Rebasing and Optimizing Cache Miss Handling: Fetching Beyond MSHR Capacity

Simranjit Singh[1] Agustín Navarro-Torres[2] Biswabandan Panda[3] Alberto Ros[1]
[1]Computer Engineering Department (DITEC), University of Murcia, Spain
[2]Dept. Informática e Ingeniería de Sistemas - I3A, Universidad de Zaragoza, Spain
[3]Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India
simranjit.singh@um.es agusnt@unizar.es biswa@cse.iitb.ac.in aros@ditec.um.es

***Abstract*—Miss Status Handling Registers (MSHRs) track cache miss information while cache lines are being fetched from higher cache levels or memory. In traditional implementations, when all MSHR entries are occupied, the cache cannot process new misses, stalling demand requests and dropping or delaying prefetches, and in the end, limiting memory-level parallelism. Scaling the number of MSHR entries is not straightforward, as they must support associative lookups for high performance. In addition, hardware prefetching further increases MSHR pressure.**

This paper first observes that academic work has largely overlooked industrial implementations of MSHRs, with important implications for performance. Then, it details the implementation of industry-like MSHR designs, namely Homeless, in which prefetch requests can still be issued when MSHRs are full but are *degraded* to fill higher cache levels, and its demand-to-prefetch (D2P) variant, where a demand miss that cannot allocate an MSHR entry is instead issued as a prefetch-style request. Finally, we analyze the need for the first-level data cache (L1D) MSHR information for both demand and prefetch misses, and propose Nomad, a microarchitectural technique that (i) enables prefetching to L1D even when the MSHR is full and (ii) allows incoming demand misses to reclaim L1D MSHR entries occupied by prefetch requests. Across six benchmark suites, Nomad delivers the strongest improvements on AI/ML and GAP, where L1D MSHR pressure is highest, and performs similarly to Homeless-D2P on the remaining suites. For AI/ML workloads and in the absence of an L1D prefetcher, Nomad reaches 11.3% speedup, while Homeless-D2P achieves 8.3%. With an L1D Berti prefetcher, Nomad reaches 14.4% speedup compared to the 9.4% achieved by Homeless-D2P. These improvements come with no storage cost.

***Index Terms*—MSHRs, prefetching, memory-level parallelism, cache miss handling, microarchitecture**

I. Introduction

Miss Status Handling Registers (MSHRs) [1] track outstanding cache misses and play an important role in enabling memory-level parallelism (MLP). In essence, the MLP is limited by the number of MSHRs in the first-level data cache (L1D). MSHRs track misses at cache line granularity. Cache misses check the MSHRs to determine if a miss for the same cache line address is already pending. If a matching request is already pending, the new request is merged (or coalesced) with the existing one, so no additional request is sent to higher cache levels, such as the second-level cache (L2) and the last-level cache (LLC). When the MSHR is full, new requests missing in cache cannot be redirected to a higher cache level, which reduces MLP. Since MSHRs are commonly implemented as fully-associative structures and require searching all their entries on any cache miss, they are difficult to scale [2]–[7].

Impact of a full L1D MSHR. When all MSHR entries are occupied, L1D cannot process new misses. Demand requests that miss in L1D are negatively acknowledged and must be retried at a later time [8], [9], introducing stall cycles that reduce MLP. Store read-for-ownership (RFO) requests [10] are dropped when no MSHR entry is available, degrading the write throughput. Hardware prefetch requests further increase the MSHR pressure and are dropped or delayed when the MSHR is full (Section II-D).

Industry alternatives. The problem of MSHR full is well recognized in the processor industry. However, their solutions are not well disclosed. Through examination of patent applications from AMD [11], Qualcomm [12], and Samsung [13], [14], as well as vendor BIOS configuration guides for Intel processors [15]–[18], we identify a variety of techniques that manage MSHR resources differently when MSHR becomes full. These techniques are described with minimal implementation detail and, to the best of our knowledge, have not been previously analyzed in the academic literature, with important performance consequences. We present the first detailed description and systematic study of one of such techniques as a representative of industry standard, namely Homeless Prefetch [16]–[18], where prefetch requests can still be issued when MSHRs are full but are *degraded* to fill higher cache levels, and its demand-to-prefetch (D2P) variant [15], where a demand miss that cannot allocate an MSHR entry is instead issued as a prefetch-style request, stressing the need for academic work to improve the standard baseline (Section IV).

The problem. Despite their diversity, all industry tech-

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO project, grant agreement No 819134), from MICIU/AEI/10.13039/501100011033 and the "ERDF A way of making Europe", EU (DAMAS project, grant PID2022-136315OB-I00), from MICIU/AEI/10.13039/501100011033 (TAPAS project, grant PDC2025-166157-I00), from Fundación Séneca (SCALE project, grant 23039/GERM/25), from MICIU/AEI/10.13039/501100011033 (APCE III project, grant PID2022-136454NB-C22), and from the Government of Aragon (T58_23R research group). This work is also supported by AMD and Intel research gifts.

niques converge on the same approach. When the L1D MSHR is full, requests fill higher-level caches (e.g., L2) rather than the L1D. This incurs additional access latency, particularly costly for store operations under a total store order (TSO) consistency model [19], which depends on timely L1D write hits to pipeline stores efficiently [20]. For L1D-targeted data prefetchers [21], [22] their potential performance is also limited. In addition, none of these techniques provides a mechanism to reclaim MSHR entries occupied by prefetch requests on behalf of incoming demand misses. As a result, demand requests have no option but to wait, even when MSHR entries are held by prefetch requests that have not yet completed.

Our observation. We observe that demand requests and prefetch requests have fundamentally different MSHR requirements. A demand miss must track the destination register, the cache line offset, data size, and how to extend the data to the register size (signed or unsigned) in order to route the returning data to the correct memory operation upon completion [20]. In contrast, a prefetch miss only needs to allocate the cache line in L1D, as it is not linked to any in-flight memory operation. The question we ask in this work is: can this asymmetry be exploited to manage MSHR resources more effectively?

Our approach. We answer affirmatively and propose *Nomad*, a set of lightweight L1D MSHR management policies that exploit the demand-prefetch asymmetry. When the L1D MSHR is full, Nomad enables three mechanisms. (i) *MSHR bypass*, where prefetch requests are issued to the higher cache level without allocating an L1D MSHR entry, yet the returned cache line is still filled into L1D. (ii) *MSHR eviction*, where MSHR entries occupied by prefetch requests are reclaimed for incoming demand misses, since the in-flight prefetch can complete without MSHR tracking. (iii) *Demand-to-prefetch (D2P)*, which, unlike industry implementations, enables L1D fills without MSHR allocation, allowing demand accesses to subsequently hit in L1D.

Contributions. We present the first detailed description and academic analysis of MSHR management techniques employed in commercial processors and their impact on hardware prefetchers. Our evaluation reveals that proper MSHR management alone can shift prefetcher speedups by several percentage points, an effect not captured in academic evaluations that drop or stall prefetch requests when MSHRs are full. We propose and evaluate Nomad, a mechanism that enables L1D fills without L1D MSHR allocation.

Key results. Nomad enables small L1D MSHR budgets to recover most of the performance lost when the MSHR is full. The largest gains over Homeless-D2P appear on AI/ML and GAP, where L1D MSHR pressure is highest, while Nomad performs similarly to Homeless-D2P on SPEC CPU2017, Ligra, GMS, and Google. Without an L1D hardware prefetcher, an 8-entry Nomad design outperforms the 16-entry Standard-MSHR baseline on SPEC CPU2017, GAP, GMS, AI/ML, and Ligra. For AI/ML workloads and in the absence of L1D prefetcher, Nomad reaches 11.3% speedup, while Homeless-D2P achieves 8.3%. With an L1D Berti prefetcher, Nomad reaches 14.4% speedup compared to the 9.4% achieved by Homeless-D2P, with per-application speedups of up to 85.4% for `619.lbm`.

II. Background

A. Miss Status Handling Registers (MSHRs)

A memory request that experiences an L1D miss must be forwarded to higher levels, and the instruction that triggered the request cannot be completed until the requested L1D line is returned. If the L1D cannot handle multiple in-flight misses, the processor pipeline stalls until the outstanding miss is resolved. Kroft [1] introduced the concept of MSHRs to allow multiple concurrent outstanding cache misses without stalling the processor. Initially, MSHRs store information about the line address requested by the processor and a valid bit. Kroft's design supports a single outstanding miss per word of the cache line being fetched. Farkas [23] further improves this design to allow multiple misses per word by extending the valid bit to `n` bits, representing each word that could be read from a cache line. The first memory request that misses in cache (*primary miss*) creates a new entry in the MSHR. Subsequent misses for the same cache line (*secondary misses*) are coalesced. The cache stalls only when the number of distinct cache lines missing simultaneously exceeds the number of available MSHR entries (a *structural-stall miss*).

B. Functionality of MSHRs

This section describes the current functionality of MSHRs that goes beyond the ability to enable multiple cache misses.

1) Resolving memory operations missing in cache. The MSHRs keep track of demand requests and supply the requested data to memory operations once the cache miss is resolved [1]. This is commonly done by tracking the offset of the requested data, the size to be loaded, the register where the data should be stored, and how to extend the data to the register size (signed or unsigned) [20].

2) Coalescing. When a new cache miss occurs, the MSHR is associatively searched for a matching entry. If one is found, the new request (secondary miss) is merged with the request tracked by the MSHR and, knowing that the cache line has already been requested from a higher cache level, simply waits for the cache line to be fetched. In this way, the number of accesses to higher cache levels is reduced. Yet, the MSHR must store the information to fill the data requested by secondary misses in the destination register.

3) Address resolution using MSHR IDs. Most interconnection protocols, such as AXI [24], in the event of a cache miss, append an identifier (ID) to the request. This ID associates the generated transaction with an MSHR entry that holds the missing address. The transaction ID is also attached to the response, allowing the MSHR to be accessed directly (without requiring an associative search). This approach enables faster and more efficient access to MSHR, an important decision when large MSHRs are used.

4) Coherence support. Cache coherence protocols use transient states and count the number of pending acknowledgments to track the status of protocol transactions [25]. When a

request triggers an invalidation or broadcast message (e.g., when requesting write permission), all acknowledgments need to be collected before resolving the cache miss. The MSHR tracks the transient states and acknowledgment counts.

