Cache Simulator Report

CS 378: Energy Efficient Computing
Karthik Nemmani and Shreyansh Dixit

Introduction

My partner and I have implemented an L1-L2-DRAM cache simulator using Dinero traces as simulation data. Based on the specifications, we have developed the memory classes L1Cache, L2Cache, and DRAM, and a runner class CacheSim that runs the actual simulation. We have used Python for our implementation.

Simulation

As aforementioned, the simulation is executed by the CacheSim class. The initialization consists of opening and reading the Dinero trace file, as well as initializing the memory structures. Running the simulation entails first parsing a trace line into the access type, address, and data (ignored). From there, we check if it's a read or write, and handle accordingly (instruction fetches are considered reads to the instruction cache).

Both are fairly standard; first the L1 cache is checked, and if there isn't a hit, then L2 is checked. For writes, the access stops after the L2 access regardless of whether there was a hit or miss since per Dr. Mootaz, there are no DRAM writes and writeback is already handled in L2. Meanwhile, for reads, there is a DRAM access if there is an L2 read miss. If one trace is being run (i.e. executing python CacheSimulator.py <input file>), a report is generated at the end displaying all relevant access, energy, and performance statistics of the trace simulation. Executing the ./run.sh script generates such reports for each trace.

Implementation

The primary commonality among the three memory structures is their usage of a global clock, which is incremented on reads and is used to compute idle consumption by multiplying a given structure's idle power with the clock value. Below are key differences in the implementations of the three structures:

I. L1 Cache

The L1 cache read begins by indexing into the cache using the set and tag bits of the address before determining whether we have hit or missed in the L1 cache. All misses are handled the same in L1 because we have implemented a write-through procedure to L2, meaning there are no dirty writebacks. Thus, the only thing to do is to set the new tag and valid bits regardless of whether an eviction occurs or not (we have written methods for both cases as a sanity check, denoted by invalid_miss and evict). Our

implementation for the miss serves as the de facto cache fill which the L2 will handle and incur the penalty for. The accesses, active energy consumption, and clock are updated accordingly.

The L1 cache write is similar to the read; however, there are two key differences. Firstly, the clock is not updated because per Dr. Mootaz, all writes are asynchronous, which doesn't cause data issues due to sequential execution. In addition, upon achieving a write hit, we write through to the L2 cache, which inherently skews our L2 hit rate to be extremely high.

II. L2 Cache

The L2 cache access is different from the L1 access because L2 is set-associative and maintains a log of dirty bits. In L2, reads and writes are nearly identical, with the only difference being that an L2 write sets the dirty bit to true. Probing the cache consists of iterating through the selected set for tag matching. If we encounter an invalid block, we keep track of it to use in case of a miss. If we miss, we check if we have encountered an invalid index; if we have, we perform the cache fill in invalid_miss, else eviction ensues, which is done by the evict method.

Firstly, we evict a random block in the selected set. To ensure no blocks in L1 are not in L2, we then send an asynchronous back-invalidation to both the L1 data and L1 instruction caches. Finally, we check to see if the block we evicted is dirty in L2, in which case we send an asynchronous writeback to DRAM before filling the cache with the DRAM data that is read. Regardless of whether we hit or miss, we must transfer the data back to L1, which causes L2 to incur a transfer penalty of 5 pJ.

In terms of energy costs, we assume sequential probing between L1 and L2. Thus, the only time the L2's active consumption is updated is either on an L1 miss (due to the access) or an L1 write hit (due to the write through).

III. DRAM

Our DRAM implementation is very simple because it lacks a storage container since data is not used per project guidelines. We use DRAM for reads and writebacks; each implementation consists of updating accesses and active energy consumption, with reads also updating the clock. All DRAM accesses cause a 640 pJ transfer penalty to transfer the data to the caches, which is added to the active energy cost for each access. We have assumed that DRAM incurs this penalty for writebacks although the data is sent from L2.

Results

Our overall testing was made up of executing 10 runs for each trace, each with implementing L2 associativities of either 2, 4, or 8. We implemented this by generating a Pandas dataframe storing all relevant access, energy, and time data, which is implemented in our Table.py file and can be run with the command python Table.py.

Mean energy consumption for each trace varied between 0.002 to 0.004 J, averaging out at around 0.0028 J. There was minimal impact from changes in associativity; however, some cases did show a slight decrease in energy consumption with increasing associativities. This phenomenon showed between an associativity of 2 and 4, but not as much between 4 and 8. In these cases, active consumption decreases in L2 and DRAM are the primary cause of the lowered energy consumption, as L1 remains constant for all associativities.

Mean time for each trace ranged from 0.45 ms to 0.6 ms, averaging out at around 0.5 ms. Similar to energy consumption, the mean time wasn't impacted too much by associativity, but most cases did show a decrease in time as associativity increased, mostly between 2 and 4.

One interesting exception was the "026.compress" trace file, which actually saw an increase in both mean energy consumption and time as associativities increased, which we can probably attribute to the frequency of random evictions in the L2 cache since the primary culprit is the increase in mean DRAM energy consumption.

Hit rates in the L1 cache were pretty high, as expected, varying between 98-99% for the instruction cache and 94-99% for the data cache, averaging at around 99.6% and 98.2%, respectively. The L2 cache rates were also fairly high due to our write-through implementation skewing the results, as each write-through from L1 to L2 counts as a hit by the inclusivity principle. They ranged between 93-99% and averaged at around 97%. We previously implemented a writeback procedure between L1 and L2, which caused far lower and more variable rates ranging from 7% to 88%, but we have since replaced that procedure with the write-through.

Attached here is the table displaying our results: simulator_results