Journal of
Low Power Electronics and Applications

Article

A Survey of Cache Bypassing Techniques

Sparsh Mittal

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA; mittals@ornl.gov; Tel.: +1-865-574-8531

Academic Editor: Swaroop Ghosh
Received: 8 March 2016; Accepted: 25 April 2016; Published: 28 April 2016

Abstract: With increasing core-count, the cache demand of modern processors has also increased. However, due to strict area/power budgets and presence of poor data-locality workloads, blindly scaling cache capacity is both infeasible and ineffective. Cache bypassing is a promising technique to increase effective cache capacity without incurring power/area costs of a larger sized cache. However, injudicious use of cache bypassing can lead to bandwidth congestion and increased miss-rate and hence, intelligent techniques are required to harness its full potential. This paper presents a survey of cache bypassing techniques for CPUs, GPUs and CPU-GPU heterogeneous systems, and for caches designed with SRAM, non-volatile memory (NVM) and die-stacked DRAM. By classifying the techniques based on key parameters, it underscores their differences and similarities. We hope that this paper will provide insights into cache bypassing techniques and associated tradeoffs and will be useful for computer architects, system designers and other researchers.

Keywords: review; classification; cache bypassing; selective caching; dead block prediction; non-volatile memory; CPU; GPU; CPU-GPU heterogeneous system

---

1. Introduction

In face of increasing performance demands and on-chip core count, the processor industry has steadily increased the depth of cache hierarchy and cache size on modern processors. As a result, the size of last level cache on CPUs has reached tens of megabytes, for example, POWER8 and Haswell processors have 96 MB and 128 MB eDRAM (embedded DRAM) last level caches, respectively [1,2]. GPUs have also followed this trend in recent years, and thus, the size of last level cache (LLC) has increased from 768 KB on Fermi to 1536 KB on Kepler and 2048 KB on Maxwell [3–6].

Over-provisioning of cache resources, however, is unlikely to continue providing performance benefits for a long time. Caches already occupy more than 30% of the chip area and power budget and this constrains the area/power budget available for cores. For applications with little data reuse, caches harm performance since every cache access only adds to the total latency. Due to this, performance with cache can even be worse than that with no cache [7,8]. These factors have motivated the researchers to explore alternate techniques to improve performance without incurring the overheads of a large-size cache.

Cache bypassing is a promising approach for striking a balance between cache capacity scaling and its efficient utilization. Also known as selective caching [9] and cache exclusion [10], cache bypassing scheme skips placing certain data of selected cores/thread-blocks in the cache to improve its efficiency and save on-die interconnect bandwidth. However, to be fully effective, cache bypassing techniques need to account for several factors and emerging trends, such as nature of processing unit (CPU or GPU), memory technology (SRAM, NVM or DRAM), cache level (first or last level cache), application characteristics, *etc.* For example, cache bypassing techniques (CBTs) proposed for CPUs may not fully exploit the optimization opportunities in GPUs [11] and those proposed for SRAM caches may not be effective for NVM caches [12]. It is clear that naively applying bypassing can even harm performance by greatly increasing the off-chip traffic and hence, intelligent techniques are required for realizing the full potential of bypassing. Several recently proposed techniques seek to address these challenges.

This paper presents a survey of techniques for cache bypassing in CPUs, GPUs and CPU-GPU heterogeneous systems. Figure 1 shows the organization of this paper. Section 2 discusses some concepts related to cache bypassing and support for it in commercial processors. It also discusses opportunities and obstacles in using cache bypassing. Section 3 summarizes the main ideas of several CBTs and classifies the CBTs based on key parameters to highlight their differences and similarities.

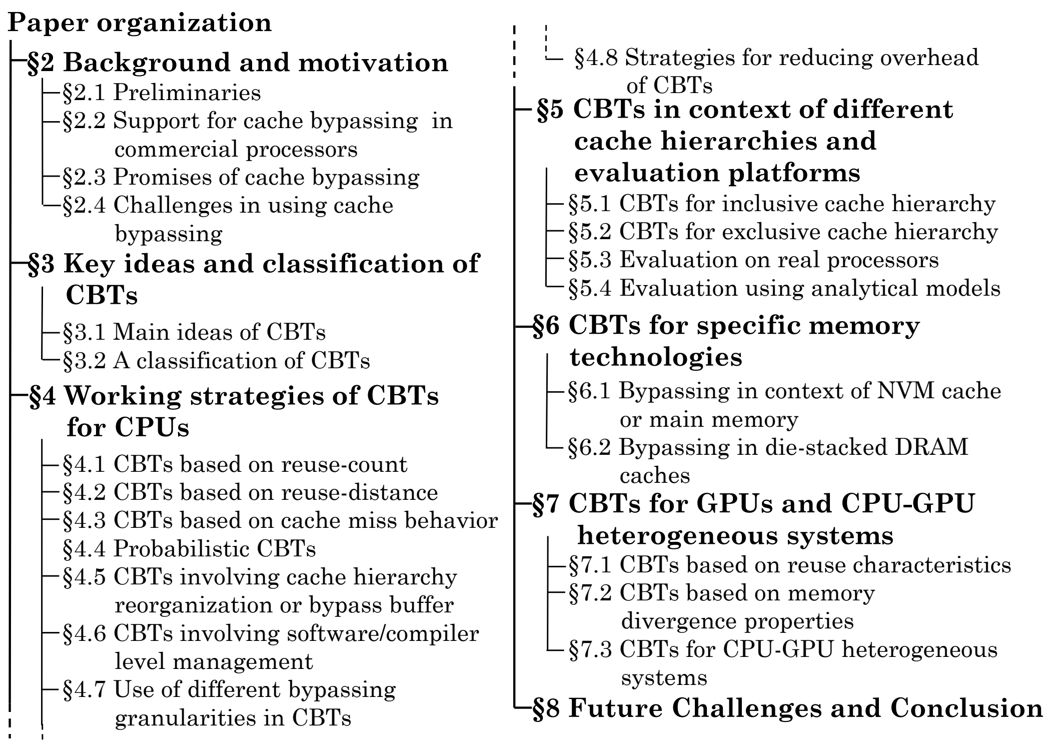
Paper organization
§2 Background and motivation
§2.1 Preliminaries
§2.2 Support for cache bypassing in commercial processors
§2.3 Promises of cache bypassing
§2.4 Challenges in using cache bypassing
§3 Key ideas and classification of CBTs
§3.1 Main ideas of CBTs
§3.2 A classification of CBTs
§4 Working strategies of CBTs for CPUs
§4.1 CBTs based on reuse-count
§4.2 CBTs based on reuse-distance
§4.3 CBTs based on cache miss behavior
§4.4 Probabilistic CBTs
§4.5 CBTs involving cache hierarchy reorganization or bypass buffer
§4.6 CBTs involving software/compiler level management
§4.7 Use of different bypassing granularities in CBTs
§4.8 Strategies for reducing overhead of CBTs
§5 CBTs in context of different cache hierarchies and evaluation platforms
§5.1 CBTs for inclusive cache hierarchy
§5.2 CBTs for exclusive cache hierarchy
§5.3 Evaluation on real processors
§5.4 Evaluation using analytical models
§6 CBTs for specific memory technologies
§6.1 Bypassing in context of NVM cache or main memory
§6.2 Bypassing in die-stacked DRAM caches
§7 CBTs for GPUs and CPU-GPU heterogeneous systems
§7.1 CBTs based on reuse characteristics
§7.2 CBTs based on memory divergence properties
§7.3 CBTs for CPU-GPU heterogeneous systems
§8 Future Challenges and Conclusion

Figure 1. Organization of the paper in different sections.

Section 4 presents CBTs proposed for CPUs in context of conventional SRAM caches. Section 5 reviews CBTs proposed for inclusive/exclusive cache hierarchies. It also discusses techniques evaluated using analytical models and real processors. Section 6 discusses bypassing techniques specific to caches designed with NVM and DRAM memory technologies. Section 7 presents CBTs for GPUs and CPU-GPU systems. In many works, bypassing is used along with other approaches e.g., cache insertion policies. While discussing these works, we mainly focus on the bypassing technique, but also briefly discuss other approaches for showing their connection and the overall approach. Since different works have used different evaluation platforms and methodologies, we mainly focus on their qualitative results. Section 8 presents the conclusion and also discusses some future challenges. We use the following acronyms frequently in this paper: cache bypassing technique (CBT), dead block predictor (DBP), explicitly parallel instruction computing (EPIC), instruction set architecture (ISA), last level cache (LLC), least recently used (LRU), miss-status holding register (MSHR), most recently used (MRU), network-on-chip (NoC), non-volatile memory (NVM), program counter (PC), spin transfer torque RAM (STT-RAM), thread-level parallelism (TLP).

2. Background and Motivation

In this section, we first discuss some concepts and terminologies which will be useful for understanding several CBTs. We then show the support for cache bypassing in commercial processors. Finally, we discuss the promises and challenges of using cache bypassing.

2.1. Preliminaries

The access stream to a given cache block can be logically divided into multiple generations. Figure 2a shows a typical access stream for one cache block. A cache miss brings a block into the cache, which begins a generation. The time period during which the block sees multiple accesses is termed as *live time* and time periods between different accesses are termed as *access intervals*. The *reuse count* shows the number of references to a block while staying in the cache and in Figure 2a, the reuse count is 4. The last access/write before eviction is termed as closing access/write, respectively [13]. After the last access, the block is termed as *dead* because it has no more reuse. Clearly, a block with zero reuse count is called *dead-on-arrival*. Eviction of a block from the cache ends one generation and the time period from insertion to eviction is called *generation time*.

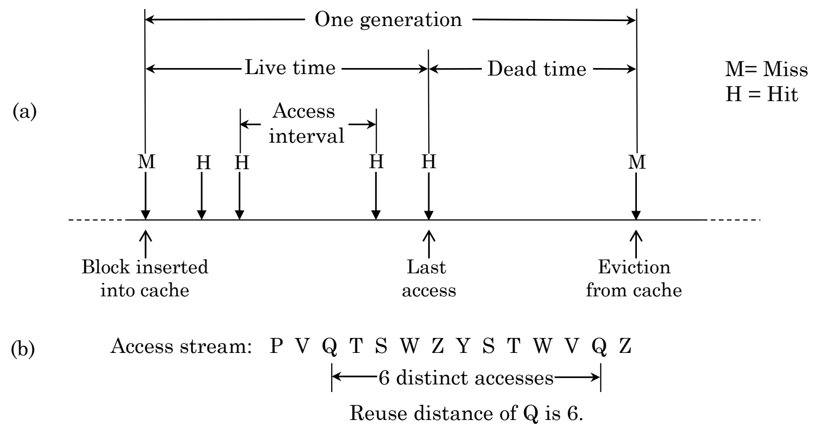

Figure 2. (**a**) Illustration of one generation in cache access stream and (**b**) Determining reuse distance.

The *reuse distance* shows the number of accesses seen by a cache set between two accesses to the same cache line (For other definitions of reuse distance, we refer the reader to [14]). This is illustrated in Figure 2b. The *program working set* is defined as the unique addresses referenced in a given time window [15]. Belady's OPT [16] is an offline replacement scheme that evicts the block accessed furthest in the future and thus provides a theoretical lower bound on miss-count.

2.2. Support for Cache Bypassing in Commercial Processors

Several commercial processors provide support for cache bypassing. For instance, Intel's i860 processor [17] provides a special load instruction termed PFLD (pipelined floating-point load). The items fetched using PFLD instruction are bypassed from cache to avoid thrashing or displacing the existing useful data in cache. The additional latency of off-chip access is avoided by virtue of pipelining, such that the load is issued several cycles before those data are actually required. The result is stored in a FIFO (first-in first-out) buffer, which is used by the processor. Due to this, use of PFLD provides better performance than making data noncacheable using page-table entries. Use of PFLD allows mixing load commands to the cache and to the external memory. Also, coherence for PFLD is maintained by first checking the cache since a normal load instruction may have brought the requested data in the cache already.

Similarly, x86 ISA provides bypass instructions for reads/writes with no temporal locality. For example, using MOVNTI instruction, a write can be sent directly to memory through a write-combining buffer, bypassing the cache [18]. For GPUs of compute capability 2.0 or higher, PTX (parallel thread execution) ISA provides load/store instructions to support bypassing [19]. For example, `ld.cg` specifies that a load bypasses L1 cache and is cached only in L2 cache and below. This request also evicts any existing L1 cache block with the same address [19].

2.3. Promises of Cache Bypassing

Cache bypassing is a promising approach for several reasons.

2.3.1. Performance and Energy Benefits

As discussed earlier, caching data of poor-locality applications can harm performance and this effect becomes increasingly pronounced with non-uniform cache access (NUCA) designs where the latency to the farthest bank greatly exceeds the average access latency. Similarly, in deep cache hierarchies, blocks which are frequently reused in higher-level caches may not show a high reuse in lower-level caches due to filtering by higher-level caches [20] and hence, bypassing these blocks can improve performance.

With an already good replacement policy (e.g., Belady's OPT policy), a bypass policy does not improve LLC hit rate, although bypassing may still save on-die interconnect bandwidth [21]. However, with inferior replacement policies (e.g., random policy), bypassing can provide large improvement in hit rate [21].

