Stream-Aware Intelligent Memory Controller through HW/SW Co-Design

Abdelrhman M. Abotaleb
Electrical and Computer Engineering,
McMaster University
Hamilton, Ontario, Canada
abotalea@mcmaster.ca

Maziar Goudarzi
Huawei Technologies, Canada
Research Center
Markham, Ontario, Canada
maziar.goudarzi@huawei.com

Tomasz Czajkowski
Huawei Technologies, Canada
Research Center
Markham, Ontario, Canada
tomasz.czajkowski@huawei.com

Reza Azimi
Huawei Technologies, Canada
Research Center
Markham, Ontario, Canada
reza.azimi1@huawei.com

Mohamed Hassan
Electrical and Computer Engineering,
McMaster University
Hamilton, Ontario, Canada
mohamed.hassan@mcmaster.ca

Abstract

Memory hierarchy often represents a significant performance bottleneck in modern computing systems. A promising direction to mitigate this bottleneck is through HW/SW coordination at the system level. However, many existing solutions require changes to legacy programming paradigms, such as ISA extensions, and often provide specialized optimizations limited to specific modules or policies within the memory hierarchy. In this work, we introduce InterStellar, a HW/SW co-design methodology that overcomes these limitations. InterStellar enables the design of a stream-aware memory controller that dynamically adapts its scheduling and memory management policies while proactively batching future stream accesses from off-chip memory. The design is optimized not only for performance, but also for energy efficiency and bandwidth utilization. On systems with eight RISC-V cores, InterStellar achieves significant end-to-end speedup compared to a commercial off-the-shelf (COTS) memory controller: up to 2.72× for PolyBench, 1.84× for HPCG, 1.24× for Rodinia, 1.47× for Parboil, and 1.29× for the Phoenix suite.

CCS Concepts

• **Hardware** → **Memory and dense storage**; **Hardware-software co-design**; • **Computer systems organization** → *Parallel architectures.*

Keywords

Hardware/Software Co-Design, DRAM, Scheduling, Intelligent Systems, Memory Controllers, Stream-Aware Systems

1 Introduction

Five main observations motivate this work. We build the logical arguments for this work sequentially through these observations as follows. ***Observation 1: Memory bottleneck.*** While there have been significant advancements in processing capabilities of computing systems, one critical bottleneck that continues to impede the overall system performance is the memory hierarchy [1–4]. For instance, despite the substantial increase in the Dynamic Random Access Memory (DRAM) capacity over the past two decades [5, 6], the reduction in DRAM access latency remained marginal [7–10]. The impact of this bottleneck is further exacerbated with the increased demand for memory bandwidth and low-latency requirements from modern applications (e.g. machine learning and artificial intelligence). Comprising several shared resources, the memory hierarchy is prone to contention among different processing elements [11], while off-chip access latency can extend to hundreds of cycles [12, 13].

Observation 2: Lack of Hardware/Software (HW/SW) coordination. Modern Commercial off-the-shelf (COTS) architectures and a significant portion of the extensive architecture research works in the past decades have focused on solutions that satisfy two criteria: 1) software-obliviousness: hardware and software only communicate through the dictated ISA contract, and 2) hardware is built mainly for general-purpose. For example, the whole memory hierarchy is built with the locality concept at the individual requests basis through the memory (e.g. load/store) instructions and in most cases memory components (e.g. schedulers/arbiters) have no notion of issuing cores, their criticality, nature of the context or requirements. Despite its advantages, this paradigm leaves significant performance opportunities on the table with all hardware optimizations being best-effort (and in many times speculative) based on local information only; mostly current instruction window scope. Therefore, several recent works argued for utilizing software-driven knowledge to optimize the memory hierarchy [14, 15] including prefetchers [16–23], cache data placement, cache replacement policies [15, 24], and cache management [25–28].

Observation 3: Narrow scope of customized HW/SW solutions. Despite the performance improvements shown by these efforts, they provide a specialized solution that is limited to optimizing a specific module or policy within the memory hierarchy. Given the high upfront cost of a new HW/SW interface, it is unrealistic from industry perspective to add a new interface for each specialized memory optimization [24]. Thus, a general HW/SW interface is vital for adoption. For memory, this interface should enable the software to expose richer semantics about memory access patterns to the hardware modules in the hierarchy. In turn, these modules can use this information to intelligently adapt their decisions and policies to cater for the performance opportunities at hand.

Observation 4: Streams as a low-cost opportunity for richer semantics. On the quest for a non-intrusive yet general enough approach for such richer interface, we find the particular idea of memory streams is quite promising. In a non-formal definition, a memory stream consists of recurring patterns of memory access, typically resulting from loops and nested loops. Several papers have explored memory streams as a richer memory representation [21, 22, 29–33]. However, all of them focus on the cache hierarchy, not the main memory. Also, they all require ISA extensions. Finally, excluding the works focusing on different computing paradigms such as in/near memory computing [32, 33], most of these works focus solely on a single memory policy; namely, prefetching.

Observation 5: Memory policies cooperation. In this paper, we take a different direction. Instead of leveraging the stream information to guide cache prefetching [21–23] or even doing the computation at Level 3 cache (L3 cache) [32, 33], we use the stream information outside of the cache hierarchy at the memory controller (MC) to guide its DRAM management policies (namely, scheduling and DRAM row management), while also proactively fetching the stream data from the off-chip memory. Fusing the stream fetching with the MC policies offer several benefits as we detail in Section 3.

To this end, we propose in this paper InterStellar as a stream-based HW/SW co-design methodology. The detailed InterStellar's methodology is presented in Section 4. We also present a novel intelligent MC solution that leverages the software provided information about streams through InterStellar to optimize the fetching decisions and degree, the request scheduling, and the DRAM row management policy. The detailed architecture of the proposed intelligent MC is also illustrated in Section 5. The implementation details, including compiler modifications and hardware cost, are detailed in Section 6. The system considerations are presented in Section 7, while the experimental environment is outlined in Section 8. Detailed evaluation is illustrated in Section 9. Our gem5 results show that applying InterStellar in an 8-core setup improves performance by up to 2.72x, 1.84x, 1.24x, 1.47x, and 1.29x for PolyBench, HPCG, Rodinia, Parboil, and Phoenix, respectively.

2 Background

2.1 DRAM Operation

DRAM consists of multiple banks that can be accessed nearly simultaneously enhancing memory system performance. Each DRAM bank is a two-dimensional grid of cells arranged in rows and columns. Accessing data from a row loads it into sense amplifiers functioning as a small cache within each bank. This is known as the *row buffer* or *DRAM page*. *Row hits* are the accesses to data in an opened row buffer. They need only to access the requested columns from the row buffer through a CAS command. Therefore, they exhibit a lower access latency known as the column latency (t_{CL}) according to the JEDEC DRAM standard [5]. In contrast, accessing rows not in the row buffer requires three prior steps. 1) Precharging the sense amplifiers with a PRE command needs row precharge cycles (t_{RP}). 2) Activating the new row with an ACT command exhibts row-to-column delay cycles (t_{RCD}). 3) Issuing a CAS command to read/write from/to the requested columns. This scenario is known as *row conflicts* and incur higher access latency: $t_{RP} + t_{RCD} + t_{CL}$. If the row buffer of a bank does not have any data (for example initially after power-on or after a DRAM refresh operation), the bank is referred to as *idle* and accessing it is a *row miss*. A row miss only needs the ACT and the CAS commands, so it incurs a latency of $t_{RCD} + t_{CL}$.

DRAM page policies define how a MC manages DRAM pages. There are three widely used DRAM page policies:

1) Opened Page Policy. The MC keeps the row buffer open after access. If a subsequent request needs data from the same row, it is accessed quickly. This approach works well for sequential data but inefficiently for poor-locality data.

2) Closed Page Policy. This policy automatically precharges the row after each read and write, closing the page regardless of row misses or hits. It offers predictable latency and is efficient when conflicts dominate access patterns. However, it fails to capitalize on the locality of sequential accesses.

3) Commercial-Off-the-Shelf (COTS) Page Policy. It dynamically switches between open and close policies based on the observed access patterns. By tracking hit/miss rates in DRAM banks, it speculates whether to keep the row open or close. It is widely used in modern COTS MCs [34–39].

2.2 Data Streams

A ***stream*** refers to a foreseeable sequence of memory access patterns that can be vectorized [29]. It represents a structured pattern of data flow, facilitating efficient data exchange and processing [23]. There are two common stream types. **1) A direct stream**, which is defined by an access size (size of the individual data element accessed in memory), stride (size between consecutive accesses), and number of strides. In its simplest format, a direct stream can be represented in the form of A[i]. A direct stream generally exhibits a high locality memory access pattern with predictable addresses. This represents a huge opportunity for intelligent hardware optimizations, if such information is conveyed from the software layer. Direct stream data processing is commonly employed in diverse fields such as media streaming, big data analytics, financial services, machine learning and AI, and supply chain management [40].

2) An indirect stream, which derives its memory addresses from the data entry of another stream (e.g. in the form of A[B[i]]). The data from the originating stream can be interpreted either as an offset from a base address or as a pointer value. Indirect streams have the capability to be linked together in chains, allowing the creation of complex multi-indirect access patterns. (eg. A[B[C[i]]]). Such indirection is common on graph analytics, machine learning, and other sparse linear algebra-based applications [41]. Indirect streams can exhibit either high or poor locality characteristics. By accurately representing the stream, it had been shown also that is possible to develop efficient hardware solutions for data-centric applications, such as the specialized streaming cores and accelerators [23, 42–44]. Although the approach can be generalized to other stream types (ex: pointer chasing, trees, and graphs), in the current work we focus only on the direct and indirect streams. Actually, other data structure access patterns can be reduced to either direct streams or indirect streams. For example, linked lists often follow a direct stream pattern when traversed, provided that the memory is not fragmented.

3 Motivation

3.1 Bridging Gaps Beyond Traditional Prefetching

Hardware prefetchers (HWPs) have long served as a practical solution for mitigating memory latency by speculatively fetching data based on observed access patterns. While effective for many workloads with regular locality, our experiments reveal scenarios where their underlying assumptions become limiting, exposing a gap for new strategies. In particular, we identify several recurring bottlenecks that arise across a variety of memory-intensive benchmarks (BMs), each highlighting specific limitations in how traditional prefetchers interact with system-level resources such as caches, and MCs. We illustrate these challenges through representative workloads exhibiting diverse access patterns and system configurations. Despite being accurate, conventional prefetching mechanisms can introduce several system-level bottlenecks. The *key limitations* we observe are summarized below:

(1) **Limited Miss Status Holding Registers (MSHR) Capacity:** MSHRs track outstanding memory misses in non-blocking caches. Once they are full, further memory requests—prefetch or demand—must stall, limiting memory-level parallelism even if the MC has available capacity.
(2) **Limited DRAM Read Queue:** When the read queue at the MC becomes full, both prefetch and demand requests experience delays, creating memory-level backpressure regardless of prefetch accuracy.
(3) **Multi-Cluster Prefetch Contention:** In multi-cluster systems, each with its own prefetcher, independent memory streams may target shared DRAM banks or rows. Without coordination, this traffic causes row conflicts, bank contention, and delays at the MC. This occurs even in single-cluster multi-core systems.

