A Bypass First Policy for Energy-Efficient Last Level Caches

Jason Jong Kyu Park
University of Michigan
Ann Arbor, MI, USA
Email: jasonjk@umich.edu

Yongjun Park
Hongik University
Seoul, Korea
Email: yongjun.park@hongik.ac.kr

Scott Mahlke
University of Michigan
Ann Arbor, MI, USA
Email: mahlke@umich.edu

***Abstract*—The last level cache (LLC) is critical for mobile computer systems in terms of both energy consumption and performance because it takes a large portion of the chip area and misses often cause expensive stalls. Prior works have studied the importance of bypassing the LLC, and focused on improving LLC performance. However, they did not fully exploit the opportunity for reducing energy consumption because they all employ a *Cache First Policy* (CFP). In CFPs, blocks are initially cached to monitor their re-reference behavior to make bypass decisions. As a result, CFPs tend to populate the LLC with useless blocks, and consume extra energy for unnecessary writes. In this paper, we take the opposite approach and propose a *Bypass First Policy* (BFP), where cache blocks are bypassed by default and only inserted if they are expected to be reused. A BFP can save significant energy by reducing the number of never-rereferenced cache blocks written to the LLC. Evaluations show that BFP reduces energy consumption by 57.1% across SPEC CPU2006 and 21.7% across MediaBench benchmark suites on average. Furthermore, BFP achieves a geometric mean speedup of 18.3% for LLC-intensive benchmarks with less than 8kB of extra storage, which is better or comparable to state-of-the-art CFPs while consuming similar or less storage overhead.**

I. Introduction

Mobile System-on-Chip (SoC) processors are becoming more like desktop-class processors. For instance, the Apple A9 is a dual core, where each core features 6-wide issue and out-of-order execution with 64kB L1 instruction/data caches and a 3MB L2 cache. It also has 4MB L3 cache, which services not only the CPU cores but all the components on the entire SoC. As the last level cache (LLC) in mobile SoC becomes larger as in desktop-class processors, it is becoming a critical component in terms of both energy consumption and performance. For example, the LLC is reported to be responsible for at least 15% of the total processor energy [1], even when the LLC employs low standby power (LSTP) technology to optimize for energy efficiency.

Although a cache is built on the assumption that temporal locality is abundant, the LLC often deviates from the fundamental assumption. Many cache blocks in the LLC do not exhibit temporal locality. For example, a traditional least-recently used (LRU) replacement policy has been shown to have large gap with the better replacement policy for the LLC [2]. Many prior works have studied LLC behavior and showed that bypassing incoming blocks [3], [2] can substantially improve LLC performance. These works observe the re-reference patterns of cache blocks in the LLC, and predict which incoming blocks are not likely to be reused by associating address region [4], program phase [2], or instruction [3], [5] to bypassing predictions.

Although these works have investigated what reference characteristics provide a good indication of bypasses, they all share a common trait of being *Cache First Policies* (CFPs) because they must cache blocks in order to monitor their re-reference behavior. CFPs conservatively over-insert blocks into the LLC to make sure that reused blocks are not bypassed by mistake. As a result, unnecessary energy is consumed when writing never-rereferenced blocks to the LLC.

In this paper, we propose a *Bypass First Policy* (BFP) for the LLC that relies on a rather counter-intuitive behavior: BFP *defaults to bypassing*. BFP reduces the number of never-rereferenced blocks by only caching blocks that are likely to be re-referenced. The key difference between CFPs and a BFP is the way they monitor the reuse characteristics to make decisions. For example, dead block predictions [6] bypass the incoming blocks when they are predicted to be dead, however, these works are CFPs because they make the predictions *after caching blocks to monitor reuse characteristics*. The counterpart in a BFP implementation would be live block prediction, where the incoming blocks are bypassed by default and an opposite prediction is made.

Because never-rereferenced blocks dominate in the LLC, a BFP can reduce the energy consumption significantly as unnecessary writes of never-rereferenced blocks to the LLC are avoided. However, we do not want to sacrifice LLC performance to achieve energy savings. To predict when a block is likely to be re-referenced, we observe the LLC behavior under optimal replacement. We restate the traditional principle of locality with a specialization for the LLC: 1) *reused* blocks are likely to be reused, and 2) memory locations within close proximity are likely to be reused if one of them is *recently re-referenced*. The key idea of BFP is that a cache block is inserted into the LLC only when a reuse (direct or nearby location) has been monitored. With the modified principles of locality, we were able to achieve comparable performance to the state-of-the-art CFPs while reducing energy consumption.

This paper makes the following contributions:

- We motivate why bypass by default is important by showing that the number of bypassed blocks in the LLC

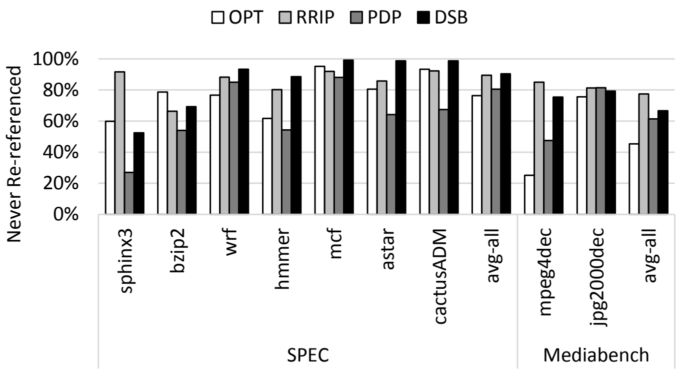

Fig. 1: Percentage of never-rereferenced blocks in the LLC under optimal replacement (OPT) [7], RRIP [2], PDP [8], and DSB [9]. A never-rereferenced block is a block that is victimized with no reuse. We show LLC-intensive benchmarks from SPEC CPU2006 and Mediabench, and the average across all the benchmarks in each suite.

is large even with optimal replacement.