CBTs can also be helpful for saving cache energy. For example, cache reconfiguration techniques work by turning off portions of cache for applications/phases with low data locality [22–24]. Since bypassing reduces the data traffic to cache, it can allow cache reconfiguration techniques to more aggressively turn-off cache for saving even larger amount of energy.

2.3.2. Benefits in NVM and DRAM Caches

High leakage and low density of SRAM has motivated researchers to explore its alternatives for designing on-chip caches, such as NVMs and die-stacked DRAM, which provide high density and consume lower leakage power than SRAM [25,26]. However, these technologies also have some limitations, for example, NVMs have low write-endurance and high write latency/energy [27,28] and hence, harmful impact of low-reuse data can be more severe in NVM caches than in SRAM caches [29]. In addition, at small feature size (e.g., smaller than 32 nm), STT-RAM suffers from read-disturbance error where a read operation can disturb the cell being read. CBTs reduce read/write traffic to NVM caches since bypassed blocks need not be accessed from cache, and thus, CBTs can address the above mentioned issues in NVM caches.

Similarly, gigabyte size DRAM caches use large block size (e.g., 2 KB) to reduce metadata overhead [30] which increases cache pollution due to low-reuse data and also wastes the bandwidth. CBTs can allow placing only high-reuse data in the DRAM caches to avoid destructive interference from low-reuse data. Clearly, CBTs can provide additional benefits for these emerging technologies.

2.3.3. Benefits in GPUs

Typical graphics applications have little locality and caching them can lead to severe thrashing. Further, the design philosophy in GPUs is to dedicate large fraction of chip resources for computation, which leaves little resources for caches. Hence, GPUs share small caches between large number of threads, for example, NVIDIA Fermi and Kepler have (up to) 48 KB L1 cache shared between 1536 and 2048 threads/core (respectively), for a per-thread capacity of 32 B and 24 B, respectively [3,4]. Similarly, the per-thread L1 cache capacity for NVIDIA Maxwell is 16 B (24 KB for 2048 threads/core) and for AMD Radeon-7, it is only 6.4 B (16 KB for 2560 threads/core) [5,31,32]. By comparison, per-thread L1 capacity in CPUs is in few KBs, for example, Intel's Broadwell processor has 32 KB L1 cache for 2 threads per core [33].

Due to the limited cache capacity, equally caching data from all threads can lead to cache pollution and hence, interference in L1D cache and in L1D-L2 interconnect generally causes major bottlenecks in GPU performance [34]. CBTs are vital for addressing these challenges since they can allow achieving performance of a larger cache (e.g., a double size cache [35,36]) without incurring the associated area/power overheads of a larger cache.

2.4. Challenges in Using Cache Bypassing

Despite its promises, cache bypassing also present several challenges.

2.4.1. Implementation Overhead

Since performing naive bypassing for all data structures or/and for entire execution can degrade performance [21,32,37], accurate identification of bypassing candidates is required for reaping the benefits of bypassing. This necessitates predicting future program behavior using either static profiling or dynamic profiling [38]; however, each of these have their limitations. Static profiling techniques use compiler to identify memory access pattern. However, lack of runtime information and variation in input datasets limits the effectiveness of these approaches.

Dynamic profiling techniques infer application characteristics based on runtime behavior. Although they can account for input variation, they incur large latency/storage overhead due to the need of maintaining large predictor tables (e.g., [39]) or per-block counters (e.g., [40]) that need to be accessed/updated frequently. Also, the techniques which make predictions based on PC (e.g., [23]) require this information to be sent to LLC with every access, which requires special circuitry.

2.4.2. Memory Bandwidth and Performance Overhead

Since bypassed requests go directly to the next level cache or memory, they may saturate the network bandwidth and create severe congestion. This increases cache/memory access latency sharply which leads to memory stalls. Further, compute resources remain un-utilized and dissipate power without performing useful work [11]. Further, in CPU-GPU systems, blindly bypassing all GPU requests may increase the cache hit rate of CPU, however, it can degrade the performance of both CPU and GPU [41]. This is because the huge number of bypassed GPU requests cause main memory contention and due to their high row-buffer locality, they may be scheduled before CPU requests.

2.4.3. Challenges in GPUs

Use of cache bypassing in GPUs often requires co-management of thread-scheduling policies (refer to Section 7), such as thread-throttling. However, reducing the degree of multithreading for improving cache utilization may lead to under-utilization of computational and off-chip memory resources [42]. Also, bypassing and thread throttling may have unforeseen impacts on algorithm and avoiding this may demand reformulation of algorithm. This requires significant programmer efforts.

2.4.4. Challenges in Inclusive Caches

A multi-level cache hierarchy is said to be *inclusive* if the contents of all higher level caches are subset of the LLC and is termed as *non-inclusive* when the higher level caches may not be subsets of LLC. An *exclusive* hierarchy guarantees that no two cache levels have common cache contents. While processors such as AMD Opteron use non-inclusive LLC, other processors such as Intel Core i7 use inclusive LLC.

Bypassing violates the assumption of inclusion and hence, using bypassing with inclusive cache hierarchies requires special provisions (refer to Section 5 for more details). For example, the bypassed block can be inserted into the LRU position [39] which ensures that the block is evicted on the next miss to cache set. This, however, still replaces one potentially useful block from the cache, which can be especially harmful for low-associativity caches. Also, in a corner case, where many consecutive accesses are mapped to a cache set, bypassed blocks compete for the LRU position. This reduces their lifetime and causes victimization of same blocks in upper level caches, degrading the performance of inclusive LLCs [43]. Other works use additional storage to track the tags of bypassed blocks for satisfying inclusion property [44]. The limitations of this approach are the additional design complexity and latency/energy overheads.

The techniques presented in next sections aim to address these challenges.

3. Key Ideas and Classification of CBTs

In this section, we first discuss some salient ideas used by different CBTs and then classify the CBTs on key parameters to underscore their features.

3.1. Main Ideas of CBTs

To get insights, we now discuss the essential ideas of architecture-level cache management which are used by various CBTs. Note that these ideas are not mutually exclusive.

1. **Criterion for making bypass decisions:**

    - To perform bypassing, different CBTs make decisions based on *reuse count* [7,9,12,20,21,23,35–37,40,45–50] or *reuse distance* [11,14,43,45,51–57] which are both related (refer to Sections 4.1 and 4.2).
    - Some other CBTs take decision based on miss-rate [10,38,48,58–62], while a few others make decision based on NoC congestion [11], cache port obstruction [29], ratio of read/write *energy* of the cache [12] or stacked-DRAM bandwidth-utilization [37] (refer to Section 4.3).
    - Some techniques bypass thread-private data from shared cache [63], while others bypass physical pages that are shared by multiple page tables [48].
    - Some techniques keep counters for every data-block and to make bypassing decision or getting feedback, they compare counters of incoming and existing data to see which one is accessed first or more frequently [39,43,54,56,64,65]. Thus, these and a few other techniques [35,66] use a *learning* approach where the value of its parameters (e.g., threshold) are continuously updated based on correctness of a bypassing decision.
    - Some techniques predict reuse behavior of a line based on its behavior in its previous generation (*i.e.*, last residency in cache) [20,23,40,49,66]. Other techniques infer reuse behavior of a line from that of another line adjacent to it in memory address space, since adjacent lines show similar access properties [54]. Similarly, the reuse pattern of a block in one cache (e.g., L2) can guide bypassing decisions for this block in another cache (e.g., L3) [21,45].

2. **Classifying accesses/warps for guiding bypassing:** Some works classify the accesses, misses or warps into different categories to selectively bypass certain categories. Ahn *et al.* [13] classify the writes into dead-on-arrival fills, dead-value fills and closing writes (refer to Section 5.1). Wang *et al.* [53] classify the LLC write accesses into core-write (write to LLC through a higher-level write-through cache or eviction of dirty data from a higher-level writeback cache), write due to prefetch-miss and due to demand miss (refer to Section 6.1). Similarly, Chaudhuri *et al.* [45] classify cache blocks based on the number of reuses seen by it and its state at the time of eviction from L2, *etc.* In the work of Wang *et al.* [67], the LLC blocks which are frequently written back to memory in an access interval are termed as frequent writeback blocks and remaining blocks (either dirty or clean) are termed as infrequent writeback blocks.

    Collins *et al.* [10] classify the misses into conflict and capacity (which includes compulsory) misses. Tyson *et al.* [61] classify the misses based on whether they fetch useful or dead-on-arrival data. For GPUs, Wang *et al.* [68] classify the warps into locality warps and thrashing warps depending on the reuse pattern shown by them. Liang *et al.* [69] classify access patterns as partial or full sharing (few or all threads share the same data, respectively) and streaming pattern.

3. **Adaptive bypassing:** Since bypassing all requests degrades performance, some techniques perform bypass only when no invalid block is available [21,45,64] or a no-reuse block is available [14,20].
4. **Cache hierarchy organization:** Some CBTs work by reorganizing the cache and/ore cache hierarchy (refer to Section 4.5). Malkowski *et al.* [70] split L1 cache into a regular and a bypass cache. B. Wang *et al.* [68] assume logical division of a cache into a locality region and a thrashing region for storing data with different characteristics and Z. Wang *et al.* [67] logically divide each cache set into a frequent writeback and an infrequent writeback list. Das *et al.* [55] divide a large wire-delay-dominated cache into multiple sublevels based on distance of cache banks from processor, e.g., three sublevels may consist of the nearest 4, next 4 and furthest 8 banks from the processor, respectively. Gonzalez *et al.* [71] divide the data cache into a spatial cache and a temporal cache, which exploit spatial and temporal locality, respectively.

   Xu and Li [46] study page-mapping in systems with a main cache (8 KB) and a mini cache (512 B), where a page can be mapped to either of them or bypassed. Etsion and Feitelson [36] propose replacing a 32 K 4-way cache with a 16 KB direct-mapped cache (for storing frequently reused data) and a 2 K filter (for storing transient data).

   Wu *et al.* [57] present a CBT for micro-cache in EPIC processor. Xu and Li [46] present a technique for bypassing data from main cache or mini cache or both caches. Wang *et al.* [53] evaluate their CBT for an SRAM-NVM hybrid cache.
5. **Use of bypass buffers:** Some works use buffer/table to store both tags and data [9,10,36,49,65,66,72] or only tags [43,56] of the bypassed blocks. Access to the cache is avoided for the blocks found in these buffers and with effective bypassing algorithms, the size of these buffers are expected to be small [43,49]. The bypassed blocks stored in the buffer may be moved to the main cache only if they show temporal reuse [9,49,73,74]. Chou *et al.* [37] buffer tags of recently accessed adjacent DRAM cache lines. On a miss to last level SRAM cache, the request is first searched in this buffer and a hit result avoids the need of miss probe in DRAM cache.
6. **Granularity:** Most techniques make prediction at the granularity of a *block* of size 64 B or 128 B. Stacked-DRAM cache designs may use 64 B block size [37] to reduce cache pollution or 4 KB block size [30,48] to reduce metadata overhead. By comparison, Alves *et al.* [23] predict when a *sub-block* (8 B) is dead, while Johnson and Hwu [65] make prediction at the level of a *macroblock* (1 KB) which consists of multiple adjacent blocks. Lee *et al.* [48] also discuss bypassing at *superpage* (2 MB to 1 GB) level (refer to Section 6.2). Khairy *et al.* [58] disable the entire cache and thus, all data bypass the cache (refer to Section 4.7). Use of larger granularity allows lowering the metadata overhead at the cost of reducing the accuracy of information collected about reuse pattern.
7. **Use of compiler:** Many CBTs use a compiler for their functioning [8,38,46,51,52,57,63,69], while most other CBTs work based on runtime information only (refer to Section 4.6). The compiler can identify thread-sharing behavior [69], communication pattern [52,63], reuse count [8,38,46] and reuse distance [51,57]. This information can be used by compiler itself (e.g., for performing intelligent instruction scheduling [57]) or by hardware for making bypassing decisions.
8. **Co-management policies:** In addition to bypassing, the information about cache accesses or dead blocks has been used for other optimizations such as power-gating [23,75], prefetching [10,50] and intelligent replacement policy decisions [14,23,50,76]. For example, data can be prefetched into dead blocks and while replacing, first preference can be given to dead blocks. The energy overhead of CBTs (e.g., due to predictors) can be offset by using dynamic voltage/frequency scaling (DVFS) technique [70].
9. **Solution algorithm:** Liang *et al.* [69] present an integer linear programming (ILP) based and a greedy algorithm for solving L2 traffic reduction problem. Xu and Li [46] present a greedy algorithm to solve page-to-cache mapping problem.
10. **Other features:** While most CBTs work with any cache replacement policy, some CBTs assume specific replacement policy (e.g., LRU replacement policy [8]).

Several strategies have been used for reducing implementation overhead of CBTs.

11. **Probabilistic bypassing:** To avoid the overhead of maintaining full metadata, many CBTs use probabilistic bypassing approach [36,37,43,56] (refer to Section 4.4).
12. **Set sampling:** Several key characteristics (e.g., miss rate) of a set associative cache can be estimated by evaluating only a few of its sets. This strategy, known as set sampling, has been used for reducing the overhead of cache profiling [13,21,37,43,45,51,56,67,76,77]. Also, it has been shown that keeping only a few bits of tags is sufficient for achieving reasonable accuracy [10,76] (refer to Section 4.8).
13. **Predictor organization:** Many CBTs use predictors (e.g., dead block predictors) for storing metadata and making bypassing decisions. The predictors indexed by PC of memory instructions incur less overhead than those indexed by addresses [20,23,35,39,53,61,70,71].

