preserved, superscalar processors employ a special mechanism, termed the *reorder buffer* ([21]). The reorder buffer forces the completion of the execution of instructions to become visible (retire) in-order. Therefore, although instructions may complete their execution, they cannot completely retire since they are forced to wait in the reorder buffer until all previous instructions complete correctly. This

capability is essential in order to keep the correct order of exceptions and interrupts and to deal with control dependencies as well. In order to tolerate the effect of control dependencies, many superscalar processors also support *speculative execution*. This technique involves the introduction of a branch prediction mechanism ([33], [37], [38], [39]) and a means for allowing the processor to continue executing control dependent instructions before resolving the branch outcome. Execution of such control dependent instructions is termed "speculative execution". In order to maintain the correctness of the execution, a speculatively-executed instruction can only retire if the prediction it relies upon was proven to be correct; otherwise, it is discarded.

In this paper we study the concept of *value prediction* introduced in [25], [26], [16], [17] and [18]. The new approach attempts to eliminate true-data dependencies by predicting at run-time the outcome values of instructions, and executing the true-data dependent instructions based on that prediction. Moreover, it has been shown that the bound of true-data dependencies can be exceeded without violating sequential program correctness. This claim overcomes two commonly recognized fundamental principles: 1. the ILP of a sequential program is limited by its dataflow graph representation, and 2. in order to guarantee the correctness of the program, true-data dependent instructions cannot be executed in parallel.

The integration of value prediction in superscalar processors introduces a new kind of speculative execution. In this case the execution of instructions becomes *speculative* when it is not assured that these instructions were fed with the correct input values. Note that since present superscalar processors already employ a certain form of speculative execution, from the hardware perspective value prediction is a feasible and natural extension of the current technology. Moreover, even though both value prediction and branch prediction use a similar technology, there are fundamental differences between the goals of these mechanisms. While branch prediction aims at increasing the number of candidate instructions for execution by executing control-dependent instructions (since the amount of available parallelism within a basic block is relatively small [21]), value prediction aims at allowing the processor to execute operations beyond the limit of true-data dependencies (given by the dataflow graph).

Related work:

The pioneer studies that introduced the concept of value prediction were made by Lipasti *et al.* and Gabbay *et al.* ([25], [26], [16], [17] and [18]). Lipasti *et al.* first introduced the notion of "value locality" - the likelihood of a previously-seen value to repeat itself within a storage location. They found that load instructions tend to exhibit value locality and they suggested exploiting this property in order to reduce memory latency and increase memory bandwidth. They proposed a special mechanism, termed "Load Value Prediction Unit" (LVP), which attempts to predict the values that were about to be loaded from memory. The LVP was suggested for current processors models (PowerPC 620 and ALPHA AXP 21164) where its relative performance gain was also examined. In their further work ([26]), they extended the notion of value locality and showed that this property may appear not only in load instructions but also in other types of instructions such as arithmetic instructions. As a result of this observation, they suggested value prediction also in order to collapse true-data dependencies.

Gabbay *et al.* ([16], [17]) have simultaneously and independently studied the value prediction paradigm. Their approach was different in the sense that they focused on exploring phenomena related to value prediction and their significance for future processor architecture. In addition they examined the different effects of value prediction on an abstract machine model. This was useful since it allowed examination of the pure potential of this phenomenon independent of the limitations of individual machines. In this paper and our previous studies we initially review substantial evidence confirming that computed values during the execution of a program are likely to be correctly predicted at run-time. In addition, we extend the notion of *value locality* and show that programs may exhibit different patterns of predictable values which can be exploited by various value predictors. Our focus on the statistical patterns of value prediction also provides us with better understanding of the possible exploitation of value prediction and its mechanisms.

This paper extends our previous studies, broadly examines the potential of these new concepts and presents an analytical model that predicts the expected increase in the ILP as a result of using value prediction. In addition, the pioneer study presented in this paper aims at opening new opportunities for future studies which would examine the related microarchitectural considerations and solutions such as [18]. The rest of this paper is organized as follows: Section 2 introduces the concept of value prediction and various value predictors and shows how value prediction can exceed current ultimate limits. Section 3 broadly studies and analyzes the characteristics of value prediction. Section 4 presents an analytical model and experimental measurements of the extractable ILP. We conclude this work in Section 5.

2. Value prediction and its potential to overcome dataflow graph limits

In this section we provide formal definitions related to the paradigm of value prediction, describe different

techniques to take advantage of this capability and demonstrate the concept through a simple, but detailed example.

2. 1. Principles and formal definitions

Almost any computer operation can be regarded as a transformation between the input it consumes and the destination value it computes. We will refer to the process of computing a destination value as the *generation of a value* and to the computed destination value as *a generated value*. For instance, the generation of a value can be explicitly computed in operations such as arithmetic operations, load or store operations, or can be implicitly computed as a result of changing status bits or other side effects. The flow of these data values between instructions in a program is represented by the dataflow graph.

The dataflow graph representation of a program is a directed graph in which the nodes represent the computations and the arcs represent true-data dependencies:

Definition 1: ***True-data dependency*** ([21]) - If an instruction uses a value generated by a previous instruction, the second instruction has a true-data dependency on the first instruction and the first instruction is also termed a true-data dependent instruction. True-data dependencies represents the flow of information from the instructions that generate it to the instructions that consume it.

Usually, the dataflow graph assumes an infinite number of resources in terms of execution units, registers etc. (in order to avoid the need to refer to structural conflicts and name dependencies) and the control dependencies are either represented by a special type of a node, or are assumed to be known in advance (and so they do not have to be considered in the graph). As a result, this representation of true-data dependencies was recognized and considered as the fundamental limit on the ILP that can ever be gained by current processors ([21]).

Value prediction aims at predicting generated values (before their corresponding operations are executed) and allowing their data dependent instructions to be executed on the basis of that prediction. As a result true-data dependent operations can be executed (speculatively) in parallel. In this paper we assume that the prediction of generated values is made in hardware at run-time, by a special mechanism termed *value predictor*. A description of the various value predictors is introduced in Subsection 2.3.

Like branch prediction, value prediction also causes the execution of instructions to become speculative. However, this time the execution of an instruction becomes speculative because it consumes values (generated by other instructions in the program) that have been predicted in advance and it is not guaranteed that these are the correct values. Thus, we term this way of execution *speculative execution based on value prediction*:

Definition 2: ***Speculative execution based on value prediction*** - is an execution of a true-data dependent instruction where: 1. not all its input values have been computed yet and 2. all the unavailable input values are supplied by the value predictor.

Note that unlike speculative execution based on branch prediction, which seeks to tolerate the effect of control dependencies and schedule instructions in the manner they are presented by the program's dataflow graph, speculative execution based on value prediction attempts to exceed the dataflow graph limits. From the hardware perspective the two kinds of speculative execution resemble each one-another, since they use similar mechanisms: prediction schemes to generate the predicted values, scheduling mechanisms capable of taking advantage of the prediction and tagging instructions that were executed speculatively, a validation mechanism to verify the correctness of the prediction and a recovery mechanism to allow the machine recover from incorrect predictions. Because of the availability of similar mechanisms in current superscalar processors, value prediction is considered a feasible concept.

The potential of using value prediction significantly depends on the *value prediction accuracy* that it can accomplish.

Definition 3: ***Value prediction accuracy*** - is the number of successful value predictions out of the overall number of prediction attempts gained by the value predictor.

Two different factors determine the value prediction accuracy: (1) the value predictor and its capabilities (in terms of resources etc.), and (2) the nature of *value predictability* that resides within the data in the program code.

Definition 4: ***Value predictability*** - is the potential that resides in a program to successfully predict the outcome values generated during its execution (out of the entire range of values that the program generates). This parameter depends on the inherent properties of the program itself, its data, its computation algorithm and the capabilities of the value predictor to reveal and exploit these properties.

In this paper, we suggest distinguishing between two different behavioral patterns of value predictability: *last-value predictability* and *stride value predictability*:

Definition 5: ***Last-value predictability*** - is a measure of the likelihood that an instruction generates an outcome value equal to its most recently generated *value*.

Definition 6: ***Stride value predictability*** - is a measure of the likelihood that an instruction generates an outcome value equal to its most recently generated value plus a fixed delta (stride), where the delta value is determined by the difference between the two most recently generated values.

In Section 3 we provide substantial evidence of the existence of these distinctive patterns and show that categorizing value predictability into these two patterns has a major significance.

2. 2. Simple example

Since the concept of value prediction is quite new, we now introduce a simple and detailed example in order to illustrate its potential. Figure 2.1 exhibits a simple C program segment that sums the values of two vectors (B and C) into vector A.

```
for(x=0;x<100000;x++) A[x]=B[x]+C[x];
```

Figure 2.1 - A sample C program segment.

Compiling this program with a C compiler which employs simple compiler optimizations*, yields the following assembly code (for a Sun-Sparc machine):

```
(1) 22f0:ld    [%i4+%g0],%l7      //Load B[i]
(2) 22f4:ld    [%i5+%g0],%i0      //Load C[j]
(3) 22f8:add   %i5,0x4,%i5        //Incr. index j
(4) 22fc:add   %l7,%i0,%l7        //A[k]=B[i]+C[j]
(5) 2300: st    %l7,[%i3+%g0]     //Store A[k]
(6) 2304: cmp  %i5,%i2            //Compare index j
(7) 2308: add  %i4,0x4,%i4        //Incr. index i
(8) 230c: bcs   0xfffffff9 <22f0> //Branch
(9) 2310: add   %i3,0x4,%i3       //Incr. index k
                                  (in branch delay slot)
```

Figure 2.2 illustrates the data flow representation of this program (each node is tagged with the corresponding instruction number), assuming that all the control and name dependencies were resolved.

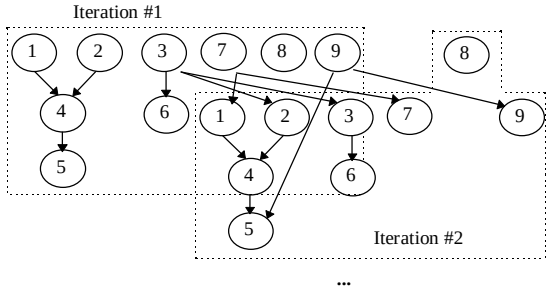


Figure 2.2 - The dataflow graph of the sample program.

In order to reach the degree of parallelism illustrated by figure 2.2, the machine should satisfy the following conditions:

1. It should know the control flow paths of the program in advance in order to eliminate control dependencies. The machine can seek to satisfy this condition by employing branch prediction.
2. Its resources in term of execution units, register file ports, memory ports etc., should satisfy the needs of the program in order to eliminate structural conflicts.
3. The number of registers should be sufficient to satisfy all name dependencies.
4. Its *instruction window* size should be big enough to evaluate all the instructions that appear in the dataflow graph.
5. The machine's fetch, decode, issue and execution bandwidth should be sufficient ([18]). This capability is particularly crucial for the fetch bandwidth, since the sequence in which instructions are stored in memory may not necessarily correspond to the execution sequence that is illustrated by the dataflow graph.

A machine that satisfies all these requirements is considered an *ideal machine*, since the only limitation that prevents it from extracting an unlimited amount of parallelism is the flow of data among instructions in the program (true-data dependencies). Note that current realistic processor architectures can only seek to approach these dataflow graph boundaries, and therefore, they are more properly termed *restricted dataflow machines* ([30]).

