Towards Economic Trace Caches: A Profile Based Approach

Saisuresh Krishnakumaran, Sai Arunachalam
{saisuresh, sai}@cs.annauniv.edu

Department of Computer Science and Engineering,
College of Engineering, Anna University,
Chennai, India.

Abstract

Trace caches have been effectively used as a solution to the problem of fetch mechanism bottlenecks in recent processors. Higher performance is gleaned by increasing the sizes of the trace caches, which brings along higher power consumption. However, in the wake of power economy, alternatives that better suit our needs have to be investigated. Towards this end, we propose a scheme that promises a higher performance with lesser overheads and the conventional trace cache size.

Our scheme lays out a profile-based approach to increase the efficiency of the trace cache. The application program is profiled on its first run and the resulting profile in used thenceforth for future runs, provided the run time environment does not change. The program is partitioned and the profiles seek to differentiate between the most important and the not so important traces in each of the subset. This information is judiciously used to achieve enhanced performance of the trace cache.

1 Introduction

With the superior capabilities of the present day super-scalar processors, a large number of instructions can be executed in a single cycle. However, instruction fetch mechanisms have to keep up with such processors if instruction level parallelism has to be exploited to the maximum. Branch prediction accuracy, instruction hit rate in the cache and dynamic execution sequences are among the factors that affect the fetch mechanism efficiency.

Trace caches [1] have been proposed as an effective solution to address the limitations of the fetch mechanism. These caches store the instruction basic blocks in the dynamic order in which they are retrieved during program execution. Each trace line stores a sequence of blocks, along with some contextual information such as the starting address of each block and the prediction bits for the trace in question. A multiple branch predictor [2] is used to predict the next ‘n’ branch outcomes during each fetch. The address of the starting block and the prediction bits are used to determine the presence of the required trace in the trace cache. If the trace is present, it is retrieved from the trace cache in a single cycle, else we resort to fetching instructions from the instruction cache.

While trace caches have helped in achieving better performance over conventional designs, there are some inherent limitations towards their use. Some of the issues are as follows:

- It has been witnessed that performance benefits can be reaped by use of larger trace caches [3], but this is discouraged by the corresponding increase in power consumption.
- Traces consisting entirely of completely contiguous blocks have a high probability of being present in the instruction cache lines. The presence of such traces in the trace cache leads to redundancy and space wastage.
- Owing to the property that 80% of the time is spent in 20% of the code, we might be able to do better than LRU, if we could use a more informed replacement logic, better performance benefits can be gleaned.

Our approach to this problem is to make use of a profile driven approach to design the replacement of traces in the trace cache. Each application program is collapsed into smaller partitions and each of these subsets is profiled in turn,

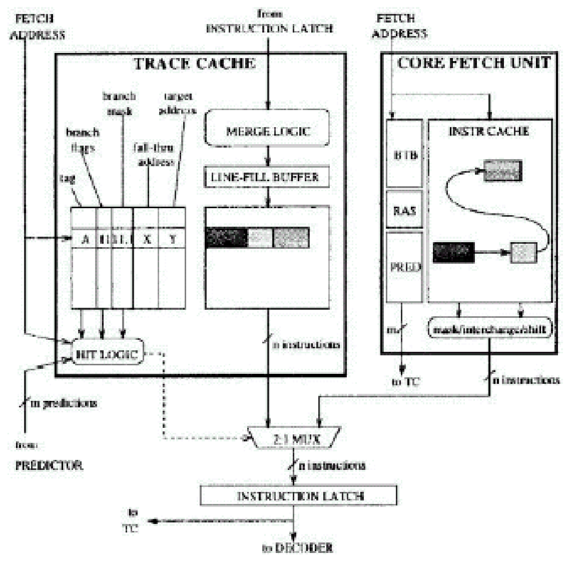

Figure 1. The Trace Cache [1]

