To appear in the 25th Annual International Symposium on Computer Architecture,Barcelona, Spain.Please do not distribute.

Modeling Program Predictability

Yiannakis Sazeides and James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
yanos@ece.wisc.edu, jes@ece.wisc.edu

Abstract

Basic properties of program predictability – for both values and control – are defined and studied. We take the view that program predictability originates at certain points during a program's execution, flows through subsequent instructions, and then ends at other points in the program. These key components of predictability: **generation**, **propagation**, *and* **termination***; are defined in terms of a model. The model is based on a graph derived from dynamic data dependences and a predictor.*

Using the SPEC95 benchmarks, we analyze the predictability phenomena both separately and in combination. Examples are provided to illustrate relationships between model-based characteristics and program constructs. It is shown that most predictability derives from program control structure and immediate values, not program input data. Furthermore, most predictability originates from a relatively small number of generate points. The analysis of obtained results suggests a number of ramifications regarding predictability and its use.

1 Introduction

The need for higher levels of instruction level parallelism is pushing high performance processor implementations toward widespread use of prediction and speculation. Ultimately this could lead to substantially new processing paradigms. However, it may be premature to start reaching for radically new paradigms at this point – first some basic research and understanding of program predictability is required. Data, addresses, and control interact in complex ways, and these interactions should be understood before prediction can be fully exploited for increased performance.

We propose a model for studying program predictability. This model, or related models, should prove useful for:

- Understanding the underlying phenomena that lead to predictability.
- Finding relationships among computations that may point to more accurate and more efficient predictors.
- Identifying critical points for prediction; i.e. places where prediction and speculation may have greater payoff.
- Eventually identifying new microarchitecture paradigms *based* on prediction rather than using it as just an add-on.

We do not achieve all these goals in this paper – we only take a first step. We concentrate primarily on model development and study the first of the above items in some depth. However, we include discussion suggesting possible directions for pursuing the others. Data value prediction is the primary focus, but we also include control prediction because the interactions between data and control are key for effective prediction and speculation. Further extensions to address and dependence prediction are clearly possible, but we do not pursue them here.

1.1 The Dynamics of Program Predictability

Because predictability is based on program behavior, we informally introduce some of the key concepts using a code example. Then, we make some general observations about program behavior that will provide a basis for understanding predictability phenomena.

Fig. 1 shows a frequently executed code sequence taken from the *invalidate_for_call* function in the SPECINT95 benchmark *126.gcc*. The code sequence tests bits in a mask that correspond to 64 machine registers; consequently, it is a loop that executes 64 iterations. Beside each instruction is a regular expression that describes the sequence of values produced by the instruction each time the function is called.

	OpCode	Operands	Values Produced
0	add	\$6,\$0,\$0	**0**
LL1:			
1	srl	\$2,\$6,5	$\mathbf{(0)}^{32}\mathbf{(1)}^{32}$
2	sll	\$2,\$2,2	$\mathbf{(0)}^{32}\mathbf{(4)}^{32}$
3	addu	\$2,\$2,\$19	$\mathbf{(0x1002f8b0)}^{32}\mathbf{(0x1002f8b4)}^{32}$
4	lw	\$2,(\$2)	$\mathbf{(0x8000bfff)}^{32}\mathbf{(0xffffffff)}^{32}$
5	andi	\$3,\$6,31	$\mathbf{(0,1,..,31)}^{2}$
6	srlv	\$2,\$2,\$3	$\mathbf{v_0,v_1,..,v_{62},v_{31}}$
7	andi	\$2,\$2,1	$\mathbf{(1)}^{14}\mathbf{(0)}^{1}\mathbf{(1)}^{1}\mathbf{(0)}^{15}\mathbf{(1)}^{33}$
8	beq	\$2,0,LL2	$\mathbf{(NT)}^{14}\mathbf{(T)}^{1}\mathbf{(NT)}^{1}\mathbf{(T)}^{15}\mathbf{(NT)}^{33}$
LL2:			
9	addiu	\$6,\$6,1	**1,2, .. ,64**
10	slti	\$2,\$6,64	$\mathbf{(1)}^{63}\ \mathbf{(0)}^{1}$
11	bne	\$2,0,LL1	$\mathbf{(T)}^{63}\ \mathbf{(NT)}^{1}$

Figure 1. Example Code from 126.gcc

For instruction 6 the v_i correspond to values not essential to the example. Note that the use of regular expressions is for illustrative purposes, program sequences do not necessarily have to be regular expressions – at least not compactly represented ones.

In the example, register \$6 is initialized by adding the value 0 to itself outside the loop (\$0 = 0, by definition in the instruction set we are using). Each time through the loop, the value in register \$6 is incremented by one from instruction 9. This means that the values in register \$6 form the sequence 0,1,2,3,...64. This sequence would be predictable by a stride predictor – i.e. the values in the sequence differ by a constant. After the second value in the sequence, a typical stride predictor would recognize the stride and start making correct predictions. Hence, assuming stride prediction, predictability has been *generated* at that point. And this predictability is *propagated* by each of the successive executions of instruction 9.

Instruction 1 also uses the values in register \$6. It shifts each of these by an immediate 5. The resulting sequence of 32 zeros and 32 ones is also largely predictable by a stride predictor (where the stride is 0). Hence, the predictability generated by the stride one sequence at the input of the shift (instruction 1) is propagated through to the output of the instruction. After 32 zero values, however, the output of instruction 1 will change to one. At that point the predictability *terminates*, but is almost immediately re-generated when the sequence of ones is detected by the predictor. Going on, instruction 2 shifts the result of instruction 1 by an immediate 2. Hence, the outputs of instruction 2 are also predictable – the predictability generated by instruction 9 propagates still further. And predictability continues to propagate through instructions 3 and 4, as can be seen by their output sequences.

Predictability is also influenced by communication between instructions. Often, there is direct communication within a basic block. When this happens, the predictability of values as they are produced is essentially the same as the predictability of values when they are consumed. There are other cases, however, where control instructions separate the producer and consumer of a value; for example, if a value is produced outside a loop and consumed repeatedly inside the loop. When control flow separates producer and consumer, the predictability characteristics of the value sequences may differ.

We study program predictability by considering sequences of values consumed and produced by instructions as a program executes. We assume prediction mechanisms that monitor these sequences and attempt to predict next value(s) in the sequence based on previously observed values. Hence, when we speak of *predictability*, we mean the ability of a specified finite state predictor to predict the next value in a sequence. A related, complementary concept is *unpredictability*, however we focus mainly on predictability. We use a pragmatic definition of predictability: it is related to specific predictors we have chosen to model.