- We analyze the behavior of an optimal LLC replacement policy and revisit the principle of locality customized to the LLC by focusing on how cache hits are generated.
- We propose a *Bypass First Policy* (BFP) that bypasses incoming blocks by default. BFP inserts a cache block only when it is likely to be reused and predicts that a cache block with a recent reuse, or with a recent reuse in the spatially nearby addresses, is likely to be reused.
- We show that a simple BFP can be implemented with a storage overhead of less than 8kB. Energy is significantly saved with a BFP because it avoids unnecessary writes to the LLC. Moreover, BFP achieves comparable or larger performance improvements than prior CFP schemes.

II. Motivation

In this section, we first motivate why cache blocks should be bypassed first instead of being inserted into the LLC by observing the number of never-rereferenced blocks in the LLC under the various replacement policies. We further study the LLC behavior under the optimal replacement to inspire when cache blocks should be inserted.

A. Optimal Replacement Study

Optimal replacement discards a cache block with the furthest reuse in the future. In practice, the optimal policy cannot be implemented because it requires oracle knowledge of future memory references. Although impractical, exploring the LLC behavior under optimal replacement discovers interesting insights on how the LLC should be managed.

Figure 1 shows the percentage of never-rereferenced blocks in the LLC under optimal replacement and other state-of-the-art CFP policies. On the x-axis, we list the LLC-intensive benchmarks from SPEC CPU2006 and Mediabench benchmark suites. We define the LLC-intensity as the percentage of instructions-per-cycle (IPC) improvement as the LLC size

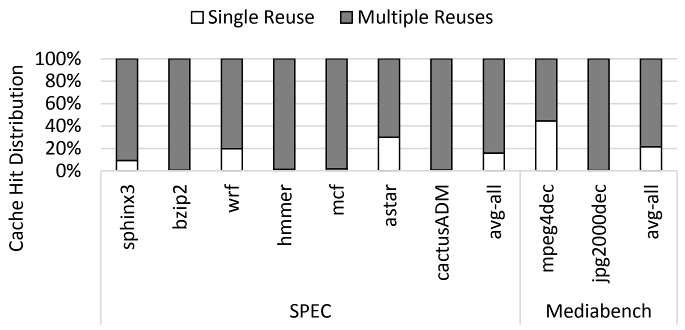

Fig. 2: Distribution of the hits to cache blocks with a single reuse and multiple reuses in the LLC under optimal replacement. We show LLC-intensive benchmarks from SPEC CPU2006 and Mediabench, and the average across all the benchmarks in each suite.

is quadrupled from 2MB to 8MB. The LLC-intensive benchmarks are the ones with more than 25% improvement. We compute the average across whole benchmark suite including non-LLC-intensive benchmarks for completeness.

From Figure 1, we can see that never-rereferenced blocks dominate in the LLC even with the optimal replacement, where an average of 76.4% and 45.3% of cache blocks are never-rereferenced in SPEC CPU2006 and Mediabench, respectively. Many previous works, which also noted this phenomenon, have explored bypass opportunities by predicting which blocks are never likely to be reused. Ideally, if we can perfectly bypass cache blocks that will never be reused, only cache blocks with definite reuses will be inserted into the LLC. With perfect bypassing, the LLC performance would be maximized and the energy consumption will be minimized because never-rereferenced data are not written to the LLC. However, as shown in Figure 1, state-of-the-art CFP policies that exploit bypasses often have larger number of never-rereferenced blocks than the optimal replacement because their bypass predictions are not perfect.

Since never-rereferenced blocks are dominant in the LLC even with the optimal replacement or any state-of-the-art CFPs, we argue that *we should rethink the conventional wisdom on the LLC, which inserts by default*. A BFP that *bypasses by default* and inserts only when a cache block is likely to be reused will significantly reduce the number of never-rereferenced blocks in the LLC, thus, saving dynamic energy significantly. However, BFP should be designed in a way so that it still provides comparable performance to CFPs as the LLC performance is also an important metric.

B. Opportunities

The key for a BFP is to predict when the reuses are likely to occur. Otherwise, the benefit of a BFP may not be realized due to reduced performance. We start from the principle of locality to predict which blocks are likely to be reused in the LLC. Figure 2 shows the distribution of the cache hits in the LLC under optimal replacement. Again, we list the LLC-intensive benchmarks from SPEC CPU2006 and Mediabench

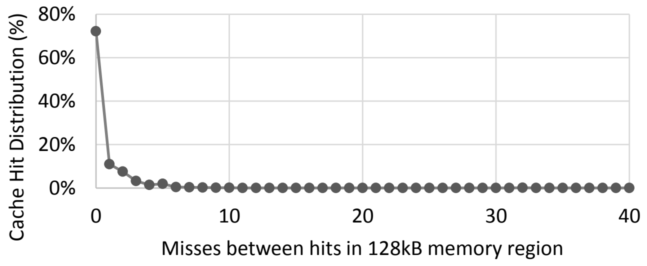

Fig. 3: Distribution of cache hits for the number of misses between hits within a 128kB memory region across the whole SPEC CPU2006 benchmarks under the optimal replacement.

benchmark suites on the x-axis. On average, 84.1% and 78.6% of the cache hits in the LLC come from the cache blocks with multiple reuses for SPEC CPU2006 and Mediabench, respectively. From Figure 2, we make following remark: *cache blocks that are reused at least once, are likely to be re-referenced in the future*. We highlight the "reuse" because typical incoming blocks in the LLC would not observe any reuses. The remark is essentially a restatement of temporal locality with a modification specific to the LLC.

