A Comparative Study of Redundancy in Trace Caches

Hans Vandierendonck[1], Alex Ramírez[2],
Koen De Bosschere[1], and Mateo Valero[2]

[1] Dept. of Electronics and Information Systems, Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium,
`{hvdieren,kdb}@elis.rug.ac.be`

[2] Computer Architecture Department, Universitat Politècnica de Catalunya,
c/ Jordi Girona 1-3, Module D6, 08034 Barcelona, Spain,
`{aramirez,mateo}@ac.upc.es`

Abstract. Trace cache performance is limited by two types of redundancy: duplication and liveness. In this paper, we show that duplication is not strongly correlated to trace cache performance. Generally, the best-performing trace caches also introduce the most duplication. The amount of dead traces is extremely high, ranging from 76% in the smallest trace cache to 35% in the largest trace cache studied. Furthermore, most of these dead traces are never used between storing them and replacing them from the trace cache.

1 Introduction

The performance of wide-issue superscalar processors is limited by the the rate at which the fetch unit can supply useful instructions. The trace cache [1] can fetch across multiple control transfers per clock cycle without increasing the latency of the fetch unit [2]. Fundamental to the operation of the trace cache is that consecutively executed instructions are *copied* into consecutive locations, such that fetching is facilitated.

In this paper, we analyse the relation between redundancy and trace cache performance. We use the term redundancy to mean *those traces and instructions stored in the trace cache that do not contribute to performance.* We consider two types of redundancy. The first type of redundancy detects multiple copies of the same instruction and is called *duplication.* The second type of redundancy is *liveness.* Dead traces are redundant (need not be stored in the trace cache), because they do not contribute to performance. We show in this paper that liveness is a greater concern in small trace caches than duplication. We study various ways to build traces and inspect their effect on redundancy.

2 Metrics

After every trace cache access, we make a break-down of the contents of the trace cache and average this break-down over all accesses. We consider two break-downs: one to quantify duplication and one to quantify liveness.

We measure 4 categories to characterise duplication. **Unused** traces correspond to trace cache frames that are never filled during the execution of the programs. Fragmentation (**frag**) is caused by traces shorter than the maximum length of 16 instructions. Duplicated intructions (**dup**) are those instructions for which there is another copy of the same static instruction in the trace cache. The first copy of each static instruction is present in the category **uniq**. This part of the trace cache would also be present in a regular instruction cache.

When studying liveness, we consider following types of traces. **Unused** traces are the same as in the previous break-down. **Quick-dead** traces are never used between being built and being replaced. **Dead** traces are stored in the trace cache but will not be used again before being replaced. Other traces are **live**.

3 Methodology

We measure the effect of the trace termination policy on duplication and liveness. The baseline policy base allows traces to grow up to 16 instructions and 3 branches. The whole blocks policy wb requires that a trace ends in a branch instruction, unless a basic block is longer than 16 instructions. The backward branch policy bb terminates traces in a backward branch only. Other variations are the *software trace cache* (STC) [3], *selective trace storage* (STS) [4] and the combination of STS and STC, abbreviated STCS. STS discriminates between blue traces that have all their internal branches not-taken and red traces which contain taken branches. Blue traces can be fetched from the instruction cache in a single cycle and need not be stored in the trace cache. With STC, the program is rewritten to bias branches towards not-taken, resulting in more blue traces.

The analysis is performed on 6 SPECint95 benchmarks running training inputs. At most 1 billion instructions are simulated per benchmark. We use a 32 kB 2-way set-associative instruction cache with 32 byte blocks. It can fetch at most one taken branch and 16 instructions per cycle. We use an MGag multiple branch predictor with 16K entries and a 2K-entry 4-way set-associative BTB and a 256-entry RAS. The trace cache size is varied from 32-4096 traces and is always 4-way set-associative. A trace can hold at most 16 instructions and 3 conditional branches.

4 Analysis

The performance is measured as FIPA: useful number of fetched instructions per fetch unit access (Figure 1). The performance results follow the trends known from the literature [3,4,5].

4.1 Duplication in the Trace Cache

The break-down of duplication (Figure 2) shows that **unused** trace cache frames only occur often for the biggest trace cache investigated, because for most programs the number of different traces is around 4K or less.

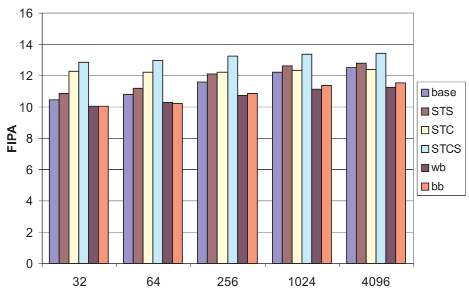

Fig. 1. FIPA for the trace cache. The horizontal axis shows the size of the trace cache expressed in traces.

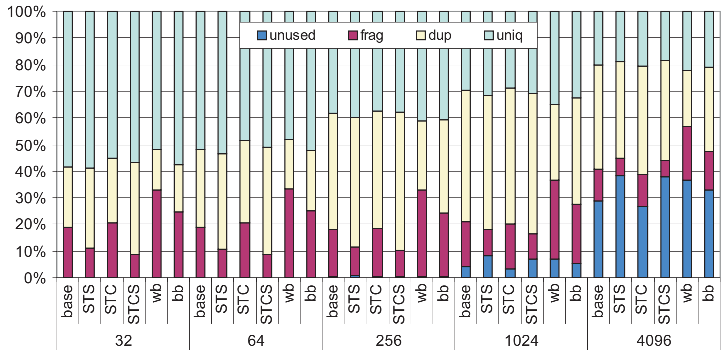

Fig. 2. Breakdown of wasted instruction frames in the trace cache.

