INDIAN INSTITUTE OF TECHNOLOGY ROPAR

COMPUTER ARCHITECTURE (CS 204)

Utility-Based Cache Partitioning

Contributors

Sourabh Sanganwar - 2020CSB1121

Vinay Kumar - 2020CSB1141

Tarushi - 2020CSB1135

Jatin - 2020CSB1090

Isha Goyal – 2020CSB1089

Course Instructor:

Dr. Shirshendu Das

Supervisor

Prathamesh Nitin Tanksale

Contents

Utility-Based Cache Partitioning (UCP)

Utility-Based Cache Partitioning is a low overhead, runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. The proposed mechanism monitors each application at runtime using a novel, cost-effective, hardware circuit that requires less than 2kB of storage. The information collected by the monitoring circuits is used by a partitioning algorithm to decide the number of cache resources allocated to each application.
This project focuses on the implementation of UCP-based cache partitioning in Champsim; a trace-based simulator.

The following are the additional modules required for the implementation of the same in the simulator:

- **Utility Monitor (UMON)**
  - Additional Tag Directory – Represented by umons, declared in cache.cc and implemented it using Dynamic Set Sampling (DSS) where each ATD has 32 sets and the underlying replacement policy is LRU.
  - UMON Global Counter – Represented by umons_counts, declared in cache.cc. It counts number of hits based on LRU position.
- **Partitioning Algorithm**: Implemented Lookahead Partitioning Algorithm

The following are the additional functions required for the implementation of UMON:

- **atd_replacement**
  - Parameters: cpu, set, tag
  - Return: NULL
  - It implements all functions of the additional tag directory, including checking hits, finding victims, and updating victims' LRU. It also updates the UMON global counter.
- **umon_lru_victim**
  - Parameters: CPU, set, address
  - Return: way
  - It finds the way of the victim block in ATD. Initially, it looks for an invalid block, however, if not found it returns the block with the maximum LRU value.
- **umon_lru_updat**
  - Parameters: CPU, set, way
  - Return: NULL
  - It updates the LRU values of the set containing the victim block in ATD.

The following are the additional functions required for updating LLC accordingly:

- **lru_victim**
  - Parameters: CPU, set, cycle count
  - Return: way
  - Initially, it checks whether the allotted partition by the partitioning algorithm needs to be updated or not using the cycle count. If needed it gets updated using the function get_new_part. Then eviction of the block takes place considering the allotted partition and the current partition of ways for cpu's in the corresponding set.
- **lru_victim_llc**
  - Parameters: CPU, set
  - Return: way
  - It finds the way of the victim block. Initially, it looks for an invalid block, however, if not found it returns the block with the maximum LRU value.
- **lru_update**
  - Parameters: CPU, set, way
  - Return: NULL
  - It updates the LRU values of the set containing the victim block.

The following are the additional functions required for the implementation of the lookahead based Partitioning Algorithm:

- **getmumax**
  - Parameters: CPU, number of unassigned ways and an array containing the
  - Return: NULL
  - The function finds the Marginal Utility. If $miss_a$ and $miss_b$ are the number of misses that an application incurs when it receives a and b ways respectively (a < b), then the Marginal Utility of increasing the number of ways from a to b is defined as $U_a^b = (miss_a - miss_b)/(b-a)$
- **get_new_part**
  - Parameters: CPU, way
  - Return: NULL
  - The function finds the optimal partitioning for the given number of CPUs and the associativity of the set using Marginal Utility. For each CPU and for each possible partition, it calculates the marginal utility and the partition that gives the maximum Marginal Utility with the minimum number of blocks is the optimal partition.

How to run the code?

The following is the structure to compile the file

$ ./build_champsim.sh ${BRANCH} ${L1I_PREFETCHER} ${L1D_PREFETCHER} ${L2C_PREFETCHER} ${LLC_PREFETCHER} ${LLC_REPLACEMENT} ${NUM_CORE}

Ex:

```
$ ./build_champsim.sh bimodal no no no no lru 1
```

The following is the structure to run the file

./run_champsim.sh [BINARY] [N_WARM] [N_SIM] [TRACE] [OPTION]

