Compiler-Assisted Value Correlation for Indirect Branch Prediction

TAN Mingxing\(^1,2\), LIU Xianhua\(^1\), ZHANG Jiyu\(^1\), TONG Dong\(^1\) and CHENG Xu\(^1,2\)

(1. Engineering Research Center of Microprocessor and System Ministry of Education, Peking University, Beijing 100871, China)

(2. Shenzhen Graduate School of Peking University, Shenzhen 518055, China)

Abstract — Indirect branch prediction is important to boost instruction-level parallelism in modern processors. Previous indirect branch predictions usually cannot achieve high performance for the ineffectiveness of correlated information. This paper proposes the Compiler-assisted value correlation (CVC), a hardware/software cooperative indirect branch prediction scheme. The key is to identify effective value correlation based on program substructures. A compiler algorithm is introduced to identify the effective value correlation based on three program substructures: virtual function calls, switch-case statements and function pointer calls. The compiler-identified value correlation is transferred to the dynamic predictor by extending the instruction set architecture. At runtime, the processor relies on a low-complexity Correlated value buffer (CVB) to maintain the compiler-identified value correlation and to guide the target address prediction for those indirect branch instructions. Our evaluations show that CVC prediction can significantly improve the performance with little extra hardware support over the traditional BTB predictor and the state-of-the-art VBBI prediction.

Key words — Processor, Indirect branch prediction, Compiler-assisted value correlation, Program substructure.

I. Introduction

Indirect branches are widely used in modern applications\(^3\), so an accurate indirect branch prediction is important for modern processors. Unfortunately, indirect branches are hard to predict because each indirect branch may correspond to multiple branch targets\(^{1,2}\). As a special type of indirect branches, function returns can be predicted using the return-address-stack\(^5\), but other indirect branches used in virtual function calls, switch-case statements, and function pointer calls are hard to predict. In this paper, indirect branch only refers those non-return indirect branches.

Correlation-based predictors\(^{1,4−6}\) are the most widely studied in indirect branch prediction. Previously proposed indirect branch predictors mainly use the history information to hint the branch prediction. These predictors only take use of the control-flow information, which is usually inefficient for indirect branches. Recently, Farooq et al.\(^5\) propose the Value based BTB indexing (VBBI) prediction, which uses date values as the correlated information to guide branch prediction. Results show value correlation is more effective than traditional history information. However, it heavily relies on the profiling phase to identify value correlation, which makes it much less attractive in practice. Furthermore, it can only treat source-level variables as the value correlation, leaving out a large number of optimizations when indirect branches are correlated to intermediate variables rather than source-level variables.

This paper proposes the Compiler-assisted value correlation (CVC) for indirect branch prediction. The key of CVC prediction is to identify the effective value correlation based on three widely used program substructures: virtual function calls, switch case statements and function pointer calls. Program substructures contain the high-level control-flow and data-flow information, which enables the predictor to makes use of both the source-level and intermediate value correlation. The compiler-assisted value correlation is identified at compile time and transferred to the predictor by extending the Instruction set architecture (ISA). At runtime, the processor uses the compiler-assisted value correlation to predict indirect branch targets by slightly extending the traditional Branch target buffer (BTB). We show that the compiler-assisted value correlation can significantly improve the prediction accuracy and performance with little extra hardware cost.

This paper is organized as follows: Section II discusses related work; Section III introduces the compiler support and Section IV introduces the hardware design of CVC prediction; Section V shows results and Section VI concludes.

II. Related Work

Traditional branch predictors are usually pure hardware schemes. Lee et al.\(^7\) firstly proposed the Branch target buffer (BTB), which always predicts the branch target as the recently executed target of the same branch instruction. It behaves low
accuracy when the actual target alternate frequently. To improve the prediction accuracy, Change\textsuperscript{[5]} proposed the Tagged target cache (TTC), which uses branch history to guide indirect branch prediction. Subsequently, a large number of history-based predictors were proposed, such as the rehashable BTB predictor\textsuperscript{[8]}, the cascade predictor\textsuperscript{[6]}, the Virtual program counter (VPC) predictor\textsuperscript{[11]}. Besides the history-based correlation, value correlation is also widely used in branch prediction. Heil et al.\textsuperscript{[9]} firstly pointed out that control flow information is inefficient for some branches and they proposed to correlate these branches to data values. Branch predictors like dynamic data dependence tracking\textsuperscript{[10]} and dynamic dataflow identification\textsuperscript{[10]} also use data values to improve prediction accuracy. Recently, Farooq et al.\textsuperscript{[5]} proposed the VBBI predictor, which uses value correlation to guide indirect branch prediction, but it heavily relies on the profiling and is still inefficient.