These bottlenecks also give rise to several observable *artifacts* that impact overall system behavior and efficiency:

(1) **Cache Pollution:** Prefetched data that is not used in time may evict useful demand data, reducing cache hit rates and increasing memory traffic.
(2) **Inefficient Use of Memory Bandwidth:** When the MC's read queue is full, prefetch requests continue to flow from the LLC, consuming on-chip interconnect without being serviced, leading to wasted bandwidth.
(3) **Elevated DRAM Energy Consumption:** Prefetches that cause frequent row activations and precharges— particularly when they lead to row conflicts or are dropped— can increase DRAM energy consumption.

To support these arguments, we now turn to quantitative observations from our experimental setup. We run three experiments using an 8-core Out-of-Order (OoO) RISC-V system configured with the baseline parameters in Section 8, with minor adjustments per experiment to isolate specific bottlenecks. **Experiment 1** uses ideal MSHRs (512 entries) but a realistic memory controller with 32-entry read queue, sweeping both the HWP degree and queue size. **Experiment 2** inverts this: it provisions an ideal 2048-entry read queue while limiting MSHRs to a practical 64 entries. **Experiment 3** To assess the efffect of DRAM contention, we experiment two sub-cases: 1) Eight-clusters, each with its own Last-Level Cache (LLC) and stream prefetcher. 2) Direct stream BMs that access the DRAM in large stride. Both cases would generate more conflicts. To isolate other bottlenecks, a setup with ideal MSHRs (512), and MC read queue size (2048) is established.

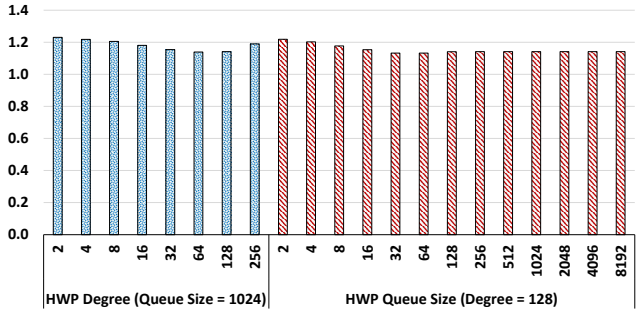


Figure 1: Speedup of *InterStellar without HWP* against *COTS with Stream HWP* using experiment 1 setup.

Selected Benchmarks

We selected a subset of BMs— `gesummv`, `dscals`, and `srad_v2` that collectively cover different characteristics of memory behavior. `gesummv` captures low-compute, memory-intensive execution with multiple streams; `dscals` from LAPACK allows us to study large-stride streaming with configurable strides; and `srad_v2` combines compute-heavy phases with significant memory intensity. These BMs were selected to illustrate different workload characteristics in terms of locality, stride, and compute/memory balance.

Experiment 1: MC Read Queue Saturation

We evaluate the impact of MC read queue saturation using the `srad_v2` BM from the Rodinia suite, which generates seven direct memory streams—ideal for stress-testing stream prefetchers. To isolate the effects of prefetcher aggressiveness:

- We vary the HWP's degree from 2 to 256, fixing the HWP queue size at 1024.
- Then, we fix the degree at 128 and sweep the HWP *queue size* from 2 to 8192.

These sweeps reveal how increasing prefetch volume stresses both the MC read queue and overall system performance.

Results revealed that InterStellar consistently outperforms the stream HWP baseline. Fig. 1 shows the speedup of InterStellar (without HWP) over *COTS with Stream HWP* across varying prefetcher parameters. If the HWP were effective, higher aggressiveness should reduce InterStellar's relative speedup. However, InterStellar achieves up to 1.23× speedup at low degrees (2–16), with gains plateauing beyond degree 32 due to memory pressure from uncoordinated prefetching. Queue size sweeps show similar limits: speedup saturates at size 32, matching the MC read queue capacity, with no benefit from larger queues. Fig. 2 confirms this by showing the normalized frequency of queue-full events—how often the queue blocks relative to total memory requests. Queue pressure spikes at degree and size 32, then plateaus, indicating full MC utilization. Beyond this point, extra prefetch traffic stalls upstream, wasting

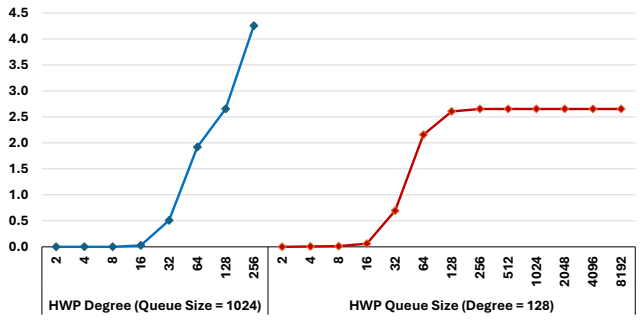


Figure 2: Normalized MC's read queue blocking times for COTS with HWP using experiment 1 setup..

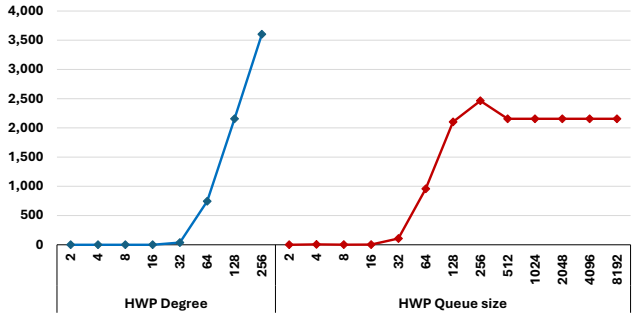


Figure 4: MSHR blocking frequency as HWP degree (left) and queue size (right) increase with experiment 2 setup.

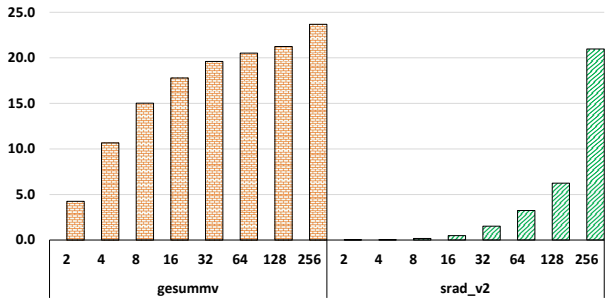


Figure 3: Cache pollution as HWP degree increases for `gesummv` (left) and `srad_v2` (right) with experiment 1 setup.

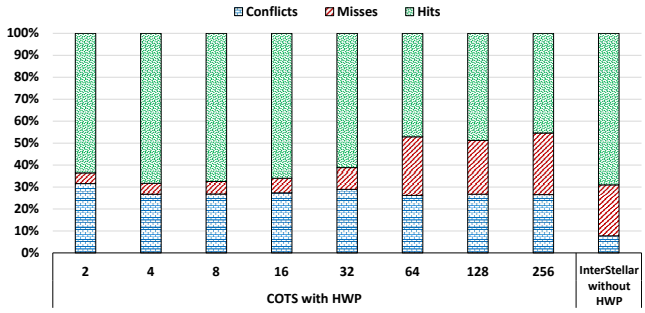


Figure 5: DRAM requests breakdown for COTS with HWP and InterStellar without HWP with experiment 2 setup.

bandwidth and explaining why further HWP aggressiveness yields no additional gains.

Fig. 3 shows cache pollution—prefetched lines evicted before use—under increasing HWP degree for two BMs: `gesummv` (left) and `srad_v2` (right). In `gesummv`, with two streams and low compute intensity, pollution exceeds 15% at degree 8 and reaches 23% at 256 due to rapid cache saturation. In contrast, `srad_v2` has compute-heavy phases as well as being memory intensive. The compute phases delay generation of more prefetches, keeping pollution low until degree 64, then rising to 21% at 256. These trends show that both access patterns and compute intensity influence prefetch effectiveness, beyond accuracy alone.

Experiment 2: MSHR Bottleneck

To assess MSHR constraints, we fix the DRAM read queue at 2048 entries and limit MSHRs to 64. Repeating the HWP parameter sweep from Experiment 1, *InterStellar* achieves up to **1.27**× speedup over the COTS baseline, but this drops to **1.17**× and plateaus as the HWP degree and queue size reach 64. Fig. 4 shows MSHR blocking frequency rising sharply at this point and saturating. This inflection aligns with the 64-entry MSHR limit, confirming that increased prefetch aggressiveness is bottlenecked by MSHR capacity.

DRAM statistics further reveal that traditional HWPs are oblivious to DRAM organization. Increasing HWP degree under MSHR bottlenecks results in fragmented, uncoordinated streams that cause row-level contention and degraded DRAM efficiency. In contrast, *InterStellar* is DRAM-aware (Section 5.2) and batches only row hits. It also applies intelligent page policies (Section 5.1) that reduce row conflicts by issuing early auto-precharges for known future conflicts—enabled by HW/SW co-design and stream-level access knowledge.

Key observations from the DRAM-level breakdown (Fig. 5):

- **COTS with HWP** starts with a 62% row hit rate at low degrees. As the degree increases, hits improve slightly but then sharply drop to 45% at degree 64—coinciding with MSHR saturation. The hit rate then plateaus. At the same point (degree 64), the combined row conflict and miss rate increases significantly—from 25% to 55%. This is due to increased out-of-order request arrivals and inter-stream interference.
- **InterStellar**, by contrast, maintains a high row hit rate (70%) and keeps conflict rate low (7%), demonstrating effective DRAM row-locality preservation and minimal request interference.

Experiment 3: Elevated DRAM Conflicts Bottleneck

While Experiment 2 showed that DRAM conflicts rise under MSHR pressure, we now demonstrate that even with ideal MSHR (512 MSHRs) and MC resources (2048 entries per read queue size at MC), DRAM-level contention remains a critical bottleneck. We analyze two representative scenarios: (1) multi-cluster interference, and (2)

(a) Multi-cluster Case (b) Large Stride Case

Figure 6: DRAM access patterns under two scenarios: (a) Eight Clusters. (b) Streams with large-stride, horizontal axis values represents strides as 1, 2, 4 cache lines.

large-stride streaming.

1. Multi-cluster contention. We simulate an 8-cluster system, where each cluster has its own LLC and independent stream prefetcher. Using `srad_v2`, we observe that after compute-heavy phases, early prefetches return and populate LLC. Subsequent accesses from the same stream thus trigger fewer prefetches. As multiple clusters do this in parallel, the resulting fragmented memory streams cause severe DRAM contention. To isolate cache pollution effect, we also run a special setup where HWP stores the prefetches internally inside an isolated buffer. Each LLC request that doesn't hit in LLC checks if it hits inside this buffer first before creating MSHR. As shown in Fig. 6a, DRAM conflict rate under COTS with HWP is more than 95%, with a row hit rate under 5%. In contrast, *InterStellar*, by batching only page-aligned hits, reduces conflict rate to 14% and raises hit rate to over 80%, resulting in a 3.1× reduction in active DRAM cycles.