When examining our sample program, it can be observed that even if both control and name dependencies were eliminated, the index manipulation still prevents different iterations of the loop from being executed in parallel. Such a phenomenon is termed *loop-carried dependencies*. Various compilation techniques such as *loop-unrolling* ([21], [36]) have been proposed to alleviate this problem. However, these techniques cannot remove the true-data dependencies inside the basic block of the loop.

* The compilation was made with the '-O2' optimization flag.

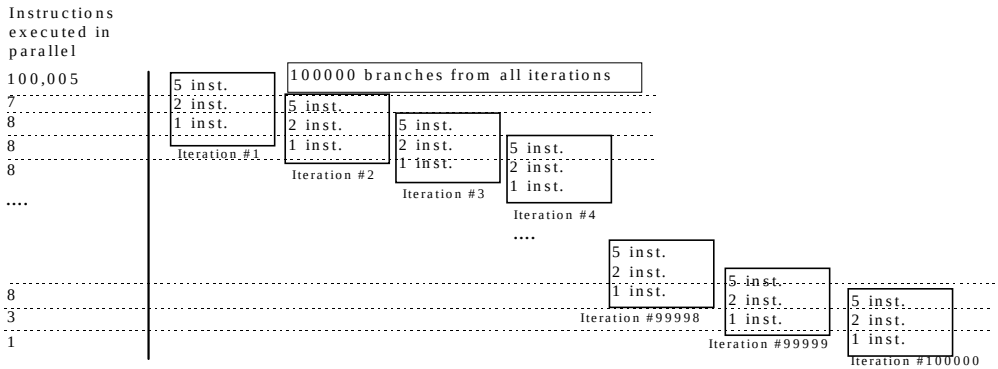


Figure 2.3 - The dataflow execution of the sample loop.

In addition loop-unrolling has several disadvantages such as: 1. significantly enlarging the code size, 2. it is sometimes incapable of evaluating the number of loop iteration at compile time, 3. increasing the usage of registers. Therefore, we decide to examine the existing code, as is generated by a standard gcc compiler (with the simple optimization) and to leave the analysis of the impact of other compilation techniques to future studies.

Figure 2.3 exhibits the overlap between different iterations of the loop during the execution of the program. From this figure we can calculate the ILP when the program runs on an ideal machine. It can be seen that the program is executed in 100,002 clock cycles if we assume that the execution of each instruction takes a single clock cycle. Hence, since the instruction count of the entire program is 900000, the ILP presented by the dataflow graph is 900000/100002 ≅ 9 instructions/clock cycle.

Value prediction, in this example can help predict the values of the index, and if the initial values of the arrays are predictable (for example if both arrays are 0) then it can predict the outcome of the load instructions and the computations (add instruction) as well. In order to illustrate the potential of value prediction and how it works, we perform various experiments and present the ILP in the following cases:

1. No value prediction is allowed.
2. Value prediction is allowed only for loads.
3. Value prediction is allowed only for ALU (arithmetic logic) instructions.
4. Value prediction is allowed for both ALU and load instructions.

In each experiment the ILP is measured for two different sizes of instruction windows (the term instruction window also represents the fetch, decode and execution bandwidth of our machine): 40 and 200. The summary of these simulation results is presented in tables 2.1 and 2.2. The first set of results refers to the case where all the data in the different arrays was initialized to the same value, say arrays of zeroes, before the execution, so that all the loaded values from memory can be predicted correctly at run-time (yet it does not matter how). The second table refers to the situation where the data in arrays was initialized (before the execution of the code segment) with random values, so that no predictable pattern can be observed for its loaded values (the way this effect was simulated in our experiments is also presented during this subsection).

The first set of experimental results indicates that when value prediction is not used, both instruction windows gain the expected ILP of approximately 9. This result is also equal to the ILP that was previously calculated using the dataflow graph in figures 2.2 and 2.3. The enlargement of the instruction window size does not improve the ILP, since loop-level parallelism cannot be exploited due to the loop-carried dependencies. Allowing value prediction for both ALU and load instructions, resolves all true-data dependencies since the index calculations as well as the add operations (that add different components from the arrays) and the load instructions can always be correctly predicted. Therefore the ILP in this set of experiments (table 2.1) is only limited by the instruction window size, nearly 40 and 200 respectively. Value prediction thus yields a 4 to 20-fold speedup! Additional results included in the first set of experiments indicate that value prediction of ALU instructions is more significant for the sample program, than value prediction of load instructions. When value prediction is allowed only for loads, no boost in the ILP is observed. This observation is indeed accurate since the elimination of true-data dependencies that are associated with the load instructions does not allow us to exploit loop-level parallelism across multiple iterations. The pair of load instructions in every basic block (iteration) limit the available parallelism within the basic block itself, however as long as the loop-level parallelism cannot be exploited, no further ILP can be gained. When value prediction is allowed only for ALU instructions it can still gain a significant boost in the ILP relative to the case when value prediction is allowed for both load and ALU

instructions. The explanation of this observation is that loop-level parallelism can be exploited due to the value prediction of the indexes that are computed by the ALU instructions. Only when loop-level parallelism is exploited does value prediction of loads provide an additional contribution to boost ILP as illustrated in table 2.1.

	Instruction window = 40	**Instruction window = 200**
No value prediction	ILP=9	ILP=9
Load value prediction	ILP=9	ILP=9
ALU value prediction	ILP=36	ILP=180
Load and ALU value prediction	ILP=40	ILP=200

Table 2.1 - The ILP when the arrays were initialized with 0's.

In order to further investigate the impact of the predictability of the data in the arrays on the overall performance of this sample code, we repeat the experiments, but this time prevent our value predictor predicting the outcome values of the data being read from the arrays. This means that neither the outcome values of the load instructions nor the outcome values of the add instruction (which adds the array components) are predictable. By such an experiment we can quantify the effectiveness of value prediction (in our sample program) when the data in the arrays is initialized to random values in such a way that they cannot be predicted by our value predictor. Note that in the previous case where the arrays were initialized with zero values, even if value prediction of load instructions was not allowed, the add instruction always generated zero values and so it could be predicted correctly. However, in this case the results of the add instruction can no longer be predicted correctly, since it adds two random values. Therefore this case eliminates the capability to predict the load as well as the add instructions. The results of this set of experiments are summarized in table 2.2.

	Instruction window=40	**Instruction window=200**
No value prediction	ILP=9	ILP=9
Load value prediction	ILP=9	ILP=9
ALU value prediction	ILP=30	ILP=150
Load and ALU value prediction	ILP=30	ILP=150

Table 2.2 - The ILP when the arrays were initialized with random values.

This table indicates that attempting to predict only the values of load instructions is useless, since the value predictor cannot predict them correctly. When value prediction is allowed only for ALU instructions the ILP becomes nearly 30 when the instruction window size is 40, and 150 when the size is 200. The significant increase in ILP is again obtained due to the loop-level parallelism that can be exploited. Employing both load and ALU value prediction does not gain ILP beyond the ILP gained by ALU value prediction since in both cases neither the loads, nor the add instructions of the array components, can be predicted correctly.

2. 3. Various value predictors

We propose three different hardware-based value predictors: the *last-value* predictor, the *stride* predictor and the *register-file* predictor. All these predictors perform a dynamic and adaptive prediction, since they collect and study history information at run-time, and with this information they determine their value prediction. Each of the three predictors has a different prediction formula. The prediction formula determines the predicted value (i.e., the manner in which a predicted destination value is determined). The hardware implementation and considerations of the value predictor are beyond the scope of this paper and are left for future studies in this area ([18]). In this study our purpose is focused on exploring value prediction phenomena from a general viewpoint, hence we discuss the predictor schemes at a high-level without referring to detailed hardware implementation. In addition, for the sake of generality, the size of the prediction table employed by these schemes is assumed to be unlimited in our experiments. For simplicity, we also assume that the predictors only predict destination values of register operands (even though all these schemes can be generalized and can be applied to memory storage operands and condition codes as well) and that they are updated immediately after the prediction is made.

Last-value predictor: predicts the destination value of an individual instruction, based on the last previously-seen value it has generated. The predictor is organized as a table (e.g., cache table - see figure 2.4), and every entry is uniquely associated with an individual instruction. Each entry contains two fields: *tag* and *last-value*. The tag field holds the address of the instruction or part of it (high-order bits in case of an associative cache table), and the last-value field holds the previously-seen destination value of the corresponding instruction. In order to obtain the predicted destination value of a given instruction, the table is searched by the absolute address of the instruction.

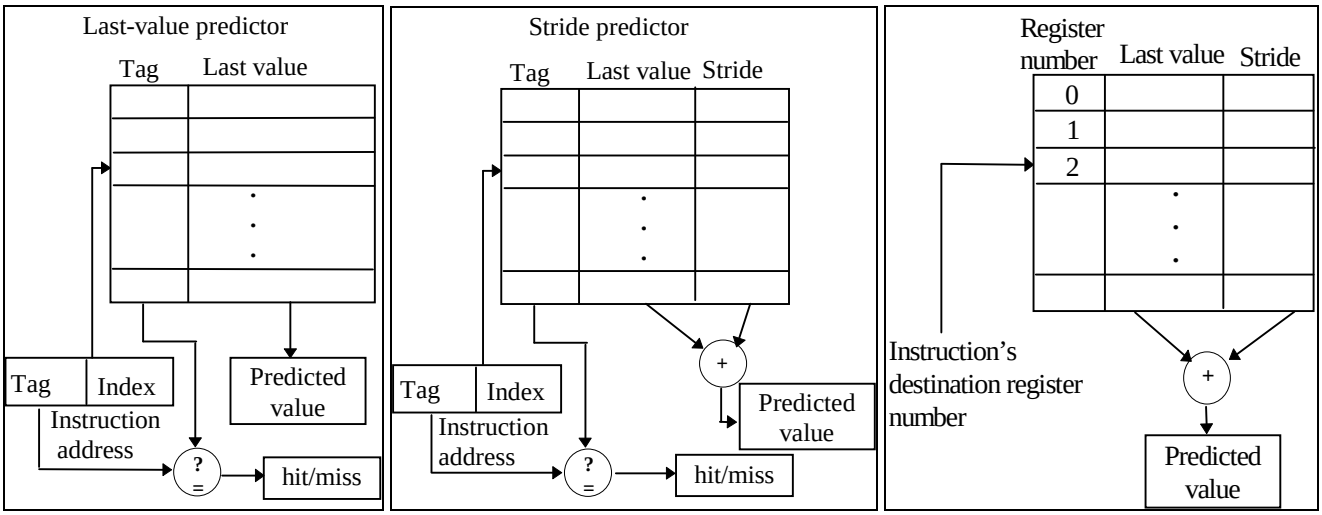


Figure 2.4 - The "last value", the "stride" and the "register-file" predictors.

Performing the table look-up in such a manner can be very efficient, since it can be done in the early stages of the pipeline (the instruction address is usually known at fetch stage). A version of the last-value predictor (using an indexed but untagged table) was proposed by Lipasti *et al.* ([26]).

Stride predictor: predicts the destination value of an individual instruction based on its last previously-seen value and a calculated stride. The predicted value is the sum of the last value and the stride. Each entry in this predictor holds an additional field, termed *stride* field, that stores the previously-seen stride of the individual instruction (figure 2.4). The stride field value is the delta between two recent consecutive destination values.

Register-file predictor: predicts the destination value of a given instruction according to the last previously-seen value and the stride of its destination register (the recent value and the stride could possibly have been determined by different instructions). The register-file predictor is organized as a table as well (figure 2.4), where each entry is associated with a *different (architectural) register*. The past information of each register is collected in two fields: a *last-value* field and a *stride* field. The last-value field is determined according to the last-value written to the corresponding register, and the stride value is the delta between two recent consecutive values that were written to the specific register (possibly by different instructions).

