Microprocessors and Microsystems 22 (1998) 315–332

MICROPROCESSORS AND
MICROSYSTEMS

Improving achievable ILP through value prediction and program profiling

Freddy Gabbay[a,*], Avi Mendelson[b]

[a]Department of Electrical Engineering, Technion – Israel Institute of Technology, 32000 Haifa, Israel
[b]National Semiconductor, Israel

Abstract

This paper explores the possibility of using program profiling to enhance the efficiency of value prediction. Value prediction attempts to eliminate true-data dependencies by predicting the outcome values of instructions at run-time and executing true-data dependent instructions based on that prediction. So far, all published techniques in this area have examined hardware-only value prediction mechanisms. In order to enhance the efficiency of value prediction, it is proposed that program profiling be employed to collect information that describes the tendency of instructions in a program to be value-predictable. The compiler that acts as a mediator passes this information to the value-prediction hardware mechanisms. Such information is exploited by the hardware in order to reduce mispredictions, better utilize the prediction table resources, distinguish between different value predictability patterns and still benefit from the advantages of value prediction to increase instruction-level parallelism. We show that the proposed method outperforms the hardware-only mechanisms in most of the examined benchmarks. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Instruction-level parallelism; Value prediction; Program profiling

1. Introduction

Modern microprocessor architectures are increasingly designed to employ multiple execution units that are capable of executing several instructions (retrieved from a sequential instruction stream) in parallel. The efficiency of such architectures is highly dependent on the instruction-level parallelism (ILP) that they can extract from a program. The extractable ILP depends on both processors' hardware mechanisms as well as the program's characteristics ([1,2]). The program's characteristics affect the ILP in the sense that instructions cannot always be ready for parallel execution due to several constraints. These constraints have been classified into three classes: true-data dependencies; name dependencies (false dependencies); and control dependencies ([1–3]). Both control dependencies and name dependencies are not 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 [4–6,1,2,7–11]. As opposed to name dependencies and control dependencies, only true-data dependencies were 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 [2].

Recent works [12–15] have proposed a novel hardware-based paradigm that allows superscalar processors to exceed the limits of true-data dependencies. This paradigm, termed value prediction, attempted to collapse true-data dependencies by predicting at run-time the outcome values of instructions and executing the true-data dependent instructions based on that prediction. Within this concept, it has been shown that the limits of true-data dependencies can be exceeded without violating the sequential program correctness. This claim breaks two accepted fundamental principles:

1. The ILP of a sequential program is limited by its data-flow graph representation.
2. In order to guarantee the correct execution of the program, true-data dependent instructions cannot be executed in parallel.

It was also indicated that value prediction can cause the execution of instructions to become speculative. Unlike branch prediction that can also cause instructions to be executed speculatively since they are control dependent, value prediction may cause instructions to become speculative since it is not assured that they were fed with the correct input values.

Recent works in the area of value prediction has considered hardware-only mechanisms. In this paper we provide new opportunities enabling the compiler to support value

* Corresponding author. E-mail: fredg@psl.technion.ac.il

prediction by using program profiling. Profiling techniques are being widely used in different compilation areas to enhance the optimization of programs. In general, the idea of profiling is to study the behavior of the program based on its previous runs. In each of the past runs, the program can be executed based on different sets of input parameters and input files (training inputs). During these runs, the required information (profile image) can be collected. Once this information is available it can be used by the compiler to optimize the program's code more efficiently. The efficiency of program profiling is mainly based on the assumption that the characteristics of the program remain the same under different runs as well.

In this paper we address several new open questions regarding the potential of profiling and the compiler to support value prediction. Note that we do not attempt to replace all the value prediction hardware mechanisms in the compiler or the profiler. We aim to revise certain parts of the value prediction mechanisms to exploit information that is collected by the profiler. In the profile phase, we suggest collecting information about the instructions' tendency to be value-predictable (value predictability) and classify them accordingly (e.g. we can detect the highly predictable instructions and the unpredictable ones). Classifying instructions according to their value predictability patterns may allow us to avoid the unpredictable instructions from being candidates for value prediction. In general, this capability introduces several significant advantages. First, it allows us to better utilize the prediction table by enabling the allocation of highly predictable instructions only. Moreover, categorizing value predictabilty patterns of instructions, lets us simultaneously exploit various value predictability patterns by using different value predictors and assigning each predictor to a different subset of instructions. In addition, in certain microprocessors, mispredicted values may cause some extra misprediction penalty due to their pipeline organization. Therefore the classification allows the processor to reduce the number of mispredictions and saves the extra penalty. Finally, the classification increases the effective prediction accuracy of the predictor.

So far, previous works have performed the classification by employing a special hardware mechanism that studies the tendency of instructions to be predictable at run-time [12–15]. Such a mechanism is capable of eliminating a significant part of the mispredictions. However, since the classification was performed at run-time, it could not allocate in advance the predictable instructions in the prediction table. As a result unpredictable instructions could have uselessly occupied entries in the prediction table and evacuated the predictable instructions. In this work we propose an alternative technique to perform the classification. We show that profiling can provide the compiler with accurate information about the tendency of instructions to be value-predictable. The role of the compiler in this case is to act as a mediator and to pass the profiling information to the value prediction hardware mechanisms through special opcode directives. We show that such a classification method outperforms the hardware-based classification in most of the examined benchmarks. In particular, the performance improvement is most observable when the pressure on the prediction table, in term of potential instructions to be allocated, is high. Moreover, we show that the new classification method introduces better utilization of the prediction table resources and avoidance of value mispredictions.

The rest of this paper is organized as follows: Section 2 summarizes previous works and results in the area of value prediction; Section 3 presents the motivation and the method proposed by this work; Section 4 explores the potential of program profiling through various quantitative measurements; Section 5 examines the performance gain of the new technique; Section 6 concludes this paper.

2. Previous works and results

This section summarizes some of the experimental results and the hardware mechanisms of the previous familiar works in the area of value prediction [12–15]. These results and their significance have been broadly studied by these works, however, we have chosen to summarize them since they provide substantial motivation to our current work.

Section 2.1 and Section 2.2 are dedicated to value prediction mechanisms: value predictors and classification mechanisms. Sections 2.3–2.5 describe the statistical characteristics of the phenomena related to value prediction. The relevance of these characteristics to this work is presented in Section 3.

2.1. Value predictors

Previous works have introduced two different hardware-based value predictors; the last-value predictor and the stride predictor. For simplicity, it was assumed that the predictors only predict destination values of register operands, even though these schemes could be generalized and applied to memory storage operands, special registers, the program counter and condition codes.

2.1.1. Last-value predictor

The last-value predictor [12,13] predicts the destination value of an individual instruction based on the last previously seen value it has generated (or computed). The predictor is organized as a table (e.g. cache table – see Fig. 1), 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.

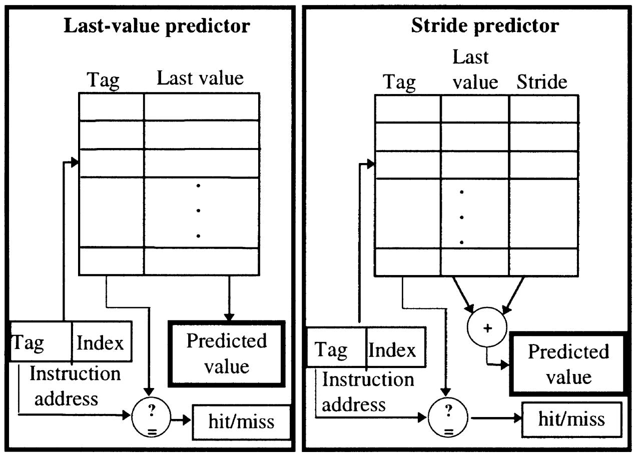


Fig. 1. The ‘last value’ and the ‘stride’ predictors.

2.1.2. Stride predictor

The stride predictor [14,15] 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 an individual instruction (Fig. 1). The stride field value is always determined upon the subtraction of two recent consecutive destination values.

2.2. Classification of value predictability

Previous works proposed classification of value predictability mainly to distinguish between instructions which are likely to be correctly predicted and those which tend to be incorrectly predicted by the predictor. A possible method of classifying instructions is to use a set of saturated counters [12,13]. An individual saturated counter (illustrated by Fig. 2) is assigned to each entry in the prediction table. At each occurrence of a successful or unsuccessful prediction the corresponding counter is incremented or decremented, respectively. According to the present state of the saturated counter, the processor can decide whether to take the suggested prediction or to avoid it.

In Section 5 we compare the effectiveness of this hardware-based classification mechanism vs the proposed mechanism.

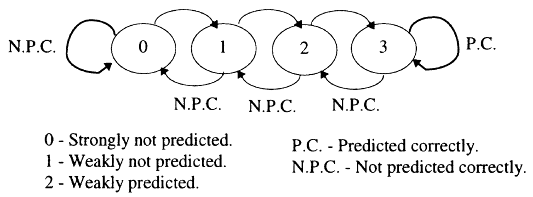

P.C.
P.C.
P.C.
N.P.C.
0
1
2
3
P.C.
N.P.C.
N.P.C.
N.P.C.
0 - Strongly not predicted.
1 - Weakly not predicted.
2 - Weakly predicted.
3 - Strongly predicted.
P.C. - Predicted correctly.
N.P.C. - Not predicted correctly.

Fig. 2. A 2-bit saturated counter for classification of value predictability.

2.3. Value prediction accuracy

The benefit of using value prediction is significantly dependent on the accuracy that the value predictor can accomplish. The previous works in this field [14,15,12,13] provided substantial evidence to support the observation that outcome values in programs tend to be predictable (value predictability). The prediction accuracy measurements of the predictors that were described in Section 2.1 on Spec-95 benchmarks are summarized in Table 1. Note that in the floating point benchmarks (Spec-fp95) the prediction accuracy was measured in each benchmark for two execution phases: initialization (when the program reads its input data) and computation (when the actual computation is made). Broad study and analysis of these measurements can be found in [14,15].

2.4. Distribution of value prediction accuracy

Our previous studies [14,15] revealed that the tendency of instruction to be value-predictable does not spread uniformly among the instructions in a program (we only refer to those instructions that assign outcome value to a destination register). Approximately 30% of the instructions are very likely to be correctly predicted with prediction accuracy greater than 90%. In addition, approximately 40% of the instructions are very unlikely to be correctly predicted (with a prediction accuracy less than 10%). This observation is illustrated by Fig. 3[1,2] for both integer and floating point benchmarks. The importance of this observation and its implication are discussed in Section 3.1.