Pure hardware predictors usually require significant hardware cost and are still ineffective because they are lack of high-level program information. To make use of high-level information, a number of hardware/software cooperative branch predictors are proposed. Deitrich et al.\textsuperscript{[11]} proposed some heuristics to utilize various program substructures, such as call, loop header, pointer, return, etc. Driesen et al.\textsuperscript{[12]} proposed to improve indirect branch prediction with source- and arty-based classification. Joao et al.\textsuperscript{[2, 13]} proposed the Dynamic Indirect branch Prediction (DIP), which dynamically predicates some indirect branches if they are hard to predict. It relies on the compiler to identify the control merge point for dynamic prediction. Recently, Lu et al.\textsuperscript{[14]} proposed a software branch hinting mechanism for cell processor. Tan et al.\textsuperscript{[15]} proposed a compiler-guided history stack for conditional branch prediction. These methods try to make use of static program information to enhance dynamic branch predictors.

III. Compiler-Assisted Value Correlation

Value-based predictors use data values to distribute different branch targets of the same indirect branch instruction. Fig.1 shows the conceptual structure of a value-based predictor. It uses the hash result of the branch address and data value to guide the predictor, so the predictor will predict different targets according to different data values for the same indirect branch instruction. In this paper, the data value information used to distribute different targets of the same branch instruction is called value correlation. Fig.1 has shown that proper value correlation is critical in value-based branch predictors.

Previous value-based predictors usually rely on significant hardware mechanism\textsuperscript{[16]} or expensive profiling\textsuperscript{[5]} to mark the value correlation. For example, dynamic dependence tracking mechanism\textsuperscript{[16]} uses a large Data dependence table (DDT) and a complex Register set extractor (RSE) to tracking data dependence for branch prediction, while VBBI predictor heavily relies on the profiling phase to identify correlated data values. These mechanisms are much less attractive to modern processors. Through analyzing those specific program substructures, we observed that the high-level control-flow and data-flow information has implied which value is strongly correlated to indirect branch and how to use the data value to predict branch target. As a result, we propose to automatically identify effective value correlation at compile time without complex hardware mechanism or expensive profiling support.

In this section, we will introduce how to identify effective value correlation and how to transfer the compiler-assisted information to the dynamic predictor.

1. Identifying value correlation

Indirect branches are usually used in three program substructures: virtual function calls, switch-case statements and function pointer calls. Specifically, the compiler identifies the value correlation based on the following principles:

- Virtual function call is usually implemented as a serial of load operations followed by an indirect branch\textsuperscript{[17]}. According to the object model in object-oriented languages (Java/C++/C#), each virtual object contains a virtual table pointer. All virtual function pointers are stored in this virtual table. For a virtual function call, it first loads the virtual table from the corresponding object header; then it loads the function pointer from the virtual table; at last, it jumps to the target virtual function with an indirect branch instruction. Each virtual object corresponds to a single virtual table, which keeps constant after the virtual object is created. In other words, the virtual table pointer value can definitely decide all the virtual function pointer value for a specific virtual object. Based on this observation, we mark the virtual table pointer value as the value correlation for those indirect branches used in virtual function calls.

- Switch-case statement can be implemented with conditional branch or indirect branch. Generally, when the number of the case blocks exceeds a threshold, the compiler will transform the switch-case statement into an indirect branch, avoiding too many conditional branches. In such a case, all case block addresses will be stored in a case table and accessed with an indirect branch instruction. For a given switch-case statement, the compiler firstly normalizes the case variable to a serial of continuous values such as 0, 1, 2, · · ·; then, it loads the corresponding case block address from the case block table at the index of the normalized case value; at last, it jumps to the corresponding case block with an indirect branch instruction. The normalized case value can definitely decide which case block will be used, so it has strong correlation with the final indirect branch. Based on this observation, we mark the normalized case value as the value correlation.