5) Writeback buffer support. Cache evictions (either dirty or noisy clean evictions [26]) are also involved in coherence management, since they often require confirmation for higher cache levels. Evictions can use the same MSHR structure as cache misses (e.g., the Gem5 [27] protocol implementation), or a different one. In this work, we assume that cache evictions use a separate writeback buffer as implemented in commercial processors [28].

C. Hardware prefetching techniques

Prefetching techniques play a pivotal role in hiding the main memory latency. However, prefetch requests also allocate MSHR entries, which makes the efficiency and scalability of prefetching fundamentally constrained by the number of MSHRs. Processors employ a range of prefetching mechanisms, including RFO prefetching and hardware prefetching, discussed below.

Hardware prefetchers observe memory accesses and proactively fill the data cache to hide the main memory latency. At the L1D level, recent academic proposals include Berti [22] and MLOP [29]. These designs are typically very aggressive, but still achieve high coverage and accuracy. Our evaluation analyzes them. For L2, Pythia [30] represents the state of the art, also providing high coverage and accuracy. We use Pythia in our baseline configuration, and it is used in all our experiments. Contemporaneous processors implement read-for-ownership (RFO) prefetch as proposed by Gharachorloo *et al.* [10]. This read-for-ownership, or read-exclusive prefetch, can be initiated for the target cache line when stores execute [10] or when stores commit [31]. In this work, we model the latter alternative: store prefetch on-commit. RFO prefetches the cache line into L1D with write permission, so that by the time the store reaches the head of the store buffer (SB), it can complete immediately. More importantly, cache hits allow writes to be pipelined, offering high write throughput even when the consistency model enforces TSO. However, if the cache line is not present in L1D with write permission, e.g., the RFO was dropped due to a full MSHR, the SB head stalls until ownership and data arrive, and younger stores following TSO need to restart the write [20].

D. Handling requests when MSHRs are full

Traditional MSHRs do not process new cache misses when all of their entries are occupied. Fig. 1 explains what happens to such misses at L1D. Demand requests are commonly placed in a retry queue (RQ) (1) [9], [32], [33], which has the highest priority to access the cache. As soon as an MSHR entry becomes available, requests from the retry queue are the first to be serviced (2). RFO requests from the processor (1) are simply discarded when the MSHR is full (2). For the hardware prefetcher, the prefetch queue (PQ) stalls issuing new requests until an MSHR entry becomes available (1). However,

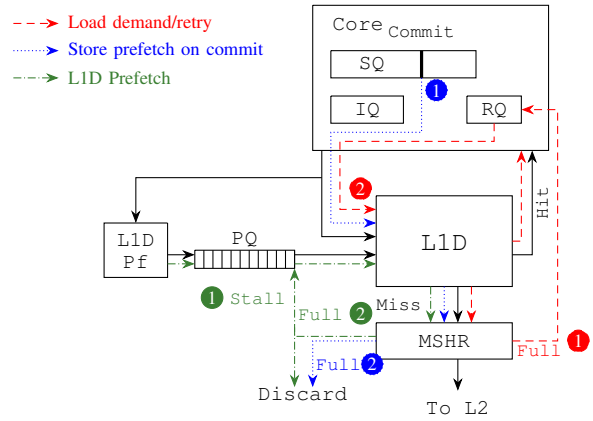


Fig. 1. Overview of actions taken when the MSHR is full

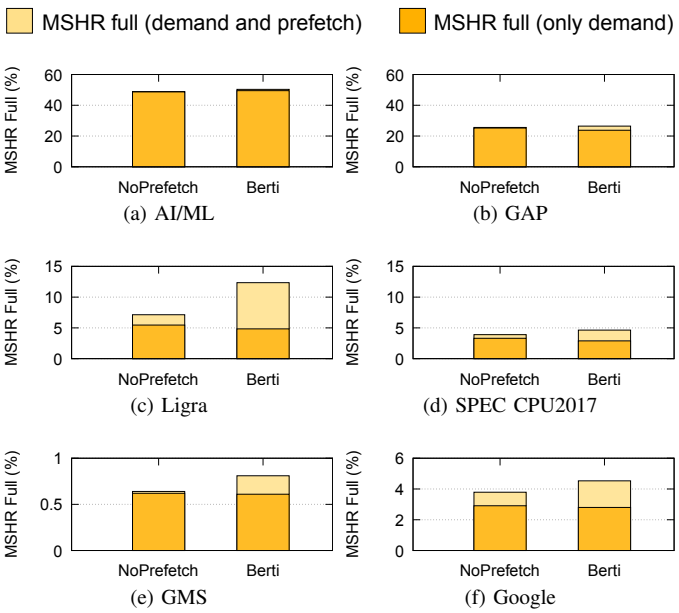


Fig. 2. Percentage of cycles in which all 16 L1D MSHR entries are occupied with no L1D prefetcher and with Berti prefetcher at the L1D. Light bars show total MSHR-full cycles; dark bars show MSHR-full cycles with only demands.

because the cache port is pipelined, a prefetch request may already be in flight when the MSHR becomes full. In such cases, if the cache access results in a miss and no MSHR is available at that moment, the prefetch request is dropped (2). A similar resource constraint can arise at the L2. Requests that require a new L2 MSHR cannot proceed beyond the private L2 until an L2 MSHR entry is free. When the L1D-to-L2 path cannot accept more requests, additional L1D misses stop issuing requests to the L2, and the L1D-side behavior follows the policy illustrated in Fig. 1. The L2 prefetch queue also stops issuing prefetches while no entry is available.

III. Motivation

A. MSHR pressure as a limiting factor for MLP

L1D MSHRs are a scarce resource in modern CPUs, placing a limit on the achievable MLP of the core. During program execution, a significant fraction of cycles is spent with all L1D

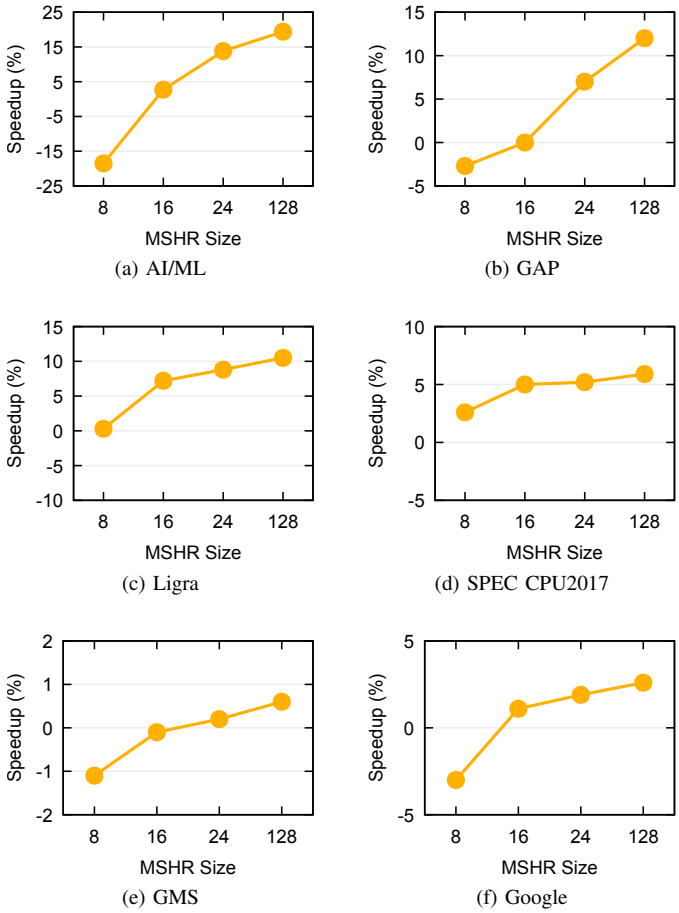


Fig. 3. Performance sensitivity to L1D MSHR size with Berti at L1D, normalized to a 16-entry Standard-MSHR and no L1D prefetcher

MSHRs occupied. During those intervals, demand requests that miss in cache are delayed, and prefetch requests are either stalled, dropped, or demoted depending on the prefetching policy. At this point, no further cache line can be fetched. Fig. 2 shows the fraction of cycles during which all entries of a 16-entry L1D MSHR are occupied for NoPrefetch and Berti (see Section VI for methodology details). Without prefetching, AI/ML and GAP saturate the MSHR for 48.9% and 25.6% of cycles, respectively, rising to 50.3% and 26.5% with Berti. Fig. 3 shows that this pressure directly limits performance. With Berti at L1D, scaling the Standard-MSHR from 16 to 128 entries improves performance from 2.7% to 19.4% on AI/ML and from 0.0% to 12.0% on GAP. These results show that increasing the number of MSHRs provides substantial performance gains.

B. Why not just increase the MSHR size?

Increasing the number of MSHRs to handle more outstanding L1D misses might seem straightforward, but this approach presents significant challenges, as acknowledged in a recent patent from AMD [5]. Adding more MSHRs significantly increases the area cost [2]–[4], [6] due to the complex hardware required. Even if designers are willing to accept this trade-off for potential performance gains, more pressing constraints like bandwidth and latency are not easily mitigated. MSHRs are typically implemented using content-addressable memory (CAM) to facilitate associative lookups [5], [6] on each cache miss. Although substantial research has aimed to enhance the scalability of these structures [34]–[36], expanding the CAM capacity remains challenging [6], [37]–[39] due to extra area and increased power consumption and multiporting, thus limiting the MLP [6]. These factors, combined with the stringent low-latency requirements of modern L1D caches, severely limit the size of high-performance MSHRs.

These limitations are evident in modern CPUs. Intel architectures provide between 10 and 16 MSHR entries, 10 in Haswell, 12 in Skylake, and up to 16 in Golden Cove [31], [40]–[42]— while AMD's Zen 4 architecture offers up to 24 MSHR entries [43]. Modern processors, on the other hand, manage MSHR saturation through alternative methods. For example, the OpenXiangShan processor [9] reserves a limited number of MSHRs to allocate prefetches. In the default configuration, 6 out of 16 to prevent prefetches from monopolizing MSHR resources. Alternatively, when MSHRs become full, processors redirect prefetches to higher cache levels [11], [16]–[18] or discard them altogether [31], as detailed in Section IV. These methods, combined with the modest number of MSHRs in current CPUs and the new advances in prefetching, highlight the need for new MSHR techniques focused on prefetch requests.

IV. MSHR Management in Commercial CPUs

Through examination of patent filings from AMD [11], Qualcomm [12], and Samsung [13], [14], as well as vendor BIOS configuration guides [16]–[18], we identify several techniques that redirect requests to a higher cache level when L1D MSHR entries are exhausted.

