Leveraging dynamic slicing to enhance indirect branch prediction

Conference Paper · October 2014

DOI: 10.1109/ICCD.2014.6974696

2 authors, including:

Walid Ghandour
American University of Beirut
10 PUBLICATIONS **23** CITATIONS

Leveraging Dynamic Slicing to Enhance Indirect Branch Prediction

Walid J. Ghandour*, Nadine J. Ghandour†
*Computer Engineering Department
Fahad Bin Sultan University, Tabuk, Saudi Arabia
Email: ghandour@utexas.edu
†Mathematics Department, Lebanese University, Beirut, Lebanon
Email: nadin.ghandour@ul.edu.lb

***Abstract*—An indirect branch specifies where the address of the next instruction to execute is located. The address to be jumped to is not known until the instruction is executed. Indirect branches can incur a significant fraction of branch misprediction overhead even though they remain less frequent than the more predictable conditional branches. Indirect branch instructions implement multiway branch statements and virtual function calls in object-oriented languages. Multiple targets indirect branch instructions are difficult to predict.**

We employ dynamic slicing to categorize indirect branches and use the most convenient predictor for each category. We also propose Dynamic Slicing directed Indirect Branch (DSIB) predictor, a profiling based target address prediction technique for indirect branch instructions.

We compare the proposed predictor to VBBI and ITTAGE predictors. DSIB enhances prediction accuracy by 13% and 9% over VBBI and ITTAGE respectively. DSIB also results in IPC improvement of 11%, 4.2% and 3.55% over BTB, VBBI and ITTAGE respectively.

***Keywords*—branch target prediction, correlation-based prediction, dynamic slicing, indirect branch.**

I. Introduction

An indirect branch specifies where the address of the next instruction to execute is located, rather than specifying the address itself, as in a direct branch. Indirect branches can depend on the value of a register or memory location. The target address is not known until the instruction is executed. A single indirect branch can have multiple targets.

Indirect branches implement common programming constructs such as switch statements, virtual function calls and calls through function pointers. Correctly predicting the target addresses of indirect branches can significantly enhance the performance of programs that make a lot of use of the programming features that result in indirect branches.

It is more difficult to predict an indirect branch that might have multiple possible targets than a conditional branch that has only two possible outcomes: taken and non-taken. Some processors, such as Intel Pentium M [1], include an indirect branch predictor.

The branch target buffer (BTB) [2] predictor is the simplest indirect branch predictor that predicts the current target address as the last one. This can achieve high accuracy for monomorphic indirect branches, branches that usually jump to the same target; however, low accuracy is achieved for polymorphic indirect branches, indirect branches that have multiple target addresses.

The two-level conditional branch prediction technique was adapted to predict indirect branches. Some bits of the target address of the currently executing indirect branch are selected and hashed into the history register. This approach intends to exploit correlation between different indirect branches, or between multiple executions of the same branch. If more than one target has the same pattern in the selected bits, the predictor will not be able to distinguish between them [3], [4], [5], [6].

This paper employs dynamic slicing to categorize indirect branches and use the most convenient predictor for each category. It also proposes Dynamic Slicing directed Indirect Branch (DSIB) predictor, a profiling based target address prediction technique for indirect branch instructions.

Program slicing is an approach that determines the set of statements on which the value of a given variable depends. It was originally introduced by Weiser [7] who proposed the first static slicing algorithm to help debugging programs. Afterward, Korel and Laski [8] proposed dynamic slicing that considers runtime information when selecting the set of statements that affects a specific program variable.

We employ dynamic slicing to capture flow of information from each indirect branch instruction to itself, and then measure the strength of flow using information theory and correlation techniques to classify the indirect branch instruction, and pass it to the appropriate predictor. For instructions that are not predictable using local prediction techniques, we also use dynamic slicing to identify, for a given indirect branch instruction, the last instruction that updates the target operand of the indirect branch before it reads it. We call this instruction *source instruction*. The instruction pointer (IP) of the indirect branch is hashed with the outcome of the source instruction to index the BTB. This index is used to read and update the value of the target address that corresponds to the executed indirect branch with the given source instruction outcome.

We employ dynamic binary instrumentation to obtain dynamic slices.

978-1-4799-6492-5/14/$31.00 ©2014 IEEE

Profiling has been used for branch prediction [9], [10], dynamic predication [11], value prediction [12], [13], [14], and program parallelization [15].

A. Paper Contributions

This paper makes the following contributions:

- Uses Dynamic Slicing to classify indirect branch instructions: (i) single target, multiple targets (ii) with / (iii) without deterministic pattern. Single target indirect branches are predicted using regular BTB predictor. Multiple targets indirect branches that follow deterministic patterns can be predicted using a value predictor[1]. Multiple targets indirect branches that do not follow deterministic patterns can be predicted by correlation over source instructions that determine the target addresses of the indirect branches.
- Presents Dynamic Slicing directed Indirect Branch (DSIB) predictor.
- Describes an implementation of the proposed predictor and shows its potential performance.