2. Large-stride streaming. We evaluate `dscal` BM from LAPACK with stride lengths of 1, 2, and 4 cache lines. Larger strides increase LLC misses, leading to more direct DRAM traffic. However, due to cache or MSHR hits from earlier prefetches, fewer new prefetches are issued per demand, and they are increasingly fragmented. This worsens DRAM access patterns—especially when prefetches cross row boundaries.

Fig. 6b shows that in COTS with HWP, DRAM conflict rate increases from 40% (stride = 1 line) to 65% (stride = 4 lines). On the other hand, InterStellar maintains a consistent hit rate of 92% with minimal conflicts by batching only row-buffer hits.

In conclusion, fusing the stream fetching in InterStellar with the MC policies offer several benefits.

1) On one side, it allows the MC to maximize row locality beyond its current reordering capabilities that is only limited to the request buffer sizes of the MC (usually at the range of 16–64 cache line requests.

2) On the other side, it guides the fetch aggressiveness of streams since fetching beyond DRAM row sizes leads to the high row conflict latency. Therefore, the fetching aggressiveness is now a function of the stream stride size (provided by the HW/SW interface), and the DRAM row buffer size (known to the MC). It is important to note that the latter is not available to prefetchers at the cache hierarchy.

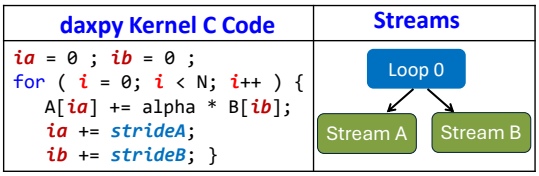


Figure 7: *daxpy* kernel C Code and its stream flow graph.

3) Moving the stream fetching to the MC enables us to delay the issue of the prefetch requests (we denote as a *stream batch*) until a demand request from the stream opens the DRAM row. Doing so, the memory scheduler schedules all the requests to this row back to back and uninterrupted ensuring that the row buffer locality remains intact. If these prefetches were issued from the cache hierarchy, there's no guarantee they would reach the MC in time for it to reorder them and take advantage of row locality. Several reasons make this behavior hard to guarantee. For instance, some prefetches may be delayed due to buffer backpressure (e.g., MSHR fullness) or because of scheduling by Network-on-Chip (NoC) routers.

4) Compared to cache prefetching, our solution does not suffer from known prefetch painpoints including data array pollution, pressuring cache resources such as buffers, MSHRs, and read queue at the MC as well as competing with genuine demand requests for interconnect and cache bandwidth.

(5) It removes the need to tag and differentiate demand vs. prefetch requests, a common practice used to deprioritize prefetches in lower-level schedulers. This is unnecessary in our design because: (a) the HW/SW co-design ensures that only data known to be accessed next is fetched; and (b) fetches are issued just-in-time, buffered at the MC, and forwarded to the cache only when requested.

(6) It enables coordinated stream fetching, balancing per-stream performance and minimizing inter-stream contention. In contrast, uncoordinated prefetching across distributed caches causes streams to compete for shared on-chip and off-chip resources, potentially negating performance gains.

It's important to note that InterStellar improves the performance under cache prefetching. We compare it against various prefetch techniques and report the performance gain when combined with prefetchers as shown in Section 9.3.

3.2 Illustrative Example

This section illustrates how data-driven, stream-aware intelligent MC decisions can significantly improve memory performance. We use the *daxpys* BM from the LAPACK [45, 46] as an example, which has two direct streams as the code snippet in Fig. 7 illustrates. While each direct stream has high intra-stream locality, they interfere and lose this locality when conflicting on the same bank. Fig. 8 shows this, with requests from A and B arriving at the MC in an interleaved manner, where one request arrives as another starts. Note that the requests shown can be either demand or prefetch ones. Also, even if the HWP issues these requests in close time proximity, they still suffer the network on chip delay, bus delay, access delay, and the bottlenecks, observed and discussed in Section 3.1, before they reach the MC. The COTS baseline policy keeps hopping rows to serve the requests from the two streams causing all requests to be

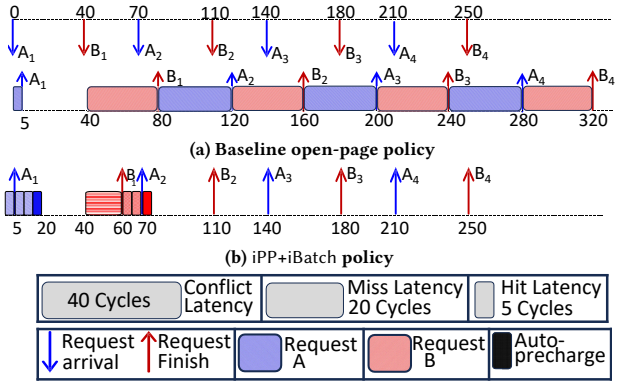


Figure 8: Two streams from *daxpy* BM: baseline DRAM latency vs. iPP+iBatch policy.

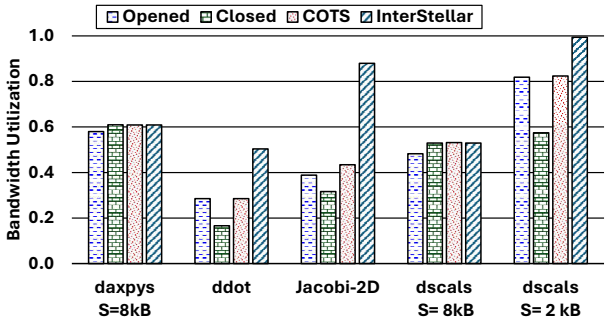


Figure 9: Bandwidth utilization for different page policies.

conflicts. Hence, it incurs large memory delays (380 cycles to serve 8 request).

It is also important to note that currently deployed reordering techniques in MCs (e.g. FR-FCFS) are limited to what requests co-exist in the MC's queues upon scheduling. So, if requests of the same stream are spaced in time as in Fig. 8a, such scheduling will not be able to optimize access pattern. On the other hand, by communicating the streams information, InterStellar enables intelligent MC decisions as follows: 1) When a request from a stream arrives, the MC associates it with its stream. Knowing the stream characteristics, i.e. direct stream's stride, the MC proactively issues future accesses for the same DRAM row (row hits) and buffer them in the MC to serve these requests when they arrive (iBatch technique). 2) When no more determined access to the opened DRAM page, the MC decides to issue the last CAS command in the batch with auto-precharge to close the row. This converts the first request from the other stream from a conflict to a miss, reducing its latency from 40 to 20 cycles (iPP policy). This is shown in Fig. 8, which uses representative timing values from DDR4 [5]. As Fig. 8b shows, combining both techniques reduces the time to serve the 8 requests from 320 to 250 cycles.

3.3 Page Policy Evaluation

Fig. 9 shows how different DRAM policies perform on certain BMs simulated using GEM5, measuring utilization by actual versus maximum bandwidth. It highlights two BMs (*daxpys* and *dscals*) where the closed policy excels over the open policy due to conflict-heavy access patterns in these BMs. While COTS outperforms both policies in all scenarios, InterStellar shows potential for even better utilization. The hardware-only speculative solution (COTS policy) has two main limitations: a) It fails to recognize access patterns in complex, multi-stream workloads. b) It cannot identify specific streams, cores, or applications linked to requests, resulting in subpar performance in multi-core environments or single-core setups with parallel streams. This occurs because the MC handles interleaved accesses from conflicting streams, causing one stream to influence policy decisions for another. This observation motivates our proposed iPP, a stream-aware intelligent page policy detailed in Section 3. It is important to note that although DRAM refreshes are enabled in our setup, InterStellar's high bandwidth utilization can be attributed to the intelligent buffer integrated into our design. This buffer proactively batches future data ahead of demand, enabling many incoming requests to be serviced directly without requiring a full DRAM access. As a result, the impact of refresh cycles on effective bandwidth in some BMs is significantly mitigated, since requests can often be overlapped with refresh activity. This behavior directly justifies the high bandwidth utilization observed in InterStellar. As Fig. 9 shows, such iPP outperforms hardware only solutions leading to better memory performance, specially in complex multi-core/multi-stream environments.

4 Architecture Extensions

InterStellar architecture is shown in Fig. 10. It consists of three main components: **1) HW/SW Interface**: A standardized interface that utilizes the Instruction Set Architecture (ISA) via compiler modifications to convey loops and streams metadata to the cores. **2) Nucleus**: InterStellar's central engine operates at the LLC level. It reads the passed metadata from the cores, computes useful streams' information, tracks LLC requests. It then appends requests with condensed data to enhance memory hierarchy performance. **3) MC Extension**: MC is extended with modules to collect a per-stream miss rate data while using passed information from Nucleus to determine software-driven memory policy and intelligent batching. InterStellar parts are discussed in Sections 4.1 - 5.

4.1 The HW/SW Interface

We do not require any ISA or operating system modifications, which is a key advantage for adopting InterStellar. In the used RISC-V ISA, we utilize Control and Status Registers (CSRs) to transfer data from software to hardware. RISC-V core has 4096 CSRs, but only 314 CSRs are allocated, leaving many CSRs for custom use [47]. RISC-V ISA has csrrw instruction for direct CSR writes. On platforms without such registers, we can use store instructions to a pre-defined reserved memory location. This suffices to serve the purpose since the hardware can then read and use these values.