AMD redirects prefetch requests to L2 based on MSHR occupancy and prefetch confidence [11]. Samsung describes a prefetch scheduling mechanism that dynamically selects the target cache level for prefetch requests based on MSHR occupancy and other system resources [13], and a two-pass prefetch mode in the Exynos microarchitecture [14] in which the prefetcher first sends a request to L2 without allocating an L1D MSHR entry, and issues a second-pass fill to L1D once an entry becomes available. Qualcomm describes converting an L1D demand miss into a prefetch-style request for the next cache level when L1 resources are unavailable [12]. Intel server platforms expose a Homeless Prefetch configuration option [16]–[18]. To the best of our knowledge, these mechanisms have not been previously described or analyzed in the academic literature. We focus on Intel's Homeless Prefetch as a commercial reference mechanism because it is a simple technique available as a configuration option on Intel server platforms.

A. Uncovering the Homeless Prefetch

No official Intel documentation describes the internal implementation of the Homeless Prefetch. Three vendor-authored SPEC CPU2017 flag descriptions (Nettrix [15], Compal [44],

and Fujitsu [45]) provide the most detailed publicly available descriptions of its behavior. We base our model on these sources.

1) Prefetch demotion: All three descriptions consistently report the following behavior. When all L1D MSHR entries are occupied and a hardware prefetch request misses in L1D, the request bypasses the L1D MSHR (no L1D MSHR entry is consumed) and is sent directly to L2. The L2 cache processes the request as usual (fetching it in case of a miss), but without sending a response to L1D to fill the line [15]. A subsequent demand access to the same address will miss in L1D but it will hit in L2, reducing latency.

Processor read-for-ownership (RFO) requests [10], [31] face the same MSHR constraints as hardware prefetches. Although none of the vendor descriptions explicitly addresses RFO handling, we apply the same demotion policy: when the L1D MSHR is full, RFO requests are sent to L2 rather than being dropped.

Demand requests that miss in L1D and find the MSHR full enter the retry queue [9], [32], [33] and are rescheduled with highest priority. When an MSHR entry becomes available, the retry queue drains first, ahead of new requests. The demand request allocates an MSHR entry, is issued to L2, and the returned cache line fills L1D.

2) Demand-to-prefetch (D2P) demotion: One of the three vendor descriptions [15] additionally describes that Homeless Prefetch allows early fetch of demand misses into the mid-level cache when resources are insufficient to track the demand in L1. A Qualcomm patent [12] describes a similar mechanism where L1D demand misses are converted into prefetch requests for the next cache level.

We implement this as an extended variant, Homeless-D2P, in which, when a demand load misses in L1D and all MSHR entries are occupied, the demand is issued to L2 as a prefetch-style request without allocating an L1D MSHR entry. The cache line is fetched into L2 (but not into L1D). The original demand remains in the retry queue and is rescheduled when an MSHR entry becomes available. Upon retry, the demand misses in L1D but hits in L2, reducing its miss latency. If the retry finds the MSHR full again, no additional prefetch requests are issued to avoid duplicate traffic.

However, these techniques still maintain the conventional assumption that prefetch misses in L1D require full L1D MSHR support. As a result, on full MSHR occupation, prefetch requests are demoted to fill L2 rather than L1D. In addition, demand misses cannot reclaim L1D MSHR entries occupied by prefetch requests. These two limitations motivate Nomad, which re-examines the need for L1D MSHR support for prefetch misses. This enables prefetching to L1D even when the L1D MSHR is full and allows incoming demand misses to reclaim L1D MSHR entries occupied by prefetch requests.

V. Nomad

Our key observation is that, unlike demand requests that miss in cache, prefetch requests do not require precise tracking of destination registers or cache line offsets. This distinction opens the door to new opportunities based on the relaxation of the conventional allocation rules in the L1D MSHR structure imposed by prefetching. Our proposal, Nomad, enables more flexible handling of prefetch requests by re-evaluating their need for L1D MSHR resources (Section V-A). With Nomad, prefetch requests can bypass L1D MSHR allocation when it is full *while still fetching the cache line into L1D* (Section V-B). In addition, a demand miss can reclaim an L1D MSHR entry currently used by a prefetch without any impact on the prefetched cache line (Section V-C). Finally, if all L1D MSHRs are occupied by demand requests that miss in cache, non-tracking prefetch requests for the same cache line can be used instead, but allowing to fill the L1D with that cache line (Section V-D).

A. Overcoming L1D MSHR functionality for prefetch requests with Nomad.

This section elaborates on how prefetch requests that miss in L1D can be handled without allocating L1D MSHR entries, revisiting each of the functionalities described in Section II-B. In this way, prefetch requests do not need to be tracked using L1D MSHRs, which increases the potential of prefetching and allows demand accesses to find more free MSHR entries. We assume that with Nomad, a prefetch miss may or may not be allocated in the L1D MSHR. We also assume that the large L2 is inclusive with respect to the small L1D, as in recent processors [46].

1) Resolving memory operations missing in cache. A prefetch action ends when the requested line is allocated in cache. Unlike demand requests that miss in cache, prefetch misses do not need to be linked to memory operations in the processor, thus not requiring keeping track of the information necessary to correctly fill the corresponding register.

2) Coalescing. Coalescing cannot be performed for prefetch requests that are not tracked by the MSHR. Hence, duplicate requests (for the same cache line) can be generated. Apart from extra bandwidth requirements, duplicate requests can be an additional source of complexity for the already complex coherence protocol. Our solution is not to allow duplicate requests past the private L2, where the actual complexity of the coherence protocol resides. As a consequence, Nomad is enabled only at the L1D MSHRs, while the L2 MSHRs still track prefetch requests, filtering duplicate coherence transactions. We leave L2 MSHR optimizations for future work.

3) Address resolution using MSHR IDs. In conventional designs, cache fills are matched to MSHR entries using the MSHR ID carried by the response. In Nomad, some prefetch requests may reach L2 without an L1D MSHR ID, either because they bypassed the L1D MSHR or because their entry was later reclaimed by a demand miss. Therefore, responses from L2 to L1D to prefetch transactions *must* include the cache line address. Upon a prefetch response, the MSHR is associatively searched to identify a matching entry, if any. Fortunately, with Nomad, a small MSHR can perform as well as very large ones, making this search cheap and fast. Since

the L2 MSHR tracks all prefetch requests and an MSHR ID can be used from that point on, no changes are needed beyond the L2.

4) Coherence support. Acknowledgment counting is managed by the L2 MSHR, and when all acknowledgments are collected, the L2 informs the L1D about the permission that has been obtained. Similarly, transient states are not necessary for prefetch requests generated at the L1D, even when prefetchers can request either read-only or read-write permissions. Usually, cache coherence protocols convey in the data message the permission for the provided cache line (e.g., via `data` or `data_exclusive` messages) [25].

5) Writeback buffer support. Cache evictions requiring coherence actions always allocate an entry in the writeback buffer so they do not allocate MSHR entries.

6) Selecting the destination way. Prefetches do not reserve a destination way on a cache miss; the way is chosen when the line returns. When a prefetch response reaches the L1D, it checks for a pending demand miss to the same line in the MSHR. If one exists, the response fills through the normal demand path. Otherwise, it calls the replacement algorithm to select a victim way, writes the victim back if it is dirty, and allocates the line. If no way is available, or the writeback cannot be performed, the returned line is discarded.

B. Bypassing the L1D MSHR while filling L1D

In the academic literature, when all L1D MSHR entries are occupied, demand requests that miss in cache are scheduled for retry, RFO requests are dropped, and the prefetch queue stalls, dropping prefetch requests that are already in flight when they miss in L1D (Section II-D). Alternatively, in industrial solutions, the L1D MSHR allocation can be bypassed at the cost of not fetching the cache line into L1D (in the best scenario into L2), a suboptimal solution.

In contrast, with Nomad, prefetch requests that miss in L1D do not allocate an MSHR entry when it is full, but after being sent to L2, they return to L1D where they are inserted, thanks to the different requirements of prefetch requests described in the previous section. In this way, RFO requests can continue fetching cache lines with read-write permission to L1D, allowing an efficient cache write pipeline under TSO. Similarly, the prefetch queue can continue issuing requests to L2 on cache misses, bringing cache lines to L1D.

C. Prefetch allocation and eviction

Although Nomad does not require the allocation of L1D MSHR entries for prefetch misses, doing so when MSHR entries are available is beneficial. Allocating an entry allows demand and prefetch requests to the same cache line to be coalesced, which filters redundant requests to L2 and saves traffic.

However, when a demand request is missing in the L1D and finds the MSHR full but with some entries occupied by prefetch requests, the demand request can immediately reclaim a slot held by a prefetch. In such cases, the first MSHR entry occupied by a prefetch request is chosen for eviction. From this point on, the MSHR entry cannot be evicted, as it is considered a *demand* entry. Reclaiming a slot only removes the L1D MSHR tracking for the prefetch; the higher-level memory request remains in progress, so the cache line will still be stored in L1D when the miss is resolved. Recall that prefetch responses always need to search the MSHR for a match, which ensures that the demand entry is identified so it can resolve the memory operation that generated the request. This prefetch eviction policy ensures in-flight prefetches can still provide benefit, even if their tracking is lost before completion.

TABLE I
L1D-L2 BUS WIDTH (IN BITS) FOR ARM CHI

Channel	Direction	Purpose	Base	Nomad
TXREQ	L1D → interconnect	Send requests	178	178
TXDAT	L1D → interconnect	Send data	374	374
TXRSP	L1D → interconnect	Send responses	77	77
RXDAT	interconnect → L1D	Receive data	374	438
RXRSP	interconnect → L1D	Receive responses	77	77
RXSNP	interconnect → L1D	Receive snoops	114	114
Total	–	–	**1194**	**1258**

D. Upgrading demand-to-prefetch

When a demand request needs an L1D MSHR entry and all are occupied by other demand requests, meaning that no prefetch entry can be reclaimed (Section V-C), the request is rescheduled in the retry queue, where it waits until an entry becomes available. An alternative, employed in industry [12], [15], is to perform a demand-to-prefetch (D2P). Specifically, in Homeless-D2P, the cache line is placed in L2, so the demand must still wait for an MSHR entry and, upon retry, pay the L2 access latency.

Under Nomad, the bypass mechanism described in Section V-B enables D2P to fill L1D directly. The original demand can then complete as a cache hit upon retry. As in Homeless-D2P, demand retries never send additional prefetch requests.

