SCIP: Selective cache insertion and bypassing to improve the performance of last-level caches

Conference Paper · December 2013

DOI: 10.1109/AEECT.2013.6716445

3 authors:

Mazen Kharbutli
Jordan University of Science and Technology
22 PUBLICATIONS **503** CITATIONS
SEE PROFILE

Moath Jarrah
Jordan University of Science and Technology
36 PUBLICATIONS **258** CITATIONS
SEE PROFILE

Yaser Jararweh
Jordan University of Science and Technology
183 PUBLICATIONS **1,227** CITATIONS
SEE PROFILE

SCIP: Selective Cache Insertion and Bypassing to Improve the Performance of Last-Level Caches

Mazen Kharbutli, Moath Jarrah, and Yaser Jararweh
High Performance and Cloud Computing (HPCC) Group
Jordan University of Science and Technology, Irbid, Jordan
{kharbutli,mjarrah,yjararweh}@just.edu.jo

Abstract*—The design of an effective last-level cache (LLC) is crucial to the overall processor performance and, consequently, continues to be the center of substantial research. Unfortunately, LLCs in modern high-performance processors are not used efficiently. One major problem suffered by LLCs is their low hit rates caused by the large fraction of cache blocks that do not get re-accessed after being brought into the LLC following a cache miss. These blocks do not contribute any cache hits and usually induce cache pollution and thrashing. Cache bypassing presents an effective solution to this problem. Cache blocks that are predicted not to be accessed while residing in the cache are not inserted into the LLC following a miss, instead they bypass the LLC and are only inserted in the higher cache levels. This paper presents a simple, low-hardware overhead, yet effective, cache bypassing algorithm that dynamically chooses which blocks to insert into the LLC and which to bypass it following a miss based on past access/bypass patterns. Our proposed algorithm is thoroughly evaluated using a detailed simulation environment where its effectiveness, performance-improvement capabilities, and robustness are demonstrated. Moreover, it is shown to outperform the state-of-the-art cache bypassing algorithm in both a uniprocessor and a multi-core processor settings.

Keywords*—*Cache Memories; Caches; LLC; Cache Bypassing; Shared Caches

I. Introduction

The design of an effective last-level cache (LLC)[1] continues to be the focus of momentous research for three main reasons: First, the ever-widening performance gap between the processor and main memory necessitates the design of an effective cache hierarchy capable of reducing the average memory access times perceived by the processor. Second, by utilizing instruction-level parallelism (ILP) and out-of-order execution, modern processors can efficiently hide a miss in the L1/L2 caches if the access hits in the LLC (L3 cache). Contrarily, it is impossible to hide the long LLC miss penalty during which the processor stalls for tens-hundreds of cycles waiting on the miss. Third, in multi-core processors with a shared LLC - the prevailing design choice nowadays - the LLC must not only maximize performance and throughput, but must also guarantee fairness between the threads competing for its limited, yet valuable, space [1], [2].

Unfortunately, many studies have pointed out the modest effectiveness of traditional cache designs when implemented in LLCs. Figure 1 shows the fraction of LLC blocks that never get re-accessed (re-referenced) while residing in the cache between the time the block is brought into the cache and the time it is evicted from the cache. In other words, these never re-accessed blocks do not receive any hits while in the LLC. Such blocks unnecessarily occupy valuable cache space and cause cache pollution and thrashing if they had replaced more useful cache blocks when they were brought into the cache. The figure shows that for about half of the 23 SPEC CPU2006 benchmarks [3] used in our study, over 90% of the blocks brought into the LLC never get re-accessed before they are evicted. On average, about 82% of all LLC blocks are never re-accessed. This confirms similar observations made by other researchers [4]–[7]. The main reason for this adversity is that locality is often filtered by the higher-level (L1/L2) caches and is, consequently, inverted in the LLC [7]–[10].

