ReDHiP: Recalibrating Deep Hierarchy Prediction for Energy Efficiency

Xun Li[1]∗ Diana Franklin[2] Ricardo Bianchini[3] Frederic T. Chong[2]

[1]*Facebook*
Menlo Park, CA, USA
xun@fb.com

[2]*University of California, Santa Barbara*
Santa Barbara, CA, USA
{franklin, chong}@cs.ucsb.edu

[3]*Rutgers University*
Piscataway, NJ, USA
ricardob@cs.rutgers.edu

***Abstract*— Recent hardware trends point to increasingly deeper cache hierarchies. In such hierarchies, accesses that lookup and miss in every cache involve significant energy consumption and degraded performance. To mitigate these problems, in this paper we propose Recalibrating Deep Hierarchy Prediction (ReDHiP), an architectural mechanism that predicts last-level cache (LLC) misses in advance. An LLC miss means that all cache levels need not be accessed at all. Our design for ReDHiP focuses on a simple, compact prediction table that can be efficiently recalibrated over time. We find that a simpler scheme, while sacrificing accuracy, can be more accurate per bit than more complex schemes through recalibration. Our evaluation shows that ReDHiP achieves an average of 22% cache energy savings and 8% performance improvement for a wide range of benchmarks. ReDHiP achieves these benefits at a hardware cost of less than 1% of the LLC. We also demonstrate how ReDHiP can be used to reduce the energy overhead of hardware data prefetching while being able to further improve the performance.**

***Keywords*-Last Level Cache; Energy; Performance**

I. Introduction

Chip multiprocessors (CMPs) are both cause and consequence of the changing constraints facing computer architects. Our work is motivated by the two following trends relating to these constraints. First, future CMPs will be more power-limited than today's CMPs. Second, the continually increasing gap between the CMP and external memory will lead to larger, deeper memory hierarchies. Third, the increasing depth of the hierarchy coupled with highly-threaded workloads will lead to more costly and numerous off-chip accesses.

While the initial CMPs were able to significantly reign in power consumption, we will soon (if we have not already) enter the "Dark Silicon" era [6]. The failure of Dennard scaling indicates that future processor designs will be power limited, and any performance gains must come with affordable power overhead. Cache designs are reflecting this trend, choosing at the lower levels of the hierarchy to access tags before accessing data because of the power required to access large, highly associative caches (Phased Cache[11]).

∗ The work is done when Xun Li is a graduate student at University of California, Santa Barbara.

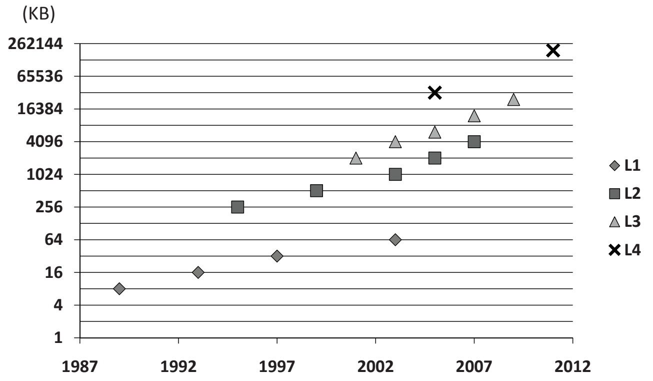

Figure 1. The size of different levels of hardware caches along with their year of appearance (roughly) in commercial processors. More levels of cache were introduced in the past decade; L4 caches have started to appear.

At the same time, on-chip caches are covering an increasing percentage of the chip, both because of their relatively low power consumption per area and to support increasing thread counts. Traditionally, the highest level caches were designed to minimize access time, and the lowest level caches were designed to maximize hit rate, with tradeoffs in the middle level cache (if any). With the rise in CMPs, the performance and bandwidth bottleneck between the chip and memory has only gotten worse, leading to deeper hierarchies. As shown in Figure 1, Level 2 and Level 3 caches have become industry standards, with increasing sizes through time. Level 4 caches are also starting to appear. Therefore, the trends are bigger and deeper. But as the hierarchy deepens, the additive latencies and energy consumption caused by serial accesses through the hierarchy become significant. We observe that in a typical four-level cache hierarchy, lower level caches (L3 and L4) despite being accessed infrequently, can consume 80% of the total dynamic cache energy, demonstrating a compelling reason to reduce energy consumption without a major impact on latency. (We detail our methodology in Section IV). Though leakage power does represent a significant fraction of cache energy, we argue that large caches today tend to be optimized for leakage power [15]. Hence, the dynamic energy consumption is still a challenging problem to solve.

Finally, multi-threaded and multi-programmed workloads increase cache pressure and, consequently, off-chip memory accesses. While deep memory hierarchies attempt to mitigate this increase in misses that go off chip, traversing the cache hierarchy makes each off-chip miss more expensive. Our design achieves much lower latency with competitive energy consumption to splitting tag and data accesses in lower level caches.

To this end, we propose *Recalibrating Deep Hierarchy Prediction (ReDHiP)*, an efficient architectural mechanism that perfectly predicts the absence of data in the last level cache (LLC) in a deep cache hierarchy (and very accurately predicts the presence of data, as well). ReDHiP 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 will be performed. If ReDHiP determines the data does not exist in the LLC, then all lower levels of cache are skipped. Otherwise, the miss will proceed as before, beginning with the L2 cache. In order to provide accuracy with low energy, we employ a very simple, small table with infrequent, low-cost recalibration.

We show that this is by far the best tradeoff in energy efficiency and performance when compared to existing schemes: ReDHiP saves 22% overall cache energy on average, and also improve application performance by 8%, with only 0.78% of LLC capacity hardware overhead. We also show that ReDHiP is able to offset the overhead of existing architectural mechanisms when used in combination with them. These mechanisms include hardware data prefetching and parallel tag-data access in lower levels of the cache hierarchy.

II. Related Work

In this section, we survey the previous work in applying various prediction techniques to reduce energy consumption, as well as alternate approaches in designing energy-efficient lower level caches. Among them, we detail two techniques that are closely related to our approach: *Counting Bloom Filter Based Prediction* and *Phased Cache*. These approaches will be compared against with our technique during our evaluation of ReDHiP.

A large body of previous work applies prediction techniques in the cache hierarchy to predict how likely a piece of data will be accessed (again) within a short time period. Prediction results can be used to guide where to insert data into the LRU queue, when to evict data, and what data to replace [13], [22], [29], [27]. The goal of these works (where to put data) is different but complementary to ours (where to find data). The closest works to ours try to predict the presence of data in a cache, often using a hash of the address to predict data presence upon a cache access [14], [17], [30], [18]. These techniques often rely on certain data structures to assist fast prediction, such as Counting Bloom Filters.