Another well-known form of locality is spatial locality. Figure 3 shows the distribution of cache hits for the number of misses between hits within a 128kB memory region under optimal replacement. We used a 128kB memory region because all the sets with the same tag in the 16-way 2MB LLC form 128kB memory regions. For a memory region with less size, a part of the set index has to be also considered. The figure suggests that cache hits occur in bursts within a 128kB memory region as 72.2% of hits occur consecutively. Note that there is a long tail beyond the forty misses between hits, which is not shown in the graph. It suggests that if the LLC starts observing reuses for a memory region, it is likely that other cache blocks in the same region will be reused in a burst. We make following remark: *if one cache block in a memory region is reused recently in the LLC, cache blocks within the same memory region are likely to be re-referenced*. We highlight the "recent" as well as the "reuse" because the spatial locality is rather observed in bursts. The remark restates the spatial locality in a specialized form for the LLC.

To summarize, we have shown that CFP mechanisms can have inaccurate inserts because never-rereferenced blocks dominate in the LLC. Consequently, it is strongly suggested that cache blocks should be *bypassed by default*, and only be inserted if they are likely to be reused. Studying the optimal replacement, we observed two important characteristics: 1) reused blocks are likely to be re-referenced, and 2) recently reused memory regions are likely to be re-referenced.

III. BFP Design

A. Bypassing First

BFP bypasses incoming blocks by default, and predicts which incoming blocks should be inserted. Intuitively, we would like to only insert blocks that are likely to be re-referenced. Recalling the modified principle of locality in Section II-B, BFP inserts blocks when they were recently bypassed and re-referenced again, or when one of the blocks within the same memory region has been recently inserted.

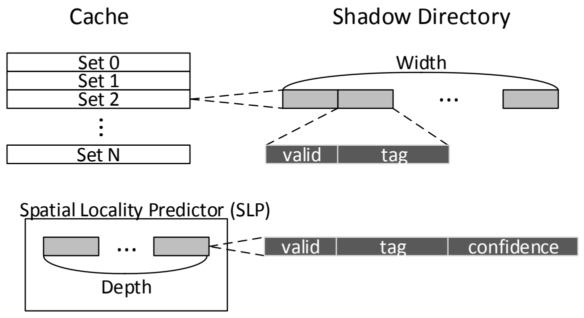

Fig. 4: Architecture overview of BFP. Each cache set has shadow directory, which is a tag array for bypassed blocks, to predict temporal locality. SLP, which is another tag array with confidence, predicts spatial locality for the entire LLC.

Figure 4 illustrates the architectural overview of BFP. To capture the modified temporal locality in the LLC, BFP attaches shadow directories [10] to the cache sets. Note that BFP has shadow directories for all the cache sets unlike previous studies [10], where the shadow directories only existed for sampling sets. These shadow directories hold the few recent bypasses to see if they are re-referenced. Each entry in the shadow directory consists of a valid bit, and tag. To monitor the modified spatial locality in the LLC, BFP has a spatial locality predictor (SLP), which stores recently inserted memory regions. Each entry in the SLP consists of a valid bit, tag, and confidence.

Figure 5 depicts the behavior of BFP when the LLC misses. ❶ When the LLC misses, ❷ the corresponding shadow directory is looked up to see whether the incoming block exists. If exists, the incoming block exhibits the modified temporal locality, thus will be inserted into the LLC. Otherwise, ❸ SLP is searched to check whether any block in the same memory region has been recently inserted. If found, the incoming block is inserted into the LLC because it shows modified spatial locality. ❹ If the incoming block does not exhibit any locality, we put the incoming block in the corresponding shadow directory and bypass the LLC. Note that writebacks are always bypassed in BFP without being inserted into the shadow directory. BFP is not in the critical path as it does not affect cache access time. Choosing whether to bypass or insert can take multiple cycles, and be processed in parallel while generating a request to the main memory.

BFP solely focuses on predicting whether incoming blocks should be bypassed or cached. The underlying promotion and replacement policy of the LLC can be chosen arbitrary. We use a 1-bit not-recently used (NRU) replacement for the practical implementation.

