2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00054

Reducing Load Latency with Cache Level Prediction

Majid Jalili, and Mattan Erez
The University of Texas at Austin
{majid,mattan.erez}@utexas.edu

***Abstract*—High load latency that results from deep cache hierarchies and relatively slow main memory is an important limiter of single-thread performance. Data prefetch helps reduce this latency by fetching data up the hierarchy before it is requested by load instructions. However, data prefetching has shown to be imperfect in many situations. We propose cache-level prediction to complement prefetchers. Our method predicts which memory hierarchy level a load will access allowing the memory loads to start earlier, and thereby saves many cycles. The predictor provides high prediction accuracy at the cost of just one cycle added latency to L1 misses. Level prediction reduces the memory access latency by 20% on average, and provides speedup of 10.3% over a conventional baseline, and 6.1% over a boosted baseline on generic, graph, and HPC applications.**

I. Introduction

Low memory-load latency is critical for high-performance computing applications. Achieving low load latency is challenging because latency has been trending up as cache hierarchies grow in capacity and complexity. Recent Intel processors, for example, have estimated second- and third-level cache (L2 and L3) latencies of 12 and 40 cycles, respectively [17]. The levels of the hierarchy are typically looked up in sequence, starting from the first-level cache (L1) and proceeding through second- and third-level caches. If the data is not found in any cache, it is fetched from memory. Deep cache hierarchies generally improve performance, but can result in higher load latencies when caches do not successfully filter requests, only adding lookup delays [6], [25], [39].

Prefetchers somewhat mitigate the latency impact of level-by-level lookup by moving data between levels prior to the execution of load instructions [13], [14], [15], [18], [19], [20], [24], [32], [40], [41]. Despite prefetch effectiveness, many loads are still exposed to sequential-lookup delays. Previous work has demonstrated miss coverage of just 24% and 40% for SPEC CPU 2006 [34] and CloudSuite [8], [9], respectively. In fact, our analysis of a large set of benchmarks running on an Intel Skylake processor demonstrates that many applications are likely to benefit from non-sequential cache access, even when cache hit rates are high.

We propose and evaluate a novel low-cost approach to dynamically skip unnecessary cache-level lookups, reducing the average load latency by $10-30\%$. Our *memory hierarchy level predictor* (abbreviated as level predictor or LP) enables directly looking up the cache level where a block resides *with minimal memory hierarchy changes*.

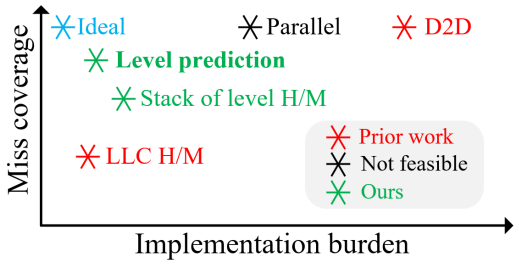

Figure 1. Comparing Level prediction to other solutions.

On an L1 miss, a (per core) LP predicts which memory levels to target, bypassing some levels and occasionally indicating partial parallel lookups across memory levels. The bypass and parallel access hardware mechanisms reuse structures that already exist in current sequential lookup implementations, requiring only small modifications to control logic. Any mispredictions that incorrectly bypass a level are also handled with cache controller modifications and reuse existing structures: unnecessary parallel accesses are terminated by modifying existing address matching-logic on the return path, and with a directory, incorrect bypasses are reissued quickly.

Though the level prediction concept and our proposed recovery mechanisms are novel, naively, the level predictor can be implemented as a per-level cache miss predictor [23], [26], [28], [31], [38]. Because miss predictors were introduced to either improve instruction scheduling or for addressing the long-latency of DRAM caches, we find that simply using, or even naively extending prior mechanisms works poorly for level prediction in modern core configuration. Prior cache miss predictors require infeasibly-large resources to provide high enough accuracy, and they squander opportunities for cross-level coordination, especially prefetchers.

Other prior work that focuses on reducing hierarchy latency includes parallel lookup across all cache levels and the Direct-to-Data (D2D) hierarchy [29]. Parallel lookup is impractical because it increases energy consumption and requires over-provisioning tag array ports. D2D eschews traditional caches by including placement information as part of address translation. While appealing conceptually, D2D is challenging to adopt in practice because the entire hierarchy, tag store, TLBs, and coherence protocol are completely

978-1-6654-2027-3/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00054

redesigned [29]. Figure 1 qualitatively compares LP with prior approaches that can reduce cache-hierarchy lookup times, showing that LP is favorable because our design is low cost, accurate, and offers a low-friction adoption path with few new components that are off the critical path, requires only small modification to controllers, and maintain existing coherence protocols.

We are the first to propose a holistic, simple, and low-cost solution for all misses. We extend the concept of hit/miss prediction to level prediction and show that this is not trivial. Stacking hit/miss predictors is both more costly and less accurate than our predictor. Furthermore, level prediction requires a new recovery mechanism that we architect by reusing existing components (e.g., consulting the directory for recovery). Thus, our approach is both performant and practical. To summarize our main contributions:

- We demonstrate that many graph analytics and scientific applications benchmarks can benefit from non-sequential lookup; including applications with a high cache hit ratio.
- We architect and evaluate an effective, yet low-cost level predictor that operates in each core on the L1 cache miss path and substantially outperforms cache miss predictors when incorporating prefetchers.
- We incorporate level prediction keeping to minimal microarchitectural changes; level mispredictions are handled by utilizing the cache coherence directory and existing address-matching logic.
- We rigorously evaluate level prediction and compare it to an idealized baseline and to the complex state-of-the-art D2D scheme [29]. Overall, level prediction improves performance by 6.1% on average over a baseline with aggressive prefetchers and modern configuration called *BaselineStrong*, and 10.3% over a baseline with configurations similar to prior work called *BaselinePrior*.
- Level prediction reduces cache hierarchy energy and average load latency by 18% and 20%, respectively, as it eliminates many unnecessary miss lookups.

II. Motivation

Many applications are memory-latency bound. To study the effectiveness of the level-by-level lookup strategy, we compare the number of misses at different levels. If the application shows good locality or has access patterns that are detected by prefetchers, the number of demand load misses decreases significantly from one level to the next. Our analysis below demonstrates that while sequential level lookup indeed works well for many applications that exhibit this type of behavior, other applications suffer from unnecessary lookups as either L2 does not successfully filter requests, or L3 provides no substantial additional benefit over the L1 and L2 caches.

The high-level insights from our analysis are summarized in Figure 2. The figure plots each evaluated application in terms of its L2 and L3 effectiveness—the filtering capability of each level of cache. The x-axis (log scale) is the ratio of L1 to L2 misses and points further on the right indicate applications for which L2 more effectively filters L1 misses from reaching L3, thus indicating that looking up L2 before L3 is the right strategy. Similarly, the y-axis (log scale) represents the effectiveness of L3 and higher points are for applications where misses from L2 are mostly hits in L3. Data on cache effectiveness is collected on a 3.2GHz Intel Core i7-8700 CPU for SPEC CPU 2017 and NAS Parallel Benchmarks applications, the gapbs graph analytics benchmark suite, and for the *hpcg*, *gups*, *stream*, *spmv*, and *bmt* kernels (see Section IV for more details on methodology).

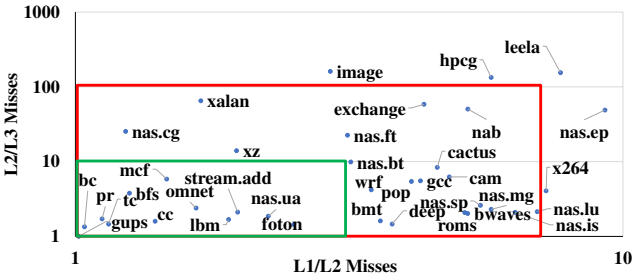

Figure 2. The x- and y-axes on this log-log plot represent the effectiveness of L2 and L3 at filtering misses, respectively. Each application is plotted according to the ratio of misses filtered by each level compared to the level above it. Applications further to the top and right work best with sequential access, while those toward the bottom left stand to benefit most from non-sequential level-predicted lookups.

