ReDHiP: Recalibrating Deep Hierarchy Prediction for Energy Efficiency

Presented by Brett Duncan

Outline

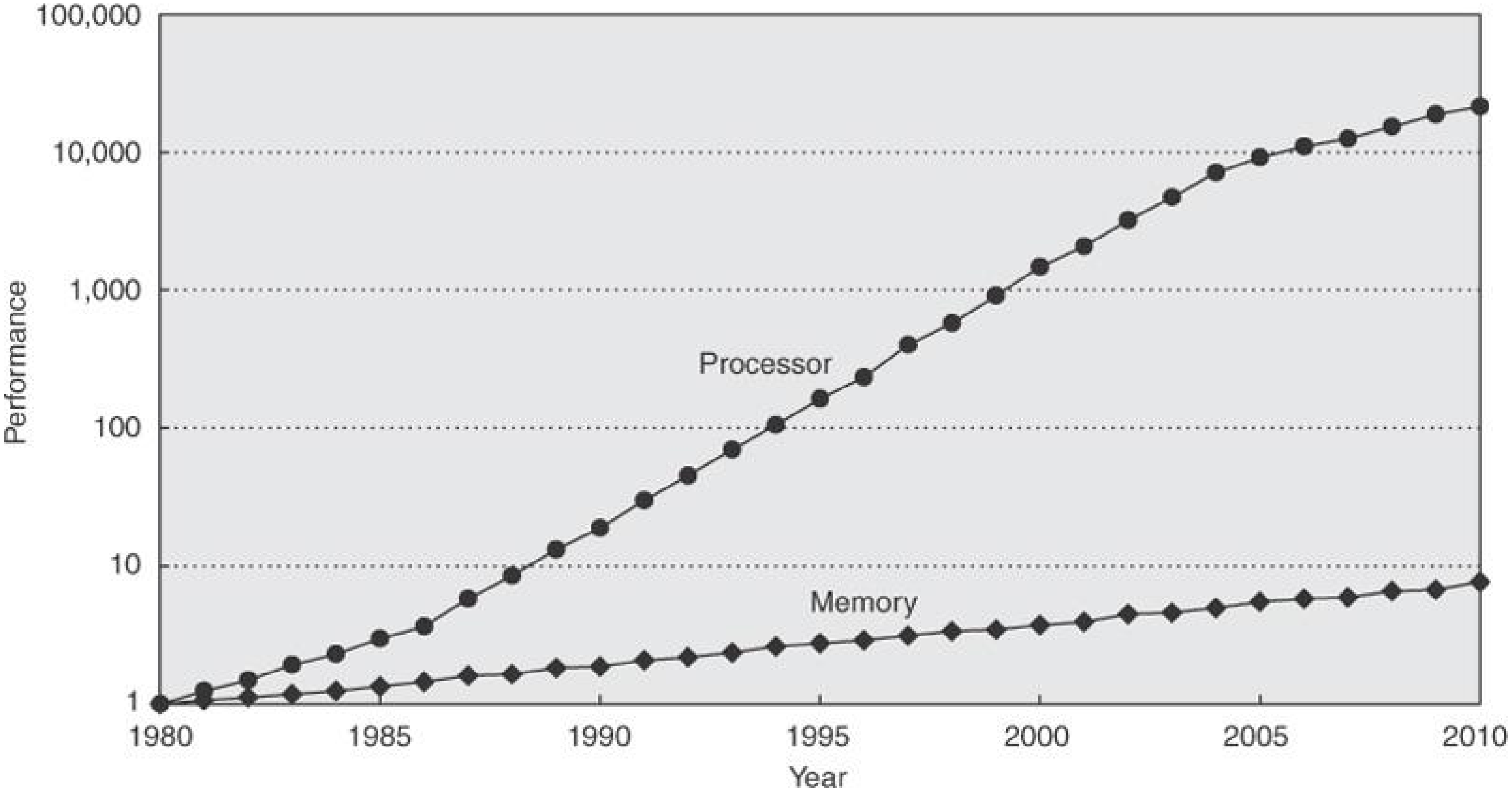
- **Motivation**
- Previous Work
- Design/Implementation
- Methodology
- Results

Motivation

- Recent hardware trends point to increasingly deeper cache hierarchies, so accesses that lookup and miss in every cache involve significant energy consumption and degraded performance.

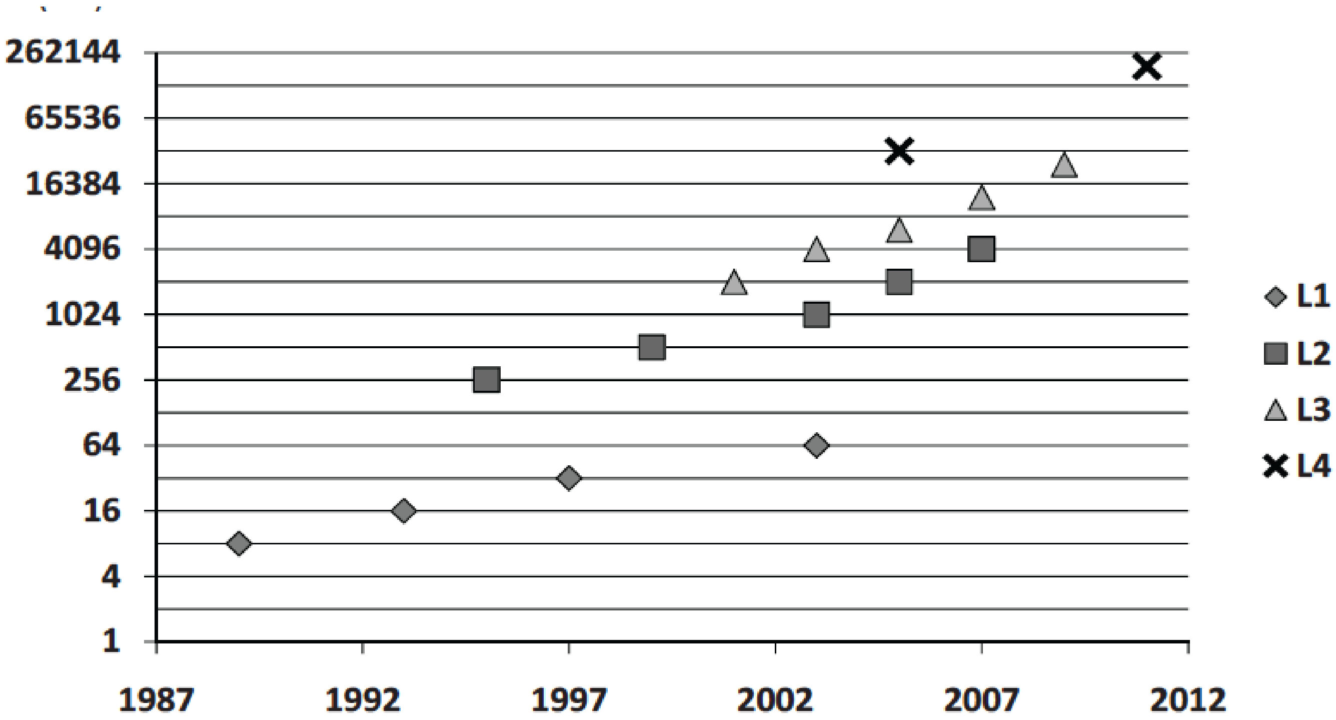
Motivation (Hardware Caches)

- Latency gap between register and memory.
- Design trade-off between size and cost.



© 2007 Elsevier, Inc. All rights reserved.

Motivation (Deep Cache Hierarchy)



- Increasing number of cores → increasing memory demand.

Motivation

- Example Cache Hierarchy: Four cores have their own L1, L2, and L3 cache. They all share the L4 cache.
- L1 is the fastest and smallest cache. The deeper into the hierarchy, the caches get slower but increase in size.