We focus on two common stream types: direct and indirect. All streams are within a loop (can be nested). We design four descriptors to encode their features then pass them to the hardware, as

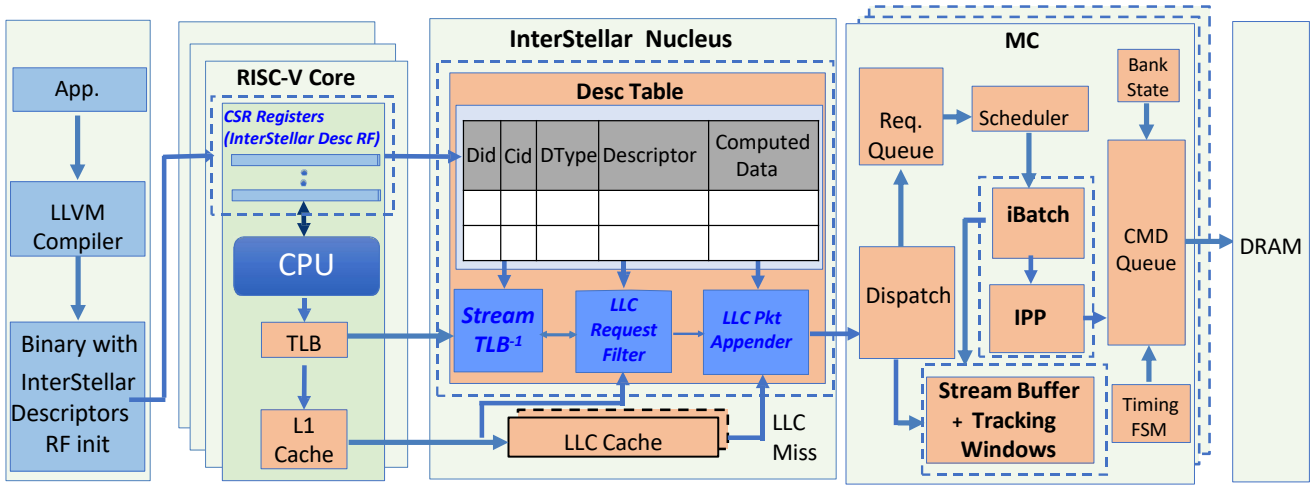


Figure 10: InterStellar Block Diagram.

shown in Fig. 11. The descriptors store the minimal necessary information characterizing the stream and its associated loop. All

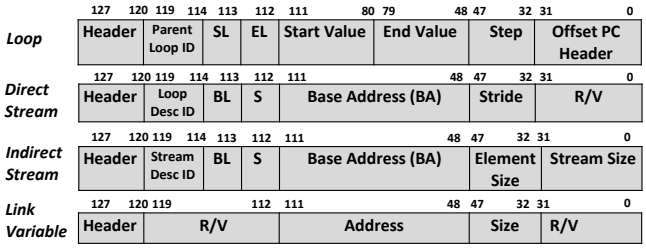


Figure 11: InterStellar descriptors format.

descriptors have a **header** field to encode the stream type, and activate it.

The **Loop Descriptor** defines loop boundaries: Start Value, End Value, and Step. If start or end values are linked to other variables (e.g., Start equation is $i = j$ or end condition is $i \leq k$), *Start Linked (SL)* or *End Linked (EL)* is set to 1. The link variables (j and k in the above examples) will have associated link descriptors. The linked variable's descriptor ID is stored in Start Value and End Value. The Parent Loop ID stores the outer loop's ID in nested loops; or 0 otherwise.

The **Direct stream descriptor** defines a direct stream by its associated innermost Loop Desc ID, the Base Address *(BA)* of its corresponding array, and the Stride length. The Stride is the byte difference between two consecutive stream elements in the linear address space. This depends on the induction variable's Step value and the stream element's size. If the *BA* connects to another variable, then *Base Linked (BL)* is set to 1, and the *BA* stores the link variable's descriptor ID.

The **Indirect stream descriptor** characterizes indirect streams. The indirect streams' accesses could be random unlike the sequential accesses for direct streams. This depends on the data values read from the index array. *Element Size* field encodes each element size in the stream. *Base Linked (BL)* is used in the same manner defined with the direct stream. Stream Size field defines the total indirect stream size.

The **Link variable descriptor** holds the *register ID*, and bytes *size* of a variable stores the loop's start or end values, or the *BA* of a direct/indirect stream as defined above.

Multi-threading and Shared Data: *Shared (S)* field indicates a shared stream between different threads, or cores. It is useful for InterStellar future cache optimizations. However, it is not needed for the current MC use-case as MC is after the coherence domain and the iBatch is fetched once by design.

Note that **all descriptors** values are known at the compile time and InterStellar is architected such that it can be easily extended to support other stream types.

4.2 Nucleus: InterStellar's Central Engine

Nucleus is the central component of InterStellar. It consumes the descriptors from the CSRs, uses them to calculate further streams' information, and track run-time information to guide all the memory hierarchy components to take better software-aware decisions. Nucleus reads from CSRs only when they are written by the application. It uses the conveyed knowledge from CSRs to infer further information. For the loop descriptor, Nucleus uses Start (S), End (E), and Step (I) values to compute the loop iterations count, as follows: $loop\_iterations_i = (E_i - S_i)/I_i$, $loopid = i$. For nested loops, counts of the parent loops are also considered.

Nucleus computes the Virtual Address *(VA)* range of the stream as the space between its VA's start and end.

For a *direct stream* i, its start and end VAs, denoted as $start\_VA_i$, and end_VA_i, are given by Equations 1 and 2, where $base\_VA_i$, and $stride_i$ are its base VA and stride, and $S_{Loop\_ID[i]}$ and $E_{Loop\_ID[i]}$ are the start and end associated loop's values.

$$start\_VA_i = base\_VA_i + S_{Loop\_ID[i]} * stride_i \quad (1)$$

$$end\_VA_i = base\_VA_i + E_{Loop\_ID[i]} * stride_i \quad (2)$$

For *an indirect stream i*, its $start\_VA_i$, and end_VA_i are computed in a similar way; See Equations 3 and 4, but rather than using the stride, Nucleus uses stream's size ($size_i$).

$$start\_VA_i = base\_VA_i \tag{3}$$

$$end\_VA_i = base\_VA_i + size_i \tag{4}$$

Computing base and end VAs helps in identifying unique streams even if the same page frame contains multiple streams. This extra computed information along side the descriptor itself is stored in the descriptor table, Desc Table in Fig. 10, where Did is the descriptor ID, Cid is the core ID, Tid is the thread ID, and DType is the descriptor type (DType = 0 for loop descriptor, 1 for direct stream, 2 for indirect stream, and 3 otherwise). As Fig. 10 shows, Nucleus resides as a module in the cache hierarchy. In this paper, we position Nucleus alongside the LLC to guide intelligent MC decisions. Even with a bankized/tiled LLC, only one Nucleus is needed. Here, all misses from the upper cache-level are simultaneously read by Nucleus and the tiled LLC. At run-time, LLC Request Filter checks whether an LLC request matches a stream in the Desc Table. If the demand miss belongs to one of the streams, LLC Request Appender appends the demand miss packet with extra information to be used in the MC towards a software-driven hardware decision. There are three key architectural challenges and decisions:

Challenge 1: *Stream information passed from the software and stored in Desc Table is in the VA domain, while LLC is physically addressed.* To address this challenge without modifying the operating system page mapping or complicating the HW/SW interface, we add the **Inverse Stream Translation Lookaside Buffer** (Stream TLB^{-1}) component as shown in Fig. 10. on a TLB miss, a parallel comparison is made between the TLB's VA and all VAs stored in the Desc Table. If a match occurs, the corresponding (PA,VA) pair is added as a new entry to the TLB^{-1}. For page or thread migration: These migrations typically cause TLB flushes or misses, so TLB^{-1} will be updated without additional circuitry. While TLB^{-1} is essential for InterStellar's operation, it has a small size for several reasons. 1) Nucleus only handles translations for active descriptors, focusing on TLB misses from direct and indirect streams. 2) Data-intensive systems use larger page sizes, enhancing performance, which simplifies the TLB's requirements [48]. 3) For direct streams or highly localized indirect streams, only recent pages that issue TLB misses are tracked as the stream always progresses to new pages. In cases of poorly localized indirect streams or hugely strided-direct stream, where stream can jump between different pages randomly, TLB^{-1} entries can be overwritten. InterStellar effectively manages this by treating such incidents as non-stream activities and adjusting policies based on a dedicated conflict tracking window for non-stream special case. The TLB^{-1} is used in a recent data movement optimization work with similar tricks to optimize its size [24]. This is discussed in Section 6.2. The **LLC Request Filtration Unit** (LLC Request Filter in Fig. 10) is responsible of submitting the incoming LLC requests to the TLB^{-1} to get the request VA if it belongs to a valid stream; then it probes the Descriptor Table to identify the stream type and ID.

Challenge 2: *It is important to keep the extra delay that* Nucleus *adds to the critical path of a memory request as minimal as possible.* To do so, we operate the main logic of Nucleus in parallel to the LLC tag access operation. LLC Request Filter takes two cycles. In the first cycle, it accesses the TLB^{-1} to obtain the VA. Then in the second cycle it accesses the Desc Table to get the corresponding stream data. Since the LLC usually takes more than two cycles to perform tag check and forward the miss to the MC, the LLC Request Filter delay is off the critical path. Therefore, the only component exists in the critical path is the LLC Pkt Appender, which simply augments extra wires to the original memory request and hence does not impact the critical path delay.

Challenge 3: *Extra wiring is required to add additional information to memory requests.* To minimize this, we keep the extra data from the Nucleus to the MC minimal. In our setup, the LLC Pkt Appender attaches the Stream ID, core ID, thread-ID, and stream type to the LLC packet. The direct stream's stride, which is constant, is only sent once in the first request to setup the MC's stream registers. Modern Systems-on-Chip interconnects like the **ARM AXI** bus already have user-defined bits sufficient to encode this information.

4.3 Illustrative Example

In the motivation section example, the *daxpy* BM features one loop and two streams. For simplicity, assume each direct stream containing 512 double values, resulting in an effective size of 8kB and a stride of 8 bytes. The loop index starts from 0 to 511, with an increment of 1. Fig. 12(a) shows the initialization details for the descriptors. The Desc Table contains streams' static information, along with the end VA for the direct streams, depicted in Fig. 12(b). Specifically, stream A's VAs range from 0x3000 to 0x3FF8, and stream B's VAs range from 0x5000 to 0x5FF8. In Fig. 12(c), two LLC miss requests are received, their addresses are 0xB200, and 0x1700. The LLC Request Filter checks TLB^{-1} to find that 0x1700 is not in any stream, and 0xB200 is a stream with VA=0x3200. Then it compares the VA against all stream tuples in the Desc Table and found that it belongs to stream A's address range. Subsequently, the LLC Request Filter sends the requests to the LLC Request Appender, which will append the packets. The first request gets the stream ID of 1, stream type of 1 (Direct Stream), and core ID of 0. The second request gets the stream ID of 0, stream type of 3 (None), and core ID of 0. Finally, it sends the packets to the MC for further processing.

5 Use-case: Intelligent MC

We provide two techniques that leverage the software access pattern for efficiency: (i) a page policy that keeps a page open or closes it, and (ii) a batching mechanism that proactively, yet deterministically, fetches stream data from an opened DRAM page.