2.5. Distribution of non-zero strides

In our previous works [14,15] we examined how efficiently the stride predictor takes advantage of the additional ‘stride’ field (in its prediction table) beyond the last-value predictor that only maintains a single field (per entry) of the ‘last value’. We considered the stride fields to be utilized efficiently only when the predictor accomplishes a correct value prediction and the stride field is not equal to zero (non-zero stride). In order to grade this efficiency we used a measure that we term ‘stride efficiency ratio’ (measured in percentages). The stride efficiency ratio is the ratio of successful non-zero stride-based value predictions to overall successful predictions. Our measurements indicated that in the integer benchmarks the stride efficiency ratio is approximately 16%, and in the floating point benchmarks it varies from 12% in the initialization phase to 43% in the computation phase. We also examined the stride efficiency ratio of each instruction in the program that was allocated to the

[1] The initialization phase of the floating-point benchmarks is denoted by #1 and the computation phase by #2.

[2] *gcc*1 and *gcc*2 denote the measurements when the benchmark was run with different input files (the same for *perl*1 and *perl*2).

Table 1
Value prediction accuracy measurements

Prediction accuracy [%]	Integer loads		ALU instructions		FP loads		FP computation instructions	
Benchmark	S	L	S	L	S	L	S	L
Spec-int95	48	50	61	53	–	–	–	–
Spec-fp95 Init. phase	70	66	52	47	–	–	–	–
Spec-fp95 Comp. phase	63	37	96	23	46	44	29	28

S — Stride predictor.
L — Last-value predictor.

prediction table. We observed that most of these instructions could be divided into two major subsets: a small subset of instructions which always exhibits a relatively high stride efficiency ratio and a large subset of instructions which always tend to reuse their last value (with a very low stride efficiency ratio). Fig. 4 draws histograms of our experiments and illustrates how instructions in the program are scattered according to their stride efficiency ratio.

3. The proposed method

Profiling techniques are broadly being employed in various compilation areas to enhance the optimization of programs. The principle of this technique is to study the behavior of the program based on one set of train inputs and to provide the gathered information to the compiler. The effectiveness of this technique relies on the assumption that the behavioral characteristics of the program remain consistent with other program's runs as well. In the first subsection we present how the previous knowledge in the area of value prediction motivated us towards our new approach. In the second subsection we present our new method and its main principles.

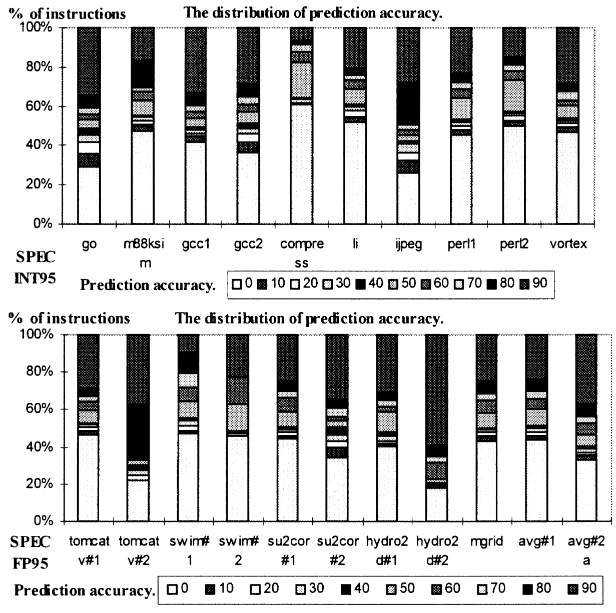


Fig. 3. The spread of instructions according to their value prediction accuracy.

3.1. Motivation

The consequences of the previous results described in Section 2 are very significant, since they establish the basis and motivation for our current work with respect to the following aspects:

1. Our measurements in Section 2.4 indicated that the tendency of instructions to be value predictable does not spread uniformly among the instructions in the program. In fact, most programs exhibit two sets of instructions — highly value-predictable instructions and highly unpredictable ones. This observation established the basis for employing classification mechanisms.
2. Previous experiments [14,15] have also provided preliminary indication that different input files do not dramatically affect the prediction accuracy of several examined benchmarks. If this observation is found to be common enough, then it may have a tremendous significance when considering the involvement of program profiling. It may imply that the profiling information which is collected in previous runs of the program (running the application with training input files) can be correlated to the true situation where the program runs with its real input files (provided by the user). This property is extensively examined in this paper.
3. We have also indicated that the set of value-predictable instructions in the program is partitioned into two subsets: a small subset of instructions that exhibit stride value predictability (predictable only by the stride predictor) and a large subset of instructions which tend to

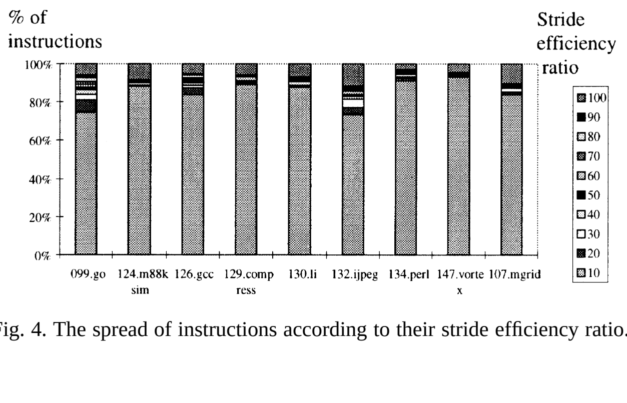


Fig. 4. The spread of instructions according to their stride efficiency ratio.

reuse their last value (predictable by both predictors). Our previous works [14,15] showed that although the first subset is relatively smaller than the second subset, it appears frequently enough to significantly affect the extractable ILP. On one hand, if we only use the last-value predictor then it cannot exploit the predictability of the first subset of instructions. On the other hand, if we only use the stride predictor, then in a significant number of entries in the prediction table, the extra stride field is useless because it is assigned to instructions that tend to reuse their most recently produced value (zero strides). This observation motivates us to employ a hybrid predictor that combines both the stride prediction table and the last-value prediction table. Such a hybrid predictor can use the last-value prediction table to predict the values of instructions which tend to exhibit last-value predictability and a stride prediction table for those whose values exhibit stride value predictability. Since the subset of instruction that exhibit stride value predictability is smaller than those that exhibit last-value predictability, we may consider using a relatively small stride prediction table and a larger last-value prediction table. Such a technique allows us utilize the extra stride field more efficiently.

3.2. A classification based on program profiling and compiler support

The method introduced by this work combined both program profiling and compiler support to perform the classification of instructions according to their tendency to be value predictable. All familiar previous works performed the classification by using a hardware mechanism that studies the tendency of instructions to be predictable at run-time [14,15,12,13]. Such a mechanism was capable of eliminating a significant part of the mispredictions. However, since the classification was performed dynamically, it could not allocate in advance the highly value predictable instructions in the prediction table. As a result unpredictable instructions could have uselessly occupied entries in the prediction table and evacuated useful instructions. In addition, the hardware-based classification was incapable of detecting the value predictability patterns of the predictable instructions, i.e. once an instruction was classified as predictable, it could determine whether it tended to exhibit either last-value predictability or stride predictability. The alternative classification technique, proposed in this paper, has two tasks:

1. Identify the highly predictable instructions in advance.
2. Once an instruction is classified as predictable, it determines its predictability type (either last-value predictability or stride predictability).

Our new method consists of three basic phases (Fig. 5). In the first phase the program is ordinarily compiled (the compiler can use all the available and known optimization methods) and the code is generated. In the second phase the profile image of the program is collected. The profile image describes the prediction accuracy of each instruction in the program (we only refer to instructions which write a computed value to a destination register). In order to collect this information, the program can be run on a simulation environment (e.g. the SHADE simulator – see [16]) where the simulator can emulate the operation of the value predictor and measure for each instruction its prediction accuracy. If the simulation emulates the operation of the stride predictor it can also measure the stride efficiency ratio of each instruction. Such profiling information could not only indicate which instructions tend to be value-predictable or not, but also which ones exhibit value predictability patterns in form of ‘strides’ or ‘last-value’. The output of the profile phase can be a file that is organized as a table. Each entry is associated with an individual instruction and consists of three fields: the instruction’s address; its prediction accuracy; and its stride efficiency ratio. Note that in the profile phase the program can be run either single or multiple times, where in each run the program is driven by different input parameters and files.

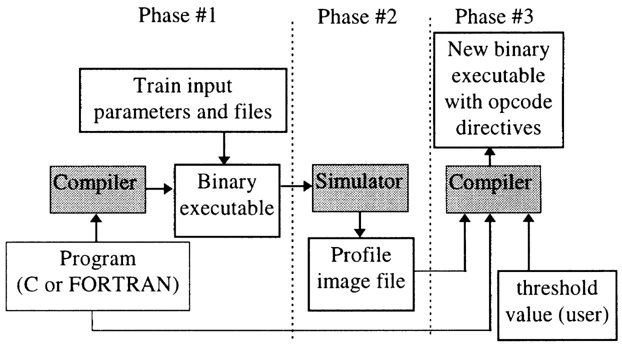


Fig. 5. The three phases of the proposed classification methodology.

In the final phase the compiler only inserts directives in the opcode of instructions. It does not perform instruction scheduling or any form of code movement with respect to the code that was generated in the first phase. The inserted directives act as hints about the value predictability of instructions that are supplied to the hardware. Note, that we consider such use of opcode directives as feasible, since recent processors, such as the PowerPC 601, made branch predictions based on opcode directives too ([17]). Our compiler employs two kinds of directives: the ‘stride’ and the ‘last-value’. The ‘stride’ directive indicates that the instruction tends to exhibit stride patterns, and the ‘last-value’ directive indicates that the instruction is likely to repeat its recently generated outcome value. By default, if none of these directives are inserted in the opcode, the instruction is not recommended to be value predicted. The compiler can determine which instructions are inserted with the special directives according to the profile image file and a threshold value supplied by the user. This value determines the prediction accuracy threshold of instructions to be tagged with a directive as value-predictable. For instance, if the user sets the threshold value to 90%, all the instructions in the profile image file that had a prediction accuracy less than 90% are not inserted with directives (marked as unlikely to be correctly predicted) and all those with prediction accuracy greater than or equal to 90% are marked as predictable. When an instruction is marked as value-predictable, the type of the directive (either ‘stride’ or ‘last-value’) still needs to be determined. This can be done by examining the stride efficiency ratio that is provided in the profile image file. A possible heuristic that the compiler can employ is: If the stride efficiency ratio is greater than 50% it indicates that the majority of the correct predictions were non-zero strides and therefore the instruction should be marked as ‘stride’; otherwise it is tagged with the ‘last-value’ directive. Another way to determine the directive type is to ask the user to supply the threshold value for the stride efficiency ratio.

Once this process is completed, the previous hardware-based classification mechanism (the set of saturated counters) becomes unnecessary. Moreover, we can use a hybrid value predictor that consists of two prediction tables: the ‘last-value’ and the ‘stride’ prediction tables (Section 2.2). A candidate instruction for value prediction can be allocated to one of these tables according to its opcode directive type. These new capabilities allow us to exploit both value predictability patterns (stride and last-value) and utilize the prediction tables more efficiently. In addition, they allow us to detect in advance the highly predictable instructions, and thus we could reduce the probability that unlikely to be correctly predicted instructions evacuate useful instructions from the prediction table.