Using this figure, we roughly classify applications into three categories: (1) applications outside the red box are a good fit for sequential level lookup and are unlikely to benefit from level prediction, (2) applications inside the green box, for which non-sequential lookup is likely to offer significant latency reductions, and (3) applications between the boxes where we expect that L2, L3, or levels can be occasionally bypassed for modest performance gains. We observe that not only graph analytics applications exhibit poor cache effectiveness, but that many other applications are likely to benefit from level prediction. We provide more details below.

Sequential Lookup Effectiveness Figure 3 provides more details on the number of misses at each cache level across the execution duration of several applications. We expect sequential level lookup to be the best design choice when caches reduce the number of misses significantly. Figure 3 (a) exemplifies such behavior and shows that for *hpcg* (which falls outside the red box of Figure 2), the number of misses significantly decreases after L2 (3× reduction) and also after L3 (a further 2× decrease). This behavior is consistent throughout execution, meaning serial lookup performs well over the course of execution.

Many other applications, however, suffer from serial lookup. Figure 3 (b-f) show applications where either L2, L3, or both are not effective. If the miss rate at any level is very high, the access latency increases unnecessarily by

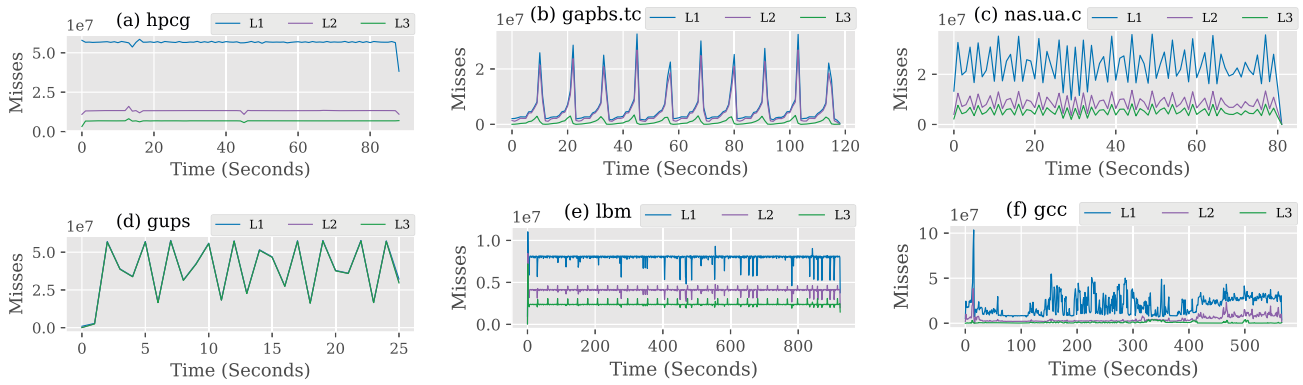

Figure 3. Miss trace of several applications across their execution. A gap between the miss rate (number of misses per time window) of different cache levels indicates effective miss filtering, while lines that are close to one another suggest a cache level lookup can be bypassed.

looking up that level. This is especially a problem when an intermediate level, has a very high miss rate. This behavior is common in graph applications, which frequently exhibit poor hit rates at L2 and moderate hit rates at L3 [3]. For example, for gapbs Triangle Count (*tc* in Figure 3 (b)), we observe a similar rate of L2 and L1 misses, indicating that L2 is ineffective. L3 only moderately reduces the number of misses. Therefore, almost all L2 accesses and the majority of L3 lookups are redundant and only increase memory access latency.

A case where L3 cache is ineffective is shown in Figure 3 (c) where the number of misses at L3 is roughly the same as at L2, despite the fact that L3 is 48× larger. Considering the fact that the L3 is large and has a high access latency, level-by-level lookup only squanders CPU cycles by looking up L3 for every single access. This problem can be exacerbated when the application has a random access pattern as in *gups* (Figure 3 (d)). Generally, random behavior impairs both prefetchers and caches, and thus almost all references to caches waste cycles that could otherwise be spent directly looking up main memory.

While it intuitively seems that sequential accesses should be the best choice for applications with simple access patterns, that is not necessarily the case. For example, Figure 3 (e) shows that despite both L2 and L3 reducing the misses for an easy-to-prefetch application like *619.lbm*, the number of misses at each level is still high. Meaning a substantial fraction of the memory requests still needs to traverse the memory hierarchy level-by-level wasting many cycles.

Application behavior may change during execution, and thus using a static strategy to lookup the caches is not optimal. Figure 3 (f) shows the miss rates for *602.gcc* from SPEC CPU 2017. L2 is not very effective at the early stages of execution (0–25 seconds), is beneficial in filtering out requests from 20–400 seconds, and decreases to lower effectiveness for the rest of execution. The hardware-level predictor can exploit this phase-dependent behavior and skip looking up levels of the hierarchy (L2 in this case) when they do not provide benefit.

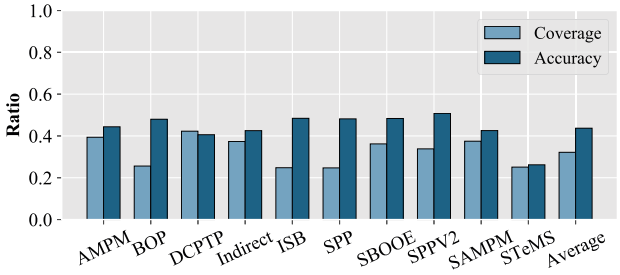

Figure 4. Coverage and accuracy of LLC prefetchers. Coverage is the fraction of misses eliminated by prefetching, and accuracy is the fraction of useful prefetches. In the best cases (e.g., DCPT [15]), 40% of misses are eliminated but with high overhead of 40% inaccuracy. High inaccuracy of prefetchers makes level prediction very challenging.

Prefetchers. Despite progress in designing prefetchers, many misses are left uncovered. Figure 4 shows the simulated coverage and accuracy of numerous state-of-the-art academic prefetchers [13], [14], [15], [18], [19], [20], [24], [32], [40], [41]. As a matter of fact, in the best-case scenarios, prefetchers can eliminate up to 50% of LLC misses, meaning many accesses still access slow main memory. The average accuracy is also low (50%-60%), meaning many unnecessary blocks are fetched and evicted with a possible negative impact on performance. We offer a new mechanism that complements advanced prefetchers and replacement policies, rather than replacing them. We attempt to handle those misses that even state-of-the-art prefetchers leave for main memory.

III. Level Prediction μArchitecture

A. Predictor Design Considerations

Levels to Predict. Without loss of generality, assume that there are four memory hierarchy levels, and any given block can reside in L1, L2, L3 and main memory. Out of these, we exclude L1 as a prediction target for two reasons. First,

many processors use virtually-indexed physically-tagged (VIPT) L1 caches (L1 and TLB are accessed concurrently), meaning the physical address needed for skipping L1 is not available during the L1 lookup itself. Second, the L1 is tightly integrated with the core pipeline, and we choose not to disrupt its timing or design. We do include L2 as a level prediction target because our evaluation results indicate that skipping L2 lookups offers substantial speedup in many of the applications that benefit from level prediction. Additionally, our level predictor is simple and scalable, and can be extended to predict more levels, if required.