B. Shadow Directory

Shadow directory keeps track of the few recently bypassed blocks for each set, and monitor the reuses. Shadow directory entry is evicted in a round-robin. This requires additional

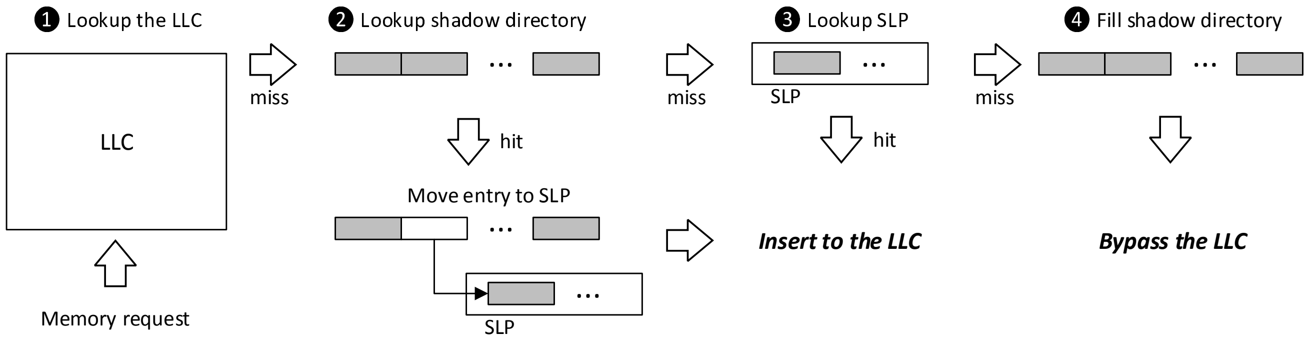

Fig. 5: Behavior of BFP when the LLC misses. BFP searches shadow directory and SLP for the locality. If locality exists, the incoming block is inserted.

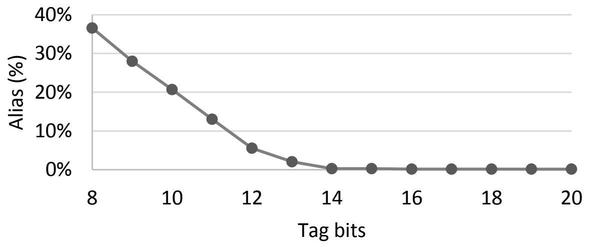

Fig. 6: Average aliases with varying tag bits in the 2MB LLC.

$log_2 width$-bit per set to denote the round-robin position. As shown in Figure 5, shadow directory is filled when the LLC misses and the incoming block is bypassed. A shadow directory entry in the current round-robin position is filled with the incoming block, and the round-robin position is incremented subsequently. A shadow directory entry is invalidated and moved to SLP if it hits.

Shadow directory has two control parameters as shown in Figure 4. First, the number of bits for each tag in the shadow directory entry controls the trade-off between the storage overhead and the aliasing between two different tags. Second, the width of shadow directory, which is the number of entries in the shadow directory per set, determines how many blocks are recorded per set, which also trade-offs between the storage overhead and the monitored number of bypasses.

Figure 6 shows the average percentage aliases over all memory references when tag bits are swept from 8-bit to 20-bit. Once partial tags uses at least 14-bit, aliases are negligible as they are reduced to almost 0%. Therefore, BFP will use a 14-bit partial tag. We found out that shadow directory gives sufficient improvement only with two entries per set as will be discussed in Section IV-C. In such case, the hardware overhead of shadow directory per set is 31-bit, where 30-bit is from an entry with a valid bit and tag, and 1-bit is from the round-robin position. In the 16-way 2MB LLC with 64B blocks, BFP only requires 7.75kB extra storage for shadow directories.

C. Spatial Locality Predictor

SLP stores the recently inserted memory regions. SLP entries are also victimized in a round-robin order, which require additional $log_2 depth$-bit for the round-robin position. Figure 5 illustrates how an entry is allocated in the SLP. When a shadow directory hits, the tag is moved from the shadow directory to the SLP. Note that depending on the size of a memory region, few extra bits from the set index may be appended. When the memory region already exists in the SLP, its confidence is increased by one. On the other hand, confidence of the SLP entry is decreased by one when a block evicted from shadow directory without a reuse is within the memory region of the SLP entry. When the confidence is zero, the SLP entry is no longer valid.

Figure 4 shows that SLP has two control parameters: depth, and tag. SLP uses 2-bit confidence. SLP performs sufficiently well with only one entry as further discussed in Section IV-C. Extra SLP entries seldom provide additional benefits because a memory region is reused in bursts as shown in Figure 3. Similar to the shadow directories, SLP uses a 14-bit partial tag. As will be discussed in Section IV-C, a 64kB memory region gives the maximum benefit for the 16-way 2MB LLC with 64B blocks, which requires one extra bit from the MSB of set index besides the 14-bit partial tag. In a nutshell, SLP requires 17-bit for the 16-way 2MB LLC with 64B blocks.

D. Set Dueling

BFP can suffer if single reuse blocks dominate because BFP's assumption on locality is violated. In fact, we found out that cache first policy (CFP), which inserts by default, is better than BFP for such cases. Set dueling [11] is a technique that dynamically selects between two cache management policies in the runtime. In set dueling, a few sets called *leader* sets are dedicated to each cache management policies. A single counter called *psel* is incremented when the first policy causes a miss in one of its leader sets, and decremented when the second policy invokes a miss in one of its leader sets. Depending on the *psel* value, the rest of the sets called *follower* sets will choose to use the policy with less misses. In the LLC, the behavior in a few leader sets can often be generalized into the whole cache behavior hence set dueling is effective in the LLC. We duel BFP with CFP to utilize CFP when BFP's assumption is violated. Shadow directories for CFP leader sets can be removed for hardware optimization as they are not used at all. However, we choose to keep shadow directories for all the sets because the overhead is only in the range of hundreds of bytes, and the uniformity across the sets may

Architecture	Parameters (SPEC CPU2006)	Parameters (Mediabench)
Processor	4-way Out-of-Order 128-entry ROB	
L1 I-Cache	32kB/4-way/64B blocks 1 cycle access	64kB/4-way/64B blocks 1 cycle access
L1 D-Cache	32kB/8-way/64B blocks 1 cycle access	64kB/8-way/64B blocks 1 cycle access
L2 Cache	256kB/8-way/64B blocks 10 cycle access	2MB/16-way/64B blocks 30 cycles access
L3 Cache	2MB/16-way/64B blocks 30 cycle access	N/A
Main Memory	200 cycle access	

TABLE I: System configuration

be more important for SRAM circuit designs. Like previous works, we use 32 leader sets for BFP and CFP, respectively. Set dueling introduces 10-bit extra storage for *psel* counter.