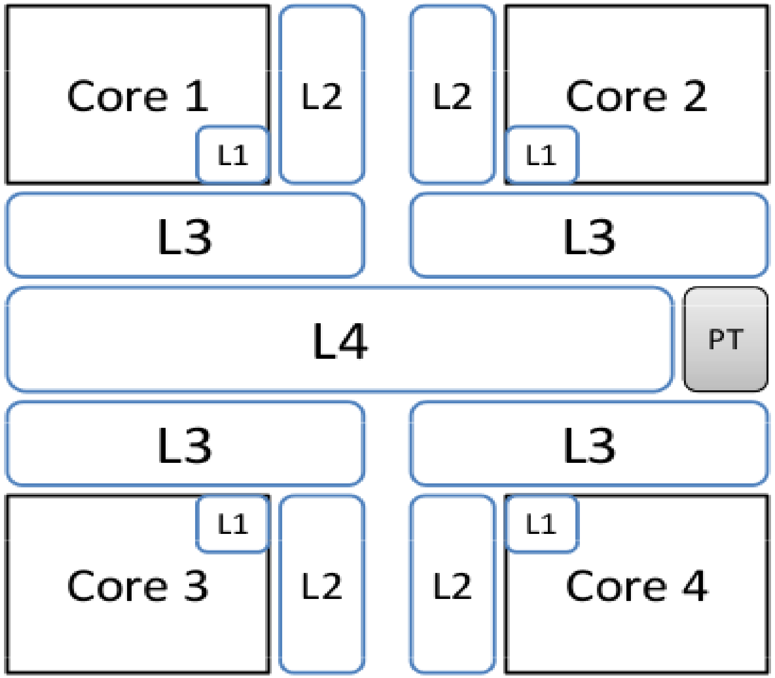


Figure 2. An example Deep Cache Hierarchy in a 4-core processor; each core has a private L1, L2 and L3 cache. All cores share the same L4 cache. A prediction tables (PT) is located aside L4.

Outline

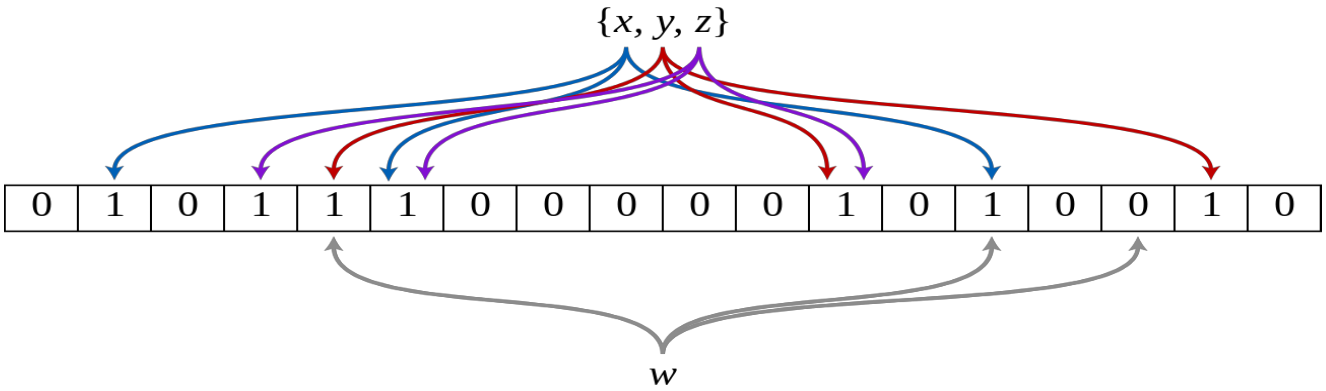
- Motivation
- **Previous Work**
- Design/Implementation
- Methodology
- Results

Previous Work

- Data Existence Prediction
  - On cache access, use hash of the address to index a prediction table and predict data presence.
  - Often complex hashing mechanisms, e.g.:
    - xor-hash
  - Often complex data structures, e.g.:
    - Counting Boom Filter
    - High cost and storage overhead

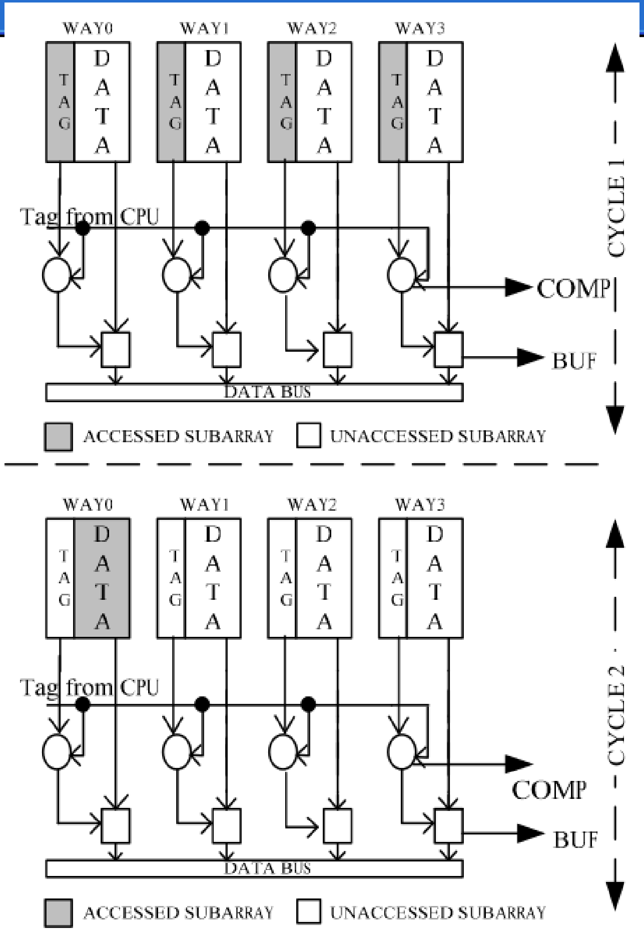
Previous Work

- Counting Bloom Filter
  - Variant of a Bloom filter, in which each table entry is a counter instead of a single bit.
  - When adding an address, counters from each hashed index are all incremented. When deleting an address, counters from each hashed index are decremented.



Previous Work

- Phased Cache
  - Tag subarrays are compared with the tag from the CPU. If there is a hit, then the corresponding data subarray is accessed and the data will be available on the data bus in the next clock cycle.



Outline

- Motivation
- Previous Work
- **Design/Implementation**
- Methodology
- Results

Design/Implementation

- ReDHiP: Recalibrating Deep Hierarchy Prediction for Energy Efficiency
  - Architectural mechanism that predicts last-level cache (LLC) misses in advance, with small hardware overhead.
  - Augments the LLC with a hardware look-up table predicting LLC data presence based on data addresses.
  - After each L1 cache miss, a ReDHiP look-up is performed.
  - If ReDHiP finds the data does not exist in the LLC, then all lower levels of cache are skipped.
  - Else, the miss will proceed as before, beginning with the L2 cache.
  - To provide accuracy with low energy, a small table with infrequent, low-cost recalibration is used.

Design/Implementation

- ReDHiP uses a prediction table that contains presence information about data in the last level cache.
- Table is accessed after every L1 cache miss.

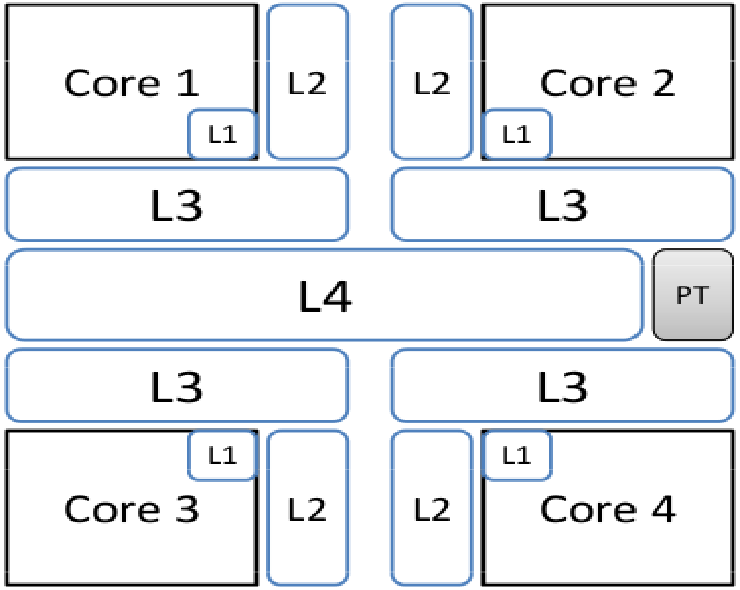


Figure 2. An example Deep Cache Hierarchy in a 4-core processor; each core has a private L1, L2 and L3 cache. All cores share the same L4 cache. A prediction tables (PT) is located aside L4.