A **Counting Bloom filter (CBF)** [7] is a variant of a Bloom filter[2] in which each table entry is a counter instead of a single bit. Upon adding an address, counters from each hashed index are all incremented; when the address is deleted, counters from each hashed index are decremented. Testing of data presence works in the same way as a Bloom filter. There are four parameters in designing counting Bloom filters in cache prediction: number of table entries, bits per counter, hash function and the number of hash functions. The number of entries has to be large enough to reduce conflicts. The number of bits per counter needs to accommodate the theoretical maximum counter value to prevent overflow. Several popular hash functions exist, including *bits-hash*, which uses the lowest N significant bits, and *xor-hash*, which xor's different parts of the address. Previous research [9] found that a one-hash-function counting Bloom filter is sufficient for cache prediction, whereas *xor-hash* achieves higher accuracy than *bits-hash* with little extra hardware, and 3-bit counters are sufficient to prevent most overflows with a 256K cache. They propose that a counter can be disabled/saturated when it exceeds its maximum capacity. The overhead of their proposed solution is as much as 6% storage overhead of the target cache, which is likely impractical for a 64MB L4 cache. In Section V, we study the performance of CBF as the table size shrinks.

To maximize performance, typical L1 and L2 caches perform tag and data accesses in parallel. Hence, the data array will always be accessed regardless of whether the access hits in the tag array. It has been proposed that for large low level caches with high associativity, since latency is less critical, tag access and data access can be serialized (**Phased Cache** [12], [11]). This approach is able to significantly reduce the energy consumption due to the large gap between tag access energy and data access energy (typically between 1:3 to 1:5). However, the performance also degrades due to serialized tag access delay. We will compare Phased Cache to our approach and demonstrate different trade-offs between performance and energy.

III. ReDHiP Design and Implementation

We introduce ReDHiP (**Re**calibrating **D**eep **Hi**erarchy **P**rediction), a mechanism that predicts data presence in the last level cache and achieves both energy efficiency and high performance with low hardware cost.

The core to our approach is a prediction table that contains presence information about data in the last level cache. The table is accessed after every L1 cache miss, because L1 hit rates tend to be high. Similar to prior work using Bloom filters, the prediction table makes conservative predictions (i.e., false positives, leading to wasted cache lookups, are permitted, but false negatives, leading to off-chip accesses

[2]A Bloom filter [4] is a probabilistic data structure that uses multiple hash functions for testing whether an element is a member of a set.

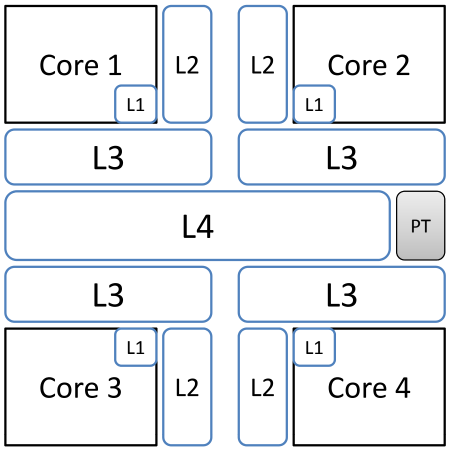

Figure 2. An example Deep Cache Hierarchy in a 4-core processor; each core has a private L1, L2 and L3 cache. All cores share the same L4 cache. A prediction tables (PT) is located aside L4.

for data that is on-chip, are not). Figure 2 depicts the memory hierarchy used for our evaluation (we only draw 4 cores in the figure due to space limitations, but use 8 cores in the evaluation). L1 through L3 are all private caches, whereas the L4 cache is shared by all cores. The Prediction Table (PT) is located aside the L4 cache to allow fast communication with it. While not every architecture looks exactly like this, it represents a typical usage of a deep cache hierarchy system. For most of the results, we assume that all levels of caches are inclusive, meaning that every level of cache contains all data from upper levels. In Sections III-C and V-B3, we will discuss how inclusion policies affect our design and evaluate the performance.

When accessing the cache hierarchy, the L1 cache is always accessed first as in traditional architectures. Upon an L1 cache miss, instead of accessing the L2 cache next, the prediction table is accessed next. If the prediction table indicates that data *is* in the LLC, then it is possible for the data to be in any level above. Hence it will conservatively access the L2 cache, as if no prediction had occurred. 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. The request is sent directly to the main memory, bypassing accesses to all other caches. After the data block is fetched from the memory and put into the LLC, the prediction table is also updated; no table update is required after evictions, as we describe below. Importantly, note that ReDHiP only changes the behavior when accessing data *not residing in any cache*, so it does not require changes to existing cache coherence protocols. All other accesses merely experience a delay between the L1 and L2 accesses.

Intuitively, achieving high prediction accuracy would require complicated, large hardware structures with delicately designed hashing and managing mechanisms. However, we find that a very simple prediction table design coupled with the simplest prediction mechanism, despite loosing accuracy over time, enables an extremely efficient recalibration algorithm, leading to high accuracy. This observation serves as the key insight to our approach, presenting an interesting tradeoff between accuracy (incurring mispredictions) and recalibration (incurring overhead).

A. 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.

Table Structure: In designing lookup structures, associativity is one critical design parameter. Adding associativity trades off accuracy for energy. Prior work including variances of Range Cache [10] and MissMap [18] use fully associative structures for tag comparisons, since performance is the first priority or the only design goal. To minimize the energy, we use a direct-mapped table in ReDHiP: the computed hash value of a given address is used directly as an index to the prediction table.

Entry Width: For structures that remove data frequently, table entries use counters to keep track of the precise number of entries with the same hash. The width of counters needs to be large enough to capture the theoretical maximum possible counter values, otherwise these counters will be simply disabled when saturated. While having large counters achieves high accuracy, the required width leads to unreasonably large tables to cover the LLC. Instead, we use a bit map. A bit is set to one when an entry is added, but it is not updated to reflect eviction. Although this approach has not been used in prior works, we find that our incremental recalibration restores sufficient accuracy.

Hash Function: Prior work such as CBF uses *xor* hash due to its relatively higher accuracy. Addresses are divided into several parts and the hash value is calculated based on the *xor* operation of these parts. We observe that an accurate mechanism requires too much overhead for too little energy savings. In contrast, an extremely simple hash function that only uses partial address bits enables an inexpensive recalibration that is not possible with *xor* hashes. In ReDHiP, the hash value is the lowest P bits of address after the block offset has been removed, referred to as *bits-hash*, shown in Figure 3. In theory, any arbitrary subset of the address bits can be used as the hash, however, using the least significant bits has a unique advantage: since cache set indexes are also calculated based upon least significant bits, as long as we use more bits in bits-hash than that in the cache, addresses that conflict in the predictor will also conflict in the cache (i.e., belong to the same cache set). This effectively constrains the theoretical maximum value of the counters in the predictor

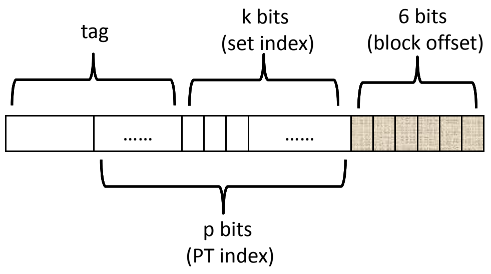

Figure 3. Data Address Format: the last 6 bits are the block offset (assuming 64-bytes block size), which is never used for indexing; the next k bits are the cache set index (2^k sets in the cache); the remaining bits are tag bits. With bits-hash, the lowest p bits after the block offset are used as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

to be equal to the cache associativity, which in turn makes the 1-bit entry width decision more practical.