The last-value predictor can only take advantage of last-value predictability since the prediction is made upon the last value, while the stride predictor can exploit both the last-value predictability and the stride value predictability that may reside in a program. The register file predictor may seem very attractive since its prediction table is relatively small. However since the predicted values are determined according to the history information of the register, there can be aliasing between different instructions that write to the same register. As a result, it may have a serious influence on the prediction accuracy that it can accomplish.

3. Experimental characterization of value predictability

This section presents results of various experiments that have been made in this research. Substantial evidence is provided to show that programs exhibit remarkable potential for value predictability. In addition a broad study of various aspects and characteristics of this phenomenon are presented.

3. 1. Simulation environment

A special trace driven simulator was developed in order to provide measurements for the experiments that are presented in the following subsections. The simulation environment was fed with the Spec95 benchmarks suite (table 3.1). The benchmarks were traced by the SHADE simulator ([31]) on Sun-Sparc microprocessor. All benchmarks were compiled with the *gcc 2.7.2* compiler with *all available optimizations*. The set of benchmarks that was used consisted of 8 integer benchmarks and 5 floating-point benchmarks. Each integer benchmark was traced for 100 million instructions (our experiments show that using longer traces barely affects our measurements). In addition, two of the integer benchmarks, *gcc* and *perl*, were examined using two different input files in order to evaluate of the effect of the input file on the characteristic of values predictability. The floating point benchmarks (except *mgrid*) consist of two major execution phases: an initialization phase and a computation phase.

SPEC95 Benchmarks		
Benchmarks	Type	Description
go	Integer	Game playing.
m88ksim	Integer	A simulator for the 88100 processor.
gcc1, gcc2	Integer	A C compiler based on GNU C compiler version 2.5.3 compiling 2 different input files.
Compress95	Integer	Data compression program using adaptive Lempel-Ziv coding.
li	Integer	Lisp interpreter.
ijpeg	Integer	JPEG encoder.
perl1, perl2	Integer	Anagram search program with two different input files.
vortex	Integer	A single-user object-oriented database transaction benchmark.
tomcatv	FP	A vectorized mesh generation program.
swim	FP	Shallow water model with 1024 x 1024 grid.
su2cor	FP	Quantum physics computation of elementary particles masses.
hydro2d	FP	Hydrodynamical Navier Stokes equations solver to compute galactical jets.
mgrid	FP	Multi-grid solver in computing a three dimensional potential field.

Table 3.1 - The Spec95 benchmarks.

In the initialization phase the data is read from large input files, while the computation phase performs the actual computation. In the further experimental results we refer to both these phases respectively. The initialization phase was traced till it was completed (the instruction count is in the order of hundreds of millions of instructions) and the computation phase was traced for 100 million instructions.

3. 2. Value prediction accuracy

The potential of value prediction may depend to a significant degree upon the prediction accuracy of the value predictor. There are two different fundamental factors which determine the value prediction accuracy: (1) the value predictor scheme itself and (2) the potential for value predictability that resides within the program code. The first component is related to the structure of the predictor and its capabilities which eventually determine the prediction formula. The second component reflects inherent properties of the program and its data, and also depends on the capabilities of the value predictor to reveal and exploit these properties. Initially, our experiments provide substantial evidence confirming that programs exhibit value predictability. In addition, they focus on the relations between these two factors by examining how efficiently various predictors exploit different value predictability patterns of computer programs (such as last-value predictability and stride value predictability). Note that throughout these experiments our approach is to focus on the related phenomena from as general a perspective as possibly rather than arguing about the hardware implementation and considerations of the predictors. We are convinced that such an approach allows us to better integrate the concept of value prediction into superscalar processors since we first accumulate substantial knowledge about the related phenomenon before making decisions about the hardware consideration.

Our experiments evaluate three value predictors: the *last-value* predictor, the *stride* predictor and the *register-file* predictor that were all described in previous section. Due to our abstract perspective, the prediction table size of both the last-value predictor and the stride predictor is considered to be unlimited in the experiments. The programs that our simulations examine include both integer and floating-point Spec95 benchmarks. In the integer benchmarks the prediction accuracy is measured separately for two sets of instructions; load instructions and ALU (arithmetic-logic) instructions. In the floating-point benchmarks, the prediction accuracy is measured for two *additional* sets: Floating-point load instructions and floating-point computation instructions.

The first set of measurements consists of the value prediction accuracy of each of the value predictors for integer load instructions in the integer benchmark, as illustrated by table 3.2. This table illustrates remarkable results - nearly 50% (on average) of the values that are generated by load instructions can be correctly predicted by two of the proposed predictors, the last-value predictor and the stride predictor. It can be observed that the value prediction accuracy of the stride predictor and the last-value predictor is quite similar in all the benchmarks, indicating that these integer load instructions barely exhibit stride value predictability. This implies that for this type of instruction in integer programs the cost of an extra stride field in the prediction table of the stride-predictor is not attractive. Moreover, the prediction accuracy of the integer loads does not spread uniformly among the integer benchmarks.

	Prediction accuracy of integer loads [%]			Prediction accuracy of ALU instructions [%]		
benchmark	Stride	Last-value	Register file	Stride	Last-value	Register file
go	29.00	36.23	3.08	62.13	61.28	6.84
m88ksim	75.95	75.93	11.60	95.86	75.88	32.04
gcc1	47.24	52.06	7.48	60.10	55.99	13.80
gcc2	46.36	51.30	6.55	61.06	54.19	16.90
compress	9.84	11.87	0.66	35.94	39.49	4.21
li	48.92	48.58	5.95	63.11	55.19	14.96
ijpeg	31.82	36.37	12.54	35.25	25.70	19.90
perl1	58.54	62.67	10.38	57.23	57.34	8.60
perl2	57.52	53.29	5.26	57.39	50.18	13.29
vortex	71.41	73.94	8.02	83.17	52.47	38.96
average	47.66	50.22	7.15	61.12	52.77	16.95

Table 3.2 - Value prediction accuracy of integer load and ALU instructions in Spec-Int95.

It is apparent that in some benchmarks the loaded values are relatively highly predictable, like the benchmarks *m88ksim* and *vortex*, where the prediction accuracy of both last-value and stride predictors is relatively high (more than 70%), while the prediction accuracy of other benchmarks, like the *compress* benchmark, is relatively low (about 10%). In all the benchmarks, the register-file predictor yields a relatively poor prediction accuracy (less than 10% on average) since it can hardly exploit in this case any kind of value predictability.

Table 3.2 presents additional impressive results about the prediction accuracy which the value predictors gain for ALU instructions in the integer benchmarks. These experiments provide additional encouraging evidence about our capability to predict outcome values. They indicate that a very significant portion of the values generated by ALU instructions are likely to be predicted correctly by our value predictors. In the average case, the stride predictor gains a prediction accuracy of 61% compared to the last-value predictor which gains only 52%. In several benchmarks, like *go* and *perl*, the last-value predictor and the stride predictor gain similar value prediction accuracy. Beyond the last-value predictability that these programs exhibit, they do not exhibit stride value predictability, and therefore both predictors gain similar value prediction accuracy. In these cases, it is expected that most of the correct value predictions of the stride predictor are accomplished by stride values that are actually zero (this expectation would be verified in later subsections). In some benchmarks, like *m88ksim* and *vortex*, although the load instructions exhibit only last-value predictability, their ALU instructions exhibit a significant amount of both last-value predictability and stride value predictability. This observation is expressed in the significant gap between the value prediction accuracy of the stride predictor and the last-value predictor. Hence, in those benchmarks which also exhibit stride value predictability it is expected that the contribution of non-zero strides to the correct value predictions in the stride predictor will be more significant compared to the previous benchmarks (this expectation would be verified as well). As in the previous case, the register-file predictor yields relatively poor prediction accuracy compared to the other predictors. The range of its prediction accuracy varies from 4.2% in the benchmark compress to 38.96% in vortex, yielding an average value prediction accuracy of nearly 17%.

An additional preliminary observation is that different input files do not dramatically affect the prediction accuracy of programs as illustrated for the benchmarks *gcc* (*gcc1* and *gcc2*) and *perl* (*perl1* and *perl2*). This property has tremendous significance when considering the involvement of the compiler in order to support value prediction. The compiler can be assisted by program profiling in order to detect instructions in the program which tend to be value predictable. This observation may indicate that the information collected by the profiler can be significantly correlated to the true situation where the application runs its real input files. An extensive study of the use of program profiling to support value prediction is presented in [17].

The next set of experiments presents the value prediction accuracy in *floating point benchmarks*. The prediction accuracy is measured in each benchmark for two execution phases[*]: initialization (denoted by #1) and computation (denoted by #2). The prediction accuracy measured for integer instructions (load an ALU) is summarized in table 3.3 and for the floating point instructions in table 3.4. It can be observed (table 3.3) that the behavior of integer load instructions in floating point benchmarks is different from their behavior in integer benchmark (table 3.2). Table 3.3 reveals that, unlike the

[*] except *mgrid* where the initialization phase is negligible.

corresponding case in the integer benchmarks where loads exhibited last-value predictability, in this case these instruction also exhibit stride value predictability. These stride patterns are exploited by the stride predictor which achieves average accuracy of nearly 70% in the initialization phase and 63% in the computation phase, in comparison to the last-value predictor which achieves average accuracy of nearly 66% in the first phase and only 37% in the second. The causes for the significant prediction accuracy gap between these predictors in the computation phase are presented later in this paper. In addition, we also notice that as in the previous cases, the register-file predictor achieves a relatively poor value prediction accuracy of only 2-4%.

When the prediction accuracy is measured for ALU instructions in the floating point benchmarks, it reveals several more interesting results as illustrated by table 3.3. In the initialization phase, the three predictors do not exhibit exceptional behavior in comparison to the integer benchmarks. However, in the computation phase of all the floating point benchmarks, the gap between the prediction accuracy of the last-value predictor and prediction accuracy of the stride predictor becomes very significant. In the computation phase most of ALU instructions exhibit stride patterns rather than repeating their recently generated values, and therefore the stride predictor can take advantage of this pattern of value predictability. The stride predictor gains in the computation phase average prediction accuracy of 95%, while the last-value predictor gains only 23%. We discuss in detail the reasons for this observation in a later subsection. In addition, unlike previous cases where the register-file predictor gained relatively poor value prediction accuracy, in this case it gains prediction accuracy of nearly 65% which even outperforms the last-value predictor.

Table 3.4 exhibits the prediction accuracy for two additional sets of floating point instructions: floating point loads and floating point computation instructions. It illustrates that our three value predictors can hardly gain significant prediction accuracy in these instructions, since floating point values show relatively poor tendency of last-value predictability as well as stride value predictability. In floating-point loads the average value prediction accuracy of the last-value predictor and the stride predictor is more than 40%, and in the floating point computation instructions they achieve less than 30% of average prediction accuracy. One of the reasons that may explain why it is harder to predict floating values with these predictors is the representation of these values. Floating point values are represented by three value fields: sign, exponent and fraction (mantissa). It is hard to expect these value predictors, which by their nature tend to fit prediction of integer values, to successfully perform prediction of floating point values. In addition, floating point computations are considerably more complex than integer computations, making them hard to predict. This can also explain why floating point loads exhibit more value predictability in comparison to the floating point computation, since one can expect to find more patterns of regularity and re-use of values in floating point loads rather than in floating point computations.