Design/Implementation

- The Prediction Table (PT) is located aside the L4 cache to allow fast communication with it.
- All levels of cache are inclusive, meaning every level of cache contains data from upper levels.

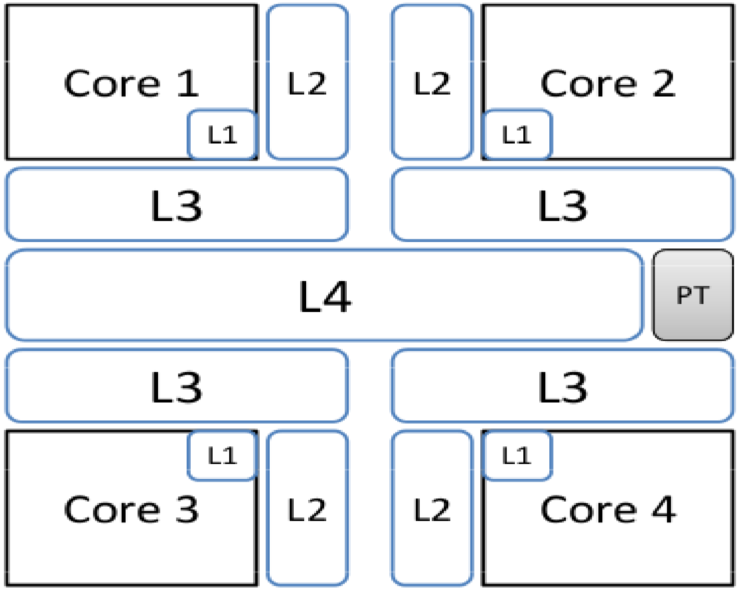
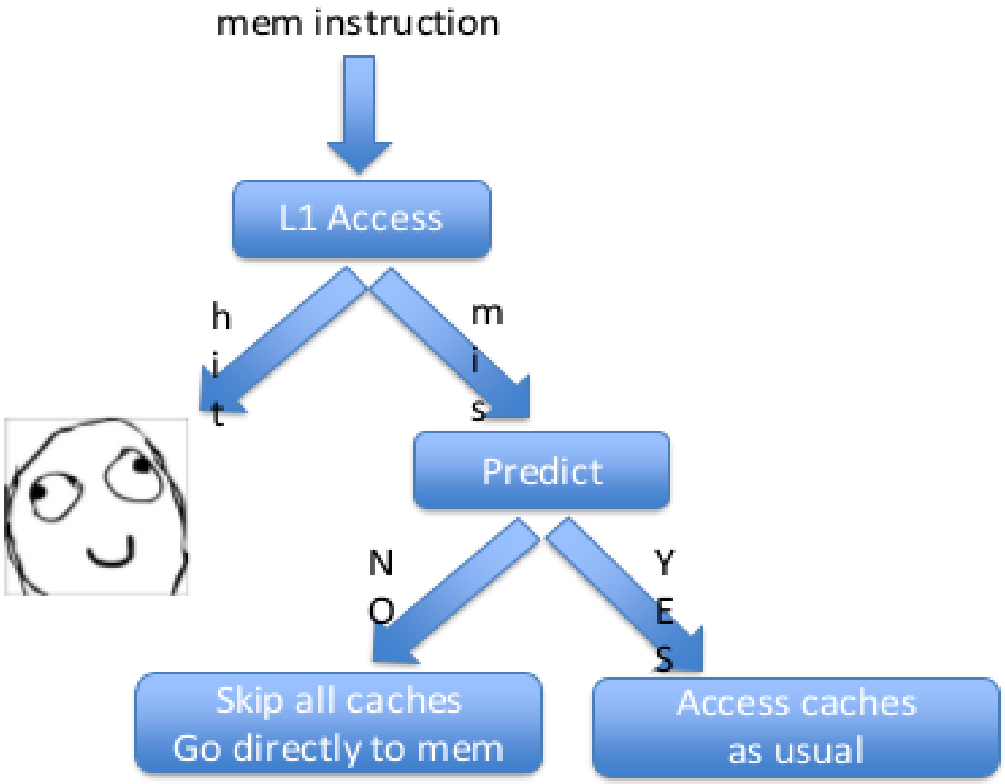


Figure 2. An example Deep Cache Hierarchy in a 4-core processor; each core has a private L1, L2 and L3 cache. All cores share the same L4 cache. A prediction tables (PT) is located aside L4.

Design/Implementation (Memory access process)

- L1 cache is accessed first. On L1 cache miss, the prediction table is accessed.
  - If data is in the LLC, then data can be in any level above, and the L2 cache is accessed next.
  - If the prediction table indicates that data is *not* in the LLC, then it can conclude that the data is not located in any cache (due to inclusiveness), ensuring no false negatives can happen.



Design/Implementation

- A very simple prediction table design coupled with the simplest prediction mechanism, despite losing accuracy over time, enables an extremely efficient recalibration algorithm, leading to high accuracy.
- This is the key insight to the ReDHiP approach, which presents an interesting tradeoff between accuracy (incurring mispredictions) and recalibration (incurring overhead).

Design/Implementation (Prediction)

- Predictions in ReDHiP are made based on data addresses.
  - Hash values of requested addresses are calculated and indexed to the corresponding entry in the table.
- There are several important design parameters: table structure, entry width, and hash function.

Design/Implementation

- Table structure
  - To minimize energy, a direct-mapped table is used. The computed hash value of a given address is used directly as an index to the prediction table.
- Entry width
  - Instead of having counters keep track of the number of entries with the same hash, a bit map is used. A bit is set to one when an entry is added, but it is not updated to reflect eviction.
- Hash function:
  - Prior work such as CBF uses xor hash due to higher accuracy. A simple hash function that uses only partial address bits enables an inexpensive recalibration that is not possible with xor hashes. For ReDHiP, the hash value is the lowest P bits of address after the block offset has been removed, referred to as bits-hash, as shown in Figure 3.

Design/Implementation

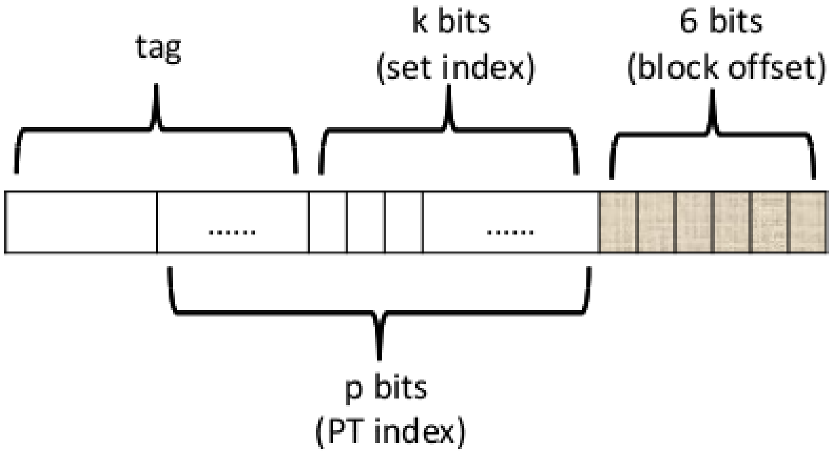
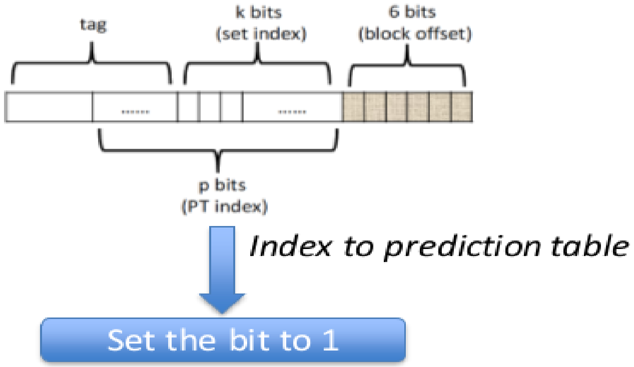


Figure 3. Data Address Format: the last 6 bits are the block offset (assuming 64-bytes block size), which is never used for indexing; the next k bits are the cache set index (2^k sets in the cache); the remaining bits are tag bits. With bits-hash, the lowest p bits after the block offset are used as index to the table. The PT index will always contain the set index as a substring, as long as $p > k$.