An effective solution to the never re-accessed blocks problem is *Cache Bypassing* [4]–[7], [11]–[13]. Cache bypassing

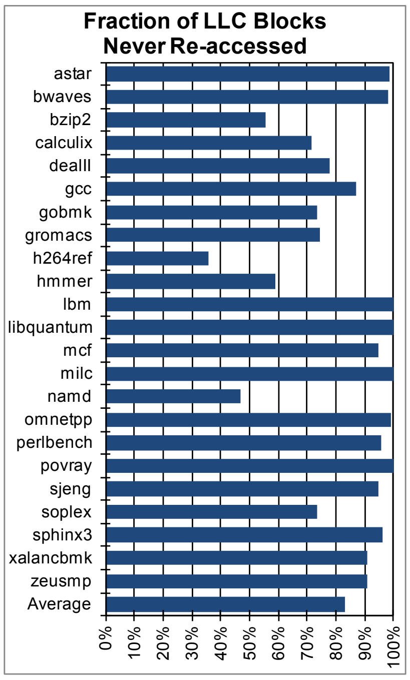

Fig. 1. Fraction of LLC blocks that are never re-accessed after being brought into the cache. Evaluation environment details are in Section IV.

[1]Throughout this paper, a three-level cache hierarchy in which the L3 cache is the LLC is assumed.

algorithms identify blocks that are predicted to never be re-accessed while in the LLC and do not insert them into the LLC on a miss. Alternatively, these blocks bypass the LLC and are inserted in the higher-level caches only. Performance advantages from bypassing stem from two factors: The first factor is the more efficient utilization of the cache hierarchy capacity by only storing blocks that are expected to be re-accessed. The second factor is the reduction of the BW-demand of the on-die interconnect [13], [14].

Unfortunately, existing cache bypassing algorithms suffer from several shortcomings: First, the hardware cost for some algorithms is prohibitively large as they require large tables and voluminous additional fields in the tag array per cache line [11]. Second, the management algorithms for some bypassing algorithms are quite complex requiring frequent access and processing of different structures and fields [4], [5], [11], [13]. Third, some algorithms are specifically designed for exclusive caches and are inapplicable to non-inclusive caches. Exclusive caches are not favored due to the enormous bandwidth pressure they impose on the on-die interconnect while moving blocks between different cache levels [6], [13]. Fourth, many cache bypassing algorithms group cache blocks into distinct groups based on certain characteristics, treating all blocks within a group similarly [4], [6], [12], [13]. This may induce many mispredictions leading to pathological performance if the behaviors of the different blocks within a group do not correlate. These shortcomings necessitate the development of new and improved cache bypassing algorithms, which is the main goal of this paper.

This paper presents SCIP: Selective Cache Insertion and byPassing, a novel, simple, yet effective, cache bypassing algorithm for non-inclusive caches. SCIP dynamically selects which blocks to insert into the LLC and which to bypass it following a miss based on past LLC access/bypass patterns of the individual cache block. Moreover, SCIP has a modest storage overhead and is implemented using a simple algorithm. SCIP is evaluated using a detailed simulation environment in both a uniprocessor and a quad-core CMP architecture models. When evaluated for a uniprocessor with a private LLC, SCIP speeds up 10 LLC-performance-dependent SPEC CPU2006 benchmarks by up to 75% and 18% on average (relative to the base NRU). When evaluated using a quad-core CMP with a shared LLC, SCIP improves 100 SPEC CPU2006 benchmark quadruples by up to 35% and 9% on average. In addition, SCIP is compared to and is shown to outperform a state-of-the-art cache bypassing algorithm: Optimal Bypass Monitor (OBM) [5] in both a uniprocessor and a CMP.

The rest of the paper is organized as follows. Section II discusses the related work and compares SCIP to other cache bypassing algorithms. Section III presents SCIP and its implementation in detail. Section IV presents the evaluation environment and Section V discusses the experimental evaluation. Finally, Section VI concludes the paper.

II. Related Work

There have been a handful of proposals for cache bypassing algorithms in the literature in the past few years. These proposed algorithms can be classified into two classes: In the first class, cache blocks are divided into groups/bins based on certain characteristics such as their addresses [4], the address (PC) of the instruction causing the miss [5], [12], the number of trips between the L2 cache and the exclusive LLC [13], and access characteristics of the blocks while in the L2 cache [6]. The major shortcoming of bypassing algorithms in this class is that they work at the group/bin granularity treating all blocks in the same group/bin similarly. Consequently, they do not distinguish between the behaviors of different blocks within a group/bin. As a result, many mispredictions leading to pathological performance can occur when the behavior of an individual block does not correlate with that of the other blocks in the group/bin to which it belongs. SCIP, on the other hand, does not divide blocks into groups/bins but, instead, treats blocks at the block granularity. SCIP learns the behavior and access patterns of each block individually and uses this learned information to make bypassing choices for that individual block.