In summary, we take the view that value sequences have characteristic patterns that lead to predictability, and these characteristics can be propagated through instructions and along dependence paths. That is, predictable inputs often lead to predictable outputs. Furthermore, predictable patterns must originate somewhere, and, similarly, predictability can be terminated. It is the relationships among generation, propagation, and termination of predictability that we are interested in studying.

1.2 Related Work

The most successful application of prediction in computer microarchitectures is for conditional branches [15, 18]. Branch prediction is used in virtually every high performance processor being sold today.

Recent results suggest that instructions exhibit data *locality*, that is, they tend to produce values from the same (often small) set a large fraction of the time [10]. Although locality by itself can be insufficient for predictability, its presence spurred the development of a number of value predictors that have accomplished increasingly higher prediction accuracy and demonstrated that values can be quite predictable [7, 9, 13, 14, 17].

Value prediction thus far has been used to enable the speculative execution of instructions, and its performance *potential* has been shown to be significant in a number of studies [7, 9, 10, 12]. However, for the potential to be realized, it is imperative to have high prediction accuracy and infrequent misspeculation. Misspeculation can be mitigated somewhat with the use of confidence mechanisms[8]; these are probably essential for effective value prediction and speculation.

Regarding actual sources of value predictability, relatively little has been done. Some anecdotal causes of predictability are given in [10], for example, but the relative importance of each is not quantified.

There are a number of other hardware and software techniques that exploit value locality in some form. One of these is *instruction reuse*[16]. Instruction reuse exploits the locality of both the inputs and outputs of an instruction by using a table to look up results computed with the same inputs at some time in the past. The concept of reuse is similar to *memoization* [1], a software technique that can be used to eliminate redundant computation by *storing precomputed results*[2]. Other software approaches that can take advantage of value behavior are *specialization* by static or *dynamic compilation*[4] and *software speculation*[6] extended to values.

1.3 Paper Overview

The paper is organized as follows: Section 2 introduces the model for predictability. Section 3 describes the simulation methodology. Predictability definitions for generation, propagation, and termination and simulations results are presented in Section 4. In Section 5, the model is used to study relationships between data and branch predictability. Section 6 summarizes the results and concludes with a discussion of possible ramifications.

2 Predictability Model

Thus far, we have discussed predictability and value behavior in terms of static program representations, with sequences of values flowing through the static representation; e.g. the static code and value sequences in Fig. 1. One could continue this approach and define a model based on the static program image. If this is done, however, evaluation of "predictability" becomes difficult, because the average prediction accuracy for the sequence generated by a static instruction falls somewhere between 0 and 1. And one must decide at what level of accuracy a static instruction becomes "predictable". By using a dynamic model, each instance of an instruction is either predictable or not. So in terms of the dynamic instances there is no ambiguity regarding predictability. And, by collecting statistics for the entire dynamic execution, we are in effect achieving an averaging over the program.

The dynamic predictability model has two principal components, a predictor and a dynamic prediction graph (DPG). We discuss the DPG in this section and describe specific predictors in Section 3.

The DPG is an acyclic directed graph derived from the dynamic dependence graph. Most nodes in the DPG correspond to instances of executed instructions, and a static instruction appears in the DPG as many times as the number of times it executes. A second node type is used to represent *input data values*. That is, values that are not directly computed by the program; these could be statically allocated or be the result of a program input operation. To simplify figures, we generically label these nodes with a **D** for data. *Immediate* values specified in an instruction are considered to be part of the instruction; they are not passed in as data values and are therefore never predicted. However, when collecting data, we keep track of instructions using immediate data because these affect predictability significantly.

A directed arc connects nodes *u* and *v* if there is a true data dependence from dynamic instruction *u* to *v*. The in-degree of a node is determined by the number of instruction source operands, whereas the out-degree of a node is determined by the number of its dependent instructions. Note that for a load instruction the in-degree is the number of address operands plus the number of stores the load depends on.

The predictor component of the model is used to predict the result (output) of an instruction at the time it is produced and predict instruction inputs at the time they are consumed. Note that a result value that is fanned out to the sources of many dynamic instructions is predicted once when it is produced and many times when it is later consumed. Each arc in the graph is labeled with a pair <x,y>, x,y $\epsilon\{$p,n$\}$. The first element of the pair is associated with the tail of the arc and indicates whether the producing instruction's result was predicted (p) or not predicted (n) correctly. The second element of the pair indicates whether the consuming instruction's source operand was predicted correctly.

D (data) nodes only have output arcs. By definition, all out-arcs from data nodes are labeled <n,y>, y $\epsilon\{$p,n$\}$; i.e. an initial data item is inherently not predictable. But it may be predicted correctly when used as an input operand by other instructions.

Generation, propagation, and termination of predictability can all be defined in terms of the DPG; this is most easily done pictorially (in Fig. 2). We show immediate values (i) inside nodes, as they are part of the instruction. Arc behavior is self-evident. Predictability is generated by a node if it has no correctly predicted inputs, yet the output is predicted correctly. Predictability is propagated by a node if it has at least one correctly predicted input and the output is predicted correctly. Predictability is terminated by a node if its output is not predicted correctly, but at least one of its inputs is. Immediate values may or may not be present in any of these cases.

To illustrate the parts of the prediction model defined thus far, Fig. 3 contains a portion of the DPG for the first three iterations of the example code in Fig. 1 with the arcs labeled using a stride predictor.

Instruction 1 has an immediate value 5 and takes a value

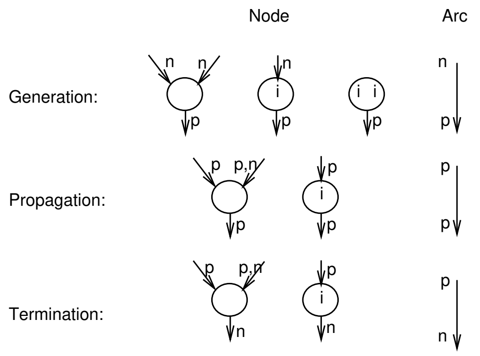

Figure 2. Predictability Definitions

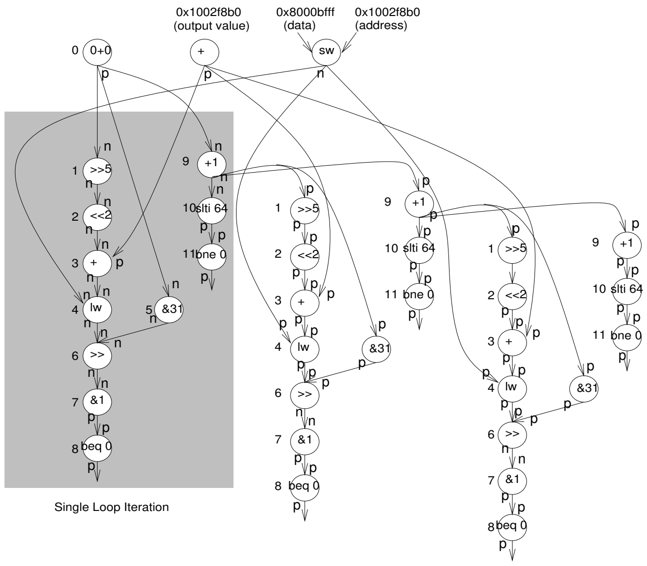

Figure 3. DPG for code in Fig. 1

from instruction 0 via register \$6. In the first instance of instruction 1, its output is predicted incorrectly – the predictor's state is such that it does not recognize a pattern in the input values. The first instance of instruction 2 is similar; non-immediate input and output are both predicted incorrectly. However, after the first instance, the inputs and outputs of both instructions become predictable because of generation on the arc between instructions 9 and 1, propagation through instruction 1, and propagation along the arc between instructions 1 to 2.