Design/Implementation

- Recalibration
  - Important to offset the simple design decisions that lead to an increase of false positives over time.
  - By recalibration, we reconstruct all values in the prediction table to reflect the up-to-date presence information in the LLC.



Adding block to cache

Design/Implementation (Recalibration)

- With traditional Bloom filters, the location of the cache lines that affect the entries would be unknown.
- It is not possible to parallelize these operations, and each tag operation would take several cycles.
- The total delay spent on recalibration for traditional predictors could be as large as several million cycles, costing large energy consumption.
- The ReDHiP hash function eliminates these problems and provides low-cost recalibration.

Design/Implementation (Recalibration)

- The prediction table address and cache set index share the same last k bits, so for each tag in a given cache set, we only need p – k bits to determine the location in the pre
- A 512K prediction table and 64MB LLC with 16-wa associativity is used in the base design.
  - p = 22; k = 16; p – k = 6
  - A continuous range of 64 (2^6) bits is defined as one li prediction table.

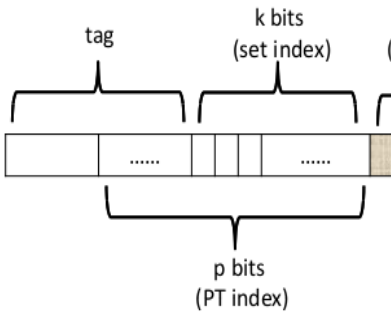


Figure 3. Data Address Format: the last 6 bit
(assuming 64-bytes block size), which is never used
k bits are the cache set index (2^k sets in the cache);
tag bits. With bits-hash, the lowest p bits after the
as index to the table. The PT index will always co
substring, as long as $p > k$.

Design/Implementation (Recalibration)

- The index in the cache is a subset of the index in the prediction table, so we need 6 more bits to determine the location in the prediction table.
- To recalibrate a hash set, 6 bits from each of the 16 tags in the set are used as index to set the bits in the corresponding prediction table line, as shown in the figure.

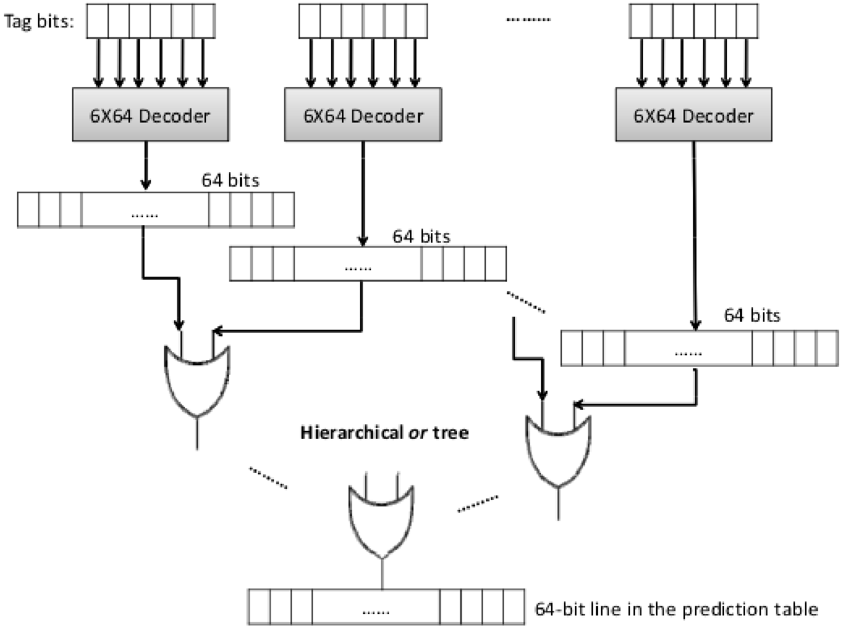
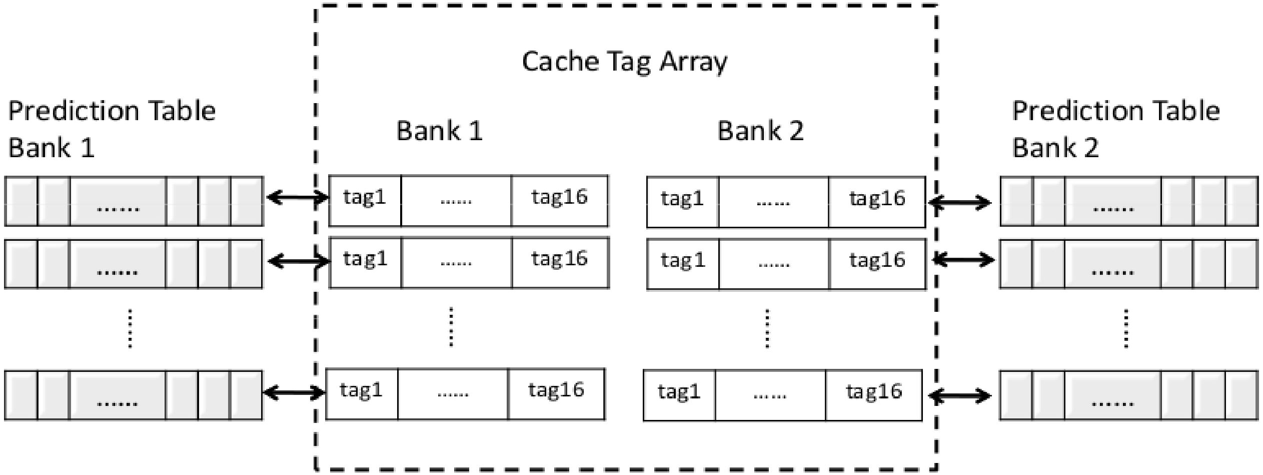


Figure 4. Hardware required to recalibrate all 16 entries in one cache set. Because the set index (16 bits) is a subset of the prediction table hash (22 bits), only 6 bits of the tag are needed. A decoder is used to expand it to a 64-bit vector with its entry set. Those 64 bit vectors are combined with a simple hierarchical *or* operation.

Design/Implementation (Recalibration)



- Because a single set is all that's needed to update a single line in the prediction table, it can be parallelized.
- The banking structure of most modern caches can be taken advantage of to recalibrate cache sets from multiple banks in parallel as shown in the figure.

Design/Implementation (Exclusive and Hybrid Cache)

- Exclusive
  - Every level of cache contains distinct data
  - Data that does not exist in the LLC might still reside in upper level caches.
  - Prediction table needs to be duplicated (and scaled down correspondingly to cache size) for every cache except L1 to predict the data presence at each level.
  - Each lower level cache starting from L2 will have a prediction table and same storage overhead ratio (0.78% for this evaluation).
  - On L1 miss, the prediction tables from every level down the hierarchy is requested simultaneously.
  - All levels that predict true residency will be accessed in sequence, and levels that predict false will be skipped.
  - Energy overhead is small due to the constant overhead ratio for each predictor/cache pair, and has the benefit that the request is sent to the lowest level where it may exist rather than always restarting at the L2 cache.

Design/Implementation (Exclusive and Hybrid Cache)

- Hybrid
  - In multi-core processors, the shared LLC can be implemented as either inclusive or pseudo-exclusive in a way that provides certain inclusive guarantees.
  - The upper private level caches can be exclusive with each other, but they must all be inclusive with the shared LLC.

