Project Proposal

Cody Martin and Adu Bhandaru

1 Problem

Applications have diverse memory access patterns. As of now, cache configurations are rigid and cannot exploit certain predictable repetition in memory access patterns. Having a dynamic cache structure that tailors itself towards the currently executing application could yield higher performance by removing a great deal of cache conflict misses that would normally occur with a statically configured cache. Many microarchitectures these days use runahead execution to predict cache misses and schedule memory requests in advance, but to put a processor in runahead-mode is costly and is only beneficial if the cache miss latency it hides is long. Instead, targeting the cache dynamically could result in similar performance increases to that of runahead execution without incurring the same high costs.

2 Novelty

People have tried to tackle this problem with prefetchers, runahead execution, and general purpose access pattern predictors to hide latency. With configurable caching, we can certainly augment these techniques, if not replace them.

Currently, many microarchitecture designers use a branch predictor that implements hysteresis to determine what prediction to use during execution of each branch in the code. We can use this same concept of hysteresis to waiver between two or more cache implementations. This is an idea that we believe to be far from anything anyone has implemented but uses techniques that have been used in the past to assure effectiveness.

3 Idea

Consider having the private L1 cache be configurable. That is, design it such that we can dynamically determine the optimal number of sets (and therefore associativity) and perhaps even the block size. Imagine implementing the selection logic for a particular block for a number of different configurations and connecting them to the same cache. At any given time only one scheme would be enabled.

The which configuration to enable could be specified either by the programmer (needs ISA exposure) or determined dynamically. If programmatically, the programmer could just choose the configuration that best suits the application. Alternatively, we could record the access patterns generated for a particular core and determine the optimal cache configuration ourselves in hardware.

Even if only two configurations were implemented, we could still achieve good performance gains. Consider the graph of associativity to number of conflict misses. If we use a saturating counter for hysteresis, you could imagine a simple mechanism for choosing which configuration to use. Even just changing to a two-way cache makes a big difference in handling requests. Further, configurable block size from just 4-byte to 8-byte could make the difference between containing a full object or only half of it.

4 Hypothesis

By having the cache configuration adapt to the needs of access pattern, we can reduce the occurrence of conflict misses. Of course, we will still experience both compulsory and capacity misses. Further, this technique can be applied to systems that already employ runahead execution etc., and we should still realize performance gains.

5 Methodology

To test this idea, we will run a standard (or well established) simulator on series of well known benchmark programs. Then we will run a version of the simulator with a configurable cache (as described) and compare the performance.

6 Plan

- Milestone 1: Do some background research on other flavors or variants of configurable caches. Perhaps this idea has been explored and we have not found the paper. Determine feasibility of designing more complicated selection logic, estimate hardware costs incurred. If there are no red-flags we will move to milestone 2. A red-flag would be something like being unable to wire up two-types of selection logic to the SRAM banks, or incur very high latencies in doing so.

- Milestone 2: Specify and possibly implement the microarchitecture in Verilog and estimate the latencies for all operations. Specifically, investigate if we can return L1 hits in the same cycle, or if we need 2 cycles (or something) for a particular configuration.
- Milestone 3: Augment the simulator with configurable caches for each core, ensuring the microarchitectural specifications are met with regards to timing. Run the simulator on a well known benchmark suite.
- Final Report: Analyze the results from the benchmark tests. Observe performance gains (or loss) and speculate as to why certain tests performed better or worse. Determine where we can make improvements and log them.