5.1 Intelligent Page Policy (iPP) Management unit

It is composed of two components to manage the direct and indirect streams independently illustrated as follows:

1) Direct stream hit counters It manages direct stream conflicts when crossing the DRAM row size boundary. Consecutive direct stream accesses number to cross the DRAM row buffer boundary

Loop descriptor data:

Desc.	ID	Start	End	Step
Loop	0	0	511	+1

Two direct streams (A,B) data:

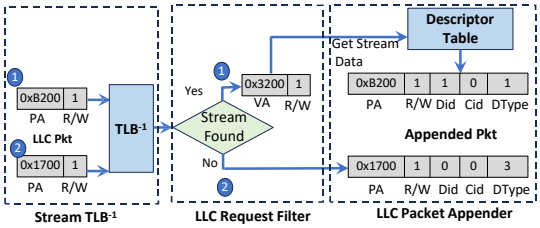
Desc.	ID	Loop ID	VA_{Start}	Stride
A	1	0	0x3000	4
B	2	0	0x5000	4

(a) Descriptors

Nucleus Descriptor Table :

Did	Cid	Tid	DType	Descriptor Data	Computed {Direct Stream End_{VA}}
0	0	0	0	{Start=0, End=511, Step=1}	-
1	0	0	1	{$Start_{VA}$=0x3000, Stride =4}	End_{VA}= 0x4FFC = 0x3000+(511*4)
2	0	0	1	{$Start_{VA}$=0x5000, Stride =4}	End_{VA}=0x6FFC = 0x5000+(511*4)

(b) Descriptor Table



(c) LLC Request filter and appender

Figure 12: Illustrative Example of descriptor table, then filter LLC requests, then append it with Did, Cid, and DType.

can be calculated by Equation 5.

$$\text{Max Hit Counts}_{stream_i} = (\text{DRAM Page Size})/(Stride(Stream_i)) \quad (5)$$

Here, $Stride(Stream_i)$ represents the larger value between the cache line size and the stride of i^{th} stream. A dedicated counter tracks DRAM read hits for each direct stream. When this counter reaches the threshold from the above formula, it resets, and the iPP management logic closes the row buffer.

2) Indirect Streams Conflicts tracking window is a module used to track DRAM page conflicts/misses per each data stream. It measures access locality of indirect streams and utilizes a counter to monitor the misses for each stream. The policy for indirect stream closes the row if the following condition is satisfied, otherwise keeps it open:

$$\text{Conflicts} > T_H \;||\; (T_L \leq \text{Conflicts} \leq T_H \;\&\&\; \text{Prev Access} = \text{Conflict}) \quad (6)$$

Where T_H, and T_L are adjustable thresholds. If the conflicts are greater than T_H, or lie between T_H and T_L with previous access was conflict, this means that accesses tends to conflict then close the row, otherwise leave it open. The conflicts tracking window block diagram is shown in Fig. 13. It measures locality by comparing the current address *Crnt Addr* in the MC with the previous address *Prev Addr* for the same bank from the history table. Since each request is tagged with stream ID and the stream type label, the tracking window is implemented on a per-stream basis. This intelligent tracking window ensures that the system is better equipped to handle diverse workloads and varying access patterns leading to improved efficiency in memory management and overall system operation. By employing both the *indirect streams' conflicts tracking window* and the *direct streams' hit row buffer read access counter*, we improve DRAM page policy and ensure that the DRAM row buffer is only closed when conflict is imminent.

5.2 Intelligent Batching (iBatch) Management unit

Key approach to enhance MC performance is by issuing read commands for future memory accesses in batches. The MC calculates the addresses of subsequent DRAM accesses based on the current

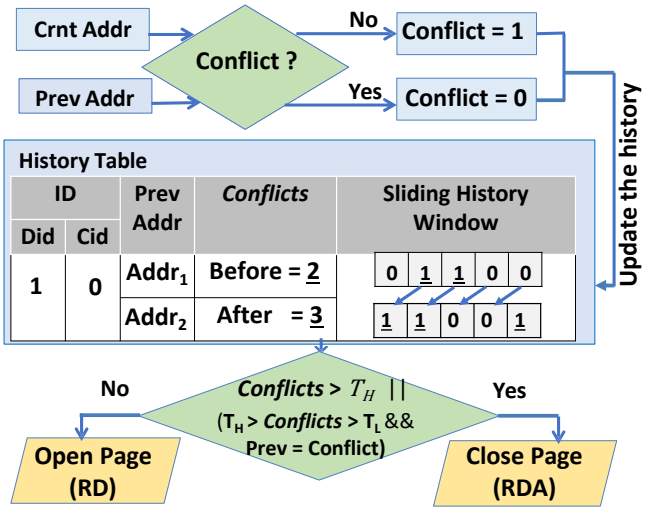


Figure 13: The Conflicts Tracking Window Mechanism.

request address and the stream's stride. The read data for the future subsequent addresses is stored in a buffer denoted as the iBuffer. Each DRAM bank has its own iBuffer. The iBatch *depth* determines the number of future requests. iBuffer *size* defines the maximum iBatch future requests number to be accessed. Fig. 14 shows the iBatch module architecture. Each iBuffer's entry contains the address, data, and two flags: W (Waiting) and R (Requested). The W flag is set to 1 if an entry is waiting for its data from the DRAM. If an incoming demand request hits in an entry with $W = 1$, it will wait until valid data comes back and 'W' reset. The R flag labels an entry that has already been requested, such that when valid data arrives, the MC can respond immediately to the LLC. When a request for subsequent elements of the stream arrives at the MC, it will hit in the iBuffer and the W flag is cleared. Fig. 15 depicts the iBatch module operation flowchart. The iBatch effectively reduces the average DRAM read latency significantly.

Indirect Streams: The iBatch technique can also be applied to indirect streams if they show good locality. To measure the indirect stream's locality, the *conflicts tracking window per indirect*

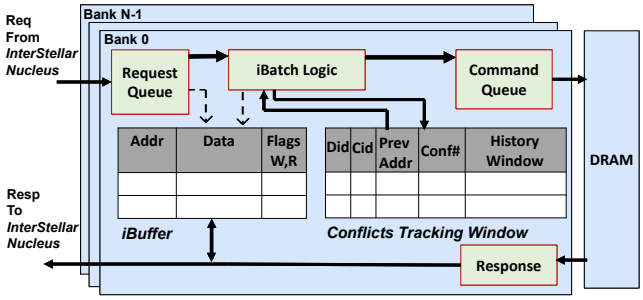


Figure 14: The iBatch Module Architecture.

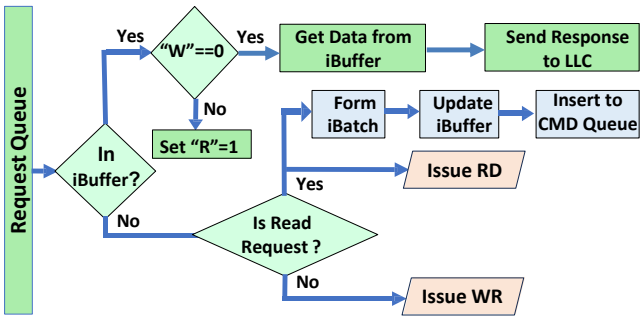


Figure 15: The iBatch Module Flowchart.

stream is employed (see Section 5.1). If the conflicts percentage in the monitored window falls below a certain threshold, this would hint reasonable locality in the indirect stream, and thus a batch for the stream can be formed from its next N forthcoming requests. If the conflict percentage decreases further, batch depth can increase, accommodating higher degrees of locality.

Configurability. iPP and iBatch feature various configurable settings tailored to application needs. In particular, tracking window size can be set for each stream allowing for customized history depths based on stream type, such as larger windows for complex patterns in indirect streams. Additionally, iBatch's depth is configurable per stream enabling system designers to balance locality and minimize interference delays based on stream characteristics.

Scalability. InterStellar was designed to work as a generalized stream-based HW/SW co-design architecture. Towards this goal, we split InterStellar into two main parts. The first is the Nucleus which can be placed at the memory hierarchy where it is best suited to record the conveyed hardware information as well as continue tracking dynamic information at the hierarchy. The second is the component-specific module that enables the component to use InterStellar information to optimize its performance. In our case, we used Nucleus at the LLC cache, while the second was the stream-specific iBatch and iPP policies inside the MC. In case for example, LLC was bankized and distributed several design options are available and the best option depends on the platform tradeoffs. Namely, Nucleus can be placed directly in front of the MC, in this case only one Nucleus is needed (saves area), while it slightly increases the miss delay (since it can no longer be parallelized with LLC tag checking). Another design is to replicate Nucleus with every LLC bank (larger area but better performance). InterStellar is scalable to work with multi-MCs as only one Nucleus is needed. Each MC has its InterStellar extensions of iPP and iBatch. Nucleus will send the appended packet to the target MC based on its address mapping, which will continue to do InterStellar memory optimizations.

6 Implementation

6.1 Compiler Support

The main responsibility of the compiler is to identify streams in the target program and establish the relationship among the streams. To achieve this, a mid-end LLVM compiler pass is developed that implements several steps as follows.

Loop Selection: First, we select target loops that potentially contain stream-based memory access. We first consider all hot loops that contain memory operations based on the loop induction variable (both directly and indirectly) to be eligible. We also use the profile information to only focus on loop nests with relatively large total iteration counts.

Stream Tree Formation: For every load/store instruction in the target loop, we assign a new stream candidate. We then use a Depth-First Search (DFS) algorithm that starts from every memory instruction in the loop and steps through the data dependence graph backwards to form a sequence of instructions used to generate the memory address. Next, our pass constructs *stream dependency trees* from the identified stream candidates in the target loop. For direct streams that only depend on the loop induction variable, we identify their base address and stride values. For the indirect streams we identify their base address and the other streams they depend on (both direct and indirect streams). Multiple streams within a loop can share the same base address, with either different stride values or being dependent on different *parent* streams.

Pruning: We use several criteria to prune stream candidates. First, we select streams with loop-invariant base addresses and stride values. We also ensure the index calculation for stream access uses only simple arithmetic (multiplication and addition by a constant). Currently, we do not support streams that depend on multiple induction variables. Finally, we use profile information to select only the streams that incur significant DRAM accesses. That means, we remove the streams that are mostly resident in different levels of CPU caches. This is done via sample-based profiling, mapping hardware events (DRAM loads/stores) to specific PC addresses. When a stream candidate is removed, their corresponding *stream subtree* is also pruned.

Stream Descriptor Generation: After pruning, the compiler generates stream descriptor *intrinsics* from the stream tree in the LLVM IR code before the loop, usually at the loop's preheader. These intrinsics are then converted to stream configuration CSR instructions in the LLVM backend.

6.2 Hardware Cost

This section presents the area and energy overheads of InterStellar's additions across the memory hierarchy.

Below, we detail the area requirements of each major component inside InterStellar:

1) Descriptors Implementation. On RV64 systems, each RISC-V CSR is 64 bits wide, enabling a 128-bit descriptor to span two

CSRs. Profiling across BMs shows that 32 CSRs are sufficient to store 16 InterStellar descriptors per core, supporting up to 9 streams and 4 loops per context per each core. These descriptors are mapped to existing RISC-V CSR registers, incurring no additional hardware cost beyond architectural support.