B. Paper Overview

The rest of this paper is organized as follows. Section II provides background information and reviews related work. Section III presents the methodology we follow. Section IV describes the proposed dynamic-slicing based indirect branch predictor. We show its potential performance in Section V. Finally, we conclude in Section VI.

II. Background and Previous Work

We survey in the following section relevant work conducted in the area of indirect branch prediction.

A. Indirect Branch Prediction

Most existing indirect branch prediction techniques can be classified as either history-based [3], [5], [16], [17], [18], [19], [20] or precomputation-based [21].

History-based techniques predict the target of an indirect instruction based on its previous history and/or history of previously executed indirect branch instructions. The branch target buffer (BTB) is used to predict the target of the branch as the last taken target [2]. This works well for branches that usually jump to the same target, which are often known as monomorphic indirect branches. However in the case of indirect branches which jump to many different targets, known as polymorphic indirect branches, the prediction accuracy is usually poor. The *tagged target cache* (TTC) prediction technique is a history based two-level technique that uses the indirect branch address and a history register to index the target cache and predict the target address. The history register contains selected bits of the target addresses of the recently executed indirect branch instructions [3]. Cascade predictor [17], Rehashable predictor [19], PPM predictor [18] and Target Address Pointer (TAP) predictor [22] intend to enhance the effectiveness of the history-based correlation by using large storage or complex algorithm. VPC prediction [20] attempts to reduce the indirect branch predictor complexity by making use of existing processor components, however it needs major modifications to the conditional branch predictor and the BTB structure. VPC prediction treats an indirect branch with T targets as T virtual direct branches, each with its own unique target address. Its additional long prediction latency significantly reduces the overall performance [22]. The GEometric History Length (GEHL) predictor features several predictor tables indexed through independent functions of the global branch history and branch address. The Optimized GEometric History Length (O-GEHL) branch Predictor improves the ability of the GEHL predictor to exploit very long histories through the addition of dynamic history fitting and dynamic threshold fitting [23]. The ITTAGE predictor [24], [25], an extension of the aggressive TAGE predictor [24], [26], achieves high prediction accuracy; However, it requires large storage and complex logic to implement the Prediction by Partial Match (PPM) algorithm. Aggressive predictors are usually inefficient because of their large storage and complexity [27].

Precomputation-based target prediction techniques attempt to pre-calculate the target address [21], [28], [29]. Most of the existing precomputation techniques address selected indirect branch instructions scenarios and act as complement to history-based techniques. Roth et al. [21] presented a pre-computation technique in the context of virtual function calls. Kaeli et al. [28] presented a technique to handle switch-case statements.

Compiler techniques such as inserting NOP and re-ordering the cases inside the switch statement are proposed to reduce collisions in the indirect branch predictor [30]. Compiler-Guided Value Pattern (CVP) prediction employs compiler-microarchitecture cooperation. CVP uses the compiler-guided value pattern as the correlated information to hint the indirect branch prediction at runtime [31].

Value Based BTB Indexing (VBBI) is a correlation-based target address prediction scheme for indirect branch instructions. VBBI requires profiling at the assembly level, mapping of indirect branch instruction to the source code, identifying the last assignment of the variable on which the indirect branch depends at the source level, and them mapping it to the assembly level [32].

We present in this paper a simple profiling-based correlation indirect branch prediction technique that employs dynamic slicing.

The following section describes information flow analysis that we use in the proposed indirect branch predictor.

B. Information Flow Analysis

A statement t is data dependent on a statement s if t uses a variable that is defined in s. A statement t is control dependent on a statement s if s decides via branches it controls whether t is executed or not [33], [34], [35]. Information flows from a source object x to a target object y whenever information stored in x is passed directly or indirectly to object y [36]. A given program statement designates a specific flow f if executing the

[1]Data value prediction is a technique to increase instruction level parallelism by attempting to overcome serialization constraints caused by true data dependences. By predicting instruction results early and passing them to dependent instructions, it can reduce the length of critical paths through a program.

statement could result in the flow *f*. Flows can be either direct or indirect.

Direct information flow is due to the propagation of information as a result of data dependence. Source variable is directly involved in the computation of the target variable value. The occurrence of a direct flow is independent of the value of its source object. Indirect information flow is due to control dependence. Source object indirectly affects the value of the target object.