E. Hardware implementation

Nomad is implemented with minimal hardware overhead, relying on simple extensions and additional logic. Importantly, Nomad does not require extra storage. The bypass MSHR mechanism needs to widen the bus for the L2-to-L1D connection (used exclusively by prefetch requests) for the physical address to be sent to L1D (e.g., 46 bits in Intel [47]). The logic for associative searches in the MSHR on the arrival of a prefetch request is borrowed from the coalescing mechanism. Evicting MSHRs requires a priority encoder to select the first entry occupied by a prefetch request. The demand-to-prefetch mechanism simply leverages the previous schemes without requiring further modifications.

Power and area overhead. We use ARM CHI [48] as an example to estimate area. ARM CHI uses six channels between the L1D and the L2, whose bit widths are calculated in Table I using the OpenXiangShan CHI interface as reference [9].

Nomad widens only the L2-to-L1D data-response[1] channel, leaving the other channels unchanged. As shown in Table I, this increases the local L1D interface width by only 64 bits, from 1194 to 1258 bits, corresponding to a 5.3% area overhead. Note that Nomad requires the extra 64 bits only for L2-to-L1D responses and not for the requests from L1D to L2. On average across all workloads, the L1D miss rate is 8.56% without prefetching. So, for these 8.56% of loads, Nomad incurs an average power overhead of 0.45%. In the presence of the Berti L1D prefetcher, the power overhead becomes 0.94%. These overheads scale linearly, as the L2-to-L1D response channel is private to each core because both L1D and L2 are private caches.

F. Non-inclusive cache hierarchies

Nomad uses an inclusive L2 with respect to L1D, as in most modern CPUs from AMD, Intel, and ARM [43], [46], [49]. Since Nomad operates at the L1D MSHRs and the L2 handles all requests conventionally from that point onward, the inclusivity policy between L2 and LLC does not affect Nomad. If L1D and L2 are not inclusive, Nomad is not affected, as the coherence directory remains inclusive even in non-inclusive hierarchies [50] and all incoming coherence messages go through the L2 before reaching L1D.

G. Nomad versus Homeless

Although Homeless (see Section IV) and Nomad target the case where the L1D MSHR is full, they differ in three important ways.

1) Handling of prefetches and RFOs. In Homeless, when a hardware prefetch or an RFO cannot allocate an L1D MSHR entry, the request is sent to L2 and the cache line is allocated there, but not in L1D (Section IV-A1). A subsequent demand access to that line therefore still misses in L1D and pays the L2 access latency. This behavior is particularly undesirable for RFOs under TSO consistency model [19], where timely L1D write hits are important for pipelining stores efficiently [20]. In contrast, Nomad allows prefetches and RFOs to bypass L1D MSHR allocation while still filling L1D.

2) Reclaiming L1D MSHR entries. Homeless does not reclaim MSHR entries occupied by prefetch requests by incoming demand misses. As a result, when the MSHR is full, a demand miss must wait in the retry queue until an entry is released. Nomad removes this restriction by allowing a demand miss to reclaim an entry held by a prefetch request without interrupting the request.

3) Demand-to-prefetch demotion. In the extended Homeless-D2P model (Section IV-A2), a demand miss that arrives when the L1D MSHR is full can be issued to L2 as a prefetch-style request without allocating an L1D MSHR entry. However, the line is placed in L2, not in L1D. The original demand must still wait for an MSHR entry, retry, miss in L1D, and pay the L2 round-trip latency. Nomad's D2P, described in Section V-D, differs in that the prefetch-style request fills L1D directly. When the demand retries, it can complete as an L1D hit rather than paying an additional L2 access.

[1]For the data channels, a 64-byte cache line is transferred over two cycles, corresponding to a 256-bit data path; the remaining bits correspond to protocol metadata.

TABLE II
SIMULATION PARAMETERS OF THE BASELINE SYSTEM

Core	
Front-end	TAGE-SC-L branch predictor [51], 6-wide fetch and decode, 144 decode queue, 205 issue queue
Back-end	ROB/LQ/SB sizes 512/192/114, 12 execution ports, 8 width commit, store prefetch on-commit [10]
Memory hierarchy	
L1I	32 KB, 8-way, 4 hit cycles, 2 read ports, 16 MSHRs
L1D	48 KB, 12-way, 5 hit cycles, 3 read ports, 2 write ports, 32-entry retry queue, 16 MSHRs, 16-entry prefetch queue, No-prefetch
L2	1.25 MB, 10-way, 15-cycle round-trip latency, 32 MSHRs, Pythia prefetcher
LLC	3 MB per core, 12-way, 60-cycle round-trip latency, 64 MSHRs
Memory	Mess simulator, DDR4-2666, 21.33GB/s per channel
Coherence	MESI directory-based protocol

TABLE III
CONFIGURATIONS OF EVALUATED PREFETCHERS

Prefetcher	Level	Parameters	Size
IP-Stride	L1D	1024 entries, degree three	9.4 KB
MLOP	L1D	256-entry AMT, 500-update, 16-degree	8.47 KB
Berti	L1D	8-set 16-way History, 64-entry 16-delta Delta, 70% MSHR Threshold	4.4 KB
Pythia	L2	2 active features, 15 actions	25.5 KB

VI. METHODOLOGY

Our evaluation employs the following simulation infrastructure. Each benchmark is processed by a detailed in-house cycle-level out-of-order core simulator using configuration parameters similar to a Golden Cove processor [52], and detailed in Table II. The processor models in detail the core microarchitecture as well as the L1D and MSHR, and has a prefetch on-commit mechanism for stores [31]. Memory requests by the core are injected into an SLICC-modeled memory hierarchy [53] that implements a directory-based coherence protocol that sends coherence transactions through the network modeled with GARNET [54] and connects to a DRAM modeled with Mess [55]. The energy consumption of caches and memory for reads and writes is obtained using CACTI 6.5 [56] at a 22nm technology node, and the MICRON DRAM Power Calculator [57]. The total energy consumption is computed by accounting for the number of accesses to tags and data across the whole memory hierarchy.

For our single-thread evaluation, we feed our simulator with traces released in the 4th Data Prefetching Championship (DPC4) [58], [59] for SPEC CPU2017, GAP, Ligra, GMS, Google, and AI/ML benchmarks. For each trace, we warm up for 50M instructions and collect statistics for 200M instructions. For SPEC, we calculate the statistics for each application using their SimPoint [60] weights. For the other suites, missing such weights, we assign equal weights for each

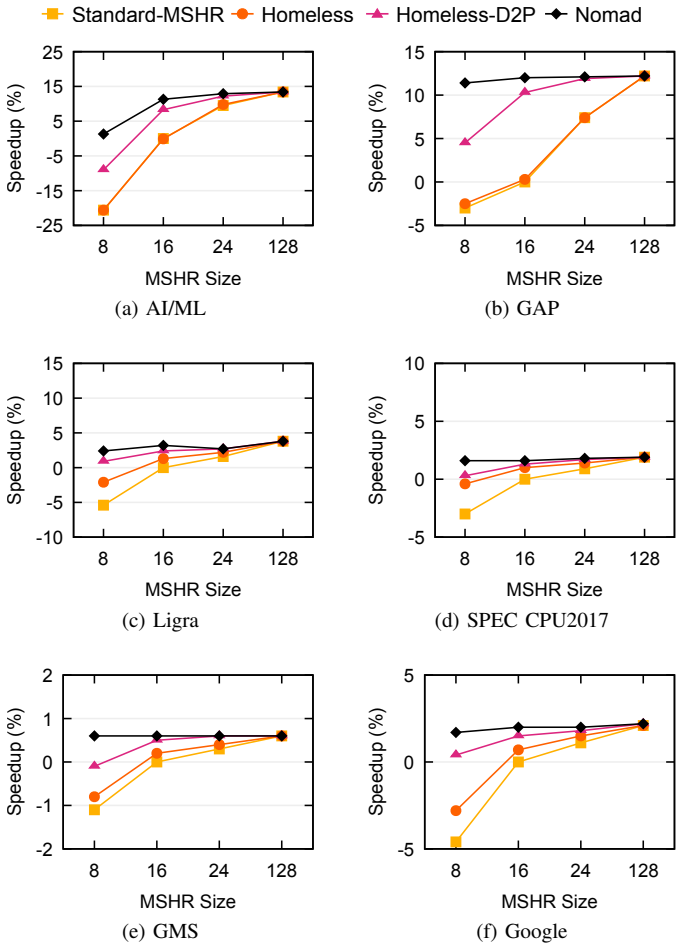


Fig. 4. Performance sensitivity to L1D MSHR size without an L1D hardware prefetcher normalized to a 16-entry Standard-MSHR and no L1D prefetcher

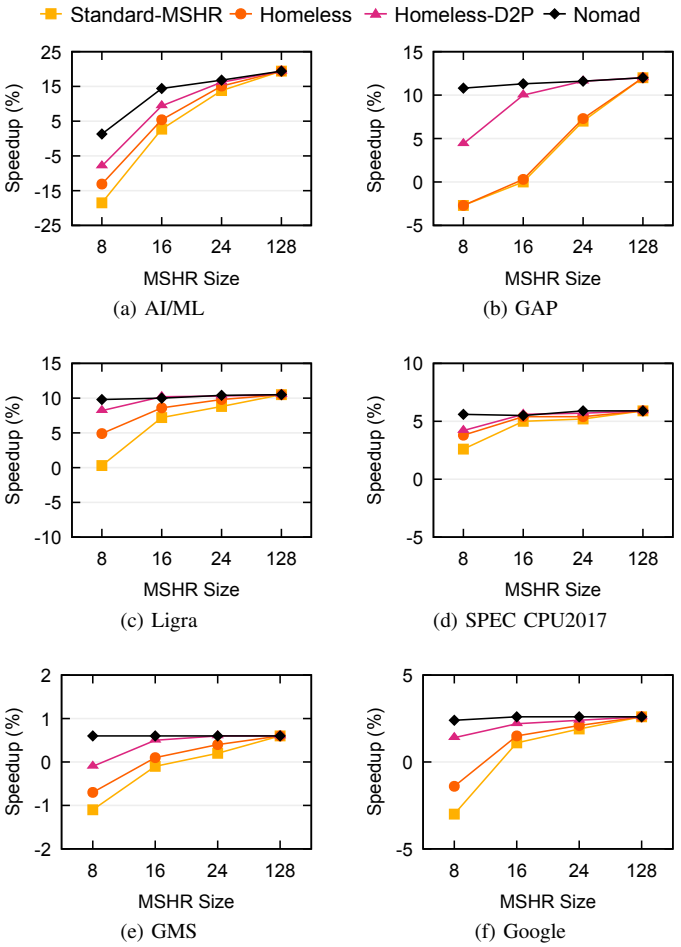


Fig. 5. Performance sensitivity to L1D MSHR size with Berti at L1D, normalized to a 16-entry Standard-MSHR and no L1D prefetcher.

interval. For our multiprogrammed evaluation, we construct 50 random four-core mixes from the DPC4 single-thread traces. For each mix, we report the normalized harmonic mean of speedups.

