Fast and Accurate Branch Predictor Simulation

Antoine Faravelon, Nicolas Fournel and Frédéric Pétrot
TIMA Laboratory, Université de Grenoble-Alpes/CNRS

***Abstract*—The complexity of embedded processors has raised dramatically, due to the addition of architectural add-ons which improve performances significantly. High level models used in system simulation usually ignore these additions as the major issue is functional correctness. However, accurate estimates of software execution is sometimes required, therefore we focus in this paper on one of theses architectural features, the branch predictor. Unfortunately, advanced branch predictors use large tables, so that models directly implementing these schemes slow down simulation dramatically. To limit the simulation overhead, we define a modeling approach that we demonstrate on a state of the art predictor. We implemented the model in a dynamic binary translation based instruction set simulator and measured an accuracy of prediction of about 95% for a run-time overhead of less than 5%.**

I. Introduction

Most of the processor models in fast system simulation technologies simply ignore micro-architectural details. When searching for performance accuracy, handling some pipeline dependencies has been proposed [1], but usually, the approach consists in adding an average statistical overhead in the time estimates. Branch prediction is an essential part of highly pipelined and superscalar processors. Its accuracy greatly impacts the efficiency of a processor, as a misprediction causes fetching and execution of instructions which, in reality, are not needed by the program, and at the end need to be undone. To give a reasonable evaluation of the execution time of a program on a given target MPSoC, modeling branch prediction is required. This paper aims at defining a modeling strategy for branch predictors, and takes as example the state-of-the art TAGE[2] predictor. The main difficulty with simulating branch prediction is to avoid slowing down the simulation, as it uses large tables to maintain per branch confidence state information and guess the branches outcome. Even a simple gshare[3] with a 16 bit long branch history requires a 2^{16} entries wide counter table. With a memory wise perfectly optimized program, it represents 2^{17} bit if the table contains 2 bit counters. Considering that entries are not accessed sequentially and that the tables are large, naïve modeling of branch predictors will result in an important number of cache misses and decrease simulation performance for any complex predictor. Branch predictors should then be simulated using a method taking memory constraints into account, *i.e.* the approximate model should be memory efficient while keeping reasonable accuracy.

The paper is organized as follows. We present in Section II the minimal background knowledge and define the problem at hand. Section III-A detail our proposal, while Section V presents and discusses the accuracy and run-time results. Section VI summarizes the work and concludes.

II. Preliminaries and Problem Definition

Branch prediction consists of guessing the outcome of a branch in a program. Prediction is based on history, *i.e.* the previous outcomes, to predict the future. The simplest predictor just predicts the last outcome as the current one, using the lower bits of the program counter to index a table. Guessing that a branch is taken while it is actually not taken, and conversely, is called a misprediction. State of the art branch predictors are using much more complex strategies, and rely on large tables to maintain history and confidence statistics, resulting in low misprediction rates[1]

Waiting to know the actual outcome of a branch is not acceptable in superscalar processors, so branch prediction plays a major rôle in keeping the functional units busy. The cost of a misprediction is around 20 cycles for modern high performance processors. Even though it may be less for the less aggressive high end embedded processors, it does have a major influence on the execution time of applications. Thus, it is important to take this architectural feature into account when using system simulation for early software development.

As branch predictors use large tables and index them irregularly, we fear that their simulation models will experience high cache miss rates. Fast processor simulation models are nowadays based on dynamic binary translation, which splits the target code at basic block's boundaries and uses an *ad-hoc* run-time to compute the next basic block to execute. This fits well with our topic as these boundaries match with the occurrence of branches. We therefore aim at defining a strategy for modeling branch predictors which has limited impact on the simulation performance within fast simulator based on dynamic binary translation.

III. Proposition

To illustrate our proposal, we take as example the TAGE predictor family.

