EE482: Advanced Computer Organization Lecture #5
Processor Architecture
Stanford University Wednesday, 10 May 2000

Trace Cache

Lecture #5: Wednesday, 15 April 2000
Lecturer: Paul Wang Lee
Scribe: Mattan Erez

Due to unfortunate circumstances this lecture was not scribed, following are several points that I remember were brought up. If anyone has something to add please tell me.

In this session we discussed three papers:

1. A. Peleg and U. Weiser, *Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line* - an Intel patent that predates the academic work on trace caches.

2. D. Friendly, S. Patel, and Y. Patt, *Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism* - describes several enhancements to the original University of Michigan view of the trace cache.

3. Q. Jacobson, E. Rotenberg, and J. Smith, *Path-Based Next Trace Prediction* - introduces a different way of predicting and indexing traces for the University of Wisconsin trace cache.

1 Intel Trace-Cache

This patent describes an instruction cache variant designed specifically for superscalar processors. It improves instruction fetch throughput while allowing a single fetched line to cross basic-block boundaries, and removing the alignment constraints of conventional instruction caches. This organization is also amenable to solving some of the more specific fetch issues of CISC processors, i.e. variable length instructions.

The trace cache is organized around *traces* which are single-entry multiple-exit sequences of dynamically executed instructions. This is in contrast to conventional instruction caches which use sequential memory addresses as their basic line of instructions. Branch prediction information is tightly integrated with the instructions to allow fetches that also cross basic-block boundaries. A trace can contain several basic blocks that are implicitly predicted to follow one another.

By fetching an entire trace many of the shortcomings of conventional instruction caches are circumvented. In instruction caches lines are aligned on address boundaries which have little to do with the actual fetch patterns, also frequent branches cause only a small part of each line fetched to be usable, as seen in the collapsing-buffer paper of

Lecture 4. In the trace cache each line contains a complete line of usable instructions (in case of a correct prediction) achieving a much higher bandwidth.

A new problem that is now encountered is that instructions are no longer accessible individually but only as groups, indexed by the *trace head* - the instruction pointer of the first instruction in the trace. By definition each basic block contains a single entry point and a single exit point, so while instructions need not be accessed individually, basic blocks should. In this patent every basic block in the cache can be accessed. This is done by replicating every basic block, such that it serves as a trace head (and may also be a second basic block of a different trace). This leads to a large amount of redundancy in the trace cache. Other trace-caches, such as Wisconsin and Michigan, do not replicate all basic-block starts leading to possibly inaccessible instructions.

Traces are built using the *line buffer* (or fill-unit). It records instructions as they are retired from the execution core, and inserts them into the trace cache when a trace end-condition (such as an indirect jump) is encountered. In the detailed configuration each line can hold up to two branches, so 3 line-buffers are provided. The first holds the trace as it is being written into the cache, the second line holds the current trace being built, and the third is for the following trace that will start with the second basic-block of the current trace.

Another interesting point of the paper is the replacement policy for traces in the cache. Instead of using a regular LRU or immediate replacement policy, traces are not cached if they are from a path that contradicts the branch predictor. This is done so that useful traces are not kicked out, a trace which goes against the branch predictor will never be fetched, even if it were stored in the cache.

The trace cache trades off hit-rate (due to replication) for bandwidth (usable instructions are grouped together and aligned).

Although this trace-cache variant has many similarities to the academia trace-caches and to more mature industry variants, it had a few limitations and was perhaps not fairly compared to conventional instruction caches, as it needs dual-ported memory which could be utilized for increasing bandwidth in a conventional organization.

2 Michigan Trace-Cache Improvements

This paper develops two techniques to improve the original Michigan trace-cache. The first, *partial matching*, removes a constraint of the original model, that discarded an entire fetched trace on a branch mismatch. Instead, all instructions fetched from the trace up to the mismatch are used by the processor. This provides a performance benefit by utilizing trace cache memory to supply as much code as possible from this level of cache. For example, if code sequence ABC is stored in the cache, but ABD is predicted, then the first segment AB is fetched from the cache. Without partial matching, the entire sequence ABD would be regarded as a miss. This ”improvement” was actually already inherent to the Intel trace-cache, and was solving a side-effect of the original Michigan mechanism.

The second improvement is *inactive issue*, which extends partial matching. When a predicted sequence is partially matched, the processor issues the entirety of the trace cache segment, marking the mismatching tail of the trace as inactive. If the inactively issued instructions are not needed, they are squashed, but they are made active if the branch was mispredicted. This reduces the effect of branch mispredictions. Inactive issue is similar to a hardware form of indiscriminate predication.

This paper simulated an unreasonably large instruction window: “16x32 instructions”, as stated in section 5, in addition to the large number of 16 functional units. Also, besides showing the obvious improvement due to partial matching, a counter-intuitive result is that with inactive issue performance improves more when the branch predictor is less accurate. This is true since the mechanism is more effective then, but overall performance is greatly reduced.

3 Wisconsin Trace-Cache - Next Trace Predictor

This paper introduces a variant on branch prediction specifically designed for trace-caches - the*next trace predictor*. The predictor is tailored to the Wisconsin style trace cache which tags each trace both with the IP of the head instruction and with a coding of all instructions in the trace (by adding bits for directions of conditional branches in the trace). Instead of predicting individual branches, using extended branch predictors, and then looking up the resulting trace in the cache, an entire trace-identifier is predicted based on the path history of previous traces. A trace-history-register, similar to a branch history register, records a hashed version of the previous trace-identifiers seen, and is used to access a second-level table for predicting the next trace. In this way several branches are implicitly predicted each cycle, without extreme modifications to current branch predictors (much more space and longer latencies are incurred for accurate multiple predictions per cycle).

Additions to this include a hybrid predictor that avoids coldstart problems and a return history stack to glean information from subroutine calls. This is found to reduce misprediction rate substantially. Since the traces, which now are treated as basic units of execution, can have multiple successors, while a typical branch can have only two, the predictor tracks sequences of trace identifiers. The hybrid predictor, which is a smaller predictor than the main trace predictor and thus requires less information to predict, is more immune to startup costs. It also helps avoid problems with aliasing into the main predictor. The final improvement in the paper, predicting alternate traces, predicts an alternative to the main path of execution. This is useful if aliasing occurs in the trace cache or if the predicted trace is not predicted with a strong likelihood. This technique can be used for fetching the alternate trace and possibly executing it before the branch is resolved, lessening the misprediction penalty.

It was noted that although the authors showed performance improvements for their technique, the metric they chose - trace prediction accuracy, prevents comparison to branch predictions, since branches are implicitly predicted.

4 Final Notes

Although not specifically detailed in the Wisconsin and Michigan papers, the two trace caches are different from each other, and also from the one presented in the patent as well as trace caches that are appearing in commercial processors (the HAL Sparc64-V, and Intel's "Willamette"). These differences are very noticeable in the various suggested improvements which can not be applied to all trace cache designs. The Michigan suggestion of partial-matching is already part of the Intel Patent, and both it and inactive-issue are inapplicable to the Wisconsin model, since after a trace misprediction there it is very unlikely that any instructions are usable (traces have unique identifiers). Similarly, the Wisconsin next-trace predictor can not be used with the Intel and Michigan trace caches, since they do not have a way of coding path information in the trace tag.