B. Recalibration

Recalibration is critical 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. Without careful design, this process can be extremely expensive in terms of both latency and energy consumption. The following operation needs to be performed on each tag from the tag array in the LLC: the full tag (42 bits in a 64-bit system) needs to be read out, hashed through a hash function, indexed into the prediction table, and the corresponding counter incremented. In a 64MB cache, there are 1 million tags. Neither the table nor the LLC may be used during this operation.

With traditional Bloom filters, the location of the cache lines that affect the entries would be unknown. It is hardly possible to parallelize these operations, and each tag operation will take at least several cycles. The total delay spent on recalibration for traditional predictors could be as large as several million cycles, also costing nontrivial energy consumption. The hash function in ReDHiP eliminates these problems and provides low-cost recalibration.

As we have shown in Figure 3, the prediction table address and cache set index share the same last k bits, hence for each tag in a given cache set, we only need $p - k$ bits to determine the location in the prediction table. Our goal of a small prediction table implies that $p - k$ is small. In our study, we use a 512KB prediction table and a 64MB LLC with 16-way associativity as our base design. In this case, p is 22, k is 16 and $p - k$ is 6. We define a continuous range of 64 (2^6) bits as one line in the prediction table. Because of the hash function chosen, the cache lines that affect a predictor line are all located in the same set in the cache. 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 single

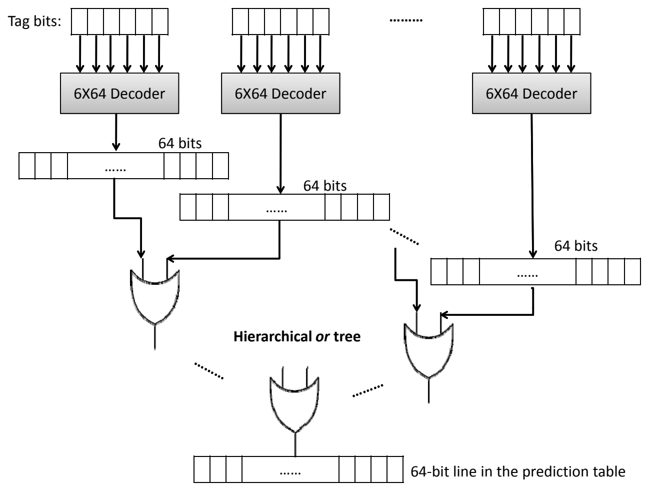

Figure 4. Hardware required to recalibrate all 16 entries in one cache set. Because the set index (16 bits) is a subset of the prediction table hash (22 bits), only 6 bits of the tag are needed. A decoder is used to expand it to a 64-bit vector with its entry set. Those 64 bit vectors are combined with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

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. Figure 4 illustrates this process. Each of the 16 entries in the set produces its own 64-bit decoder output. Instead of summing the values in the corresponding locations in the 64-bit vectors that would have been needed in counter-based predictions, we only need to *or* them. Presence is all the 1-bit counter expresses, not a count of how many lines map to it. This logic is simple to implement and can finish within one cycle; any slight complexity added to the predictor will prohibits the possibility of this recalibration process.

Finally, the fact that a single set is all that is necessary to update a single line in the prediction table leads to opportunities for parallel updates. We can take advantage of the banking structure of most modern caches, and recalibrate cache sets from multiple banks in parallel as shown in Figure 5.

In summary, the simple nature of hash tables with 1-bit entries and simple hash function enables an efficient and highly parallel recalibration mechanism. Different design effort with different parallel degree can be applied to trade off between recalibration latency and hardware complexity. We will show in our evaluation that a medium-effort design with recalibration every 1000,000 L1 misses is sufficient to achieve high accuracy with trivial overhead.

C. Exclusive and Hybrid Cache

In practice, caches can be designed with various inclusion policies. In an extreme case, the cache hierarchy can be completely exclusive – every level of cache contains distinct data. Under such a configuration, data that does not exist in the LLC might still reside in upper level caches. We propose that, for a completely exclusive cache architecture,

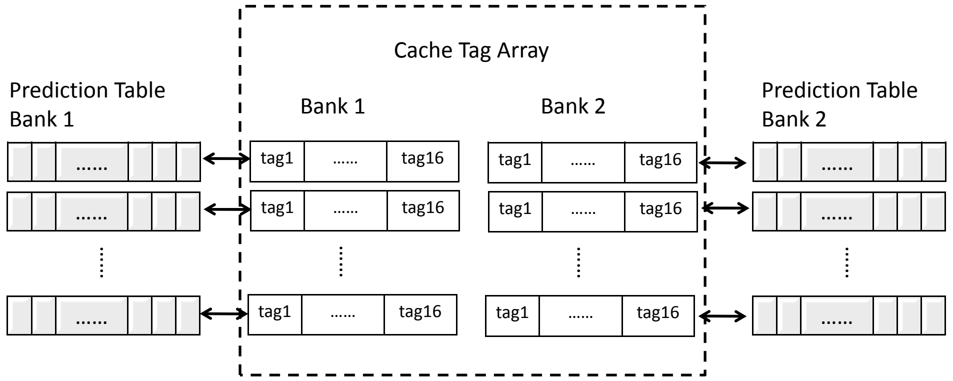

Figure 5. The tag array of the last level cache along with the prediction table. The prediction table is multi-banked in the same way as the tag array. Each set in the tag array (containing 16 tags) corresponding to a 64-bit line in the prediction table.

the 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 cache starting from L2 will be associated with a prediction table at the same storage overhead ratio (i.e. 0.78% as we use in the evaluation). Upon an L1 miss, the prediction tables from every level down the hierarchy is requested simultaneously. These prediction results are collected and all levels that predict true residency will be accessed in sequence. Levels that predict false will be skipped. While this approach seems complex, the extra energy overhead is small due to the constant overhead ratio for each predictor/cache pair. In addition, it has the benefit that the request is sent to the lowest level where it may exist rather than always restarting at the L2 cache.

In multi-core processors, implementing an exclusive LLC shared by many cores has many challenges. As a common tradeoff, the shared LLC can be implemented either inclusive or pseudo-exclusive in a way that provides certain inclusive guarantees. In this case, upper private level caches can be exclusive with each other, but they must all be inclusive with the shared LLC. Under such a configuration, no changes are required for ReDHiP, since ReDHiP only relies on the inclusive property of the LLC. We refer to this configuration as a hybrid architecture, and examine how various inclusion policies interacts with ReDHiP.