In the second class, the behavior of the individual blocks is learned at run-time and used to make bypassing decisions [7], [11]. SCIP falls into this class. However, the hardware cost for some of the algorithms in this class is relatively high as they require large tables and voluminous additional fields in the tag array per cache line [11]. To reduce the prohibitive storage overhead, tables may only store partial history to keep their sizes manageable, limiting their potential coverage. As we will show, SCIP's efficient and well-designed implementation requires modest storage overhead while achieving large performance improvements.

One of the most recently proposed cache bypassing algorithms - Optimal Bypass Monitor (OBM) [5] - was proposed by Li et al. OBM attempts to learn and predict the behavior of the optimal bypassing. It utilizes a Replacement History Table (RHT) to keep track of recent incoming-victim block pairs. A PC-indexed Bypass Decision Counter Table (BDCT) of saturating counters is used for learning and deciding between bypassing and replacement. OBM requires access to the two tables on every cache access which increases the cache power consumption and may affect its critical path. Contrarily, SCIP only accesses its prediction table on a cache miss. In Section V, SCIP is compared against and is shown to outperform OBM in both a uniprocessor and a multi-core settings.

Other cache optimizations which improve the cache's utilization and performance by improving its hashing/indexing function [15], insertion/replacement algorithm [2], [7]–[10], [16], [17], or partitioning algorithm [18], [19] are orthogonal to cache bypassing and can be implemented conjointly.

III. SCIP: Selective Cache Insertion and Bypassing

This section first examines the potential performance improvement from cache bypassing and the distribution of inserted/bypassed blocks in an optimal setting. After that, the implementation of SCIP in addition to its hardware and storage organization is explained. Finally, other implementation issues are discussed.

A. Cache Bypassing: Potentials and Properties

Figure 1 earlier showed the large fraction of blocks that never get accessed while residing in the cache, which is a

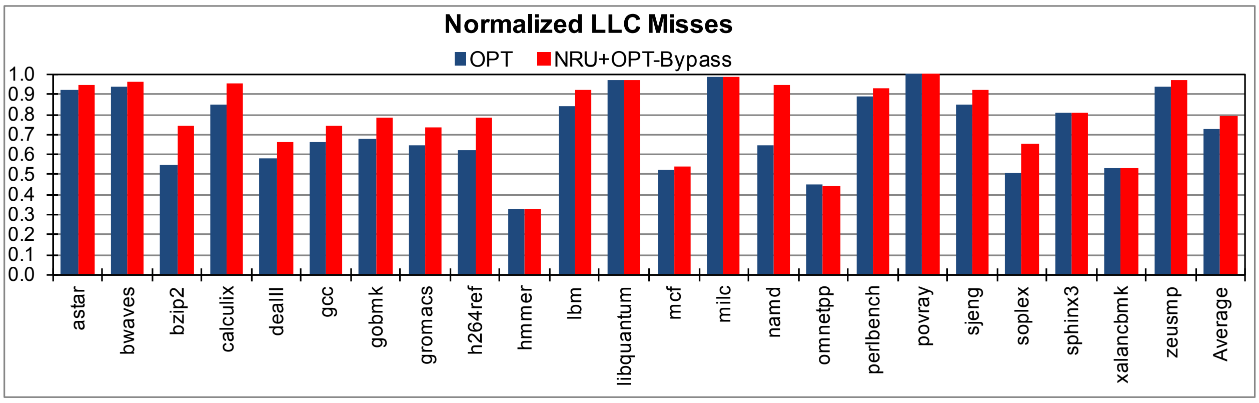

Fig. 2. LLC misses normalized to the base cache replacement algorithm (NRU). Evaluation environment details are in Section IV.