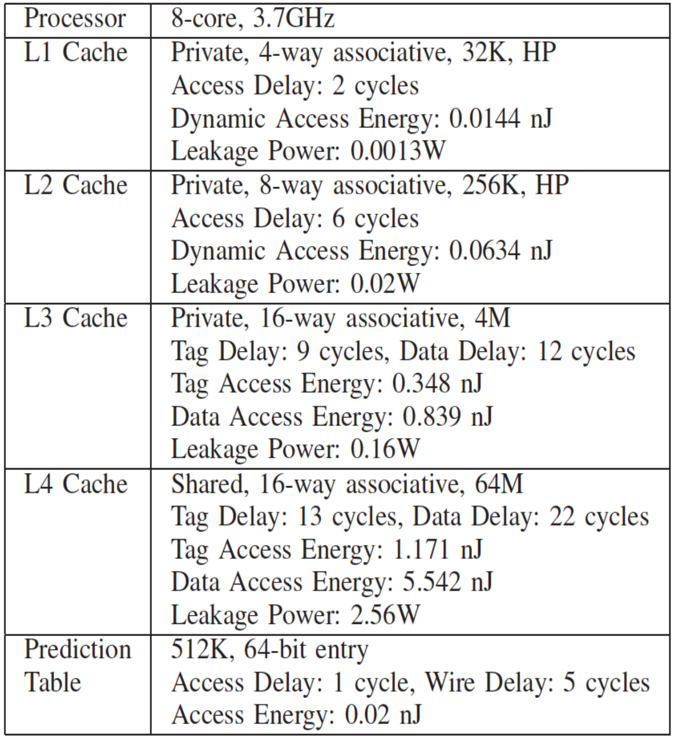
Outline

- Motivation
- Previous Work
- Design/Implementation
- **Methodology**
- Results

Methodology

- Simulation Setup
  - Cacti to model latency and energy consumptions
  - Cache simulator
    - Main memory not modeled
    - Cache is inclusive
  - Benchmarks
    - SPEC 2006
      - astar, bwaves, cactusADM, GemsFDTD, lmb, mcf, milc, soplex, *mix*
    - Graph500
    - Cominatorial BLAS Library
    - Use Pin to collect memory traces: 500M

Methodology

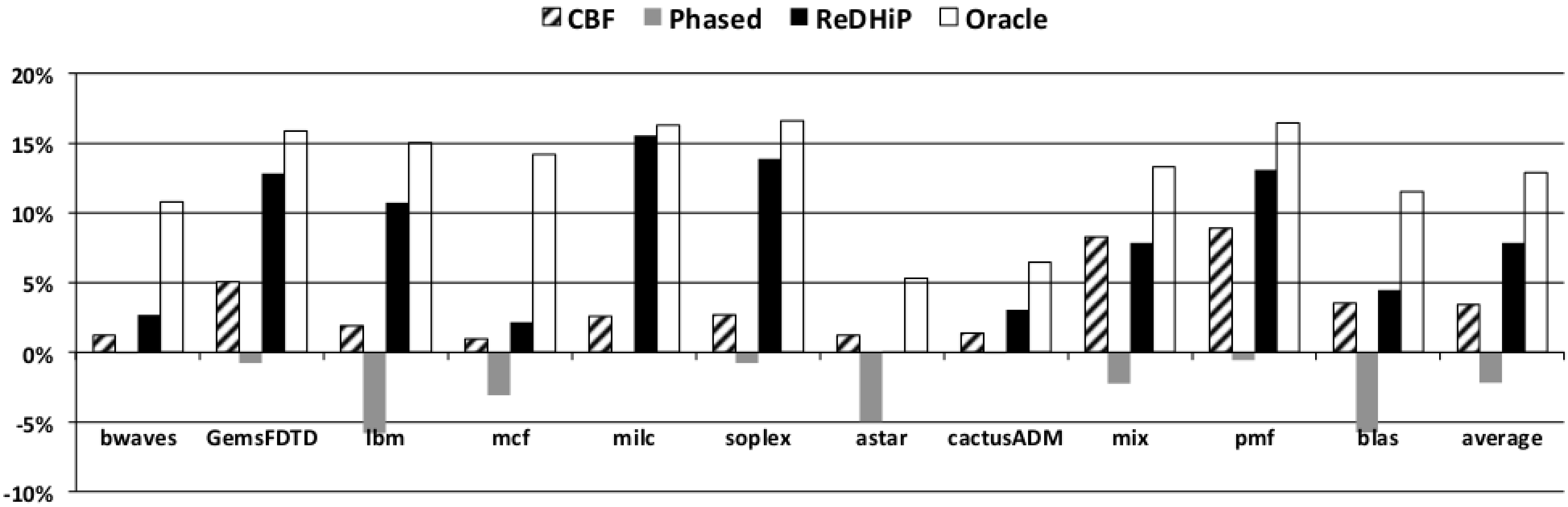


Processor	8-core, 3.7GHz
L1 Cache	Private, 4-way associative, 32K, HP Access Delay: 2 cycles Dynamic Access Energy: 0.0144 nJ Leakage Power: 0.0013W
L2 Cache	Private, 8-way associative, 256K, HP Access Delay: 6 cycles Dynamic Access Energy: 0.0634 nJ Leakage Power: 0.02W
L3 Cache	Private, 16-way associative, 4M Tag Delay: 9 cycles, Data Delay: 12 cycles Tag Access Energy: 0.348 nJ Data Access Energy: 0.839 nJ Leakage Power: 0.16W
L4 Cache	Shared, 16-way associative, 64M Tag Delay: 13 cycles, Data Delay: 22 cycles Tag Access Energy: 1.171 nJ Data Access Energy: 5.542 nJ Leakage Power: 2.56W
Prediction Table	512K, 64-bit entry Access Delay: 1 cycle, Wire Delay: 5 cycles Access Energy: 0.02 nJ

Outline

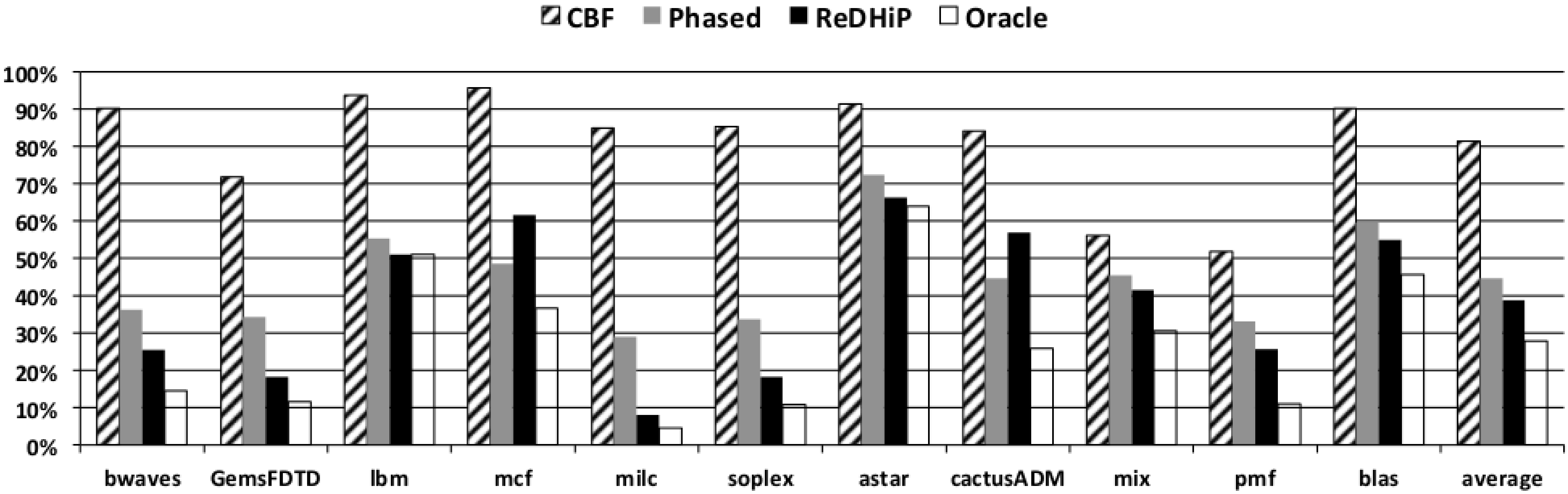
- Motivation
- Previous Work
- Design/Implementation
- Methodology
- **Results**

Results (Performance)