For our multi-thread evaluation, we run all SPLASH-4 benchmarks [61] using simmedium inputs and 16 threads. The simulator is fed with instructions provided by Sniper [62]. We collect statistics for the whole parallel region. For all benchmark suites, we use the geometric mean to average the speedups. Our study incorporates both memory-intensive and non-memory-intensive workloads.

We compare Nomad against two different versions of the Homeless prefetcher: the base Homeless configuration, where L1D prefetch requests and processor RFO requests are demoted to fill L2 when the MSHR is full; and an optimized Homeless-D2P configuration, where demand requests that miss in cache may also be issued to L2 as prefetch requests. Both versions are configured to achieve the best possible performance for that technique. Our baseline is a Standard-MSHR configuration where all L1D misses are stalled if there is no space available in the MSHR.

We analyze Nomad using four L1D prefetching configurations: NoPrefetch, IP-Stride, MLOP, and Berti. The prefetchers insert requests into the PQ, which drains when there are L1D read ports not used by demand requests during the current cycle. Table III details their configurations. Every prefetcher is individually fine-tuned for optimal performance using a baseline 16-entry L1D MSHR. Also, Berti needs to be adjusted by training with zero latency for bypassed or discarded prefetch requests, establishing a lower performance bound, as it uses an extended MSHR to track prefetch latency. Pythia is always used in our experiments as the L2 prefetcher, and RFOs, described in Section II-C, remain enabled in all configurations.

VII. Evaluation

Our key evaluation results are summarized in Table IV. Nomad outperforms the Standard-MSHR and Homeless on all six benchmark suites without additional storage. Nomad also outperforms Homeless-D2P, improving IPC by 14.4% on AI/ML (vs. 9.4% for Homeless-D2P) and by 11.3% on GAP (vs. 10.0%). On the remaining suites, Homeless-D2P and Nomad provide similar performance. Homeless-D2P has slightly higher speedup on SPEC CPU2017 and Ligra, while Nomad has slightly higher speedup on GMS and Google. With 16 entries, Nomad recovers 74% of the 128-entry performance ceiling on AI/ML and 94% on GAP.

TABLE IV
EVALUATION SUMMARY WITH BERTI AT L1D, 16 L1D MSHR ENTRIES. MSHR FULL IS THE FRACTION OF CYCLES WITH ALL 16 STANDARD-MSHR ENTRIES OCCUPIED. SPEEDUP AND ENERGY ARE NORMALIZED TO A 16-ENTRY STANDARD-MSHR AND NO L1D PREFETCHER.

	MSHR	**Speedup (%)**					**Dynamic energy overhead in (%)**			
Suite	Full (%)	Standard	Homeless	Homeless-D2P	**Nomad**	128-e MSHR	Standard	Homeless	Homeless-D2P	**Nomad**
AI/ML	50.3	2.7	5.4	9.4	**14.4**	19.4	1.0	1.6	3.9	**3.4**
GAP	26.5	0.0	0.3	10.0	**11.3**	12.0	7.1	9.2	9.5	**9.3**
Ligra	12.4	7.2	8.6	10.2	**10.0**	10.5	1.7	2.3	2.0	**2.3**
SPEC	4.6	5.0	5.4	5.6	**5.5**	5.9	5.8	6.2	6.4	**7.1**
GMS	0.8	−0.1	0.1	0.5	**0.6**	0.6	27.8	29.8	29.2	**29.5**
Google	4.5	1.1	1.5	2.2	**2.6**	2.6	3.5	3.9	2.8	**1.8**

The gains come from two complementary sources whose relative weight varies with MSHR pressure and prefetch coverage. On SPEC CPU2017 and Ligra, bypass and eviction keep prefetch requests flowing into L1D even on a full MSHR, recovering most of the performance lost due to MSHR stalls. On GAP, where Berti's L1D prefetch coverage on the most MSHR-intensive workloads is low, D2P accounts for nearly all of the improvement by converting blocked demands into L1D-filling prefetch requests. AI/ML benefits from both mechanisms, with bypass and eviction raising speedup from 2.7% to 5.8% and D2P raising it further to 14.4%.

Performance improvements come at a modest energy cost. Nomad's dynamic energy remains comparable to Homeless-D2P across all suites, and slightly lower on AI/ML (3.4% vs. 3.9%) and GAP (9.3% vs. 9.5%). L1D-to-L2 traffic increases as Nomad issues requests that the Standard-MSHR would have stalled or dropped, but LLC-to-DRAM traffic remains similar across all configurations (Section VII-H).

A. Performance vs. L1D MSHR Size

Figs. 4 and 5 show performance as the number of L1D MSHR entries varies from 8 to 128, without L1D prefetching (Fig. 4) and with Berti (Fig. 5). Results are normalized to a 16-entry Standard-MSHR without L1D prefetching. In both settings, performance improves as the L1D MSHR budget increases, and Nomad recovers a large fraction of that performance with substantially fewer entries. Without an L1D prefetcher, the largest gains appear on the suites with the highest L1D MSHR pressure. At 16 entries, Nomad reaches 11.3% on AI/ML and 12.0% on GAP. With only 8 entries, Nomad outperforms the 16-entry Standard-MSHR on all six suites. These gains arise because, once the L1D MSHR is full, the baseline cannot process new misses, whereas Nomad allows blocked demand misses, as prefetch-style requests, and RFOs, bypassing the L1D MSHR, to make forward progress.

With Berti at L1D, the same trend remains and pressure on the L1D MSHR increases because demand misses and prefetch requests compete for the same entries. Nomad allows the prefetcher to continue issuing prefetch requests when MSHR are full and the largest gains again appear on AI/ML and GAP, where Nomad reaches 14.4% and 11.3%, respectively, at 16 entries. Berti's accuracy remains almost unaffected, with a maximum drop from 86.2% to 84.7% in Google. Across all suites, Nomad delivers accuracies in the range of 76.9% to 98.0%.

B. Per-Application Analysis

Fig. 6 breaks down the per-application speedup with Berti and 16 L1D MSHR entries. The largest gains appear on workloads that saturate the MSHR for a large fraction of execution. On AI/ML, `llama2-7b` reaches 3.8% with the Standard-MSHR, relative to the 16-entry Standard-MSHR without an L1D prefetcher. Bypass allows 49 prefetch requests per kilo-instructions (PKI) to be issued when the L1D MSHR is full, raising performance to 6.7%. Eviction reclaims 2.9 L1D MSHR entries PKI from prefetch-occupied slots, reaching 7.5%. D2P converts 128.5 demand misses PKI into prefetch-style requests that fill L1D without MSHR allocation, reaching 16.8%. On GAP, the PageRank benchmarks show negligible speedup with Standard-MSHR and Berti. Berti's L1D cache prefetch coverage on this workload is only 1.3%, so bypass and eviction provide negligible improvement. D2P turns 180.9 PKI demand misses into L1D-filling prefetch-style requests, reaching 43%.

The largest regression is `607.cactuBSSN` (−5.2% with Nomad vs. −0.1% with Standard-MSHR at 16 entries). This benchmark exhibits interleaved access patterns whose reuse distances exceed the cache residency time. Nomad gives the prefetcher more room to issue requests, but those requests arrive too early and are evicted before demand, turning the additional prefetch traffic into cache pollution. This behavior has been already reported previously [63], [64] and is not specific to Nomad. For instance, with the Standard-MSHR, performance degrades as entries increase (−0.1% at 16, −3.8% at 24, −4.8% at 128). This trend is consistent with Berti's accuracy on the problematic simpoint `607.cactuBSSN_s-2421B`, where accuracy drops from 0.48% at 16 entries to 0.14% at 24 and 0.12% at 128.

C. Effect of Prefetch Throttling

Berti selects the target cache level for each prefetch request based on per-delta confidence, and includes an L1D MSHR-occupancy-based throttling mechanism that redirects all L1D prefetch requests to L2 when 70% of entries are occupied. These mechanisms are similar to AMD's patented prefetch management strategy [11]. Our Standard-MSHR baseline includes this throttling mechanism. To isolate its contribution