increasing the locality in the sample stream. The primary function of the profiling process is to identify the most frequently used traces and coupling them with an importance factor, which is a function of their usage frequency. This process of branding the traces as important is split into two levels as follows. Each time a trace is brought into the trace cache, it is checked whether details pertaining to this trace are present in a profile table (introduced by the design). If no such details are found, the trace is placed in the trace cache and its usage counter is initiated to one. Thenceforth, each time the trace is hit its usage counter is incremented. Once its usage increases beyond a threshold t1, its details are written to an entry in the profile table. If, after this, the trace is evicted from the trace cache, its profile table entry is updated to the last value. If after some time, the same trace is brought into the trace cache, then these details are ported to the usage counter of the trace cache entry. When the usage counter value for any trace crosses a second threshold t2, its profile table entry is made irremovable but its usage counter continues to get updated. If at any point in time, there is a need for an entry in the profile table, then an entry whose usage is between t1 and t2 is chosen for eviction. It should be noted here that entries with usage counters whose values are greater that t2 cannot be removed. If the profile table gets filled with such entries, then we start denying the request for an entry each time it is made, and a counter for the number of denials is maintained. When this counter reaches 15 denials, we interrupt the processor, meaning to imply that the profile has matured and write the information to a memory file. In this way, the entire program is used for reaping information about the usage of traces in each of the partitions.

Once the entire program is profiled, the subsequent runs of the program use the profile information to determine the residency of traces in the trace cache. The replacement logic is based on the importance of the traces relative to each other. At times when the importance factors of two different traces are the same, superiority is established by the nature of the blocks in the traces. Traces that consist of taken branches later in the trace are considered as better candidates for replacement compared to those having earlier taken branches.

The overhead of profiling can be by means of a profiling co-processor [4], or support directly provided by hardware [5]. In this paper, we abstract the problem of profiling effectively and present only the design modifications in the architecture and the algorithm that can be used for the replacement logic.

This paper is organized as follows. Section 2 skims over the modifications and additions that might be required in the hardware. Section 3 explains the process of profile building and run time usage of the same. Section 4 discusses the perceived benefits followed by the conclusion in Section 5.

2 Design Issues

The proposed scheme would require certain modifications to the existing design.

2.1 Profile Table

In order to profile the traces, we would require a profile table where trace information is maintained while the subsets are monitored. The table should accommodate entries for at least as many traces as can be present in the trace cache at any given time. Each entry in the profile table should consist of the following fields:

- Starting block addresses
- Prediction bits corresponding to the blocks in the traces
- The importance factor corresponding to the trace usage frequency in each of the program subsets. Care should be taken to avoid wrap-around of this field.

- A Mask bit to indicate if the entry can be chosen for replacement

2.2 Trace Cache Modifications

The trace cache line has to admit a few modifications to take advantage of the proposed design. Each entry in the trace cache should include a field to maintain the pending usage. This would correspond to the importance factor the first time a profiled trace is brought into the trace cache.

A bit is included to decide if the trace is an important one or a non-important one. Usage of this bit is elaborated in Section 3.2.

3. Profile Scheme

The idea behind our profiling scheme is to identify the most important traces in each of the program partitions. When the profile is used in the subsequent runs, this profile should be able to help us differentiate between important and not so important traces. Retention of the frequently used traces helps reduce overheads in trace reconstruction and wastage of space in the cache. This conclusion is derived from the fact that there are chances that sparsely used traces might replace frequently used traces when the entire trace cache consists of only important traces. Misses for the important traces and their construction may entail this. The authors believe that the proposed approach might not only increase the trace hits but also save the line fill buffer logic from extra work.

This section is divided into two subsections; one explaining the process of profile construction and the other relates how profiles are used for the replacement logic.

3.1 Profile Construction

The profile should capture the details about the frequent traces in a compact manner.

3.1.1 Trace Maturing

When a trace is brought into the trace cache for the first time, its usage counter is initialized to one. Every hit increments this counter and when the value rises above a threshold t1, an entry for this trace is made in the profile table and all its details like the starting block addresses, branch pattern bits and the usage counter value are written into it. Hereafter, hits update the usage counters both in the trace cache and the profile table. Even if this trace is removed from the cache, the entry in the profile table is undisturbed. The next time (if there is one) the same trace is brought into the trace cache, its usage counter is initialized to the value from the profile table and, as before, is incremented with each hit. Once this usage counter crosses another threshold t2, we decide the trace has 'matured' and mask its entry in the profile table by setting the mask bit to 1. When all the entries are masked its time to write the profile to a file in memory. It should be noted that all entries continue to be updated till the profiling is done. The interrupt strategy that we discuss in Section 3.1.3 helps building the profile for each partition of the program.