To illustrate memory operations, instruction 4 is a load instruction that reads a value that was stored earlier. Note that the value being read from memory is considered an input value to the load; the value placed in the result register is considered to be an output.

The model as presented does not directly capture the influence of control instructions on predictability (for example, branch and some jump instructions do not have any data dependent successors). Control instructions are important, however, and we attempt to model them – albeit in an *ad hoc* way. For branch and jump instructions, we depart slightly from a proper graph model by providing output arcs for branch directions and target PCs. However, these arcs do not connect to other nodes. This allows us to determine the influence of data predictability on control predictability.

To model the influence of control on data predictability, we focus on the way control flow affects the passing of data values between instructions. Because of control flow structures such as loops, a particular instance of a static instruction (corresponding to a single node in the DPG) may pass the same data value to multiple nodes in the DPG that all correspond to the same static instruction. When this happens, all such arcs passing the same data value are defined to be *repeated-use* arcs. If there is only one arc passing a value from a node to the instances of a given static instruction, then it is defined to be a *single-use* arc.

3 Simulation Methodology

Trace driven simulation was used to build the DPGs for some of the SPEC95 benchmark programs (all integer and representative floating point.) The simulator is based on the SimpleScalar toolset [3]. Benchmarks were compiled using the gcc compiler provided with the toolset using -O3 optimization.

DPG statistics are shown in Table 1. We note that the ratio of edges to nodes is about 1.5 for integer benchmarks and about 1.7 for floating point benchmarks. This fraction is a rough indicator of the number of immediate instruction values. The fraction of D nodes is negligible; fewer than .03 percent. Most of the benchmarks have fewer than 1 percent arcs connected to D nodes; the largest fraction is 2.6 percent (for *m88ksim*).

Predictability behavior was studied for three predictors: last-value, stride, and context–based. We chose these three predictor types because the first two are commonly proposed, and because the third (with a large prediction table) has the best prediction performance we have been able to produce thus far – although the design space for data predictors has barely been explored.

The last-value predictor is based on one proposed in [10] with 2^{16} entries and a 2 bit saturating counter replacement policy. In general, it predicts that a value is the same as the last time. The counter provides hysteresis; for example, the prediction value is replaced when the counter indicates two bad predictions in a row.

The stride predictor is the *2-delta* predictor, first proposed for addresses in [5], with 2^{16} entries. This predictor

Benchmark	Input Flags	Nodes (mil)	Arcs (mil)
compress	400000 e 2231	102	142
gcc	gcc.i	200	286
go	9 9	132	183
ijpeg	specmun.ppm	128	196
perl	scrabbl.in	39.8	59.0
m88ksim	modified train	119	161
vortex	modified train	101	149
xlisp	7 queens	199	292
applu	5x5x5 grid	60.0	99.6
fpppp	4 atoms	149	266
mgrid	32x32, 8 iter	119	219
swim	256x256 grid	199	326

Table 1. Benchmark Characteristics

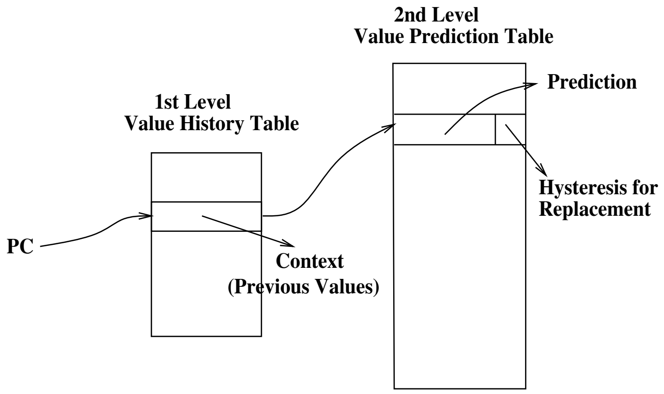

Figure 4. Two Level Context-Based Predictor

uses hysteresis to replace the predicted stride only when a different stride appears twice in a row.

The context-based predictor is a version [13] of the predictor proposed in [14]. The predictor uses a two level table, similar to two level branch predictors [18], and is shown in Fig. 4. The first level table has 2^{16} and the second level table 2^{20} entries. A first level table is accessed with a truncated instruction program counter (PC). The first level entry holds the last 4 values (in hashed form) produced by the instruction(s) that reference that entry – there can be aliasing in the tables. These 4 values form a "context" that is used to access the second level table. The second level table contains a predicted next value and a 3-bit saturating counter to guide replacement. An important property of the context-based predictor is that the second level table is shared. This does affect results and was done in part to reduce the memory required by our simulations. It also enables some instructions to benefit from the learned behavior of another instruction, although destructive interference is also possible. Sharing effects will be discussed when they affect results significantly.

Value predictors were used to predict all inputs and outputs of instructions with the exception of conditional branch directions for which a 64K entry gshare predictor is used [11]. The use of a branch predictor enables us to study the relation between branch and value predictability (Section 5). This particular branch predictor was chosen because it represents one of the better performing. An interesting alternative would be a two-level predictor that more closely mirrors the structure of the context-based predictor.

For all three prediction methods, we used separate, but identical, predictors for instruction inputs and outputs to prevent direct input/output prediction "short circuits" involving the same dynamic instruction. But, for certain instruction types that always pass an input value through to the output, only the input is predicted and the same predictability is assigned to the output. This was done for memory instructions and register indirect jumps. As a result such instructions never generate predictability.

An important caveat is that the predictors are immediately updated following a prediction. Introducing delayed update timing would have imposed particular implementation idiosyncrasies that may have limited the scope of the results in other ways.