3. 3. Distribution of value prediction accuracy

The value prediction accuracy that was measured in the previous subsection is an average number that is important in order to evaluate the performance of the predictors. However this average number does not provide information about distribution of the prediction accuracy among the instructions in the program. The following measurements attempt to provide a deeper study of the statistical characteristics of value prediction by examining the distribution of value prediction accuracy, and also discuss how this knowledge can be exploited.

Figure 3.1 illustrates the distribution of value prediction accuracy of the stride predictor among the instructions in the program (referring only to the value-generating instructions). It indicates that the prediction accuracy does not spread uniformly among the instructions in the program. More than 40% of the instructions are very likely to be correctly predicted with prediction accuracy greater than 70%. In addition, approximately the same number of instructions are very unlikely to be correctly predicted. These instructions exhibit less than 40% a prediction accuracy.

These results motivate us to develop mechanisms that would allow us to distinguish between the predictable and unpredictable instructions and avoid the unpredictable ones. Such classification contributes to each of the following aspects:

1. The classification can significantly increase the effective value prediction accuracy of the predictor by eliminating the value prediction of the unlikely predictable instructions. A preliminary study of the effect of such classification on overall performance is discussed in a later section of this paper.
2. The replacement mechanism of the prediction table can exploit this classification and prioritize entry replacement for greater efficiency. Eventually, this has the potential to significantly increase effective utilization of the prediction table.
3. In certain microprocessor architectures mispredicted values may cause some an misprediction penalty due to their pipeline organization. By classifying the instructions, the processor may refrain from predicting values from the class of unpredictable instructions and avoid the misprediction penalty.

	Prediction accuracy of integer loads [%]			Prediction accuracy of ALU instructions [%]		
benchmark	Stride	Last-value	Register file	Stride	Last-value	Register file
tomcatv#1	59.82	53.61	4.95	50.87	46.97	13.88
tomcatv#2	99.22	61.91	7.57	99.54	41.83	19.58
swim#1	80.98	81.56	0.02	88.95	79.43	11.09
swim#2	14.69	19.48	0.00	99.57	0.07	99.83
su2cor#1	71.31	65.78	5.76	60.09	57.68	14.49
su2cor#2	47.50	34.65	0.02	91.41	44.72	41.22
hydro2d#1	69.51	61.18	7.03	58.42	49.93	15.94
hydro2d#2	-	-	-	99.23	15.51	87.67
mgrid	91.88	30.47	1.10	88.20	14.21	69.71
average #1	70.40	65.53	4.44	51.66	46.80	11.08
average #2	63.32	36.62	2.17	95.59	23.26	63.60

Table 3.3 - Value prediction accuracy of integer load and ALU instructions in Spec-FP95.

	Prediction accuracy of FP loads [%]			Prediction accuracy of FP computation inst. [%]		
benchmark	Stride	Last-value	Register file	Stride	Last-value	Register file
tomcatv#2	22.95	6.32	0.16	21.88	15.08	2.13
swim#1	86.21	82.78	0.00	23.15	19.88	1.79
swim#2	18.03	26.09	1.57	15.54	21.38	0.16
su2cor#2	38.87	39.44	21.22	16.36	16.63	9.99
hydro2d#2	88.72	89.63	46.56	89.68	89.89	42.79
mgrid	18.81	18.33	4.71	7.11	6.87	4.04
average	45.59	43.76	12.37	28.95	28.28	10.15

Table 3.4 - Value prediction accuracy of FP load and computation instructions in Spec-FP95.

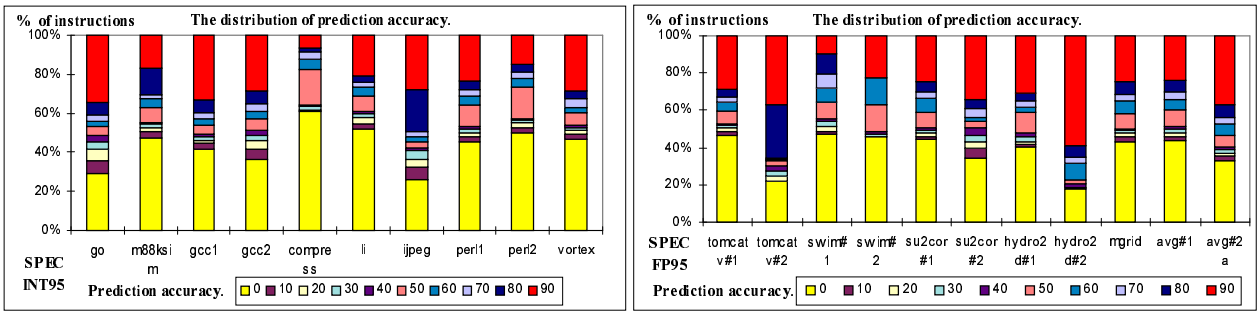
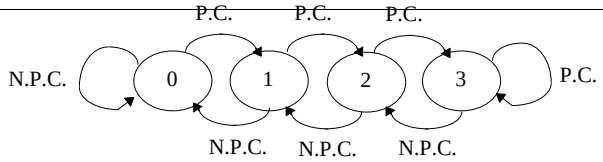


Figure 3.1 - The distribution of instructions according to their value prediction accuracy.

Two methods suggest themselves for classification:

The first method assigns an individual saturated counter (figure 3.2) to each entry in the prediction table. At each occurrence of a successful prediction the counter is incremented, or conversely decremented. By inspection of the saturated counter, the processor can decide whether to consider the suggested prediction or to avoid it. Such a method for value prediction classification was introduced by Lipasti *et al.* in [25].



0 - Strongly **not predicted**.
1 - Weakly **not predicted**.
2 - Weakly **predicted**.
3 - Strongly **predicted**.

P.C. - Predicted correctly.
N.P.C. - Not predicted correctly.

Figure 3.2 - A 2-bit saturated counter for value prediction classification.

Our measurements confirm the results of Lipasti *et al.* where it is shown that such a classification mechanism

can work efficiently to eliminate value misprediction. We observed that in the integer benchmarks almost 94% of the value misprediction can be eliminated and 92% of the correct prediction are correctly classified by this mechanism. In the floating point benchmarks, the numbers are very similar: nearly 95% of the mispredictions are eliminated and almost 95% of the correct prediction are correctly classified. Beyond the efficiency of the saturated counters to classify the value predictability of instructions, we have revealed another interesting phenomenon related to this mechanism in our study. We have measured the ratio of the number state transitions that each of the automates made to the number of accesses to each of the corresponding entries in the value prediction table. These measurements provide significant information concerning the speed with which this classification method converges to the correct classification. These measurements are summarized in figure 3.3, which indicates that most of the saturated counters are locked on the correct classification after a relatively very small number of transitions. This observation also strengthens our previous experiments about the distribution of the prediction accuracy that was illustrated in figure 3.1. For the two sets of instructions, highly predictable and the unlikely predictable, the classification mechanism can be very confident about the outcome of the prediction (whether it succeeds or fails).

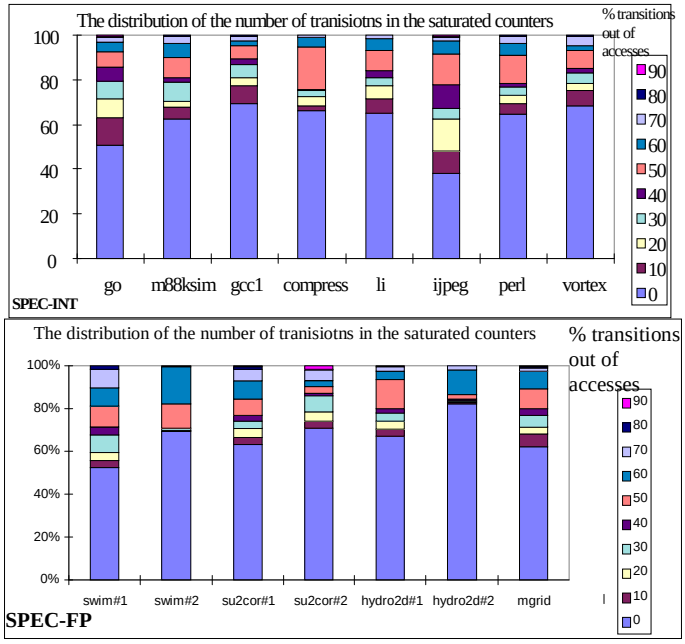


Figure 3.3 - The distribution of the number of transitions in the saturated counters among instructions in programs.

Another method that we consider for classification is the use of compiler profiling. The compiler can collect information about the value predictability of instructions according to previous runs of the program. Then it can place directives in the opcode of the instructions, providing hints to the processor for classifying the instructions. From a previous subsection we recall that the measurements of the expected correlation of value prediction accuracy between different runs of the program with different input files are encouraging. The use of program profiling for value prediction classification is presented in [17].

3. 4. Non-zero strides and immediate operations

The stride predictor extends the capabilities of the last-value predictor since it can exploit both last-value predictability and stride value predictability. In this subsection we examine how efficiently the stride predictor takes advantage of its additional stride fields beyond the last-value predictor to exploit stride value predictability. We consider the additional stride fields to work efficiently only when the predictor accomplishes correct value predictions that are based on non-zero strides. In table 3.5 we present the ratio of successful predictions that were based on non-zero strides to the overall number of correct predictions. In the integer benchmarks this ratio is more than 16%, and in the floating point benchmarks it varies from 12% in the initialization phase to 43% in the computation phase. The relatively high ratio of non-zero strides in the floating point computation phase is explained by the significant contribution of immediate integer add and subtract instructions to the successful predictions.

Ratio of successful non-zero stride-based predictions out of overall successful predictions.			
Spec95 integer		**Spec95 floating point**	
Benchmark	[%]	Benchmark	[%]
go	8.83	tomcatv#1	13.93
m88ksim	16.42	tomcatv#2	55.14
gcc1	12.79	swim#1	8.98
gcc2	15.44	swim#2	65.9
compress95	6.52	su2cor#1	9.35
li	15.22	su2cor#2	27.74
ijpeg	36.37	hydro2d#1	16.28
perl1	7.57	hydro2d#2	15.09
perl2	15.27	mgrid	51.4
vortex	30.34	**average#1**	**12.14**
average	**16.48**	**average#2**	**43.06**

Table 3.5 - The distribution of non-zero strides.

This table, however, may lead the reader to an incorrect conclusion about the effectiveness of the stride predictor in exploiting non-zero strides and its significance to the expected ILP improvement. For instance, it shows that 16.4% out of successful predictions in the benchmark *m88ksim* are because of non-zero strides and 15.2% in the benchmark *li*. Does this mean that the contribution of the stride predictor and non-zero strides to these two benchmarks is the same? Obviously not; one should be aware of the fact that these results should be given the appropriate weight, i.e. their relative number of appearances in the program's execution. Moreover, the connection between the prediction accuracy and the expected boost in ILP is not straightforward ([18]) since the distribution of its contribution may not be uniform. If the importance of non-zero strides is crucial to the ILP of the application, even an improvement of approximately 10% in the prediction accuracy can be very valuable. A broader study of the contribution of strides to the ILP is presented in Section 4. In addition, knowledge about the characteristics of non-zero strides may motivate future explorations of the potential for hybrid predictors, which combine both the last-value predictor and the stride predictor. Here, the last-value predictor would be dedicated for zero strides and the stride predictor would be dedicated for non-zero strides.