- Function pointer call is usually implemented as a load instruction followed by an indirect branch. The target of function pointer call is completely decided by the pointer value, so it is straightforward for compiler to identify the pointer value as the value correlation. In case that the function pointer call exists as an element of another object, the compiler can also treat the corresponding object address as the value correlation.

We show that the compiler automatically identifies the ef-
fective value correlation without profiling. Furthermore, our mechanism can make use of intermediate value information such as the virtual table value and the normalized case value, so the value correlation is more effective than previous source level or register level value correlation.

2. Transferring the compiler-assisted information

We extend the Instruction set architecture (ISA) to transfer the compiler-assisted information. Specifically, a new instruction, called guiding instruction, is added to the ISA. Fig. 2 shows the guiding instruction format. The “opcode” is a reserved opcode in original ISA and “ra” is the number of register containing the correlated data value. The “offset” and “s” bit indicate the signed distance from a guiding instruction to the corresponding indirect branch instruction. Note that the ISA extension may raise the binary compatibility issue, which can be addressed through decoupling the architected ISA and the physical ISA with binary translation system[18].

![Fig. 2. Guiding instruction formation](image)

Compared with previous approach that augments indirect branch instruction with hint information[5], our explicit guiding instructions allow us to track multiple control-flow execution paths by inserting multiple guiding instructions for each one indirect branch. Fig. 3 shows an example that illustrates the difference between hint instructions and guiding instructions. Assuming the indirect branch has three execution paths: def1 with probability 0.3, def2 with probability 0.2 and def3 with probability 0.5. Since the hint instruction can mark at most one value correlation for each indirect branch, it catches at most 50% of all value correlations. On the contrary, our mechanism can catch all value correlations by inserting multiple guiding instructions for the single indirect branch.

IV. Hardware Design of CVC Predictor

CVC predictor is based on traditional BTB predictor. Fig. 4 shows the hardware structure of our CVC predictor. The only extra component is the Correlated value buffer (CVB), which maintains the compiler-assisted value correlation. CVB is a cache-like hardware structure and can be implemented as direct-map or way-associative, tagless or tagged. In our simulation, we implement a 16 entries full-associative CVB.

The CVB is read by indirect branch instructions in the fetch stage and updated by guiding instructions in the retire stage. When a guiding instruction is retired, it immediately reads out the correlated data value (cvalue) from the Regfile. At the same time, the CVB index is computed by adding the signed offset to the indirect branch instruction PC. Then the processor looks up the CVB and writes the cvalue into the corresponding entry, otherwise a new entry is allocated according to the first-in-first-out replacement policy, and the allocated entry is initialized as cvalue. Subsequently, when the corresponding indirect branch instruction is fetched, the processor reads out the correlated data value from the CVB and then generates the predicted target address from the BTB at the index computed by hashing the correlated value and the indirect branch instruction address. Note that each indirect branch accesses the CVB at the index of its PC, but each guiding instruction accesses the CVB at the index computed by adding the “signed offset” to its PC. In other words, a guiding instruction and its corresponding indirect branch instruction access the CVB at the same index.

![Fig. 4. Hardware structure of PSA prediction](image)
the target overriding, so the total size of the CVB structure is 528 bits. Note that CVC predictor stores all indirect branch targets into the traditional BTB structure, so no large extra storage is required. As a result, CVC prediction can be implemented in modern processors with little extra hardware storage.

V. Experimental Result

We extend SimpleScalar/Alpha\cite{19} simulator to evaluate our CVC prediction. Table 1 shows the parameters of our baseline processor. Benchmarks are selected from SPEC CPU 2000 INT suite and SPEC CPU INT 2006 suite\cite{20}. We choose those benchmarks in SPEC CPU INT 2000 and 2006 suites that gain at least 5% performance improvement with a perfect indirect branch predictor. All compiler supports are implemented in the GCC-4.2 toolchain\cite{21}. Final binaries are compiled with -O2 optimization level.