A. Principle

The main idea is to transform the tables of the reference TAGE architecture, which are global, *i.e.* shared between all branches, into data local to every branch. This allows to allocate entries only as they are needed, and intuitively, we

[1]The best teams compete during the "Championship Branch Prediction", whose 4th occurrence took place in 2014 in conjunction with ISCA-41 (`http://www.jilp.org/cbp2014/`).

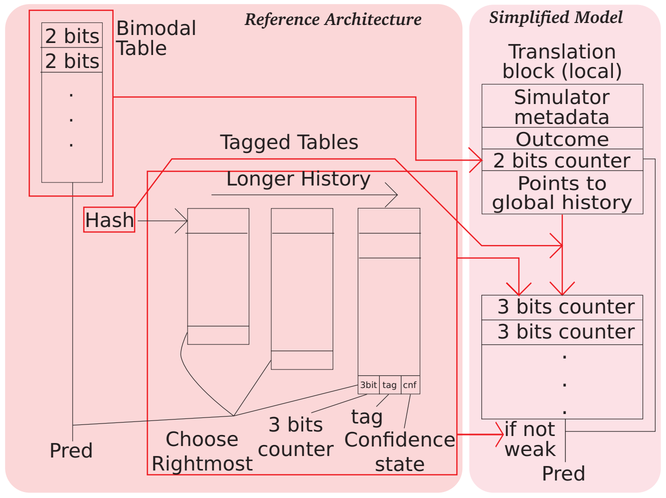

Fig. 1. Reference Architecture and Simplified Model

believe this will enhance locality and limit the number of cache misses when simulating the simplified model.

That idea in mind, all elements of the reference architecture were processed from the simplest to the most complex. The bimodal table, as can be seen on Figure 1, is reduced to a 2 bit counter which can be stored in the local data for each branch. Then, for the tagged tables, two parts are distinguished, the tables themselves and the way they are indexed. For the tables, following the same principle, only one was implemented at first. In the same way as for the bimodal table, only the potentially reachable entries, *i.e.* the ones concerning already seen branches, are allocated locally for each branch. This results in a local table that is indexed by the global history, which, in the reference architecture, was hashed with the program counter (PC) of the branch.

The choice of the entry is then naturally simplified as there is only one, local, tagged table to choose from. To choose between tagged table and bimodal table, we look at the confidence state of the entry in the tagged table, if it is "weak", then the prediction will be given by the bimodal table, otherwise it will be given by the tagged one. This should ensure that, as in the reference architecture, tagged prediction is given only if the tagged entry is "sure" of its prediction. The tag of the tagged table is not useful anymore, as entries concerning one branch are now specific to it, in other words, aliasing, *i.e.* multiple branches accessing the same entry, is not possible in the simplified model.

The resulting model uses memory local to the current basic block, which tends to enhance locality, but it also uses more memory, as information shared due to aliasing is now duplicated. The information used by the simplified predictor is similar to that of the reference architecture: the outcome of the currently executed branch (or at least the condition so as to be able to compute it), its own data, a global history of branch outcomes and, finally, the structure of the already executed code (to store data per branch). All of these can be retrieved from a simulator.

B. Analysis

The main consequence of the structure used for the simple model is the disappearance of aliasing. All the information is local to a branch, including counters (entries) which can then only be accessed by this branch. This would give a slight advantage if we were looking for doing better prediction, but our goal is to do the same (good or bad) prediction, so we need to quantify this difference.