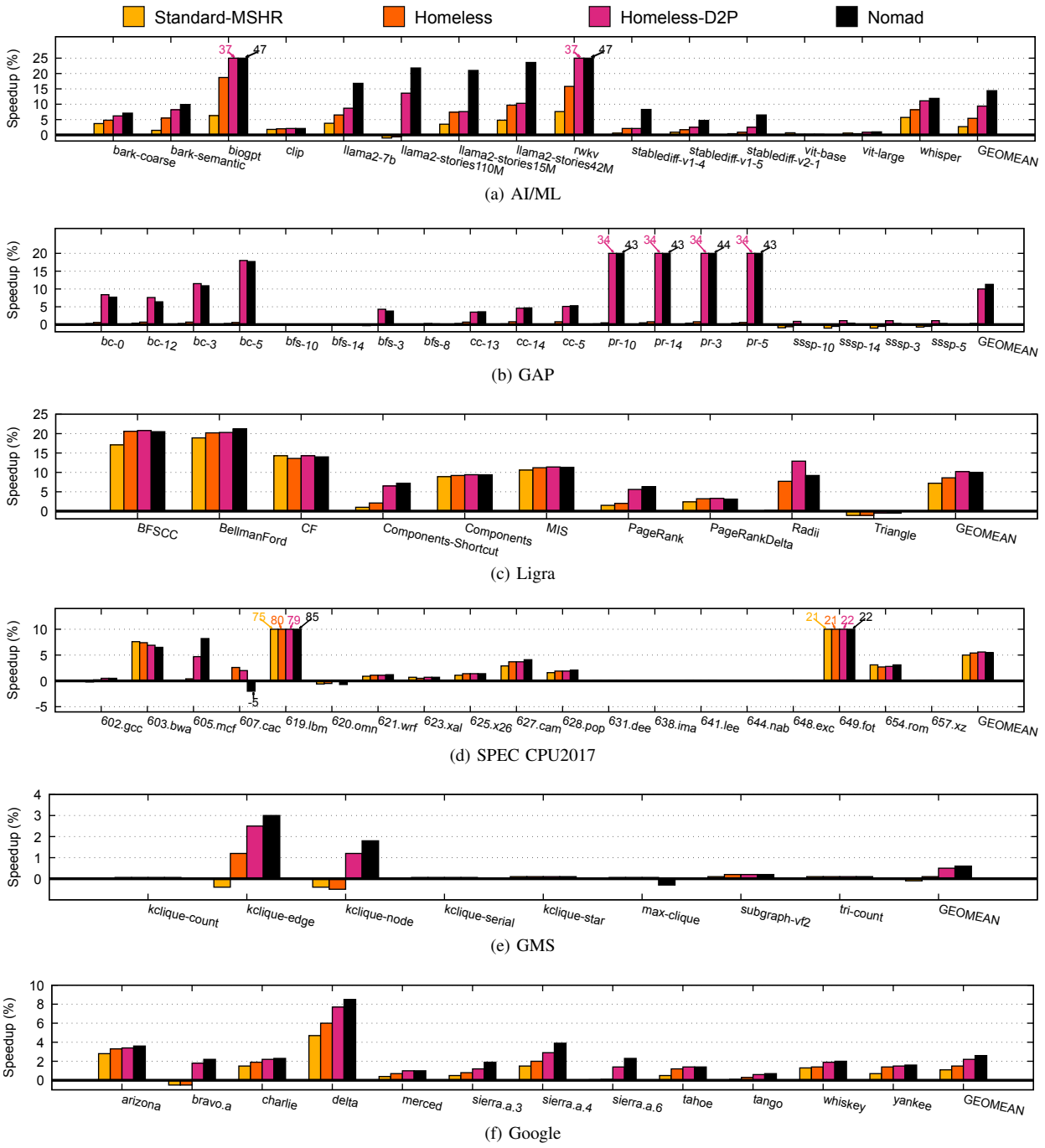


Fig. 6. Per-application speedup with Berti at L1D and a 16-entry L1D MSHR normalized to a 16-entry Standard-MSHR and no L1D prefetcher

as a throttle mechanism, we disable it and measure the impact. At 16 L1D MSHR entries, disabling the throttling mechanism slightly degrades performance on most suites. The largest drops occur on AI/ML (from 2.7% to 1.8%) and Ligra (from 7.2% to 6.2%), while SPEC CPU2017 changes only marginally (from 5.0% to 4.8%), and Google, GAP and GMS remain unchanged. This mechanism helps prevent prefetches from occupying MSHR slots by limiting new prefetch allocations as the MSHR approaches capacity. However, it can only prevent new prefetch allocations. Entries already held by prefetches remain occupied and cannot be reclaimed.

D. Comparison with Industry Techniques

Fig. 5 compares Nomad against our best-effort models of Homeless and Homeless-D2P. All three techniques improve over the Standard-MSHR, particularly on suites with high MSHR pressure. On AI/ML with 16 entries, the Standard-MSHR achieves 2.7%, Homeless raises this to 5.4%, Homeless-D2P to 9.4%, and Nomad to 14.4%.

The key distinction is where requests fill the cache hierarchy once the L1D MSHR is full. In Homeless and Homeless-D2P, requests that bypass L1D MSHR allocation are demoted to fill L2 rather than L1D, whereas Nomad enables those requests

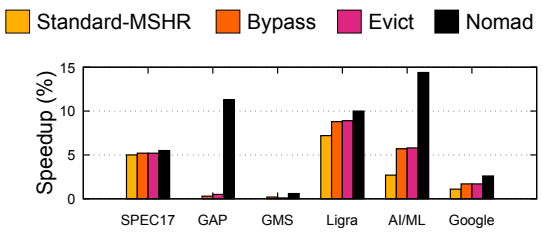


Fig. 7. Ablation of Nomad components with Berti at L1D and 16 L1D MSHR entries

to fill L1D without L1D MSHR allocation. This difference is reflected in L1D demand MPKI and Nomad is the only mechanism that reduces it. With Berti and 16 entries, Nomad reduces L1D demand MPKI from 113.8 to 105.6 on AI/ML (7.2%) and from 82.9 to 78.3 on GAP (5.5%) relative to the Standard-MSHR, with per-workload reductions reaching 28% on biogpt and 27% on rwkv. Homeless and Homeless-D2P leave L1D demand MPKI within 1% of the Standard-MSHR on these two workloads, as their fills are placed in L2. As Nomad preserves L1D placement even when no MSHR entry is available, it also tolerates smaller L1D MSHR budgets. With eight entries and Berti, Nomad matches or comes close to the performance of a 16-entry Homeless-D2P on SPEC CPU2017, GAP, Google, GMS, and Ligra. AI/ML is the exception, where the remaining pressure is high enough that eight entries still leave substantial performance on the table.

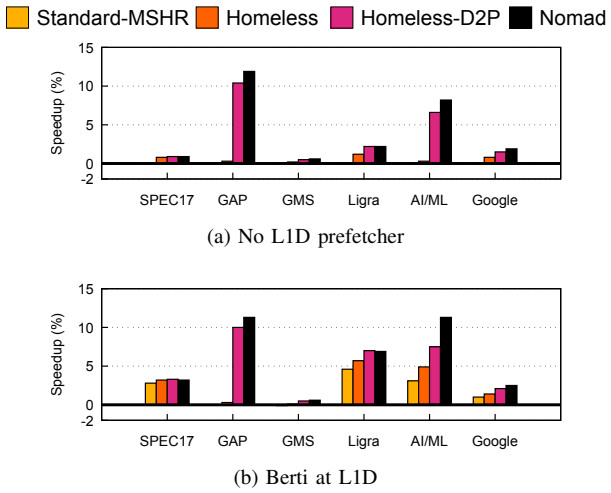
E. Ablation study

Fig. 7 isolates the contribution of each Nomad component by incrementally enabling bypass, eviction, and D2P on top of the Standard-MSHR, using Berti with 16 entries.

Bypass allows prefetch requests to proceed without allocating an MSHR entry when the MSHR is full. On AI/ML and Ligra, it bypasses, on average, 11.8 and 15.2 prefetch requests per kilo-instructions (PKI), respectively, raising performance from 2.7% to 5.7% and from 7.2% to 8.8%. Gains are smaller on the remaining suites. In GAP, Berti's L1D cache prefetch coverage is low in workloads (e.g., PageRank) with the highest MSHR pressure, while GMS, Google and SPEC CPU2017 exhibit lower overall MSHR contention (0.8%, 4.5% and 4.6% of cycles with a full MSHR vs. 50.3% in AI/ML), limiting bypass opportunities.

Eviction is built on top of bypass. When the MSHR is full, a demand miss can reclaim a slot held by a prefetch without interrupting the in-flight request. This complements the Berti watermark strategy (Section VII-C), which prevents new prefetch allocations beyond 70.0% occupancy but cannot free entries already held. Because the watermark already constrains how many entries prefetches occupy, eviction rates remain moderate, reaching 2.7 PKI on Ligra, 1.3 on GAP, and 0.4 on AI/ML. Eviction adds up to 0.2 over bypass alone on GAP and 0.1 on Ligra and AI/ML.

D2P issues the demand to L2 as a prefetch-style request that fills L1D directly when all MSHR entries are held by demand requests and no prefetch entry can be reclaimed. D2P



(a) No L1D prefetcher

(b) Berti at L1D

Fig. 8. Performance under limited DRAM bandwidth with 16 L1D MSHR entries

to L1D is the largest contributor to Nomad's performance, issuing 36.4 PKI on AI/ML and 12.5 on GAP. On AI/ML, it raises performance from 5.8% with bypass and eviction alone to 14.4%. On GAP, where Berti's prefetch coverage limits bypass and eviction gains, D2P lifts performance from 0.5% to 11.3%.

F. Limited DRAM bandwidth

Fig. 8a and Fig. 8b show the speedup under limited DRAM bandwidth (1/8 of the baseline) without L1D prefetching and with Berti, respectively. All results are relative speedup over the Standard-MSHR baseline at the same bandwidth level.

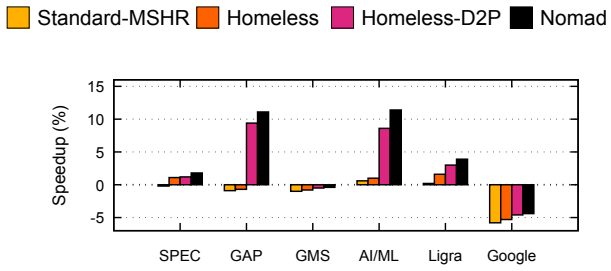
The relative advantage of Nomad and Homeless-D2P is preserved under limited bandwidth. On AI/ML without L1D prefetching, Nomad reaches 8.2% while Homeless-D2P achieves 6.6%, compared to 11.3% and 8.3% under full bandwidth, respectively. With Berti, Nomad reaches 11.3% and Homeless-D2P 7.5%, compared to 14.4% and 9.4% under full bandwidth.

G. Evaluation with Additional L1D Prefetchers

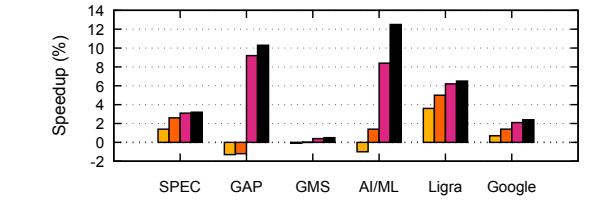
To assess whether Nomad generalizes beyond Berti, we also evaluate it with MLOP, a non-IP-based prefetcher, and IP-Stride, a simple prefetcher representative of commercial designs used by both AMD and Intel [31], [46]. Fig. 9 shows that the same trend holds across both prefetchers. On AI/ML, Nomad reaches 11.4% with MLOP, compared to 8.6% with Homeless-D2P and 0.6% with the Standard-MSHR. With IP-Stride, Nomad reaches 12.5%, compared to 8.4% with Homeless-D2P and -1.0% with the Standard-MSHR. These results show that the gains are not specific to Berti and generalize across other L1D prefetchers.