realistic reflection of the potential of bypassing. In order to better gauge the advantages of cache bypassing, an upper bound study of the potential improvement from bypassing was performed. Figure 2 shows the LLC misses for two alternative cache replacement/bypassing algorithms normalized to the base cache replacement algorithm (NRU)[2]. The first algorithm (*OPT*) is the optimal cache replacement algorithm [20] without cache bypassing. OPT is theoretical and cannot be implemented in real processors since it requires future knowledge. However,it can be effectively used to evaluate the potentials of cache bypassing since it gives the lowest possible bound for the number of misses. The second algorithm uses the base NRU replacement with an optimal (oracle) cache bypassing algorithm (*NRU+OPT-Bypass*). The optimal cache bypassing algorithm bypasses an incoming cache block if its future re-use distance is larger than the future re-use distance of the cache block chosen for eviction [5], [6]. The figure shows that both *OPT* and *NRU+OPT-Bypass* outperform the base NRU algorithm eliminating 28% and 21% of the cache misses, respectively. More interestingly is the ability of cache bypassing to bridge about 75% of the performance gap between the theoretical optimal cache replacement algorithm (OPT) and an inferior, yet practical, cache replacement algorithm (NRU).

We also found that the optimal cache bypassing algorithm bypassed on average 77% of all incoming cache blocks over the 23 benchmarks. This large fraction of bypassed blocks correlates with the large fraction of never re-accessed blocks in Figure 1. This high bypassing ratio is indicative of the aggressiveness of the optimal bypassing algorithm in bypassing cache blocks. As will be shown next, this aggressiveness is adopted in the design of SCIP.

Cache blocks can be categorized based on their re-access interval (short, moderate, long) [9], [10]. Blocks that have short re-access intervals can be effectively accommodated by the L1/L2 caches which absorb all the hits on these blocks. Blocks that have long re-access intervals cannot be accommodated by the cache hierarchy and will suffer cache misses at all cache levels. Blocks that have moderate re-access intervals cannot be accommodated by the smaller L1/L2 caches, but can be effectively accommodated by the larger LLC. Consequently, only cache blocks with moderate re-access intervals should be placed in the LLC. Contrarily, blocks with short/long re-access intervals should not be inserted into the LLC and should bypass it instead. An effective cache bypassing algorithm must, therefore, identify the re-access intervals of the cache blocks and insert/bypass them accordingly. This is exactly what SCIP is designed to do.

[2]The Not Recently Used (NRU) replacement algorithm is an approximation of the Least-Recently Used (LRU) algorithm and is commonly-used in modern high-performance processors due to its simplicity and low overhead [10]. LRU and NRU performed equally (within 1%) in all our studies.

B. SCIP's Implementation Details

TABLE I. SUMMARY OF SCIP'S OPERATIONS

On an LLC hit on block B	B.used = 1;

On an LLC miss on block B	if(bypassBuf[B] $<$ 3) {
	   bypassBuf[B]++;
	   doBypass(B);
	}
	else {
	   find a victim block V using NRU;
	   bypassBuf[V] = V.used * 3;
	   doInsert(B);
	   B.used = 0;
	}

Table I summarizes the different SCIP operations. Each cache line in the tag array is augmented with a single bit $used$ that is reset when the block is inserted into the cache following a miss, and is set when the block is accessed while in the cache (cache hit). Once the block is chosen for replacement, the $used$ bit indicates whether the block was accessed while residing in the cache or not. In addition, SCIP uses a small history table ($bypassBuf$) to store the access/bypass history of individual blocks (whether they are in the cache or not) in their previous life generations. A 2-bit saturating counter is stored per cache block.

On an LLC miss on block B, block B's counter in the $bypassBuf$ is consulted. If the counter's value is smaller than 3, then it is predicted that block B will not be re-accessed while in the cache, and is, consequently, bypassed. Block B's counter in the $bypassBuf$ is also incremented. On the other hand, if block B's counter's value is 3, then block B is inserted into the cache. The underlying cache replacement algorithm (e.g. NRU) is consulted to choose a victim block V for replacement. SCIP is independent from, and can work in unison with, any underlying cache replacement algorithm. Upon replacement, block V's counter in the $bypassBuf$ is updated based on whether block V was accessed while residing in the cache (counter set to 3) or not (counter reset to 0).