4 Simulation Results

4.1 Overall Results

In Fig. 5 we present the overall generation, propagation and termination behavior of nodes and arcs. To permit relative comparison between node and arc behavior, throughout the paper the y-axes are expressed as a percentage of the total nodes and arcs. There is a set of bars for each benchmark. Each set gives results for all three predictors: last-value (**L**), stride (**S**), context-based (**C**). The two sets of bars at the right are averages of the integer and floating point benchmarks. Averages are calculated as the arithmetic mean. Note that the sum of all arcs and nodes for each case is less than 100%. The missing portion corresponds to the fraction of nodes and arcs that propagate unpredictability; i.e. have all unpredicted inputs and outputs.

All benchmarks appear to follow similar trends, i.e. there is cross-program consistency regarding predictability. Context-based prediction works better, as expected. Nodes and arcs tend to generate about the same amount of predictability, and their propagation is also similar – with arcs propagating slightly more than nodes. However, significantly more predictability is terminated at nodes than on arcs; reasons will be given later. Overall, propagation is the dominant predictability behavior. By adding the middle (darker) segments of the node and arc graphs, we see that on average 40%–65% of the nodes/arcs in the integer benchmarks and 25%–60% in the floating point benchmarks (de-

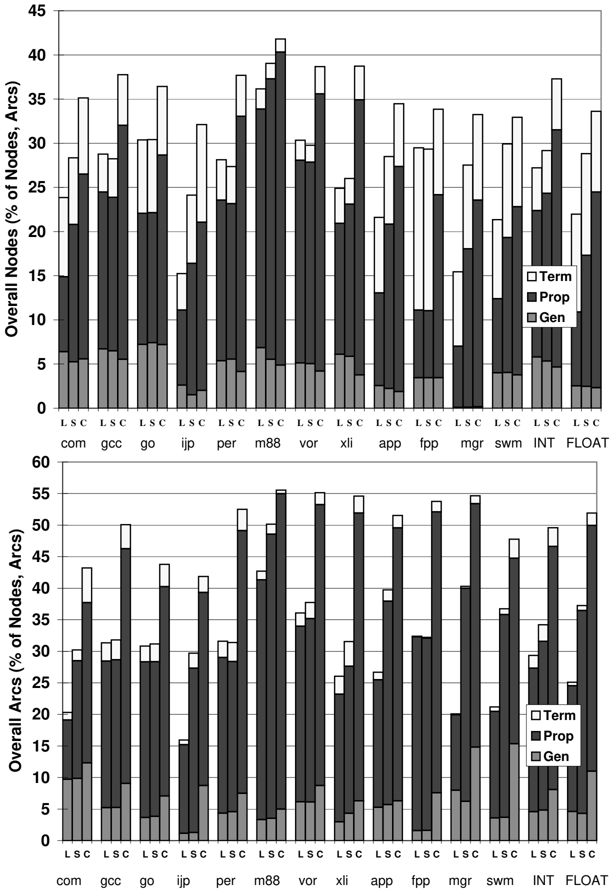

Figure 5. Overall Node and Arc Predictability

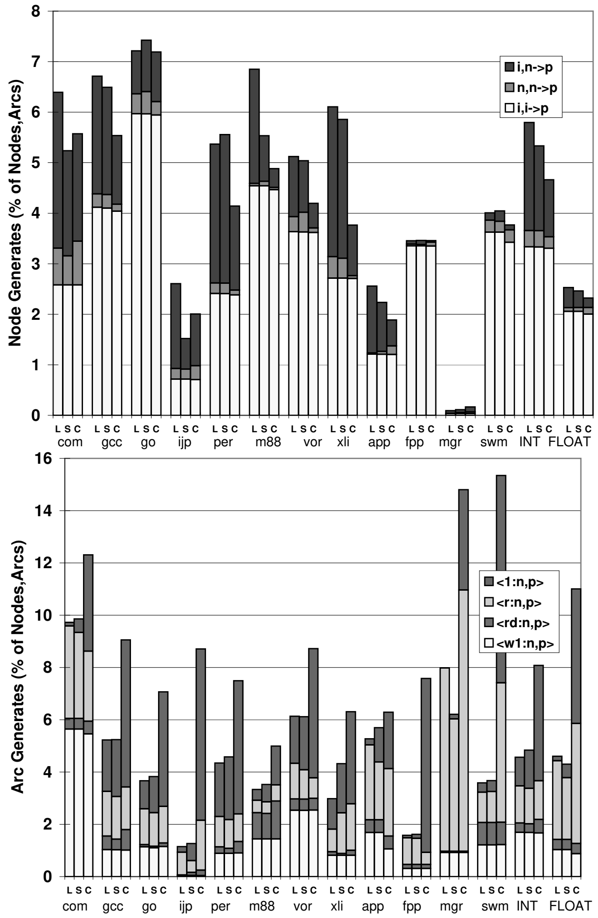

Figure 6. Node and Arc Generation

pending on the predictor used) propagate predictability.

In the following three subsections, we provide more detailed results for predictability generation, propagation, and termination.

4.2 Generation of Predictability

We first consider generation of predictability. Although the generate nodes and arcs comprise a small fraction of all nodes and arcs, an important point is that their overall influence is bigger than their numbers: a single generate can affect the predictability of many instructions via propagation. This is explored further in Section 4.5.

Our first conclusion is that predictability is most commonly generated by iterative control flow (e.g. as in loops). In terms of the graph model, iterative control flow appears as repeated-use arcs that generate predictability. In terms of programs, these arcs occur frequently when an instruction outside a loop initializes a memory or a register with a value that is unpredictable at the time it is generated, yet, because of repeated-use inside the loop the value is predictable.

To support our first conclusion, refer to Fig. 6. Note that y-axes of both graphs are expressed in the same units: percentage of total nodes and arcs, but the upper graph is scaled to cover a smaller range. In the arc generation graph, the bottom three segments all correspond to repeated-use arcs. With stride and last-value prediction, these three segments account for much more more generation than any of the other segments; for context-based prediction, single-use arcs (discussed below) provide about as much. We divided the repeated-use arcs into three segments for more detailed understanding of the phenomena at work.

The bottom segment, denoted <**w1:n,p**>, occurs when a static source instruction producing the value executes only once during the entire lifetime of the program. We call this repeated-use *write-once* control flow. An example of program behavior that can lead to write-once generation is the use of a global data pointer.

The second segment, denoted <**rd:n,p**> , occurs when the source data value comes from program input data (data nodes). We call this *repeated-input use* control flow. Reads from statically allocated arrays can lead to repeated-input use predictability. Generation due to write-once and repeated-input use is important because their corresponding values remain invariant for a long time (often for the entire program execution) and offer potential for static/dynamic specialization.

The third segment from the bottom includes all the other cases of repeated-use, i.e. where the source instruction is executed multiple times, and is denoted as <**r:n,p**>.