Level Prediction Approach. A level predictor must determine whether to skip L2 lookup, L3 lookup, or both. This is distinct from prior work on miss predictors and we find that such prior approaches cannot be directly extended to level prediction despite the fact that level prediction can be performed as a set of per-level hit/miss predictions. The primary reason is that the resources to accuracy characteristics of prior miss predictors, especially if considered for multiple levels, cannot be incorporated in an L2 and queried per L1 miss.

In particular, miss predictors based on classic binary predictors [28], [31], [38] require large tables to achieve high accuracy in the presence of sophisticated prefetchers. We demonstrate this by extending the TAGE-based predictor of Sim et al. [31] to provide ternary level predictions (L2, LLC and main memory). We find the alternative "miss map" approach to miss prediction [23], [26], where per-address presence information is maintained at a specific cache level, more suitable for level prediction yet shows that sophisticated extensions are required.

Our design-space exploration identifies several crucial issues with prior miss-prediction designs, even when extended to LP. We find that the MissMap approach requires tables that are too large for the tight resource budget of LP while the history-based predictors work poorly in the presence of advanced prefetchers, unless significant additional storage resources are made available.

Impact of Prefetchers. One important reason for the poor performance of stacked miss predictors is the presence of advanced prefetchers. While prior publications report very high prediction accuracy for a range of history-based PC-indexed [28], [38] and address-indexed [26], [31] miss predictors, prior evaluations did not include sophisticated prefetchers. We find that prefetchers add "noise" to the hit/miss history and substantially degrade history-based predictor accuracy. We evaluate a range of predictors with and without advanced prefetchers and find that prefetchers necessitate much larger prediction structures.

Furthermore, there is an opportunity for coordinating the prefetcher and level predictor: the prefetcher may update the level predictor about data movement within the hierarchy to improve prediction accuracy. We add such updates to both our LP-extensions of prior miss predictors and to our own LP design described below. We find that even with such updates, history-based predictors perform poorly because either the additional histories from prefetches or the additional entries introduced by prefetch updates overwhelm the prediction tables at sizes that are reasonable for the tight LP resource constraints.

B. Cache Level Predictor Microarchitecture

Given the level prediction considerations described above, we opt for a novel LP microarchitecture that is inspired by the MissMap approach [23] but extends it in three critical ways (Figure 5). First, we extend the MissMap to a *location map* (LocMap), which provides the location of each block (L2, L3, or MEM), requiring 2 bits of metadata per block. Second, the original MissMap is implemented as a cache and loses all information about a block when a MissMap entry is evicted. We find that this either requires a very large MissMap table or leads to low LP accuracy. Instead, we implement the LocMap as an in-memory table containing location information for every block in physical memory, which is cached in a small metadata cache. The metadata is memory-mapped to a reserved physical address range (cf. [10]) and the metadata cache is connected to the memory hierarchy through the L2 cache with L2 cache-block granularity accesses. The long-term location information is then available even after an eviction. Third, because the metadata cache is small and full LocMap access has high latency, we add a small history-based Popular Level Detector that provides fast level prediction on a metadata cache miss. This history predictor requires just three counters.

The LP is physically integrated with the L2 cache in each core and communicates with the LocMap through the LLC. When a block is evicted from an LP cache, it is written back to the LLC. When a block is needed by an LP it is requested through the LLC. With this design, the LP does not require new interfaces to memory and benefits from LLC caching.

The level predictor requires 4 pieces of information from L2 and L3 to keep track of locations. All information eventually flows to the off-chip LocMap through the LP caches. Information on (1) L2 fills (demand and prefetch) and (2) L2 evictions is available within the L2. Information on (3) LLC prefetcher fills and (4) LLC evictions are communicated back to LPs from the LLC controller. For evictions, we assume the LLC tracks which core installed the block and informs only the LP in that core. Similarly, on an LLC prefetch fill, we inform the LP in the core whose L2 miss triggered the LLC prefetch. Such "source core" tracking is needed for (some) cache-partitioning mechanisms as well, so is not in itself novel or unique. We emphasize that LocMap updates are not broadcast across the NoC and incur low overhead.

C. LocMap and Location Tracking

The LocMap is a flat table in system-reserved physical memory and is accessed with the same granularity as the data cache. The LocMap holds the level information of all memory blocks using 2 bits of metadata (there are 3 possible levels to predict: L2, LLC and main memory). Each 64-bit LocMap entry holds location information of 32 blocks. To provide fast access to LocMap, hot metadata is cached on-chip. For 64B cache blocks, this metadata scheme incurs only $\frac{2}{512} = 0.39\%$ overhead.

LocMap Access. The LocMap is accessed (through the metadata cache) on every L1 cache miss, and when it is updated. Each block in physical memory is mapped to an entry in LocMap. Hence, to access the LocMap, we need to generate an address from the block's physical address. To do so, we employ a simple one-to-one mapping. We assume that the base address to the LocMap table is set by the operating system and that the memory access granularity and cache blocks are 64B. Each 64B cache block requires 2 bits in the LocMap such that information for 256 cache blocks fits into a single 64B block of the LocMap (matching the memory access and cache granularity). , the memory address corresponding, the memory address corresponding to a LocMap entry is: $\textit{LocMap Address} = \textit{Base Address} + \textit{Physical Address} >> 14$. This physical address of a LocMap block is first looked up in the per-core LocMap metadata cache, which is filled on a miss through the data cache hierarchy and main memory.

LocMap Update. Level prediction does not need to be 100% accurate. Hence, we can carefully trade accuracy for power-efficiency. This can be achieved by updating the LocMap on certain events. We update the LocMap only on demand cache fills, dirty evictions, and prefetch fills that are metadata cache hits. Thus, the LocMap may hold possibly-stale information because it is not updated on all events happening in the cache. Prefetch fills that are metadata cache misses do not update the LocMap because the traffic this would incur with our aggressive prefetchers is substantial and not worth improving the already-high prediction accuracy (see Section V). Because of the aggressive prefetchers, there are frequent clean evictions and these do not update the LocMap as well. Finally, to avoid changes to the coherence protocol and actions, coherence-induced level changes (i.e., invalidations) are also ignored. Again, staleness is tolerable because the predictor already performs well and because misprediction recovery is inexpensive.

Metadata Cache. The sizing of the metadata cache is important. If the miss ratio is too high, the problem is two-fold. First, many off-chip requests are issued to update the LocMap. Second, the prediction accuracy may degrade as we have to rely on the statistical Popular Level Detector, which is less accurate than the LocMap. At the same time, the size of the LocMap metadata cache is constrained to maintain low access latency and energy—the benefits of level prediction can easily be overwhelmed by an expensive predictor. We show this in the evaluation by demonstrating the detrimental impact of a higher-energy predictor, for example.

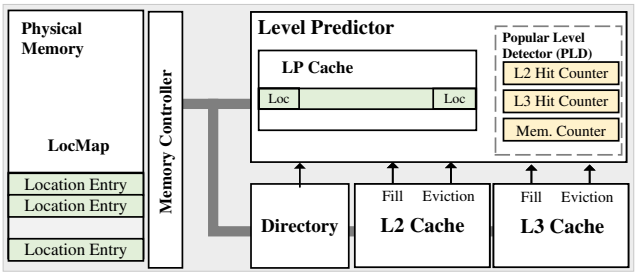

Figure 5. The proposed architecture.

We experimented with a range of sizes and concluded that a 2KB, 2-way set associative cache best balances these tradeoffs and offers the minimal predictor-energy point across multiple benchmark suites: SPEC CPU 2017, GAPBS, NAS, and other applications (*bmt*, *hpcg*, *spmv*, *gups*, and *stream*. We find that a predictor with a 1KB LocMap metadata cache relies too frequently on the Popular Level Detector and suffers from lower accuracy, while the accuracy benefits of 4KB and 8KB metadata caches are negligible.