In order to clarify the principles of our new technique we are assisted by the following sample C program segment:

for $(x=0;\ x<100\,000;\ x++)$ $\mathbf{A}[x]=\mathbf{B}[x]+\mathbf{C}[x]$;

The program sums the values of two vectors, **B** and **C**, into vector **A**.

In the first phase, the compilation of the program with the *gcc* 2.7.2 compiler (using the ‘-O2’ optimization) yields the following assembly code (for a Sun-Sparc machine on SunOS 4.1.3):

```
(1) 1d [%i4 +  %g0], %17        //Load B[i]
(2) 1d [%15 +  %g0], %i0        //Load C[j]
(3) add %i5, 0x4, %i5           //Increment index j
(4) add %17, %i0, %17           //A[k] = B[i] + C[j]
(5) st %17, [%i3 +  %g0]        //Store A[k]
(6) cmp %i5, %i2                //Compare index j
(7) add %i4, 0x4, %i4           //Increment index i
(8) bcs 0xfffffff9              //Branch to (1)
(9) add %i3, 0x4, %i3           //Increment index k (in the delay slot)
```

In the second phase we collect the profile image of the program. A sample output file of this process is illustrated by Table 2. It can be seen that this table includes all the instructions in the program that assign values to a destination register (load and add instructions). For simplicity, we only refer to value predictions where the destination operand is a register. However our methodology is not limited by any means to being applied when the destination operand is a condition code, program counter, a memory storage location or a special register.

Table 2
A sample profile image output

Instruction address	Prediction accuracy (%)	Stride efficiency ratio (%)
1	10	2
2	40	1
3	99.99	99.99
4	20	1
7	99.99	99.99
9	99.99	99.99

In this example the profile image indicates that the prediction accuracy of the instructions that compute the index of the loop was 99.99% and their stride efficiency ratio was 99.99%. Such an observation is reasonable since the destination value of these instructions can be correctly predicted by the stride predictor. The other instructions in our example accomplished relatively low prediction accuracy and stride efficiency ratio. If the user determines the prediction accuracy threshold to be 90%, then in the third phase the compiler would modify the opcodes of the add operations in addresses 3, 7 and 9 and insert into these opcodes the ‘stride’ directive. All other instructions in the program are unaffected.

4. Examining the potential of profiling through quantitative measurements

This section is dedicated to examining the basic question: can program profiling supply the value prediction hardware mechanisms with accurate information about the tendency of instructions to be value-redictable? In order to answer this question, we need to explore whether programs exhibit similar patterns when they are being run with different input parameters. If under different runs of the programs these patterns are correlated, this confirms our claim that profiling can supply accurate information.

For our experiments we use different programs, chosen from the Spec95 benchmarks (Table 3), with different input parameters and input files. In order to collect the profile image we traced the execution of the programs by the SHADE simulator [16] on Sun-Spare processor. In the first phase, all benchmarks were compiled with the *gcc* 2.2.2 compiler with all available optimizations.

For each run of a program we create a *profile image* containing statistical information that was collected during run-time. The profile image of each run can be regarded as a vector $\bar{\mathbf{V}}$, where each of its coordinates represents the value

Table 3
Spec95 benchmarks

Spec 95 benchmarks Benchmarks	Description
go	Game playing
m88ksim	A simulator for the 88100 processor
gcc	A C compiler based on GNU C 2.5.3
compress 95	Data compression program using adaptive Lempel–Ziv coding
li	Lisp interpreter
ijpeg	JPEG encoder
perl	Anagram search program
vortex	A single-user object-oriented data-base transaction benchmark
mgrid	Multi-grid solver in computing a three dimensional potential field

prediction accuracy of an individual instruction (the dimension of the vector is determined by the number of different instructions that were traced during the experiment). As a result of running the same program n times, each time with different input parameters and input files, we obtain a set of n vectors $\mathbf{V} = \{\bar{\mathbf{V}}_1,\bar{\mathbf{V}}_2,\ldots,\bar{\mathbf{V}}_n\}$ where the vector $\bar{\mathbf{V}}_j = (v_{j,1},v_{j,2},\ldots,v_{j,k})$ represents the profile image of run j. Note that in each run we may collect statistical information of instructions which may not appear in other runs. Therefore, we only consider the instructions that appear in all the different runs of the program. Instructions which only appear in certain runs are omitted from the vectors (our measurements indicate that the number of these instructions is relatively small). By omitting these instructions we can organize the components of each vector such that corresponding coordinates would refer to the prediction accuracy of same instruction, i.e. the set of coordinates $\{v_{1,l},v_{2,l},\ldots,v_{n,l}\}$ refers to the prediction accuracy of the same instruction i under the different runs of the program.

Our first goal is to evaluate the correlation between the tendencies of instructions to be value-predictable under different runs of a program with different input files and parameters. Therefore, once the set of vectors $\mathbf{V} = \{\bar{\mathbf{V}}_1,\bar{\mathbf{V}}_2,\ldots,\bar{\mathbf{V}}_n\}$ is collected, we need to define a certain metric for measuring the similarity (or the correlation) between them. We choose to use two metrics to measure the resemblance between the vectors. We term the first metric the maximum-distance metric. This metric is a vector $\mathbf{M(V)}_{\max} = (m_1,m_2,\ldots,m_k)$ whose coordinates are calculated as illustrated by Eq. (1), the $\mathbf{M}_{\max}$ metric:

$$m_i = \max\{|v_{1,i} - v_{2,i}|, |v_{1,i} - v_{3,i}|, \ldots, |v_{1,i} - v_{n,i}|, \quad (1)$$

$$|v_{2,i} - v_{3,i}|, |v_{2,i} - v_{4,i}|, \ldots |v_{2,i} - v_{n,i}|,$$

$$\ldots$$

$$|v_{(n-1),i} - v_{n,i}|\}$$

Each coordinate of $\mathbf{M(V)}_{\max}$ is equal to the maximum distance between the corresponding coordinates of each pair of vectors from the set $\mathbf{V} = \{\bar{\mathbf{V}}_1,\bar{\mathbf{V}}_2,\ldots,\bar{\mathbf{V}}_n\}$. The second metric that we use is less strict. We term this metric the average-distance metric. This metric is also a vector, $\mathbf{M(V)}_{\text{average}} = (m_1,m_2,\ldots,m_k)$, where each of its coordinates is equal to the arithmetic-average distance between the corresponding coordinates of each pair of vectors from the set $\mathbf{V} = \{\bar{\mathbf{V}}_1,\bar{\mathbf{V}}_2,\ldots,\bar{\mathbf{V}}_n\}$ (Eq. (2), the $\mathbf{M}_{\text{average}}$ metric).

$$m_i = \text{average}_{\text{arith.}}\{|v_{1,i} - v_{2,i}|, |v_{1,i} - v_{3,i}|, \ldots, |v_{1,i} - v_{n,i}|, \quad (2)$$

$$|v_{2,i} - v_{3,i}|, |v_{2,i} - v_{4,i}|, \ldots |v_{2,i} - v_{n,i}|,$$

$$\ldots$$

$$|v_{(n-1),i} - v_{n,i}|\}$$

Obviously, one can use other metrics in order to measure the similarity between the vectors, e.g. instead of taking the arithmetic average we could take the geometric average. However, we think that these metrics sufficiently satisfy our needs.

Once our metrics are calculated out of the profile image, we can illustrate the distribution of its coordinates by building a histogram. For instance, we can count the number of $\mathbf{M(V)}_{\max}$ coordinates whose values are in each of the intervals: [0,10], [10,20], [30,40], …, [90,100]. If we observe that most of the coordinates are scattered in the lower intervals, we can conclude that our measurements are similar and that the correlation between the vectors is very high. Figs. 6 and 7 illustrate such histograms for our two metrics $\mathbf{M(V)}_{\max}$ and $\mathbf{M(V)}_{\text{average}}$ respectively.

In these histograms we clearly observe that in all the benchmarks most of the coordinates are spread across the

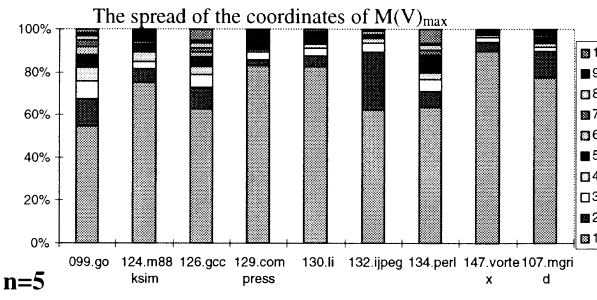


Fig. 6. The spread of $M(V)_{\max}$.

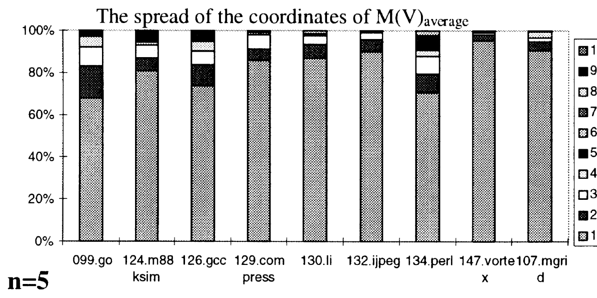


Fig. 7. The spread of $M(V)_{\text{average}}$.

lower intervals. This observation provides the first substantial evidence that confirms one of our main claims – the tendency of instructions in a program to be value predictable is independent of the program's input parameters and data. In addition it confirms our claim that program profiling can supply accurate information about the tendency of instructions to be value predictable.

As we have previously indicated, the profile image of the program that is provided to the compiler can be better tuned so that it can indicate which instructions tend to repeat their recently generated value and which tend to exhibit patterns of strides. In order to evaluate the potential of such classification we need to explore whether the set of instructions whose outcome values exhibit tendency of strides is common to the different runs of the program. This can be done by examining the stride efficiency ratio of each instruction in the program from the profile image file. In this case, we obtained from the profile image file a vector $\bar{S}$, where each of its coordinates represents the stride efficiency ratio of an individual instruction. When we run the same program n times (each time with different input parameters and input files) we obtain a set of n vectors $\mathbf{S} = \{\bar{\mathbf{S}}_1,\bar{\mathbf{S}}_2,\ldots,\bar{\mathbf{S}}_n\}$ where the vector $\bar{\mathbf{S}}_j = (s_{j,1},s_{j,2},\ldots,s_{j,k})$ represents the profile image of run j. Once these vectors are collected we can use one of the previous metrics either the maximum-distance or the average-distance in order to measure the resemblance between the set of vectors $\mathbf{S} = \{\bar{\mathbf{S}}_1,\bar{\mathbf{S}}_2,\ldots,\bar{\mathbf{S}}_n\}$. For simplicity we have chosen this time only the average-distance metric to demonstrate the resemblance between the vectors. Once this metric is calculated out of the profile information, we obtain a vector $\mathbf{M(S)}_{\text{average}}$. Similar to our previous analysis, we draw a histogram to illustrate the distribution of the coordinates of $\mathbf{M(S)}_{\text{average}}$ (Fig. 8).

