University of Jordan

Computer Engineer and Networks Master

Advanced Computer Architecture, Research Report

Trace Cash.

Name: Dua’a Ibrahim Al-Najdawi

Date: 2/12/2009

Abstract:

The trace cache is a proposed solution to achieving high instruction fetches bandwidth by buffering and reusing dynamic instruction traces. This work presents a new block-based trace cache implementation that can achieve higher IPC performance with more efficient storage of traces. The trend in superscalar design has been wider dispatch/issue window, more resources (i.e. functional units, physical registers, etc) and deeper speculation. However, despite these hardware enhancements, there exist bottlenecks that diminish the throughput. One such hindrance is the execution of long noncontiguous instruction sequences that cannot be fetched in a continuous stream from traditional instruction caches because instructions are stored in a static order in which they were compiled. A new cache structure, called Trace Cache, was proposed by Rotenberg et.al in [1]. This cache stores instructions in their dynamic order of execution, and hence provides a high bandwidth for instruction fetching.

Introduction:

Now all the processors are superscalar processors and have the instruction level parallelisms technique. The superscalar design has to increase the scale of these techniques: wider dispatch/issue, larger windows, more physical registers, more functional units, and deeper speculation. As the issue width of superscalar processors is increased, instruction fetch bandwidth requirements will also increase.

As mentioned in reference 1 and 2, as aggressive instruction-level parallelism techniques are widely used in superscalar processor design, the dispatch/issue window becomes larger and larger and branch speculation is getting deeper, therefore, instruction fetch bandwidth is becoming a performance bottleneck. There are also some other factors appearing when issue rate exceeds four instructions per cycle:

(1) Branch throughput: if only one condition branch is predicted per cycle, then the window can grow at the rate of only one basic block per cycle.

(2) Noncontiguous instruction alignment: instructions to be fetched may not be in contiguous cache locations because of conditional branch or jump instructions, so it will cause a high fetching latency if there is no other logic to fetch these instructions in parallel and align them and pass them up the pipeline.

(3) Fetch unit latency: if a branch was mispredicted, recovery much be done to resolve this mispredition. Some instructions have to be squashed from pipeline stages and fetching needs to be redirected. The startup cost of redirecting fetching will cause fetch unit latency.

Rotenberg et al [1] proposed Trace Cache technique to address the above problems. Trace Cache is a hardware structure, as shown in figure1, each line of which stores a snapshot, or trace, of dynamic instruction stream. A trace is a sequence of at most n instructions and at most m basic blocks starting at any point in the dynamic instruction stream. A trace is specified by a starting address and a sequence of up to m-1 branch outcomes which describe the path followed. A line of trace cache is filled as instructions are fetched from the instruction cache. If the same trace is encountered again in the course of executing the program, it is fed directly to the decoder. Otherwise, fetching normally proceeds from the instruction cache. The reason Trace Cache works is program properties of temporal locality and easily predicted branch behavior.