D. Popular Levels Detector

When there is a miss in the metadata cache, it is not possible to wait to fetch the LocMap entry from main memory because this takes longer than the cache lookup itself. One option is to follow a serial access pattern and predict L2 as the location of the block. However, this option is too conservative and does not cover applications with high metadata cache miss ratios such as *pr*, *bc*, *tc* of the GAPBS benchmark suite. Hence, in conjunction with the LocMap, we devise a simple counter-based mechanism to find the most frequently accessed levels and suggest those as the prediction target(s).

Because the metadata hit rate is relatively high (95% on average across 37 applications), we can be more aggressive on (relatively rare) misses and predict more than one level to increase the prediction accuracy. This is particularly helpful if the counters are not strongly biased toward one level. If one level is suggested, we call the prediction a single-way prediction, and a multi-way prediction otherwise. Multi-way prediction for uncertain cases increases the prediction accuracy, but requires a more complicated lookup.

We use 3 counters, one per cache level and main memory. Upon a hit at a level the corresponding counter is incremented by 1 and others are decremented by 1. This helps to rapidly find popular levels and prevents counter saturation. When a prediction is required, candidates are selected as follows. The counters are sorted and the topmost is picked as the first candidate. If its counter is higher than a threshold, then only this level is selected as the level to look up. Otherwise, the level with the second highest counter is also considered to be

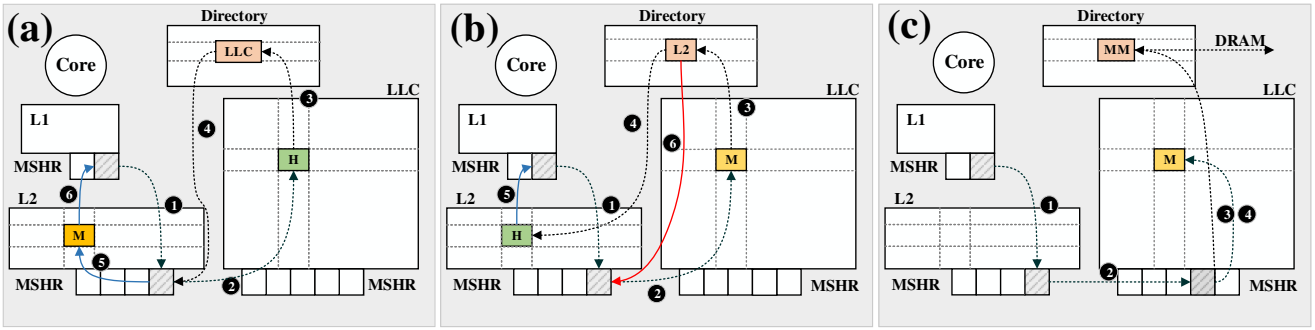

Figure 6. Example of parallel access with cache level prediction: (a) One-way correct prediction (prediction=LLC, actual location=LLC); (b) One-way wrong prediction (prediction=LLC, actual location=L2); (c) Multi-way with a wrong prediction (prediction=LLC and main memory, actual location=main memory)

a possible destination (two parallel lookup targets). If again the sum of the first and the second counters does not reach a predefined threshold, the third level is also included (three parallel lookup targets). Hence, depending on the counter values the locality predictor may issue single- or multi-way predictions.

E. Misprediction Detection and Recovery

Mispredicting a level that is closer to the core than the actual cache level does not require detection and recovery, and is simply a lost opportunity to reduce latency. For example, if the block is only in main memory and LLC is suggested, then not looking up L2 saves cycles, but looking up the LLC wastes CPU cycles; MSHR [22] entries are still allocated along the request path and will be eventually filled as the response arrives. Therefore, this type of misprediction is safe as it does not violate correctness or functionality.

However, mispredicting a miss and bypassing a level that has more up to date data does require recovery as stale data is fetched. For example, if data is present in L2, but only L3 is suggested by the level predictor, then stale data may be incorrectly fetched from L3. To cope with this problem, we slightly modify the directory controller. Normally, the directory is checked to make sure that private caches of other processors are not holding the data block. When level prediction bypasses L2, we also check the directory, as we may have skipped the private cache. Fortunately, this can be done effectively for free. Recent processors collocate the directory with LLC tags, meaning when the cache is looked up, the information of the directory is also available [36]. This enables simple misprediction detection.

Another misprediction type is where main memory is predicted while the block is actually cached. This misprediction is also detected by the directory, which is queried in any case before accessing main memory.

We change the cache controller to raise a signal when a misprediction is detected. Misprediction recovery first issues a packet to the actual level, requesting to satisfy the pending request and then deallocates all MSHR entries past the actual level. This can be implemented as a new transaction over the shared bus.

We present examples of detection and recovery schemes from the simplest (one-way correct prediction) to the most complex (multi-way wrong prediction) case below.

One-way correct prediction. Figure 6 (a) shows an example for one-way correct prediction. Assume the predicted location is LLC, and the block is present in the LLC (green box). When an L1 miss occurs, an MSHR entry is allocated in L1 and the request is sent to L2 (❶). Then without accessing L2, an MSHR entry (denoting an L2 miss) is allocated in L2. Hence, L2 is bypassed and the request is forwarded to the LLC (❷). Note that allocating an entry in the MSHRs for bypassed levels (in this case L2) is necessary, otherwise it would be impossible to fulfill the request later on the fill path. At the LLC, the tag-store is checked (❸). Since, the block already exists in the cache and the directory confirms that the block is not stale, the LLC responds to the request (❹). The block is sent to L2 and the L2 MSHR is deallocated, and it is filled into L2 (❺). Finally, the block is forwarded to L1 which responds to the CPU request (❻).

One-way wrong prediction. Figure 6 (b) shows an example where a block is present in L2, but the predictor suggests LLC. In this example, steps ❶, ❷, and ❸ are the same as before; MSHR entries are allocated in L1 and L2 and the request is subsequently sent to the LLC. However, when the tag store is looked up in the LLC, the extended way information indicates that the block is present in L2 ❹. Thus, the cache controller sends a new request to L2 to fulfill the request ❺. In L2, the block is found and is forwarded to L1 ❻. Also, a signal is raised to deallocate the L2 MSHR entry ❼. The only modification here is to add the ability to cache controller to deallocate the MSHR entry.

Multi-way wrong prediction. Figure 6 (c) shows an example where LLC and DRAM have been suggested as targets by the predictor, but the block is present in DRAM. Steps ❶, ❷, and ❸ are the same as in both the previous examples.

However, when the packet reaches to LLC, both LLC and directory are accessed (steps ❸ and ❹ are simultaneous actions), and after finding the location of the block which is the main memory, the request is forwarded to the main memory. Finally, when the request comes back, it can follow the normal path that any miss sourced in DRAM can take. Note that in our design the directory and LLC tags are collocated, thus as soon as the tag is accessed, the request is sent to main memory.

IV. Evaluation Methodology

A. Simulated Systems

We use the gem5 full-system cycle-level simulator to conduct the experiments [12]. We model a 3-level cache hierarchy where L1 and L2 are inclusive and private and L3 is non-inclusive and shared. L1 and L2 are parallel caches where tag and data stores are accessed in parallel with access latencies of 4 and 12 cycles, respectively. L3 is a sequential cache with latencies of 15 cycles and 25 cycles for tag and data, respectively. There is a first-level TLB of 64 entries, a second- level TLB with 3072 entries that are equally partitioned between 4KB and 2MB pages (1536 each). The L2TLB is 4-way set associative with a 4-cycles access latency. There are 2 page walkers per core.