At least 10% of the trace cache is wasted on **fragmentation**. The impact of fragmentation is largely independent of the trace cache size. STS and STCS reduce fragmentation because red traces are longer than blue traces [4]. The wb and bb policies increase fragmentation. Because the traces are terminated on all or some branch instructions, they are a lot shorter than with the base policy. For this reason, the performance of the wb and bb policies is lower than with the other policies.

Duplication varies strongly over the different trace termination policies and trace cache sizes. In the base case, 20% to 50% of the trace cache stores duplicates of instructions already present in the trace cache. This number is the largest for the 1K-entry trace cache. For the larger trace cache it is less. Instead, trace cache frames are left empty. This trend occurs for all trace termination policies.

STS increases duplication over the base policy, e.g.: from 43% to 48% in the 256-entry trace. Duplication is increased because the STS policy does not store blue traces but stores red traces instead. The red traces are longer and hence additional copies of instructions are inserted in the trace cache. This way, fragmented space is converted into duplicated instructions. When the trace cache is very large (e.g.: 4K traces) then the fragmented space is turned into unused trace frames.

The wb policy reduces duplication but increased fragmentation removes all benefits. STS is more effective to reduce the number of constructed traces, because it does not impact the fragmentation.

A large part of the trace cache is wasted, e.g. 60% of the 256-entry trace cache. This indicates that the trace cache is very inefficient in storing the program: A trace cache needs to be about 2.5 times as big as an instruction cache to store the same piece of code.

The trace termination policy influences duplication but not in a straightforward way. The best performing policy (STCS in the 1K-entry trace cache) also has the highest duplication. Therefore, when looking for new techniques to improve trace cache performance, one should not be concerned with reducing duplication per se, because the amount of duplication my increase with increasing performance, as is the case when using selective trace storage.

4.2 Liveness of Traces

The liveness analysis (Figure 3) shows that the trace cache contains many dead traces, around 76% for the smallest to 35% for the biggest trace caches. For the small trace caches, the majority of the dead traces is built but never used. This part slowly decreases with increasing trace cache size and for the 1K-entry trace cache, around 50% of the dead traces have never been used since they were built. This trend is largely independent of the trace termination policy.

In short, the contention in the trace cache is very high and most of the traces are dead. It is therefore worthwhile to consider not storing some traces, as these traces will not be used. In fact, STS gains performance this way.

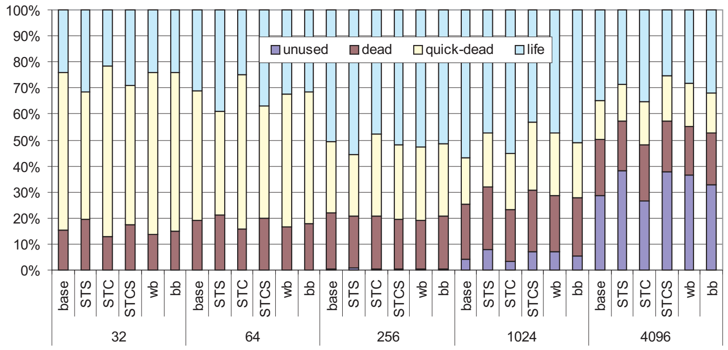

Fig. 3. Breakdown of life and dead traces.

5 Conclusion

This paper studies duplication and liveness under several optimisations. Duplication occurs frequently, e.g.: 43% in the 256-entry trace. However, duplication does not seem to be strongly correlated to trace cache performance, although the best-performing trace caches also introduce the most duplication.

The amount of dead traces is extremely high, ranging from 76% in the smallest trace cache to 35% in the largest trace cache studied. Furthermore, most of these dead traces are never used between storing them and replacing them from the trace cache.

Acknowledgements

We want to thank Joseph L. Larriba Pey for his useful comments on this work. Hans Vandierendonck is supported by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT). This work has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and by a research grant from Intel Corporation. This research is partially supported by the Improving the Human Potential Programme, Access to Research Infrastructures, under contract HPR1-1999-CT-00071 "Access to CESCA and CEPBA Large Scale Facilities" established between The European Community and CESCA-CEPBA.

References

1. A. Peleg and U. Weiser, "Dynamic flow instruction cache memory organized around trace segments independent of virtual address line," *U.S. Patent Number 5.381.533*, Jan. 1995.
2. E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Conference on Microprogramming and Microarchitecture*, pp. 24–35, Dec. 1996.
3. A. Ramírez, J. Larriba-Pey, C. Navarro, J. Torrella, and M. Valero, "Software trace cache," in *ICS'99. Proceedings of the 1999 International Conference on Supercomputing*, June 1999.
4. A. Ramírez, J. Larriba-Pey, and M. Valero, "Trace cache redundancy: Red & blue traces," in *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, Jan. 2000.
5. S. Patel, M. Evers, and Y. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 262–271, June 1998.