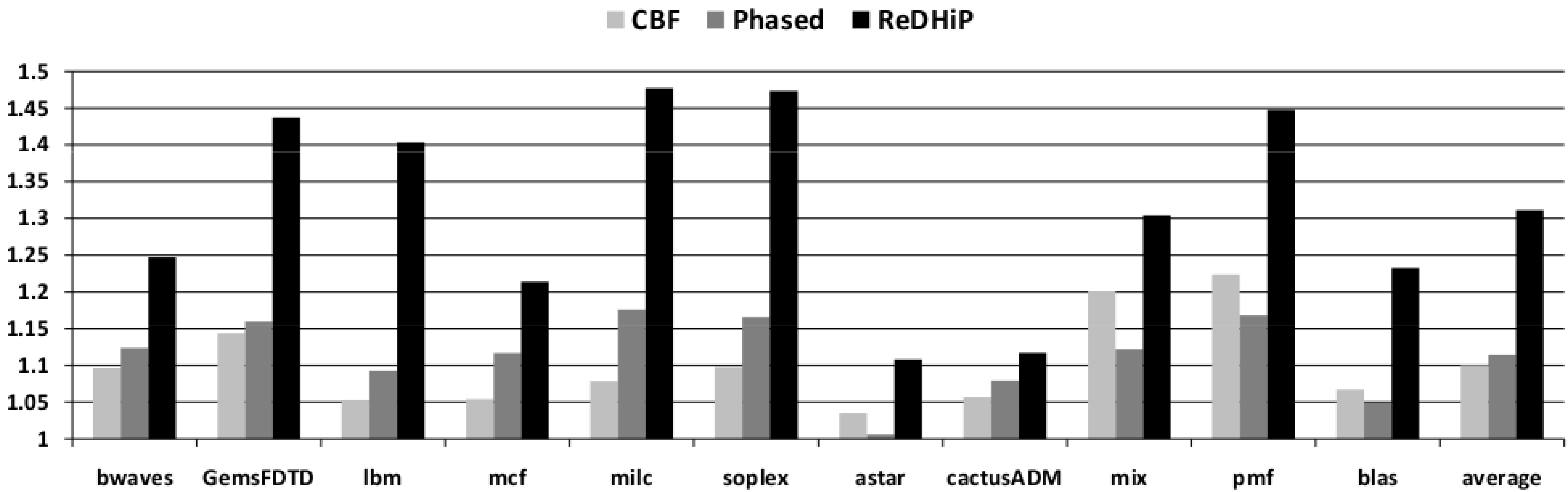
- Performance Speedup of: Oracle (Perfect predictor), CBF (counting bloom filter), phased cache, ReDHiP. Compared against a base case in which no prediction/optimization is used.

Results (Dynamic Energy Consumption)



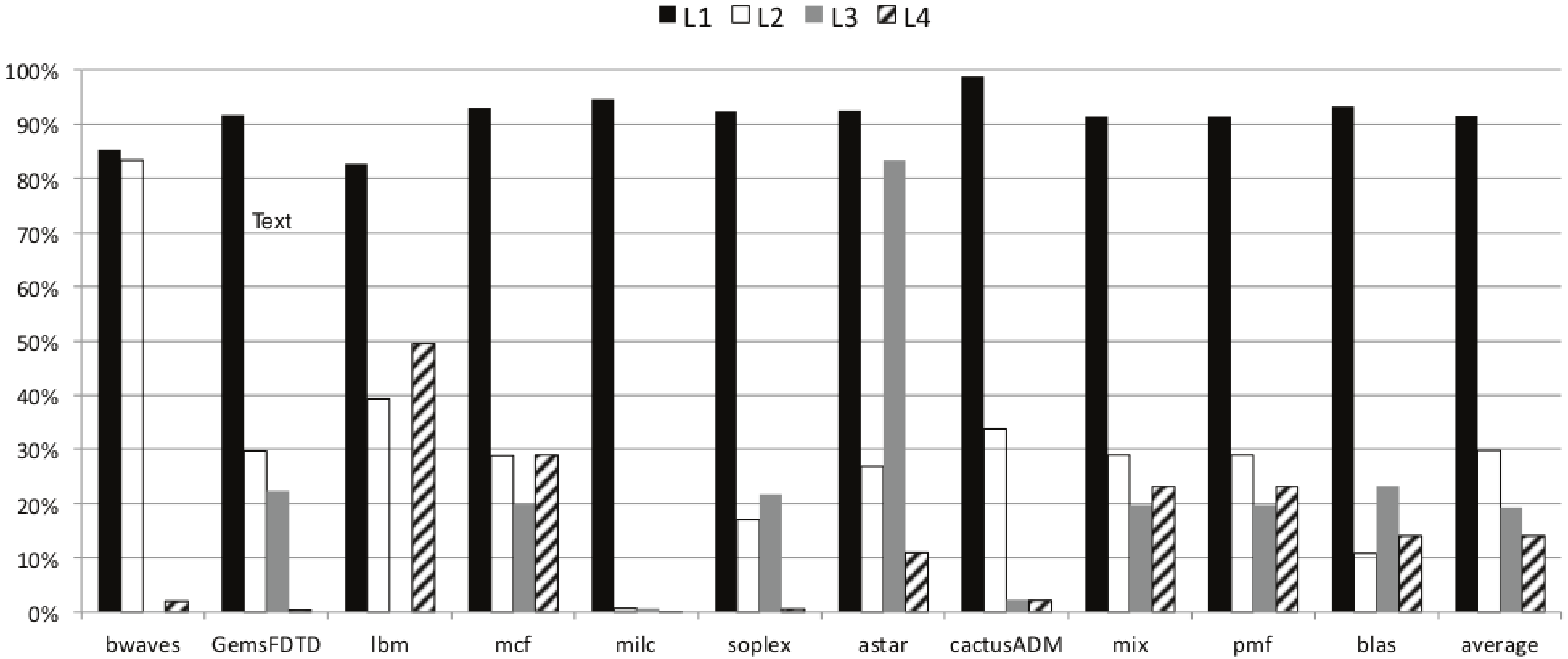
- Dynamic energy consumption of: Oracle, CBF, Phased Cache, and ReDHiP, normalized to the base case.

Results (Performance Gain * Energy Savings)



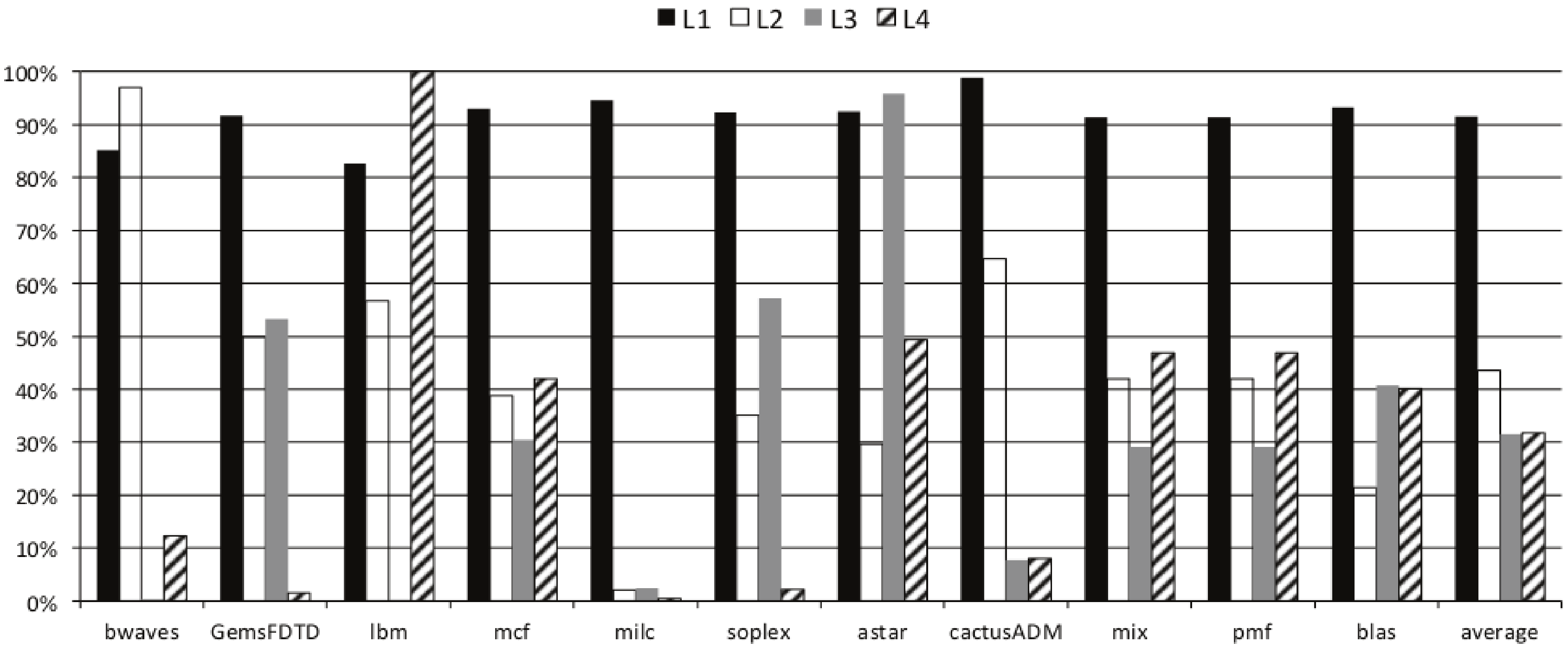
- Performance-energy metric of CBF, Phased Cache, and ReDHiP. (performance gain * energy savings). Higher is better.