We compare the final level predictor configuration described in Section III-B with the following systems:
(1) *BaselineStrong*: a realistic system configuration with an advanced prefetch scheme (Table I);
(2) *BaselinePrior*: a weak-prefetcher configuration that matches the best methodology used in prior work (Table I);
(3) 2KB and 8KB TAGE-based level predictors. We augment each entry of TAGE with 3 counters representing the three memory levels. Having three counters, we use similar heuristic as in Popular Level Detector to suggest a level. We compare both a 2KB TAGE-based level predictor that has comparable predictor energy to our final design but suffers from more mispredictions and to an 8KB TAGE (accuracy competitor) that approaches the accuracy of our final design, but which consumes far more energy;
(4) D2D [29]: a design that extends TLB entries with location information of the blocks (levels and ways), thus traversing the memory hierarchy with a single lookup. D2D relies on a centralized cache-like structure called Hub to keep track of location information. D2D has a high implementation cost, but offers high energy efficiency high accuracy. We assume that the D2D *Hub* is an 8-way 4KB cache. Additionally, we assume that the eTLB requires 10% higher energy per access as it increases the length of entries [29]; and
(5) Ideal: a system where misses do not incur any performance penalty; we configure a 0-cycle miss latency with other functionality of the simulator remaining the same (e.g., MSHR misses are still counted for misses).

B. System Configuration

We experimented extensively with a wide range of state-of-the-art prefetchers and their combinations. The highest-performing scheme overall in our experiments uses the DCPT prefetcher [15] with degree 2 in L3, stride prefetchers of degrees 2 for L2, and BOP [24] for L1 with a degree of 1. DCPT exhibits the highest coverage and high accuracy (Figure 4) and worked well in combination with the L1 and L2 prefetchers.

We also find that always enabling these prefetchers significantly degrades system performance for some applications (e.g., 605.mcf) because the prefetchers contend too strongly with demand requests. We therefore implement two prefetch throttling mechanisms. In the first scheme, we reserve 25% of MSHR entries for demand accesses, which decreases the prefetch rate and maintains some minimum demand request service. The second throttling mechanism is that we monitor the performance of the prefetcher periodically and disable a prefetcher when its accuracy drops below 40%. Specifically, in each epoch of 10 million accesses, the prefetchers operate for the first 1 million accesses, then the prefetcher accuracy determines if the prefetcher remains enabled for the following 9 million accesses.

We simulate out-of-order cores with a fetch width of 4 instructions, 192 ROB entries, and 64-entry store and 32-entry load queues. The frequency of the system is set to 4 GHz.

We use a single DDR4-2400 x64 channel (one command and address bus), with timings based on a DDR4-2400 8 Gbit datasheet (Micron MT40A1G8) in an 8×8 configuration. Total channel capacity is 16GB. This maintains a reasonable core-to-memory ratio for the simulations.

C. Benchmarks

We evaluate the applications of: (1) SPEC CPU 2017 [33], (2) GAPBS [11] (*pr*, *tc*, *cc*, *bfs*, and *bc*), (3) NAS (*cg*, *ft*, *is*, *mg*, and *ua*) and (4) *bmt*, *hpcg*, *stream-copy*, and *gups*. In the evaluation section we report averages for the full benchmark suites, but choose to highlight 21 applications to maintain readability of figures. We pick 12 applications that we expect to highly-benefit from level prediction (within the green box of Figure 2) and 9 applications that we expect to exhibit smaller benefits (from within the red box).

All SPEC CPU applications are run with the reference inputs. We use the Twitter [1] dataset for GAPBS, with the exception of *tc* that uses a synthetic graph of 2^{25} nodes. For NAS, input class *C* is used. For *gups*, we replace the random generator with the C++ built-in random generator to ensure that the table is randomly accessed. The table size is 8GB and 4 million locations are accessed. We compile all benchmarks with gcc/gfortran and -O3 flags.

We use the SimPoint methodology [16] to find representative regions of each application. We use 2 SimPoints of 250 million instructions each and 250 million instructions

Table I
EVALUATED SYSTEM CONFIGURATION.

Processor	Single and Quad-core, 4.0 GHz, Ubuntu 16.04 OS. ROB:192, LQ:64, SQ:64, Fetch-width=4
L1 Cache	32kB 4-way; LRU; 4 cycles. Prefetchers: *BaselinePrior* [29], [30]: Stream, *BaselineStrong*: BOP with degree=1
L2 Cache	256KB 8-way; LRU; 12 cycles, Prefetchers: *BaselinePrior* [29], [30]: None, *BaselineStrong*: Stride with degree=2
L3 Cache	2MB single-core and 8MB multi-core; 16-way; Sequential cache (15+25). Prefetchers: *BaselinePrior* [29], [30]: None, *BaselineStrong*: DCPT prefetcher degree of 2
Main Memory	16 GB: DDR4-2400 x64, 8x8 Micron MT40A1G8

Table II
MULTI-PROGRAM AND MULTI-THREADED APPLICATIONS.

mix1: GAPBS.bfs, lbm, NAS.lu, bmt	mix2: roms, NAS.mg, fotonik3d, gcc
mix3: omnetpp, GAPBS.pr, cam, NAS.cg	mix4: cam, NAS.cg, wrf, NAS.bt
mix5: GAPBS.bfs, lbm, wrf, NAS.bt	
MT1: GAPBS.pr with 2 threads	MT2: GAPBS.pr with 4 threads

of warmup. For kernels (*gups*, *stream*, *bmt*), we annotate the code with gem5 pragmas to simulate just the region of interest.

For multi-core evaluation, we use a set of multi-program and multi-threaded applications listed in Table II. We use level prediction accuracy from single core simulation to observe how different applications with high, medium, and low prediction accuracy interact. From application mixes, mix1, mix3, and mix5 have 2 high expected benefit applications (green box) and 2 expected medium benefit applications (red box), mix2 has 1 high-benefit application and 3 medium-benefit applications (red box), and mix4 has 4 expected medium-benefit applications. For multi threadeds, we focus on GAPBS.pr with 2 and 4 threads. Given the GAPBS.pr has one the lowest single-core hit prediction accuracy, we can observe how level prediction accuracy changes as the contention increases, and how the accuracy is impacted as the LocMap does not update the prediction table on snoop invalidations.

V. EVALUATION RESULTS

A. Single-Core Performance

Figure 7 and Figure 8 show the IPC improvements for the 2KB and 8KB TAGE predictors, D2D, our final LP, and the idealized system, when normalized to *BaselinePrior* and *BaselineStrong*, respectively. The geomean speedups for *BaselinePrior*/*BaselineStrong* are: 2KB-TAGE 9%/3.8%, 8KB-TAGE 9.4%/4.8%, D2D 11.2%/6.3%, final LP 10.3%/6.1%, and Ideal 11.6%/6.7%.

There are two key takeaways overall. First, the final LP is within 90% of the ideal speedup and within (95%) of the far more intrusive D2D architecture, on average. At the same time, LP offers both far better, and more robust performance improvements than when adapting prior miss predictors to level prediction. The second key takeaway is that prefetchers have a big impact on the relative success of the different schemes. With the poor prefetchers of *BaselinePrior* (mirroring prior work), the impact of reducing miss latency is greater and the relative performance of all the schemes is closer to that of the idealized system. With the more realistic prefetchers of *BaselineStrong*, the potential speedup is lower, and importantly, the benefits of LP remain far more robust and closer to ideal than with the TAGE-based approach. This is especially evident for benchmarks for which prefetching is highly effective, such as stream and 619.lbm.

We make four additional important observations on the realistic results of *BaselineStrong* (Figure 8). First, with two exceptions, speedup correlates well with the level-prediction potential discussed in Section II and summarized in Figure 2. The largest speedup is achieved for those applications for which sequential lookup is harmful (application within the green box in Figure 2): 619.lbm, 649.foton, all the gapbs applications, and gups. The first exception is 605.mcf where speedup is just 3%. The results of a top-down microarchitecture analysis [37] show that 605.mcf is both memory-bound (35%) and front-end-bound (31%). This, together with relatively high memory-level parallelism, limits the potential benefit of level prediction. The second exception is nas.is, which falls outside the green box of Figure 2 but still exhibits high speedup. We attribute this anomaly to the better prefetchers available on the commercial Intel core compared to our simulated processor. Indeed, when performing the same analysis with our simulated result, nas.is falls within the green box.