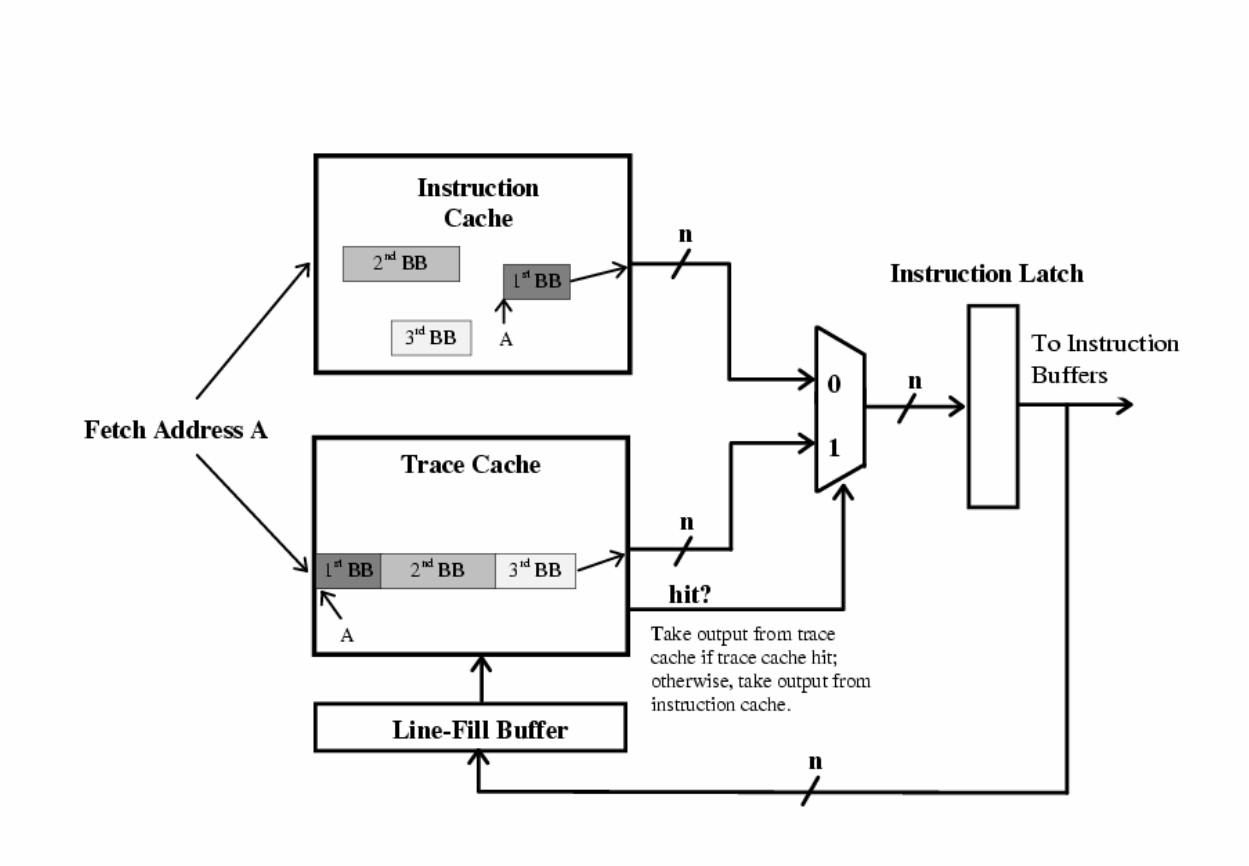

Figure1:trace cash[3]

Starting of the trace cache:

The earliest widely acknowledged academic publication of trace cache was by Eric Rotenberg, Steve Bennett, and Jim Smith in their 1996 paper "**Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching**." propose supplementing the conventional instruction cache with a trace cache as shown in figure 1. For the Instruction Benchmark Suite (IBS) and SPEC92 integer benchmarks, a 4 kilobyte trace cache improves performance on average by 28% over conventional sequential fetching.

Then the same researchers write "**ATrace Cache Microarchitecture and evaluation "**. The microarchitecture provides high instruction fetch bandwidth with low latency by explicitly sequencing through the program at the higher level of traces, both in terms of 1) control flow prediction and 2) instruction supply. For the SPEC95 integer benchmarks, trace-level sequencing improves performance from 15 percent to 35 percent over an otherwise equally sophisticated, but contiguous, multiple-block fetch mechanism[2].

Data Trace Cache:An Application Specific Cach eArchitecture [3]

The researchers here focus on tree data structures which are responsible for a significant component of the memory traffic in several applications. We have observed that tree accesses create a simple to characterize trace of memory references and propose a data trace cache design to exploit the locality of reference in these data traces.

And their study reveals that data trace caches can reduce the total number of misses from 7% to 53% for accesses to rooted tree data structures as compared to a conventional cache for a variety of applications for small cache sizes (256 - 1024 bytes). Such caches are in keeping with the philosophy of victim caches, stream buffers, and pre-fetch buffers in that relatively small investments in silicon can realize substantive reduction in off-chip memory bandwidth demand.

Trace Cache Sampling Filter [2]

I is simple mechanism to increase the utilization of a small trace cache, and simultaneously reduce its power consumption. The sampling filter exploits the "hot/cold trace" principle, which divides the population of traces into two groups.

The first group contains "hot traces" that are executed many times from the trace cache and contribute the majority of committed instructions. The second group contains "cold traces" that are rarely executed, but are responsible for the majority of writes to an unfiltered cache.

The sampling filter selects traces without any prior knowledge of their quality. However, as most writes to the cache are of "cold traces" it statistically filters out those traces, reducing cache turnover and eventually leading to higher quality traces residing in the cache.

[1]Results show that the sampling filter can increase the number of hits per build (utilization) by a factor of 38, reduce the miss rate by 20% and improve the performance-power efficiency by 15%.Further improvements can be obtained by extensions to the basic sampling filter: allowing "hot traces" to bypass the sampling filter, combining of sampling together with previously proposed filters, and changing the replacement policy in the trace cache. Those techniques combined with the sampling filter can reduce the miss rate of the trace cache by up to 40%.

From recent solutions I pick new hardware technique called diverge on miss :[4]

The new processors are multicore with wide SIMD (Single Instruction, Multiple Data). The researcher in reference [4] introduce a hardware technique called “diverge on miss” that allows SIMD cores to better tolerate memory latency for workloads with non-contiguous memory access patterns. Individual threads within a SIMD “warp” are allowed to slip behind other threads in the same warp, letting the warp continue execution even if a subset of threads are waiting on memory. Diverge on miss can either increase the performance of a given design by up to a factor of 3.14 for a single warp per core, or reduce the number of warps per core needed to sustain a given level of performance from 16 to 2 warps, reducing the area per core by 35%.

Conclutions :

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. Towards this end, I found research that promises a higher performance with lesser overheads and the conventional trace cache size. Which is the Trace Cache Sampling Filter, and diverge on miss.

References:

1- Eric Rotenberg, James E. Smith, Steve Bennett,"Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching" micro, pp.24, 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'96), 1996.

2- MICHAEL BEHAR , AVI MENDELSON , AVINOAM KOLODNY, "Trace Cache Sampling Filter".

3- Eric Rotenberg, Steve Bennett, and James E. Smith. A Trace Cache Microarchitecture and Evaluation, in IEEE Transactions on Computers, 48(2):111-120, February 1999.

4- David Tarjan, Jiayuan Meng and Kevin Skadron ,(2009),Increasing Memory Miss Tolerance for SIMD Cores.

5- Trace Cache, Bing Chen , Musawir Ali. , www.cs.ucf.edu

6- trace cache, Leon Gu,Dipti Motiani