3.2. A Classification of CBTs

To emphasize the differences and similarities of the CBTs, Table 1 classifies them based on key parameters. We first categorize the CBTs based on their objectives and from this, it is clear that CBTs have been used for multiple optimizations such as performance, energy, *etc.* Generally, CBTs aimed at improving performance also save energy, for example, by reducing miss-rate and main memory accesses. The key difference between techniques to improve performance and energy efficiency are that their algorithms may be guided by a performance or energy metric. For example, an energy saving CBT may occasionally bypass a block for saving energy even at the cost of performance loss (e.g., due to higher miss-rate). Similarly, a bypassed block may later need to be accessed directly from main memory which may incur larger energy than the case where the block was in cache already. A CBT designed to improve performance may still bypass this block if it improves performance, whereas a technique designed for saving energy may not bypass this block.

As for CBTs designed to ensure timing predictability, they work on the idea that the access latency of a bypassed block is equal to the memory access latency. Also, bypassing some blocks can allow ensuring that other blocks always remain in cache and the access latency for these blocks is equal to cache access latency. Since caches are major sources of execution time variability [78], CBTs can be useful in alleviating the impact of such timing unpredictability. Thus, these CBTs may primarily focus on removing uncertainty in hit/miss prediction instead of improving performance.

Table 1 also classifies the works based on the level in cache hierarchy where a CBT is used and the nature of cache hierarchy. First-level and last-level caches show different properties [79,80], for example, filtering by first-level cache reduces the locality seen by last level cache and hence, dead-block prediction schemes, the length of access intervals, *etc.* are different in those caches (also see Section 2.3.1).

Table 1. A classification of research works.

Classification	References
Study/optimization objective	
Performance	[7,9–14,20,21,29,32,34–41,43–55,57–64,66–70,74–77,81–84]
Energy	[12,13,23,35,36,44,46,47,52,53,55,60,67,70,75,77,83]
Predictability	[7,84,85]
Level in cache hierarchy	
First-level cache	[7,10,11,23,32,34–36,38,46,47,52,58,59,61,62,65,66,68,69,74,75,81,83]
Mid/last-level cache	[12–14,20,21,23,32,37,39–41,43–45,48–56,58,60,63–65,67,70,75–77,82–84]
Micro-cache	[57]

4. Working Strategies of CBTs for CPUs

In this and the coming sections, we discuss many CBTs by roughly organizing them into several groups. Although many of these techniques fall into multiple groups, we discuss them in a single group only.

4.1. CBTs Based on Reuse-Count

Kharbutli and Solihin [20] present cache replacement and bypassing schemes which utilize counter-based dead block predictors. They note that both live times and access intervals (refer to Section 2.1) are predictable across generations. Based on it, they design two predictors, a live-time predictor and an access interval predictor. The former predictor counts the references to a cache block during the time it stays in the cache continuously and the latter predictor counts the references to a set between two successive references to a given cache block. When these counters reach a threshold, the block is considered dead and becomes a candidate for replacement. The threshold is chosen as the largest of all live times (or access intervals) in current and previous generations. Thus, the threshold of every block is potentially different and is learnt dynamically. These generations are identified by the program counter of the instruction that misses on the block. A predictor table stores this information for blocks that are not in cache and that are fetched again in cache. They further note that bursty accesses to blocks are typically filtered by L1 cache and hence, many blocks in L2 cache are never-reused. Their predictors identify such blocks by seeing whether their thresholds were zero in previous two generations, implying that they were not reused while residing in L2 cache. If the target set has no dead block, then the predicted no-reuse block is bypassed from the cache, otherwise, it is allocated in the L2 cache. They show that their CBT improves performance significantly.

Xiang *et al.* [49] note that CBTs generally bypass never-reused lines, however, such lines do not occur frequently in many applications and this limits the effectiveness of those CBTs. Instead of bypassing *only* never reused lines, they propose bypassing less reused lines (LRLs). Their technique predicts the reuse frequency of a miss line based on reuse frequency observed in the previous occurrence of that miss. Then, LRLs are bypassed and kept in a separate buffer. The short lifespan of LRLs enables their technique to use a small buffer and quickly retire majority of LRLs. The lines which cannot be retired are inserted back to L2, or are discarded based on a per-application retirement threshold. Thus, bypassing LRLs enables L2 cache to effectively serve applications with large working set sizes. They show that their technique reduces cache miss-rate and improves performance.

Kharbutli *et al.* [40] present a CBT that makes bypass decisions on a cache miss based on previous access/bypass pattern of the blocks. With each cache line, a ”USED” bit is employed that is set to zero when the block is allocated in cache and set to one on a cache hit. Thus, at the time of replacement, USED bit shows whether an access was made to the block during its residency in cache. Their technique also uses a history table to record the access/bypass history of every block in their previous generations using 2-bit saturating counter per block. On a cache miss on block P, its counter in the table is read. If the counter value is smaller than 3, P is not expected to be reused while residing in cache and hence, P is bypassed and its counter is incremented. However, if counter value is 3, P is inserted into cache. A victim block is found using cache replacement policy and the counter value of this victim block in the table is updated depending on whether it was accessed while residing in cache (counter set to 3) or not (counter set to 0). Thus, a block is allocated in the cache, if it was accessed in the cache during its residency in the last generation or if has been bypassed 3 times. To adapt to changing application behavior, all counter values are periodically set to zero. Their technique provides speedup by reducing the miss rate.

4.2. CBTs Based on Reuse-Distance

Duong *et al.* [14] propose a reuse-distance based technique for optimizing replacement and performing bypass. Their technique aims to keep a line in the cache only until expected reuse happens and cache pollution is avoided. This reuse distance is termed 'protecting distance' (PD) and it balances timely eviction with maximal reuse. For LRU policy, PD equals cache associativity, but their technique can also provide PD larger than the associativity. When a line is inserted in cache or promoted, its distance is set to be PD. On each access to the set, this value for each line in the set is decreased by 1 and when this value for a line reaches 0, it becomes unprotected and hence, a candidate for replacement (victim). Since protected lines have higher likelihood of reuse than missed lines, on a miss fetch, if no unprotected line is found in the set, the fetched block bypasses the cache and is allocated in higher level cache. Thus, both replacement scheme and bypass scheme together protect the cache lines. PD is periodically recomputed based on dynamic reuse history such that the hit rate is maximized. By reducing the miss rate, their technique improves performance significantly.

Das *et al.* [55] note that wire energy contributes significantly to the total energy consumption of large LLCs. They propose a technique which reduces this overhead by controlling data placement and movement in wire-energy-dominated caches. They partition the cache into multiple cache sublevels of dissimilar sizes. Each sublevel is a group of ways with similar access energy, e.g., sublevel 0, 1 and 2 may consist of the nearest 4, next 4 and furthest 8 banks from the processor, respectively. Based on the recent reuse distance distribution of a line, a suitable insertion and movement scheme is used for it to save energy. For example, if a line is expected to receive only one hit after insertion in the cache, moving it to closer cache locations incurs larger energy than accessing them once from a farther location. Similarly, if some lines show reuse within first 4 ways, but no further reuse until cache capacity is exceeded, then these lines can be inserted in the 4 nearest ways and when they are evicted from these ways, they can be evicted from the cache (instead of being placed in the remaining 12 ways). In a similar vein, the lines which are expected to show no reuse are bypassed from the cache. They use their technique for both L2 and L3 caches and achieve large energy saving.

Yu *et al.* [54] note that cache lines which are adjacent in memory address space show similar access properties, for example re-reference intervals (RRI) [86], reuse distance, *etc.* For example, if P and Q are consecutive, and P is a dead block, then Q is also expected to be dead. Based on this, they use a table for recording the RRI of cache blocks. When a cache block is to be inserted in an LLC set, the table is accessed for obtaining the expected RRI from that of an adjacent block. This is compared with a threshold (maximum RRI) and if the RRI of the new block is greater than the threshold, it is considered dead and bypassed from the cache. Otherwise, the cache block with the largest RRI from that LLC set is considered and its RRI is compared with that of incoming block. If the RRI of incoming block is greater, the incoming block is bypassed, otherwise, it is inserted in the cache. The RRI of an entry in the table is decreased on a hit in LLC and is increased when a corresponding victim block is replaced. They show that their technique reduces cache misses and improves performance.

Feng *et al.* [51] present a CBT for avoiding thrashing in LLC. They note that in case of cache thrashing, the forward reuse distance of most accesses are greater than cache associativity. Their technique inserts additional phantom blocks in the regular LRU stack, which gives the illusion of higher associativity. A phantom block does not store tag or data, but otherwise works in the same way as the normal block in LRU stack. When the replacement candidate chosen is a phantom block, the cache is bypassed and data are directly sent to the processor. To find the suitable number of added phantom blocks, they use different number of phantom blocks (e.g., 0, 16, 48 *etc.*) with a few sampled sets. Periodically, the phantom-block count which leads to fewest cache misses is selected for the whole cache and this helps in adapting to different applications/phases and keeping high-locality data in cache while bypassing dead data. They show that their technique improves performance by reducing cache misses. They also study use of compiler to provide hints. For this, the application is executed with training data set and for each main loop in the application, optimal phantom-block count is obtained by experimenting with different values. These hints are inserted in the application before each main loop and are used during application execution. They show that by using these hints, further reduction in cache misses can be obtained for benchmarks with high miss-rate.

Li *et al.* [39] note that an optimal bypass policy (that bypasses a fetched block if its reuse distance equals or exceeds that of the victim chosen by replacement policy) achieves performance close to Belady's OPT plus bypass policy (that first allocates blocks with smallest reuse distance in cache and then bypasses remaining blocks). Since optimal bypass policy cannot be practically implemented, they present a CBT that makes bypass decisions by emulating the operation of optimal bypass policy. Their technique uses a 'replacement history table' that tracks recent incoming-victim block tuples. Then, every incoming-victim block tuple are compared with this table to ascertain the decision of optimal bypass on a recorded tuple. For example, if incoming block is accessed before victim block, no bypassing should be done, but if victim block is accessed first or none of them are accessed in future, bypassing should be performed. To record these learning results, PC-indexed 'bypass decision counters' are used, which are decremented or incremented, depending on whether replacement or bypassing (respectively) is performed for an incoming block. On a future miss, this counter is consulted for an incoming block. A non-negative value of the counter signifies that optimal bypass policy would perform larger number of bypasses for this block in a recent execution window, and hence, their technique decides to perform bypassing. Conversely, a negative value leads to insertion of the incoming block in the cache with replacement of a victim block. They show that their technique provides speedup by reducing the miss rate.

4.3. CBTs Based on Cache Miss Behavior

Collins *et al.* [10] present a CBT which is based on a miss-classification scheme. They use a table which stores the tag of most recently evicted cache block from each set. If the tag of the next miss in a set is same as that stored in the table, it is marked as conflict miss, since it might have been hit with slightly-higher associativity. Otherwise, the miss is a capacity miss (which also includes compulsory miss). Even storing few (e.g., lower eight) bits of the tag provides reasonably high classification accuracy, although the accuracy increases with increasing tag bits that are stored. They use this information for multiple optimizations, such as cache prefetching, victim cache and cache bypassing, *etc.* For bypassing, they note that accesses leading to capacity misses show short and temporary bursts of activity. Based on it, their technique bypasses any capacity miss and places it in a bypass buffer. By reducing miss rate, their technique provides large performance improvement.

Tyson *et al.* [61] note that a small fraction of load instructions are responsible for the majority of data cache misses. They present a CBT which measures miss rates of individual load/store instructions. The data references generated by the instructions, which lead to highest miss rate, are bypassed from the cache. They also propose another version of this technique which records the instruction address which brought a line into the cache. Using this, a distinction is made between those misses that fetch useful (*i.e.*, later reused) data into cache and those misses that fetch dead-on-arrival data. Based on this, only the latter category of data references are bypassed from the cache. They show that their technique improves hit rate and bandwidth utilization.

4.4. Probabilistic CBTs

Etsion *et al.* [36] note that of the blocks comprising program working set, a few blocks are accessed very frequently and for the longest duration of time, while remaining blocks are accessed infrequently and in a bursty manner. Hence, set-associative caches serve majority of references from the MRU position and thus, they effectively work as direct-mapped caches, while expending energy and latency of set-associative caches. Based on this, they propose a technique which serves hot blocks efficiently and bypasses transient (cold) blocks. They propose two approaches for identifying hot blocks. In threshold based approach, a block that is accessed more than a threshold number of times (e.g., 16) is considered 'hot' and in probabilistic approach, a hot block is identified by randomly selecting memory references by running a Bernoulli trial on every memory access since the long-residency blocks are most likely to get selected. Of these, probabilistic approach does not require any state information and provides comparable accuracy as the threshold-based predictor. In place of a set-associative cache, they propose using a direct-mapped cache for serving hot blocks and a small fully-associative filter to serve the transient blocks. To reduce the overhead of the filter, they use a buffer that caches recent lookups. They show that with a 16 KB direct-mapped L1 cache and a 2 K filter, their technique outperforms a 32 K 4-way cache and also provides energy savings.