Results (Cache Hit Rate / Base)



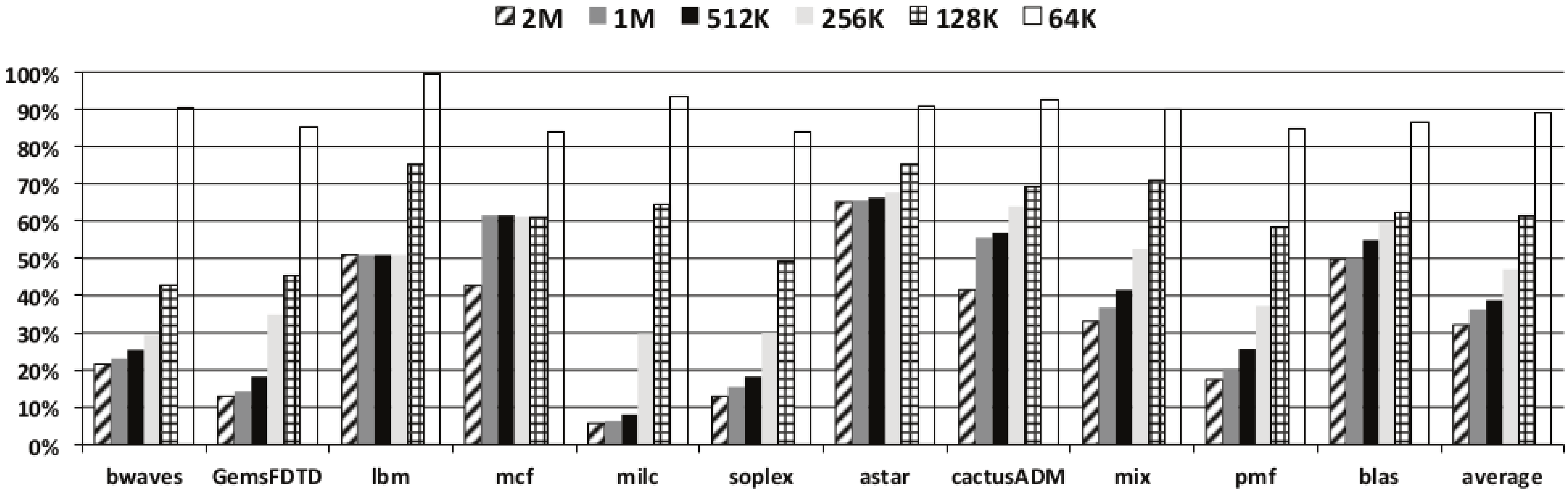
- Hit rate of each level cache for all benchmarks in the base case where no prediction mechanism is used.

Results (Cache Hit Rate / ReDHiP)



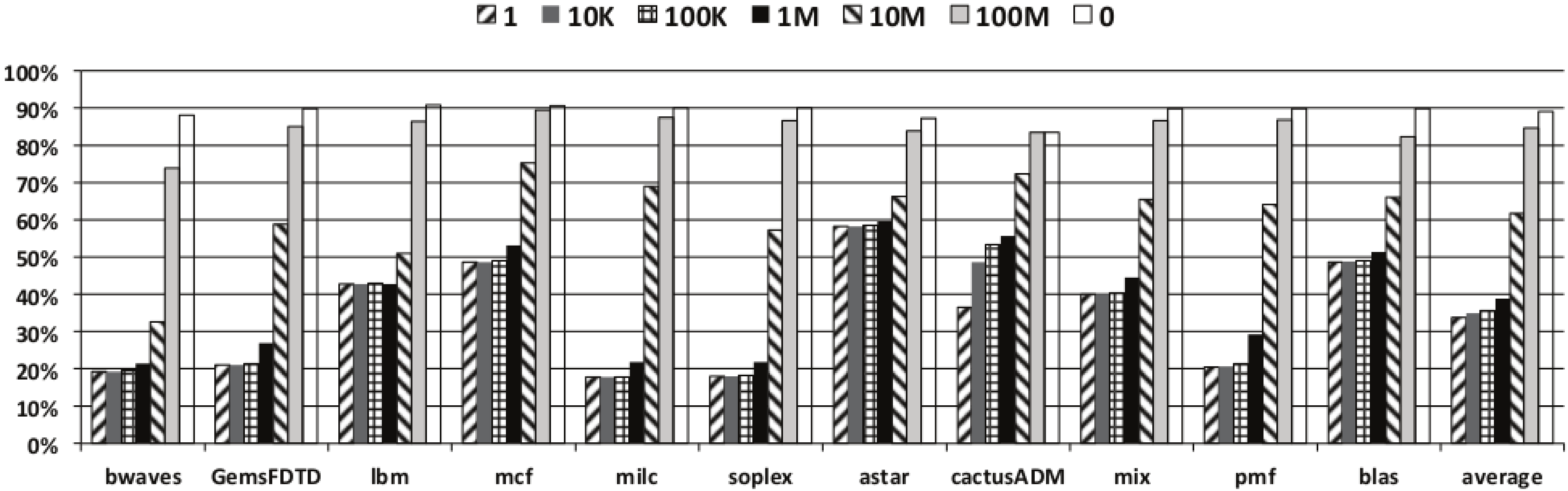
- Hit rate of each level cache for all benchmarks when ReDHiP is applied.

Results (Dynamic Energy Consumption)



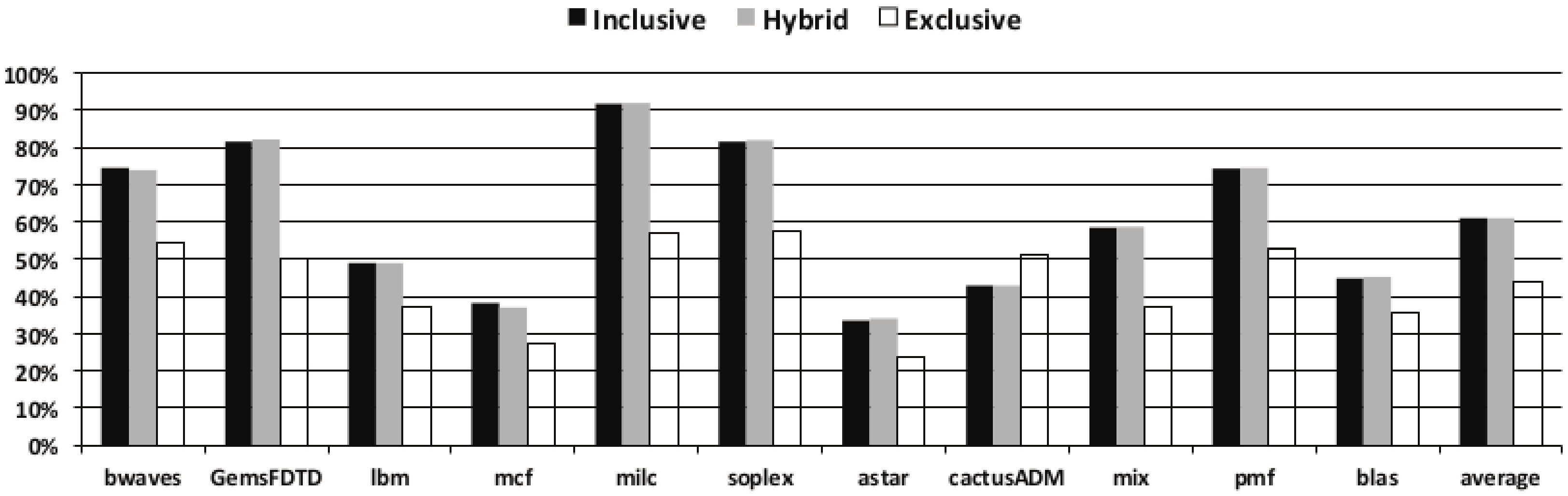
- Dynamic energy consumption of ReDHiP under different prediction table sizes. Normalized to the base case where no prediction mechanism is used.