Note that a block is inserted into the cache (its corresponding counter in the $bypassBuf$ is equal to 3) in two cases: First, if it was re-accessed while residing in the cache in its previous life generation, which is a strong indication that it will very likely be re-accessed again in this life generation due to the natural repetitive behavior of cache blocks. Second, if it was bypassed in the last three times it was brought into the cache hierarchy. This gives SCIP an opportunity to re-learn the behavior of the cache block which may change across different execution phases. Moreover, this 3 bypasses vs. 1 insertion relation for a block corresponds to an aggressive bypassing rate of 75%, which, as indicated earlier, is the average bypassing rate of the optimal bypassing algorithm (77%).

Moreover, programs normally go through many execution phases with different blocks behaviors. To accommodate these different execution phases, SCIP resets all counters in the $bypassBuf$ to zeros periodically (every 8K LLC misses) to allow for re-learning.

The $bypassBuf$ is direct-mapped and tagless with 4K 2-bit entries. Its total size is equal to a modest 1KB, which is less than 0.1% of the size of a 1MB cache. Moreover, the $bypassBuf$ is only accessed on an LLC miss allowing it to be placed off-chip, if needed, and its access time to be fully-overlapped with the LLC miss latency. To index into the table, the block address is divided into 12-bit parts that are XOR-ed together to produce a 12-bit index. We evaluated aliasing effects in the $bypassBuf$ and found that they were minimal. All counters are initialized to a value of 0. In addition, each LLC cache line in the tag array is augmented with a single 1-bit $used$ field. Assuming a 1MB LLC with 64 Byte lines, the total storage overhead for the $used$ fields would be only 2KB, which is less than 0.2% of the cache size.

The discussion above signifies the simple implementation of SCIP and its very low hardware overhead. Nevertheless, as will be shown in Section V, SCIP can improve the performance and effectiveness of the LLC significantly.

C. Other Implementation Issues

Multi-core processors sharing the LLC has become the dominant computing architecture nowadays. When SCIP is implemented in a shared cache, each running thread possesses its own $bypassBuf$ which allows SCIP to accommodate different threads with different blocks behaviors. Sharing the $bypassBuf$ can lead to bias and skewness problems. SCIP's algorithm (Table I) remains exactly as explained earlier.

To support bypassing, it becomes necessary to relax the multilevel cache inclusion property between the LLC and the higher-level (L1/L2) caches. Commercial processor designers have already adopted fully or partially exclusive LLCs as in the AMD Opteron [21]. The cost of relaxing the inclusion property is in duplicating the L1/L2 coherence directory arrays at the LLC level so that the tags can be checked against coherence requests without blocking the processor from accessing the L1/L2 cache in a timely manner. Moreover, Gupta et al. [22] proposed a technique to enable cache bypassing in inclusive LLCs by adding a bypass buffer alongside the LLC to store the tags of bypassed blocks to support the inclusion property.

IV. Evaluation Environment

SCIP is evaluated using a detailed execution-driven cycle-accurate simulator based on [23]. We model a 4-way out-of-order dynamically-scheduled superscalar processor with a 128-entry ROB and 8 stages (with different execution-stage delays). We model a cache hierarchy with 3 levels that is based on an Intel Core i7 system. The different memory hierarchy parameters are listed in Table II. Latencies in the table correspond to contention-free conditions. Port/bus contention and queuing delays are all modeled in detail.

TABLE II. Parameters of the Simulated Memory Hierarchy

L1 Inst. Cache:	32 KB, 4-way, 64B Lines, Private, WB, LRU, 1-cycle, 32-entry MSHR
L1 Data Cache:	32 KB, 8-way, 64B Lines, Private, WB, LRU, 2-cycles, 32-entry MSHR
L2 Cache:	256 KB, 8-way, 64B Lines, Private, WB, LRU, 10-cycles, 32-entry MSHR
L3 Cache (LLC):	1 MB per core (1MB/4MB for uni-/quad-core) 16-way, 64B Lines, Shared, WB, NRU, 30-cycles, 32-entry MSHR
Main Memory:	250-cycles
Memory Bus:	Split-transactions, 16B-wide, 1/4 Proc. Freq.