Gao *et al.* [56] present a CBT which performs random bypassing of cache lines based on a probability. This probability is increased or reduced depending on the effectiveness of bypassing, which is recorded based on whether a bypassed line is referenced before the replacement victim. For this, an additional tag and a competitor pointer is used with each set. On a line bypass, this tag holds the tag of the bypassed line and competitor pointer records the replacement victim which would have been evicted without bypassing. Bypassing is considered effective or ineffective, depending on whether the competitor or bypassed tag (respectively) is accessed before the another. When a cache fill happens at the location pointed by competitor pointer, both the competitor pointer and additional tag are invalidated. To evaluate the effect of bypassing when 'no-bypassing' decision is chosen, some recently allocated lines are randomly selected for 'virtual bypassing'. Also, the additional tag holds the tag of replacement victim and competitor pointer holds the position of incoming block. If access to replacement victim happens before the incoming block, bypassing is deemed effective. Using set sampling, two dueling policies are evaluated and the winner policy is finally used for the cache.

4.5. CBTs Involving Cache Hierarchy Reorganization or Bypass Buffer

Malkowski *et al.* [70] present a CBT which reduces memory latencies by bypassing L2 cache for load requests which are expected to miss. They divide the 32 KB L1D cache into a 16 KB regular and a 16 KB bypass portion. The regular L1D cache uses line size of 32 B, while bypass L1D cache uses line size of 128 KB, which is the line size of L2 and data-size transferred from memory in each request. A load-miss predictor (LMP) is also used, which is indexed by PC of the load instruction. The data request of a load instruction accesses both regular and bypass caches. If both show a miss, LMP is accessed, which predicts either a hit or a miss. LMP tracks only those loads that miss in both bypass and regular cache and a newly encountered load is always predicted hit. A predicted hit progresses along the regular cache hierarchy. Depending on whether data are found on a predicted hit, a correct or incorrect prediction is noted in the LMP. If data are found at any cache level, they are not fetched into bypass cache. On a predicted miss, a request is sent to L2 cache by regular L1 and in parallel, an early load request is sent to main memory by bypass cache. If data are found in L2 cache, the ongoing memory request is cancelled, data are stored in regular cache, and the load is considered a correct prediction. If data are not found in L2 cache, main memory provides the data and since the memory access was issued early, its latency is partially hidden, which improves performance. The prediction is flagged as correct and the data are stored in bypass cache. A store instruction proceeds along the path used by load instruction to that address. The L2 cache acts as victim cache for the bypass cache. They show that their technique provides speedup but increases power consumption. By using DVFS along with their technique, both performance and power efficiency can be improved.

John and Subramanian [9] present a CBT which uses an assist structure called annex cache to store blocks which are bypassed from main cache. In their design, all entries to main cache come from annex cache, except for filling at cold start. A block in annex cache is exchanged with a conflicting block in main cache, only when the former has seen two accesses after the latter was accessed. Thus, low-reuse items are bypassed from main cache and only those blocks which have shown locality are stored in main cache. The main difference between annex cache and victim cache is that the annex cache can be directly accessed by the processor. They show that their technique outperforms conventional cache, and performs comparably to victim caches.

Jalminger and Stenström [66] present a CBT which makes bypassing decisions based on reuse behavior of a block in previous generations. Since the reuse history pattern of a block may span over multiple lifetimes in cache, they use a predictor to estimate future reuse behavior by finding repetitive patterns in blocks' reuse history. A block with no predicted reuse is stored in a bypass buffer while remaining blocks are stored in the cache. For both allocated and bypassed blocks, their actual reuse pattern is used to find whether the prediction was correct and to update the predictor. The predictor is organized as a two-level design such that one table tracks reuse history of each block and using this as an index, a second table is accessed whose output is used for predicting future reuse. They show that even with a single-entry bypass buffer, their technique reduces the L1 cache miss rate significantly.

4.6. CBTs Involving Software/Compiler Level Management

Wu *et al.* [57] present a bypassing technique for micro-cache (μcache) in EPIC processor, such as Itanium. The μcache is a small cache between core and L1 cache and its size may be 64 B with a 2 KB L1 cache. In statically scheduled EPIC processors, compiler is aware of the distance between a load and it reuse. Based on it, their technique uses compiler analysis and profiling to find loads which should bypass μcache. Assuming L1 latency as T_1 cycles, μcache should only store data that will be required before next T_1 cycles, otherwise, the load should directly access L1. Thus, an effective bypassing technique can allow μcache to store only critical data that are immediately reused. In their technique, the compiler performs program dependency analysis before instruction scheduling for identifying loads which are reused T_1 (or more) cycles after they are issued. The scheduler tries to schedule these loads with T_1 cycle latency, since otherwise, they would be scheduled such that their results are required in T_μ cycles (the latency of accessing μcache) due to the fact that by default, the scheduler assumes the load to hit in μcache. At the completion of instruction scheduling, the loads with no reuse in next T_1 cycles are marked to bypass the μcache. Finally, cache profiling is done to identify additional loads for bypassing. If a load misses μcache and the loaded data are not reused, the load is marked for bypassing μcache, which avoids the overhead of accessing μcache. They show that their technique reduces μcache miss rate and improves program performance.

Chi *et al.* [8] present a CBT which makes bypassing decisions based on cost and benefit of allocating a data-item in the cache. The cost of caching is the time to access memory for fetching data and is doubled if caching a block replaces a dirty block. The benefit from caching is the product of number of accesses to data during its cache residency and the difference between access time of memory and cache. In their technique, the compiler builds the control flow graph of the program. For each control flow path, initially all the references are assumed to be cached. Then, for each reference, the cost and benefit from caching the associated line are evaluated assuming LRU replacement policy. At the end, all references for which the cost of caching exceeds its benefit are marked for bypassing. They show that their technique provides large application speedup.

Park *et al.* [52] note that several memory access patterns such as streaming and producer-consumer communication may lead to inefficient use of caches. They propose two instructions for controlling the level where data structures are stored. *Unallocating load* signifies a read to a data-item that should not be inserted in caches smaller than reuse distance of the data. *Pushing store* is a write which stores data in a specific cache-level on a specific core (e.g., a consumer thread's core). For example, in producer-consumer communication, the data written by one thread are read by other threads. On using an invalidation-based coherence protocol, for each cache block, the producer invalidates its current sharers, fetches the block in its cache(s) and completes the write. After this, the consumer searches the producer and copies the block from the cache of the producer. However, any block fetched in any cache of the producer is not read by it, unless both producer and consumer share a cache. Using pushing store, the block can be directly pushed to the cache of the consumer, instead of invalidating it. On a subsequent read operation to the shared data by the consumer, it's local cache will already have the updated block. Thus, these instructions decrease coherence traffic and coherence misses for the consumer. Similarly, to improve cache efficiency with streaming pattern, the reuse distance can be provided with their proposed instructions for any variable, based on which the variable can be bypassed from any cache level. Thus, through these instructions, application knowledge can be conveyed to the hardware and they are useful primarily when the working set exceeds a certain cache level. Their approach maintains program correctness and improves performance and energy efficiency.

4.7. Use of Different Bypassing Granularities in CBTs

Alves *et al.* [23] note that a large fraction of cache subblocks (e.g., 8 B subblock in a 64 B block) brought in the cache are never reused. Also, most of the remaining subblocks are only used a few times (e.g., 2 or 3 times). They present a technique for predicting the reference pattern of cache sub-blocks, including *which* subblocks will be accessed and *how many times* they will be accessed. They store past usage pattern at subblock level in a table. This table is indexed by PC of load/store instruction which led to cache miss and the cache block offset of the miss address. Use of PC with offset provides high coverage even with reasonable-size table because a memory instruction sequence generally references the same fields (subblock) of a record (block). Based on the information from this table, on a cache miss, only the subblocks that are expected to be useful are fetched. Also, when a subblock has been touched for expected number of times, it is turned-off. They also optimize the replacement policy to first evict those blocks for which all subblocks have become dead. This helps in offsetting any cache misses caused due to mispredictions in their scheme. They show that their technique saves both leakage and dynamic energy.

Johnson and Hwu [65] present a CBT that performs cache management based on memory usage pattern of the application. Since tracking the access frequency of each cache block incurs prohibitive overheads, they combine adjacent blocks into larger-sized 'macroblocks', although the limitation of using large granularity is that their technique cannot distinguish whether a single block was accessed *N* times or *N* blocks saw one access each. The size of macroblocks is chosen such that the cache blocks in a macroblock see relatively uniform access frequency, and the total number of macroblocks in the accessed portion of memory still remains small. For example, by experimenting with 256 B, 1 KB, 4 KB and 16 KB, they find that 1 KB macroblock size provides a good balance. They also use a memory access table (MAT), which uses one counter for each macroblock. MAT is accessed in parallel to data cache and its counters are incremented on every access to the corresponding macroblocks. On a cache miss, the MAT counter of victim candidate is decremented and then compared with the MAT counter of the fetched block. If the former is larger, the fetched block is bypassed, otherwise, the victim block is replaced as done in normal caches. Decreasing the counter of victim block ensures that after a change in the program phase, new blocks can replace existing blocks which have now become useless. In case where data show temporal locality but low access frequency, many useful blocks may be bypassed from the cache. To avoid this, they place the bypassed blocks in a small buffer which allows accessing them with low latency and exploiting temporal locality. To exploit spatial locality, they provision dynamically choosing the size of fetched data on a cache bypass to balance bus traffic reduction and miss-rate reduction. They show that their technique provides large application speedup.

4.8. Strategies for Reducing Overhead of CBTs

Khan *et al.* [76] note that consistency of memory access patterns across sets allows sampling references to a fraction of sets for making accurate prediction compared to the techniques which track every reference [22]. Further, a majority of temporal locality from LLC access stream is filtered by the mid-level cache which reduces the effectiveness of trace-based predictors in LLC. Based on these, they propose a sampling dead block predictor which samples PCs to find dead blocks. It uses a sampler with partial tag array. For example, with a sampling ratio of 1/64 and 2048 sets in LLC, the sampler has only 32 sets. Only lower 15 bits of tags are maintained since exact matching is not required. Use of sampling reduces area and power requirements. Further, sampler decouples the prediction scheme and LLC design and hence, while LLC may use a low-cost replacement policy, the predictor can use LRU policy since by virtue of being deterministic, LRU allows easier learning and is not affected by random evictions. Also, the sampler can have different associativity than LLC, e.g., using a 12-way sampler with a 16-way LLC provides better accuracy than a 16-way sampler as it evicts the dead blocks more quickly. To predict if a block is dead, their technique uses only the PC of last-access instruction, instead of the trace of instructions referring to that block. Although the sampler still stores the trace metadata, the small size of sample tag array keeps the area and timing overhead small. To reduce the conflicts in prediction table, they use a skewed organization whereby three tables are used instead of one, and each table is indexed by a different hash function. If their DBP predicts a block to be dead-on-arrival, it is bypassed from LLC. Also, their replacement policy preferentially evicts dead blocks. They show that their technique reduces cache misses and improves performance.

5. CBTs for Different Hierarchies and Evaluation Using Different Platforms

Due to the inclusion/non-inclusion requirement, the nature of cache hierarchy impacts the design and operation of the bypassing technique (also see Section 2.4.4). For this reason, in Table 2 we classify the CBTs based on cache hierarchy for which they are proposed.

Table 2 also classifies the works based on their evaluation platform. This is important since real systems allow accurate evaluation and full design-space exploration whereas simulators offer flexibility to evaluate variety of techniques which may even be infeasible to implement on real hardware. Analytical performance models show limit gains from CBTs, independent of a particular application or hardware. Clearly, all three approaches are indispensable for obtaining important insights about CBTs.

Table 2. A classification based on cache hierarchy and evaluation platform.

Classification	References
Nature of cache hierarchy	
Inclusive	[13,43,44]
Exclusive	[21,45]
Non-inclusive	Most others
Evaluation Platform	
Real-hardware	[32,46,69,73]
Analytical performance models	[12,29]
Simulator	Nearly all others

We now discuss the CBTs for inclusive and exclusive cache hierarchies and those evaluated on real processors (also see Section 2.2) and using analytical models.

5.1. CBTs for Inclusive Cache Hierarchy

Gupta *et al.* [43] present a CBT for inclusive caches. Their technique uses a bypass buffer (BB) which stores the tags (but no data) of the cache lines that are bypassed (skipped) from LLC. When BB becomes full, a victim tag is evicted from it and the corresponding cache lines in higher level caches are invalidated to satisfy inclusion property. They note that with an effective bypassing algorithm, the lifetime of a bypassed line in higher level caches should be relatively short and these lines are expected to be dead or already evicted when the tag is evicted from the BB. Hence, a small BB is adequate for ensuring inclusion and achieving most performance gains of bypassing. They show that use of BB enables a bypassing algorithm designed for non-inclusive caches [56] to provide nearly same performance gains for inclusive caches. They also use BB to reduce the implementation cost of CBT proposed by [56] (refer to Section 4.4). For this, a competitor pointer is added with each BB-entry and not with each cache set. Also, for virtual bypassing, a BB-entry is allocated for the replaced block. Thus, the reuse information collected by BB can help in simplifying the design of bypassing algorithms.