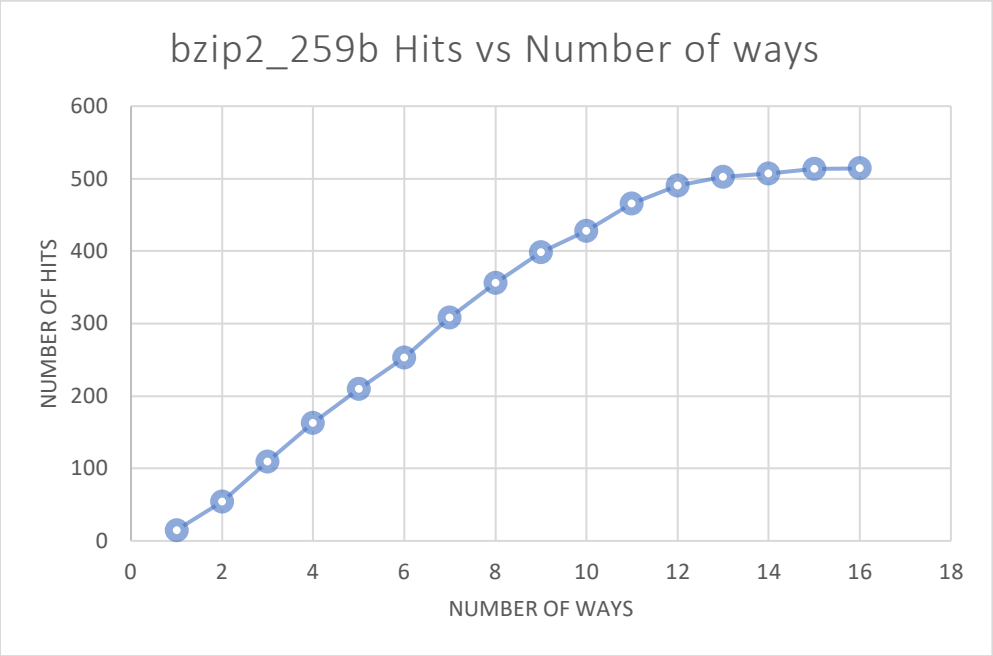
Ex.

```
$ ./run_champsim.sh bimodal-no-no-no-no-lru-1core 1 10 400.perlbench-
41B.champsimtrace.xz
```

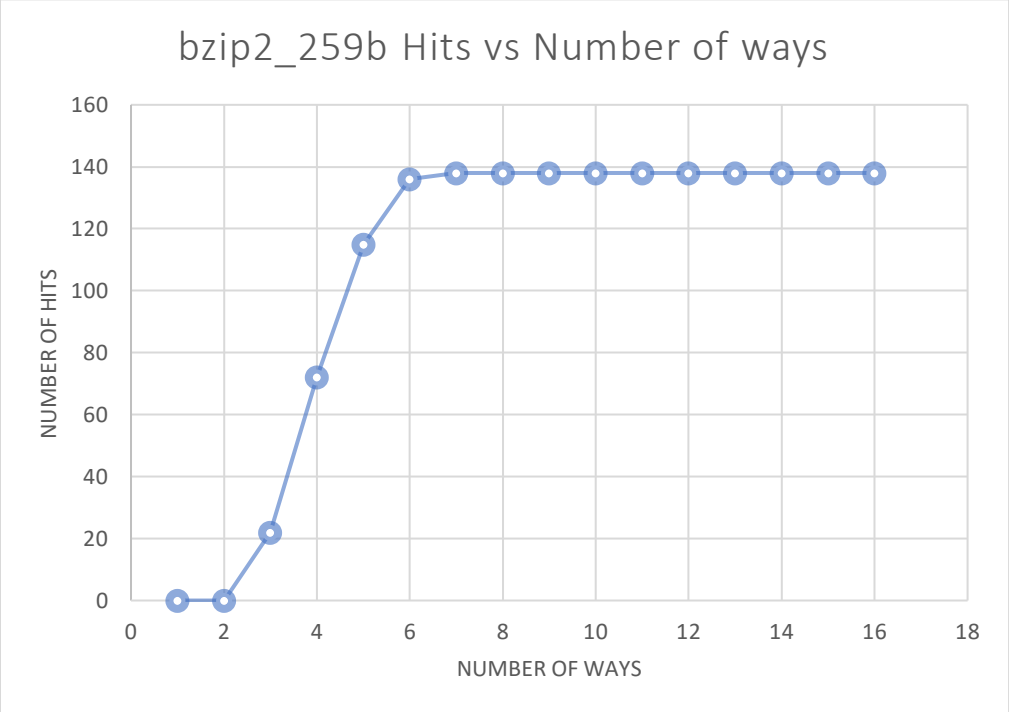
Statistics

We have implemented UCP for 2 cores with different number of instructions. Traces of each core are bzip2_259B and lbm_1004B respectively. Following are the references from the results.