IV. Methodology

For our evaluation, we select 8 benchmarks from SPEC 2006 [3] and two large-scale applications. The SPEC 2006 benchmarks are: astar, bwaves, cactusADM, GemsFDTD, lbm, mcf, milc, soplex. This SPEC subset focuses on exercising the deep memory hierarchy, omitting benchmarks that have very high L1 cache hit rates or low memory traffic. In the case when the L1 cache miss rate is very low or the LLC is rarely used, our prediction mechanism would be disabled to not waste energy or add latency. We also include two applications

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

Table I
Architecture parameters in our simulation. Since Phased Cache would access tag array and data array sequentially in large caches, we provide separate numbers for tag array and data array for L3 and L4. While all other schemes access them in parallel.

with large-scale datasets: the Graph500 [2] benchmark implemented using the Combinatorial BLAS Library [5] and a probabilistic matrix factorization algorithm implemented using GraphLab Library [19]. These two benchmarks represent state-of-art machine learning algorithms. The selected SPEC benchmarks typically consume tens to hundreds of megabytes memory, while the large-scale applications use several gigabytes of memory.

We implement a high-level simulation environment using Pin [20] running on x86-64 Linux hosts. Each benchmark is instrumented with pintool to collect memory references. 1.5 billion instructions are traced for each benchmark (warm-up phases are all skipped). Each trace contains an average of 500 million memory accesses. For each SPEC benchmark we collect one trace file. For CombBLAS and GraphLab, which both support parallel computation, we specify each application to execute in 8 processes and trace each process simultaneously (which will give 8 traces for both benchmarks). To demonstrate the impact of cache interference among different types of applications, we also include a *mix* simulation in which each of the 8 cores is running a different SPEC application, which also gives 8 traces.

The traces collected above are then feed into a cycle-accurate cache energy simulator that simulates a 8-core processor with a 4-level deep cache hierarchy. When simulating each benchmark, we multi-program them by duplicating the trace into 8 copies running on each core. For each of CombBLAS, GraphLab and the mix of SPEC benchmarks, we simply run 8 traces on 8 cores.

Cacti 6.5 [1] is used to estimate the latency and dynamic energy of all levels of cache as well as our prediction table (including both access overhead and recalibration overhead). Note that because our prediction table is direct-mapped with only 64-bit entries, its dynamic access energy is much smaller than the L2 cache despite being the same size. Leakage power is estimated based on the recent published data for cache designs [25] (more accurate on large caches). The latency of wiring from the processor to the prediction table is estimated using data from [23].

We focus on the cache behavior, hence the memory is not modeled in our simulator but treated as a data store that always hits on requests (with no delay and no energy consumption). We also do not include other parts of the processor when modeling energy consumption. However, to demonstrate the performance impact of ReDHiP comprehensively, we include the delay of computation (i.e. non-memory instructions) in the performance evaluation. For simplicity, we estimate the timing of each instruction using the average CPI of each application. While this is likely to have inaccuracies, the relative order of memory references is precise enough to simulate realistic cache behaviors. This estimated timing tends to only affect the overall execution time, while it should have no substantial effect on our comparison results.

Table I shows the architecture configuration. For most experiments, we assume a completely inclusive cache architecture, which is more commonly used and practical compared to fully exclusive caches. In the final results section, we evaluate schemes with different inclusion policies, as described in Section III. The hybrid scheme uses an inclusive, shared LLC while all private caches are exclusive. The exclusive scheme has an exclusive policy for the LLC, as well.

For the ReDHiP scheme, we model a 512KB prediction table (0.78% hardware overhead of LLC) with 1-bit entries and bits-hash using the lowest significant 22 bits of address other than the block offset bits. Recalibration is performed incrementally with an update for every table entry every 1 million L1 misses. Given an average L1 hit rate of 91.5% across all our benchmarks, the recalibration will be performed at every 11.76 million L1 accesses across all cores. In our 8-core configuration where each core runs 500 million memory reference instructions, 340 recalibrations will be executed in total. We assume a medium-effort parallelization for recalibration: the prediction table is split into 4 banks so that 4 sets can be recalibrated at the same time. The total overhead of completely recalibrating the table is 16K cycles (1 million tags in total, 16 tags from the same set can be processed within a cycle, and 4 sets calibrated in parallel). The overhead (energy and delay) of recalibration is included in the results. All caches, when they are accessed, have parallel tag and data array accesses.

We compare ReDHiP to four other configurations. The *Base* case, on which all results are normalized, has no prediction or optimizations. Tags and data arrays are accessed in parallel at all levels of cache to reduce latency. The counting Bloom filter-based prediction scheme from [9] (*CBF*) is given the same area budget (512KB table) as ReDHiP. Tags and data arrays are accessed in parallel when the table indicates that the data could be in the cache. The *Phased Cache* scheme is applied to the L3 and L4 caches, in which tag arrays and data arrays are accessed sequentially to save energy. An *Oracle* predictor always accurately predicts the LLC data existence without any overhead. This is not the same as constant recalibration, because the single bit format is inherently inaccurate since multiple entries map to the same entry in the table.

V. RESULTS

In this section, we first show the performance gain and energy savings of our ReDHiP implementation, then we perform detailed sensitivity analysis to study the impact of different design decisions and parameters. Finally, we evaluate how ReDHiP interacts with and improves prefetching.

A. Performance and Energy

Figure 6 shows the *performance* of different techniques. We can see that the Phased Cache suffers a performance hit because it accesses the data array after the tag check, leading to a performance degradation of 3% on average. Because the counting Bloom filter-based approach was limited to the table size of ReDHiP, it achieves less than 4% speedup. In contrast, ReDHiP improves performance by an average of 8%, with a theoretical maximum bound of 13% as shown in the case of an Oracle predictor, despite a 3% penalty for the prediction overhead (ReDHiP without overhead can achieve 10% performance gain).