Ahn *et al.* [13] present a technique which bypasses dead writes to reduce write overhead in NVM LLCs. They classify the writes into three types, viz. (1) dead-on-arrival fills; (2) dead-value fills and (3) closing writes (refer to Section 2.1). A dead-on-arrival fill happens due to streaming pattern (a memory region is never re-accessed after a cache fill) and thrashing pattern (between two accesses to the block, many other blocks in the same set are also accessed). Dead-value fill write is one where the filled block gets overwritten before being read. They use a dead-block predictor which predicts (1) and (2) by correlating dead blocks with instruction addresses that lead to those cache accesses. Further, (3) is predicted using the last-touch instruction address of the block to be written back. This scheme works well for non-inclusive caches. For inclusive caches, however, bypassing (1) and (2) violates inclusion property. To address this, they insert these blocks into the LRU position *without writing their data* and flag it as ″void″. Accesses to ″void″ blocks are treated as misses, but the coherence state bits of 'void' blocks are updated as if they were valid. This maintains inclusion while still reducing write-energy through bypassing. They show that their technique provides large speedup and energy saving.

5.2. CBTs for Exclusive Cache Hierarchy

Chaudhuri *et al.* [45] present a cache hierarchy-aware bypassing scheme for exclusive LLC (L3) and replacement scheme for inclusive LLC. They note that at the end of a block's residency in L2 cache, the future reuse pattern can be estimated based on that observed during its stay in L2 cache. Based on the factors such as number of reuses seen by a block, its state at the time of eviction from L2 and the request (L3 hit or L3 miss) that inserted the block in L2, they categorize L2 blocks in different classes. For example, one class shows the blocks that were filled in L2 on LLC miss, had 'modified' state at the time of eviction and observed exactly one demand use while it was resident in L2. Their technique dynamically learns the reuse probabilities of these classes and by comparing them with a threshold, flags an L2 evicted block as dead or live. Based on this, if the upcoming reuse distance of this block is much larger than LLC capacity, then this block is marked as a candidate for early eviction in LLC, which allows keeping high locality blocks in the LLC. Further, this information is used to make bypassing decisions in an exclusive LLC. When an L2 evicted block is dead, if the target L3 set has an invalid way, the evicted block is allocated in L3 at the LRU position (*i.e.*, highest age). However, if the L3 set has no invalid way, the evicted dead block is bypassed from L3 and is written to memory (if dirty) or dropped (if clean). Their experiments show that their technique reduces cache misses and improves performance.

Gaur *et al.* [21] present bypass and insertion algorithms for an exclusive LLC (L3). A block resides in an exclusive LLC from the time of eviction from L2 to the time it is evicted from LLC or is recalled by L2. For an LLC block, they define the recall distance as the average number of LLC allocations between this block's allocation in LLC and its recall from L2. With an exclusive LLC (L3), a block is allocated in L2 after being fetched from main memory. When it is evicted, it makes its first trip to LLC, which is defined as trip count (TC) being zero. If it is recalled by L2, it makes second trip to LLC (TC = 1). Thus, a large value of trip count shows low average recall distance for a block and the blocks with TC = 0 are candidates for bypassing. Further, the L2 use count of a block shows the number of demand fills plus demand hits seen by it while it stays in L2. Thus, trip count relates with the mean distance between short-term reuse clusters of a block and the use count shows the size of last such cluster. Using these, their technique identifies dead and live blocks and based on this, dead blocks are inserted in LLC only if an invalid location exists in the corresponding set, otherwise, they are bypassed from LLC. They show that their technique improves the performance significantly.

5.3. Evaluation on Real Processor

HP-7200 processor [73] uses a fully-associative on-chip assist cache (2 KB) which is placed parallel to a large direct-mapped data cache (4 KB to 1 MB). A block fetched from memory is first allocated in assist cache. Only when a block shows temporal reuse, it is moved to the data cache, otherwise, it is written-back to memory, bypassing the data cache. This avoids thrashing commonly-observed in direct-mapped data caches.

Xu and Li [46] present a CBT for processors which allow specifying the cache mapping for every virtual page (*i.e.*, whether it is mapped to main cache, mini-cache or is bypassed). For example, Intel StrongARM SA-1110 processor [87] uses an 8 KB 32-way main cache and an 512 B 2-way mini-cache, both of which have 32 B line size and are indexed and tagged by virtual addresses. These caches are mutually exclusive and the compiler can map a page to either of them or bypass it by setting suitable bits. The purpose of mini cache is to hold large data structures so that cache thrashing in main cache is avoided. They show that the optimal page-to-cache mapping problem, which minimizes average memory access time, is NP-hard. Hence, they propose a polynomial-time heuristic that uses greedy strategy to map most accessed pages in the main cache. This memory-profiling guided heuristic begins with the assumption that all pages are mapped to main cache. Then, pages are considered in decreasing order of access count, and they are selectively mapped to mini-cache or are bypassed such that memory access time does not increase. They show that their technique reduces execution time and energy consumption.

5.4. Evaluation Using Analytical Models

Some CBTs use analytical models to guide their bypassing algorithm. Use of analytical models does not incur overhead at run-time, however the limitation of these models is that they may not accurately account for input and runtime variations. We now discuss some of these techniques.

Zhang *et al.* [12] present a CBT for NVM caches that works based on statistical behavior of the entire cache, and not merely a single block. They define data reuse count (DRC) as the total number of references to a block after its allocation in cache. They analytically model the energy cost of bypassing or not-bypassing a block in L2 cache, in terms of read and write energy-values of L2 and L3 cache. They note that for symmetric memory technologies (e.g., SRAM and eDRAM), L2 write energy is much smaller than L3 read energy, but for asymmetric technologies (NVMs), they can be comparable. Hence, only those blocks which show DRC higher than a threshold (called bypassing depth) should be allocated in L2. For example, a block with DRC ≥ 1 can be allocated in an SRAM L2, but only those with DRC ≥ 6 should be allocated in an STT-RAM L2 (L3 is STT-RAM in both cases). The value of bypassing depth is updated periodically. They show that their technique improves the performance and energy efficiency significantly.

Wang *et al.* [29] note that for area optimization, large LLCs are typically designed using single-port memory bitcells instead of multi-port bitcells. However, in a single-port cache, an ongoing write may block the port and delay subsequent performance-critical read requests. Also, in a multicore processor, write requests from one core may obstruct accesses from other cores. Also, due to the high latency of NVM, this issue is more severe in NVM caches than in SRAM caches. They propose a technique to mitigate such port obstruction in NVM LLCs. They analytically model the cost and benefit from cache bypassing in terms of read/write latency of LLC and main memory. Based on it, the processes which may cause LLC port obstruction in any execution interval are detected and the data from these processes are bypassed from LLC. They show that their technique saves energy and also improves performance.

6. CBTs for Specific Memory Technologies

As discussed in Section 2.3.2, CBTs can be highly effective in addressing limitations and exploiting opportunities presented by NVMs and DRAM. For example, cache bypassing reduces accesses to cache which improves the lifetime of NVM caches [72]. Similarly, cache bypassing can mitigate bandwidth bottleneck in large DRAM caches [37]. Table 3 summarizes the CBTs proposed for these technologies and we now discuss them briefly.

Table 3. A classification of CBTs for NVM and DRAM caches.

Classification	References
Bypassing NVM cache	[12,13,29,44,53]
Bypassing cache for reducing accesses to NVM memory	[67]
Bypassing DRAM cache	[37,48]

6.1. Bypassing in Context of NVM Cache or Main Memory

Wang *et al.* [53] present a block placement and migration policy for SRAM-NVM hybrid caches. They classify the LLC write accesses into three classes: core-write (write to LLC through a higher-level write-through cache or eviction of dirty data from a higher-level writeback cache), prefetch-write (write due to prefetch-miss) and demand-write (write due to demand miss). They use access pattern predictors to identify dead blocks and write-burst blocks. These predictors work on the intuition that the future access pattern of a memory access instruction PC is likely to be similar to that in previous accesses. They define the read-range of a demand/prefetch-write access as the largest interval between consecutive reads of the block from the time of filling until time of eviction. The demand-write blocks with zero read range are dead-on-arrival and such blocks are bypassed from the LLC. Further, demand-write blocks with immediate or distant read-range are placed in NVM (e.g., STT-RAM) ways, which reduces the pressure on SRAM ways and leverages the large capacity provided by NVM. They show that their technique reduces writes to NVM and improves performance.

Wang *et al.* [67] note that writing back dirty data to NVM main memory incurs high latency and energy overheads. They propose a technique which aims to keep frequently used data blocks in LLC, based on the insight that such data are also frequently written-back data. They dynamically partition each LLC set into a 'frequent' and an 'infrequent' writeback list. Then, the optimal size of each list is found based on the miss penalty for clean and dirty blocks. For example, for a 16-way cache, the size of these lists can be 4 and 12, respectively. If the optimal size of frequent writeback list equals the associativity of cache, their technique further uses set-sampling to check whether bypassing the read requests from cache provides smaller number of misses than not bypassing them. Based on this, the decision about bypassing the cache is taken. They show that thrashing workloads especially benefit from bypassing and overall, their technique leads to significant reduction in writebacks to NVM main memory.

6.2. Bypassing in Die-Stacked DRAM Caches

Chou *et al.* [37] note that DRAM caches consume bandwidth not only for data transfers on cache hits, but also for secondary operations e.g., miss detection, fill on a miss and writeback probe. They propose a technique which minimizes bandwidth used for each of these secondary operations. Since DRAM caches can stream multiple tags on every access, their technique buffers the tags of recently referenced adjacent cache lines in a separate storage. On a miss to on-chip last level SRAM cache (LLC), the request is first looked up in this storage and a hit in this avoids the need of miss probe in DRAM cache. To reduce bandwidth of cache fills, no-reuse lines can be bypassed. Since naive bypassing hurts hit rate, they perform bandwidth-aware bypass. They define a probabilistic-bypassing scheme which bypasses certain fraction (e.g., 90%) of total misses from the cache. Their technique uses set-dueling to dynamically choose a scheme from no-bypassing and probabilistic-bypassing that provides least miss-rate and then uses this scheme for the whole cache. Thus, their technique trades off bandwidth-saving with hit-rate degradation and allows controlling the hit-rate loss. To reduce bandwidth wasted in writeback probe, they use one bit with each cache line in LLC that tracks whether the line is present in DRAM cache. On eviction of a dirty line from LLC, this bit is checked, and if this bit indicates that the line is not present in DRAM cache, a writeback probe is avoided. By virtue of reducing bandwidth consumption, their technique reduces queuing delay which leads to reduced cache hit latency and improved performance.

Lee *et al.* [48] note that traditional DRAM caches use both TLB and cache tag array for performing virtual-to-physical and physical-to-cache address translation. However, these designs incur significant tag store overhead. They propose using caching granularity to be the same as OS page size (e.g., 4 KB) which avoids the need of tags altogether. They use a cache-map TLB (cTLB) which holds virtual-to-cache address mappings, instead of virtual-to-physical mappings. On a TLB miss, the requested block is allocated in cache (if not there already) and both page table and cTLB are updated with virtual-to-cache mapping. With large DRAM caches, an access to memory region within TLB reach always produces a cache hit since TLB directly provides the cache address of the desired block without requiring tag-checking. The remaining cache space works as victim cache for recently evicted memory pages of cTLB. For performing bypassing, they use an additional bit in page table which decides whether a page bypasses the DRAM cache (but not the on-chip caches e.g., L1 and L2). Using this, pages containing no or few useful blocks can be bypassed from DRAM cache. Similarly, for architectures that use superpages (e.g., 2 MB–1 GB), a superpage can be bypassed from DRAM cache if it does not have sufficient temporal or spatial locality. Further, when a same physical page is shared by multiple page tables, a physical page may be cached at multiple locations and to avoid this, shared pages can be bypassed from DRAM cache. To illustrate the potential of their design, they propose a CBT which sets bypassing flag for pages that have access count smaller than 32, assuming a page size of 4 KB and block size of 64 B. Their technique improves performance by reducing bandwidth consumption and increasing DRAM cache hit-rate.

7. CBTs for GPUs and CPU-GPU Heterogeneous Systems

Table 4 summarizes the CBTs proposed for GPUs and CPU-GPU systems and also highlights their characteristics.

Table 4. A classification of CBTs proposed for GPUs and CPU-GPU systems.

Classification	References
GPU	[7,11,32,34,35,38,47,58–60,62–64,68,75,83,88]
GPU in CPU-GPU system	[41,77,82]
CPU	Nearly all others
Key idea/feature	
Bypassing based on reuse behavior	[7,34,35,38,47,58,59,63,75]
Bypassing based on memory divergence properties	[11,32,34,38,59,60,62,64,68,69]
Bypassing when resources are scarce	[34,83]
Use of core sampling	[11,47,77]

We first discuss some key ideas used by these CBTs and then discuss several CBTs.