The second observation is that bypassing L2 is very useful. Bypassing L3 lookup and directly accessing memory instead has somewhat higher benefits than skipping the much-lower latency L2, but bypassing L2 is still very useful. Even just bypassing L2 offers $> 5\%$ speedup for many applications.

Third, LP nearly matches the speedup of Ideal and D2D in all but two cases: 650.roms and nas.is. Both benchmarks exhibit high speedup potential and lower prediction accuracy compared to other applications with high-speedup potential. As discussed below and shown in Figure 12, LP only successfully bypasses a relatively small fraction of accesses (40%), while also requiring recovery relatively frequently (20%). The other two similar applications are 605.mcf and nas.ft, but those offer minimal speedup opportunity. The reason for the large speedup difference with nas.is is different. For nas.is, the LP relies heavily on the PLD, which frequently suggests parallel L2 and L3 access. This increases pressure on the cache ports, realizing lower speedup than Ideal and D2D, which do not attempt parallel accesses.

Finally, Figure 9 shows the average memory access latency for LP and Ideal. *BaselineStrong* is shown with a red line. The average memory access latency is improved by 20% on average. Graph applications obtain lower memory access latency with LP because they have high miss ratios at all levels and avoiding those unnecessary lookups helps reduce memory access latency. The trends match the speedup trends overall.

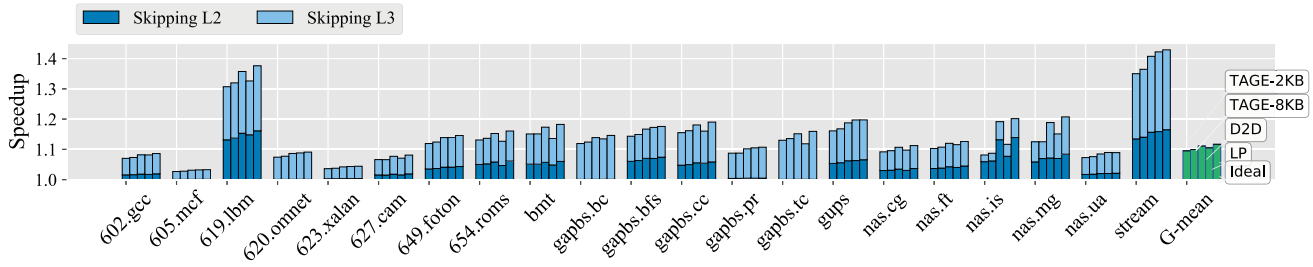

Figure 7. IPC improvement for a single core normalized to *BaselinePrior*. For each benchmark, the bars are 2KB-TAGE, 8KB-TAGE, D2D, level prediction, and Ideal from left to right.

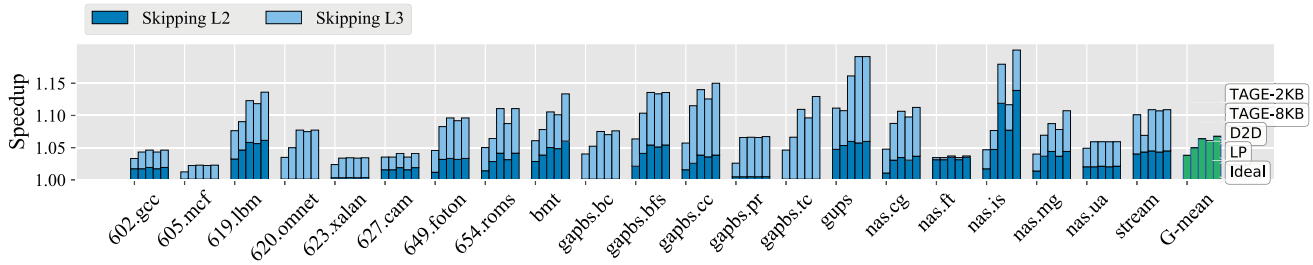

Figure 8. IPC improvement for a single core normalized to *BaselineStrong*. For each benchmark, the bars are 2KB-TAGE, 8KB-TAGE, D2D, level prediction, and Ideal from left to right.

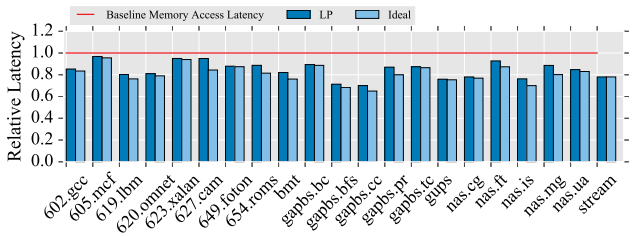

Figure 9. Single-core memory access latency.

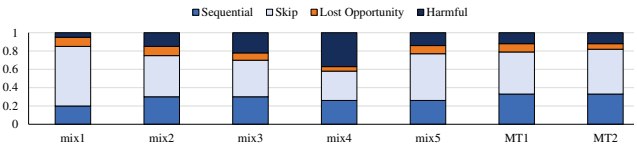

Figure 10. Multi-core level prediction accuracy.

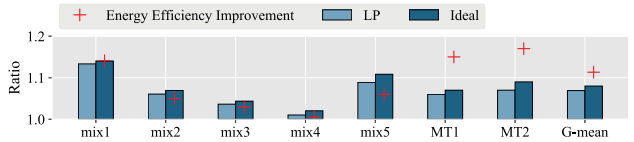

Figure 11. IPC and energy consumption of the multi-core system normalized to the baseline of Table I.

B. Prediction Accuracy

The LP predictions may be: (1) correctly predicted sequential (sequential); (2) correctly predicted skip (skip); (3) wrongly predicted sequential (opportunity loss); or wrongly predicted skip requiring recovery (harmful). Additionally, some predictions are multi-way and add some overhead despite reducing access latency. The overall prediction accuracy (Figure 12) is very high. Only 605.mcf, 620.omnetpp, 649.foton, 654.roms, and nas.ft exhibit accuracies under 90%, and only gups and nas.is exhibit non-negligible lost opportunities. It is also clear that LP correctly identifies a large number of useful non-sequential lookup opportunities.

Interestingly, the lower accuracies are not a result of high LocMap metadata cache miss rates. As shown in Figure 13, those applications with lower accuracies still exhibit reasonable metadata cache hit ratios and high accuracy LDP predictions. Instead, the mispredictions are a result of stale LocMap information originating from a combination of aggressive prefetchers and poor metadata cache locality. When the prefetchers are aggressive, more clean lines are evicted without updating the LocMap. When the metadata cache replacement rate is high, a larger number of prefetch fills miss in the metadata cache and also do not update the LocMap.

Figure 13 also demonstrates the importance of the Popular Levels Detector (PLD). Several benchmarks (605.mcf, 620.omnetpp, gapbs.bc, gapbs.pr, gups, and nas.is) exhibit high LocMap metadata cache miss rates, but those misses use the PLD, which offers high accuracy for these benchmarks. Note that the figure shows the PLD accuracy only considering

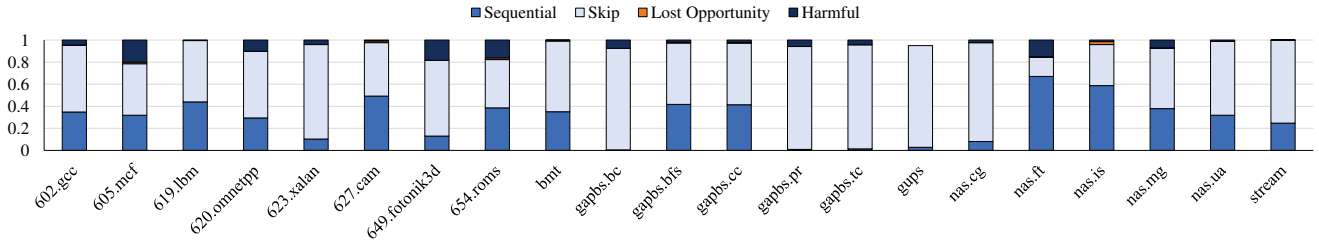

Figure 12. Breakdown of level prediction accuracy. Useful is the accesses that skip at least one level correctly. Opportunity loss is the fraction of lookups the could have avoided. Harmful is the fraction misprediction in all lookups.