Nucleus (Nucleus). The main overhead in **Nucleus** is the Desc Table, which stores static and runtime metadata per stream. Given S streams and C cores, each entry requires 128 bits (static), 2 bits (type), and 40 bits (end VA), resulting in a total size of: $S \times C \times (\log_2 S + \log_2 C + 170)$ bits. For 16 streams and 8 cores, this equates to 2.76 kB. Other logic, including the LLC Request Filter and LLC Pkt Appender, use comparators and simple data path extensions. A 256-entry inverse TLB (TLB^{-1}) is also used, which suffices across all evaluated BMs.

Memory Controller Extension. The memory controller is extended with an iBuffer and a Conflict Tracking Window. Each iBuffer entry holds a 64-byte cache line and 42 bits of metadata. The total size is $R \times B \times 554/8$ bytes for R entries and B banks. With 64 entries and 16 banks, this totals 69 kB, or 1.7% of a 4 MB LLC. The textsfconflict tracker Window consists of 128 compact shift registers for an 8-core, 8-stream system, which is a minimal logic overhead.

Energy Overhead. The dominant energy cost comes from the largest InterStellar's component: The iBuffer. For a rough estimate of the energy consumed when operating on the iBuffer, [49] reports an access cost of 2.4 pJ per 32-bit read/write for an SRAM operating at 2500 MHz. Given a 4 MB working set (1,048,576 accesses), the total energy is: $E_{\text{iBuffer}} = 2.52\,\mu\text{J}$. This is negligible compared to total memory energy consumption, accounting for less than 0.3% for the same working set, as measured using DRAMPower [50].

Despite this minimal energy overhead, InterStellar reduces the major bulk of energy consumption— the DRAM energy—by 24% on average, with reductions reaching up to 60%. This is because batching reduces the number of ACT and PRE commands, thereby lowering the energy associated with DRAM command execution. A full evaluation of DRAM energy savings is presented in Section 9.4.

7 System Considerations

There are several key design considerations for efficient stream-aware memory management, they include the following:

The iBuffer Overhead Placing iBatched lines directly into the LLC removes the need for a dedicated iBuffer, reducing hardware complexity. However, uncontrolled insertion may cause cache pollution by evicting useful data from other streams or the working set. InterStellar makes the batch size user-configurable: smaller batches reduce pollution risk but would limit performance gains.

The Address Mapping In production systems, column bits are preserved to maintain sequential locality, while higher-order bits (bank, bank-group, channel) are randomized to enhance parallelism. This ensures that InterStellar's iBuffer can still exploit column-level locality for prefetching, while benefiting from balanced load under real-world mappings.

Tagging Hardware Prefetch Requests: As shown in Section 9.3, InterStellar enhances memory performance when combined with hardware prefetchers. However, prefetchers often suffer from inaccuracies, which can degrade InterStellar's efficiency by batching large amounts of unnecessary data. To address this, tagging prefetch requests allows InterStellar to distinguish between demand requests and speculative prefetches. By integrating prefetch tagging mechanisms, InterStellar ensures that only demand requests are batched, preventing unnecessary data movement and improving overall memory efficiency.

Dynamic Loop Trip Counts and Strides: The loop start, end values, or strides are either statically known at compile time or computed dynamically at runtime. When known at compile time, the compiler can generate the descriptors statically. However, when these values are computed at runtime, they must be stored in registers. The link descriptor illustrated in Section 4.1 facilitates this process by ensuring that dynamically determined loop parameters can be read at run-time, enabling stream-aware memory optimizations. ***TLB^{-1} Alternative Solution:*** Instead of placing the Nucleus alongside the LLC, an alternative approach is to position the Nucleus before the TLB, operating in the VA domain. This enables early stream tagging before address translation occurs, offering several advantages: 1) *Eliminates the need for TLB^{-1}*, simplifying the memory management pipeline. 2) *Seamless integration with AXI interfaces*, allowing the propagation of the stream ID without modifying the hardware interfaces. However, this approach introduces a trade-off: Since the Nucleus must now operate in the VA domain, it requires an instance per core rather than a shared global engine. Although this eliminates the need for TLB^{-1}, it may pose scalability challenges as the count of cores increases.

Table 1: Evaluation Parameters

OoO Core@2.4GHz (1, 4, 8 Cores)	8-wide Superscalar , 64 IQ 32 LQ & SQ , 192 ROB entries.
InterStellar Engine	iBuffer = 4 kB/bank. iPP Unit = 256 registers
L1-I / L1-D	64 kB - 4 way - 3-cycles latency 16 MSHRs
L2	512 kB - 8 way - 10-cycles latency 32 MSHRs
L3 (Shared LLC)	2 MB - 16 way - 30-cycles latency 64 MSHRs
HW Prefetchers	AMPM - BOP - IMP - SlimAMPM - SPP - Stride (Stream)
DRAM Organization	Micron DDR4 2400MHz, 8GB x8.
DRAM Addressing	RoBaBgColRaCh
DRAM Scheduling	FRFCFS Priority Hit

8 Experimental Environment

Platform. We implemented InterStellar in GEM5 integrated with Ramulator. The RISC-V system uses the high-performance Neoverse V1 memory hierarchy [51], and InterStellar parameters are shown in Table 1. We acknowledge that gem5+Ramulator accuracy has been revisited recently [52]. To validate our setup, we performed basic sanity checks. For unloaded latency, we used SPMV (pointer-chase access) and confirmed latencies align with expected DRAM

Table 2: Used BMs ID, Name, Suite, Stream type (T): (DS: Direct with small stride, DL: Direct with large stride, DM: Direct with mixed small and large strides, and MX: Mixture of direct and Indirect stream.), # of streams (S), and Pattern (P).

ID	BM	Suite	T	S	P	ID	BM	Suite	T	S	P	ID	BM	Suite	T	S	P
A	atax	ALGEBRA	DS	1	2D	B	bicg	ALGEBRA	DS	1	2D	C	daxpy	LAPACK	DS	2	1D
D	ddot	LAPACK	DS	2	1D	E	doitgen	ALGEBRA	DS	3	3D	F	dscal	LAPACK	DS	1	1D
G	fdtd-2d	Stencil	DS	3	2D	H	floyd-warshall	Dyn. Prog.	DS	1	2D	I	gesummv	BLAS	DS	5	2D
J	heat-3d	Stencil	DS	2	3D	K	Jacobi-1d	Stencil	DS	2	1D	L	Jacobi-2d	Stencil	DS	2	2D
M	memcpy	Memory	DS	2	1D	N	scusumbkn	LAPACK	DS	2	1D	O	seidel-2d	Stencil	DS	4	2D
P	daxpys	LAPACK	DL	2	1D	Q	ddots	LAPACK	DL	2	1D	R	dscals	LAPACK	DL	1	1D
S	scusumbkns	LAPACK	DL	2	1D	T	gemver	BLAS	DM	9	2D	U	mvt	BLAS	DM	5	2D
V	knn-1d	Data Mining	MX	3	2D	W	knn-2d	Data Mining	MX	5	2D	X	knn-3d	Data Mining	MX	7	2D
Y	spmv	HPCG	MX	5	2D	Z	histo	Parboil	MX	2	2D	α	stencil	Parboil	MX	2	2D
β	needle	Rodinia	MX	5	2D	γ	srad_v2	Rodinia	MX	7	2D	δ	histogram	Phoenix	MX	4	1D
ϵ	linear regression	Phoenix	DS	2	2D	ζ	mm	Phoenix	DS	3	2D	η	pca	Phoenix	DS	3	2D
κ	reverse index	Phoenix	MX	7	2D	λ	string match	Phoenix	DS	1	1D	μ	word count	Phoenix	MX	2	1D

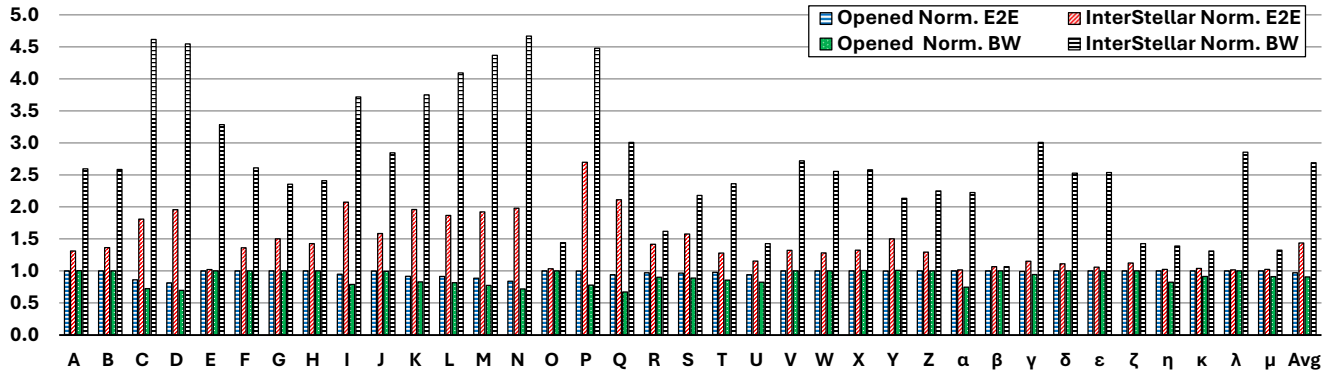


Figure 16: Single core normalized E2E to COTS E2E cycles and normalized bandwidth to COTS bandwidth results.

timing and capture row-buffer/conflict effects. For loaded latency, we used *daxpy* BM to stress bandwidth and observed results consistent with the theoretical and sustained bandwidth of the device. Additionally, handcrafted microbenchmarks verified row-buffer hit/miss latencies and bank-level parallelism. These checks give us confidence our platform exhibits realistic DRAM behavior.

Baseline controllers. We compare InterStellar against four solutions, two of them are COTS-based solutions: 1) Opened page policy (FR-FCFS) and 2) COTS adaptive page policy [38], while the other two are hardware-only research solutions: 3) Parallelism aware batch scheduling (PARBS) [53], and 4) Blacklisted memory scheduler (BLISS) [54]. We also evaluate using six different HWPs: AMPM, BOP, IMP, slimAMPM, SPP, and Stride HWP[41, 55–57]. **PARBS and BLISS** are hardware-only techniques that utilize the batching concept, which makes them closely related to our solution. These techniques cluster requests based on the core ID leading to improvements over the baseline. However, since one core can run multiple streams, PARBS and BLISS do not address conflicts within streams from the same core. While PARBS and BLISS target multi-core systems, our solution improves single and multi-core setups. So, we compare with them in multi-core scenario for consistency.

Evaluation Metrics. We use 3 metrics for evaluation:
1) *Normalized end-to-end (E2E) Speedup of Policy P*, where we define *E2E Speedup*$_P$ as the ratio between Execution time of COTS adaptive page Policy and Execution time of Policy P.
2) *Normalized Bandwidth (BW) Improvement of Policy P*, For this metric, we define memory BW (BW_P) as the ratio between the total bytes exchanged with DRAM and the total active DRAM cycles duration for policy P. *BW Improvement*$_P$ is the ratio between BW_P and COTS's BW.
3) *Normalized Energy Consumption of Policy P*, Energy is estimated using DRAMPower [50] with traces generated by Ramulator. The energy consumption of policy P is normalized versus COTS.