Again in this histogram we clearly observe that in all the benchmarks most of the coordinates are spread across the lower intervals. This observation provides evidence that confirms our claim that the set of instructions in the program that tend to exhibit value predictability patterns in form of stride is independent of the program's input parameters and data. Therefore profiling can accurately detect these instructions and provide this information to the compiler.

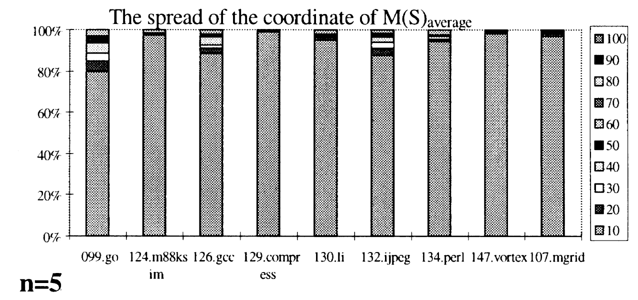


Fig. 8. The spread of $M(S)_{\text{average}}$.

5. The effect of the profiling-based classification on value-prediction performance

In this section we focus on four main aspects:

1. The classification accuracy of our new mechanism.
2. Its potential to better utilize the prediction table entries.
3. Its effect on the extractable ILP when using value prediction.
4. The potential benefits of using a hybrid value predictor. In the presented performance evaluation we also compare the new technique versus the hardware only classification mechanism (saturated counters).

5.1. The classification accuracy

The quality of the classification process can be represented by the classification accuracy, i.e. the fraction of correct classifications out of overall prediction attempts. We measured the classification accuracy of our new mechanism and compared it to the hardware-based mechanism. The classification accuracy was measured for the incorrect and correct predictions separately (using the 'stride' predictor), as illustrated by Figs. 9 and 10, respectively. Note that these two cases represent a fundamental tradeoff in the classification operation since improving the classification accuracy of the incorrect predictions can reduce the classification accuracy of the correct predictions and vice versa.

Our measurements currently isolate the effect of the prediction table size since in this subsection we wish to focus only on the pure potential of the proposed technique to successfully classify either correct or incorrect value predictions. Hence, we assume that each of the classification mechanisms has an infinite prediction table (a stride predictor), and that the hardware-based classification mechanism also maintains an infinite set of saturated counters. The effect of the finite prediction table is presented in the next subsection.

Our observations indicate that in most cases the profiling-based classification better eliminates mispredictions in comparison with the saturated counters. When the threshold value of our classification mechanism is reduced, the classification accuracy of mispredictions decreases as well, since the classification becomes less strict. Only

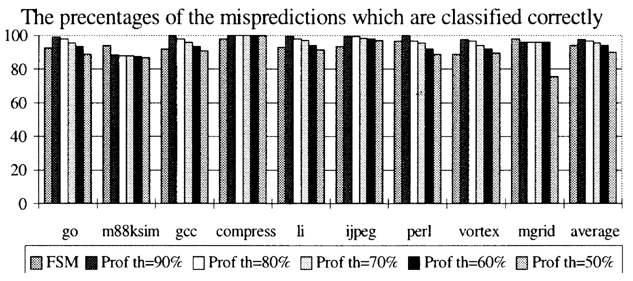


Fig. 9. The percentages of the mispredictions which are classified correctly.

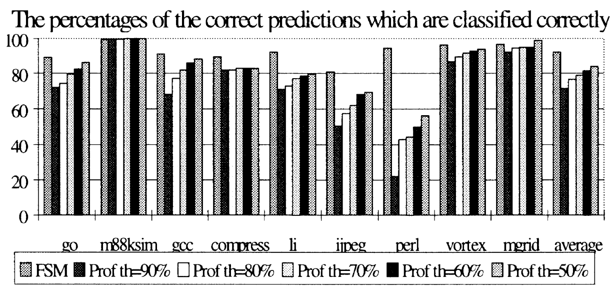


Fig. 10. The percentages of the correct predictions which are classified correctly.

when the threshold value is less than 60% does the hardware-based classification gain better classification accuracy for the mispredictions than our proposed mechanism (on the average).

Decreasingly the threshold value of our classification mechanisms improves the detection of the correct predictions at the expense of the detection of mispredictions. Fig. 10 indicates that in most cases the hardware-based classification (denoted as 'FSM') achieves slightly better classification accuracy of correct predictions in comparison with the profiling-based classifications (denoted as 'Prof.'). Notice that this observation *does not imply at all* that the hardware-based classification outperforms the profiling-based classification, because the effect of the table size was not included in these measurements.

5.2. The effect on the prediction table utilizaton

We have already indicated that when using the hardware-based classification mechanism, unpredictable instructions may uselessly occupy entries in the prediction table and can purge out highly predictable instructions. As a result, the efficiency of the predictor can be decreased, as well as the utilization of the table and the prediction accuracy. Our classification mechanism can overcome this drawback, since it is capable of detecting the highly predictable instructions in advance, and hence decreasing the pollution of the table caused by unpredictable instructions.

Table 4 shows the fraction (in percentages) of potential candidates which are allowed to be allocated in the table by our classification mechanism (with various threshold values) out of these allocated by the saturated counters. It can be observed that even with a threshold value of 50%, the new mechanism can reduce the number of potential candidates by nearly 50%. Moreover, this number can be reduced even more significantly when the threshold is tightened, e.g. a threshold value of 90% reduces the number of potential candidates by more than 75%. This unique capability of our mechanism allows us to use a smaller prediction table and utilize it more efficiently.

Our next measurements present the total number of correct predictions and the total number of mispredictions that each of the classification methods yields under various prediction table sizes. We do not present the prediction accuracy gained by the predictor in this comparison, since it may mislead us in our conclusions. For example, suppose that we have two different mechanisms, one that gains 90% prediction accuracy and another than gains 50% only. The first mechanism deals with 20% of the candidate instructions while the second one deals with 40% of the candidates. On one hand, the first mechanism is indeed more accurate that the second one. However, on the other hand, the second mechanism has more candidates and as result it may yield more correct predictions. Therefore, in our next experiments we compare the increase or decrease in the absolute number of correct predictions and mispredictions gained by each of the mechanisms.

The predictor, used in our experiments, is the 'stride predictor', which was organized as a 2-way set-associative table. In addition, in the case of the profiling-based classification, instructions were allowed to be allocated to the prediction table only when they were tagged with either the 'last-value' or the 'stride' directives. The results for each benchmark that was used are summarized in Appendix A. It can be observed that the profiling threshold plays the main role in the tuning of our new mechanism. By choosing the right threshold, we can tune our mechanism in such way that it outperforms the hardware-based classification mechanism in most benchmarks. In the benchmark *go* we can accomplish both a significant increase in the number of correct predictions and a reduction in the number of mispredictions. When the table size is 0.25, 0.5 or 1 K entries we can use a threshold value in the range of 50%–90% so that our mechanism accomplishes both more correct predictions and less incorrect predictions than the hardware-only mechanism. For instance, when using a 0.25 K-entry table and taking a threshold value of 60% the predictor yields an increase of 35% in the total number of correct predictions and a decrease of 22% in the number of mispredictions (relative to the hardware-based classification). When the table size is 2 or 4 K entries the range of threshold values that yields both an increase in the number of correct predictions and a decrease in the number of mispredictions (relative to the hardware-based classification) is 50%–70%. Increasing the prediction table size reduces the pressure on the prediction table and thus we can afford allocation of more instructions by using a less tightened threshold value. Our observations also indicate that the improvement achieved by our classification mechanism is more significant when the prediction table is relatively small

Table 4
The fraction of potential candidates to be allocated relative to those in the hardware-based classification

Profiling threshold (%)	90	80	70	60	50
The fraction of potential candidates to be allocated relative to those in the saturated counters (%)	24	32	35	39	47

since this is the case when the pressure on the prediction table is crucial. For instance, when taking a threshold value of 60% the relative increase in the number of correct predictions is 35%, 27%, 23%, 13% and 3.5% when the prediction table size is 0.25, 0.5, 1, 2 and 4 K entries, respectively. This phenomenon is clearly observed in the benchmark *go* because the working set of instructions in this benchmark is very large. Moreover, when decreasing the threshold value from 90% to 10% we observe that the total number of correct predictions and the total number of mispredictions increase since the classification becomes less strict. However, when changing the threshold value from 10% to 0% we also observe that the number of correct predictions is slightly decreased. This observation is indeed valid since this modification increases the pressure on the prediction table significantly in such a way that it cannot yield more correct predictions although the classification is less strict. This phenomenon is also observed in other benchmarks that have large working sets of instructions, like *gcc* and *vortex*.

Unlike the benchmark *go*, *m88ksin* and *compress* have relatively very small working sets of instructions, and thus we can barely observe any change in the total number of correct predictions and the number of mispredictions when we change the prediction table size. Since their working sets are very small they can much less exploit the benefits of our classification mechanism and as a results we cannot even find a threshold value that yields both an increase in the total number of correct predictions and a decrease in the number of mispredictions relative to the hardware-based classification. It can also be observed that when using a threshold value in the range 0%–70% in *m88ksim* there is a significant increase in the number of mispredictions gained by our classification mechanism relative to the hardware-based classification. This increase is not expected to significantly affect the extractable ILP, since the prediction accuracy of this benchmark is already very high. In *compress*, however, we observe a significant increase in the number of mispredictions only when the threshold value is 0%. This observation is indeed consistent with our previous measurements that were presented in Fig. 3. We have observed that in this benchmark the set of unpredictable instructions is significantly bigger than the set of predictable ones. Therefore, when setting the threshold value to 0%, we expose the predictor to the large set of unpredictable instructions that eventually increases the yield of mispredictions.

In the benchmark *gcc*, we reveal very similar behavior to that observed in *go*, since it also maintains a very large working set of instructions. When the prediction table size is 0.25–2 K entries we can use a threshold value in the range of 70%–90% and gain both an increase in the number of correct predictions, and a reduction in the number of mispredictions. Moreover, when the prediction table size is 4 K entries the range of threshold values which yields both an increase in the correct predictions and a decrease in the incorrect prediction is 60%–70%. In *gcc*, as in *go*, we also observe that the increase in the prediction table size reduces the gain of our classification mechanism relative to the saturated counter because of the reduction in the pressure on the prediction table. In addition, unlike *m88ksim* which is barely affected when increasing the prediction table size, *gcc* better takes advantage of this increase since it maintains much larger working sets of instructions.