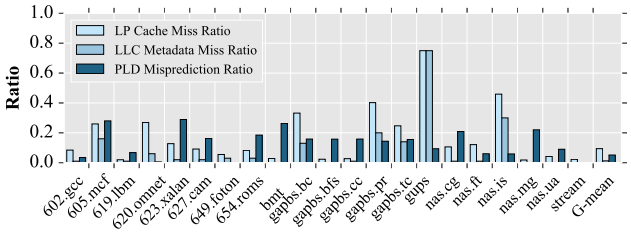

Figure 13. Metadata miss ratios of LP cache and LLC, and misprediction of the Popular Level Detector.

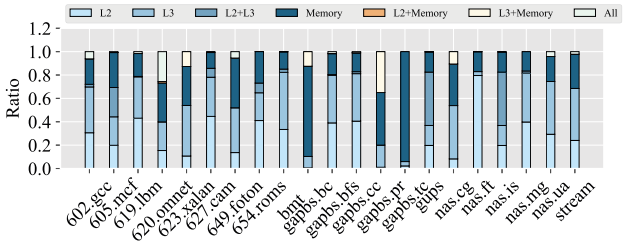

Figure 14. Levels suggested by the level predictor.

those accesses that use it (metadata cache misses).

There are two reasons for the high accuracy of the PLD. The first is that some applications exhibit clear cache-level usage patterns, as discussed in Section II and summarized in Figure 2. For example, gapbs.bc and gapbs.tc exhibit very low hit rates in both L2 and L3, allowing the PLD to frequently suggest skipping both levels.

The second reason is that the PLD can suggest multi-way access to multiple levels in parallel, and thus not mispredict but with access overhead. This is shown in Figure 14 with good examples being: 620.omnetpp with 25% of PLD predictions suggesting all levels in parallel, gapbs.pr with 35% of PLD predictions of parallel accesses to memory and L3, and nas.is with 50% of PLD predictions requesting simultaneous access to L2 and L3. However, for the most part multi-way prediction is rare.

C. Cache Energy Dissipation

We use CACTI to obtain energy per access and then accumulate the total energy. There are two major contributors to the energy consumption of predictors: (1) how frequent the structures are accessed to update and for prediction, and (2) how frequently we have to refer to the directory because of mispredictions. D2D also has two sources of energy overhead: (1) accessing larger TLB entries, and (2) updating the Hub on TLB misses and new insertions. While there are no mispredictions because D2D is a precise scheme, applications with a high TLB miss rate (e.g. *is*) need to access the hub more frequently, and energy consumption increases. Note that our energy analysis here refers to access energy alone and does not account for the additional energy savings resulting from the higher performance provided by level prediction.

Figure 15 shows the energy consumption of LP normalized to the baseline system and also compares this energy to the 2KB and 8KB address+history TAGE variants. We make four important observations. First, our LocMap + PLD predictor is substantially better than either TAGE variant. The 2KB TAGE has the same access energy as the LP, but its accuracy is far lower, which increases recovery overhead. In contrast, the 8KB TAGE offers similar (just slightly lower) prediction accuracy, but its access energy is far higher, resulting in significant additional cache-hierarchy energy.

The second observation is that LP saves energy in all but two cases, with an average energy saving of 16%. The predictor is accurate and the recovery scheme simple, such that on average only 1% of the cache-hierarchy energy is spent on recovery.

The third observation is that in the two cases where energy is slightly increased, the overhead was a result of the very small benefit opportunity available. In 620.omnetpp, the energy overhead was the result of frequent all-level predictions by the PLD, while for nas.ft the overhead was a result of the low potential coupled with a relatively low prediction accuracy of just 80%, and nearly all of those were for sequential lookup. Finally, D2D also has overheads and can only improve on our LP by 3% on average.

D. Multi-Core Results

For multi-core simulation, we enable one LP per core. Figure 10 shows the predictor accuracy for the five multi-process mixes and the multi-threaded applications. Overall, LP accuracy with four cores is lower than with a single core. This is because contention on LLC is greater while prefetch aggressiveness is also substantially larger because

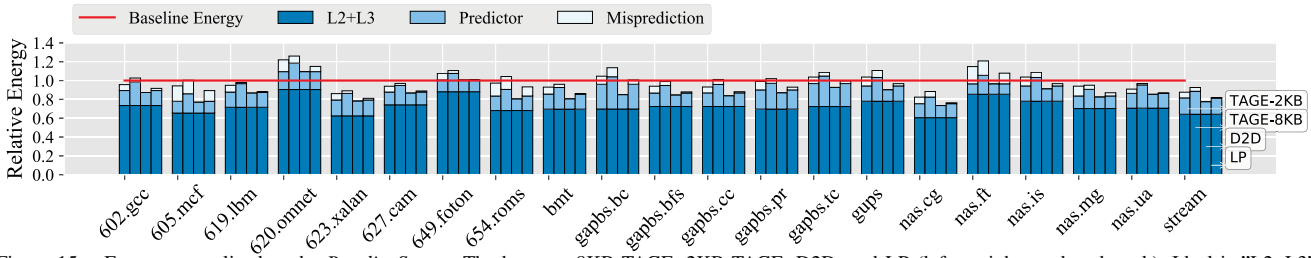

Figure 15. Energy normalized to the *BaselineStrong*. The bars are 8KB-TAGE, 2KB-TAGE, D2D, and LP (left to right per benchmark); Ideal is "L2+L3" only.

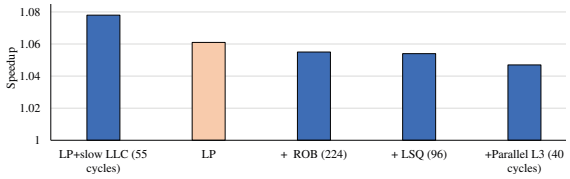

Figure 16. Sensitivity analysis to ROB size, LSQ size and LLC latency for a single core evaluation (average of all benchmarks). We evaluated LP in previous section.

more prefetchers operate in parallel. Still, accuracy is high with the exception of *mix4*. For multi-threaded applications, we run gapbs.pr with 2 and 4 threads on a 4-core processor. While accuracy changes a little between 2 and 4 threads, both harmful and opportunity-loss mispredictions are more frequent than a single thread. This is expected because not only there are greater LLC contention and prefetch aggressiveness, but there is also some degradation in LocMap accuracy. This is because the LocMap is not updated with coherence events and because the LocMap uses a single entry per block even though it may or may not be cached in multiple private L2 caches.

Figure 11 shows the speedup and relative energy-efficiency improvement for the mixes. Notably, level prediction always provides some speedup and cache-energy improvement. For multi-program mixes, *mix1* has the highest IPC improvement (13%), and *mix4* has the lowest (1.2%). The main reason is that *mix4* is composed of four low-MPKI applications and offers minimal speedup potential, whereas *mix1* has 4 very high MPKI applications and offers high speedup potential. Overall, the speedup geomean with LP is 6%, achieving a large fraction of the potential 7% geomean speedup of the idealized system. Energy efficiency is improved by an average (geomean) of 8%, which is again, more than 85% of the potential energy benefits of the ideal case.