Assuming that indices in the table, computed using the lower bits of the branch addresses, are uniformly distributed,the probability of having two branches share the same bimodal table entry is $\frac{nb\_of\_branches}{table\_size}$, $nb\_of\_branches$ being the number of executed branches. If $table\_size$ is big enough, this difference will be small. For the Tagged Tables, which have different sizes, the probability that the predictor tries to access the wrong entry is the same. But in each entry there is a tag which should match the current branch. The value seems to be uniformly distributed and as such the probability to have one value in the set $[0..2^{tag\_size}]$ is $\frac{1}{2^{tag\_size}}$. Computation of tag being independent, the probability that two branches get the same tag is $\frac{1}{2^{tag\_size}} \times \frac{1}{2^{tag\_size}} = \frac{1}{2^{2\times tag\_size}}$. Tagged table aliasing probability in reference architecture is then $\frac{nb\_of\_branches}{tagged\_table\_size} \times \frac{1}{2^{2\times tag\_size}}$. Again, when $tagged\_table\_size$ and tag_size have big enough values, collisions occur seldom, and can be neglected at first order. Also, it should be noted that the reference architecture implements hysteresis bits sharing in its bimodal table while our model does not, which should contribute slightly to a higher prediction accuracy though, according to [4], it should be moderated.

IV. Related Works

Some architectural studies have relation to this work. They start from a full featured predictor, and perform successive degradations to get a simpler architecture using less resources [5]. These efforts cannot use local data, as they are really architectures, and we take somehow the opposite direction, by adding more complex elements starting from the simplest ones.

As far as simulation is concerned, compared to existing work, this approach is usable in a real DBT simulator in an early design perspective, thanks to its ease of implementation and efficiency. Indeed, as one can see in [6], most work in simulation either advertises no branch prediction simulation or only simplistic approaches such as the one in [7] and [8] which sort branches according to a predicted state (taken, not taken, unknown or a slightly more refined cut for the second one), giving then a bound on execution time depending on which category the branch is in. Other works are based on statistical models, such as [9] and [10]. Static analysis though is not adapted to DBT as this would slow down code generation dramatically. As for statistical models, they do not really model a predictor in any precise way but only reflects a rate of prediction in a training set though they do give good results in term of same prediction rate. Furthermore, the definition of the training set and the runtime of a training phase would take too much time to be practical for an early design phase. Only the Chronos simulator advertised dynamic simulation for "popular" predictors [11]. But after investigations, it appeared that, in its actual form, this simulator is based on static analysis and uses target binary version of the program only to extract the runtime addresses of branches.

V. Experiments

A. Implementation

1) DBT simulator: Our simplified model was added in the RABBITS [12] simulation framework, which uses Qemu as processor model. This simulator already provides metadata for every translated basic block it has gone through. Considering that there is one translated basic block per branch, this allows to add the simplified model's own metadata to the pre-existing structures. This implementation allows to evaluate this work in a realistic environment, thus measuring its impact on execution time within a full featured DBT simulator.

2) cbp3 simulator: The reference architecture was already implemented in the simulator of the Championship Branch Prediction of the Journal Of Instruction Level Parallelism. After implementing the simplified model into it, this simulator along with the official traces provided with it were used to evaluate the precision of the simplified model, defined as the percentage of predictions which are the same for both models. It is also used to compare this simulator precision with the TAGE reference implementation and with other predictors.

B. Results

1) Precision: Precision is an important feature. The higher it is, the closer the estimate of performances will be to reality.

As can be seen on Figure 2, precision is above 80% for all

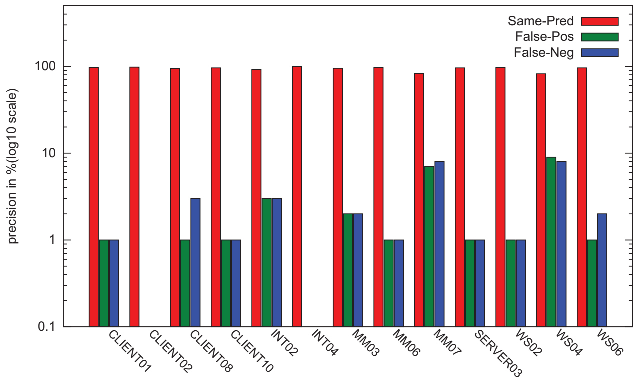

Fig. 2. Precision of the simulator

traces, the minimum being reached on the trace WS04 with around 82% of predictions being the same for both models. In most traces, precision is much higher, average on this set is around 94,5% and it features all worst cases for our simplified model.