1. In CPU-GPU systems, requests from GPUs can be bypassed by leveraging latency tolerance of GPU accesses (Table 4).
2. Several techniques perform bypassing primarily based on reuse characteristics (or utility) of a block (Table 4). For example, these techniques may bypass streaming or thrashing blocks.
3. Under GPU's lock-step execution model, using different cache/bypass decision for different threads of a warp would create differences in their latencies and hence, all the threads would be stalled till the completion of last memory request. By making identical caching/bypassing decision for all threads, and by caching few warps at a time, these memory divergence issues can be avoided (Table 4). Based on these, some techniques seek to cache a warp fully and not partially [11,32,34,38,59,60,68]. Some techniques work by caching/bypassing two warps together or individually [69] or performing request reordering [34,64]. Thus, these techniques perform bypassing together with a thread management scheme.
4. Some techniques perform bypassing when the resources (e.g., MSHR) for servicing a miss cannot be allocated (Table 4).
5. For several GPU applications, the cores show symmetric behavior and hence, by comparatively evaluating different policies on just *few cores*, the optimal policy can be selected for *all the cores*. This strategy, referred to as core sampling, has been used by several CBTs to reduce their metadata overheads (Table 4). Li *et al.* [47] use core-sampling to ascertain cache friendliness of an application such that one core uses their bypassing scheme and another core uses default caching scheme and best scheme is found by comparing their miss-rates. Mekkat *et al.* [77] determine the impact of bypassing on GPU performance by using two different bypassing thresholds with two different

cores. Chen *et al.* [11] estimate 'protecting distance' on a few cores and use this value for the remaining cores.

7.1. CBTs Based on Reuse Characteristics

Li *et al.* [47] propose a CBT for L1D caches in GPU. Their technique decouples the tag and data stores of L1D cache and uses locality filtering in tag store to decide which memory requests can allocate data blocks in data store. Each tag store entry keeps a reference counter (RC) to record the reuse frequency for that address. On a memory request, the tag store is probed. On a miss, a new tag entry is allocated and if no free entry is available, the entry with the smallest RC is selected as victim. In both cases, the request is bypassed from the cache. If tag store probe shows a hit, the corresponding data store entry is checked. If such an entry exists, the request proceeds as the regular cache hit. Otherwise, RC is incremented and compared against a threshold. If RC is lower than the threshold, the block is assumed to show little or no reuse and hence, is bypassed from the cache. However, if RC exceeds the threshold, a new entry is allocated in the data store and if no free block exists, a victim block is evicted. Also, the RC value of all other entries in the set are reduced by one, to ensure that entries with no reuse or distant reuse are eventually evicted from the tag store. This approach benefits cache-unfriendly applications, however, for cache-friendly applications, it delays storage of data in the cache which harms performance. To avoid this, they detect cache friendliness of an application during execution. Using core-sampling, one core uses their approach, while another core uses default caching approach. Periodically, the miss-rates of both the cores are compared and the approach used in the core showing lower miss-rate is then used for all the cores. They show that their technique provides energy saving and speedup in cache-unfriendly irregular applications without affecting cache-friendly regular applications.

Tian *et al.* [35] present a CBT which bypasses streaming values from L1 cache. They use PC of the last memory instruction to predict dead blocks since indexing the predictor using PCs of memory instructions incurs smaller storage overhead than indexing using addresses accessed as there are only few distinct PCs. On every access to the predictor table, a confidence value is obtained. If this value exceeds a threshold, the block accessed by that PC is predicted to be dead. Since wrong predictions lead to additional accesses to lower-level caches, they propose a scheme to correct mispredictions. On an L1 cache bypass, the information is sent to the L2 cache and is stored with the L2 block. If the block is accessed again before eviction from L2, this information is also sent along with the requested data. This indicates a possible error in prior bypass prediction. Based on it, this block is not bypassed the next time, but is inserted in L1 cache for verification and exploiting possible data reuse. By virtue of reducing cache pollution and avoiding unbeneficial cache fills/evictions, their technique improves performance and saves energy.

Choi *et al.* [63] propose write-buffering and read-bypassing for managing GPU caches. These techniques control data placement in shared L2 cache for reducing the memory traffic. They identify the data usage characteristics by code analysis and use this information to perform data placement for individual load/store instructions in the cache. By leveraging this, write-buffering uses shared cache for inter-block communication so that intermediate data need not be stored in off-chip memory. Read-bypassing avoids allocating streaming data in shared cache which are used by one thread-block only. These data are directly allocated in per-block shared memory or L1 cache. This frees L2 cache for storing shared data and/or data allocated for write-buffering. By virtue of reducing off-chip traffic, their techniques achieve large performance improvement.

Lee *et al.* [75] present a programming model/architecture co-optimization technique which utilizes the disciplined memory model of OpenCL. In GPU programming models, properties of memory objects used by a kernel need to be clearly expressed by the programmer and with OpenCL, a kernel function can only access the linear memory space passed explicitly through the input arguments. Further, any memory object is persistent over kernel execution, unlike CPU where memory objects can be dynamically allocated and deallocated. Their technique uses this semantic information to improve cache energy efficiency. They study the variation in cache hit rate across the linear memory address space and observe that the region of the consecutive address space has strong correlation with that of each memory object. This happens because GPU applications are generally optimized to use scratchpad memory which also increases the locality in L2 cache. Also, due to well-defined kernel boundary, a phase change can be clearly detected. Using these facts, their region-aware caching technique collects L1 and L2 hit rates in a training phase and based on these, selectively bypasses a memory region in L1 and L2 caches to save dynamic energy. Since the change in phase within a kernel is much smaller compared to that in CPU, the data collected in training phase accurately represents the entire kernel behavior. Based on the application working set size, their technique also performs way-based reconfiguration of L2 cache to save leakage energy. They show that their technique saves energy in L1 and L2 caches without harming performance or increasing off-chip accesses.

Huangfu *et al.* [7] study the impact of using L1D cache in GPUs for real-time computing and observe that without the cache, GPUs achieve higher average-case performance and better timing predictability. This happens because due of contention, use of cache does not reduce memory access latency and caches complicate the computation of worst-case execution time. To address this, they propose a CBT for L1D cache. They define utilization rate of a load as the fraction of bytes fetched from global memory which are actually consumed by GPU. Utilization rate measures spatial locality and reuse count measures temporal locality. Using profiling, the data accesses with low load utilization rate and low reuse count are detected and are then bypassed from cache. By virtue of reducing global memory traffic and L1D miss-rate, their technique improves average-case performance. For timing-predictability in real-time systems, they further recommend use of static timing analysis schemes with GPU caches.

Khairy *et al.* [58] propose a CBT for bypassing streaming applications from L1 and L2 caches. Their technique records the L1 cache miss rate in each execution interval. At the end of an interval, if the miss-rate is found to be larger than a threshold, the cache is disabled and all accesses bypass the L1 cache. Since the application behavior changes over time, when L1 is disabled, its cache controller still remains enabled. It updates tags only and computes the new miss-rate. If miss-rate is found to be smaller than a threshold, cache is enabled again. The L2 cache also uses similar bypassing scheme. Their technique improves performance of streaming applications.

7.2. CBTs Based on Memory Divergence Properties

Wang *et al.* [68] note that under lock-step execution model of GPU, a warp can be executed only when none of its thread has outstanding memory request. Also, inter- and intra-warp conflicts can reduce data locality in cache. This requires synergistic management of L1D cache and warp scheduling policies and they propose a technique which addresses this need. In their technique, the scheduling priority of the fetching warp determines the cache insertion position of an incoming data block. The cache ways are logically divided into a locality region and a thrashing region. Accordingly, the active warps are divided into two groups, viz. locality warps and thrashing warps, based on their scheduling priority. Replacement is performed only in thrashing region. Cache blocks of locality warps are inserted into locality region, which insulates them from thrashing traffic. Further, divergent loads of thrashing warps are inserted near the LRU positions which reduces their cache residence time. Due to constrained replacement, it is possible that occasionally, their technique may not find a replacement candidate. In such a case, either the L1D cache is bypassed or missing access is repetitively replayed until a block in the thrashing region becomes replaceable. Bypassed requests go to lower cache and the returned data are directly written to register file. Overall, their technique prioritizes coherent loads over divergent loads, so that data blocks of a load instruction are cached as a whole group and not partially. This increases the number of fully cached loads that are ready for execution by the warp schedulers. They show that their technique improves the performance significantly.

Zheng *et al.* [59] present a technique which intelligently allocates L1 cache and bandwidth to effectively utilize cache, bandwidth and compute resources. They infer the warp memory access pattern and the number of memory accesses from the coalescing unit. Also, data locality is inferred from miss rate of data cache. To leverage data locality in L1 cache, the number of warps that can allocate data in cache are limited, such that their footprint can fit into L1 cache. Depending on footprint of warps and size of L1 and L2 cache, additional warps bypass the L1 cache which reduces cache thrashing. However, these bypassed warps are allowed to run to utilize bandwidth and compute units and bandwidth saturation is avoided by limiting the total number of running (cached plus bypassed) warps. Remaining warps that are waiting for running warps to exit/stall are de-scheduled. They show that their technique brings large performance improvement.

Jia *et al.* [34] note that equally sharing GPU L1 caches among its many threads leads to severe cache contention and slowdown for each thread. Also, in GPU, optimizing total memory request processing rate is more important than minimizing latency of individual requests. Based on these, they propose two prioritization schemes which prioritize a few active threads at a time and on their completion, prioritize other threads to use the cache. *Cache bypassing* scheme detects the requests which are expected to cause thrashing or stalls and bypasses them from cache. If a missed request cannot allocate even a single resource required for its processing (e.g., cache line, a miss queue entry, an MSHR entry *etc.*), it stalls the pipeline and needs to be retried later. Their cache bypassing scheme avoids this by directly sending such requests to main memory, mitigating pipeline congestion. Also, the returned data are directly written to registers without being allocated in the cache. Thus, bypassing avoids congestion in cache and pipeline especially in case of bursts of conflicting memory requests. *Request reordering* scheme uses a buffer to rearrange a memory access stream such that requests from related threads are grouped and are issued to cache together. This increases memory access locality and allows the cache to effectively hold working sets of a few threads at a time. They show that their technique improves overall throughput by reducing both inter-warp and intra-warp cache contention and increasing per-thread cache utilization. The technique proposed by Dai *et al.* [83] also performs bypassing when resources required for processing a miss cannot allocated. They apply their technique to L1D and L2 cache individually and together and achieve large performance and energy gains.

Mu *et al.* [64] present a CBT that works based on data locality of concurrent memory requests. In GPUs, memory accesses of a warp are sent to LLC as a single (or few) coalesced request(s). Of the addresses fetched, those actually used by GPU are termed as effective addresses (This idea is similar to the utilization rate defined by Huangfu *et al.* [7]). Memory requests with higher number of effective addresses have higher probability of reuse since more warps are likely to access these addresses. Based on this, their technique assigns multiple (e.g., 32) priority levels to memory requests, such that those with higher number of effective addresses get higher priority. When a cache line sees a hit by a memory request, its priority is set to highest level since it has high likelihood of reuse. On a cache miss, first an invalid line is searched. If no invalid line is found, the priority of a request is compared to that of the line with least priority in the target set. If the former is higher, it replaces the existing line in the cache, otherwise, it bypasses the cache. Periodically, the priority of all cache lines is decreased by one to facilitate their replacement and exploit temporal locality. They also propose reordering the memory requests based on their divergence behavior to reduce the average stall time of warps. They show that their technique provides performance improvement.

Liang *et al.* [69] present a GPU CBT which works by selecting global load instructions (GLI) for bypassing. Their technique finds the data reuse, memory traffic and load efficiency of load instructions. For every GLI (i), they find the number of L1 accesses and L1 hits (H_i). Also, for any two load instructions (say i and j), they also find $H_{i,j}$, which is the number of L1 hits on bypassing all the load instructions except i and j. Then, the value of $H_{i,j} - (H_i + H_j)$ is computed and if it is positive, then both i and j should be cached together to exploit data locality between them, otherwise, one or both of them should be bypassed. Based on this, they construct an L2 traffic reduction graph, which is used to reduce L2 traffic by caching (to exploit data locality) or bypassing (to avoid conflict or low load efficiency). The load efficiency of a GLI depends on its access pattern, cache line size and memory coalescing policy. Using static analysis, they identify three access patterns, viz. streaming access, partial sharing (few threads share the same data) and full sharing (all threads share the same data) and compute their load efficiency using analytical formula. The load efficiency of other access patterns is taken as that of the whole application which is obtained from a profiler. They show that selectively bypassing load instructions for reducing L2 traffic is NP-hard and hence, they propose an ILP-based algorithm and a (polynomial-time) heuristic algorithm. The heuristic algorithm works by iteratively trying to cache the load instructions, and then selecting the ones which provide largest traffic reduction. They show that both ILP and heuristic algorithms provide comparable performance, which is superior to the performance with cache-all and bypass-all schemes.

Chen *et al.* [11] note that in GPUs, bypassing generates massive amount of memory requests which can saturate the MSHR resources and NoC and DRAM bandwidth. This is especially true when memory divergence leads to large working set size and hence, most memory requests need to be bypassed to keep the cache efficient. This necessitates use of warp management (e.g., scheduling or throttling) schemes along with bypassing scheme. They propose a coordinated bypassing and warp throttling technique where warp throttling modulates degree of multithreading and bypassing uses the cache space for hot cache lines to improve cache utilization. They use reuse-distance based bypassing scheme [14], such that protecting distance predictor is used with L1 cache of one (or few) core(s) and the estimate of protecting distance obtained is used for all the cores. Their technique monitors L1 contention and NoC congestion to control the number of active warps (AW). The application begins execution with optimum number of active warps which is found by static profiling. Then, the active-warp count is adjusted based on observed bypassing rate. If NoC latency exceeds a threshold or change in NoC latency exceeds another threshold, AW is reduced. Similarly, AW is increased if NoC latency falls below a threshold. Also, if AW becomes lower than a threshold, it is gradually adjusted to bring NoC latency with a target range. Thus, their technique keeps the NoC congestion in a low range. They show that their technique outperforms the optimal static warp throttling scheme.