Intuitively, repeated-use arcs would most often result in last-value predictability, and in the integer benchmarks this borne out. Stride prediction and context-based prediction include last-value prediction as a special case, yet they provide similar generation for repeated-use arcs. On floating point benchmarks, context-based prediction does provide more generation, but this is mainly caused by programs that repeatedly scan the same array of input data. Context-based predictors generate predictability at these points.

The second conclusion regarding generation is that single-use arcs also contribute significantly to generation, especially for context-based prediction. These arcs are denoted <**1:n,p**> in the bottom graph of Fig. 6. In terms of program structure, these often occur when the producing and consuming instructions are in different basic blocks, and they are separated by some kind of “filtering” conditional branch that converts the unpredictable producer sequence into a predictable consumer sequence. A simple case occurs when a branch tests a relatively hard-to-predict sequence for some constant value. Any consuming instructions on the true path of the branch will see a very predictable (i.e. constant) sequence. Context-based prediction works better than last-value or stride for single-use because of cases where filtering branches test for a range of values. Constructive sharing in the prediction table also helps context-based prediction for these cases.

The third conclusion regarding generation is that instructions with all immediate inputs are responsible for most of the predictability caused by instructions. Results for node predictability generation are shown in the top graph of Fig. 6. Instructions with only immediate values are denoted as **i,i–> p** . In terms of programs, these occur for load immediate instructions or when instructions initialize a register (typically to zero).

Generation also occurs at some nodes when there is at least one unpredicted input. These cases are denoted with **n,n–>p** and **i,n–>p** in the graph. The majority of these, 70%-95%, depending on the predictor used, are due to branch, compare, logical, and shift instructions. These operations often produce few unique output values and often produce the same value for many consecutive executions. That branches and compares have few output values is obvious. Logic operations often use masks that select a small number of bits, and shifts often have large shift counts that clear out large number of bits (see instruction 1 in Fig 1). Often the shift count or logical mask is an immediate, so the majority of the time one of these instruction generates predictability, it involves at least one immediate value.

For node generation, *mgrid* stands out; it has almost no generation at nodes because very few instructions in this benchmark have immediate inputs.

4.3 Propagation of Predictability

Most propagation occurs on single-use arcs, denoted as <**1:p,p**> in the bottom graph of Fig. 7. This should be expected because single-use arcs include all the dependences between instructions in the same basic block.

Propagation through repeated-use arcs, denoted as <**r:p,p**>, occurs more often in the floating point benchmarks and happens in doubly nested loops where the output value of an instruction in the outer loop is predictable and an inner loop makes repeated-use of that value. Propagation due to <w1:p,p> arcs is rare and shown for completeness.

Propagation through nodes most often occurs when all inputs are predictable. These cases are denoted **p,p–> p** and **p,i–> p** in the top graph of Fig 7. This is expected behavior – if all inputs of an instruction are predictable, the output is often predictable as well.

Memory instructions are responsible for most of the nodes that propagate predictability and have an unpredictable input (denoted as **p,n–>p**). These propagates occur when data is predictable but the register used to calculate the address is not.

4.4 Termination of Predictability

Most termination of predictability happens at instructions where a predictable input is combined with an unpredictable input, and the instruction output is unpredictable. These are shown in the top graph of Fig. 8, and are denoted **p,n–>n**. Additional analysis (not given here) revealed that this is caused primarily by memory instructions with predictable addresses but unpredictable data. The remainder of the **p,n–> n** terminations are mostly due to integer or floating add instructions.

Another significant cause of termination occurs because of control flow and is similar to the “filtering” described for the generation of predictability. However, in this case the filtering terminates predictability. That is, the producer and consumer instructions may be in different basic blocks with a conditional branch in between. Even though the producer instruction has a predictable output, the conditional branch only allows the consumer to see some proper subset of the producer instruction’s outputs. Because only a subset is observed at the consumer, they may be much less predictable. In terms of the graph model, these are single-use arcs and are denoted as <**1:p,n**> in the lower graph of Fig. 8.