To evaluate SCIP, we use the SPEC CPU2006 benchmarks [3], which are the standard benchmarks used in evaluating the performance of microprocessors in both the industry and academia. Since SCIP attempts to improve the efficiency and performance of the LLC, its impact will be mostly evident in a subset of the benchmarks where the performance of the LLC significantly impacts the execution time. Out of the 23 SPEC CPU2006 benchmarks used in our study, we have identified 10 LLC-performance-dependent benchmarks which are the focus of our evaluation in Section V. These LLC-performance-dependent benchmarks reap at least 10% performance improvement (speedup) when the LLC size is increased from 1 MB to 4 MB. These 10 LLC-performance-dependent benchmarks are listed in Table III along with their LLC miss rates and the speedups achieved from increasing the LLC size from 1 MB to 4 MB. The benchmarks were compiled with *gcc* using -O3 optimization level. The reference input sets were used for all benchmarks.

TABLE III. The LLC-Performance-Dependent Benchmarks Used in the Evaluation With Their LLC-Based Statistics

Benchmark	Miss Rate	Speedup from 4x LLC size
bzip2	39%	33%
dealII	37%	17%
gcc	72%	37%
gromacs	43%	10%
hmmer	20%	22%
mcf	64%	155%
omnetpp	93%	166%
soplex	40%	59%
sphinx3	93%	11%
xalancbmk	81%	265%

SCIP is evaluated under two configurations: a uniprocessor with a private LLC, and a quad-core processor with a shared

LLC. In uniprocessor simulations, each benchmark is simulated for three billion dynamic instructions after fast forwarding the first five billion instructions. In quad-core simulations, the first five billion instructions in each benchmark are first fast forwarded then the simulation runs until each benchmark has completed three billion dynamic instructions. If a benchmark completes three billion instructions, it continues execution until all benchmarks do. However, results are reported for the first three billion instructions for each benchmark. 100 heterogeneous quadruples from the 23 benchmarks are randomly constructed, in which at least two LLC-performance-dependent benchmarks appear in each quadruple.

V. Experimental Evaluation

In this section, SCIP is evaluated using two architectural configurations: a uniprocessor with a private LLC (Section V-A) and a quad-core CMP with a shared LLC (Section V-B).

A. *Evaluation in a Uniprocessor Architecture*

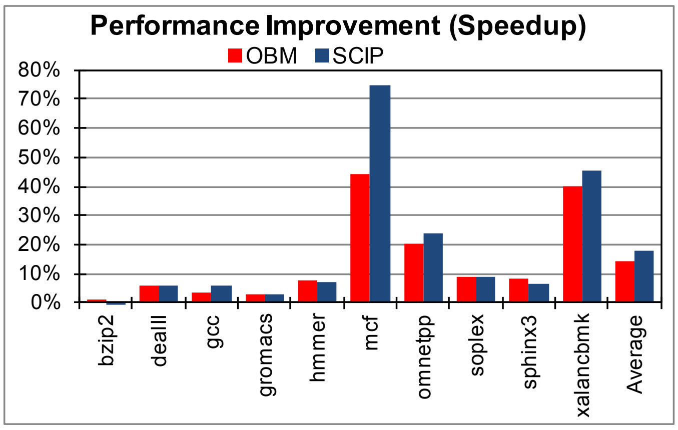

Fig. 3. Performance improvement (speedup) in a uniprocessor.

Figure 3 shows the percentage of IPC performance improvement (speedup) of SCIP and another state-of-the-art cache bypassing algorithm: the Optimal Bypass Monitor (OBM) [5]. Speedups are calculated relative to the base cache configuration with no cache bypassing (Table II). Based on our experiments, OBM was the best performing cache bypassing algorithm to date. The figure shows that SCIP achieves an average speedup of about 18% for the LLC-performance-dependent benchmarks and up to 75% (in mcf). SCIP speeds up 8 of the 10 benchmarks by 5% or more (dealII, gcc, hmmer, mcf, omnetpp, soplex, sphinx3, and xalancbmk) demonstrating its effectiveness. Moreover, SCIP does not slow down any benchmark demonstrating its robustness. OBM also performs well but is clearly outperformed by SCIP. Overall, the average speedup achieved by SCIP is larger than that achieved by OBM (18% vs. 14%). SCIP outperforms OBM significantly in mcf, omnetpp, and xalancbmk. OBM slightly outperforms SCIP in sphinx3 due to the over-aggressiveness of SCIP in bypassing blocks for this benchmark.