Li *et al.* [32] present a technique for bypassing global memory reads in L1, L2 and read-only caches in GPUs. Data are transferred between interconnect and registers via L1 cache in Fermi and some Kepler GPUs and via read-only cache in Maxwell and some Kepler GPUs. Also, data are transferred between global memory and interconnect via L2 cache in Fermi, Kepler and Maxwell GPUs. They note that for cache-insensitive applications, increasing the thread volume leads to better bandwidth utilization, which improves memory-system throughput until the bandwidth becomes saturated. For cache sensitive applications, increasing thread volume increases memory-system throughput due to better utilization of cache, however, beyond a certain thread volume, increased cache congestion sharply reduces the throughput. They use a CBT to maintain the ideal thread volume for optimizing cache performance. This technique uses a threshold such that warps with index higher than the threshold bypass the cache. To find the threshold, they use a static and a dynamic technique. The static technique experiments with all possible threshold values, e.g., for GPU applications with 16 warps in a thread block, it bypasses between 0 to 15 warps. The dynamic technique uses sampling approach such that different thread blocks use different thresholds, and the threshold value leading to the least execution time is selected. They observe that benefit of bypassing is larger in L2 than in L1 and thus, system performance is more sensitive to L2 cache than L1 cache. Also, bypassing benefit on L1 and L2 caches are not cumulative. They also suggest bypassing and cache sizing strategies for optimizing performance on each of Fermi, Kepler and Maxwell GPUs.

Xie *et al.* [38] use a CBT which uses both compile-time and run-time information to perform bypassing. At compile time, profiling is used to identify the global loads that have either very high or poor locality, which is decided based on their hit rates. Based on this, these loads are cached or bypassed (respectively) for all the threads. For the remaining loads, the decision to bypass is taken at runtime. For this, a subset of active thread blocks are bypassed which reduces cache contention and pipeline stall. The remaining active thread blocks use the cache which exploits data-locality for them. Thus, thread level parallelism is not sacrificed and massive multithreading is maintained for optimizing throughput. They show that their technique provides large speedup.

Ausavarungnirun *et al.* [60] note that different warps in GPU show different amount of memory divergence in shared L2 cache. For example, for some "mostly-miss" warps, most threads see cache miss, while for other "mostly-hit" warps, most threads see cache hit. Further, due to TLP in a GPU, a burst of memory requests can arrive at L2 cache and cause queuing delays of hundreds of cycles. They propose a technique for addressing such memory divergence. Using the property that the memory divergence behavior of a warp persists for long execution periods, their technique collects the history for characterizing warp behavior. Using this, their technique aims to convert 'mostly-hit' warps into 'all-hit' warps since it totally avoids stall time for those warps. This is achieved by a warp-type-aware cache insertion policy that prioritizes requests from mostly-hit warps to effectively allocate extra cache space to them. For doing this, their technique converts 'mostly-miss' warps into 'all-miss' warps since it does not incur extra stall penalty for them and their cache space can be allocated to mostly-hit warps. Further, since future memory requests of mostly-miss warps would not be cached in L2, they can bypass the cache and thus, totally avoid unbeneficial L2 access and queuing delays. They show that their technique improves performance and energy efficiency.

7.3. CBTs for CPU-GPU Heterogeneous Systems

In shared LLC heterogeneous architectures, cache accesses from GPU may significantly reduce the LLC share of CPU applications and hence, cause large performance loss. Mekkat *et al.* [77] note that GPUs can hide memory access latency through multithreading and based on it, their technique throttles LLC accesses from GPU to increase the cache quota of latency-sensitive CPU applications. At any time, the number of 'ready-to-schedule' warps provides a measure of TLP. With large number of warps, TLP is higher and thus, higher memory access latency can be tolerated by the GPU. In other words, the available TLP shows the cache sensitivity of a GPU application. In their technique, GPU memory requests bypass the LLC if GPU shows large TLP or is insensitive to LLC performance. Based on core-sampling idea, their technique uses two different bypassing thresholds (a higher threshold and a lower threshold) to two different cores to assess how GPU performance is impacted by bypassing. If the performance difference between the cores is small, the application is cache insensitive, otherwise, bypassing is assumed to have large impact on application performance. Further, the impact of GPU bypassing on performance of CPU is assessed using cache set-sampling. Based on these, the aggressiveness of GPU bypassing is regulated. They show that their technique improves performance significantly.

Wang *et al.* [41] study the performance impact of LLC (L3) sharing between CPU and GPU in a fused (integrated) CPU-GPU system. Since GPU generates much larger memory traffic than CPU, LLC sharing leads to sharp reduction in LLC hit rate of CPU, but does not affect the hit rate of GPU. To address this, they study bypassing of GPU requests from LLC. While this improves the hit rate of CPU, it harms the performance of both CPU and GPU due to severe main memory contention. To mitigate such bandwidth interference between CPU and GPU, they propose partitioning of memory channels between them, although it also reduces the bandwidth available for them. They observe that channel partitioning improves CPU performance, especially for memory-intensive workloads and thus, reduced interference offsets the impact of reduced bandwidth. However, GPU performance drops further since the bandwidth requirement after LLC bypassing cannot be met with a single memory channel. Thus, both LLC space and memory bandwidth are crucial for maintaining GPU performance. Overall, their study highlights the need of careful management of LLC in fused CPU/GPU systems.

8. Future Challenges and Conclusions

A majority of existing cache bypassing techniques have been proposed in context of discrete CPUs and GPUs. Since both CPU and GPU have unique features, fused (integrated) CPU-GPU heterogeneous architectures are expected to become dominant computing platform in near future [42], as evident from recent commercial designs, e.g., AMD's accelerated processing units (APUs), Intel's Ivybridge and NVIDIA's Echelon project. However, cache management in such heterogeneous systems also presents significant design challenges and hence, partial retrofitting of existing CBTs for these systems will be insufficient. Design of novel CBTs for these systems will be a major research challenge for computer architects.

Existing processors use several cache management techniques such as cache reconfiguration, cache compression, prefetching, *etc.* [15,79]. Synergistic integration of cache bypassing with these techniques will have a significant bearing on its adoption in commercial processors. For example, bypassing low-reuse blocks allows aggressively prefetching useful blocks in the cache without causing cache pollution. Evidently, a careful study of interaction between cache bypassing and existing cache management schemes will be an interesting problem for system designers.

In presence of faults due to process variation or limited write endurance, there is a gradual degradation in cache capacity [89]. In such a scenario, reducing the cache traffic becomes even more important. While existing works have used cache bypassing mainly for performance and energy optimizations, exploring the use of bypassing for tolerating faults will be a promising area for future research.

The presence of error-tolerant applications/phases and perceptual limitations of users allow trading-off storage and computation accuracy for improving energy/performance and this is known as approximate computing approach [90]. In such scenarios, some blocks can be bypassed from cache and can be later approximated which avoids the need of memory access. We believe that this bypassing approach will be very interesting in near future since it incurs only small and acceptable loss in accuracy while providing much larger gains in efficiency compared to the existing CBTs which work under the requirement of fully correct execution and hence, can only provide limited efficiency gains.

In this paper, we presented a survey of cache bypassing techniques. We discussed CBTs for dominant computing systems, viz. CPU, GPU and CPU-GPU systems. To underscore the similarities and differences between different works, we organized them in several categories and discussed their key insights. It is hoped that by providing a bird's eye view of the research field, this paper will offer clear directions for future developments in the area.

Acknowledgments: This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Fluhr, E.J.; Friedrich, J.; Dreps, D.; Zyuban, V.; Still, G.; Gonzalez, C.; Hall, A.; Hogenmiller, D.; Malgioglio, F.; Nett, R.; *et al.* 5.1 POWER8^{TM}: A 12-core server-class processor in 22 nm SOI with 7.6 Tb/s off-chip bandwidth. In Proceedings of the International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 9–13 February 2014; pp. 96–97.
2. Kurd, N.; Chowdhury, M.; Burton, E.; Thomas, T.P.; Mozak, C.; Boswell, B.; Lal, M.; Deval, A.; Douglas, J.; Elassal, M.; *et al.* 5.9 Haswell: A family of IA 22 nm processors. In Proceedings of the International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 9–13 February 2014; pp. 112–113.
3. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2009. Available online: http://goo.gl/X2AI0b (accessed on 27 April 2016).
4. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture:Kepler GK110/210. 2014. Available online: http://goo.gl/qOSWW1 (accessed on 27 April 2016).
5. Harris, M. 5 Things You Should Know about the New Maxwell GPU Architecture. 2014. Available online: http://goo.gl/8NV82n (accessed on 27 April 2016).
6. Mittal, S. A survey of techniques for managing and leveraging caches in GPUs. *J. Circuits Syst. Comput.* **2014**, *23*, 229–236.
7. Huangfu, Y.; Zhang, W. Real-Time GPU Computing: Cache or No Cache? In Proceedings of the International Symposium on Real-Time Distributed Computing (ISORC), Auckland, New Zealand, 13–17 April 2015; pp. 182–189.
8. Chi, C.H.; Dietz, H. Improving cache performance by selective cache bypass. In Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Kailua-Kona, HI, USA, 3–6 January 1989; Volume 1, pp. 277–285.

9. John, L.K.; Subramanian, A. Design and performance evaluation of a cache assist to implement selective caching. In Proceedings of the International Conference on Computer Design, Austin, TX, USA, 12–15 October 1997; pp. 510–518.
10. Collins, J.D.; Tullsen, D.M. Hardware identification of cache conflict misses. In Proceedings of the International Symposium on Microarchitecture, Haifa, Israel, 16–18 November 1999; pp. 126–135.
11. Chen, X.; Chang, L.W.; Rodrigues, C.I.; Lv, J.; Wang, Z.; Hwu, W.M. Adaptive cache management for energy-efficient GPU computing. In Proceedings of the 47th International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014; pp. 343–355.
12. Zhang, C.; Sun, G.; Li, P.; Wang, T.; Niu, D.; Chen, Y. SBAC: A statistics based cache bypassing method for asymmetric-access caches. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), La Jolla, CA, USA, 11–13 August 2014; pp. 345–350.
13. Ahn, J.; Yoo, S.; Choi, K. DASCA: Dead write prediction assisted STT-RAM cache architecture. In Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 25–36.
14. Duong, N.; Zhao, D.; Kim, T.; Cammarota, R.; Valero, M.; Veidenbaum, A.V. Improving cache management policies using dynamic reuse distances. In Proceedings of the 45th International Symposium on Microarchitecture, Vancouver, BC, Canada, 1–5 December 2012; pp. 389–400.
15. Mittal, S. A Survey of Architectural Techniques For Improving Cache Power Efficiency. *Sustain. Comput. Inform. Syst.* **2014**, *4*, 33–43.
16. Belady, L.A. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.* **1966**, *5*, 78–101.
17. Atkins, M. Performance and the i860 microprocessor. *IEEE Micro* **1991**, *11*, 24–27.
18. Intel Corporation. *Intel 64 and IA-32 Architectures, Software Developer's Manual, Instruction Set Reference, A-Z*; Intel Corporation: Santa Clara, CA , USA, 2011; Volume 2.
19. NVIDIA Corporation. *Parallel Thread Execution ISA Version 4.2*; NVIDIA Corporation: Santa Clara, CA , USA, 2015.
20. Kharbutli, M.; Solihin, Y. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.* **2008**, *57*, 433–447.
21. Gaur, J.; Chaudhuri, M.; Subramoney, S. Bypass and insertion algorithms for exclusive last-level caches. In Proceedings of the 38 th International Symposium on Computer Architecture (ISCA), San Jose, CA, USA, 4–8 June 2011; pp. 81–92.
22. Mittal, S.; Zhang, Z.; Vetter, J. FlexiWay: A Cache Energy Saving Technique Using Fine-grained Cache Reconfiguration. In Proceedings of the 31st IEEE International Conference on Computer Design (ICCD); Asheville, NC, USA, 6–9 October 2013.
23. Alves, M.; Khubaib, K.; Ebrahimi, E.; Narasiman, V.; Villavieja, C.; Navaux, P.O.A.; Patt, Y.N. Energy savings via dead sub-block prediction. In Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), New York, NY, USA, 24–26 October 2012; pp. 51–58.
24. Mittal, S.; Zhang, Z. EnCache: A Dynamic Profiling Based Reconfiguration Technique for Improving Cache Energy Efficiency. *J. Circuits Syst. Comput.* **2014**, *23*, 1450147.
25. Mittal, S.; Vetter, J.S.; Li, D. A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-volatile On-chip Caches. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 1524–1537.
26. Mittal, S. A Survey of Power Management Techniques for Phase Change Memory. *Int. J. Comput. Aided Eng. Technol.* **2014**.
27. Mittal, S.; Poremba, M.; Vetter, J.; Xie, Y. *Exploring Design Space of 3D NVM and eDRAM Caches Using DESTINY Tool*; Technical Report ORNL/TM-2014/636; Oak Ridge National Laboratory: Oak Ridge, TN, USA, 2014.
28. Mittal, S.; Vetter, J.S. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 1537–1550.
29. Wang, J.; Dong, X.; Xie, Y. OAP: An obstruction-aware cache management policy for STT-RAM last-level caches. In Proceedings of the Conference on Design, Automation and Test in Europe, Grenoble, France, 18–22 March 2013; pp. 847–852.
30. Mittal, S.; Vetter, J. A Survey of Techniques for Architecting DRAM Caches. *IEEE Trans. Parallel Distrib. Syst.* **2015**, doi:10.1109/TPDS.2015.2461155.