Evaluation benchmarks. We use 36 BMs with various memory access patterns to evaluate InterStellar (Table 2). This includes 21 direct stream BMs from PolyBench [58], and 15 mixed stream BMs from Phoenix [59], Parboil [60], Rodinia [61], and HPCG [62]. We also conduct experiments with heterogeneous workloads, where different BMs are executed on different cores. In total, we evaluate eight such combinations (Z1–Z8), as summarized in Table 3.
For the LAPACK BMs, we evaluate four kernels—`daxpy`, `ddot`, `dscal`, and `scusumbkn`—each in two stride modes (Table 2):
1) Fixed stride (C, D, F, N): Running with fixed stride of 8 bytes (one double).
2) Stride sweep (P, Q, R, S): Sweeping strides in the range of 16–4096 bytes; indicated by appending "s" to the kernel name. For example, `daxpy` is the fixed 8-byte case, while `daxpys` is the stride-sweep case (16–4096 bytes).

Table 3: Selected BMs for the mixed-case eight-core setup.

ID	C1	C2	C3	C4	C5	C6	C7	C8	ID	C1	C2	C3	C4	C5	C6	C7	C8
Z1	A	B	F	I	A	B	F	I	Z2	M	I	T	Y	M	I	T	Y
Z3	M	I	V	Y	M	I	V	Y	Z4	M	U	D	Y	C	I	V	F
Z5	A	U	δ	Y	C	ϵ	V	F	Z6	T	N	γ	M	F	B	W	I
Z7	V	C	F	I	V	C	F	I	Z8	V	W	A	B	V	W	A	B

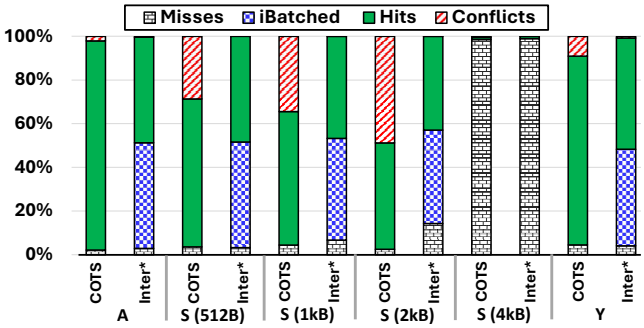


Figure 17: Single core DRAM statistics for selected BMs, where Inter* refers to InterStellar.

9 Evaluation

n this section, we present the evaluation results of InterStellar, including analyses of single-core and multi-core performance, the impact of hardware prefetchers, and DRAM energy consumption.

9.1 Single Core Performance (Breakdown of Performance Gain Sources)

The single core experiments purpose is to evaluate the benefits of InterStellar based on individual BM nature. Fig. 16 shows the normalized E2E speedup and BW improvement on single core for all BMs. InterStellar achieves average normalized speedup of 1.6x and average BW improvement of 3.3x. The speedup is primarily due to three factors: (a) iBatch guard against misses happening when reading multiple streams data sequentially, therefore it reduces most of the misses to iBatches. (b) iPP manages to convert most of the DRAM conflicts to misses, and c) iBatch is able to guard consecutive read accesses to an opened DRAM row from write conflicts, which effectively converts some DRAM conflicts into DRAM hits. This is shown in Fig. 17. To understand how InterStellar improves DRAM stats, let's discuss InterStellar optimizations on the following categories: **1) Direct Stream with Small Stride** happens when direct stream runs with strides less than cache line size. For example: BMs ***A-O***. in Table 2. Such BMs show high locality, if only one stream is running such as BM ***A***, then a huge hit rate is expected at the DRAM, but with multiple streams running, then more conflicts and considerable misses arise such as with BMs ***L-I***. Fig. 17 shows how InterStellar manages to convert most of the hits to iBatches, and eliminates most of the conflicts and misses happen in this case.
2) Direct Stream with large Strides occurs when a direct stream runs with strides greater than cache line size (e.g., BMs: ***P-U***). As stride size increases, conflicts also rise. iPP minimizes these conflicts, enhancing speed. However, iBatch performs best with smaller strides, effectively handling up to 512 Bytes. Beyond this, doubling of the stride halves ibatchable requests but as misses dominate, this allows for iPP to influence speedup. This is shown in Fig. 17 with sweeping stride from 512B to 8 kB with ***S*** BM. The maximum iPP-only speedup of 1.25x is seen at a 4 kB stride in the ***Q*** BM.
3) Indirect streams mix direct and indirect streams. For example BM: ***Y***. iPP excels with indirect streams by managing conflicts, while batching protects direct stream counters enhancing iPP's effectiveness on indirect streams.

9.2 Multi-Core Results

Fig. 18 shows the normalized speedup against COTS adaptive DRAM page policy for **8-cores** systems tested with InterStellar, PARBS, BLISS, and opened page policy.

InterStellar achieves a speedup up to 2.7x. In comparison, PARBS achieves speedup up to 1.6x, while BLISS achieves speedup up to 1.2x. InterStellar offers superior speedup compared to both PARBS and BLISS. Fig. 19 illustrates the InterStellar's speedup components for the 8-cores case. While PARBS and BLISS batch requests based on the source core, this enhances the performance when running different workloads on different cores and also excels when workloads running only single stream (like Z1 case). PARBS and BLISS convert most of misses to hits but also introduce conflicts when switch between different cores or within the same core if there are multiple conflicting streams. On the other hand because InterStellar works on stream basis, it effectively converts most of conflicts and misses to hits and iBatches.

9.3 Effect of Hardware Prefetchers

The purpose of these experiments is to evaluate the InterStellar both against prefetching techniques to illustrate the impact of the points discussed in Section 3. Additionally, to show that InterStellar is also orthogonal to existing cache prefetch techniques and is not a replacement of them, we show the results when both InterStellar as well as cache prefetching is enabled. We implemented InterStellar on an 8-core system without HWPs and over the six HWPs in Table 1. Fig. 20 shows speedups of up to 3x against HWPs and 2.5x on top of them. Fig. 21 shows the BW improvement of InterStellar without HWP over COTS with HWP. The average improvement is 2.02x due to the reduced active DRAM cycles resulted from lowever DRAM conflicts withInterStellar. InterStellar without HWP outperform COTS with HWP collectively due to observations discussed in Sec 3.1.

9.4 DRAM Normalized Energy Consumption Results

Fig. 22 shows normalized energy consumption across selected BMs for InterStellar (without HWP), COTS with HWP, PARBS, and BLISS. InterStellar achieves the highest energy reduction over all BMs. On average, 24% compared to COTS without HWP and 17% compared to COTS with HWP. This improvement results from fewer conflicts, which reduce command count and overall DRAM energy. With BMs that contains multiple streams like *daxpy, ddot*, and *scusumkbn*, the energy reduction reaches up to 60% comparing COTS without HWP and up to 54% comparing to COTS with HWP.

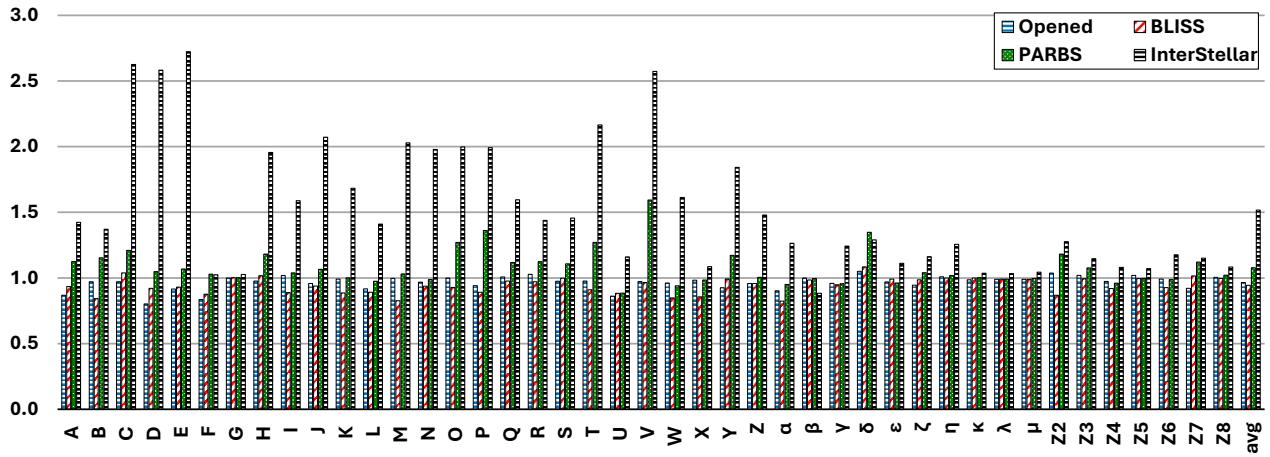


Figure 18: Normalized E2E speedup to COTS policy E2E cycles for all BMs in Eight-cores Setup.

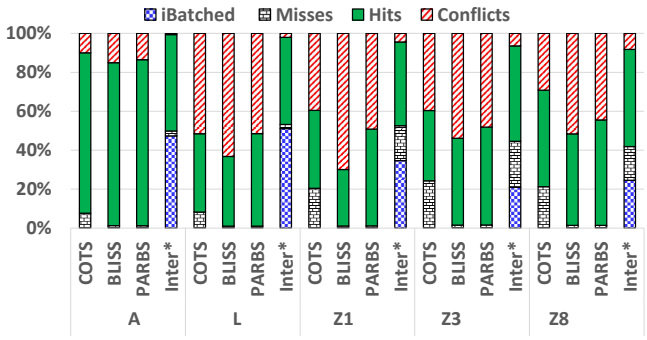


Figure 19: Distribution of DRAM request types and their percentages for selected BMs on an 8-core system, where Inter* refers to InterStellar.

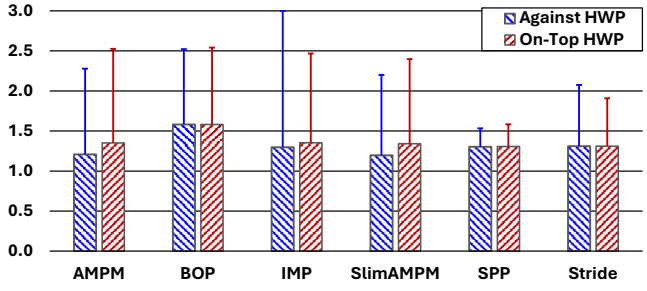


Figure 20: 8-core E2E speedup of InterStellar vs. HWP, averaged for all BMs.