H. Traffic and Energy

Fig. 10 shows traffic per kilo-instructions at three levels of the memory hierarchy with Berti at L1D and 16-entry L1D MSHRs. Traffic from L1D to L2 increases over the Standard-MSHR baseline for all mechanisms (Fig. 10a). The

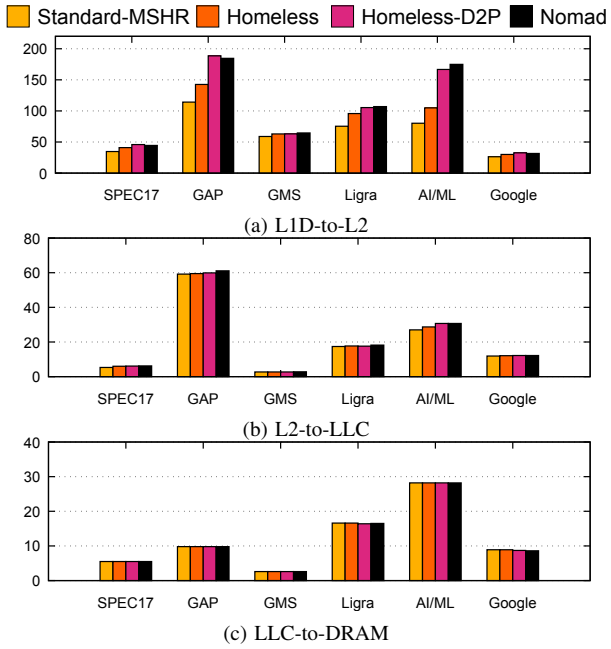


(a) MLOP



(b) IP-Stride

Fig. 9. Performance comparison with MLOP and IP-Stride and 16 L1D MSHR entries



(a) L1D-to-L2

(b) L2-to-LLC

(c) LLC-to-DRAM

Fig. 10. Traffic per kilo-instructions across the memory hierarchy with Berti and 16-entry L1D MSHRs

largest increases appear on AI/ML from 80.2 to 174.9 Accesses per kilo-instructions (APKI) and GAP from 114.2 to 184.6 APKI, which are also the suites where Nomad provides the largest performance gains. This increase is expected and has two sources. First, bypass issues prefetch requests that the Standard-MSHR baseline would have dropped or stalled. Second, because bypassed and D2P requests are not tracked by the L1D MSHR, they cannot be coalesced at L1D, producing duplicate requests for cache lines that may already be in flight. This increase in traffic is largely absorbed by the L2 cache. L2-to-LLC traffic (Fig. 10b) increases modestly, and LLC-to-DRAM traffic (Fig. 10c) remains similar across all

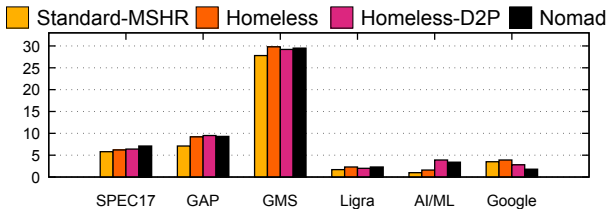
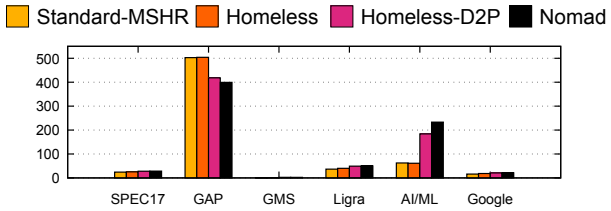
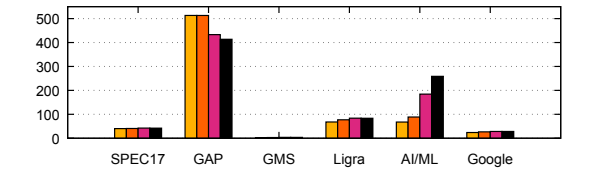


Fig. 11. Dynamic energy in the memory hierarchy normalized to a 16-entry Standard-MSHR and no L1D prefetcher



(a) No L1D prefetcher



(b) Berti at L1D

Fig. 12. L2 MSHR-full cycles per kilo-instructions

configurations and suites. This attenuation occurs because the L2 MSHR tracks all requests from L1D, including those issued without an L1D MSHR ID, and coalesces duplicate transactions before they propagate to the LLC.

The dynamic energy consumption in the memory hierarchy, normalized to the Standard-MSHR baseline is shown in Fig. 11. Across all suites, Nomad's dynamic energy remains comparable to Homeless-D2P. The largest difference over the Standard-MSHR appears on AI/ML (3.4% for Nomad, 1.0% for Standard-MSHR). For Google, Homeless-D2P and Nomad reduce dynamic energy because fewer L2-Pythia requests reach DRAM, lowering LLC-to-DRAM traffic from 8.9 APKI to 8.7 and 8.6 APKI. Our energy model accounts for the cost of the associative MSHR lookup required when a prefetch response arrives without an MSHR ID, i.e., bypassed and evicted prefetch requests (Section V-A). Importantly, this overhead is only incurred when bypass or D2P is active due to L1D MSHRs being full. In contrast, scaling the MSHR to a larger size increases the CAM area and per-access power on every cache miss. Note that workloads where Nomad reduces execution time, it also lowers static (leakage) energy.

I. L2 MSHR Pressure

Fig. 12 reports L2 MSHR-full cycles per kilo-instructions. The 32 L2 MSHR entries are shared by requests arriving from L1D and by requests generated by the L2 Pythia prefetcher. For SPEC CPU2017, GMS, Google, and Ligra, Nomad affects this metric only modestly. AI/ML shows a larger increase. In this suite, the Standard-MSHR keeps the L1D MSHR full for 50.3% of cycles (Fig. 2), serializing many misses before they

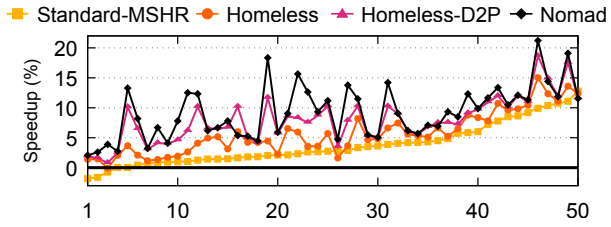
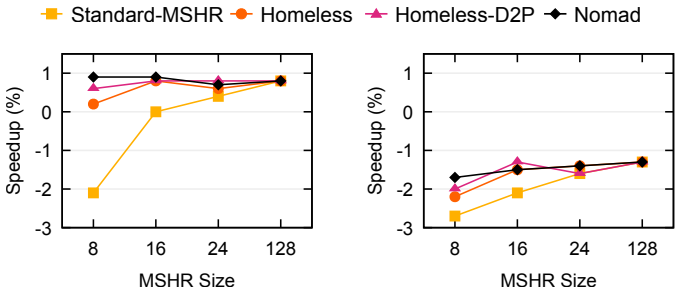


Fig. 13. Multicore speedup with Berti at L1D, normalized to a 16-entry Standard-MSHR without an L1D prefetcher



(a) NoPrefetch at L1D (b) Berti at L1D

Fig. 14. Sensitivity studies for Splash-4x workloads

reach L2. Nomad removes this L1D-side throttle, allowing more misses to reach L2 concurrently and increasing L2 MSHR pressure. GAP shows a different trend because the baseline is already under high pressure at L2. With Berti, Nomad exposes more L1D-side requests; the private L2 resolves many, and L1D-originated traffic, including demand-to-prefetch requests, is prioritized over L2-generated prefetches. Together, these effects allow Nomad to improve performance without increasing L2 MSHR pressure.

J. Multiprogrammed Multicore Workloads

Fig. 13 reports harmonic mean of speedups for 50 four-core mixes with Berti at L1D, normalized to a 16-entry Standard-MSHR without an L1D prefetcher. Nomad provides 9.2% speedup, compared with 7.9% for Homeless-D2P, 5.6% for Homeless, and 3.7% for the Standard-MSHR. Nomad is the highest-performing policy in 44 of the 50 mixes. In the remaining mixes, Homeless-D2P is higher. The only clear exception is a mix that includes `607.cactuBSSN`, matching the harmful-prefetch behavior discussed in Section VII-B.

Traffic follows the same trend as in the single-core evaluation. Nomad increases private L1D-to-L2 traffic from 59.3 to 108.8 requests per kilo-instructions, but this increase is largely contained by the private L2. L2-to-LLC traffic rises from 15.7 to 18.4 requests per kilo-instructions, while LLC-to-DRAM traffic remains nearly unchanged, from 20.1 to 20.7 requests per kilo-instructions.

K. Multi-threaded applications

Fig. 14 reports performance on SPLASH-4x [61] with 16 threads. SPLASH-4x has lower L1D MSHR pressure, so the differences between MSHR management policies are small. Without an L1D prefetcher, an 8-entry Standard-MSHR degrades performance by 2.1% relative to 16 entries. Nomad with only 8 entries provides a speedup of 0.9% over the 16-entry Standard-MSHR, while Homeless and Homeless-D2P reach only 0.2% and 0.6%, respectively. With Berti at 16 entries, `lu_ncb` and `ocean_ncp` cause all configurations to regress; Homeless-D2P reduces the loss from -2.1% to -1.3%, while Homeless and Nomad reach -1.5%. Absolute gains are smaller than on AI/ML or GAP, consistent with the lower MSHR pressure of SPLASH-4x, but the same trends hold.

VIII. Related Work

Parallel cache searches. Existing proposals have shown the benefits of speculatively issuing parallel memory requests across multiple cache levels when there is high confidence of a miss [14], [65]. This is particularly attractive for prefetching, where the goal is to bring in data that is not already present in the cache. While prior work has demonstrated the value of speculatively issuing simultaneous requests across multiple cache levels, our approach is both orthogonal and highly complementary. By relaxing the traditional requirement that every miss must immediately allocate a full MSHR in the upper-level cache, we enlarge the design space. For example, a prefetch may be issued directly to the L2 cache with the intent to fill L1.

Software prefetching. In software prefetching the programmer or compiler provides hints to the hardware to bring data into the cache before it is accessed. However, the effectiveness of this technique is often fundamentally limited by the underlying hardware resources, most notably the number of available L1D MSHRs, which can limit the achievable gains from prefetching [41], [66].