Program slicing is an approach that determines the set of statements on which the value of a given variable depends. It was originally introduced by Weiser [7] who proposed the first static slicing algorithm to help debugging programs. Afterward, Korel and Laski [8] proposed dynamic slicing that considers runtime information when selecting the set of statements that affects a specific program variable. The previously mentioned algorithms use *backward* analysis, which requires the a priori availability of the execution trace. Forward computing dynamic slicing and information flow analysis algorithms, which we use in this paper, do not need the a priori availability of the execution trace; hence, they are more convenient for online and interactive applications. Forward computing dynamic slicing was initially introduced by Korel and Yalamanchili [37].

In the following two sections we provide a brief description of the information theory and correlation techniques used to calculate the strength of the information flow mentioned above.

C. Information Theory Background

Information theory was originally developed by Shannon in the context of secure communication and cryptography. It has been applied in diverse fields.

Entropy is a measure of the uncertainty of a random variable. It explicitly quantifies the information content in a given source of data. Conditional entropy of a random variable X given that the random variable Y is known measures the remaining uncertainty about the value of X after knowing the value of Y [38], [39]. Mutual information is a measure of the amount of information one random variable contains about another. It can be normalized to a value between 0 and 1.

D. Correlation-Based Techniques

The Pearson product moment correlation coefficient, which is also known as *Pearson's r* or *standard r*, measures the strength of the correlation (linear dependence) between two variables. It ranges between -1 and $+1$ [40].

The correlation ratio or *eta coefficient* provides the same value as *standard r* for linearly related variables and a greater value for non-linearly related variables. The difference between *eta* and *r* presents a measure of the non-linearity of the relation between two variables [41], [42].

Normalized mutual information and correlation techniques are used to quantify the strength of information flows, which in turn is used to classify indirect branch instructions and correlate some of these instructions with source instructions on which they strongly depend.

III. Methodology

Pin [43] is a dynamic binary instrumentation tool. Instrumentation is a technique that inserts code into a program to collect run-time information. *Pin* does not need the source code, and does not require recompilation and post-linking. It provides rich APIs that allow the user to write instrumentation tools called pintools in C, C++ or assembly. It is supported on Linux and Windows for x86, x86-64 and Itanium architectures. It can instrument real-life applications such as database and web browsers, and multithreaded applications.

Pin has been widely used in several research community. It is used in computer architecture research for trace generation [44], branch predictor, cache modeling [45], memory access scheduler [46], value prediction [14], fault tolerance studies, emulating speculation, emulating new instructions, power, area and timing modeling [47].

We use *Pin* to implement a dynamic slicing tool for x86 binaries and employ the tool to study dynamic slicing based indirect branch prediction. The tool determines information flow between instructions and calculates their strength. Figure 1 provides an overview of information flows generation using our pintool.

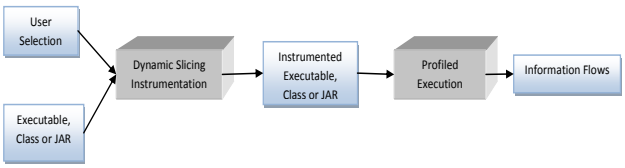

Fig. 1. Overview of the slicing tool operation.

By default, the tool calculates assembly dynamic slices of all dynamically executed instructions. We provide several command line options that enable the user to specify the types of instructions to instrument, select a particular routine to instrument, specify the routine at which instrumentation starts and the routine at which it terminates, specify the number of instructions to skip before starting instrumentation, specify a particular path of interest, specify an upper limit on the number of instructions to instrument, et cetera. In this paper, we are interested in indirect branch instructions only.

Figure 2 presents an overview of strength of flow computation and slicing directed indirect branches prediction. An analysis tool, that takes as input the information flows and their corresponding values obtained using the slicing tool, identifies instructions that always take the same value and computes the strength of the remaining identified dependences using information theoretic and statistical metrics applied on their associated values. Instructions that always take the same value are predicted using traditional BTB predictor. According to their strength of flow, the remaining instructions are predicted using the appropriate predictor.

We use *PTLsim* [48], a cycle accurate microprocessor simulator and virtual machine for x86 and x86-64 instruction sets, to model dynamic slicing based indirect branch predictor. *PTLsim* simulates x86 codes after converting complex instructions into RISC-like micro-ops (uops), a technique used in Intel and AMD processors [49]. *PTLsim* produces up to 16 micro-ops per x86 macro-instruction. We assign an ID that uniquely identifies uops that are obtained from the same x86 instruction.

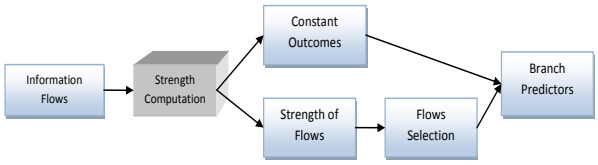

Fig. 2. Overview of strength of flow computation and indirect branches prediction.

The value of a uop ID ranges from 0 to 15, and it is assigned in order to each uop. The x86 instruction IP and the uop ID are hashed to uniquely identify each uop.