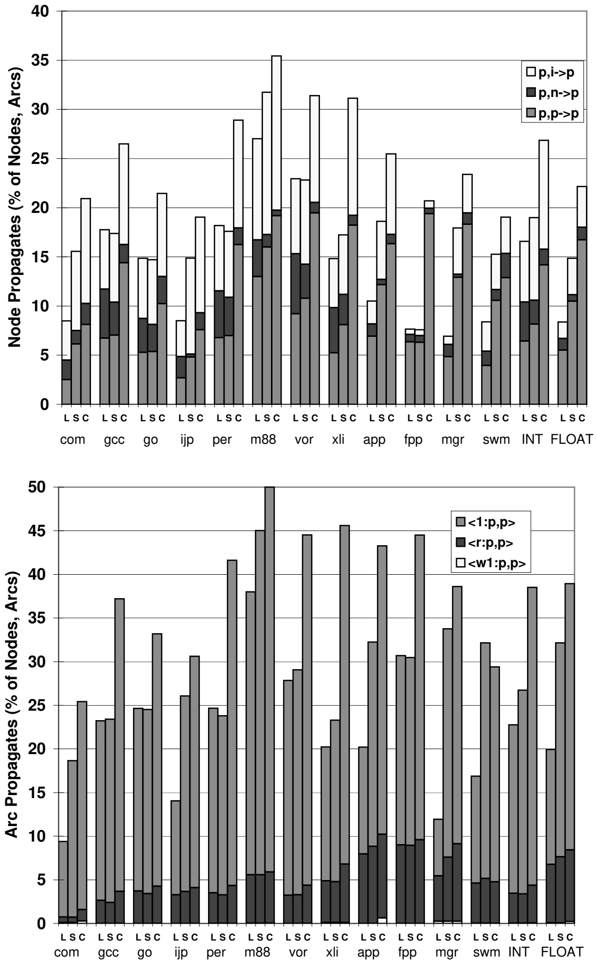

Figure 7. Node and Arc Propagation

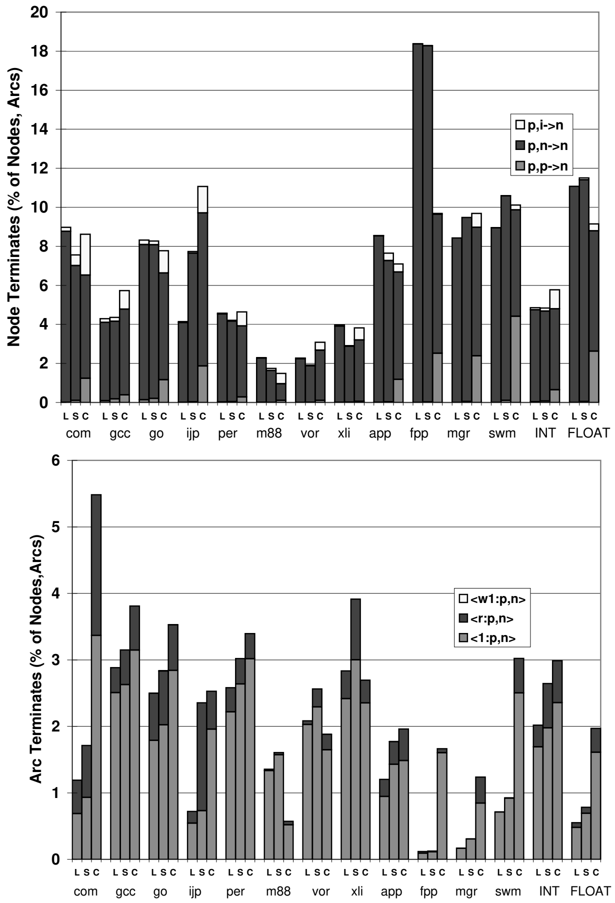

Figure 8. Node and Arc Termination

Control flow termination also occurs when different instances of the same static instruction consume values produced by different static instructions. The values produced by some of the static instructions are predictable, but their combination may not be. This suggests predictors that combine values with the PCs of producing instructions. Termination due to <w1:p,n> arcs is negligible and shown for completeness.

Turning back to nodes, one would expect termination when all instruction inputs are predictable to be rare. And for last-value and stride prediction, they are indeed rare. These cases are denoted **p,p–>n** and **p,i–>n** in Fig. 8. For context-based prediction, however, **p,p–>n** and **p,i–>n** nodes are much less rare. This is often due to the limited context history length maintained in the predictor. Consider the sequence that consists of the integers from 0-9, repeatedly. A context-based predictor would successfully predict these values if its history length is one. However if this sequence is the input to an AND instruction and the other input is a mask consisting a single 1 in the 4th bit position, the output sequence becomes 00000000110000000011... A context-based predictor with history length one will now yield some mispredictions each time through the repeating sequence. Further analysis (not detailed here) shows this often happens with compare, logical, shifts, and branch instructions. This suggests that when predicting the output values of these instructions, it may be beneficial to correlate with (possibly predicted) input data values of the same or preceding instructions.

4.5 Path Behavior

Thus far, we have analyzed the three major components of predictable behavior separately. It is perhaps more interesting to consider their inter-relationships. In particular, we focus on predictable paths. A *predictable path* begins at a generate node or arc, and includes only propagate nodes and arcs. An interpretation of paths is that all predictable values along a path owe their predictability to the original generator(s). Or, conversely, a specific generator may influence a number of predictable values “downstream”.

We conjecture that paths indicate potential correlation

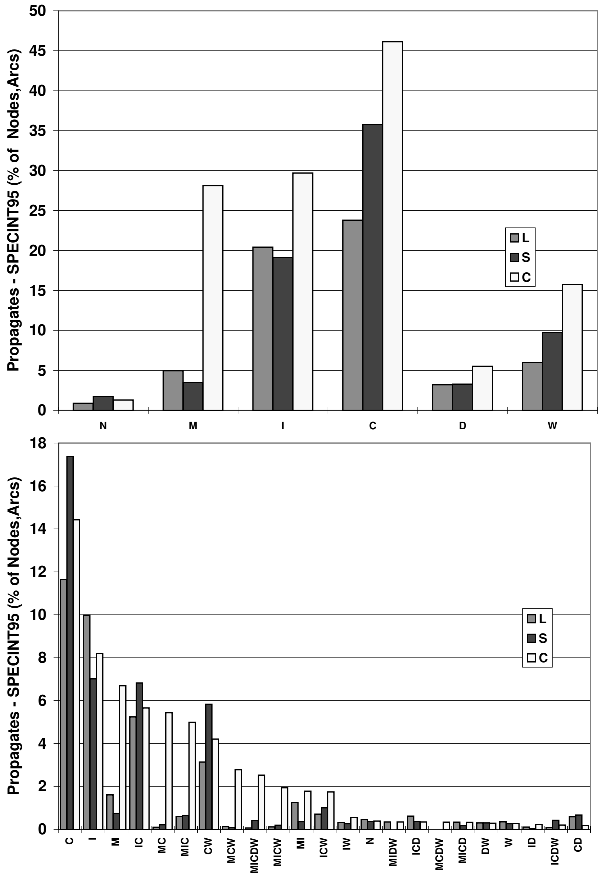

Figure 9. Overall and Combinations Contribution of Generates to Propagation

among values. That is, a predictable value upstream probably correlates well with a predictable value downstream along a predictable path. This can potentially be used to aid in prediction. The distance between the two may indicate how early a correlating value is available, and the paths flowing into a given node may indicate the upstream values that should be used for correlation.

Note that all subpaths of a path are considered as paths. This leads us to analyze all the predictable paths, and for each determine the generating node or arc at its beginning. These, in effect, are the sources of predictability.

In doing the path analysis, we combine some of the generation classes. In particular we consider the following six major classes of generators:

C, control flow, including both <r:n,p> and <1:n,p>.

D, input data, <rd:n,p>.

W, write-once, <w1:n,p>.

I, nodes, all immediate inputs , i,i–>p.

N, nodes, all inputs unpredictable, n,n–>p.

M, nodes, mixed immed. and unpredict. inputs, n,i–>p.

Results given in Fig. 9 are for all three predictors and are averages over the integer benchmarks. The graphs show the percentages of propagating arcs and nodes on predictable paths beginning at the various classes of generators. The top graph in the figure shows the overall contributions from each generator type. Note that in the top graph, a given arc or node can be included more than once if it is on paths from more than one generator class. The results indicate that the dominant mechanism influencing predictability is control flow. For context-based predictors, generation at single and repeated-use arcs initiate predictable paths that include 45 percent of all arcs or nodes in the DPG. Instructions with only immediate inputs are the second most important generator. With context-based prediction, they initiate paths comprising about 30 percent of the DPG graph.

On the other hand, the predictability influenced by program input data (D) is relatively small. This leads us to the general conclusion that most predictability is caused by program structure and internal data values, i.e. control flow and immediates, and not by program input data.

The bottom graph in Fig. 9 shows the contributions of specific combinations of generators; these are the top 24 combinations when sorted with respect to their set sizes for context-based prediction. In this graph, a given arc or node is only counted once. The set labeled "C" is the percentage of arcs and nodes that are influenced only by control flow generation. This is the largest set for all three predictors, containing roughly 12 to 17 percent of the arcs and nodes, depending on the predictor. The set "I" are influenced only by i,i–>p generators; i.e. instructions that have not data inputs. These account for 10 percent of the arcs and nodes with last-value prediction, and suggest the use of some form of specialization or dynamic constant propagation. The set "CI" are influenced by both control flow and immediate generators. For two of the three predictors, CI is the third largest set, and underscores the importance of control flow and immediates. For the context-based predictor, CI is the fourth largest set. The set of "M", nodes with mixed unpredictable and immediate inputs is slightly larger; this again indicates the influence of immediate values.