We find that non-zero strides appear for various reasons, e.g. immediate add and subtract instructions, and computations of addresses of memory references that step with a fixed stride on arrays in memory. Figure 3.4 illustrates the contribution of immediate add and subtract instructions to the *overall* number of successful predictions in the stride predictor. In the integer benchmarks it is nearly 20% (on average) and in the floating point benchmarks it varies from nearly 15% in the initialization phase to more than 30% in the computation phase (on average). The significant gap between the contributions in the initialization phase and in the computation phase of the floating point benchmarks can be explained by the fact that most memory accesses of floating point benchmarks consist of stride patterns ([28]). When the fraction of the successful value predictions which are used for address calculations of memory pointers (for data items only) is examined, it reveals that this number is considerably more significant in the computation phase than the initialization phase as illustrated by figure 3.5. The next subsection confirms this observation as well.

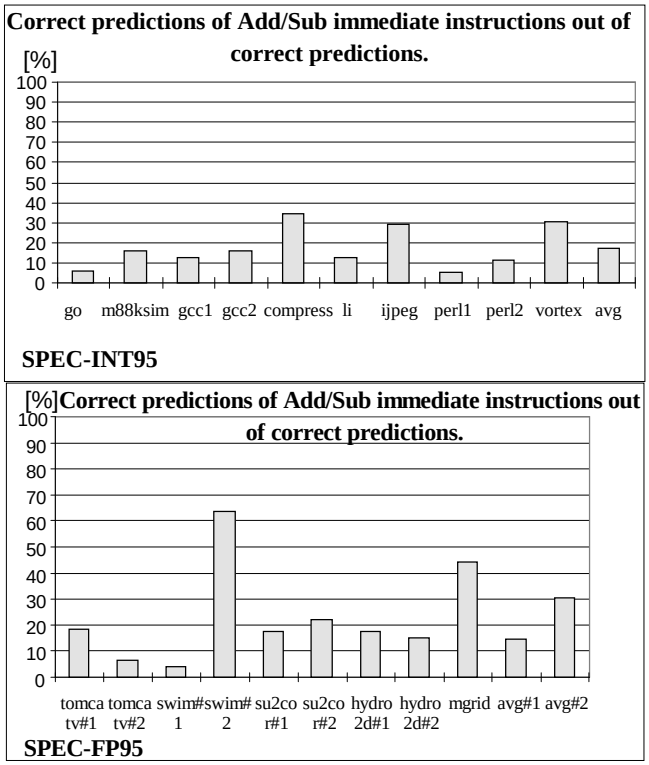


Figure 3.4 - The contribution of Add/Sub immediate instructions to the successful predictions.

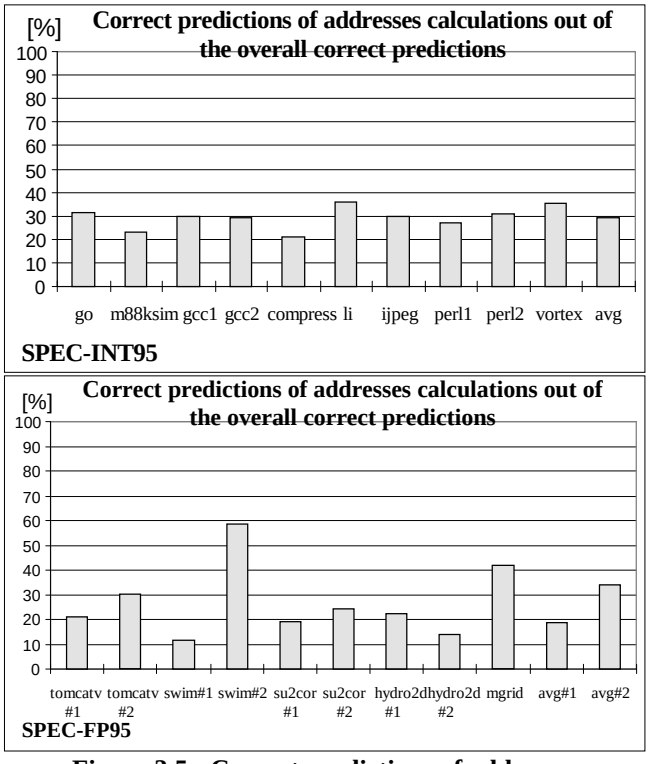


Figure 3.5 - Correct predictions of addresses calculations out of successful predictions.

3. 5. Characteristics of value predictability in load instructions

In this subsection we focus on the characteristics of value prediction with respect to load instructions. Load instructions may have a significant impact on the program performance in two major aspects: 1. they may cause the processor to stall due to memory latencies and 2. they may cause true-data dependencies (since they generate values that can be consumed by other instructions). Our main purpose is to examine how the concept of value prediction can affect and support these two aspects. Up to the present researchers have addressed the first aspect by improving the performance of cache memories or by using prefetching mechanisms.

In order to explore the contribution of value prediction to the first aspect, we examine all the load memory references that are cache misses and their loaded memory values. We only refer to the cache misses since these are the cases when memory system delays occur. First, we are interested in studying what portion of these loaded memory values are value-predictable and what the relations are between them and the potential to predict their *memory address*. The potential to predict memory address (address predictability) is determined by the memory locality patterns of the program. This property was widely exploited by different data prefetching mechanisms ([7], [15], [19], [28]). Examining the value predictability patterns in the load cache misses and comparing them to the address predictability patterns can provide valuable information to the following questions:

1. What is the effectiveness of value prediction in reducing the penalty of load cache misses by attempting to predict their values (as proposed by Lipasti *et al.* in [25])?
2. How successfully can value prediction compete with other techniques such as data prefetching?

The structure of the competitive address predictor scheme that was used in our experiments is similar to our value predictor schemes. It is organized as a table (for instance a cache table) with an unlimited number of entries, where each entry is uniquely assigned to a previously-seen load instruction. Each entry contains three fields: *Tag* - identifies the individual load instruction according to its instruction address, *Last address* - the address of the last cache block that was fetched by the load instruction and an *Address stride* - which is determined according to the difference between two consecutive memory accesses of each of the individual loads. The predicted address which the prefetching scheme fetches is determined according to the last address field plus the stride field. The value predictor chosen to compete the address predictor is the last-value predictor, since it has gained the best prediction accuracy for load instructions. The data cache parameters that were chosen for the experiments are quite typical to common modern microprocessors (PowerPC™, Pentium™, PentiumPro™): 16 KB size, 4-way set associative and 32 bytes line size.

Figure 3.6 illustrates the correlation between the *address prediction accuracy* and value prediction accuracy out of the load misses. This figure exhibits four possible sets of load references: 1. load references where *both the data values and addresses values* can be predicted correctly, 2. load references where *only the addresses values* can be predicted correctly , 3. load references where *only the data values* can be predicted correctly and 4. load references where *neither the data values nor the addresses values* can be predicted correctly. It can be observed that out of the load misses in the integer benchmarks, the misses attributable to the third set (predictable data values only) is comparable to the misses falling into the second set (predictable addresses values only). Therefore, in these benchmarks, value prediction can contribute significantly by handling a substantial set of load references which cannot be handled by data prefetching. However, it can be seen that in the floating point benchmarks, most of the load cache misses which exhibit correctly predicted data values also exhibit correctly predicted addresses values, i.e., the portion of the third set is negligible. Figure 3.7 is similar to figure 3.6 - it illustrates the correlation between the address prediction accuracy and value prediction accuracy when the bars of each benchmark are given the appropriate weight according to the load miss rate.

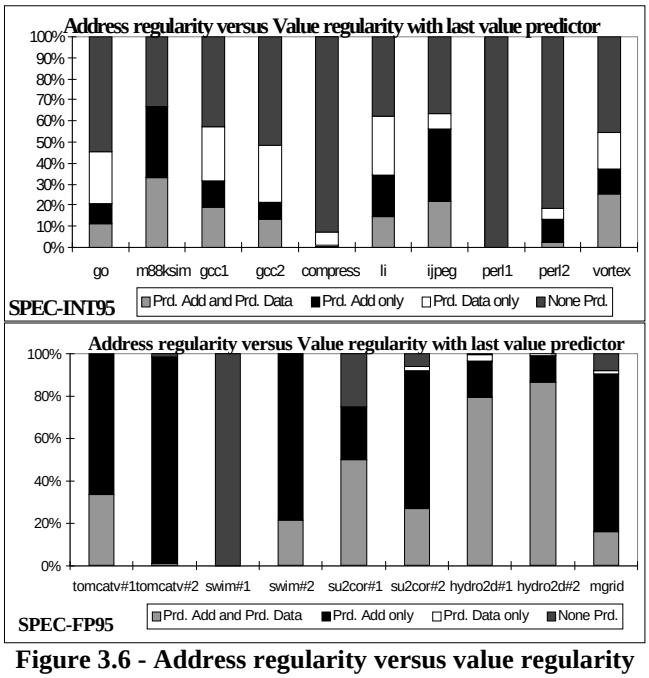


Figure 3.6 - Address regularity versus value regularity out of the overall load misses.

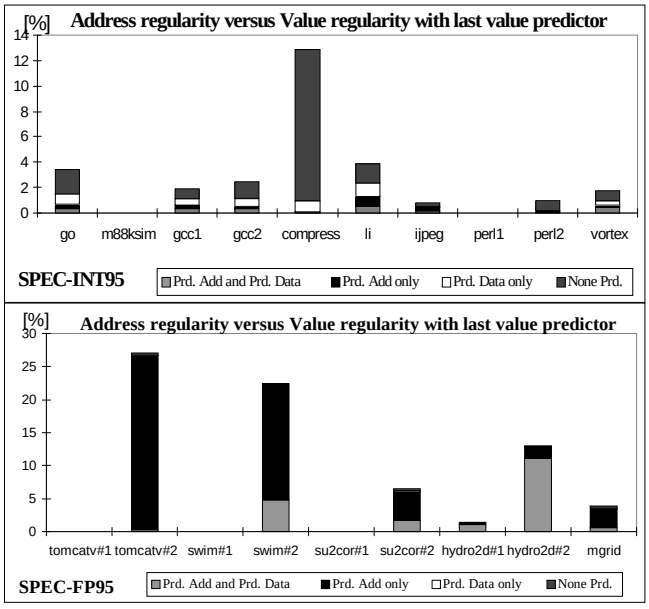


Figure 3.7 - Address regularity versus value regularity weighted with loads miss rate.

In order to address the second aspect, we measure the ILP that can be exploited when using value prediction only for load instructions. Our experiments assume a perfect memory system, i.e., memory references never stall the processor. We choose to eliminate the contribution of memory system latency in this set of experiments since the system can seek satisfying this assumption by using various data prefetching techniques such as [7], [15], [19] and [28]. Note that this assumption even degrades the potential benefits of value prediction since long latency instructions can even better take advantage of value prediction ([25], [26]). Moreover, we prefer focusing on the pure potential of value prediction to collapse true-data dependencies associated with load instructions rather then dealing with impact of memory system latency since such issue is system dependent. In addition, in order to avoid discussing individual implementation issues, an abstract machine is considered with an unlimited number of execution units and physical registers, but with a restricted instruction window size. Each instruction is assumed to take only a single cycle. In addition, it is assumed that all branches are correctly predicted in advance. The ILP that such a machine can exploit is limited by the dataflow graph of the program and by the size of the instruction window. Figure 3.8 shows the ILP that can be gained by employing the last-value predictor in comparison to a machine that does not employ value prediction. It indicates that in some benchmarks like *m88ksim*, *li* and *perl* the contribution of load value prediction is significant while in some other benchmarks like *compress*, *vortex* and *mgrid* it is barely noticeable. These variations are highly dependent on the value predictability patterns that these programs exhibit and their contribution, in particular to the value predictability patterns of the load instructions.