We conduct our study using SPEC CPU 2006 integer benchmarks [50]. A description of the SPEC CPU 2006 programs is provided in Table I. All of the SPEC CPU 2006 benchmarks were run for 1 billion committed instructions with the configuration parameters for our simulations given in Table II.

We describe in the following section our indirect branch prediction technique.

IV. Indirect Branch Prediction

In this section we present and describe our newly introduced indirect branch prediction approach.

We initially classify indirect branches into three categorizes: single target, multiple targets with deterministic pattern and multiple targets with non-deterministic pattern. Instructions of each category are predicted using the most convenient technique. This has the advantage of achieving higher prediction accuracy rate, using smaller prediction tables and decreasing the destructive impact of aliasing when it is present. Reducing prediction tables sizes and using highly accurate predictors help reducing energy consumption.

Regular BTB [2] is used to predict single target instructions. Value prediction techniques [51] are used to predict multiple targets with deterministic pattern instructions. We employ Differential Finite Context Method (DFCM) value prediction technique [52].

Multiple targets with non-deterministic pattern indirect branch instructions are predicted using global BTB predictor. When the source instruction outcome is not ready, we predict the target indirect branch instruction using the path history scheme [53].

Figure 3 provides an overview of the proposed prediction technique.

A. Indirect Branch Instructions Classification

Using dynamic slicing, profiling and correlation techniques, we classify instructions into three categories. We identify single target indirect branch instructions as the indirect branch instructions identified as constant outcome instruction using the dynamic slicing analysis tool. A *constant outcome* instruction is an instruction that always produce the same outcome.

We identify instructions that have strong linear or non-linear relations into themselves as multiple targets with deterministic pattern instructions. An Instruction with strong linear

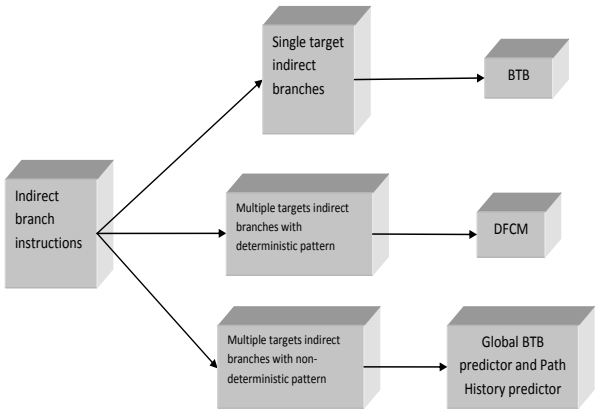

Fig. 3. Overview of the proposed prediction technique.

/ non-linear flow into itself, is an instruction whose successive outcomes can be related via a linear / non-linear relation. This relation can be used to obtain the next outcome knowing the current one. Techniques shown in Sections II-C and II-D are used to identify linear and non-linear flows.

The remaining instructions are the multiple targets with non-deterministic pattern. For this category, we use dynamic slicing to identify for each indirect branch a set of source instructions (one or more instructions) on which it strongly depends. We hash the IP of the indirect branch of interest with the outcome of a given source instruction to index a BTB buffer and predict the outcome of the target indirect branch instruction. When the source instruction outcome is not ready, we predict our target indirect branch instruction using path history prediction technique [53].

In the following section we describe the dynamic slicing algorithm we use to identify correlation between source instructions and indirect branches.

B. Indirect Branch Correlation Algorithm

Following is the summary of steps of the dynamic slicing algorithm we use to identify (source instruction, indirect branch) pairs.

```
Summary of Steps:
1. Each time an instruction ins updates an
   implicit or explicit operand, we set the
   corresponding entry of the updated
   operand to ins in the regDepLast map,
   for register operands, and in the
   memDepLast map for memory operands.
   regDepLast is a <REG, ADDRINT> map.
   memDepLast is a <ADDRINT, ADDRINT> map.
2. When an indirect branch instruction is
   encountered, we add the value of its
   target operand in regDepLast or
   memDepLast, according to the type of
   the operand, to the entry of the branch
   instruction in srcInst map.
   srcInst is a <ADDRINT, ADDRINT_set> map.
   We use ADDRINT_set to account for the
   case where an indirect branch can have
   more than one source instruction.
```

Benchmark	Application Area	Description
astar	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.
gcc	Compiler	Based on gcc Version 3.2, generates code for Opteron.
gobmk	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game.
hmmer	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs).
h264ref	Video Compression	A reference implementation of H.264AVC, encodes a videostream using 2 parameter sets. The H.264AVC standard is expected to replace MPEG2.
libquantum	Physics Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
mcf	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
omnetpp	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
perlbench	Programming Language	The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
sjeng	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.
xalancbmk	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

TABLE I. CINT2006 BENCHMARKS.