Despite the performance gain achieved by ReDHiP, it is also able to save significant dynamic cache access energy. Figure 7 shows the *cache dynamic energy consumption* of

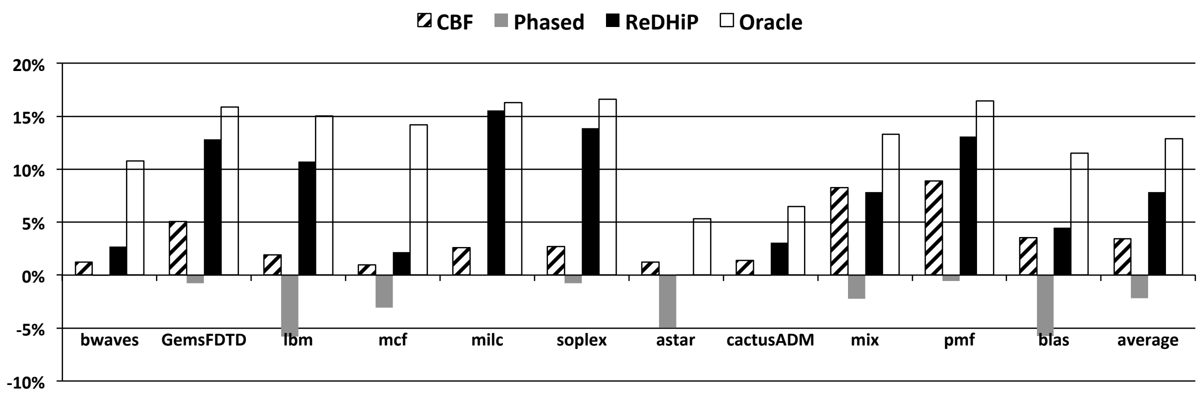

Figure 6. Performance speedup of: Oracle (Perfect predictor), CBF (Counting Bloom Filter based prediction), Phased Cache and ReDHiP. All mechanisms are compared against a base case in which no prediction/optimization is used. Positive numbers mean speedup while negative numbers mean performance degradation. ReDHiP and CBF are both using a 512KB prediction table. Prediction and recalibration overhead is included in ReDHiP (about 3%).

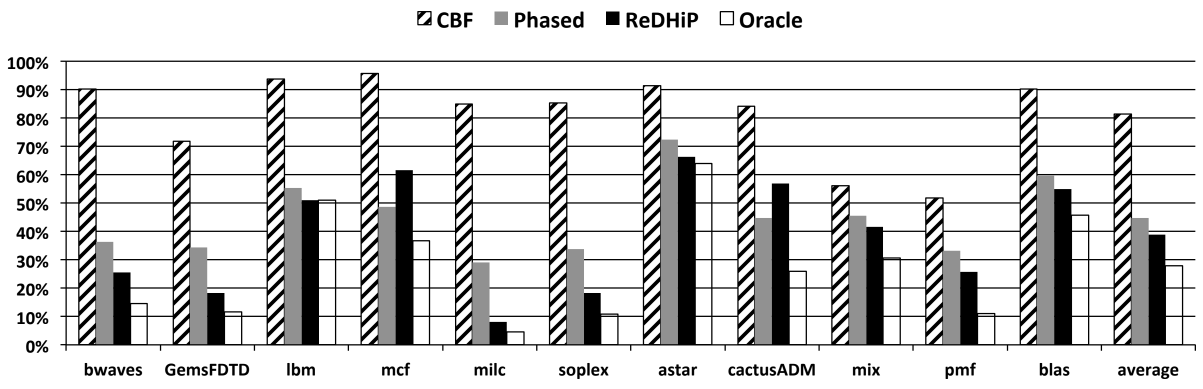

Figure 7. Dynamic energy consumption of: Oracle, CBF, Phased Cache and ReDHiP, normalized to the base case.

different techniques, normalized to the base case. CBF only saves an average of 18% dynamic energy because it is not as accurate as ReDHiP (leading to many more wasted cache accesses), while Phased Cache saves 55% (with a performance loss of 3% as we have shown). On the other hand, ReDHiP behaves better than the other two techniques again, with an average of 61% dynamic energy saving. The prediction and recalibration energy overhead only accounts for less than 1% of total dynamic energy. The gap between ReDHiP and the Oracle prediction (71% saving, a gap of 10%) lies not in the prediction or recalibration overhead, but in the fundamental inaccuracy of the predictor that cannot be resolved by recalibrations (since multiple items map to the same location in the table).

Prediction mechanisms reduce not only dynamic energy, but also static energy due to the reduction in execution time through improved performance. A speedup of 8% directly translates to 8% static energy saving. *Overall, we find that ReDHiP is able to save an average of 22% total energy consumption.*

In order to better get a better sense for the two contributions of performance and energy savings, we define a *performance-energy metric* to be the product of performance gain and total energy savings (including both static and dynamic). A performance speedup of X with a total energy saving of Y will give $(X) \times (Y)$ in the metric. A scheme that benefits in both areas will have a higher product. Figure 8 shows the comparison of the *performance-energy metric* between different techniques (Oracle being a theoretical limit, not an actual scheme). ReDHiP achieves by far the best trade-off between performance and energy, while requiring only 0.78% of LLC storage overhead.

The performance gains and energy savings of ReDHiP come mostly from significant reduction of lower level cache misses guided by the prediction. Figure 9 and Figure 10 deplot the hit rates of each level of cache for all benchmarks in the base case (where no prediction mechanism is used) and in ReDHiP. The hit rate of L1 is not affected since prediction only happens at L1 misses. ReDHiP improves the hit rate of L2, L3 and L4 by an average of 14%, 12% and 18% respectively.

B. Sensitivity Analysis

Different design decisions and parameters can significantly impact the efficiency of our mechanism. In this

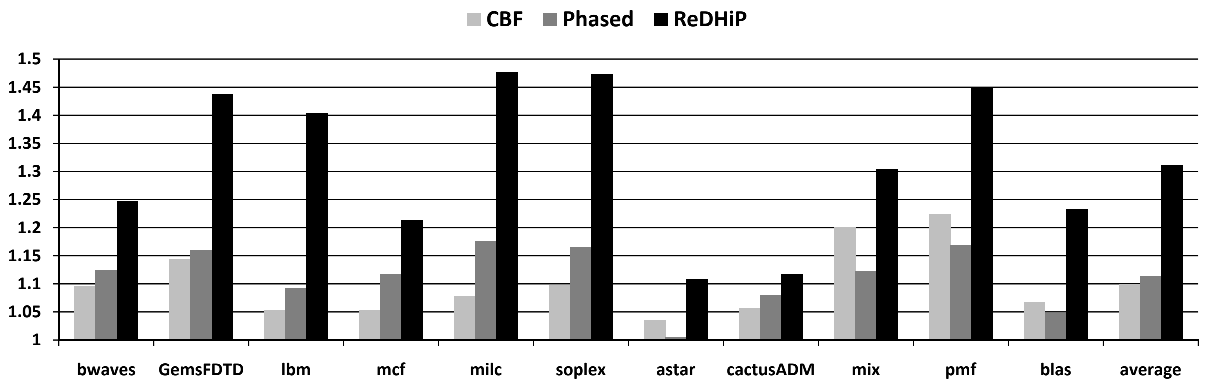

Figure 8. Performance-energy metric of CBF, Phased Cache and ReDHiP. The metric is calculated as the product of performance gain and energy savings (higher metric value indicates better scheme).