E. BFP and Inclusion Property

Bypassing the LLC breaks the inclusion property unless upper level caches also bypass the block. However, it is likely that a block is never reused in the LLC because its reuses are satisfied by the upper level caches. BFP can exploit previous techniques to preserve the inclusion property. In RRIP [2], the incoming block is predicted to have low rereference interval so that inclusion property can be preserved until the next few misses. Gupta et al. [12] showed that a small bypass buffer is enough to preserve inclusion property for bypassed blocks until upper level caches can satisfy all the reuses. In terms of energy perspective, the second approach is better because the energy cost of updating a small bypass buffer is smaller than the energy cost of updating a cache line in the LLC.

A bypass buffer is a fully associative cache structure. If a block is bypassed in the LLC, it is inserted into the bypass buffer. The bypass buffer is maintained by pLRU algorithm, and allows a block to remain for inclusion property during the maximum of N misses when it has N-entry. Gupta et al. [12] have shown that 16-entry bypass buffer can preserve inclusion property long enough to have negligible performance impact.

IV. Results

We use a modified version of CMP$im [13], a Pin-based [14] trace-driven simulator to evaluate BFP. A 4-way out-of-order core with 128-entry reorder buffer (ROB) is used for evaluation. We model a desktop-class CPU with a three-level cache system for SPEC CPU2006, and a mobile CPU with a two-level cache system for Mediabench [15]. The upper level caches use traditional LRU replacement. The system configuration parameters are summarized in Table I.

We use LLC-intensive benchmarks from SPEC CPU2006 and Mediabench to evaluate BFP. Benchmarks are simulated for 1 billion instructions. We use SimPoint [16] to identify a single representative 1 billion instruction interval. To select the LLC-intensive benchmarks, we rank benchmarks with the percentage of speedup when the LLC is increased from 2MB to 8MB, and pick the ones with more than 25% improvement. From Mediabench, we ran H.264, MPEG-4, and JPEG-2000. For non-LLC-intensive benchmarks, there is negligible change

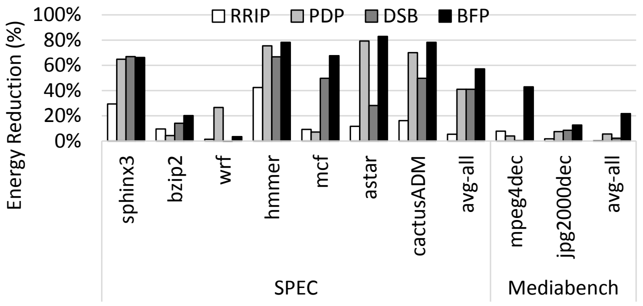

Fig. 7: Comparison of energy reduction over LRU for RRIP, PDP, DSB, and BFP. To be fair, a bypass buffer was attached to PDP, DSB, and BFP to store bypassed blocks for inclusion property.

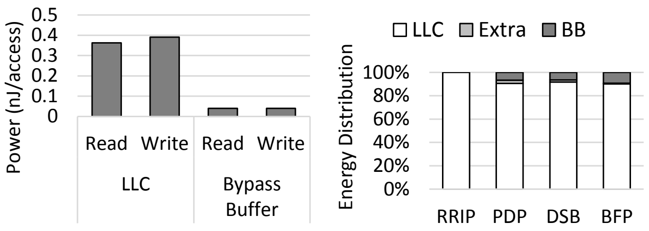

Fig. 8: Dynamic power consumption for the 2MB LLC and the bypass buffer (Left), and a distribution of energy consumption in RRIP, PDP, DSB, and BFP (Right). In the right graph, LLC, Extra, and BB refer to energy consumed by the LLC, extra storage, and bypass buffer, respectively.

in performance with BFP as well as the prior replacement policies that are compared. The baseline LRU replacement does not employ any bypassing mechanisms.

We used CACTI [17] to estimate the energy consumption of BFP. We used 22nm technology LSTP (Low Standby Power) process in CACTI. We assumed 2GHz processor to estimate the static energy. We gathered the accesses to the LLC as well as to the extra BFP structures in the simulated workloads to measure the dynamic activities. The shadow directory is modeled as a 2-way associative cache structure with a 14-bit tag, where the data array was ignored. The SLP is modeled as a direct-mapped cache with a 14-bit tag, where the data array was ignored. We also modeled a bypass buffer, which is a 16-entry, fully associative cache structure.

We compare BFP to three prior works for the LLC: DRRIP [2], PDP [8], and DSB [9]. The respective authors provided C++ source codes that work in the CMP$im environment. They all are CFPs but employ bypasses to improve the LLC performance. We use 2-bit DRRIP with $\epsilon = 1/32$, 32 sampling sets, and 10-bit *psel* counter. We use dynamic PDP-2 with 32-entry RD sampler, $S_c = 4$, and $d_{max} = 256$. We use the first configuration among the three available configurations for DSB.