Machine width	4-wide fetch, 4-wide issue, 4-wide commit
Instruction L1 Cache	32KB 4-way, 2 cycles latency
Data L1 Cache	16KB 4-way, 2 cycles latency
L2 Cache	256KB 16-way, 6 cycles latency
L3 Cache	4MB 32-way, 16 cycles latency
Mem Latency	140 cycles
Branch Predictor	Combined bimodal and gshare, 64K meta, 64K bimodal, 64K gshare; 4K-entry, 4-way BTB with LRU replacement; 15 cycle min. branch misprediction penalty

TABLE II. PARAMETERS FOR OUR ARCHITECTURAL CONFIGURATION.

We outline in Algorithm 1 the pseudo-code of the dynamic slicing algorithm we implement using *Pin* to identify (source instruction, indirect branch) pairs.

Algorithm 1 Source Instruction Identification Algorithm

```
1: if (isIndirectBr(ins)) then
2:    if (INSOperandIsReg) then
3:       srcIns = regDepLast[insOperand]
4:    else if (INSOperandIsMem) then
5:       srcIns = memDepLast[insOperand]
6:    end if
7:    if (srcIns != NULL) then
8:       srcInst[ins].add(srcIns)
9:    end if
10: else if (isRegWriteIns(ins)) then
11:    regDepLast[targetOperand(s)] = ins
12: else if (isMemWriteIns(ins)) then
13:    memDepLast[targetOperand] = ins
14: end if
```

C. Indirect Branch Prediction Implementation Details

Indirect branch instructions are classified into three categories as mentioned in Section IV-A.

Constant outcome instructions are predicted using regular BTB predictor. The BTB is indexed using the IP of the indirect branch. BTB has the potential to achieve very high accurate prediction for this type of instructions since they always branch to the same target.

A context is a finite ordered sequence of values. A context based predictor makes value prediction based on its observation of previous patterns. An order *k Finite Context Method* (FCM) predictor uses k preceding values [54]. *Differential Finite Context Method* (DFCM) hashes recent history of strides rather than values and looks up a stride in the hashtable to make a prediction. DFCM has the potential to be more space efficient and faster to warm up than FCM [52].

We use DFCM predictor [52] for multiple targets with deterministic pattern indirect branch instructions. Since these instructions have a deterministic pattern, DFCM has the potential to predict their outcomes with very high accuracy. The DFCM is indexed using the IP of the indirect branch.

We use a separate BTB to predict multiple targets with non-deterministic pattern indirect branch instructions. The IP of the indirect branch is hashed with the outcome of the source instruction to index the BTB. Different targets of a given indirect branch are stored in the BTB at different indices according to the outcome of the source instruction.

When the source instruction outcome is not ready when needed to predict the outcome of the indirect branch, we use path history prediction technique [53] to predict the outcome of the indirect branch.

Path history consists of the target addresses of the dynamically executed branches that lead to the current branch. Global Path History Register (GPHR) is used to store this history. When a branch instruction is executed, GPHR is updated by concatenating part of the target address of the branch instruction with part of the content of the GPHR. The address of an indirect branch is hashed with the content of the GPHR to access the target buffer.

The following two approaches can be used to correlate the indirect branch instruction with its source instruction:

1) The offset of the indirect branch instruction from the source instruction is encoded in the indirect branch instruction employing its unused bits. In the case where an indirect branch instruction has more than one source instruction, we select the most frequently seen one. In the case where the format of the instruction allow the encoding of the offset from *n* source

instructions, we encode the most frequently *n* seen source instructions.
Hint bits, encoded in the instruction, are used to communicate compiler analysis to the architecture. Several processors instruction set architectures (ISA) already support hint bits. As an example, Itanium ISA [55] contains hint bits for branch prediction and memory locality. Researchers have proposed the usage of hint bits for value prediction [56]. Implicit hint is an approach that encodes hints in the registers names without changing the ISA [57].

2) A guiding instruction is inserted after the source instruction. This is a new instruction that needs to be added to the ISA. The opcode is a reserved opcode of the orignal ISA. The instruction holds the number of the register that contains the correlated value.

In this paper we use the first approach, and encode only one offset in the indirect branch instruction.

D. Hardware Support

We classify indirect branch prediction into three categories. The first category, constant target instructions, uses the regular BTB technique and requires a branch target buffer only.

The second category requires the hardware needed to implement a DFCM predictor [52]. DFCM requires two tables. The first table holds the last value and the history for each instruction. The second table holds the stride, the difference between its last two outcomes, for each instruction.

The third category requires a BTB predictor augmented with a Source Instruction Buffer (SIB) and a history path predictor which requires a tag-less target cache[53].

Each entry in SIB has three fields: indirect branch IP, source instruction IP, and the value of the source instruction. When the indirect branch is fetched, it reads the source instruction value to form the index to access the BTB; On the other hand, source instructions write their values to the SIB in the write-back stage.