2) Performances: Two main sets of results will be presented in this section, the execution time and number of L2 cache misses on RABBITS alone and with each model.

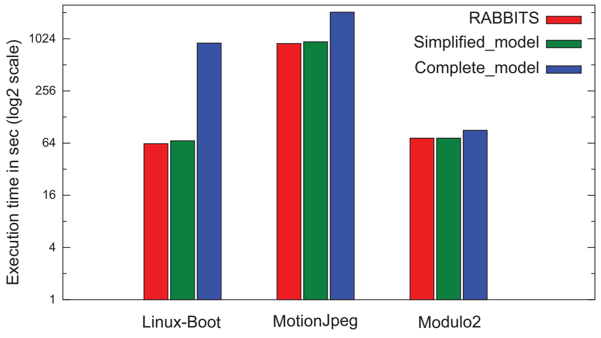

Fig. 3. Execution time on RABBITS alone and with each model.

Execution time for RABBITS with the reference architecture of the TAGE range between 1,5 and 15 times that of RABBITS alone. The Linux boot sequence takes about 15 minutes on our test platforms instead of 1 minute for RABBITS alone. Our model performs much better, execution time raises by only about 5% as Figure 3 shows. This difference in performance can be explained by the study of cache misses in each case. Indeed, when looking at Figure 4, we observe that RABBITS with TAGE reference architecture suffers of more than 19 times more cache misses than alone on Linux boot sequence. This confirms our intuition that the naïve approach will be limited by memory and, in particular, by caches capacity. The simpler model which was designed with these limits in mind only causes twice as much cache misses in the worst case.

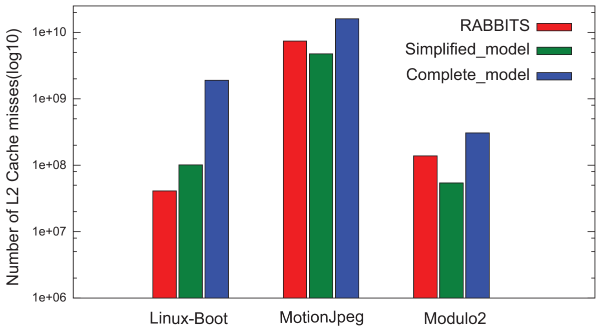

Fig. 4. Number of L2 cache misses

3) Comparison to other predictors: When looking at the simple model comparison with ISL-TAGE (reference architecture), a 18bit gshare and PC-indexed bimodal table (see Figure 5), it is first obvious that our model is quite far from the bimodal table. Indeed, the percentage of identical predictions is less than 50%. On the other hand, it is closer to the gshare, but so is the TAGE in fact. The trace that should mainly be observed is WS04. This trace which, according to our experiments, seems to rely heavily on TAGE-like history management. Our simple model is quite closer to the TAGE reference architecture than to the gshare one. This confirms

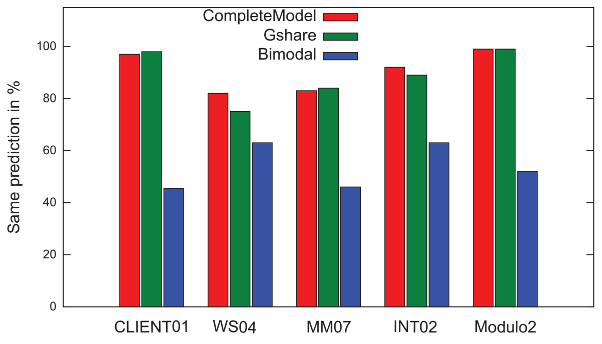

Fig. 5. Ratio of identical predictions between our simplified model and actual predictors

that our method is optimized for the predictor we aimed at modeling. It shall be noted that it is not a generic model, and that the behaviour reproduced is the one of the TAGE family.