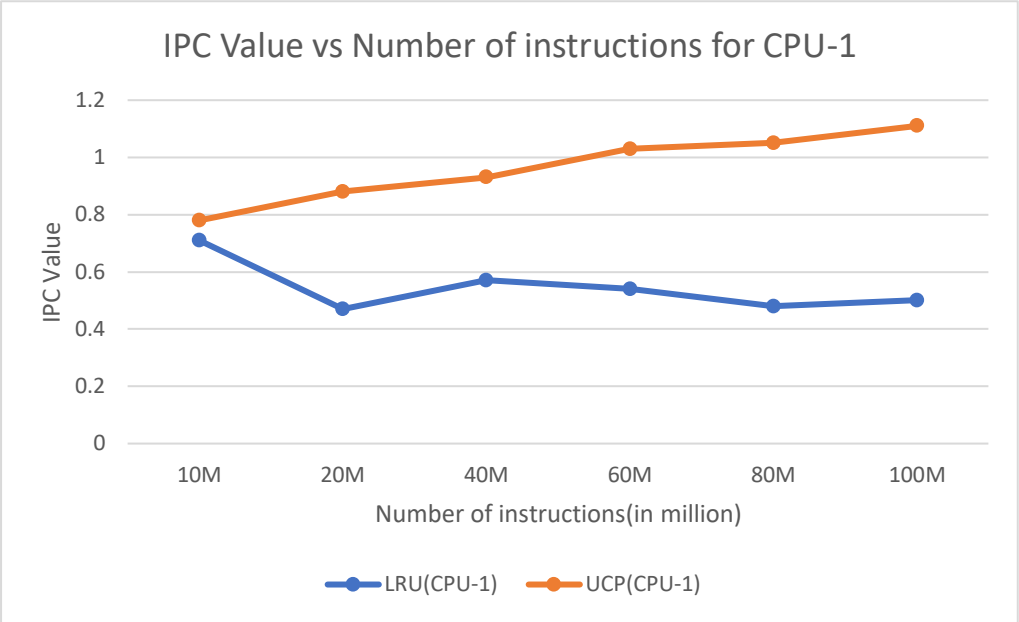
The following graph shows utility of bzip2_259B trace



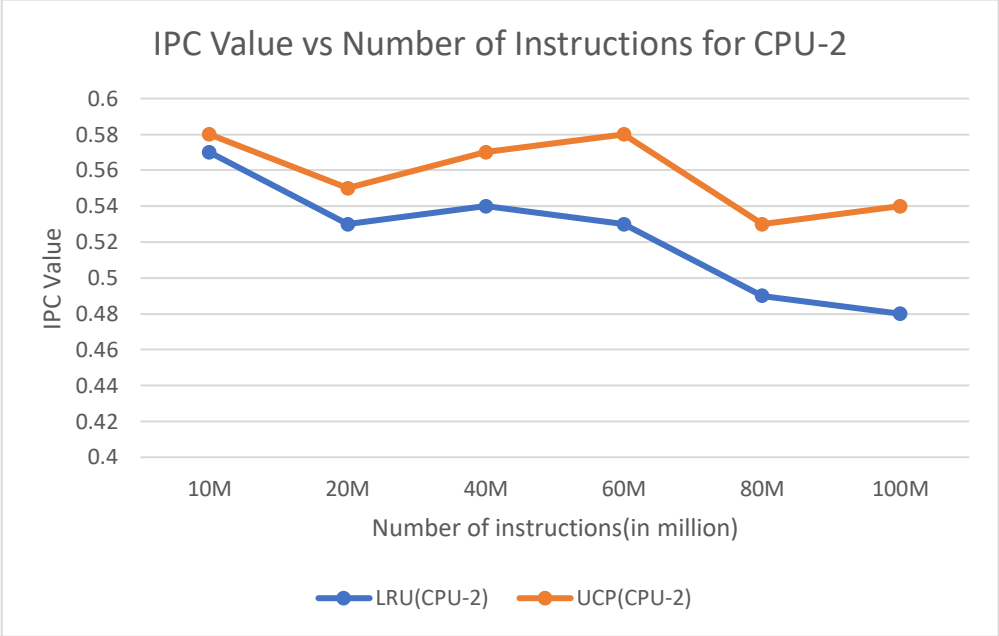
Following graph shows utility of lbm_1004B trace



The following graph is for IPC Value vs Number of instructions for CPU running bzip2_259B trace



The following graph is for IPC Value vs Number of instructions for CPU running lbm_1004B trace



The average number of ways allotted by UMON for CPU-1 (bzip2_259B) are 13 and for CPU-2(lbm_1004B) are 3.

From the above statistics, we can see there is a substantial increase in IPC value from LRU to UCP.

References:

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4041865