In the benchmark *li* our classification mechanism can yield both an increase in the total number of correct predictions and a decrease in the number of mispredictions (relative to the hardware-based classification) only when the prediction table size is small. This can be satisfied only when the prediction table size is 0.25 or 0.5 K entries by taking a range of threshold values of 60%–90% and 60%, respectively. For instance, when using a 0.25 K-entry table and taking a threshold value of 60%, our classification mechanism achieves an increase of 13% in the number of correct predictions and a decrease of 7% in the number of mispredictions relative to the hardware-based classification. When using the same threshold value and a table of 0.5 K-entry our classification mechanism achieves an increase of 2% in the number of correct predictions and a decrease of 26% in the number of mispredictions. When taking bigger prediction tables, the pressure on the table is reduced so that we cannot find a threshold value that yields both an increase in the number of correct predictions and a decrease in the number of mispredictions relative to the hardware-based classification.

In *ijpeg* our classification mechanism cannot yield both an increase in the number of correct predictions and a decrease in the number of mispredictions relative to the hardware-based classification. Even when taking a 0.25 K-entry prediction table it requires a relatively very low threshold value (of 40%) in order to yield approximately the same number of correct predictions as gained by the hardware-based classification. However, it still cannot yield less mispredictions in this case than the hardware-based classification.

In *perl* our profiling-based classification can significantly outperform the hardware-based classification. When the table size is 0.25–0.5 K entries we can achieve both an increase in the number of correct predictions and a decrease in the number of mispredictions (relative to the hardware-based classification) by taking a threshold value in the range of 70%–90%. For example, by using a threshold value of 70% we can achieve an increase of 38% in the number of correct predictions and a decrease of 21% in the number of mispredictions when the table size is 0.25 K-entry. When taking a 0.5 K-entry prediction table and the same threshold value the increase in the number of correct predictions becomes 32% and the decrease in the number of mispredictions is approximately 23% relative to the hardware-based classification. For a 1 or 2 K-entry table the range of threshold values that achieves an improvement in the number of correct predictions and a decrease in mispredictions is 60%–90% and 60%–70%, respectively. When the prediction table size is 4 K entries, there is no threshold value that can yield both an increase in the correct prediction and a decrease in the mispredictions.

The benchmark *vortex* also maintains a large working-set of instructions like *go* and *gcc*. When we use a relatively small prediction table (0.25, 0.5 or 1 K-entries) the pressure on the prediction table (in terms of candidate instructions for allocation) becomes very significant. In this case the range of threshold values that yields both an increase in the correct predictions and a decrease in mispredictions relative to the saturated counters is 80%–90%. An 80% threshold value yields an increase of 36%, 22% and 9% in the number of correct predictions when the table size is 0.25, 0.5 and 1 K-entries, respectively. At the same time, such a threshold value yields a 50%–60% decrease in the number of mispredictions relative to the hardware-based classification. When the table size is 2 or 4 K entries, we cannot find a threshold value that yields both an increase in the number of correct predictions and a decrease in mispredictions.

In *mgrid* the prediction table size barely affects the number of correct predictions which each of the mechanisms can gain. In addition, the profiling-based classification cannot yield both an increase in the number of correct predictions and a decrease in the number of mispredictions relative to the hardware-based classification. Even when the prediction table size is relatively small (0.25 K-entry) we need a relatively very small threshold value (30%) in order to gain any increase in the number of correct predictions. Such a threshold value, however, yields a very significant increase in the number of mispredictions.

5.3. The effect of the classification on the extractable ILP

In this section we examine the ILP that can be extracted by value prediction under different classification mechanisms. Our experiments consider an abstract machine with a finite instruction window ([14]). We have chosen this experimental model since the effect 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 limited by the dataflow graph of the program and its instruction window size. In order to reach the dataflow graph boundaries, such a machine should employ: (1) an unlimited number of resources (execution units etc.) to avoid structural conflicts; (2) an unlimited number of registers to avoid false dependencies; (3) perfect (either static or dynamic) branch prediction mechanisms to avoid control dependencies; and (4) its instruction fetch bandwidth should be sufficient, since the execution order of instructions in a program may not correspond to the order in which they are stored in memory. 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 the related phenomena without being affected by the limitations of individual machines.

The instruction window size that we use for the simulation of our abstract machine is 40. It may seem that using a bigger instruction window and increasing the machine's bandwidth (fetch, decode, issue and execution bandwidth) would expose the execution core to a greater number of candidate instructions and hence may reduce the benefit of value prediction. Our previous study ([18]) indicates, however, that the effect of the instruction window size may not be so straightforward and therefore we keep this issue out of the scope of this paper.

The value predictor, used in our experiments, is the 'stride predictor', which was organized as a 2-way set associative table. In the case of the profiling-based classification, instructions were allowed to be allocated to the prediction table only when they were tagged with either the 'last-value' or the 'stride' directives. Moreover, in case of value-misprediction, the penalty in our machine is 1 clock cycle.

Our experimental results, summarized in Appendix B, present the increase in ILP gained by using value prediction under different classification mechanisms relative to the case when value prediction is not used. In most benchmarks we observe that our mechanism can be tuned, by choosing the right threshold value, in such a way that it can achieve better results than those gained by the saturated counters. In the benchmark *go* the threshold value that yields the best results for our classification mechanism is 50% when the prediction table size is 0.25–2 K entries and 40% when using a 4 K-entry prediction table. In all these cases, our classification mechanism (with this threshold value) outperforms the hardware-based classification. As the prediction table size is increased the gap between the performance of the profiling-based classification and the saturated counters decreases since the pressure on the prediction table is reduced. It can be observed, for instance, that when using our classification mechanism with a 0.25 K-entry table only and taking a threshold value of 50% we can achieve better ILP than using the saturated counters with a 1 K-entry table. In addition, it can also be seen that the threshold value that yields the best ILP may not necessarily correspond to the range of threshold values which yield both an increase of correct predictions and a decrease in mispredictions (as was found in the previous subsection). The reason for this observation in such cases is that the contribution of the correct predictions that our classification mechanism exposes is more significant then the few mispredictions that it failed to avoid relative to the saturated counters. For instance, when using a 0.25 K-entry table and threshold value of 50%, our classification mechanism yields approximately 40% more correct predictions than the saturated counters, while increasing the number of mispredictions in 5% only. However, when using a threshold value of 0%, value prediction performs worse than not using any value prediction. In this case, the number of mispredictions is more significant than the number of correct predictions and due to the penalty that we pay in each case of misprediction the performance degrades.

In *m88ksim* the ILP is barely affected when increasing the prediction table size due to its relatively small working-set of instructions. In all the predictions table sizes that were used the hardware-based classification slightly outperforms the profiling-based classification. In addition, the profiling threshold that achieves the best ILP for our classification mechanism is 10%.

In *gcc* our classification mechanism can outperform the hardware-based classification in all the prediction table sizes that were used in our experiments. When the prediction table size is 0.25, 0.5 and 2 K-entries the best threshold value is 40% and for 1 or 2 K-entry the best threshold is 30%. It can be observed, for instance, that if we use a 0.25 K-entry prediction table and take a threshold value of 40% our classification mechanism achieves even better ILP than using a 1 K-entry table with the saturated counters. It can also be observed that as we increase the prediction table size the gap between the performance of the two classification mechanisms decreases since the pressure on the prediction table is reduced.

The ILP in *compress*, like in *m88ksim*, is also hardly affected when we modify the prediction table size because of its relatively small working set. We observe that the threshold value that yields the best performance for our classification is 30%. In this case it also slightly outperforms the saturated counters. In addition, when the threshold value is 0%, the number of mispredictions is so significant that it does worse than not using value prediction.

In *li* our classification mechanism can achieve better ILP than the saturated counters only when the prediction table size is 0.25 or 0.5 K entries (the best ILP is obtained when the threshold value is 50% and 40%, respectively). When increasing the table size, the pressure on the table decreases as well and the saturated counters gain slightly better ILP than our classification mechanism. It can also be observed that when using the profiling-based classification with a 0.25 K-entry table and a threshold value of 50% it achieves comparable ILP to the saturated counters when using a 0.5 K-entry. This means that our mechanism can use 50% less resources and gain similar performance.

The ILP gained in *ijpeg* is hardly affected by the prediction table size. Even when using relatively small prediction tables the saturated counters slightly outperforms the profiling-based classifications. Moreover, we also observe that in all prediction table sizes the threshold value that yields the best ILP for the profiling-based classification is 60%.

In *perl* our classification technique significantly achieves better ILP than the hardware-based classification. The threshold value that yields the best ILP for our classification technique is 30% (for all the prediction table sizes). It can also be observed that the profiling-based classification (with a threshold value of 30%) can achieve a similar ILP increase relative to the saturated counters while using a smaller by half value prediction table.

In *vortex* our classification mechanism can outperform the saturated counters in all the examined sizes of the prediction table. When the table size is 0.25 or 0.5 K entries the threshold value that yields the best ILP is quite tight – 70%. As the table is increased this value becomes much less tight – 60% for 1 K-entry table and 50% for 2–4 K entries table. As was observed in *perl* and in some other benchmarks, our classification mechanism (using the best threshold value) can utilize the table resources more efficiently and achieve in this benchmark a similar ILP increase relative to the saturated counters while using a prediction table which is half the size.

The ILP in *mgrid* is barely affected by the prediction table size. It can be observed that the threshold value that yields the best ILP for the profiling-based classification is 60%. However, it still cannot outperform the saturated counters. These observations are consistent with those reported in the Section 5.2, where we observed that the profiling-based classification could not offer any significant increase in the number of correct predictions relative to the saturated counters.

5.4. The potential benefits of using a hybrid value predictor

In this section we evaluate the performance and the potential benefits when using a hybrid value predictor with our profiling-based classification. Our hybrid predictor consists of two prediction tables a 'last-value' prediction table and a relatively small 'stride' prediction table. We have previously indicated (Section 3.1) that such a technique enables better utilization of the prediction table resources since we observed that the subset of instructions that exhibit stride value predictability is significantly smaller than those exhibiting last-value predictability. We have chosen for our next experiment a 1 K-entry last-value prediction table and 128-entry stride prediction table (8-times smaller than the last-value table). Each table was organized as a 2-way set-associative cache. The ratio of the last-value prediction table size to the stride prediction table size was determined according to the results that were reported in Fig. 4 where we observed that the subset of instructions that exhibit stride value predictability is approximately 10-times smaller than those exhibiting last-value predictabilty.