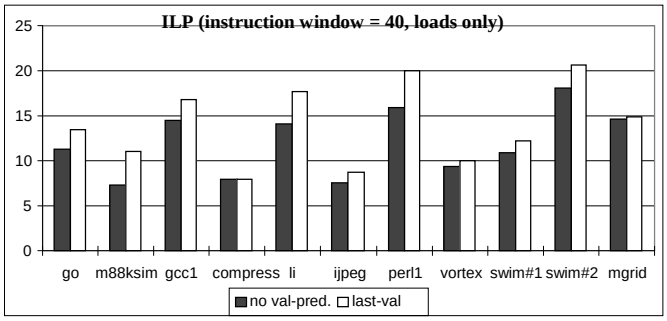


Figure 3.8 - The ILP gained by loads value prediction.

4. Analytical and experimental analysis of the ILP increase

The ILP that a processor can exploit in the case of a serial program is an indicator of the amount of resources that it can efficiently utilize simultaneously. In this section we quantitatively study the ability of value prediction to boost the ILP from two viewpoints: analytical and experimental.

4. 1. An analytical model

So far we have discussed the characteristics and potential of using value prediction. Before presenting actual measurements, which depend on the benchmarks we use, we present a simple analytical model that can be employed for both estimating the potential of value prediction to increasing the ILP and for further understanding the related phenomena.

The dataflow graph (*DFG*) presentation of a program, is given by a directed graph *G(V,S)*. Each node, $v \in V$, represents a single operation, or a set of operations which are executed as a single atom. Each arc, $s \in S$, represents a true-data dependency between two nodes. Given a *DFG* representation, we define the critical path to be the longest (in term of execution time) path, *C*, that connects the entry point to the termination point. The critical path, *C*, forms a linear graph. For simplicity, we assume that each node (operation) of the critical path *C* takes *T* cycles for execution. Since the execution of the operations in *C* cannot overlap, the total execution time of C is $n \times T$ cycles (where *n* is the number of nodes in *C*). If we assume that our machine has an unlimited amount of resources, the execution time of the critical path determines the execution time of the entire program, since the execution of the other paths in the *DFG* can be overlapped with the execution of the critical path *C* with no resources conflict.

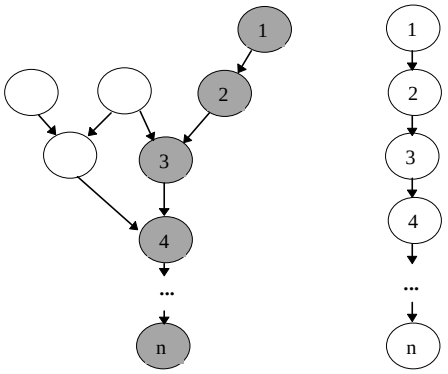


Figure 4.1 - The DFG, *G(V,S)*, and the critical path *C*.

When value prediction is employed, we attempt to correctly predict the outcome of each operation before it is executed. Thus, if we could predict all outcome values correctly, and we had an unlimited number of resources, we could execute any program in two steps, one that executes the instructions and the other that verifies the correctness of the prediction. In reality, we cannot predict all the values correctly, and so for each arc $s \in S$ in the *DFG*, we attach a number p_s $(0 \le p_s \le 1)$ that represents the probability to correctly predict the result of node *s* (note that all the arcs that come from the same node have the same probability). We term this weighted graph the *speculative dataflow graph* (*SDFG*). An example of an *SDFG* that corresponds to the *DFG* in figure 4.1 is illustrated in figure 4.2.

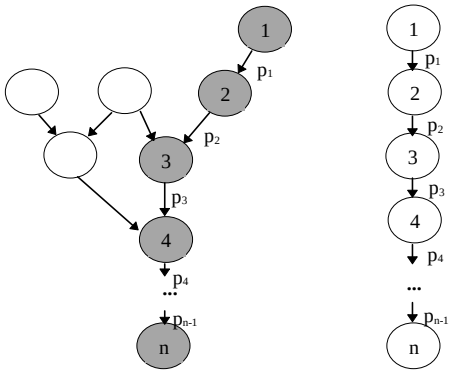


Figure 4.2 - The SDFG, $G_S(v, s)$, and the critical path graph *C*.

From the *SDFG* we can also extract the critical path *C* (that forms a linear graph) and evaluate the execution time when instruction are executed speculatively based on their predicted values. In our analysis we only focus on the execution time of the critical path, since we assume that it dictates the execution time of the entire program and all the execution of the other paths can overlap. We evaluate the critical path execution time in two cases: when the instruction window size is unlimited and when the window size is finite. In both cases our model assumptions are:

1. We consider an abstract machine with an unlimited number of execution units and physical registers.
2. For simplicity we consider the probabilities p_i for $1 \le i < n$ to be statistically independent. We have found this approximation to be valid since instructions in the critical are also data dependent on instructions from other paths in the DFG. Moreover, if the value predictor fails to predict an input value of instruction it does not necessarily imply that it fails to predict its output as well.
3. We assume that $p_i = p$ for $1 \le i < n$, where p is the measured prediction accuracy of the value predictor.
4. True-data dependent instructions are allowed to be executed and committed in parallel as long as their input (possibly predicted) values are found to be correct. This means that instructions in the linear graph are allowed to complete (commit) in parallel until the first failure (value misprediction).
5. An instruction that was fed with an incorrect input value needs to be *re-executed when its correct input value is ready*.

4. 1. 1. Infinite instruction window size model

When the instruction window size is infinite, the processor can simultaneously examine the potential ready-to-execute instructions of the entire program. In order to illustrate the parallel execution of the critical path, *C*, we use the execution graph, $G_E(V_E, S_E)$, which is illustrated by figure 4.3. The execution graph illustrates all the possible execution sequences of the critical path *C*. Each node $v \in V_E$ denotes an execution of an operation. Nodes which are executed based on speculative input value are denoted with the subscript letter '*s*'. Each arc, $s \in S_E$, denotes a possible execution sequence. For each arc, we assign two numbers: 1. Pr_i - the probability to correctly or incorrectly predict the outcome value of the operation in node *i*, and 2. t_i - the cost in terms of clock cycles of the transition in case of a correct or an incorrect value prediction. Notice that the probabilities Pr_i for arcs that comes for speculatively executed instructions can be extracted from the SDFG (figure 4.2). When instructions are re-executed, due to value misprediction, Pr_i is considered to equal *1*. In addition if $t_i = 0$, it implies that the two operations linked by the arc can be executed in parallel. For instance, one of the possible execution sequences is when all the instructions are predicted correctly. The probability of such an event is p^{n-1} and the entire execution time is *T* since all the instructions are executed simultaneously.

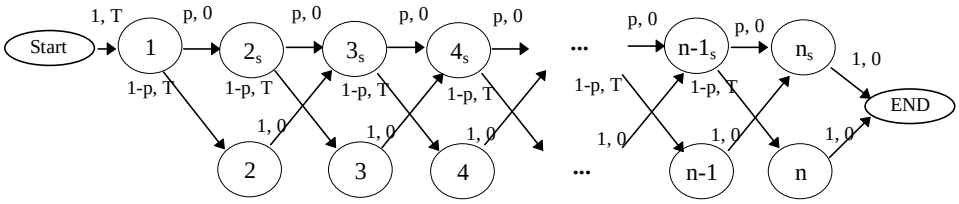


Figure 4.3 - The execution graph of critical path *C*.

In general, the probability to execute a certain path, $\sigma = (s_1, s_2, ..., s_n) \in G_E(S, V)$, is given by $P_\sigma = \prod_{i=1}^{n-1} \Pr_{s_i}$ and the entire execution time of σ, can be obtained by $T_\sigma = T + T_P$, where $T_P = \sum_{i=1}^{n-1} t_{s_i}$. It can be noticed that the element T_P has a binomial distribution, as illustrated by equation 4.1:

$$Prob(T_P = k \cdot T) = (1-p)^k \cdot p^{n-1-k} \cdot \binom{n-1}{k} \quad ,0 \le k \le n-1$$

Equation 4.1 - The binomial distribution of T_P.

One can also observe that "placing" the value misprediction in the linear graph *C* is equivalent to "choosing" *k* out of *n-1* where probability to "choose" is *1-p* (value misprediction). As illustrated by equation 4.2 we can calculate the average of T_σ, i.e., the average execution time of the critical path *C*.

$$E(T_\sigma) = T + T\sum_{i=0}^{n-1} i \cdot (1-p)^i \cdot p^{n-1-i} \cdot \binom{n-1}{i} =$$

$$T + T(n-1) \cdot (1-p)$$

Equation 4.2 - The average execution time of the critical path C.

The average boost in the ILP (or in the execution time) of *C* is given by equation 4.3:

$$ILP\ boost_{avg} = \frac{n \cdot T}{E(T_\sigma)} = \frac{n}{1 + (n-1) \cdot (1-p)} \cong \frac{1}{1-p}$$

Equation 4.3 - The average boost in the ILP of the critical path C.

4. 1. 2. Finite instruction window size model

We can improve our previous model and consider the effect of a finite instruction window size when we use value prediction. For simplicity, we only consider the execution of the critical path *C* and we ignore the effect of the execution of the other paths. In addition, we also assume that instructions which are executed correctly evacuate the instruction window and allow other instructions to enter the instruction window as potential candidates for parallel execution. When value prediction is not used the critical path will be executed in $n \cdot T$ cycles since the execution time of the critical path is bounded because of true-data dependencies and not because of the window size. In steady state, the number of instructions that evacuate the instruction window is equal to the number of new instructions that enter the window. When value prediction is used, the number of instructions that evacuate the instruction window is also equal to the number of instructions that were predicted correctly until the first instruction *in the window* fails (incorrectly predicted). Let *w* be the size of the instruction window ($w>1$) and let *L* be the random variable that denotes the number of instructions which evacuate the instruction window at each step. The distribution of *L* is given by equation 4.4:

$$Prob(L = k) = \begin{cases} (1-p)p^{k-1} & 1 \le k \le w-1 \\ p^{w-1} & k = w \end{cases}$$

Equation 4.4

The average of *L* is denoted by equation 4.5:

$$E(L) = \frac{(1-p^{w-2})}{1-p} - (w-1) \cdot p^{w-2} + w \cdot (1-p) \cdot p^{w-1} \cong$$

$$\frac{1}{1-p} - w \cdot p^{w-2}[1-(1-p) \cdot p]$$

Equation 4.5

Therefore in this model the average ILP boost of value prediction is.

$$average\ boost\ in\ ILP = \frac{E(L)}{1} \cong$$

$$\frac{1}{1-p} - w \cdot p^{w-2}[1-(1-p) \cdot p]$$

Equation 4.6

Note that this result consists of two elements: the first element $\frac{1}{1-p}$ represents the boost in the case of an infinite instruction window and the second element takes into account the effect of the window size.

4. 2. Experimental framework

Our experiments consider an abstract machine with a finite instruction window. We have chosen this experimental model since the concept of value prediction is entirely new and the discussion of particular implementation issues inherent to different processor architectures is beyond of the scope of this paper. The ILP which such a machine can gain (when it does not employ value prediction) is dictated by the dataflow graph of the program and its instruction window size. We have previously indicated that in order to reach the dataflow graph boundaries, a machine should employ: 1. unlimited number of resources (execution units etc.), 2. unlimited number of registers, 3. perfect (either static or dynamic) branch prediction mechanisms and 4. its instruction fetch bandwidth should be sufficient. In addition, we also assume, for simplicity, that each instruction can be executed in a single cycle. This abstract machine model is very useful for the preliminary studies of value prediction, since it provides us with a means to examine the pure potential of this phenomenon without being affected by the limitations of individual machines. As a part of our abstract perspective, we also assume that there is no extra penalty when values are not predicted correctly, since both Lipasti *et al.* ([26]) and our previous works ([16], [17]) have shown that most of the value mispredictions can be eliminated by employing a classification mechanism. In the following experiments we measure the effect of various value prediction policies and prediction schemes on the ILP under two different instruction window sizes. In the next subsection these measurements are compared versus the analytical model.