A. Energy Consumption

We first show how much BFP can save energy compared to CFPs that also employ bypasses. Figure 7 shows the comparison of energy reduction over the traditional LRU

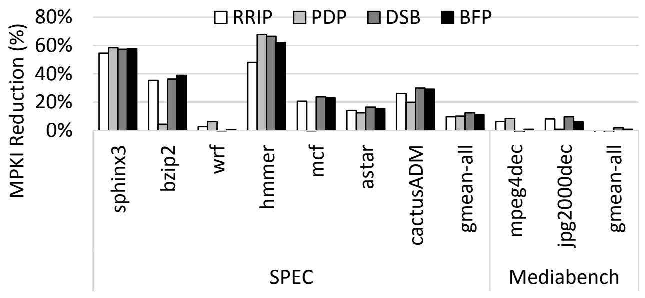

Fig. 9: Comparison of MPKI reduction over LRU for BFP and prior works.

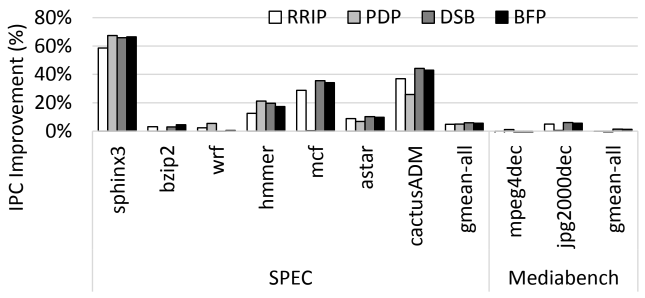

Fig. 10: Comparison of IPC improvement over LRU for BFP and prior works.

replacement. The LLC-intensive benchmarks on the x-axis are listed in LLC-intensive order. For both SPEC CPU2006 and Mediabench, we also show the average across whole benchmarks from each suite. We add a bypass buffer to PDP, DSB, and BFP as explained in Section III-E for fairness because blocks are actually bypassed in PDP, DSB, and BFP, while RRIP inserts a block to the LLC with low priority, which will be evicted at the next miss. Note that PDP does not include the energy consumed by the logic to compute PD.

In Figure 7, RRIP, PDP, DSB, and BFP can reduce energy consumption by 5.4%, 41.1%, 41.1%, and 57.1%, respectively, for entire SPEC CPU2006 benchmark suite. For Mediabench, RRIP, PDP, DSB, and BFP can reduce energy consumption by 0.3%, 5.5%, 2.2%, and 21.7%, respectively. As shown by the results, BFP saves much more energy compared to CFP schemes, where wrf is the only exception where PDP saves more than BFP. In wrf, set dueling results in using CFP mode rather than BFP mode hence BFP consumes similar energy to the baseline. In general, BFP avoids unnecessary writes to the LLC from never-rereferenced blocks, and writes them to the bypass buffer. Figure 8 (Left) shows that the bypass buffer costs much less power than the LLC. Figure 8 (Right) shows that extra energy consumption from additional storage is negligible in all the schemes. Energy is also saved from improved performance, and this reduction is similar for BFP and CFPs because they perform similarly.

B. Performance

Figure 9 shows the MPKI reduction over LRU for RRIP, PDP, DSB, and BFP. Figure 10 illustrates the IPC improvement of RRIP, PDP, DSB, and BFP over LRU. On average, MPKI is reduced by 9.7%, 10.1%, 12.5%, and 11.2% for the entire SPEC CPU2006 benchmark suite with RRIP, PDP, DSB, and BFP, respectively. For Mediabench, MPKI is reduced by -1.9%, -3.1%, 1.9%, and 0.9% in RRIP, PDP, DSB, and BFP, respectively. As a consequence, the system performance is improved by an average of 4.8%, 4.0%, 5.8%, and 5.7% for SPEC CPU2006 benchmark suite in RRIP, PDP, DSB, and BFP, respectively. For Mediabench, -0.1%, -0.4%, 1.5%, and 1.3% improvements are achieved with RRIP, PDP, DSB, and BFP, respectively. If we only consider the LLC-intensive benchmarks, the geometric mean improvements over all the benchmarks are 15.8%, 12.7%, 18.6%, and 18.3% for RRIP, PDP, DSB, and BFP, respectively.

An interesting case is discovered in Figure 10. While PDP outperforms RRIP in $sphinx3$ and $hmmer$, RRIP performs better than PDP for mcf and $cactusADM$. BFP achieves the best performance in both cases by performing closely to PDP in $sphinx3$ and $hmmer$, and outperforming RRIP in mcf and $cactusADM$. This shows that BFP can provide benefits for all the workloads while RRIP and PDP works better than each other for a different subset of workloads. Compared to DSB, BFP achieves the similar performance improvement.

C. Sensitivity

Figure 11 depicts the sensitivity of BFP to the various configuration parameters. The geometric mean of IPC improvement across the LLC-intensive benchmarks, and all the benchmarks in the suites are presented to show the sensitivity.

Figure 11 (a) illustrates the sensitivity of BFP to the number of entries per set in shadow directory when it is swept from one to five. There is a huge gap between when it is increased from one to two. After the width becomes more than three, the performance improvement starts to decrease slowly. This is mainly due to $sphinx3$. While the performance slowly improves for most of the benchmarks when the number of entries is incremented, $sphinx3$ heavily suffers from more entries because monitoring more bypassed blocks escalate the number of inserts significantly to thrash the LLC. On the other hand, mcf and $cactusADM$ are continuously improved as the number of entries is incremented. For example, the performance of mcf can be improved upto 39.1% when the number of entries is five. While the shadow directory performs the best when there are three entries, we decided to use two entries per set as it consumes less storage overhead with similar performance to the three.