We now expand our analysis to predictable trees. That is, a generator node or arc is at the start of multiple predictable paths, which, collectively, form a tree. For this analysis, we concentrate on a few specific benchmarks.

Fig. 10 illustrates some of the characteristics of the predictability trees for the benchmark *126.gcc* using the context-based predictor. The curve marked "trees" shows the lengths of the longest paths within predictable trees. This is a cumulative graph, so, for example, about 90 percent of the generates form trees with longest paths containing 8 or fewer propagating nodes and arcs. This graph indicates that *most* of the predictable trees are relatively shallow. However, a few of the trees are very deep, and include a large number of the propagate nodes and arcs. This is shown in the second curve labeled "aggregate propagation".

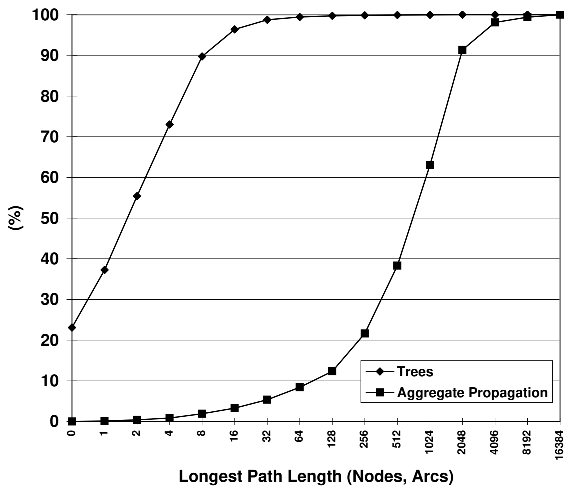

Figure 10. Longest Tree Path and Aggregate Propagation

Aggregate propagation is the total number of nodes and arcs in all trees. This graph shows, for example that 80 percent of aggregate propagation is due to trees with longest path length 256 or more. Hence, relatively few generates influence a large proportion of the predictability. This suggests that predictors that correlate on a few points of generation can be used in making a large proportion of predictions.

Taken collectively, the predictability trees are intermingled to form the predictable regions. That is, a given propagate node or arc can belong to multiple trees. Hence, one can start at a propagate node or arc and trace back to generator nodes and arcs, and get additional information. The top graph in Fig. 11 shows the number of generates that influence a given propagate for the benchmarks *129.compress*, *099.go*, and *126.gcc*, with the context-based predictor. This is also a cumulative graph. The data indicate that about 70%–85% of the propagates are influenced by fewer than 4 generates. This suggests that the predictability trees are not highly intermingled. It also indicates that to predict a value by correlating with all of its upstream generators, relatively few correlating values would be needed.

The bottom graph in Fig. 11 shows the distance between a propagate and the earliest generate that influences it. That is, the longest distance along a propagating path from any generate to the propagate node or arc under consideration. This is also a cumulative graph. For a simple control flow program (loop dominated such as *compress*) about 50% of the propagate nodes/arcs are no more than 64 nodes and arcs away from the farthest generate that influences them. On the other hand, these lengths are greater for complex control programs, *go* and *gcc*, where about 50% of the propagates are influenced by a generate 1024 or more away. Such distances may indicate how early a correlated prediction can be made, i.e. when correlating on the earliest generator.

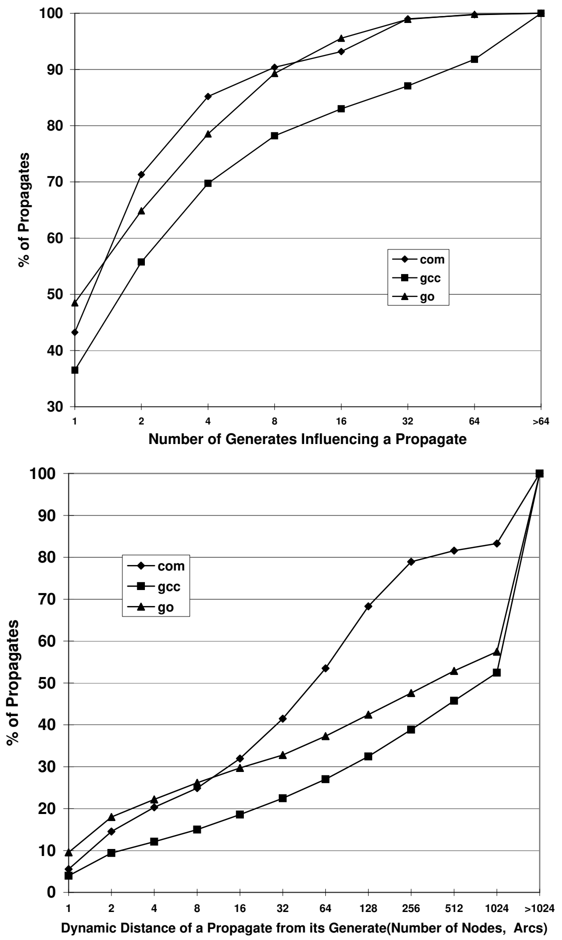

Figure 11. Number of Generates Influencing a Propagate and Dynamic Distance between Propagates and Generates

4.6 Predictable Contiguous Sequences

Although these results do not come from the graph model, because they are not based on dependences, it is interesting nevertheless to consider contiguous sequences of predictable instructions in the dynamic instruction stream. These are instructions where all inputs and outputs are correctly predicted. We feel this is of interest because an implementation using predictability may work better on contiguous sequences of instructions rather than single instructions.

Fig. 12 shows the number instructions (nodes) contained within instruction sequences of different lengths. This graph includes averaged benchmark data for all three predictors and integer benchmarks.

