

# A Replacement Policy Based on Dynamic Profiling and Hashed Information

Marios Kleanthous<sup>\*,1,4</sup>, Sami Yehia<sup>†,2</sup>,  
Yiannakis Sazeides<sup>\*,1</sup>, Emre Ozer<sup>‡,3</sup>

<sup>\*</sup> *University of Cyprus, 75 Kallipoleos Street, CY-1678 Nicosia, Cyprus*

<sup>†</sup> *THALES Research & Technology, RD 128 - 91767 Palaiseau cedex, France*

<sup>‡</sup> *ARM Ltd, 110 Fulbourn Road, Cambridge CB1 9NJ, United Kingdom*

---

## ABSTRACT

KEYWORDS: replacement policy, dynamic profiling

## 1 Introduction

Current superscalar general purpose processors invest in all kinds of ways to predict data in order to exploit ILP. For example, value predictors and branch predictors keep the history of previous instructions in order to correctly predict the data or outcome of upcoming instructions. Unfortunately, these structures are very large and complex most of the time.

Although these structures are big, only a subset of the entries in these structures is effectively useful because of non-regular data or very rarely used entries. We also observed that it is difficult to reduce the number of entries of these structures without degrading its prediction accuracy because smaller structures will cause useful entries to be evicted by other non useful one.

Embedded processors are unable to afford big and power hungry predictor structures. A cost efficient method is needed to select the best subset of instructions that will update a small predictor without the use of such big data structures. In other words, a new re-

---

<sup>1</sup>E-mail: {mklean,yanos}@cs.ucy.ac.cy

<sup>2</sup>E-mail: {sami.yehia}@thalesgroup.com

<sup>3</sup>E-mail: {emre.ozel}@arm.com

<sup>4</sup>This author wishes to thank HiPEAC and ARM for offering him the opportunity of an industrial Internship that resulted to this work.

placement policy must be considered that will take into account not only the timestamp (assuming a LRU policy) but also the performance gain of an instruction in order to select the best possible subset of instructions for updating.

Therefore, we propose a technique that exploits small structures using a profiling table that uses a number of entries at the same order of existing structures but with less information to determine the impact of each entry on the performance. Knowing the importance of such entries allows smaller structures to be updated with the best possible subset of entries.

## 2 Related Work

Prior work mainly proposes replacement policies that decide which entry in the predictor table will be replaced or take a decision if an entry will be replaced based on its performance using event counters [2, 3] or hysteresis counters. Event counters keep track of the performance of each entry based on different events like miss or hit. Hysteresis counters provide delay of the replacement of an entry to avoid premature replacement.

Furthermore, there are various techniques [4, 5] to filter the instructions that update the predictors. Such filtering reduces the pressure due to replacements and improves the prediction rate. Also confidence history is used in order to avoid replacing entries that deliver good predictions by other that have poor performance. Still, their method relies on having relatively big structures to achieve accurate predictions

Using a very small structure of 8 entries for example, it will be very difficult for the hysteresis counters to work since there will not be enough time to get hits and increase the counters.

We propose the use of a small predictor with a dynamic profiler. The profiler will have a number of entries at the same order of existing big structures but the size of the entry will be much smaller. The profiler will provide the decisions for replacement based on hashed information.

Also, our approach can achieve the filtering and confidence of unpredictable instructions by keeping track of the performance of each instruction using the profiler.

## 3 Profiling based replacement policy

Although the proposed solution can be used for different data structures we will use a simple Stride Value Predictor [1] as our baseline predictor to demonstrate the idea.

A Stride Value Predictor is accessed by a load instruction during the decode stage using the PC of the load instruction. If there is a tag match then the processor speculatively updates the destination register of the instruction and the prediction is checked for correctness on the writeback stage. If the loaded value matches the prediction then the value predictor is simply updated with the new value and stride. If the loaded value does not match the prediction then the predictor is updated and also the pipeline is flushed and instructions are reexecuted with the correct value. If we have a miss in the value predictor during the decode stage, a new entry corresponding to the new load instruction will be inserted in the value predictor on the writeback stage.

Studying the behaviour of load instructions and big value predictors, we discovered that usually a smaller percentage of those instructions are predictable and even a much smaller

percentage of those instructions needs to be in the predictor at the same time at any given period of time. Trying to use a very small value predictor, of 8 entries for example, was very difficult for the techniques mentioned before to work since there was not enough time to get hits and increase the counters that those policies are based on because of the continuous updates and replacements of the 8-entry table.

The previous observation raises the need of a cheap way to know the performance of the instructions trying to update the value predictor without keeping those instructions in the small 8-entry table that provides the predictions. In order to achieve this, we propose the use of a dynamic profile table with hashed information in combination with the small 8-entry fully associative value predictor. The profile table will provide all the information needed to decide which and when an entry will be inserted in the value predictor.

Figure 1 shows the proposed Profile Table along with its associated 8-entry predictor. The fields of the profiler are as follow:

- Hash value: Hash value is the last 3 bits of the loaded value plus the sign of the value.
- Hash stride: Hash stride is the last 3-bits plus the sign of the difference between the old hashed value and the new hashed value.
- Counter: 3-bit counter that indicates the performance of the instruction.
- Context: The context number is a unique number assigned heuristically to all loads that belong to the same contexts.

In order to detect and assign the context for an instruction we use a simple heuristic algorithm. Initially current context (Ccontext) is set to 1, and all entries of the profiler are invalid. When adding a new entry to the profiler, we assume that we are in Ccontext=1 as long as we are adding new entries in the profiler and it is assigned to all subsequent invalid entries. Now, if a valid entry whose context equal to the current context (Ccontext=1) is encountered then we assume that we are in the loop corresponding to the current context. This current context is maintained until a new invalid entry or an entry with a different context is encountered. At this point we assign a newly generated context number to these entries.