TABLE IV. Miss Rates and Bypass Rates for OBM and SCIP in a Uniprocessor

Benchmark	Miss Rate			Bypass Rate
	Base	OBM	SCIP	SCIP
bzip2	39%	38%	41%	87%
dealII	37%	25%	25%	87%
gcc	72%	66%	62%	90%
gromacs	43%	35%	35%	93%
hmmer	20%	12%	13%	82%
mcf	64%	47%	38%	92%
omnetpp	93%	38%	31%	92%
soplex	40%	31%	31%	84%
sphinx3	93%	76%	80%	95%
xalancbmk	81%	49%	46%	90%

Table IV lists the miss rates for the base LLC configuration without bypassing (Base) and with bypassing using OBM and SCIP. The table shows that both SCIP and OBM can effectively reduce the miss rates by bypassing useless cache blocks while reserving useful ones in the cache. Moreover, the table also lists the bypassing rate of SCIP for the 10 benchmarks. SCIP bypasses 82% - 95% of the incoming cache blocks. This aggressive bypassing rate correlates with the large fraction of never re-accessed blocks shown in Figure 1 demonstrating SCIP's effectiveness in identifying these blocks. Moreover, as discussed earlier, such a high bypassing rate was also observed for the optimal bypassing algorithm (Figure 2).

We also evaluated the effectiveness and performance of SCIP in larger LLCs (2 MB and 4 MB) and found SCIP to be also effective when implemented in the larger LLCs achieving average speedups of 15% and 16%, respectively. A detailed evaluation of SCIP's performance in larger LLCs is not included due to paper space limitations.

B. *Evaluation in a Multicore Architecture*

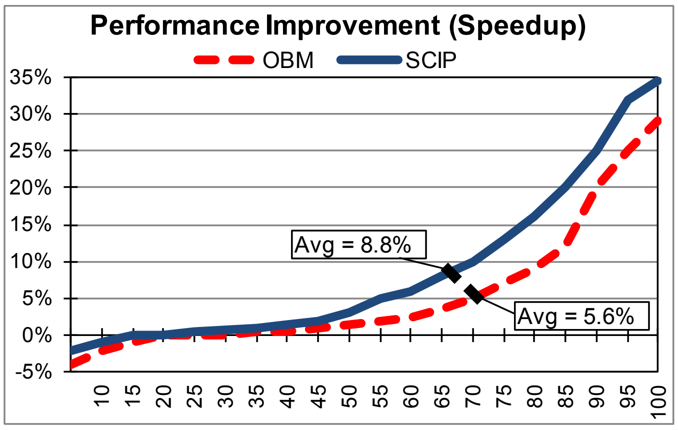

Fig. 4. Performance improvement (speedup) in a multicore processor.

Figure 4 shows the normalized performance improvements of OBM and SCIP for 100 benchmark quadruples when evaluated in a quad-core shared-LLC multicore (CMP) processor. The harmonic mean of weighted IPC values normalized to the base configuration is used. This metric is commonly used because it is a measure of both performance improvement (speedup) and fairness [19], [24]. The horizontal axis covers the 100 benchmark quadruples sorted based on improvement. The figure shows that SCIP achieves performance improvements of about 9% on average and up to 35%. Furthermore, 31 of the 100 benchmark quadruples are improved by 10% or more. These improvements demonstrate SCIP's ability to significantly improve performance and fairness in shared LLCs. As in the uniprocessor evaluation earlier, OBM also performs well but is clearly outperformed by SCIP.