Table 1. Baseline processor configuration

<table>
<thead>
<tr>
<th>Performance parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front end</td>
<td>4 instruction per cycle; fetch ends at the first predicted taken branch or indirect branch.</td>
</tr>
<tr>
<td>Execution core</td>
<td>4-wide decode/issue/execute/commit; 512-entry ROB; 128-entry LSQ.</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>12kB hybrid prediction (8k-entry bimodal, 8k-entry selector, 32k-entry gshare); 4k-entry; 4-way BTB; 32-entry return-address-stack; 15 cycle min. branch misprediction penalty.</td>
</tr>
<tr>
<td>On-chip caches</td>
<td>16kB, 4-way, 1-cycle L1 D-cache; 16kB, 2-way, 1-cycle L1 1-cache; 1MB, 8-way, 10-cycle unified L2 cache; All caches have 64B block size.</td>
</tr>
<tr>
<td>Memory</td>
<td>150-cycle average memory latency.</td>
</tr>
</tbody>
</table>

1. Distribution of indirect branch targets

Indirect branches are hard to predict because each indirect branch instruction may correspond to multiple dynamic targets. Fig.5 shows the distribution of the number of dynamic targets across executed indirect branches. Almost half indirect branches have more than 5 dynamic targets. The multiple dynamic targets for each indirect branch invalidate traditional taken/not-taken direction predictors for conditional branches.

2. Performance

We compare our CVC prediction with the baseline BTB predictor\cite{7} and the VBBI predictor\cite{5}. To the best of our knowledge, VBBI predictor\cite{5} is the state-of-the-art indirect branch predictor. Fig.6 shows the performance of different indirect branch predictors. On average, CVC prediction improves the prediction accuracy by 32% over the BTB predictor and by 14% over VBBI predictor, which results in the performance improvement of 21% and 5%. Results show that CVC prediction can significantly improve the performance over previous indirect branch predictors.

Compared with VBBI predictor, CVC prediction performs better for three reasons. First, VBBI predictor relies on the manual profiling to identify proper value correlation, so the training inputs have strong impact on the effectiveness of the value correlation, while our CVC predictor relies on the compiler to automatically identify the value correlation based on program substructures. Second, VBBI predictor can only identify the source-level variables as the value correlation, while our CVC predictor can make use of those intermediate variables, such as the virtual table address and the normalized case value. Third, VBBI predictor can only track single correlated variable for each indirect branch, while our CVC prediction can track multiple correlated variables by inserting multiple guiding instructions for a single indirect branch instruction.

Note that the extra guiding instructions also hurt the performance in some programs. For example, VBBI and CVC predictor achieves similar prediction accuracy in the program perlbench, but CVC predictor performs worse than VBBI in this program because of the extra guiding instructions.

3. Sensitivity to microarchitecture parameters

CVC prediction is sensitive to BTB size, because it stores...
all indirect branch targets into the BTB structure. Fig.7 shows the impact of different BTB sizes on CVC prediction. The BTB structure is always configured as tagged and 4-way associative. Results show that the performance improvement increases as the BTB size increases. However, CVC prediction still achieves significantly performance improvement even with small BTB sizes.

CVC prediction is also sensitive to the pipeline stages and issue widths. We evaluate our CVC predictor on a less aggressive processor, which is configured as 2-issue and 8-stage. The minimal branch misprediction penalty is 3 cycles. The other configuration parameters of the less aggressive processor are similar to Table 1. On average, CVC prediction achieves the prediction accuracy of 93%, which is roughly competitive to the aggressive baseline processor. However, it only improves the performance by 7.5% over the baseline BTB predictor and by 2.7% over VBBI predictor. It shows that accurate indirect branch prediction is more substantial in aggressive processors.

VI. Conclusion

This paper proposes the CVC prediction, a hardware/software cooperative indirect branch prediction. The key idea is to identify the effective value correlation based on high-level program substructure information. Program substructure implies which value is strongly correlated to the branch target, thus improves the effectiveness of the correlated information. CVC prediction relies on both compiler support and hardware support to utilize the value correlation. With the help of the compiler-assisted assisted value correlation information, CVC prediction can achieve higher prediction accuracy and better performance with little hardware storage support.

Our future work is to reduce the side effect of target overriding in CVC prediction. The target overriding requires two predictions for each indirect branch instruction and flushes all instructions fetched between the two predictions; thus increases the BTB accesses and complicates the hardware design.

References


TAN Mingxing is a Ph.D. candidate in the School of Electronics Engineering and Computer Science at Peking University, Beijing, China. His research interests include microprocessor design and compiler optimization. (Email: tanningxing@mprc.pku.edu.cn)

LIU Xianhua is a lecturer in the School of Electronics Engineering and Computer Science at Peking University, Beijing, China. His research interests include computer architecture, and compiler optimization. (Email: liuxianhua@mprc.pku.edu.cn)