3.1.2 Profile Entry Replacement

It is only natural that all the entries in the profile table might get filled and a suitable way has to be found to profile the other traces that might secure hits greater that t1. We maintain an index into the entries and follow a simple round-robin replacement of those entries, which are not masked yet.

3.1.3 Interrupt Strategy

All the entries in the profile table will eventually mature and no more entries will be available for replacement and hence all requests for new entries are denied. Once this happens, we start a counter that keeps track of the number of denials and when this reaches a value of 15, its time to write the profile into a file in memory. An interrupt is generated to be serviced by the processor. On every profile write, all the trace cache entries' usage counter is reset to zero.

3.2 Profile Usage

The profile can be used at runtime if the conditions of program execution remain the same as when the profile was initially constructed.

When the execution environment is confirmed to not have changed, profiles corresponding to each subset of the program are loaded into the profile table. For purposes of this discussion, let us define *important* traces as those that have been profiled and others as *not so important* traces. Before a line is constructed by the line fill buffer logic, the profile table is looked up to determine if an important trace is going to be constructed. If a match is found, then the trace is constructed and loaded into the cache, its usage counter is set to the importance factor from the profile table and the importance bit in the trace cache is set.

If a match is not found, then a random value between 0 and t1 is computed. This is the value to be stored in the trace usage counter accompanied by the resetting of the importance bit, if this trace successfully finds an entry in the cache. Each time a trace is hit in the cache, its importance factor is decremented by one.

The replacement logic is discussed next. When all lines in the trace cache have their importance bits set, the following scheme is used.

- If a line constructed by the line fill buffer logic is an important trace (determined by looking up the profile table), the trace with the least importance factor is evicted. The entry corresponding to this trace in the profile table is modified to reflect the latest importance factor value.

- If a non important trace is going to be constructed by the line fill buffer logic, then a random value between 0 and t1 is computed. If the new trace's random value is greater than the trace with least importance factor, the latter is replaced by a newly constructed trace else the new trace is not constructed at all. In case the new trace finds an entry, then its importance factor is initiated to the random value computed.

If there are one or more non important traces in the cache, then an index maintained into the trace cache evicts the non important traces in a round robin fashion whenever a new trace is constructed (both important and non-important).

In the above cases, comparing the prediction bits between traces having the same importance factor breaks the tie. Traces that consist of taken branches earlier in the trace are considered superior. The rationale behind this decision is explained in Section 4. If the prediction bits of the traces in question are identical, one of them is randomly chosen for eviction.

4 Benefits

Several benefits would arise from the use of an informed replacement logic. The benefits can be better appreciated in the light of some trace cache performance features.

It is seen that the performance of trace caches increases with an increase in size. The results used in this section are taken from an exploration of the trace cache design space [3]. The IPC shown here is the average for a set of 5 test programs (s95-compress-small, s95-gcc, s95-li-train, s95-m88ksim-test and s95-perl-train) of the SimpleScalar toolkit [6].

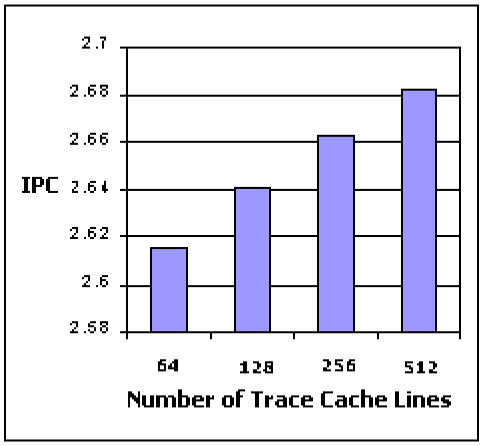

Fig 2: Effect of Trace Cache Size on IPC

Increased size in the size of the cache would allow an increased number of traces to be in the cache at any given point in time. This would lead to more hits and performance gain. However, it is evident that increase in the size is not economical, as it would increase the power consumption of the cache. We could do better if we could get more number of trace hits with the same size of the trace cache. It is towards this that our scheme tries to progress. By allowing only the most important traces i.e. the most frequently used traces to be resident in the trace cache, the space is used economically and to its best.