Results (Dynamic Energy Consumption)



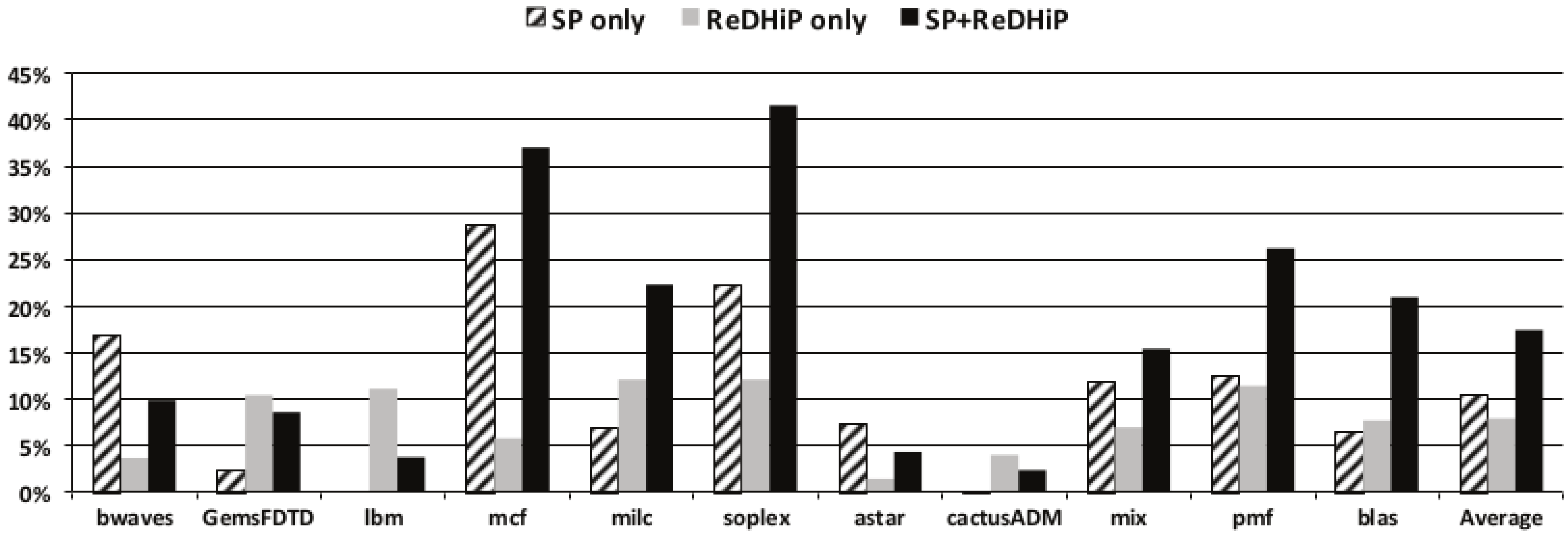
- Dynamic energy consumption of ReDHiP under different recalibration frequencies (i.e. number of L1 misses between two recalibration), normalized to the base case where no prediction mechanism is used. 1 means recalibrating after every L1 miss (perfect recalibration), 0 means no recalibration.

Results (Dynamic Energy Savings)



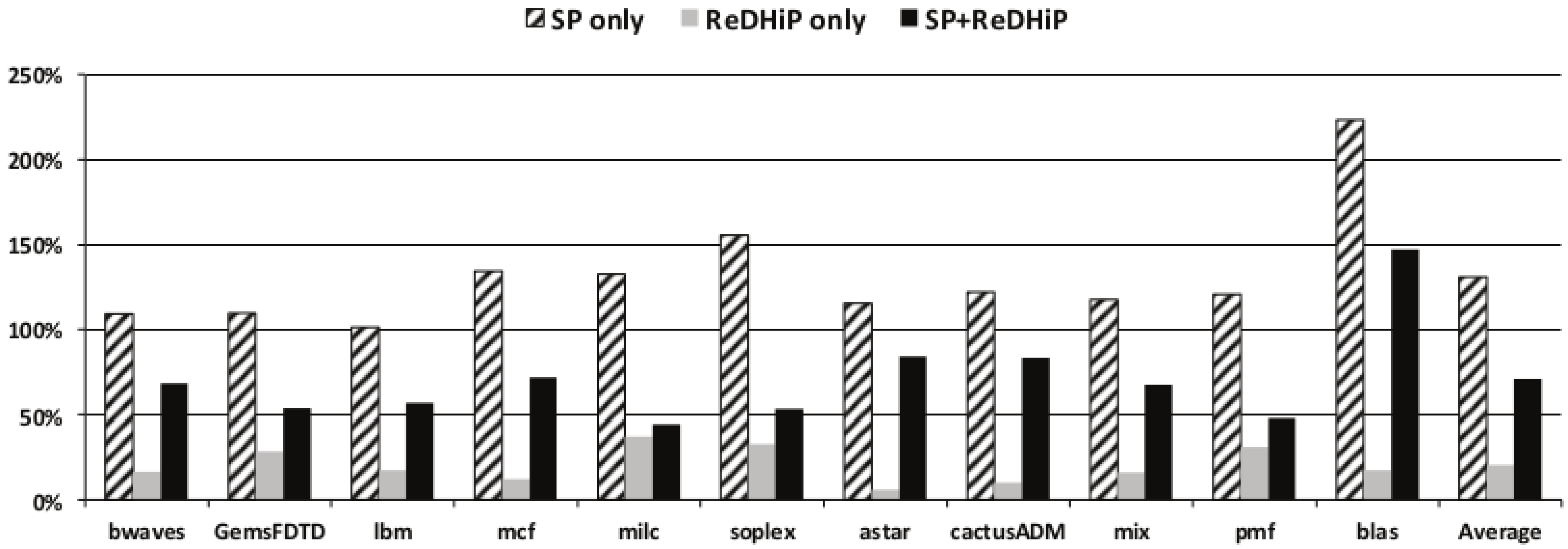
- Dynamic energy savings of ReDHiP under different cache inclusion policies: fully inclusive, a hybrid architecture (in which L2 and L3 exclusive while L4 is inclusive), and fully exclusive. All results are normalized to the base case in which no prediction mechanism is used.

Results (Dynamic Energy Consumption)



- Performance speedup of: SP (stride prefetch) only, ReDHiP only (prediction only), SP+ReDHiP (both prediction and prefetching normalized to a base case).

Results (Dynamic Energy Consumption)



- Dynamic energy consumption of: SP (stride prefetch) only, ReDHiP only (prediction only), SP+ReDHiP (both prediction and prefetchings).

References

[1] Xun Li; Franklin, D.; Bianchini, R.; Chong, F.T., "ReDHiP: Recalibrating Deep Hierarchy Prediction for Energy Efficiency," Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pp.915,926, 19-23 May 2014

[2] Megalingam, R.K.; Deepu, K.B.; Joseph, I.P.; Vikram, V., "Phased set associative cache design for reduced power consumption," 2nd IEEE International Conference on Computer Science and Information Technology, 2009. ICCSIT 2009. pp.551, 556, 8-11 Aug. 2009