Figure 11 (b) shows the sensitivity of BFP to the number of entries in SLP when it is changed from one to four. Although there is no significant change with more entries, IPC improvement with the single entry is slightly higher than the others in the LLC-intensive benchmarks. Because spatial locality is observed in bursts as shown in Section II-B, having more entries in SLP provides no additional benefits. In fact, SLP may pollute the cache by generating more inaccurate inserts beyond the burst. BFP chooses to use one SLP entry.

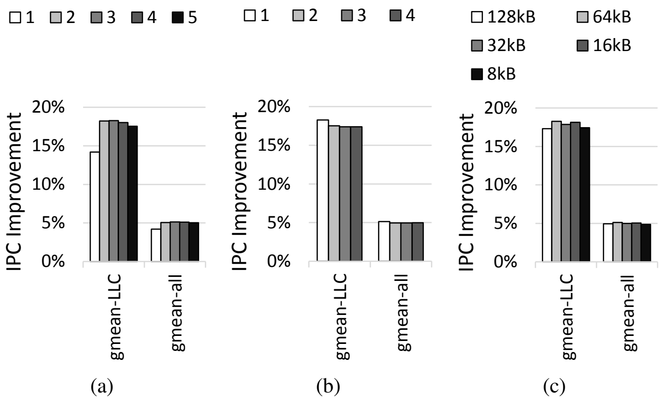

Fig. 11: Sensitivity of BFP to (a) the number of entries per set in shadow directory, (b) the number of entries in SLP, and (c) memory region size. A geometric mean of IPC improvements for the LLC-intensive benchmarks and the whole benchmark suites are presented.

Figure 11 (c) depicts the sensitivity of BFP to the size of a memory region when it is altered from 128kB to 8kB. When all the benchmarks are considered, the size of a memory region has negligible impact. For the LLC-intensive benchmarks, there is no general trend. Having a larger memory region allows SLP to quickly grasp the burst because more sets are monitored. On the other hand, unrelated data structures will be grouped into the same memory region as the size of a memory region grows. While the trend of dominant effect is consistent within a single benchmark, it is not identical across diverse benchmarks. For example, $sphinx3$ consistently performs better with a smaller memory region while $bzip2$ constantly shows more improvements with a larger memory region. Although the difference is small, we still picked a 64kB memory region for BFP because it showed the highest performance improvement in the LLC-intensive benchmarks.

D. Hardware Overhead

BFP can achieve the performance improvement with low storage overhead. Table II compares the hardware overhead of prior works with BFP. PDP and DSB use sampling, and the corresponding storage overhead is shown in the third row. In PDP, 64 sets are sampled together, meaning there are 32 sampling sets (2048 / 64 = 32) in our configuration. In DSB, 32 sets are sampled together, meaning there are 64 sampling sets (2048 / 32 = 64) in our configuration. RRIP and PDP both use 2-bit per cache block state, which consumes 8kB in total. RRIP adds 10-bit *psel* counter as an extra storage. PDP needs an extra storage of 515-bit for the reuse distance (RD) sampler per sampled set, and 0.13kB for the array of RD counters. PDP also requires logic to compute the protecting distance (PD), which is reported to be 10K NAND gates [8]. DSB has 5-bit per cache block state, which is 20kB in total. DSB also requires 22-bit per set for adaptive bypassing, and 748-bit per ATD. BFP uses 1-bit per cache block state, which consumes 4kB, and 31-bit extra storage overhead per set for shadow directory, which is 7.75kB in total. It also utilizes 17-bit for SLP, and 10-bit for *psel* counter. Compared to RRIP and PDP, BFP achieves better performance with comparable hardware overhead. Compared to DSB, BFP achieves similar performance with 2.7x less storage overhead.

	RRIP [2]	PDP [8]	DSB [9]	BFP
Per line	2-bit	2-bit	5-bit	1-bit
Per set			22-bit	31-bit
Per sampled set		515-bit	748-bit	
Extra	10-bit	10K gates* 0.13kB	51-bit	17-bit SLP 10-bit *psel*
Total	8.0kB	10.14kB + 10K gates	31.3kB	11.75kB

*Logic to compute PD

TABLE II: Comparison of hardware overhead. 64 and 32 sets are sampled together in PDP and DSB, respectively.

V. Related Work

A significant number of studies have been proposed to improve the LLC over LRU. We categorize prior works into 1) CFPs, and 2) energy saving techniques for caches.

A. Cache First Policy

CFPs exploiting bypass have received attention from many literatures [3], [4], [9]. The primary concern in the bypass research is to decide which signature correlates well with bypass. Program phase-based predictions [2], which typically utilizes set dueling, and address-based predictions [4] are commonly approaches because they give benefits with small storage overhead. DSB [9] monitors bypasses to check whether they are effective, and changes bypassing probability adaptively. Dead block prediction [6] predicts whether a cache block is dead. A cache block is defined to be dead between the last reference to itself and the time when it is evicted. A cache block, which is dead on insertion, can be bypassed. RRIP [2] place incoming blocks in the lower priority position, which is less aggressive version of bypassing. PDP [8] showed that the LLC performance can be improved when cache blocks are protected for certain number of accesses. When all the cache blocks in a set are protected, the incoming blocks are bypassed. EAF [18] also exploited the modified principle of temporal locality for the LLC: reused blocks are likely to be reused soon. EAF is still a CFP as it stores the recently evicted block addresses to predict the temporal locality. RC [19] exploited the modified principle of temporal locality to reduce the data array size for the shared LLC.