This design tries to do away with situations wherein a very sparsely used trace might evict a frequently used trace when the entire cache is full of frequent traces. The next fetch for this frequent trace might result in a miss and will lead to reconstruction of the trace. The sparsely used trace might also be removed soon from the cache. It is perceived that such extravagant usage of the trace cache wouldn't be permitted in our design.

When a couple of traces are vying for trace cache residency and happen to have identical importance factors, our design gives superiority to those traces that have earlier taken branches (i.e. non-contiguous blocks appear earlier in the trace). The rationale behind such an idea is based on two properties:

i. Multiple branch predictors have a lower accuracy due to the fact that they have to predict a number of branches in advance [7].
ii. Contiguous blocks have a high probability of being found on a single line in the instruction cache.

From the first property, we see that the accuracy of multiple branch predictors is limited. Further, the usefulness of the blocks that comprise the tail of the trace is appreciably lesser that those that initiate one, as is illustrated by Figure 3. The values correspond to an average for 5 programs (same as in Fig.2).

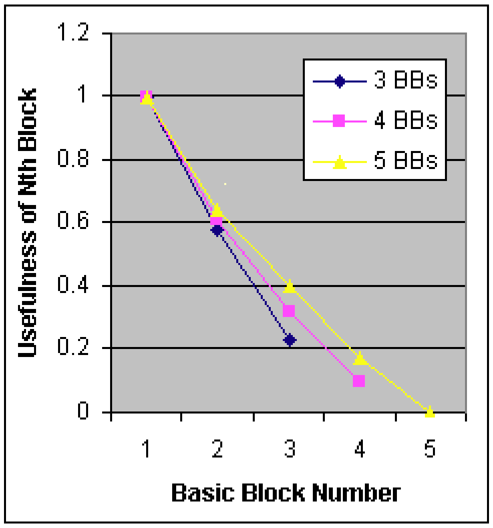

Figure 3: Usefulness of Basic Blocks

The utility of the posterior blocks in a trace (Fig 3) is seen to be less. This is a manifestation of the poor accuracy of prediction for the last few blocks by the multiple branch predictor.

So assuming only the initial blocks in every trace are useful, we give more importance to traces which consist of taken-branch blocks in the earlier part of the trace. These traces might take more cycles to fetch than those that have contiguous blocks (if these blocks are present in a single line in the instruction cache). We base choosing our trace candidate for eviction, when more than one trace has the same importance factor, on this explanation.

5 Conclusion

We present a scheme in this paper that tries to increase the efficiency of trace caches without increasing their size. The scheme is aimed at increasing the trace hits by retaining only the most important traces in the cache.

The authors believe that a cycle level simulation of the proposed architecture could quantify the gains lucidly, at the same time bringing out any expensive trade-off that might have to be cogitated upon.

6 References

[1] E. Rotenberg, S. Bennett, and J. Smith, “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching". In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996

[2] Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt “Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache., In *Proc. 7th ACM International Conference on Supercomputing*, 1993

[3] Paul W. Lee, Mahesh J. Madhav, Anamaya Sullerey, “Exploring the Trace Cache Design Space”, Project Report, http://cva.stanford.edu/ee482a/projects.html

[4] Craig B. Zilles, Gurindar S. Sohi, “A Programmable Co-processor for Profiling”. In *Proceedings of 7th International Conference on High Performance Computer Architecture (HPCA-7)*, 2001

[5] J.Dean, J.Hicks, C.Waldspurger, W.Weihl, and G.Chrysos, “Profile Me: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors”. *In Proceedings of 30th International Symposium on Microarchitecture*, 1997

[6] D.C. Burger and T.M.Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, 1997

[7] Q. Jacobson, E. Rotenberg, and J. Smith, “Path-Based Next Trace Prediction.” In *Proceedings of the 30th International Symposium on Microarchitecture*, November 1997