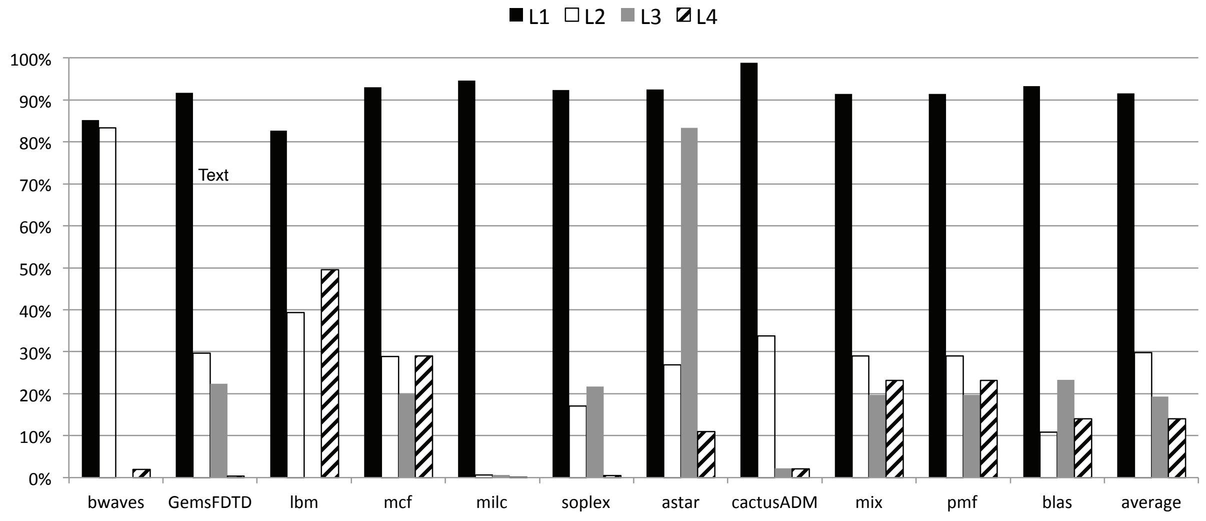

Figure 9. Hit rate of each level cache for all benchmarks in base case where no prediction mechanism is used.

section, we justify the choice of several major parameters including table size and recalibration frequency. We will also study the impact of various cache configurations.

The predominant effect to latency and energy is the accuracy of the table. The overhead is a very secondary effect. To highlight the impact of ReDHiP's prediction accuracy, we next focus on dynamic energy and, for these results only, ignore the prediction overhead.

1) Prediction Table Size: In our base results, we model a prediction table with a reasonably small capacity, 512KB. Figure 11 shows how dynamic energy consumption varies when the size of the prediction table changes from 64KB to 2MB. The recalibration frequency is constant (1M) for all of them. As can be seen from the figure, higher accuracy can be achieved by using a larger prediction table. The prediction accuracy gain starts to become marginal when the table size goes beyond 512KB, while the table can become almost useless when the size goes below 64KB. Both 256KB and 512KB could be reasonable design choices. We choose to use a 512KB table because, even at that size, it is only 0.78% of the LLC, so we believe this is worth the gains in accuracy.

2) Recalibration Frequency: Each entry in the prediction table is recalibrated periodically to maintain enough prediction accuracy. Short recalibration periods provide better accuracy but lead to higher recalibration overheads, whereas infrequent recalibration does not yield enough prediction accuracy to achieve reasonable benefits. Choosing a reasonable recalibration period in order to achieve the best balance between prediction accuracy and overhead is critical. Figure 12 shows how dynamic energy consumption due to the accuracy of the table changes when the recalibration frequency varies from every L1 miss (i.e. perfect recalibration), to every 10,000 L1 misses up to Infinite (i.e. never recalibrate). As the figure shows, there is a precipitous drop from recalibrating every 1M L1 misses down to recalibrating every 100M L1 misses. It is critical that recalibration occur at least every 1M L1 misses; more frequent recalibrations provide little additional savings. 1M

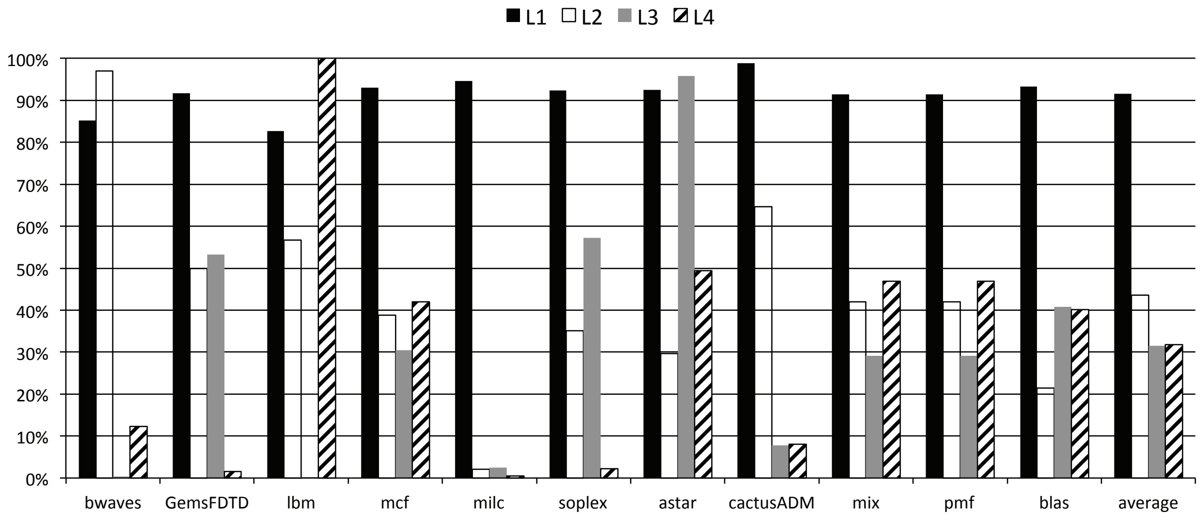

Figure 10. Hit rate of each level cache for all benchmarks when ReDHiP is applied.

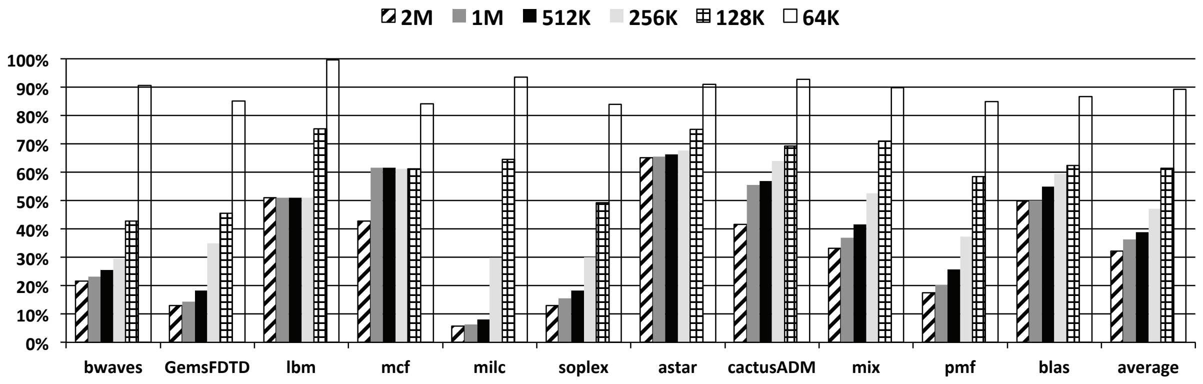

Figure 11. Dynamic energy consumption of ReDHiP under different prediction table sizes, normalized to the base case where no prediction mechanism is used.

is the clear choice based on accuracy alone.