Since we categorize the indirect branch instructions. Small tables are required for each one of the above mentioned predictors. The hardware of the used predictors are generally simple.

E. Implementation Cost

Using profiling, we classify instructions and use the most convenient predictor for each type of instructions. This information is passed to the hardware using indirect branch encoding.

We use 1K-etnries BTB for predicting constant target instructions. We employ DFCM with 200 etnries level-1 table and 300 entries level-2 table for predicting multiple targets indirect branch instructions with deterministic pattern. We implement 520 bits Source Instruction Buffer (SIB), and 1 K-entries BTB for predicting the remaining instructions.

V. Experimental Results

A. Benchmarks Characteristics

Table III presents the characteristics of indirect branch instructions in SPEC CPU2006 evaluated benchmarks. We employ all SPEC CPU2006 benchmarks except *bzip2* that we have problem in building and running it.

B. Comparison with State of Art Predictors

We compare DSIB prediction with BTB [2], VBBI [32] and ITTAGE [25], [24]. Table IV shows the configuration of the used predictors.

Figure 4 shows the prediction accuracies for the BTB, VBBI, ITTAGE and DSIB predictors. DSIB achieves 37%, 13% and 9% on average better prediction accuracy than BTB, VBBI, and ITTAGE respectively.

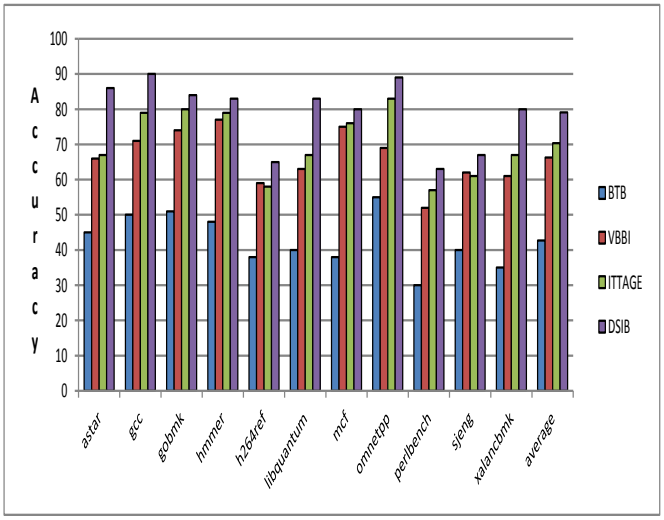

Fig. 4. Indirect branch prediction accuracies.

Figure 5 shows the IPC improvement for the VBBI, ITTAGE and DSIB predictors over BTB. DSIB achieves 11%, 4.2% and 3.55% on average better IPC improvement than BTB, VBBI and ITTAGE respectively.

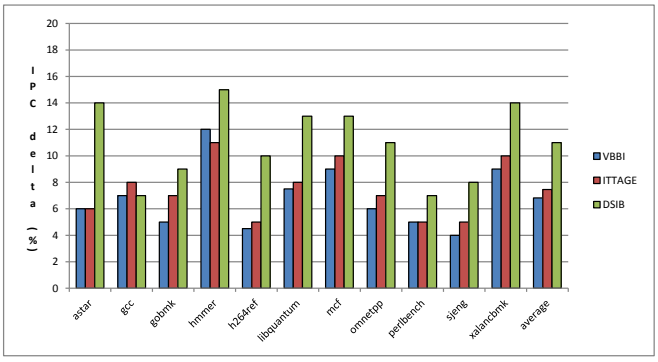

Fig. 5. VBBI, ITTAGE and DSIB IPC improvement over BTB.

VBBI predicts indirect branches that are hard-to-predict using the value of hint instructions. Other indirect branch instructions are predicted using traditional BTB. Identifying

	astar	gcc	gobmk	hmmer	h264ref	libquantum	mcf	omnetpp	perlbench	sjeng	xalancbmk
Static Indir. Br.	27	571	63	91	57	29	21	123	195	54	216
Dynamic Indir. Br. (K)	344	24805	81	31609	46469	31	737	14095	116	79121	1320
%Static Indir. / Total Static Instr.	$2 * 10^{-3}$	$6.7 * 10^{-4}$	$2.7 * 10^{-4}$	$1 * 10^{-3}$	$3.4 * 10^{-3}$	$2.76 * 10^{-3}$	$6.1 * 10^{-3}$	$8 * 10^{-4}$	$6.5 * 10^{-4}$	$1.7 * 10^{-3}$	$2.2 * 10^{-4}$
%Dynamic Indir. / Total Dynamic Instr.	$5.2 * 10^{-6}$	$3.5 * 10^{-3}$	$1.3 * 10^{-4}$	$8 * 10^{-4}$	$2.08 * 10^{-3}$	$3.77 * 10^{-5}$	$1.1 * 10^{-4}$	$3.5 * 10^{-3}$	$7.4 * 10^{-3}$	$3 * 10^{-3}$	$6.4 * 10^{-4}$

TABLE III. CHARACTERISTICS OF SPEC CPU2006 EVALUATED BENCHMARKS.