In the previous section we broadly studied the characteristics of the value prediction accuracy that various value predictors schemes can gain. It is important to indicate that the connection between the value prediction accuracy gained by these predictors and the expected boost in the ILP may not be straightforward. It is not sufficient that these schemes can correctly predict outcome values, these predictable values should also be in the "right places", where their contribution to the ILP would be significant (such as critical paths). In this subsection we will present a set of measurements that will indicate that value prediction can have a substantial contribution to the exploited ILP.

The gain of ILP available with value prediction is examined for two different value predictors, the last-value predictor and the stride predictor. Each of these predictors can operate in two modes: the first mode, termed the *scalar generation mode*, allows generation of only a single value prediction for an individual copy of an instruction that resides in the instruction window, while the second mode, termed the *eager generation mode* allows the predictor to generate multiple value predictions assigned to multiple copies of an individual instruction (if any) in the instruction window (e.g. in case of a loop). The hardware implementation and considerations of the eager generation mode is beyond the scope of this paper and they are presented in [18].

Figure 4.4 illustrates the ILP achieved using value prediction when the instruction window size is 40. It also compares the ILP achieved by different predictor schemes (last-value predictor and stride predictor) and prediction modes (scalar mode and eager mode) versus the ILP when value prediction is not employed. Indeed this figure indicates that the potential of value prediction to exceed the current ILP limitations is tremendous, e.g. in the benchmark *gcc* the ILP is increased from 14 to nearly 22, in *m88ksim* from 7 to 34, in *perl* from 15 to nearly 25 and in *vortex* from nearly 10 to 33. Figure 4.4 also illustrates that the stride predictor significantly accomplishes better performance than the last-value predictor in those benchmarks which exhibited stride value predictability (like *m88ksim* and *vortex*) in our previous experiments. For instance, in the benchmark *m88ksim* the stride predictor boosts the ILP from approximately 7 to 34, while the last-value predictor only achieves ILP of 13. In the rest of the benchmarks (like *go* and *li*) both predictors gain similar ILP with relatively smaller advantage to the stride predictor.

Another interesting observation shown by these experiments is that the eager mode barely improves the ILP that the last-value predictor achieves in all the benchmarks. However, the eager mode significantly improves the ILP that the stride predictor gains in those benchmarks exhibiting stride value predictability. For instance, in the benchmark *m88ksim* the stride predictor operating in eager mode gains ILP of 34, while the same predictor in scalar mode gains only ILP of 20. This phenomenon seems reasonable, since those instructions with output values exhibiting a tendency to appear in strides are likely to appear recurrently in the instruction window, like instructions in loops, and in order to better exploit them, the predictor should be allowed to operate in eager mode. In the floating point benchmarks, *swim* and *mgrid*, all the value predictors achieve similar ILP. Since the computation phase of the floating point benchmarks is less constrained by true-data dependencies, our measurements exhibit more ILP in comparison to the

initialization phase that tends to behave like an integer program. It can also be observed that although the stride predictor has significantly exhibited better prediction accuracy than the last-value predictor in ALU instructions, the overall ILP increase of both predictors is relatively limited since:

1. Usually the size of basic blocks and loop bodies in floating point programs is relatively big. As a result, the instruction window becomes too small to hold multiple iterations of a loop or even several basic blocks. Therefore, many of the loop-carried dependencies barely affect the ILP.
2. Both the last-value predictor and the stride predictor gained relatively small prediction accuracy in floating-point instructions which may also affect their achievable performance.

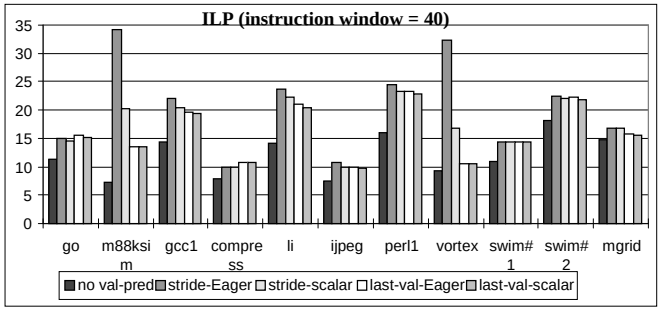


Figure 4.4 - The ILP gained by value prediction when instruction window size is 40.

Enlarging the instruction window size can enable current processors to look further ahead to find independent candidate instructions for parallel execution. In order to examine how this enlargement affects a machine that employs value prediction, we perform further experiments that are illustrated in figure 4.5. This figure exhibits the same measurements as figure 4.4, however this time when the instruction window size is 200. These measurements show that as the instruction window size is increased the extracted ILP grows as well. However, the most interesting observation that these experiments present is that the enlargement of the instruction window particularly affects the performance of the eager generation mode and the stride predictor. A bigger instruction window significantly increases the likeliness that it would maintain repeated copies of a same basic block or a same instruction simultaneously, such as multiple iterations of a loop. Such patterns can be usefully exploited by the eager generation mode. This mode allows the predictor to generate multiple value predictions to multiple copies of the same instruction and hence it can better utilize the deeper look-ahead provided by the enlargement of the instruction window. The stride predictor can also take advantage of these patterns, since appearances of recurrent instructions in the instruction window are also likely to generate output values that progress in strides.

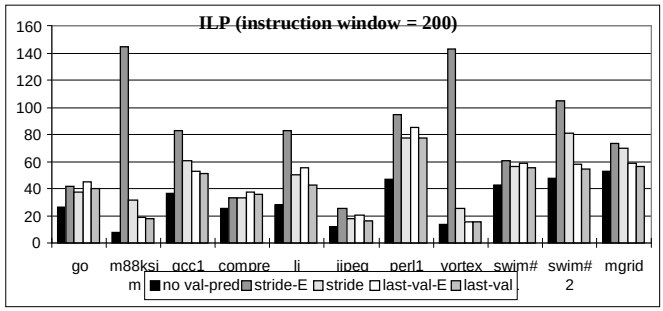


Figure 4.5 - The ILP gained by value prediction when instruction window size is 200.

In addition, it can be observed that even benchmarks which did not exhibit significant stride value predictability in the previous experiments, like *gcc*, *li* and *perl*, are significantly affected when they employ both stride predictor and eager generation mode. For instance, in benchmark *gcc* a stride predictor that operates in eager mode increases the ILP from 36 to 82, while the same predictor in scalar mode gains ILP of only 60. In addition the gap between the stride predictor and the last-value predictor, which gains ILP of approximately 50, becomes more noticeable. In the benchmark *li* the effect of the eager mode is even noticeable on the last-value predictor. The last-value predictor increases the ILP of this benchmark from 28 to 55 while the same predictor operating in scalar mode gains only 42. However, the best ILP among all the predictors in this benchmark is gained by the stride predictor (operating in eager generation mode) which boosts the ILP to 82. In the benchmarks that exhibited stride predictability (*m88ksim* and *vortex*) the effect of the instruction window enlargement is the most observable. In *m88ksim* the stride predictor in eager mode increases the ILP from 7.4 to 144 while the same predictor in scalar mode gains only 31. In *vortex* similar patterns are observed: the ILP is increased from 13.5 to 142 by the stride predictor operating in eager mode, while the same predictor in a scalar mode gains ILP of nearly 26. In addition, the enlargement of the instruction window affects the extractable ILP in the floating point benchmarks as well. The gap between the stride predictor in eager mode and the other schemes becomes much more significant since the instruction-window size enlargement can better expose the loop-carried dependencies. The stride predictor increases the ILP of *swim* (in the computation phase) from 47 to 104, and in the benchmark *mgrid* it increases the ILP from 53 to 73.

These results indicate that the potential of value prediction to increasing the ILP beyond the dataflow graph limitations is tremendous. In addition, till now several

studies such as [21] indicated that large instruction windows may not be cost-effective since they do not offer sufficient increase in the ILP to justify their hardware-cost. When value prediction is employed this claim may no longer be true. In addition, we have seen that both stride predictor and eager generation mode may significantly gain better ILP particularly when the size of the instruction window is increased. One of the directions that we consider ([18]) is to maintain a hybrid approach that consists of both predictor schemes (last-value predictor and stride predictor) and both value prediction generation modes (scalar mode and eager mode). This approach, motivated by our experiments, indicates that on one hand the absolute number of instructions exhibiting stride value predictability is relatively smaller than those exhibiting last-value predictability, however on the other hand, value prediction based on strides can significantly increase the ILP particularly in big instruction windows. Hence, in order to take advantage of these observations a machine could partition the limited resources assigned to the value prediction schemes more efficiently, e.g. by employing a small prediction table for the stride-predictor and a bigger table for the last-value predictor and only allowing value predictions based on strides to be generated in eager mode. These issues and many other implementation consideration issues are left for possible future studies.

4. 3. Comparison between the experimental and analytical results

In our comparison between the experimental measurements and the analytical results we consider the abstract machine model that we described in previous subsections. The configuration used for the comparison is the stride predictor operating the eager generation mode. The *experimental* increase in the ILP is obtained straightforwardly from the experimental measurements that were presented in the previous subsection. In order to calculate the *analytical* increase in the ILP predicted by our analytical model we use equations 4.3 and 4.6. Notice, that these equations need to be assigned with the prediction accuracy, *p*, that the predictor gains in each benchmark. The prediction accuracy, *p*, can be obtained from the measurements presented in Subsection 3.1.

Figure 4.6 illustrates the boost in the ILP under three different instruction window size: 40, 200 and infinite. For the finite instruction windows it illustrates both the experimental ILP increase and the analytical ILP increase. For the infinite instruction window it only illustrates the analytical evaluation. It can be observed that for most of the benchmarks the analytical model provides a good estimation for the ILP increase that is very close to the experimental results. On the other hand, for some benchmarks, the analytical model underestimated the potential of using value prediction. The reason for this observation is that in these programs, there are different "hot spots" which have a non-uniform relative contribution to the ILP.

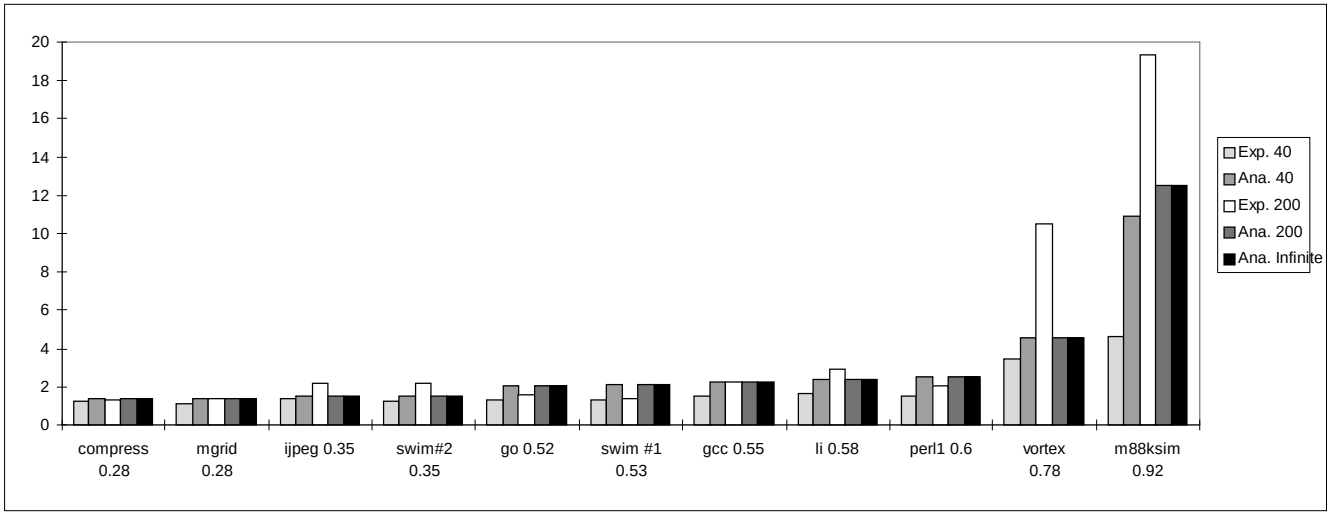


Figure 4.6 - Experimental versus analytical results of the expected boost in the ILP.