In our next experiments we assume that the user provides the compiler and the profiler with an additional threshold value termed *stride threshold*. When the compiler decides to tag an instruction as 'value predictable' (by comparing its individual prediction accuracy with the first profiling threshold value provided by the user) it still needs to determine the directive type. This is done by comparing the stride-efficiency ratio that was measured for the instruction with the stride threshold. If the stride-efficiency ratio is greater than the stride threshold, the instruction is tagged with the stride directive; otherwise it is tagged with the last-value directive. In order to determine the profiling threshold for each benchmark we have used the results from the previous subsection and have chosen the threshold value that yields

Table 5
The profiling threshold values and the stride threshold values that were used for the hybrid predictor

Benchmark	Profiling threshold (%)	Stride threshold (%)
go	50	10
m88ksim	10	40
gcc	30	10
compress	30	60
li	40	40
ijpeg	60	10
perl	30	10
vortex	60	90
mgrid	60	10

the best ILP. In order to determine the stride threshold, we were assisted by the measurements presented in Fig. 4. This figure allows us to detect the stride threshold value that will allow the majority of instructions which exhibit stride value predictability to be tagged with the stride directive. For instance, in the benchmark go we observe (in this figure) that the majority of instructions that exhibit stride value predictability have a stride efficiency ratio greater than 10%, and therefore we set this stride threshold value to be 10% as well. In Table 5 we summarize the profiling threshold values and the stride threshold values that were taken for the examined benchmarks.

In our next experiments we compare the performance of our hybrid predictor that uses the profiling classification to 1 K-entry, 2-way set-associative stride predictor which uses either saturated counters or profiling-based classification[3] (as in the previous subsection). If we assume that the last-value field in the prediction table is 4 bytes as well as the stride field, the stride prediction table size (not including the tags) is 8 KB. The size of our competitive hybrid predictor (not including the tags) is 4 KB for the last-value prediction table and 1 KB for the stride prediction table. We are about to show that our 5 KB hybrid predictor can gain comparable or even better performance than the stride predictor (when using either hardware-based or profiling-based classification).

First, we compare the number of correct predictions and the number of mispredictions that are gained by the hybrid predictor and by the stride predictor when using the profiling-based classification. These results are illustrated in Fig. 11. In all the examined benchmarks we observe that the hybrid predictor yields similar numbers of correct predictions and mispredictions relative to the stride predictor and in several cases it even slightly gains more correct predictions and less mispredictions. This figure provides the first evidence indicating that our hybrid predictor is significantly more cost-effective than the stride predictor.

Our next measurements compare the ILP increase that is gained by using value prediction (relative to the case when

[3] The profiling threshold values that are used for each benchmarks in the case of stride predictor are assumed to be those which yield the best ILP as it was indicated in Section 5.3 and Table 5.

Table 6
The ILP Increase gained by the three prediction mechanisms

ILP increase Benchmark	Stride + FSM (%)	Stride + profiling (%)	Hybrid + profiling (%)
go	12.6	16.2	16.9
m88ksim	586.7	576.8	583.8
gcc	18.1	24.5	26.4
compress	11.6	13.2	19.1
li	43.4	42.5	44.5
ijpeg	14.2	11.9	12.1
perl	25.3	33.1	37.8
vortex	168.1	185.7	185.8
mgrid	26.6	12.3	12.4

value prediction is not used) for three value prediction mechanisms: (1) a stride predictor using saturated counters; (2) a stride predictor which uses profiling-based classification; and (3) a hybrid predictor which uses profiling-based classification. Our results, summarized in Table 6, indicate that our hybrid predictor always outperforms the stride predictor which uses the profiling-based classification. In addition, in six out of our nine benchmarks it also outperforms the stride predictor which uses hardware-based classification. In the benchmarks *m88ksim* and *ijpeg*, although the hybrid predictor gains slightly less performance than the stride predictor with the hardware-based classification, it is still much more cost-effective using the hybrid predictor.

6. Conclusions

This paper introduced a profiling-based technique to enhance the efficiency of value prediction mechanisms. The new approach suggests using program profiling in order to classify instructions according to their tendency to be value-predictable. The collected information by the profiler is supplied to the value prediction mechanisms through special directives inserted into the opcode of instructions. We have shown that the profiling information which is extracted from previous runs of a program with one set of input parameters is highly correlated with the future

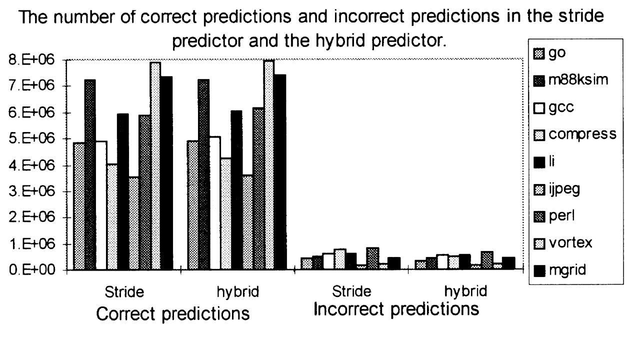


Fig. 11. The number of correct and incorrect predictions that are gained by the stride predictor in comparison to the hybrid predictor.

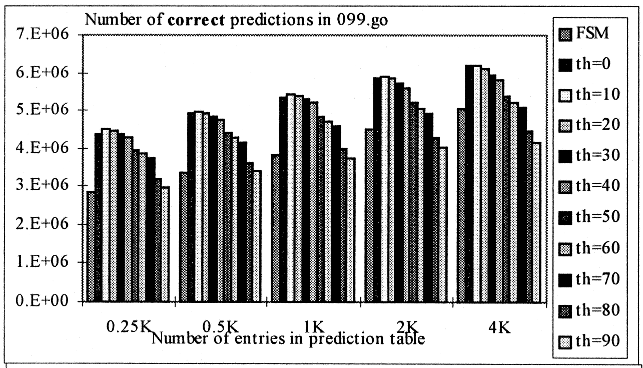
runs under other sets of inputs. This observation is very important, since it reveals various opportunities to involve the compiler in the prediction process and thus to increase the accuracy and the efficiency of the value predictor.

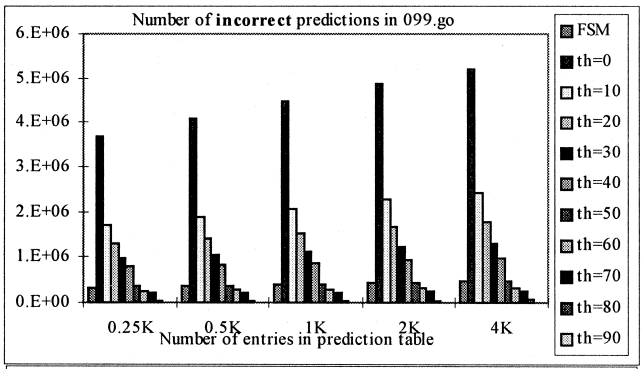
Our experiments also indicated that the profiling information can distinguish between different value predictability patterns (such as ‘last-value’ or ‘stride’). As a result, we suggested using a hybrid value predictor that consists of two prediction tables: the last-value and the stride prediction tables. A candidate instruction for value prediction can be allocated to one of these tables according to its profiling classification. We have shown that this capability allows us to exploit both value predictability patterns (stride and last-value) and utilize the prediction tables more efficiently.

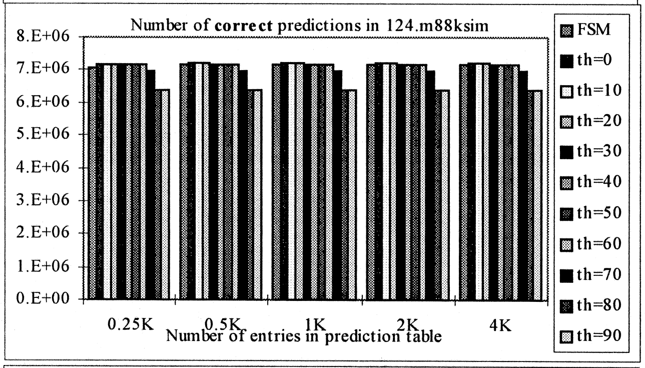
Our performance analysis showed that the profiling-based mechanism could be tuned by choosing the right threshold value so that it outperformed the hardware-only mechanism in most benchmarks. In many benchmarks we could accomplish both a significant increase in the number of correct predictions and a reduction in the number of mispredictions. In addition we have evaluated the performance of the hybrid predictor using profiling-based classification and compared it with the stride predictor when using either hardware-based or profiling-based classification. Our experiments indicate that the hybrid predictor is the most cost-effective predictor among the prediction mechanisms which have been evaluated. We have demonstrated this by showing that a 5 KB hybrid predictor can gain better or comparable performance than an 8 KB stride predictor.

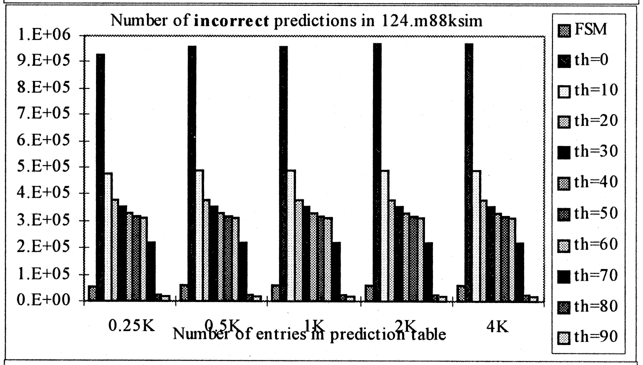
The innovation in this paper is very important for future integration of the compiler with value prediction. We are currently working on other properties of the program that can be identified by the profiler to enhance the performance and the effectiveness of value prediction. We are examining the effect of the profiling information on the scheduling of instruction within a basic block and the analysis of the critical path. In addition, we are also exploring the effect of different programming styles such as object oriented on the value predictabilty patterns.