ITTAGE	256 Kbits, including 15 tagged tables and 1 tagless table. Tables are indexed ITTAGE with hashing of the global branch history, path history and branch address. No compiler support or profiling is required.
VBBI	1040 bits storage with target overriding support. 4K-entries BTB. Benchmarks are profiled using the same reference input data set.
DSIB	1K-etnries BTB for predicting constant target instructions. DFCM with 200 etnries level-1 table and 300 entries level-2 table for predicting multiple targets indirect branch instructions with deterministic pattern. 520 bits Source Instruction Buffer (SIB), and 1 K-entries BTB for predicting the remaining instructions.

TABLE IV. CONFIGURATION OF ITTAGE, VBBI AND DSIB PREDICTORS.

the hint instruction of a hard-to-predict indirect branch involves mapping from assembly to high level language and then mapping back to assembly. VBBI uses the old hint value to compute the BTB index, when the hint instruction has not finished its execution when the indirect branch instruction is fetched. A second target address prediction is made when the hint instruction output is available, if the new output is different from the old one. This is called target address prediction overriding.

ITTAGE predictor requires large number of bits for global history to catch distant branch correlation; However, only few branches make use of this information.

Our technique has the advantage of using a mathematical approach to classify indirect branch instructions, and predicting instructions of each category using the most convenient prediction technique. This reduces the negative aliasing impact and allow the usage of small tables to achieve high accuracy. Only those instructions who are not easy to predict using a local prediction technique are predicted using correlation over source instructions. We present a rigorous approach to identify a source instruction that can be used to predict a given indirect branch instruction. If the value of the source instruction is not available, we use path history prediction.

VI. CONCLUSION

We proposed and evaluated in this paper the Dynamic Slicing directed Indirect Branch (DSIB) predictor. The predictor employs dynamic slicing to classify indirect branch instructions and predict instructions of each category using the most convenient predictor. It also identifies for some instructions that are not predictable using local prediction techniques, the source instruction that last updates the value of the target operand of the target indirect branch, and uses the outcome of the source instruction to predict the indirect branch. We evaluated DSIB using SPEC CPU 2006 integer benchmarks and showed that it outperforms both VBBI and ITTAGE.

REFERENCES

[1] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine, "The intel pentium m processor: Micorarchitecture and performance," *Intel Technology Journal*, vol. 7, 2003.

[2] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, no. 1, 1984.

[3] P. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," *ISCA'97*.

[4] K. Driesen and U. Holzle, "Limits of indirect branch prediction," Santa Barbara, CA, USA, Tech. Rep., 1997.

[5] ——, "Accurate indirect branch prediction," *SIGARCH Comput. Archit. News*, vol. 26, no. 3, 1998.

[6] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," *ISPASS*, April 2009.

[7] M. Weiser, "Program slicing," ser. ICSE '81, 1981, pp. 439–449.

[8] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, pp. 155–163, October 1988.

[9] A. Ramirez, J. L. Larriba-Pey, and M. Valero, "Branch prediction using profile data," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, ser. Euro-Par '01, 2001, pp. 386–393.

[10] A. Mahesri, A. Fadol, and C. Zilles, "Profile-assisted dynamic branch prediction."

[11] H. Kim, J. Joao, O. Mutlu, and Y. Patt, "Profile-assisted compiler support for dynamic predication in diverge-merge processors," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 367–378.

[12] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?" in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997, pp. 270–280.

[13] B. Calder, P. Feller, and A. Eustace, "Value profiling and optimization," *Journal of Instruction Level Parallelism*, vol. 1, no. 1, pp. 1–6, 1999.

[14] W. J. Ghandour, H. Akkary, and W. Masri, "The potential of using dynamic information flow analysis in data value prediction," *PACT '10*, pp. 431–442, September 2010.

[15] T. Moseley, D. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th international conference on Computing frontiers*, 2007, pp. 143–152.

[16] J. Kalamatianos and D. R. Kaeli, "Predicting indirect branches via data compression," *MICRO-31*, 1998.

[17] K. Driesen and U. Holzle, "The cascaded predictor: economical and adaptive branch target prediction," *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture, MICRO 31*, pp. 249–258, 1998.

[18] J. Kalamatianos and D. R. Kaeli, "Predicting indirect branches via data compression," *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture, MICRO 31*, pp. 272–281, 1998.

[19] T. Li, R. Bhargava, and L. K. John, "Adapting branch-target buffer to improve the target predictability of java code," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 2, pp. 109–130, 2005.