VI. Conclusion and Future Works

A. Analysis of this work

In this work, we have shown that a simplified model of the TAGE and potentially any predictor can achieve high accuracy. This model, being designed and implemented with memory efficiency in mind, allows simulation to perform better than a direct implementation of the reference architecture. Identical prediction rate was overall quite satisfying compared to the most complex existing version of the reference architecture at the time of this research. The model is however not a generic model, but the result of an analysis of the target predictor architecture. This means that its behaviour will be close to that of the original predictor, as opposed to being statistically close.

However, the needed approximation, or simplification, removes the ability of the simplified model to acquire some precise details, at least in its current state. For example, attempt to add Loop Prediction as described in [13] results in the simple model being less precise. Implementation of other details such as confidence in predictions also gives negative results.

In addition to its accuracy, the other advantage of this simplified model lies in the ease of implementation. The information required by it are close to that of the simulator, both in content and structure. Both the simulator and model uses data that are local to each translated block/branch. Thus, already existing metadata can be reused and no new data structure needs to be added, as missing elements required by the branch predictor's simplified model can be included in the existing ones. This result in our model's implementation taking less than 200 lines of C code while the reference one requires around 1000.

Thanks to the data being now local and to the simplification of the predictor, the use of fewer tables in particular, much less cache misses are caused by our model than by the reference one. It results in the simulator's execution time overhead being greatly reduced, less than 5% for our model against up to 15 times longer execution with the reference one.

B. Future Work

While this work offered satisfying results, work could be done to improve the simulation of tagged table. Simulating multiple ones first should lead to higher accuracy. Adding aliasing though would be impractical as this would lead, just as in the reference architecture, to many memory accesses which would in turn cause cache misses and slow down execution of the simulation. On the other hand, implementation of the simulator in native simulation could lead to increase in both precision and performance. By computing dependency of one branch to previous branches, one could know which table should be used for this branch and, with the complete control flow graph, the exact or at least an estimate of the number of accessible entries for each branch. This would further reduce the impact on memory and allow to simulate more precisely the tables. Also, some predictions could probably be done statically, as some existing works point out, improving precision and performance at runtime.

References

[1] C.-K. Lo, L.-C. Chen, M.-H. Wu, and R.-S. Tsay, "Cycle-count-accurate processor modeling for fast and accurate system-level simulation," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*, March 2011.

[2] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-Level Parallelism*, vol. 8, february 2006.

[3] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[4] G. H. Loh, D. S. Henry, and A. Krishnamurthy, "Exploiting bias in the hysteresis bit of 2-bit saturating counters in branch predictors," *Journal of Instruction-Level Parallelism*, vol. 5, March 2003.

[5] P. Michaud, "A ppm-like, tag-based branch predictor," *Journal of Instruction-Level Parallelism*, vol. 7, April 2005.

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. old Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. S. lat, and P. Stenstrom, "The worst-case execution time problem — overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, April 2008.

[7] A. R. Xianfend Li and T. Mitra, "Modeling out-of-order processors for software timing analysis," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, december 2004, pp. 92 – 103.

[8] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Journal of Real time Systems*, vol. 18, no. 2-3, pp. 249–274, May 2000.

[9] B. Franke, "Fast cycle-approximate instruction set simulation," in *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, 2008, pp. 69–78.

[10] B. Franke and D. C. Powell, "Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2009, pp. 315–324.

[11] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for wcet analysis," *Real-Time Systems*, vol. 34, no. 3, pp. 195–227, 2006.

[12] M. Gligor, N. Fournel, and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible MPSoC simulation," in *Proceedings of the 7th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis*, 2009, pp. 71–80.

[13] A. Seznec, "The l-tage branch predictor," *Journal of Instruction-Level Parallelism*, vol. 9, July 2007.