The use of hashed values along with the counter allows the profiler to approximately but very accurately calculate how much an instruction will contribute to the performance. For example, each time a load instruction reaches the writeback stage it tries to update the profiler. The hashed value plus the hashed stride are compared with the hash of the loaded value and if they are equal then the counter is updated with the number of stalls that the load caused. Otherwise, if the hashed information does not match then the counter is decremented by one (counter gets values between 0 and 7). The profiler is always updated with the new hashed value and hashed stride.

The behaviour of our value predictor during the prediction and recovery phase is identical to the simple stride value predictor [1] described at the beginning of this section. Our profiler comes to work only at the writeback stage when the value predictor is updated. Specifically, (a) if an entry corresponding to the load instruction is already in the value predictor then the profiler and the predictor are updated and (b) if an entry corresponding to the load instruction is not in the value predictor then only the profiler is updated and also decides whether and which entry in the value predictor should be replaced.

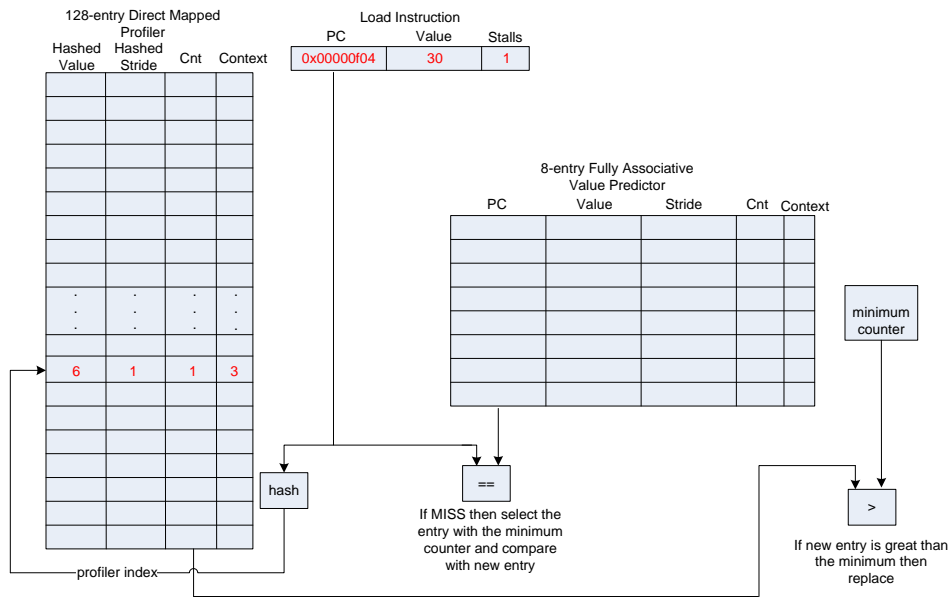


Figure 1: A 128-entry direct mapped profile table with an 8-entry fully associative Value Predictor

Once the profiler is updated a an entry has to be replaced in the value predictor the following steps are performed:

1. Access the value predictor and find a candidate entry for replacement
  - (a) If one or more entries in the value predictor do not belong to the current context, then from those entries, the entry with the minimum counter will be selected (A low value in the counter indicates that the load cause few stall cycles, thus has little impact on performance, or is hardly predictable).
  - (b) Else if all the entries in the value predictor belong to the current context then the entry with the minimum counter will be selected
2. Compare the counter and context of the candidate entry for replacement with the counter and context of the new load instruction
  - (a) If the candidate entry belongs to a different context than the current context then replace the entry with the new load instruction
  - (b) Else if the candidate belongs to the current context but has a smaller counter than the counter of the new load instruction, then replace the entry with the new load instruction
  - (c) Else if the candidate entry for replacement belongs to the current context and has also bigger counter than the new load instruction then do nothing

To sum up, the program properties and regularities that the Profile Table exploit to select the best subset with low cost are the following:

1. Using the last few bits of values and stride: can approximately predict if a loads was going to be correctly predicted or mispredicted
2. Using the context: can approximately detect loads within a loop and avoid predictor pollution caused by loads outside the current loop
3. Using the counters: can profile and remember the stalls caused by that load before, and avoid polluting the predictor with unpredictable loads.
4. No need of tag match in the profiler: rely on the fact that a big (128 entries) direct mapped table will avoid conflicts within the same context

## 4 Conclusions and Future work

In order to compare the performance improvement with previous techniques we still use the filtering techniques such as updating the value predictor only with load instructions and only those that caused a stall, for both the baseline and our proposed algorithm. We used a 5 stage pipeline model to collect information on the trace and detect potential removal of stall cycles in case of a correct prediction with out any timing simulation.

Fig. 2 shows the results using the percentage of stalls successfully removed, correctly predicted, from all the stalls caused during execution. The line "8 entries" correspond to an 8-entry fully associative value predictor using LRU, the line "128 entries + 3bit" correspond to an 8-entry fully associative value predictor using LRU plus a 3-bit counter for each entry to provide hysteresis during replacement. A hysteresis counter is set to maximum vallue when we have a correct prediction and is decrement on a misprediction. An entry cannot be replaced if the counter is not zero. Finally the "Selective 128" correspond to an 8-entry value predictor with the proposed dynamic profiler with 128 entries.

The results indicate that the potential performance improvement is significant when compared to the LRU policy (17% more stalls removed). Also the results show that the mechanism can aproximate the performance of the 128 entries fully associative predictor with just 20% of it's size.

For future work, an optimal replacement analysis has to be done to show how close we on the limit. Also experiments under a timing model to show how much we can improve the performance by removing 17% more stalls because of load instructions.