3) Exclusive and Hybrid Cache

: We also consider two other policies to see how various inclusion policies might affect our mechanism – fully exclusive and hybrid. As described in Section III-C, the cache architecture can be fully exclusive, with all levels of cache containing distinct data. In this case, low level caches typically serve as a victim cache for upper levels. Due to the challenge in implementing the cache coherence efficiently, caches tend to be designed partial-inclusive. We pick one typical configuration in which private caches (L1, L2, L3) are exclusive while the shared LLC is inclusive. This hybrid configuration makes a reasonable trade-off between capacity and coherence overhead.

Figure 13 shows the *dynamic energy savings* of ReDHiP under different cache inclusion policies. In the hybrid scheme, where the LLC is still inclusive with the rest of the hierarchy, there is no change to the ReDHiP implementation and negligible change to the results. This makes sense, because there is little change to the behavior at the LLC, only to where items are found within the hierarchy. The exclusive case, however, leads to vastly more data residing on chip. Placement on the chip changes much more often, so each cache maintains its own table, causing more overhead but offering more accurate knowledge of where the item might reside. The extra overhead is only partially offset by the better knowledge, resulting 15% less every savings than with the hybrid or inclusive policies. Even so, this represents more than a 40% benefit over the baseline architecture.

C. Prediction and Hardware Prefetching

One might think that the performance gains we achieve, because they only apply to data residing off-chip, can be achieved similarly through dynamic data prefetching [24], [16], [28]. If the data is prefetched, then an off-chip access is not necessary, rending our scheme less useful. We find this not to be the case.

Prefetching only reduces the delay of data access by making them appear earlier, but it comes at an energy cost. In fact, prefetching based on imperfect predictions will

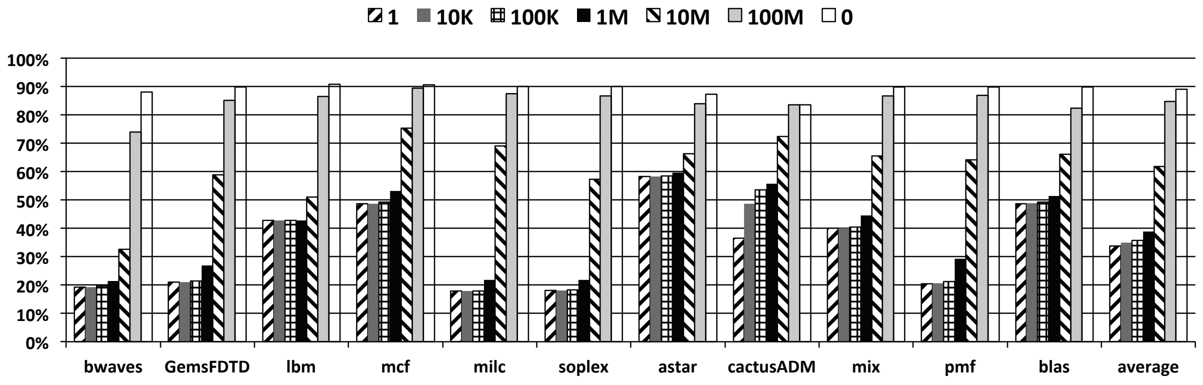

Figure 12. Dynamic energy consumption of ReDHiP under different recalibrations frequencies (i.e. number of L1 misses between two recalibration), normalized to the base case where no prediction mechanism is used. 1 means recalibrating after every L1 miss (perfect recalibration), 0 means no recalibration.

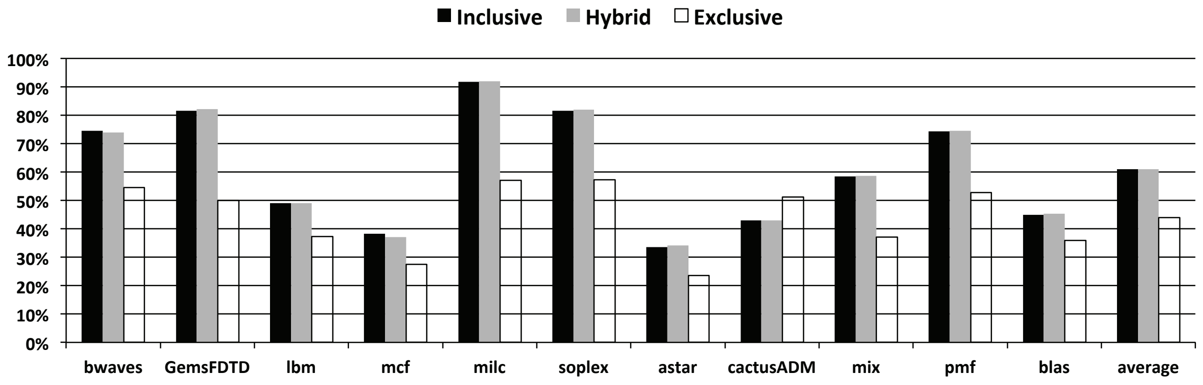

Figure 13. Dynamic energy **savings** of ReDHiP under different cache inclusion policies: *Fully Inclusive*, a *Hybrid* architecture in which L2 and L3 are exclusive while L4 is inclusive, and *Fully Exclusive*. All results are normalized to the base case where no prediction mechanism is used. Comparisons are made between the same cache inclusion policies.

request unneeded data, resulting in extra cache accesses and cache pollution. The dynamic energy overhead is proportional to the number of unneeded prefetches. On the other hand, ReDHiP achieves significant energy savings, but does little for latency (when compared to the cost of an off-chip access). Because of these complementary benefits, the two techniques form a perfect match. We implemented a simple hardware stride prefetcher [8]. Although the stride prefetcher is not as efficient as state-of-art prefetching mechanisms [21], [26], we make the prefetch table size large enough so that its accuracy is comparable with the best prefetching techniques. Figure 14 and Figure 15 show the performance and energy comparison when using prefetching alone, ReDHiP alone, or both together, normalized to a base case where no mechanisms are used. From the figures we can see that these two techniques can be combined with each other seamlessly and obtain the advantages from both. For performance, they are complete complementary. Prefetching is effective when there are predictable access patterns, but ReDHiP is useful regardless of the pattern. When used in combination, prefetching accelerates prefetchable accesses, and ReDHiP accelerates non-prefetchable accesses. Energy is a slightly different story. Prefetching definitely comes at great cost, which ReDHiP is able to offset with on-chip savings. In the end, the performance benefits are additive, resulting in great savings when they are combined. The resulting energy, though, is somewhere in between the savings from ReDHiP and the cost of perfetching.