Scalable MSHR architectures. Asiatici et al. [7] presented FPGA-optimized MSHRs employing cuckoo hash tables in block RAM instead of associative memory. Gu and Chen [4] identify that conventional MSHRs in GPU caches typically allocate a fixed number of entries and slots per entry, causing inefficient utilization. They propose Dynamically Linked MSHRs (DL-MSHR), which dynamically allocate MSHR entries from a shared pool of slots.

IX. Conclusion

We presented Nomad, a new L1D MSHR design that ensures high-performance MSHR management for demand and prefetch requests. Nomad is based on three key decisions to take when the L1D MSHR is full: (i) bypassing the MSHR for prefetch requests, (ii) evicting MSHR entries occupied by prefetched requests to accommodate demand requests, and (iii) issuing demand requests as prefetch requests able to fill L1D. For AI/ML workloads and with an L1D Berti prefetcher, Nomad reaches 14.4% speedup compared to the 9.4% achieved by industry solutions without incurring storage cost.

Acknowledgment

We are very grateful to the anonymous reviewers for the time and care they devoted to our paper. We also thank the members of the CASPER group for their contributions to improving the final draft.

REFERENCES

[1] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *8th Int'l Symp. on Computer Architecture (ISCA)*, May 1981, p. 81–87.

[2] Ceze, Luis, J. Tuck, and J. Torrellas, "Are we ready for high memorylevel parallelism?" in *4thWorkshop on Memory Performance Issues (WMPI)*, 2006.

[3] Tuck, James, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *39thInt'l Symp. on Microarchitecture (MICRO)*, 2006.

[4] G. Yongbin and L. Chen, "Dynamically linked mshrs for adaptive miss handling in gpus," in *Proceedings of the ACM International Conference on Supercomputing*, 2019.

[5] J. M. King, "Systems and methods for tracking data cache miss requests with data cache tags," AMD, Aug. 2024, uS Patent 12,072,803.

[6] Power, Jason, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated cpu-gpu systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[7] Asiatici, Mikhail, , and P. Ienne, "Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas," in *2019 International Symposium on Field-Programmable Gate Arrays*, Feb. 2019, pp. 310–319.

[8] S. Anand, M. Friedman, M. Giardino, and G. Alonso, "Skip it: Take control of your cache!" in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1077–1094.

[9] O. Contributors, "Xiangshan: An open-source risc-v processor," https://github.com/OpenXiangShan/XiangShan, 2024, accessed: 2024-11-20.

[10] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[11] J. Kalamatianos, R. N. Bhargava, and R. Jayaseelan, "Prefetching to a cache based on buffer fullness," AMD, Dec. 2014, uS Patent 8,909,866.

[12] Alexander, K. Jason, Robinson, and E. Francis, "Prefetch mechanism for servicing demand miss," Apr. 3 2018, uS Patent 9,934,149.

[13] A. R. L. G. C. S. Grayson, "Prefetch chaining," US Patent 9569361 B2, Samsung Electronics Co., Ltd., 2014.

[14] B. Grayson, J. Rupley, G. D. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 40–51.

[15] "Cpu2017 flag description nettrix r620 g50," https://www.spec.org/cpu2017/results/res2023q2/cpu2017-20230328-35156.flags.html, 2024, report generated on 2024-01-29; Accessed 2025-11-16.

[16] Dell Technologies, "Dell poweredge xe9680 installation and service manual," https://www.dell.com/support/manuals/es-es/poweredge-xe9680/xe9680_ism_pub/processor-settings?guid=guid-71fdb36a-23ad-4720-b453-7347fd93e697&lang=en-us, accessed 2025-11-09.

[17] Fujitsu Technology Solutions, "Bios settings. 4th and 5th generation intel xeon scalable processor-based primergy server generation," https://sp.ts.fujitsu.com/dmsp/Publications/public/wp-sapphirerapids-bios-settings-primergy-ww-en.pdf, accessed 2025-11-09.

[18] Lenovo, "Lenovo uefi setup utility with intel xeon 6th processor," https://pubs.lenovo.com/uefi_xeon_6th/processors, accessed 2025-11-09.

[19] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[20] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[21] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 118–131.

[22] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *55th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 975–991.

[23] K. I. Farkas and N. P. Jouppi, "Complexity/performance tradeoffs with non-blocking loads," in *21st Int'l Symp. on Computer Architecture (ISCA)*, 1994, p. 211–222.

[24] *AMBA AXI Protocol Specification*, ARM, 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest

[25] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[26] R. Fernández-Pascual, A. Ros, and M. E. Acacio, "To be silent or not: On the impact of evictions of clean data in cache-coherent multicores," *Journal of Supercomputing (SUPE)*, Mar. 2017.

[27] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, M. D. S. Boris Shingarov, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[28] "Arm technical reference manual," 2008, revision E.

[29] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *The 3rd Data Prefetching Championship*, Jun. 2019.

[30] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 1121–1137.

[31] Intel, "Intel® 64 and ia-32 architectures optimization reference manual," www.intel.com, Apr. 2024.

[32] Fuguet and César, "Hpdcache: Open-source high-performance l1 data cache for risc-v cores," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023.

[33] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5. International Symposium on Computer Architecture Valencia, Spain, 2020, pp. 1–7.

[34] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE journal of solid-state circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[35] R. Karam, R. Puri, S. Ghosh, and S. Bhunia, "Emerging trends in design and applications of memory-based computing and content-addressable memories," *Proceedings of the IEEE*, vol. 103, no. 8, pp. 1311–1330, 2015.

[36] X. Yin, C. Li, Q. Huang, L. Zhang, M. Niemier, X. S. Hu, C. Zhuo, and K. Ni, "Fecam: A universal compact digital and analog content addressable memory using ferroelectric," *IEEE Transactions on Electron Devices*, vol. 67, no. 7, pp. 2785–2792, 2020.

[37] L. Wang, X. Zhang, S. Wang, Z. Jiang, T. Lu, M. Chen, S. Luo, and K. Huang, "Asynchronous memory access unit: Exploiting massive parallelism for far memory access," *ACM Transactions on Architecture and Code Optimization*, 2024.

[38] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 99–111.

[39] G. H. Loh, "3d-stacked memory architectures for multi-core processors," *ACM SIGARCH computer architecture news*, vol. 36, no. 3, pp. 453–464, 2008.

[40] Intel, https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf, Intel, 2021.

[41] Kühn, Roland, J. Mühlig, and J. Teubner, "How to be fast and not furious: Looking under the hood of cpu cache prefetching," in *Proceedings of the 20th International Workshop on Data Management on New Hardware*, Jun. 2024, pp. 1–10.

[42] Wang, Han, M. Tang, K. Xu, and Q. Wang, "Bandwidthbreach: Unleashing covert and side channels through cache bandwidth exploitation," *arXiv preprint arXiv:2306.01996*, 2023.

[43] AMD, *Software Optimization Guide for the AMD Zen4 Microarchitecture*, https://www.amd.com/content/dam/amd/en/documents/ processor-tech-docs/software-optimization-guides/57647.zip, AMD, May 2024.

[44] Compal Electronics, "BIOS and OS Tuning Descriptions for SPEC CPU2017," https://www.spec.org/cpu2017/flags/Compal-Platform-Flags-Linux-Intel_v1.1.html, 2025, last updated September 30, 2025; Accessed 2026-03-23.

[45] "SPEC CPU2017 Flag Description – Platform Settings, V1.0-SPR-RevD," https://www.spec.org/cpu2017/flags/Fujitsu-Platform-Settings-V1.0-SPR-RevD.html, 2023, accessed 2026-03-23.

[46] "Software optimization guide for the amd zen5 microarchitercture," 2024, publication# 58455, Revision 1.00.

[47] I. Corporation, "Second generation intel xeon scalable processors," https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-xeon-scalable-datasheet-vol-2.pdf, Datasheet 338846-001US, Apr. 2019.

[48] *AMBA CHI Architecture Specification*, Arm Ltd., https://developer.arm.com/documentation/ihi0050/latest/.

[49] *Arm Cortex-A76 Core Technical Reference Manual: About the L2 Memory System*, Arm Ltd., 2018. [Online]. Available: https://developer.arm.com/documentation/100798/latest/L2-memory-system/About-the-L2-memory-system

[50] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 888–904.

[51] A. Seznec, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016.

[52] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel Alder Lake CPU architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, Mar. 2022.

[53] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[54] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[55] P. Esmaili-Dokht, F. Sgherzi, V. S. Girelli, I. Boixaderas, M. Carmin, A. Monemi, A. Armejach, E. Mercadal, G. Llort, P. Radojković, M. Moreto, J. Giménez, X. Martorell, E. Ayguadé, J. Labarta, E. Confalonieri, R. Dubey, and J. Adlard, "A Mess of Memory System Benchmarking, Simulation and Application Profiling," in *57th Int'l Symp. on Microarchitecture (MICRO)*, 2024, pp. 136–152.

[56] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0," HP Labs, Tech. Rep. HPL-2009-85, Apr. 2009.

[57] "Calculating memory power for ddr4 sdram," Micron Technology, Inc., Tech. Rep. TN-40-07, Aug. 2018.

[58] "DPC-4 Released Traces," https://console.cloud.google.com/storage/browser/dpc4-all-traces, 2026.

[59] "The 4rd data prefetching championship (dpc-4)," Jun. 2025, traces available at https://console.cloud.google.com/storage/browser/dpc4-all-traces. [Online]. Available: https://sites.google.com/view/dpc4-2026

[60] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, Jun. 2003.

[61] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Splash-4: A modern benchmark suite with lock-free constructs," in *Int'l Symp. on Workload Characterization (IISWC)*, Nov. 2022, pp. 51–64.

[62] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Conf. on Supercomputing (SC)*, Nov. 2011, pp. 52:1–52:12.

[63] A. Navarro-Torres, S. Singh, B. Panda, and A. Ros, "The entangling data prefetcher," Sydney, Australia, Feb. 2026. [Online]. Available: https://webdiis.unizar.es/~agusnt/PDF/26/26_DPC4-EDP.pdf

[64] T. Koizumi, T. Nakamura, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "T-skid: Predicting when to prefetch separately from address prediction," in *25thDesign, Automation, and Test in Europe (DATE)*, 2022, pp. 1389–1394.

[65] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1–18.

[66] F. Mahling, M. Weisgut, and T. Rabl, "Fetch me if you can: Evaluating cpu cache prefetching and its reliability on high latency memory," in *Proceedings of the 21st International Workshop on Data Management on New Hardware*, 2025, pp. 1–9.