VI. Conclusion

We have presented SCIP, a novel, simple, yet effective cache bypassing algorithm. First, we asserted other researchers' observation of the large fraction of LLC blocks that never get re-accessed while in the cache, consequently, inducing cache pollution and thrashing. Second, we studied the performance of an optimal (oracle) cache bypassing algorithm and demonstrated the large potential of cache bypassing to improve cache performance and efficiency. Third, we proposed SCIP (Selective Cache Insertion and Bypassing). SCIP dynamically learns the behavior of cache blocks at runtime at the block granularity, storing a block's learned behavior in a compact 2-bit counter in a history/prediction table. This learned information is than used to make insertion/bypassing decisions following a cache miss.

SCIP was thoroughly evaluated using a detailed execution-driven simulation environment in both a private-LLC uniprocessor and a shared-LLC quad-core processor. In all experiments, SCIP's robust performance and ability to significantly improve the cache's performance and efficiency was demonstrated. Moreover, SCIP was compared against and found to outperform the state-of-the-art cache bypassing algorithm (Optimal Bypass Monitor - OBM). SCIP's performance improvements are achieved using a simple algorithm with modest hardware overhead, making SCIP an effective and attractive cache bypassing algorithm.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th edition, Morgan Kaufmann, 2011.

[2] M. Kharbutli and R. Sheikh, "LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm," IEEE Transactions on Computers, in press (available online), 2013.

[3] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006.

[4] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu, "Run-Time Cache Bypassing," IEEE Transactions on Computers, 48(12):1338-1354, 1999.

[5] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal Bypass Monitor for High Performance Last-Level Caches," Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 2012.

[6] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing Hierarchy-Awareness in Replacement and Bypass Algorithms for Last-Level Caches," Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 2012.

[7] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," IEEE Transactions on Computers, 57(4):433-447, 2008.

[8] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency," Proc. 41st Annual Int. Symposium on Microarchitecture (Micro), 2008.

[9] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. Steely Jr., and J. Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching," Proc. 44th Annual Int. Symposium on Microarchitecture (Micro), 2011.

[10] A. Jaleel, K. Theobald, S. Steely Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," Proc. 37th Annual Int. Symposium on Computer Architecture (ISCA), 2010.

[11] H. Dybdahl and P. Stenstrom, "Enhancing Last-Level Cache Performance by Block Bypassing and Early Miss Determination," Proc. 11th Asia-Pacific Conf. on Advances in Computer System Architecture (ACSAC), 2006.

[12] T. Piquet, O. Rochecouste, and A. Seznec, "Exploiting Single-Usage for Effective Memory Management," Proc. 12th Asia-Pacific Conf. on Advances in Computer System Architecture (ACSAC), 2007.

[13] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-Level Caches," Proc. 38th Int. Symposium on Computer Architecture (ISCA), 2011.

[14] M. Zahran and S. McKee, "Global Management of Cache Hierarchies," Proc. 7th ACM Int. Conf. on Computing Frontiers, 2010.

[15] M. Kharbutli, Y. Solihin, and J. Lee, "Eliminating Conflict Misses Using Prime Number-Based Cache Indexing," IEEE Transactions on Computers, 54(5):573-586, 2005.

[16] V. Seshadri, O. Mutlu, M. Kozuch, and T. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 2012.

[17] R. Sheikh and M. Kharbutli, "Improving Cache Performance by Combining Cost-Sensitivity and Locality Principles in Cache Replacement Algorithms," Proc. 28th IEEE Int. Conf. on Computer Design (ICCD), 2010.

[18] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 2004.

[19] M. Qureshi and Y. Patt, "Utility-based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," Proc. 39th Int. Symposium on Microarchitecture (Micro), 2006.

[20] L. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," IBM Systems Journal, 5(2), 1966.

[21] Advanced Micro Devices (AMD). Family 10h AMD Opteron Processor Product Data Sheet, June 2010. support.amd.com/us/Processor_TechDocs/40036.pdf.

[22] S. Gupta, H. Gao, and H. Zhou, "Adaptive Cache Bypassing for Inclusive Last Level Caches," Proc. 27th Int. Parallel and Distributed Processing Symposium (IPDPS), 2013.

[23] V. Krishnan and J. Torrellas, "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors," Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 1998.

[24] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," Proc. Int. Symposium on Performance Analysis of Systems and Software, 2001.