VI. Conclusions

Technology trends indicate that deep cache hierarchies with 4 or more levels will become pervasive. Accessing data that does not reside in the cache hierarchy requires searching through all levels of cache, leading to high energy and performance overhead. In this paper, we propose an efficient deep hierarchy prediction mechanism that accurately predicts LLC misses with small hardware overhead. We achieve this through an extremely simple prediction table design with a low-overhead recalibration mechanism. Simulation on a range of benchmarks shows that our mechanism is able to save 22% overall energy consumption, as well as improve performance by 8%, with a small hardware overhead equivalent to only 0.78% of LLC. This combination of performance gains and energy savings

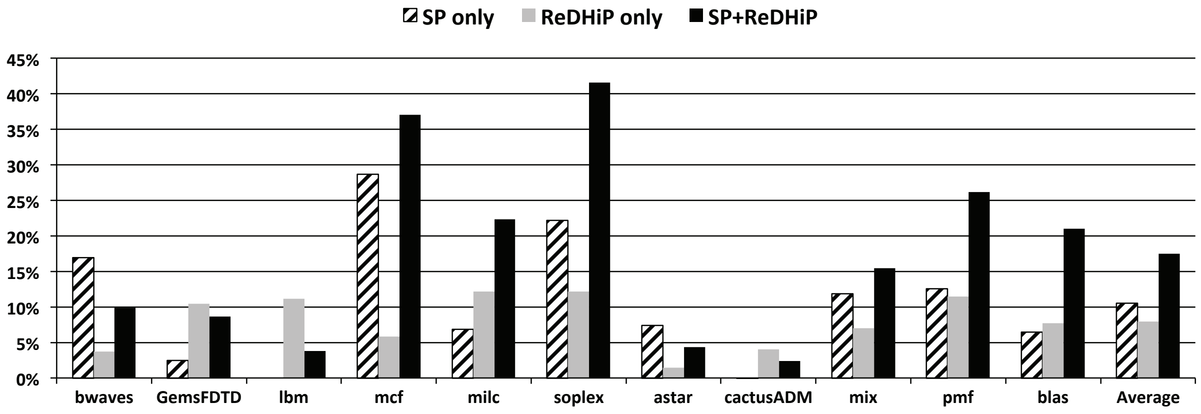

Figure 14. Performance speedup of: SP only (Stride Prefetch only), ReDHiP only (prediction only), SP+ReDHiP (both prediction and prefetching), normalized to a base case where no mechanisms are used.

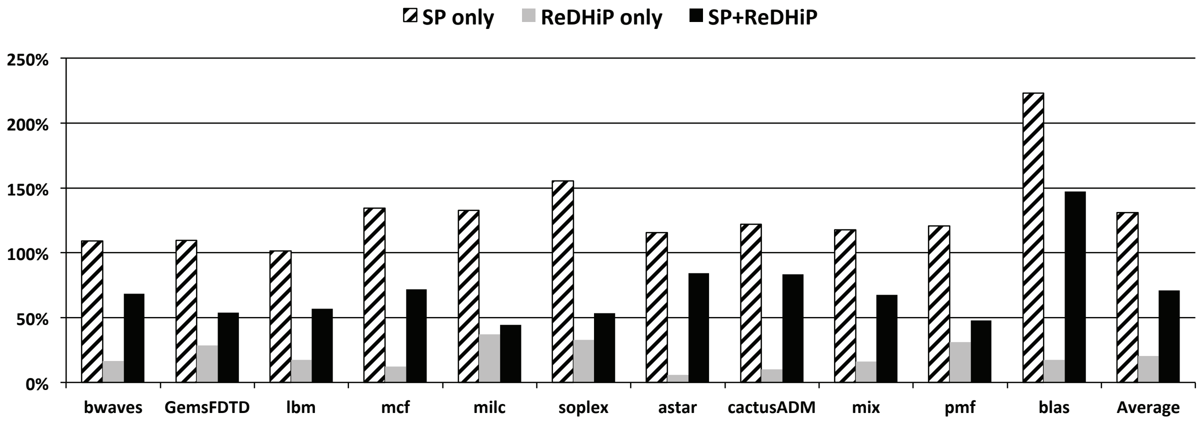

Figure 15. Dynamic energy consumption of: SP only (Stride Prefetch only), ReDHiP only (prediction only), SP+ReDHiP (both prediction and prefetching).

are more attractive than previous approaches. Finally, ReDHiP can be combined with prefetching to achieve higher performance than either alone, with a compromise between the two in energy savings.

Acknowledgement

This work was supported by NSF grants CCF-1017578 and CCF-1218794, and Google.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring agencies.

References

[1] "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," http://www.hpl.hp.com/research/cacti/.

[2] "The graph 500 list," http://www.graph500.org/.

[3] "SPEC 2006," http://www.spec.org/cpu2006/.

[4] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[5] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[8] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 102–110.

[9] M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee, "Efficient system-on-chip energy management with a segmented bloom filter," in *Proceedings of the 19th International Conference on Architecture of Computing Systems*, ser. ARCS'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 283–297.

[10] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin, "A case for unlimited watchpoints," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 159–172.

[11] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "Sh3: High code density, low power," *IEEE Micro*, vol. 15, no. 6, pp. 11–19, Dec. 1995.

[12] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ser. ISLPED '99. New York, NY, USA: ACM, 1999, pp. 273–275.

[13] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 60–71.

[14] T. Juan, T. Lang, and J. J. Navarro, "The difference-bit cache," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA '96. New York, NY, USA: ACM, 1996, pp. 114–120.

[15] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design*, 2011.

[16] W.-f. Lin, "Reducing dram latencies with an integrated memory hierarchy design," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 301–.

[17] L. Liu, "Cache designs with partial address matching," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 128–136.

[18] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 454–464.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[21] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 96–.

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391.

[23] P. Ruch, T. Brunschwiler, W. Escher, S. Paredes, and B. Michel, "Toward five-dimensional scaling: How density improves efficiency in future computers," *IBM J. Res. Dev.*, vol. 55, no. 5, pp. 593–605, Sep. 2011.

[24] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 63–74.

[25] Y. Wang, U. Bhattacharya, F. Hamzaoglu, P. Kolar, Y. Ng, L. Wei, Y. Zhang, K. Zhang, and M. Bohr, "A 4.0 ghz 291mb voltage-scalable sram design in 32nm high-κ metal-gate cmos with integrated power management," in *Solid-State Circuits Conference-Digest of Technical Papers*, 2009.

[26] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 222–233.

[27] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 430–441.

[28] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Pacman: Prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 442–453.

[29] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183.

[30] C. Zhang, F. Vahid, J. Yang, and W. Najjar, "A way-halting cache for low-energy high-performance systems," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 34–54, Mar. 2005.