Reevaluating the Renamed Trace Cache Architecture

Yen-Chun Wang Kazuo Horio Ryota Shioya Masahiro Goshima Shuichi Sakai

Dept. of Information and Communication Eng,

University of Tokyo

Abstract

In order to exploit parallelism, modern superscalar processors utilize register renaming to solve data dependency problems. The RMT(register mapping table) used in register renaming is said to be one of the most energy consuming components in the processor due to its high access frequency and large area.

Multi-port structure of the RMT can gain area exponentially to its port number which can make it unrealistic when implementing high width processors[1]. *The RTCA, abbreviation for Renamed Trace Cache Architecture(also known as the anti-dualflow architecture*[2]*) is an architecture proposed to solve these problem that the RMT brings about. Within this architecture, the path between two dependent operand is explicitly shown in order to solve the dependency that in return can take off the renaming stage of the pipeline. However, extra tags are needed in addition to the trace cache that RCTA uses, which can lead to degradation in performance.*

In this paper, we evaluate and compare the RCTA to a typical trace cache architecture. It is shown that the increment in trace cache tags can cause an amount of degradation in average fetch IPC. Nevertheless, the shorten pipeline can potentially maintain overall performance.

1. Introduction

Superscalar processors are usually separated into the frontend side and backend side by instruction issue buffers, such as issue queues or reservation stations. With the urge of gaining higher instruction-level parallelism (ILP), these issue buffers tend to become larger. As the instruction window grows larger, the port number in the RMT also increases which leads to large area. The RTCA is proposed to solve such problems that RMT may cause.

In the RCTA, instructions are translated to a [-n] form where the n stands for the distance between the dependent operands. Since the dependency is expressed explicitly in the instructions, register renaming can be omitted. The main problem of this form is that the distance between operands is path relevant, in other words, path information is also necessary to determine the dependency.

The RCTA uses a trace cache architecture as a base. Upon a trace cache miss, instructions are translated to the [-n] form and stored in the trace cache. This translation bandwidth can be chose rather small, hence constraining the port numbers of the mapping table used. On the other hand, path information is represented by additional tags in the trace cache, which can also lead to higher miss rates.

The trace cache was proposed to increase the instruction fetch bandwidth in the frontend side. In this paper we evaluate the RCTA and quantitatively show the impact it can have on the fetch bandwidth compared to the trace cache architecture. Although results show degradation in average fetch group size, the shorter pipeline is considered compensative in overall performance.

2. The Trace Cache Architecture

The trace cache was first proposed by Rotenberg et al.[3] in order to achieve high instruction fetch bandwidth. Instructions are stored by their static compiled order in the instruction cache, making instruction blocks of a stream possibly scattered inside the cache. The trace cache aligns the scattered blocks into a trace and makes it possible to fetch according to the dynamic instruction stream.

A trace is a specified by a starting address and a sequence of branch outcomes, which describe the path followed. Trace cache hit is determined by comparing the address and multiple branch outcomes. The trace cache utilizes temporal locality and branch behavior.

3. Renamed Trace Cache Architecture

The first step in the RCTA is translating instructions into the [-n] form. Basically, this is done in the same way as register renaming. The relation of an instruction index and its corresponding operand is stored in a mapping table. When there is a pair of dependent operands encountered, the mapping table is accessed in order to compute the difference between them,

which stands for the displacement of the dependent operands. This difference is further compared to the window size. The instruction is translated into the [-n] form only when the difference is small than the window size. In addition, operands obtain values from physical registers if instructions are in the [-n] form and from logical registers if they are not.

Translation process only occurs when there is a trace cache miss. Instructions finished translating are then further stored in the trace cache. As mentioned above, the displacement between dependant operands in relevant to the path the trace has taken. As result, path information is necessary while fetching inside the trace cache. The following are some chose parameters.

- *H* Starting address of path.
- *L* Path length.
- *pbflag* Branch history contained.
- *Jtarget* Target of indirect branches

These parameters are transformed into additional tags before storing into the trace cache.

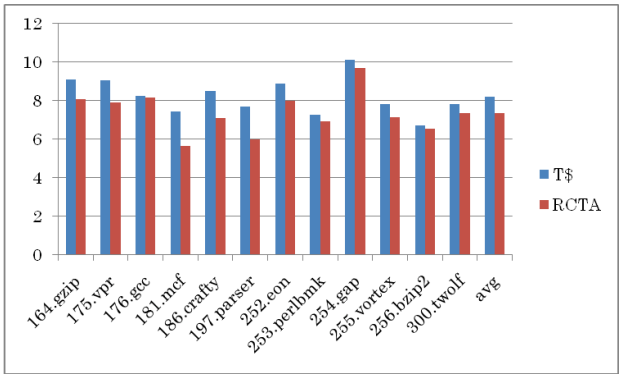
Indexing and tag comparison is somehow different from the instruction cache. Besides the fetch address, a L_{min} is chose to reduce conflict miss. The path branch history and PC history that L_{min} corresponds to, along with the fetch address, is used to compute the index.

After indexing into a set inside the trace cache, the parameter *L* inside the traces are used for tag comparison. First, *L* is used to find the starting address *H* and past branch flags *pbflag*. If the branch and PC history of length *L* is concurrent, the followed path is matched. A multiple branch predictor and particular hit logic is used to compare the branches inside the trace. Trace hit only occurs when the path tags and the tags of the conventional trace cache are both assured equal.

4. Simulation and Results

Onikiri processor simulator[4] and SPECCPU CINT2000 benchmark are used in our simulation. Integer benchmarks rather than floating point benchmarks are chosen because they have more complex control flows. The processor is 16-way superscalar. 1K entry, 4-way trace cache is assumed. The instruction cache is accessed after the trace cache miss is confirmed.

The tag difference in the trace cache can affect the frontend fetch bandwidth. The figure above shows the difference in fetch IPC of the RCTA and conventional trace cache architecture. Since



RCTA needs more tags to express the path information, increase in total trace number is obvious. The additional tag comparison can cause higher conflict miss rate, which leads to a lower fetch bandwidth. The L_{min} chose to reduce conflict miss in the figure above is set to the value 7. Overall, the RCTA shows an average of 10.3% degradation of fetch IPC. We quantitatively express the impact of RCTA on fetch bandwidth.

5. Summary

The RCTA is proposed to solve RMT problems of register renaming. Its unique translation can explicitly show the displacement of dependent instructions. In order to do this, additional tags are needed in the trace cache to express path information. On the other hand, because the dependency of instruction are solved after this translation, register renaming stage can be skipped and achieve a shorter pipeline. Our evaluation shows degradation in fetch IPC compared to the conventional trace cache architecture. This difference is considered compensative in the shorter pipeline.

References

[1] Tatsumi, Y. and Mattausch, H. *Fast quadratic increase of multiport-storage-cell area with port number*. Electronics Letters, Vol. 35, No.25, pp. 2185-2187 (1999).

[2] 一林宏憲, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一. 逆アーキテクチャ. 先進的計算基盤システムシンポジウムSACSIS 2008,pp.245-254(2008).

[3] E. Rotenberg, S. Bennett, J. E. Smith. *Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching*. Proc. of the 29th MICRO,pp.24-35(1996).

[4] 渡辺憲一, 一林宏憲, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬」の設計, 先進的計算基盤システムシンポジウム SACSIS2007（ポスター), pp. 194–195 (2007).