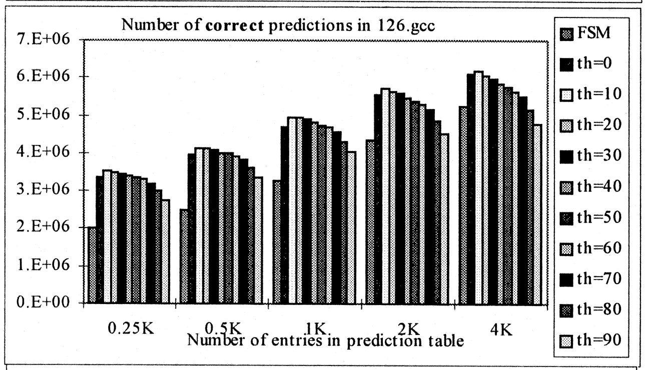
Appendix A

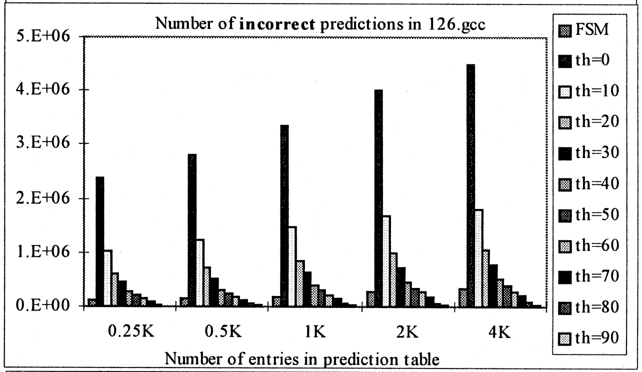

Number of correct predictions in 099.go
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

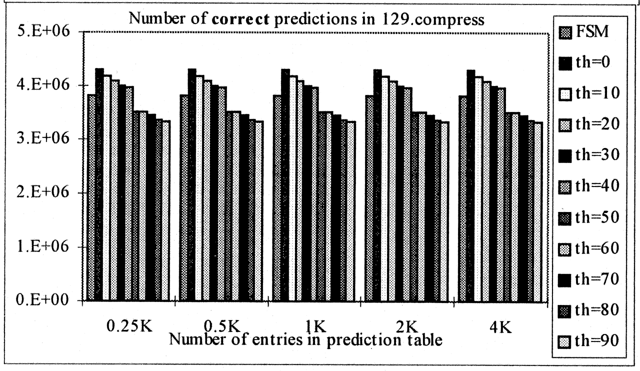

Number of incorrect predictions in 099.go
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

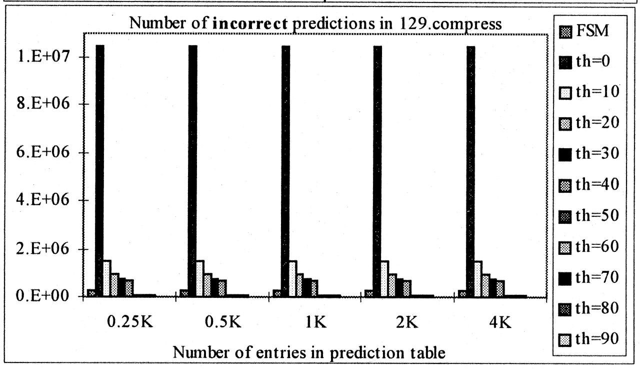

Number of correct predictions in 124.m88ksim
8.E+06
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

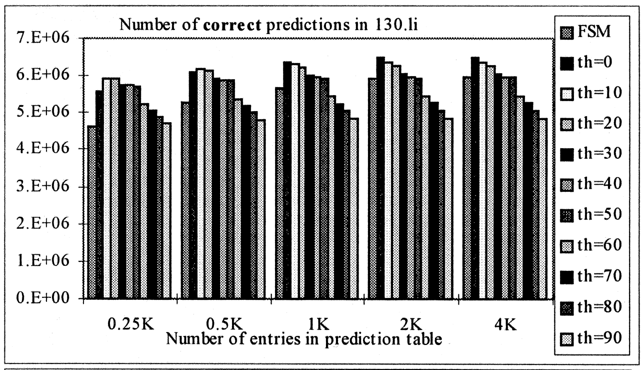

Number of incorrect predictions in 124.m88ksim
1.E+06
9.E+05
8.E+05
7.E+05
6.E+05
5.E+05
4.E+05
3.E+05
2.E+05
1.E+05
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

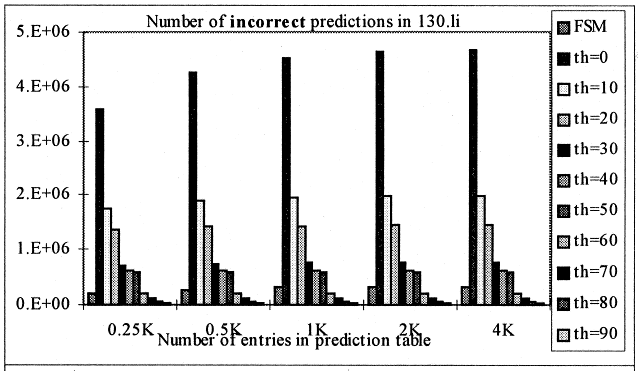

Number of correct predictions in 126.gcc
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

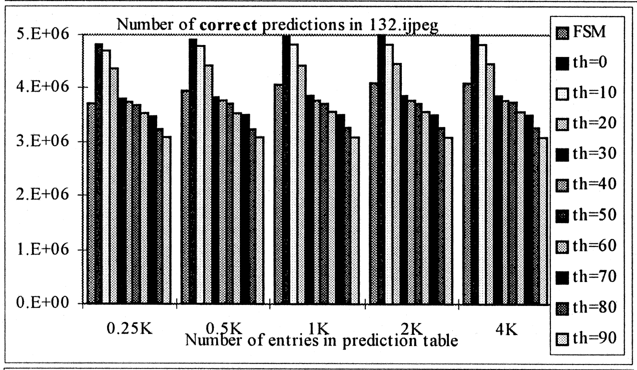

Number of incorrect predictions in 126.gcc
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

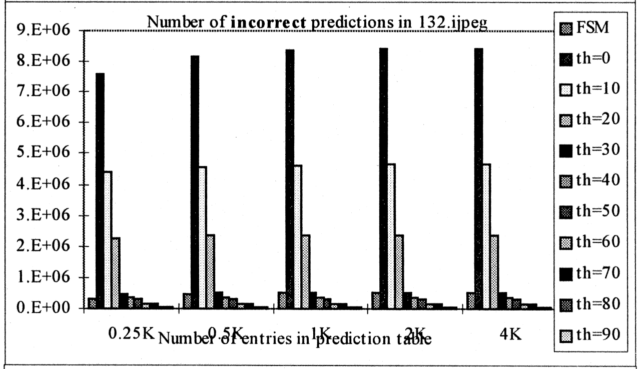

Number of correct predictions in 129.compress
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

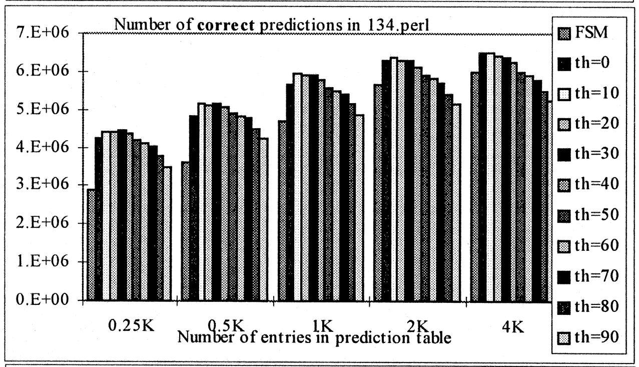

Number of incorrect predictions in 129.compress
1.E+07
8.E+06
6.E+06
4.E+06
2.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

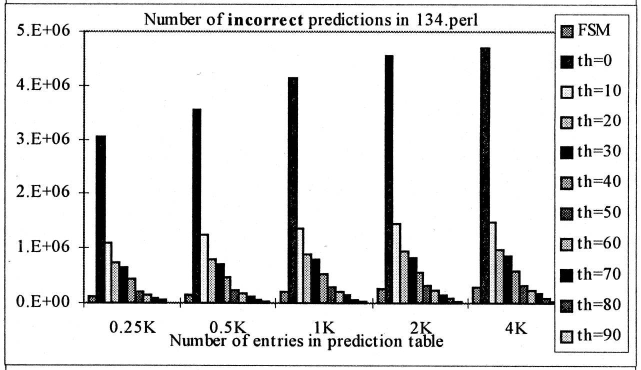

Number of correct predictions in 130.li
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

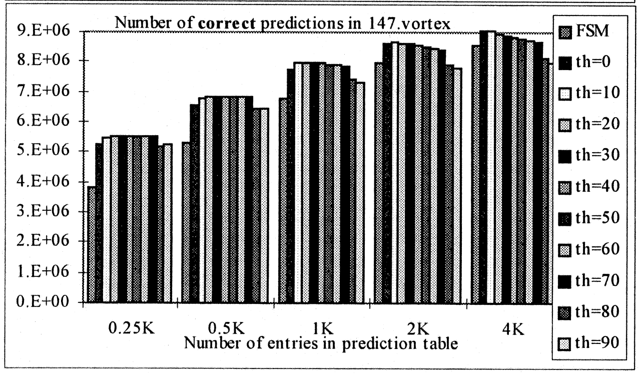

Number of incorrect predictions in 130.li
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

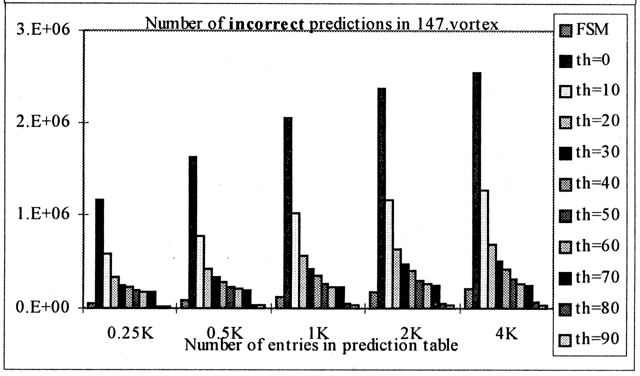

Number of correct predictions in 132.ijpeg
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

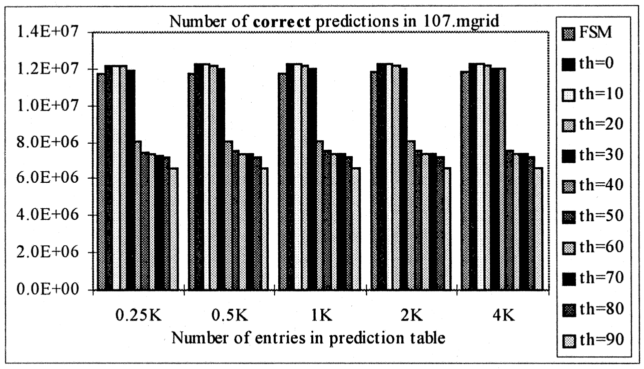

Number of incorrect predictions in 132.ijpeg
9.E+06
8.E+06
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

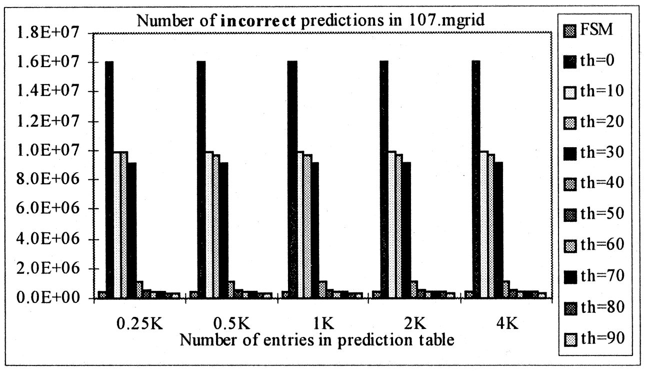

Number of correct predictions in 134.perl
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90


Number of incorrect predictions in 134.perl
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