The data show that fairly long predictable sequences are common with all three predictors. For example, with the context predictor, 13% of the instructions are in blocks of length 9-16. By adding data points, it can be determined that 40% of the instructions, with the context-based predic-

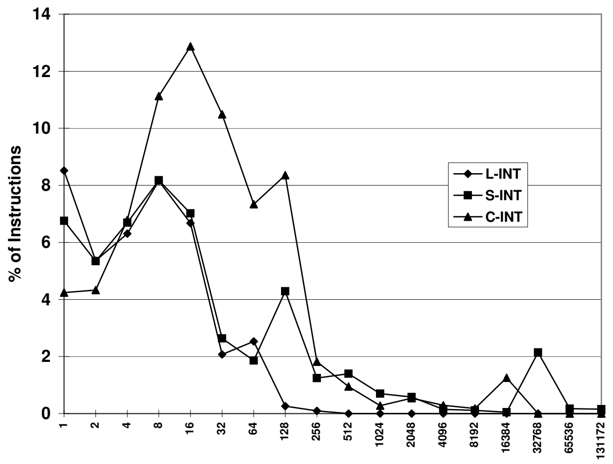

Figure 12. Predictable Sequence Length

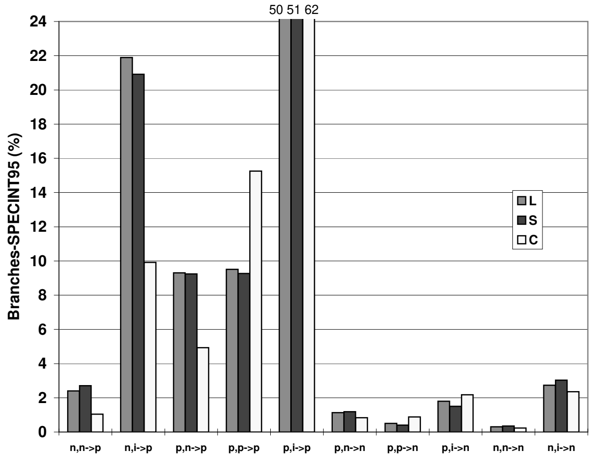

Figure 13. Branch Predictability Behavior

tor, are in sequences containing 9-256 instructions. Last value prediction is also interesting. The data show that many instructions occur in sequences where all instructions have the same inputs and outputs as their previous execution.

5 Branch Instructions

In this section we apply the predictability model to conditional branch instructions, and consider ways that data predictability may affect branch predictability.

Fig. 13 shows the predictability behavior of all branch instructions for the integer benchmarks. Recall that branch outputs are predicted with a 64K entry gshare predictor. The x axis of the graph is labeled using notation similar to that we have been using. Only branch instructions (nodes) are considered. Branches that generate or propagate predictability are labeled x,y–> p, where x and y are either n, p, or i. Mispredicted branches are labeled x,y–> n. The overall prediction accuracy of gshare was 93%.

Many of the branch nodes propagate predictability; i.e. when their output is predictable they have at least one input that is value predictable. This is the case for 70%-82% of the branches, depending on the predictor used. Only a small fraction of branches 1%–2% are predicted correctly when all inputs are unpredictable values.

It is perhaps more interesting to consider mispredicted branches. The data show that branch mispredictions when all inputs are unpredictable are relatively rare (fewer than 0.5% of all branches). Branches with only unpredictable inputs and immediate values account for about 2% of the branches. Slightly over half of the branch mispredictions, however, occur when all input values are predictable (p,p–>n or p,i–>n). This suggests that branch prediction can be enhanced by incorporating data values into the predictor in some form – for example, including input values from previous instances of the same static branch in a history register. Updating the history in a deterministic way would be significant design issue, however.

6 Summary and Discussion

We have developed a model for studying program value and control predictability. In terms of the DPG, we described some readily identifiable constructs that lead to predictability generation, propagation, and termination. The relative contribution of the different causes for each behavior is investigated using three different predictors: last-value, stride and context-based. First we studied the three categories of predictability in isolation. We then performed path analysis to determine which are the important sources of predictability. Finally, we considered branch predictability in terms of value predictability. Some of the more significant results are:

- Most predictability can be traced back to program control structure and immediate values. Program input data are a relatively unimportant source of predictability.

- The majority of generates are at the beginning of paths that propagate through fewer than 8 nodes and arcs. However, a very small fraction of generates originate very long paths and influence the majority of the predictability.

- Predictability is often terminated by unpredictable memory data or when a correctly predicted value is combined with an incorrectly predicted value in an instruction. Another significant way that predictability is terminated is through single-use control flow.

That is, conditional branches between the producer and the consumer causes irregularity in patterns read from one or many predictable producers.

- Slightly over half of mispredicted branch instructions have predictable input values.

There are a number of possible applications of the model. One application is making better predictors. The model can point to cases where there should be correlation among data and control values. It may be advantageous to feed control information into data predictors and vice versa. Or data values corresponding to different instructions may be correlated; for example, the value at a generate point may correlate with values on paths that it originates. This suggests that occurrences of **p,p–>n** and **p,i–>n** can possibly be exploited to improve prediction accuracy. That is, one could perform output predictions by correlating on predecessor instructions' input values.

The model may also be used to explore new paradigms. For example the large number of **p,p–>p** and **p,i–>p** nodes and <p,p> arcs naturally suggest speculation and/or reuse/memoization of **regions** with predictable nodes and arcs. Regions could possibly start execution early, and/or be precomputed or eliminated all together in hardware and/or software.

Finally, as we did this work, it became evident that unpredictability is as interesting as predictability. For example, in the branch part of the study, insight was gained by considering the behavior of the unpredicted branches. Similarly, study of unpredictable values may give insight into making them predictable; this remains for future research.

7 Acknowledgements

This work was supported in part by NSF Grants MIP-9505853 and MIP-9307830 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

The authors would like to thank Mike Smith for his valuable comments on an earlier version of this work and Subramanya Sastry for his suggestions in refining and better understanding the predictability model.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill Book Company, New York, 1985.

[2] J. L. Bentley. *Writing Efficient Programs*. Prentice-Hall Inc., New Jersey, 1982.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[4] C. Consel, L. Hornof, F. Noel, J. Noye, and N. Volanschi. A Uniform Approach for Compile and Run-time Specialization. Technical Report 979, INRIA, December 1995.

[5] R. J. Eickemeyer and S. Vassiliadis. A Load Instruction Unit for Pipelined Processors. *IBM Journal of Research and Development*, 37(4):547–564, July 1993.

[6] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[7] F. Gabbay and A. Mendelson. Speculative Execution Based on Value Prediction. Technical Report (Available from http://www-ee.technion.ac.il/fredg), Technion, November 1996.

[8] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, December 1996.

[9] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, December 1996.

[10] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Data Speculation. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[11] S. McFarling. Combining Branch Predictors. Technical Report DEC WRL TN-36, Digital Western Research Laboratory, June 1993.

[12] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace Processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 138–148, December 1997.

[13] Y. Sazeides and J. E. Smith. Implementations of Context–Based Value Predictors. Technical Report ECE-TR-97-8, University of Wisconsin-Madison, Dec. 1997.

[14] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, December 1997.

[15] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[16] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 194–205, June 1997.

[17] K. Wang and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 281–290, December 1997.

[18] T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.