Some have studied instruction-based predictions [3], [6], [5], [20], which incur an extra overhead of transferring the instruction identifier (typically, the partial program counter) to the LLC. These studies usually extend on the their prior works by associating instruction identifiers with the desired property. For example, SHiP [5] further extends RRIP by correlating re-reference interval with a PC or instruction sequence. Because instructions provide more fine-grained correlation than phases, these predictions provide further improvement. The same strategy can be also applied to BFP to give additional benefits, but we leave them as future works.

The key difference between BFP and these works is that these works perform unnecessary writes of never-rereferenced blocks to the LLC because they are in the class of CFPs. BFP can save more energy as these writes are avoided, and also achieve similar or better performance compared to the CFPs.

B. Energy Saving Techniques for Caches

Numerous works have studied energy saving techniques for caches. Leakage power has been tackled both in circuitry and architectural levels, where architectural modifications exploit the circuitry [21], [22]. Circuitry techniques are orthogonal to BFP, and can provide additional improvements from static energy saving. Some have looked at compressing frequent values [23] or selectively activating ways [24] to reduce the dynamic power. Many of these schemes either tradeoff between the performance and energy or focus on reducing energy consumption without improving the performance, while BFP achieves both.

VI. Conclusion

In this paper, we proposed a *Bypass First Policy* (BFP), an LLC management policy that bypasses by default. Traditional CFPs tend to populate the LLC with useless blocks as bypasses dominate. A BFP, on the other hand, effectively reduces the number of inserted blocks, which significantly reduces energy consumption. To provide comparable performance to CFP schemes, BFP investigates the principle of locality in the LLC under the optimal replacement policy, and suggests a slightly modified version of the principle for the LLC: 1) reused cache blocks are likely to be re-referenced, and 2) cache blocks within recently reused memory region are likely to be re-referenced. We implemented a simple BFP using shadow directories and a spatial locality predictor, which consumes less than 8kB of extra storage overhead. Experiments show that the BFP reduces energy consumption by 57.1% and 21.7% for SPEC CPU2006 and MediaBench benchmark suites, respectively, on average. Furthermore, BFP improves system performance by a geometric mean of 18.3% for the LLC-intensive benchmarks, which is better or comparable to the state-of-the-art CFPs.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and feedbacks. This work is supported by the National Science Foundation under grant SHF-1217917.

References

[1] M. N. Bojnordi and E. Ipek, "DESC: Energy-efficient data exchange using synchronized counters," in *Proc. of the 46th Annual International Symposium on Microarchitecture*, 2013, pp. 234–246.

[2] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 60–71.

[3] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," in *Proc. of the 28th Annual International Symposium on Microarchitecture*, Dec. 1995, pp. 93–103.

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

[5] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. S. Jr., and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proc. of the 44th Annual International Symposium on Microarchitecture*, 2011, pp. 430–441.

[6] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proc. of the 28th Annual International Symposium on Computer Architecture*, Jun. 2001, pp. 144–154.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[8] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. of the 45th Annual International Symposium on Microarchitecture*, 2012, pp. 389–400.

[9] H. Gao and C. Wilkerson, "A dueling segmented LRU replacement algorithm with adaptive bypassing," in *Proc. of the* 1^{st} *JILP Workshop on Computer Architecture Competitions*, 2010.

[10] H. Dybdahl and P. Stenstrom, "An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors," in *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 2–12.

[11] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proc. of the 33rd Annual International Symposium on Computer Architecture*, Jun. 2006, pp. 167–178.

[12] S. Gupta, H. Gao, and H. Zhou, "Adaptive cache bypassing for inclusive last level caches," in *2013 IEEE International Symposium on Parallel and Distributed Processing*, 2013, pp. 1243–1253.

[13] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A pin-based on-the-fly multi-core cache simulator," in *Proc. of the* 4^{th} *Annual Workshop on Modeling, Benchmarking, and Simulation*, 2008, pp. 28–36.

[14] 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 *Proc. of the '05 Conference on Programming Language Design and Implementation*, Jun. 2005, pp. 190–200.

[15] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. of the 30th Annual International Symposium on Microarchitecture*, 1997, pp. 330–335.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[17] 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 *Proc. of the 2011 International Conference on Computer Aided Design*, 2011, pp. 694–701.

[18] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012, pp. 355–366.

[19] J. Albericio, P. Ibanez, V. Vinals, and J. M. Llaberia, "The reuse cache: Downsizing the shared last-level cache," in *Proc. of the 46th Annual International Symposium on Microarchitecture*, 2013, pp. 310–321.

[20] J. J. K. Park, Y. Park, and S. Mahlke, "Fine grain cache partitioning using per-instruction working blocks," in *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2015, pp. 305–316.

[21] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," *Proc. of the 29th Annual International Symposium on Computer Architecture*, pp. 148–157, 2002.

[22] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, Feb. 2003.

[23] J. Yang and R. Gupta, "Energy efficient frequent value data cache design," in *Proc. of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002, pp. 197–207.

[24] 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 *Proc. of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001, pp. 54–65.