For multi-threaded applications, speedup improves as the thread count increases. This is despite the prediction accuracy slightly decreasing because of higher LLC contention. This same LLC contention, however, also increases speedup potential because memory is accessed more frequently.

E. Sensitivity Analysis

Figure 16 compares the average normalized IPC across all applications for 5 experiments: (1) the Baseline system with a slower LLC (55 cycles) to represent new machines with large LLCs, (2) the Baseline system; (3) Baseline but with a more aggressive core (ROB=224); (4) system 3 plus a larger LSQ (96 entries); and (5) system 4 (ROB=224, and LSQ=96) with a parallel-lookup (faster) LLC.

We normalize the IPC with level prediction to that of the same configuration of each experiment without level prediction. Speedup geomeans of the 5 systems are 7.8%, 6.1%, 5.5%, 5.4%, and 4.7%. The overall improvement decreases as the systems become more aggressive. However, even for the most aggressive configuration, level prediction provides 4.7% speedup compared to the clearly aggressive baseline.

F. Bandwidth Analysis

The level predictor has two possible sources of bandwidth overhead: (1) communicating metadata with the LocMap; and (2) mispredictions that suggest skipping all caches while the data is on chip. The LP cache exhibits high hit ratios (>91% shown in Figure 13 first bar) because each cache line covers 16KB of addresses. Coupled with effective LLC LocMap caching (98% of LLC metadata hit ratio shown in Figure 13 second bar), negligible off-chip bandwidth is consumed. The directory stops mispredictions from unnecessarily accessing main memory.

G. Extending LP to Processors with Snoop Filters

LP is currently described and evaluated for a full directory, however non-inclusive caches with snoop filters are also easily supported. Snoop filters track all L2 blocks but are allowed to not track all LLC blocks. While our directory-based recovery is no longer guaranteed, the necessary changes to recovery are minor: If the snoop-filter lookup returns 'not-present' for a block that is incorrectly predicted as bypassing L3, the block is simply dropped at L3 *insertion time*. We expect the impact of the slower recovery to be small given the high LP accuracy and the good coverage of snoop filters (e.g., ARM recommends covering 75% of total cache capacity [7]).

H. Overhead Analysis

Our design requires only a 2KB metadata cache per core as well as three 32-bit wide counters. For each 64B-block in physical memory 2 bits are assigned leading to a memory

overhead of 0.39%. The directory remains unchanged, and only the cache controller is notified with a mechanism to deallocate the MSHR entries on a misprediction.

VI. Related Work

Per level hit/miss predictor is either used in front of L1 cache [26], [38] to handle instruction scheduling better, or only employed at L4 caches [23], [28], [31]. The insight behind L4 hit/miss prediction that the miss penalty is high, and blindly accessing cache incurs high performance and power consumption costs. Our proposal extends such a solution to all memory hierarchy, which is getting deeper recently.

D2M [30] finds the location of a block in the memory hierarchy with a single lookup. D2M separates metadata from the data hierarchy, and uses pointers to forward each request to its destination. D2M requires significant changes to the current processor, such as enlarging TLB entries and adopting a new coherence scheme.

Cache bypassing [21], [35] selectively insert data blocks in the cache. Because many applications with streaming behavior have little reuse, this bypassing retains more valuable data on-chip. Way prediction reduces energy consumption by avoiding searching all ways to match the tag [27].

Software prefetching is an appealing solution to reduce memory access latency for applications with complex address patterns [3], [4], [5]. This technique is useful when the memory access pattern is complex and thereby cannot be captured by hardware prefetchers Pro-actively finding the best time slot to issue the software prefetch request has been studied in [2]. Event-driven software prefe tch generation has been proposed in [4]. Although software prefetch can increase the coverage to almost 100%, it severely suffers from lack of timeliness. its effectiveness.

Many spatial and temporal prefetchers have been proposed in the past decade [3], [4], [5] [13], [14], [15], [18], [19], [20], [24], [32], [40], [41]. Spatial prefetchers rely on spatial behavior of access patterns to predict the next address, and require complex logic to detect access patterns. Temporal prefetchers record past addresses and use them for future predictions, and need a significant amount of metadata to record past sequences. Such prefetchers peak at $\sim 40\%$ miss coverage [9].

LP achieves high accuracy with a simple table. In contrast to D2D and D2M [29], [30], the overall system design remains untouched because LP utilizes existing resources and mechanisms. Our unified predictor needs smaller space and attains better accuracy as it aggregates information from multiple levels and incorporate information from the prefetchers.

VII. Conclusion

We propose a cache level predictor in order to enhance the lookup strategy in multiple-cache setting. This technique filters unnecessary accesses to intermediate levels, and thus reduces the cumulative miss latency. The proposed system enhances the IPC by 6.1% compared to the baseline.

VIII. Acknowledgements

We thank our anonymous reviewers for their valuable feedback and suggestions. The authors acknowledge Texas Advanced Computing Center (TACC) for providing computation resources. This work was funded by the National Science Foundation Grant #1719061.

References

[1] "Twitter graph." [Online]. Available: http://an.kaist.ac.kr/~haewoon/release/twitter_social_graph/twitter_rv.tar.gz

[2] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926254

[3] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017, p. 305–317.

[4] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 578–592. [Online]. Available: https://doi.org/10.1145/3173162.3173189

[5] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses: A microarchitectural perspective," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3319393

[6] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras, "Filter caching for free: The untapped potential of the store-buffer," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 436–448. [Online]. Available: https://doi.org/10.1145/3307650.3322269

[7] ARM, "https://developer.arm.com/documentation/100023/0100/functional-description/operation/snoop-filter?lang=en."

[8] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 131–142.

[9] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 399–411.

[10] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 237–248, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2508148.2485943

[11] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[13] S. R. Brown N. T., "Sandbox based optimal offset estimation," in *DPC2*, 2014.

[14] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 152–162. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155638

[15] M. Grannaes, M. Jahre, and L. Natvig, "Multi-level hardware prefetching using low complexity delta correlating prediction tables with partial matching," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 247–261.

[16] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *J. Instruction-Level Parallelism*, vol. 7, 2005. [Online]. Available: http://www.jilp.org/vol7/v7paper14.pdf

[17] Intel, "Intel® 64 and ia-32 architectures optimization reference manual," 2017.

[18] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 499–500. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542349

[19] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 247–259. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540730

[20] P. V. G. Jinchun Kim and A. L. N. Reddy, "Lookahead prefetching with signature path," in *DPC2*, 2015.

[21] T. L. Johnson, D. A. Connors, M. C. Merten, and W. . W. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.

[22] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, p. 81–87.

[23] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2011, pp. 454–464.

[24] P. Michaud, "A best-offset prefetcherr," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[25] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 96–109.

[26] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 189–198. [Online]. Available: https://doi.org/10.1145/514191.514219

[27] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 2001, pp. 54–65.

[28] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 235–246.

[29] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Navigating the cache hierarchy with a single lookup," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 133–144.

[30] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "A split cache hierarchy for enabling data-oriented optimizations," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 133–144.

[31] J. Sim, G. H. Loh, H. Kim, M. OConnor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 247–257.

[32] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555766

[33] S. Song, Q. Wu, S. Flolid, J. Dean, R. Panda, and J. Deng, "Experiments with spec cpu 2017 : Similarity , balance , phase behavior and simpoints," 2018.

[34] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019, pp. 449–461.

[35] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 76–88.

[36] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2019.00004

[37] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[38] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999.

[39] V. Young, C. Chou, A. Jaleel, and M. Qureshi, "Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 328–339.

[40] V. Young and A. Krisshna, "Towards bandwidth-efficient prefetching with slim ampm," in *DPC2*, 2016.

[41] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 178–190.