31. AMD. AMD Graphics Cores Next (GCN) Architecture. 2012. Available online: https://goo.gl/NjNcDY (accessed on 27 April 2016).
32. Li, A.; van den Braak, G.J.; Kumar, A.; Corporaal, H. Adaptive and Transparent Cache Bypassing for GPUs. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Austin, TX, USA, 15–20 November 2015.
33. Hagedoorn, H. Core i7 5775C Processor Review: Desktop Broadwell—The Broadwell-H Architecture. 2015. Available online: http://goo.gl/1QFwja (accessed on 27 April 2016).
34. Jia, W.; Shaw, K.; Martonosi, M. MRPB: Memory request prioritization for massively parallel processors. In Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 272–283.
35. Tian, Y.; Puthoor, S.; Greathouse, J.L.; Beckmann, B.M.; Jiménez, D.A. Adaptive GPU cache bypassing. In Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, San Francisco, CA, USA, 7 February 2015; pp. 25–35.
36. Etsion, Y.; Feitelson, D.G. Exploiting core working sets to filter the L1 cache with random sampling. *IEEE Trans. Comput.* **2012**, *61*, 1535–1550.
37. Chou, C.; Jaleel, A.; Qureshi, M.K. BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches. In Proceedings of the 42nd International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 13–17 June 2015.
38. Xie, X.; Liang, Y.; Wang, Y.; Sun, G.; Wang, T. Coordinated static and dynamic cache bypassing for GPUs. In Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, CA, USA, 7–11 February 2015; pp. 76–88.
39. Li, L.; Tong, D.; Xie, Z.; Lu, J.; Cheng, X. Optimal bypass monitor for high performance last-level caches. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, Minneapolis, MN, USA, 19–23 September 2012; pp. 315–324.
40. Kharbutli, M.; Jarrah, M.; Jararweh, Y. SCIP: Selective cache insertion and bypassing to improve the performance of last-level caches. In Proceedings of the IEEE Conference on Applied Electrical Engineering and Computing Technologies (AEECT), Amman, Jordan, 3–5 December 2013; pp. 1–6.
41. Wang, P.H.; Liu, G.H.; Yeh, J.C.; Chen, T.M.; Huang, H.Y.; Yang, C.L.; Liu, S.L.; Greensky, J. Full system simulation framework for integrated CPU/GPU architecture. In Proceedings of the International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Hsinchu, Taiwan, 28–30 April 2014; pp. 1–4.
42. Mittal, S.; Vetter, J. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* **2015**, *47*, 69:1–69:35.
43. Gupta, S.; Gao, H.; Zhou, H. Adaptive cache bypassing for inclusive last level caches. In Proceedings of the International Symposium on Parallel & Distributed Processing (IPDPS), Cambridge, MA, USA, 20–24 May 2013; pp. 1243–1253.
44. Kim, M.K.; Choi, J.H.; Kwak, J.W.; Jhang, S.T.; Jhon, C.S. Bypassing method for STT-RAM based inclusive last-level cache. In Proceedings of the Conference on Research in Adaptive and Convergent Systems, Prague, Czech Republic, 9–12 October 2015; pp. 424–429.
45. Chaudhuri, M.; Gaur, J.; Bashyam, N.; Subramoney, S.; Nuzman, J. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, Minneapolis, MN, USA, 19–23 September 2012; pp. 293–304.
46. Xu, R.; Li, Z. Using cache mapping to improve memory performance handheld devices. In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, USA, 10–12 March 2004; pp. 106–114.
47. Li, C.; Song, S.L.; Dai, H.; Sidelnik, A.; Hari, S.K.S.; Zhou, H. Locality-Driven Dynamic GPU Cache Bypassing. In Proceedings of the International Conference on Supercomputing (ICS), Newport Beach, CA, USA, 8–11 June 2015.
48. Lee, Y.; Kim, J.; Jang, H.; Yang, H.; Kim, J.; Jeong, J.; Lee, J.W. A fully associative, tagless DRAM cache. In Proceedings of the International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; pp. 211–222.

49. Xiang, L.; Chen, T.; Shi, Q.; Hu, W. Less reused filter: Improving L2 cache performance via filtering less reused lines. In Proceedings of the 23rd International conference on Supercomputing, Yorktown Heights, NY, USA, 8–12 June 2009; pp. 68–79.
50. Liu, H.; Ferdman, M.; Huh, J.; Burger, D. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In Proceedings of the International Symposium on Microarchitecture, Como, Italy, 8–12 November 2008; pp. 222–233.
51. Feng, M.; Tian, C.; Gupta, R. Enhancing LRU replacement via phantom associativity. In Proceedings of the 16th Workshop on Interaction between Compilers and Computer Architectures (INTERACT), New Orleans, LA, USA, 25 February 2012; pp. 9–16.
52. Park, J.; Yoo, R.M.; Khudia, D.S.; Hughes, C.J.; Kim, D. Location-aware cache management for many-core processors with deep cache hierarchy. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17–22 November 2013; p. 20.
53. Wang, Z.; Jiménez, D.A.; Xu, C.; Sun, G.; Xie, Y. Adaptive placement and migration policy for an STT-RAM-based hybrid cache. In Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 13–24.
54. Yu, B.; Ma, J.; Chen, T.; Wu, M. Global Priority Table for Last-Level Caches. In Proceedings of the International Conference on Dependable, Autonomic and Secure Computing (DASC), Sydney, Australia, 12–14 December 2011; pp. 279–285.
55. Das, S.; Aamodt, T.M.; Dally, W.J. SLIP: Reducing wire energy in the memory hierarchy. In Proceedings of the International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; pp. 349–361.
56. Gao, H.; Wilkerson, C. A dueling segmented LRU replacement algorithm with adaptive bypassing. In Proceedings of the JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship (JWAC), Saint-Malo, France, 20 June 2010.
57. Wu, Y.; Rakvic, R.; Chen, L.L.; Miao, C.C.; Chrysos, G.; Fang, J. Compiler managed micro-cache bypassing for high performance EPIC processors. In Proceedings of the 35th Annual IEEE International Symposium on Microarchitecture, Istanbul, Turkey, 18–22 November 2002; pp. 134–145.
58. Khairy, M.; Zahran, M.; Wassal, A.G. Efficient utilization of GPGPU cache hierarchy. In Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, San Francisco, CA, USA, 7 February 2015; pp. 36–47.
59. Zheng, Z.; Wang, Z.; Lipasti, M. Adaptive cache and concurrency allocation on GPGPUs. *IEEE Comput. Archit. Lett.* **2015**, *14*, 90–93.
60. Ausavarungnirun, R.; Ghose, S.; Kayiran, O.; Loh, G.H.; Das, C.R.; Kandemir, M.T.; Mutlu, O. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In Proceedings of the International Conference on Parallel Architecture and Compilation (PACT), San Francisco, CA, USA, 18–21 October 2015.
61. Tyson, G.; Farrens, M.; Matthews, J.; Pleszkun, A.R. A modified approach to data cache management. In Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, USA, 29 November–1 December 1995; pp. 93–103.
62. Dai, H.; Gupta, S.; Li, C.; Kartsaklis, C.; Mantor, M; Zhou, H.. A Model-Driven Approach to Warp/Thread-Block Level GPU Cache Bypassing. In Proceedings of the Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016.
63. Choi, H.; Ahn, J.; Sung, W. Reducing off-chip memory traffic by selective cache management scheme in GPGPUs. In Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, London, UK, 3 March 2012; pp. 110–119.
64. Mu, S.; Deng, Y.; Chen, Y.; Li, H.; Pan, J.; Zhang, W.; Wang, Z. Orchestrating cache management and memory scheduling for GPGPU applications. *IEEE Trans. Very Large Scale Integr. Syst.* **2014**, *22*, 1803–1814.
65. Johnson, T.L.; Hwu, W.M.W. Run-time adaptive cache hierarchy management via reference analysis. In Proceedings of the International Symposium on Computer Architecture, Denver, CO, USA, 1–4 June 1997; Volume 25, pp. 315–326.
66. Jalminger, J.; Stenström, P. A novel approach to cache block reuse prediction. In Proceedings of the 42nd International Conference on Parallel Processing, Kaohsiung, Taiwan, 6–9 October 2003; pp. 294–302.
67. Wang, Z.; Shan, S.; Cao, T.; Gu, J.; Xu, Y.; Mu, S.; Xie, Y.; Jiménez, D.A. WADE: Writeback-aware dynamic cache management for NVM-based main memory system. *ACM Trans. Archit. Code Optim.* **2013**, *10*, 51:1–51:21.

68. Wang, B.; Yu, W.; Sun, X.H.; Wang, X. DaCache: Memory Divergence-Aware GPU Cache Management. In Proceedings of the 29th International Conference on Supercomputing, Newport Beach, CA, USA, 8–11 June 2015; pp. 89–98.
69. Liang, Y.; Xie, X.; Sun, G.; Chen, D. An Efficient Compiler Framework for Cache Bypassing on GPUs. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 18–21 November 2013.
70. Malkowski, K.; Link, G.; Raghavan, P.; Irwin, M.J. Load miss prediction-exploiting power performance trade-offs. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, CA, USA, 26–30 March 2007; pp. 1–8.
71. González, A.; Aliagas, C.; Valero, M. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In Proceedings of the 9th International Conference on Supercomputing, Barcelona, Spain, 3–7 July 1995; pp. 338–347.
72. Mittal, S.; Vetter, J. A Technique For Improving Lifetime of Non-volatile Caches using Write-minimization. *J. Low Power Electron. Appl.* **2016**, *6*, 1.
73. Chan, K.K.; Hay, C.C.; Keller, J.R.; Kurpanek, G.P.; Schumacher, F.X.; Zheng, J. Design of the HP PA 7200 CPU. *HP J.* **1996**.
74. Karlsson, M.; Hagersten, E. Timestamp-based selective cache allocation. In *High Performance Memory Systems*; Springer: New York, NY, USA, 2004; pp. 43–59.
75. Lee, J.; Woo, D.H.; Kim, H.; Azimi, M. GREEN Cache: Exploiting the Disciplined Memory Model of OpenCL on GPUs. *IEEE Trans. Comput.* **2015**, *64*, 3167–3180.
76. Khan, S.; Tian, Y.; Jiménez, D. Sampling dead block prediction for last-level caches. In Proceedings of the International Symposium on Microarchitecture (MICRO), Atlanta, GA, USA, 4–8 December 2010; pp. 175–186.
77. Mekkat, V.; Holey, A.; Yew, P.C.; Zhai, A. Managing shared last-level cache in a heterogeneous multicore processor. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Edinburgh, UK, 7–11 September 2013, pp. 225–234.
78. Mittal, S. A Survey Of Techniques for Cache Locking. *ACM Trans. Des. Autom. Electron. Syst.* **2016**, *21*, 49:1–49:24.
79. Mittal, S. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.* **2016**.
80. Mittal, S.; Cao, Y.; Zhang, Z. MASTER: A multicore cache energy saving technique using dynamic cache reconfiguration. *IEEE Trans. Very Large Scale Integr. Syst.* **2014**, *22*, 1653–1665.
81. Kampe, M.; Stenstrom, P.; Dubois, M. Self-correcting LRU replacement policies. In Proceedings of the 1st Conference on Computing Frontiers, Ischia, Italy, 14–16 April 2004; pp. 181–191.
82. Ma, J.; Meng, J.; Chen, T.; Shi, Q.; Wu, M.; Liu, L. Improve LLC Bypassing Performance by Memory Controller Improvements in Heterogeneous Multicore System. In Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Hong Kong, 9–11 December 2014; pp. 82–89.
83. Dai, H.; Kartsaklis, C.; Li, C.; Janjusic, T.; Zhou, H. RACB: Resource Aware Cache Bypass on GPUs. In Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW), Paris, France, 22–24 October 2014; pp. 24–29.
84. Lesage, B.; Hardy, D.; Puaut, I. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In Proceedings of the 18th International Conference on Real-Time and Network Systems, Toulouse, France, 4–5 Novermber 2010; p. 2283.
85. Hardy, D.; Piquet, T.; Puaut, I. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS), Washington, DC, USA, 1–4 December 2009; pp. 68–77.
86. Jaleel, A.; Theobald, K.B.; Steely, S.C., Jr.; Emer, J. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 37th International Symposium on Computer Architecture, Saint-Malo, France, 19–23 June 2010; pp. 60–71.
87. Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*; Intel Corporation: Santa Clara, CA, USA, 2000.

88. Xie, X.; Liang, Y.; Sun, G.; Chen, D. An efficient compiler framework for cache bypassing on GPUs. In Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 18–21 November 2013; pp. 516–523.
89. Mittal, S. A survey of architectural techniques for managing process variation. *ACM Comput. Surv.* **2016**, *48*, Article No. 54.
90. Mittal, S. A survey of techniques for approximate computing. *ACM Comput. Surv.* **2016**, *48*, Article No. 62.