5. Conclusions, contributions and future directions

In this paper we presented an analytical and experimental study of the characteristics of *value prediction*. This concept is based on the observation that programs tend to re-use their recently generated values during execution. By taking advantage of this phenomenon, we can allow the system to collapse true-data dependencies and perform speculative execution based on predicted values.

So far, all modern computer systems have been based on the assumption that the dataflow graph of a sequential program forms an upper fundamental bound of the instruction-level parallelism. In order to better utilize the parallel resources in the system, the design of modern computers was focused on resolving name and control dependencies. Value prediction demonstrates that a similar technique can be used to improve parallelism by allowing the execution of data dependent operations out-of-order. We believe that this fundamental principle opens new horizons for future computer architectures.

Throughout this study, we have examined the concept of value prediction from three different perspectives: 1. we studied the characteristics of the phenomenon from the viewpoint of the program code, 2. we explored how these characteristics can be taken advantage of in order to improve the extractable ILP out of a sequential program, and 3. we provided a probability model in order to attain an analytical evaluation of the potential of value prediction to boosting ILP.

The main contributions of this study can be summarized as follows:

1. We extended the concept of value prediction and provided a related terminology. We introduced the notion of value predictability and distinguished between two different types of value predictability: last-value predictability and stride value predictability.
2. We presented substantial evidence confirming that programs tend to re-use their recently generated values and to exhibit predictable patterns of data values. In addition, we showed that programs can exhibit two kinds of value predictability patterns, last-value and stride. We also examined the distribution of these properties between different programs, instruction types and data types.
3. We introduced various value predictors and examined how efficiently they exploit different value predictability patterns.
4. We showed that the prediction accuracy does not distribute uniformly among the instructions in a program. Most programs exhibit two sets of instructions, highly value-predictable instructions and highly unlikely-predictable ones. These observations are the motivation to develop classification mechanisms, such as saturated counters, that were found to be very useful in preventing the "unlikely predictable" instructions from being candidates for value prediction.
5. Our preliminary observation indicates that different input files do not greatly change the value predictability of a program. This observation is encouraging for our future intention to use profiling-based compiler techniques that could classify the value predictability of instructions based on previous runs of the programs.
6. We showed that the use of value prediction techniques makes substantial contributions in respect to the extractable ILP. In addition, we have presented two different value prediction modes: the scalar generation mode and the eager generation mode, and examined their impact on the extractable ILP. Our ILP measurements indicate that the stride predictor significantly accomplishes better ILP than the last-value predictor in those benchmarks that exhibited strides value predictability, while in the rest of the benchmarks both predictors gain similar ILP with relatively smaller advantage to the stride predictor. We also observed that the eager mode particularly improves the overall performance of the stride predictor in those benchmarks which exhibited stride value predictability.
7. We showed that enlarging the instruction window, significantly improves the ILP gained by the eager generation mode and the stride predictor. We also observed that programs which exhibit last-value predictability can gain certain benefit from eager generation mode as well in this case.
8. We provided an analytical model which can be used to better understand the characteristics of value prediction, and to obtain a rough estimation of the potential of using it.

The study presented in this paper shows the importance of value prediction. We believe that using this new technique will lead to new directions in designing future computer architectures. Now that we have established the basis and the evidence of the new phenomenon, we intend to continue our research towards achieving a better understanding of the capabilities of value prediction and how it can be employed. We are in the process of examining different hardware mechanisms for efficient implementation of value prediction. In addition, we are looking at combining compiler support that can exploit our knowledge about value predictability.

References

[1] T. L. Adam, K. M. Chandy and J. R. Dickson. A Comparison of List Scheduling for Parallel Processing Systems. Communications of ACM, vol. 17, Dec., 1974, pp. 685-690.

[2] A. V. Aho, R. Sethi and J. D. Ullman. Compilers: Principles, Techniques and Tools. Reading, MA: Addison-Wesly, 1986.

[3] A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. In H. Ganzinger (ed.) Proceedings of the 2nd European Symposium on Programming. pp. 221-235. New-York, Mar., 1988.

[4] R. J. Blainey. Instruction Scheduling in the TOBEY Compiler. IBM J. Res. Develop., Vol. 38, No. 5, Sep., 1994, pp. 577-593.

[5] P. -Y. Chang, M. Evers and Y. N. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. The Conference on Parallel Architectures and Compilation Techniques (PACT), 1996, pp.48 - 57.

[6] A. E. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS Family. Computer, Vol. 14, Sep., 1981, pp.18-27.

[7] T. F. Chen and J. L. Bear. Effective Hardware-based Data Prefetching for High-performance Processors. IEEE Transactions on Computers, 44(5): 609-623, May, 1995.

[8] S. Davidson, D. Landskov, B. D. Shriver and P. W. Mallet. Some Experiments in Local Microcode Compaction for Horizontal Machines. IEEE Transactions on Computers, Vol. C-30, no. 7, July, 1981, pp. 460-477.

[9] T. A. Diep, C. Nelson and J. P. Shen. Performance Evaluation of the PowerPC 620 micro-architecture. In proceeding of the 22nd International Symposium on Computer Architecture. June, 1995, pp. 163-174.

[10] D. R. Ditzel and H. R. McLellan. Branch Folding in the CRISP microprocessor: Reducing the Branch Delay to Zero. Proceeding of the 14th International Symposium on Computer Architecture. June, 1987.

[11] J. H. Edmondson, P. Rubinfeld, R. Preston and V. Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. IEEE Micro, April, 1995, pp. 33-43.

[12] J. R. Ellis. Bulldog: A Compiler for VLIW Architecture. MIT Press, Cambridge, Mass., 1986.

[13] J. A. Fisher. The Optimum of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources. Ph.D. dissertation, Technical Report COO-3077-161. Courant Mathematics and Computing Laboratory, New York University, New York, October, 1979.

[14] J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branches from Previous Runs of a Program. Proceeding of the 5th Conference of Architectural Support for Programming Languages and Operating Systems. IEEE/ACM, Boston, Oct., 1992.

[15] J. W. C. Fuand J. H. Patel. Stride Sirected Prefetching in Scalar Processors. In the 25th Annual International Symposium on Microarchitecture, pp. 102-110, Portland, Oregon, Dec., 1992.

[16] F. Gabbay and A. Mendelson. Speculative Execution based on Value Prediction. EE Department TR #1080, Technion - Israel Institute of Technology, Nov., 1996.

[17] F. Gabbay and A. Mendelson. Can Program Profiling Support Value Prediction? Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December, 1997.

[18] F. Gabbay and A. Mendelson. The Effect of Instruction-Fetch Bandwidth on Value Prediction. In proceeding of the 25th International Symposium on Computer Architecture, June 1998.

[19] J. Gonzales and A. Gonzales. Speculative Execution via Address Prediction and Data Prefetching. Porceedings of the 11th Internation Conference on Supercomputing , pp. 196-203, July 1997.

[20] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. Microprocessors Report vol. 9, num. 2, Feb. 16, 1995.

[21] M. Johnson. Superscalar Microprocessor Design. Prentice Hall, Englewood Cliffs, 1990, N.J.

[22] R. M. Keller. Look-ahead Processors. Computing Surveys, volume 7, no.4, Dec., 1975, pp. 177-195.

[23] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW processors. SIGPLAN Conference on Programming Languages Design and Implementation, ACM (June), Atlanta, Ga., 318-328.

[24] M. Lam. A Systolic Array Optimizing Compiler. Boston: Kluwer, 1989.

[25] M. H. Lipasti, C. B. Wilkerson and J. P. Shen. Value locality and load value prediction. In proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), Oct., 1996.

[26] M. H. Lipasti, C. B. Wilkerson and J. P. Shen. Exceeding the dataflow limit via value prediction. In proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Dec. 1996.

[27] S. McFarling and J. Hennessy. Reducing the Cost of Branches. Proceedings of the 13th International Symposium on Computer Architecture. June, 1986. pp. 396-403.

[28] S. S. Pinter and A. Yoaz. Tango: a Hardware-based Data Prefetching Technique for Super-scalar Processors. In proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Dec., 1996.

[29] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proceeding of the 14th Annual Workshop on Microprogramming, Oct., 1981, pp. 183-198.

[30] M. Simone, A. Essen, A. Ike, A. Krishnamoorthy, T. Maruyama, N. Patkar, M. Ramaswami, M. Shebanow, V. Thirumalaiswamy and D. Tovey. Implementation Trade-offs in Using Restricted Data Flow Architecture in High Performance RISC Microprocessor. In proceeding of the 22nd International Symposium on Computer Architecture, June 1995, pp. 151-162.

[31] Introduction to Shade, Sun Microsystems Laboratories, Inc. TR 415-960-1300, Revision A of 1/Apr/92.

[32] A. Smith and J. Lee. Branch Prediction Strategies and Branch-Target Buffer Design. Computer 17:1, Jan. 1984. pp. 6-22.

[33] J. E. Smith. A Study of Branch Prediction Techniques. In proceeding of the 8th International Symposium on Computer Architecture, June 1981.

[34] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM J. Research and Development 11:1, pp. 25-33, Jan., 1967.
[35] D. W. Wall. Limits of Instruction-Level Parallelism. Proceedings of the 4th Conference on Architectural Support for Programming Languages and Operating Systems. Apr., 1991. pp. 248-259.
[36] S. Weiss and J. E. Smith. A Study of Scalar Compilation Techniques for Pipelined Supercomputers. Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems. Oct., 1987, pp. 105-109.
[37] T. Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture, Dec. 1992.
[38] T. Y. Yeh and Y. N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. Proceedings of the 19th International Symposium on Computer Architecture. May, 1992. pp. 124-134.
[39] T. Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that Uses Two Levels of Branch History. Proceedings of the 20th International Symposium on Computer Architecture. May, 1993. pp. 257-266.

Freddy Gabbay received the B.Sc. (summa cum laude) and M.Sc. degrees in electrical engineering from the Technion - Israel Institute of Technology, Haifa, Israel in 1994 and 1995 respectively. Currently he is a Ph.D. student (since 1995) in the Electrical Engineering Department at the Technion. His main research interest is computer architecture.

Abraham (Avi) Mendelson received the B.Sc. and the M.Sc. degrees in computer science from the Technion, Haifa, Israel in 1979 and 1982, and the Ph.D. degree from the ECE department, University of Massachusetts in 1990. He is a Lecturer of Electrical Engineering and a member of the Parallel systems laboratory at the Technion, Israel. His main research interests are in computer architectures, operating systems and distributed algorithms. Dr. Mendelson is a member of the Association for Computing Machinery and the IEEE Computer Society.