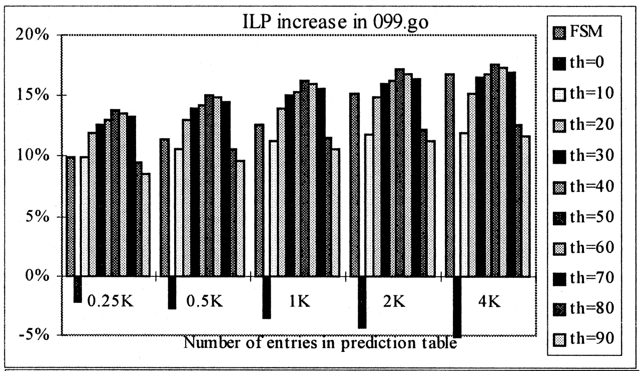

Number of correct predictions in 147.vortex
9.E+06
8.E+06
7.E+06
6.E+06
5.E+06
4.E+06
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

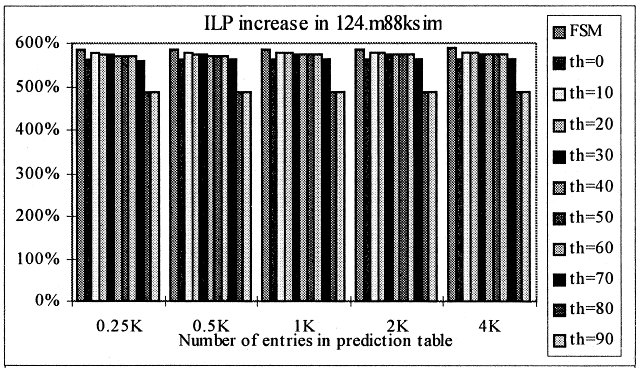

Number of incorrect predictions in 147.vortex
3.E+06
2.E+06
1.E+06
0.E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

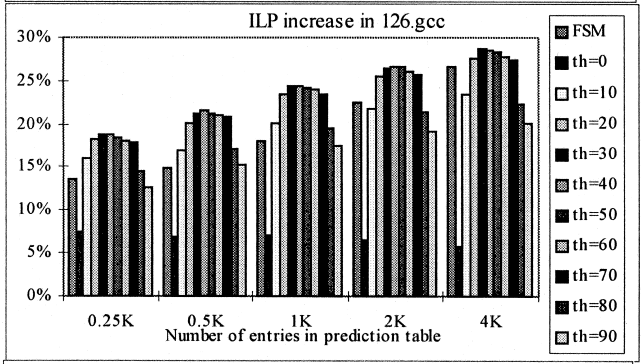

Number of correct predictions in 107.mgrid
1.4E+07
1.2E+07
1.0E+07
8.0E+06
6.0E+06
4.0E+06
2.0E+06
0.0E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

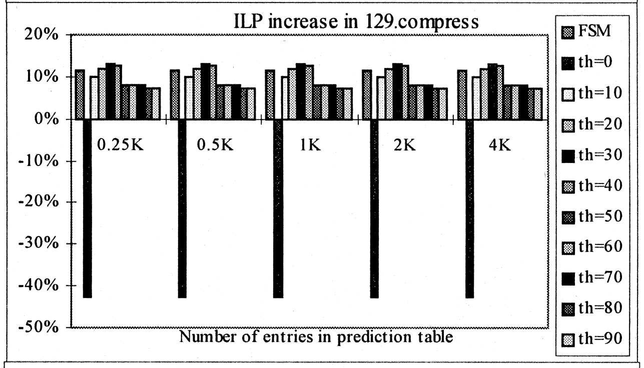

Number of incorrect predictions in 107.mgrid
1.8E+07
1.6E+07
1.4E+07
1.2E+07
1.0E+07
8.0E+06
6.0E+06
4.0E+06
2.0E+06
0.0E+00
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

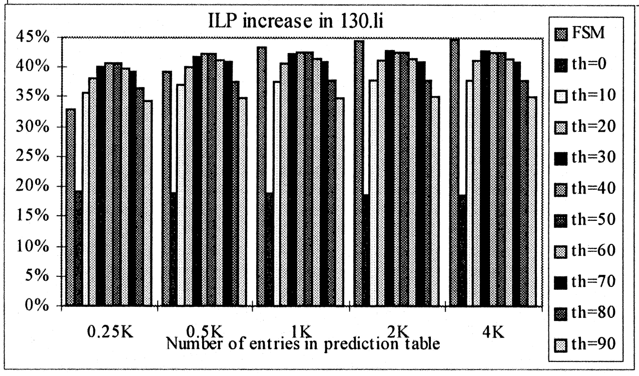
Appendix B

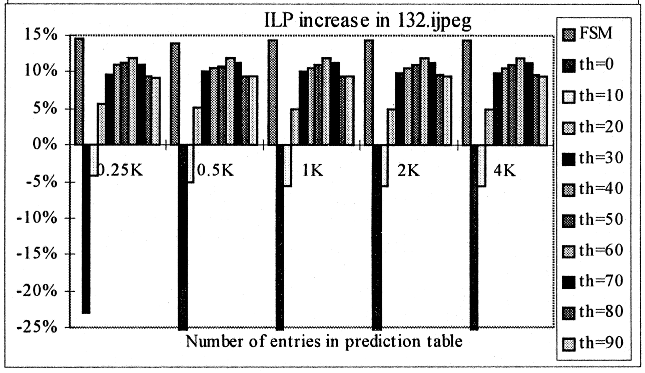

ILP increase in 099.go
20%
15%
10%
5%
0%
-5%
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

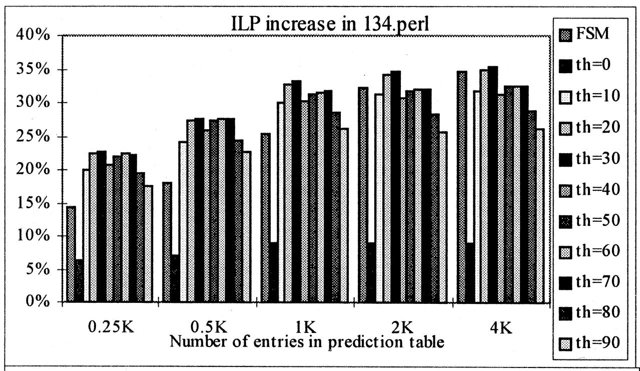

ILP increase in 124.m88ksim
600%
500%
400%
300%
200%
100%
0%
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

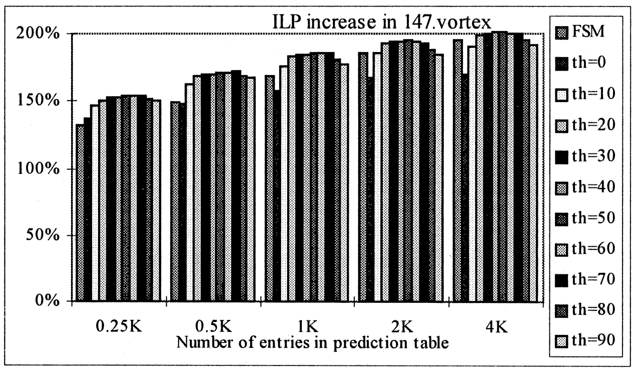

ILP increase in 126.gcc
30%
25%
20%
15%
10%
5%
0%
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90

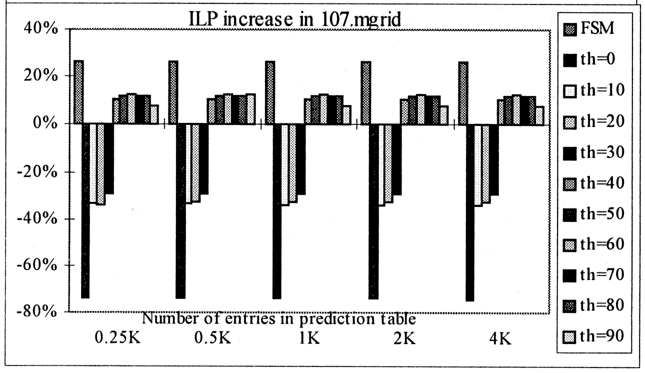

ILP increase in 129.compress
20%
10%
0%
-10%
-20%
-30%
-40%
-50%
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90


ILP increase in 130.li
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90


ILP increase in 132.ijpeg
15%
10%
5%
0%
-5%
-10%
-15%
-20%
-25%
0.25K
0.5K
1K
2K
4K
Number of entries in prediction table
FSM
th=0
th=10
th=20
th=30
th=40
th=50
th=60
th=70
th=80
th=90







References

[1] Hennessy J.L., Patterson. D.A., Computer Architecture a Quantitative Approach, 2nd ed. Morgan Kaufman, New York, 1995.

[2] Johnson. M., Superscalar Microprocessor Design. Prentice Hall, Englewood Cliffs, NJ, 1990.

[3] Wall. D.W., Limits of instruction-level parallelism, in: Proc. 4th Conf. on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 248–259.

[4] Davidson, S. D., Landskov, B., Shriver D., Mallet, P.W., Some experiments in local microcode compaction for horizontal machines. IEEE Transactions on Computers C-30 (7) (1981) 460–477.

[5] Ellis, J.R., Bulldog: a Compiler for VLIW Architecture. MIT Press, Cambridge, MA, 1986.

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

[7] Lam, M.S., Software Pipelining: An Effective Scheduling Technique for VLIW Processors. Proc. of the SIGPLAN'88 Conference on Programming Languages Design and Implementation. ACM, 1988, pp. 318–328.

[8] A. Smith, J. Lee, Branch prediction strategies and branch-target buffer design, Computer 17 (1) (1984) 6–22.

[9] Smith, J.E., A study of branch prediction techniques. In: Proc. of the 8th International Symposium on Computer Architecture, June, 1981.

[10] Weiss S., Smith. J.E., A study of scalar compilation techniques for pipelined supercomputers. In: Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, Oct., 1987, pp. 105–109.

[11] Yeh T.Y., Patt. Y.N., Alternative implementations of two-level adaptive branch prediction. In: Proc. of the 19th International Symposium on Computer Architecture, May, 1992. pp. 124–134.

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

[13] Lipasti, M.H., Shen, J.P., Exceeding the dataflow limit via value prediction. In: Proc. of the 29th annual ACM/IEEE International Symposium on Microarchitecture, Dec., 1996.

[14] Gabbay F., Mendelson, A., Speculative execution based on value prediction. EE Department TR #1080, Technion – Israel Institute of Technology, Nov., 1996.

[15] Gabbay F., Mendelson, A., Using value prediction to increase the power of speculative execution hardware. In: ACM Transactions on Computer Systems, Aug., 1998 (to appear).

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

[17] Motorola, PowerPC 601 User's Manual. 1993, No. MPC601UM/AD.

[18] Gabbay F., Mendelson, A., The effect of instruction fetch bandwidth on value prediction. In: Proceedings of the 25th International Symposium on Computer Architecture, June, 1998.

Freddy Gabbay received 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 B.Sc. and M.Sc. degrees in computer science from the Technion, Haifa, Israel in 1979 and 1982, and a 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.