[20] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "Vpc

prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," *ISCA-34*, 2007.

[21] A. Roth, A. Moshovos, and G. S. Sohi, "Improving virtual function call target prediction via dependence-based pre-computation," *ICS '99*, 1999.

[22] Z. Xie, D. Tong, M. Huang, X. Wang, Q. Shi, and X. Cheng, "Tap prediction: Reusing conditional branch predictor for indirect branches with target address pointers," *Proceedings of the 2011 IEEE 29th International Conference on Computer Design, ICCD '11*, pp. 119–126, 2011.

[23] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. IEEE Computer Society, 2005, pp. 394–405.

[24] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length predictors," *Journal of Instruction-Level Parallelism (JILP)*, 2006.

[25] A. Seznec, "A 64-kbytes ittage indirect branch predictor," *In the 3rd Championship Branch Prediction*, 2011.

[26] ——, "A new case for the tage branch predictor," *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44 '11*, pp. 117–127, 2011.

[27] L. Porter and D. M. Tullsen, "Creating artificial global history to improve branch prediction accuracy," *ICS-23*, pp. 266–275, June 2009.

[28] D. R. Kaeli and P. G. Emma, "Improving the accuracy of history-based branch prediction," *IEEE Transactions on Computers*, vol. 46, no. 4, 1997.

[29] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, Mar. 2008.

[30] J. Mccandless and D. Gregg, "Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan 2012.

[31] M. Tan, X. Liu, T. Tong, and X. Cheng, "Cvp: an energy-efficient indirect branch prediction with compiler-guided value pattern," *Proceedings of the 26th ACM international conference on Supercomputing, ICS '12*, pp. 111–120, 2012.

[32] M. Farooq, L. Chen, and L. K. John, "Value based btb indexing for indirect jump prediction," *HPCA*, 2010.

[33] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, July 1987.

[34] A. Podgurski, "Significance of program dependences for software testing, debugging and maintenance," Ph.D. dissertation, Computer Sc. Dept., U. of Mass, 1989.

[35] A. Podgurski and L. Clarke, "A formal model of program dependencies and its implications for software testing, debugging, and maintenance," *IEEE TSE*, vol. 16, pp. 965–979, 1990.

[36] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communication of the ACM*, vol. 20(7), pp. 504–513, 1977.

[37] B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices, issta," pp. 66–79, 1994.

[38] D. E. Denning, *Cryptography and Data Security*. Addison-Wesley, 1982.

[39] M. Bishop, "Computer security: Art and science," vol. 2, December 2002.

[40] S. Kachigan, "Statistical analysis: An interdisciplinary introduction to univariate and multivariate methods," 1986.

[41] S. Siegel, *Nonparametric Statistics For The Behavioral Sciences*. NY: McGraw-Hill, 1956.

[42] D. J. Krus, "Visual statistics: Chapter 14, regression on multiple categories, visual-statistics.net," 2006.

[43] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," ser. PLDI '05, 2005.

[44] X. Gao, M. Laurenzano, B. Simon, and A. Snavely, "Reducing overheads for acquiring dynamic memory traces," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, 2005, pp. 46–55.

[45] A. Jaleel, R. Cohn, C. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proc. of the The Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2008, pp. 28–36.

[46] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009, pp. 469–480.

[47] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 146–160.

[48] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," *In Proceedings of 2007 International Symposium on Performance Analysis of Systems and Software (ISPASS2007)*, pp. 23–34, 2007.

[49] D. B. Papworth, "Tuning the pentium pro microarchitecture," *IEEE Micro*, vol. 16, pp. 8–15, April 1996.

[50] "The Standard Performance Evaluation Corporation (SPEC). The SPEC Benchmark Suite," http://www.spec.org.

[51] Y. Sazeides and J. E. Smith, "The predictability of data values," *Micro-30*, December 1997.

[52] B. Goeman, H. Vandierendonck and K. De Bosschere, "Differential fcm: Increasing value prediction accuracy by improving table usage efficiency," *HPCA'01*.

[53] P. Chang, E. Hao, and Y. Patt, "Target prediction for indirect jumps," in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA '97, 1997.

[54] Y. Sazeides and J. Smith, "Implementations of context-based value predictors," ECE-TR-97-8, University of Wisconsin-Madison, Tech. Rep., 1997.

[55] "Intel itanium architecture software developers manual," January 2006, document Number: 245319-005.

[56] Q. Zhao and D. J. Lilja, "Static classification of value predictability using compiler hints," *IEEE Transactions on Computers*, vol. 53, pp. 929–944, Aug 2004.

[57] H. Vandierendonck and K. D. Bosschere, "Implicit hints: Embedding hint bits in programs without isa changes," *ICCD*, pp. 364–369, Oct. 2010.