10 Related Work

Since we already covered the stream-based related work in Section 1, we elaborate in this section on two directions of work: 1)

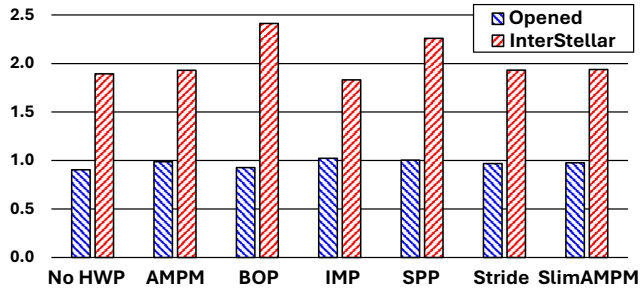


Figure 21: 8-core Average normalized BW improvement of InterStellar vs. HWPs, averaged across all BMs.

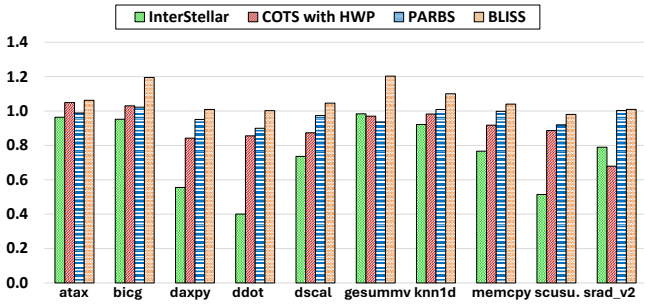


Figure 22: 8-core Normalized energy consumption reduction compared to COTS for selected BMs.

other generalized HW/SW frameworks, and hardware-only memory controller solutions. **HW/SW Co-design General Frameworks.** Täko gives more control on memory policies to software for certain memory regions (tagged as phantom data). Cache hardware triggers software callbacks in response to cache operations (misses, evictions, and writebacks) on phantom data. In turn, the programmer can run specialized software routines upon these calls based on the data being accessed. Modifications include a reconfigurable dataflow hardware engines to execute these routines.

XMem [15, 63], and MetaSys [64] follows a different approach by tagging memory regions with extra semantic information. They utilize a specific region of virtual memory (VM) to store hardware-software communication messages known as Atoms. These messages contain metadata related to various data structures. The ISA is expanded to include new instructions that facilitate the creation, mapping, and unmapping of atoms. XMem, MetaSys, and Täko annotate memory regions requiring both ISA modification and OS support. They also require extra delay cycles to fetch the tagging information or for the call back execution in case of Tako. InterStellar doesn't require these changes facilitating adoption with the current programming paradigm. Also, in mentioned solutions if data structures overlap in a page because they annotate a page, they can't differentiate, but in InterStellar granularity depends on the data structure.

Hardware Memory Controller Solutions. On the other side, there are many DRAM MC optimizations, through profiling data dynamically in the MC , batching and prioritizing some memory requests can achieve better fairness and performance for multi-core systems [54, 65–71] , or utilize reinforcement learning towards self optimizing DRAM controller [72] which manages to reach 1.3x average speedup over FR-FCFS legacy controllers.

11 Conclusion

We propose InterStellar as a general stream-based hardware-software co-design framework. InterStellar opens up several research directions to explore different memory hierarchy optimizations that don't require ISA modifications and can operate seamlessly without intervention from programmers. We use InterStellar to intelligently fuse stream data fetching from the off-chip memory with the memory controller scheduling and page management policies. Evaluation show that InterStellar considerably outperforms hardware-only solutions in terms of the end to end performance, as well as DRAM bandwidth, and DRAM energy . Additionally, it shows that while InterStellar also shows better performance than cache prefetching, it also provides additional performance gains if enabled on top of such prefetchers.

References

[1] Amol Bakshi, Jean-Luc Gaudiot, Wen-Yen Lin, Manil Makhija, Viktor K Prasanna, Wonwoo Ro, and Chulho Shin. Memory latency: to tolerate or to reduce. In *12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). Invited Ppaper*, 2000.

[2] Carlos Carvalho. The gap between processor and memory speeds. In *IEEE International Conference on Control and Automation (ICCA)*, 2002.

[3] Stijn Eyerman, Wim Heirman, and Ibrahim Hur. Dram bandwidth and latency stacks: Visualizing dram bottlenecks. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022.

[4] Chia-Yin Liu, Hasitha Muthumala Waidyasooriya, and Masanori Hariyama. Data-transfer-bottleneck-less architecture for fpga-based quantum annealing simulation. In *7th International Symposium on Computing and Networking (CANDAR)*, 2019.

[5] JESD79-4A: DDR4 SDRAM. Standard, JEDEC, Arlington, VA, 2021.

[6] JESD79-3D: DDR3 SDRAM. Standard, JEDEC, Arlington, VA, 2012.

[7] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *ACM/IEEE 19th International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[8] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomputing Frontiers and Innovations*, 1(3), 2014.

[9] Haocong Luo, Taha Shahroodi, Hasan Hassan, Minesh Patel, A. Giray Yağlıkçı, Lois Orosa, Jisung Park, and Onur Mutlu. Clr-dram: A low-cost dram architecture enabling dynamic capacity-latency trade-off. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[10] Hasan Hassan, Minesh Patel, Jeremie S. Kim, A. Giray Yaglikci, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, and Onur Mutlu. Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability. In *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[11] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *ACM/IEEE 34th Real-Time Systems Symposium (RTSS)*, 2013.

[12] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS '16)*, 2016.

[13] Mohamed Hassan. Reduced latency dram for multi-core safety-critical real-time systems. *Real-Time Systems*, 56(2), 2020.

[14] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[15] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B. Gibbons, and Onur Mutlu. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[16] E.H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *International Conference on Parallel Processing (ICPP)*, pages 281–284, 1994.

[17] Tzi cker Chiueh. Sunder: a programmable hardware prefetch architecture for numerical loops. In *ACM/IEEE Conference on Supercomputing (SC)*, 1994.

[18] Zhenlin Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C.C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *ACM/IEEE 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[19] S.P. Vander Wiel and D.J. Lilja. A compiler-assisted data prefetch controller. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 1999.

[20] Prabhat Jain, Srini Devadas, and Larry Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. *Computation Structures Group, Laboratory for Computer Science CSG Memo*, 2001.

[21] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. Stream floating: Enabling proactive and decentralized cache optimizations. In *ACM/IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[22] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *ACM/IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[23] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *ACM/IEEE 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[24] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. Täkö: A polymorphic cache hierarchy for general-purpose optimization of data movement. In *ACM/IEEE 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[25] Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. Pacman: Program-assisted cache management. In *International Symposium on Memory Management (ISMM '13)*, ISMM '13, 2013.

[26] Abhisek Pan and Vijay S. Pai. Runtime-driven shared last-level cache management for task-parallel programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[27] Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericas, and Per Stenstrom. Radar: Runtime-assisted dead region management for last-level caches. In *ACM/IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[28] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[29] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *ACM/IEEE 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[30] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[31] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores. *IEEE Transactions on Computers (TC)*, 70(2), 2020.

[32] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration. In *ACM/IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[33] Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. Infinity stream: Portable and programmer-friendly in-/near-memory fusion. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[34] James M Dodd. Adaptive page management. *Google Patents. US Patent 7,076,617 B2*, 2006.

[35] Rajinder Gill. Everything you always wanted to know about SDRAM (memory): But were afraid to ask. *AnandTech*, 2010.

[36] Intel xeon processor x5650 12m cache 2.66 ghz 6.40 gts intel qpi product specifications, 2010.

[37] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. Reverse-engineering embedded memory controllers through latency-based analysis. In *ACM/IEEE 21st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[38] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. Exposing implementation details of embedded dram memory controllers through latency-based analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(5), 2018.

[39] Mohsen Ghasempour, Aamer Jaleel, Jim D. Garside, and Mikel Luján. Happy: Hybrid address-based page policy in drams. In *ACM/IEEE 2nd International Symposium on Memory Systems (MEMSYS '16)*, MEMSYS '16, NY, USA, 2016.

[40] Hamid Nasiri, Saeed Nasehi, and Maziar Goudarzi. Evaluation of distributed stream processing frameworks for iot applications in smart cities. In *Journal of Big Data*, 2019.

[41] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. In *ACM/IEEE 48th Annual International Symposium on Microarchitecture (MICRO)*, 2015.

[42] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[43] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Sparsecore: stream isa and processor specialization for sparse computation. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[44] Andrey S Molyakov. Secured supercomputer technologies in russia: Functional computing units based on multithread-stream cores with specialized accelerators. In *7th International Congress on Information and Communication Technology (ICICT)*, 2022.

[45] Antoine Petitet, R. Whaley, Jack Dongarra, and A. Cleary. Hpl 2.0 – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. 2008.

[46] Anderson, E. et al. LAPACK Users' Guide. https://www.netlib.org/lapack/, 1999. Accessed: 2025-06-03.

[47] Andrew Waterman, Krste Asanovi, and John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* RISC-V International, 20211203 edition, 2021.

[48] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. Perforated page: Supporting fragmented memory allocation for large pages. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[49] Rupesh D. Kadhao, Siddharth R. K., Nithin Kumar Y. B., Vasantha M. H., and Devesh Dwivedi. A 2.5 ghz, 1-kb sram with auxiliary circuit assisted sense amplifier in 65-nm cmos process. In *36th International Conference on VLSI Design and 22nd International Conference on Embedded Systems (VLSID)*, 2023.

[50] Chandrasekar, Karthik and Weis, Christian and Li, Yonghui and Goossens, and Goossens, Kees. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. http://www.drampower.info, 2025. Accessed: 2025-06-03.

[51] Andrea Pellegrini. Arm neoverse n2: Arm's 2nd generation high performance infrastructure cpus and system ips. In *IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[52] Pouya Esmaili-Dokht, Francesco Sgherzi, Valéria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adrià Armejach, Estanislao Mercadal, Germán Llort, Petar Radojković, Miquel Moreto, Judit Giménez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. A mess of memory system benchmarking, simulation and application profiling. In *ACM/IEEE 57th International Symposium on Microarchitecture (MICRO)*, 2024.

[53] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch-scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM/IEEE 35th International Symposium on Computer Architecture (ISCA)*, 2008.

[54] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *IEEE 32nd International Conference on Computer Design (ICCD)*, 2014.

[55] Pierre Michaud. Best-offset hardware prefetching. In *ACM/IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[56] Vinson Young and A Krishna. Towards bandwidth-efficient prefetching with slim ampm. *The 2nd Data Prefetching Championship*, 2015.

[57] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *ACM/IEEE 49th Annual International Symposium on Microarchitecture (MICRO)*, 2016.

[58] Louis-Noel Pouchet and Tomofumi Yuki. PolyBench/C 4.2.1. http://polybench.sourceforge.net/, 2016. [Online; accessed 1-July-2023].

[59] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++ modular mapreduce for shared-memory systems. In *2nd International Workshop on MapReduce and its Applications*, 2011.

[60] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. 2012.

[61] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[62] Jack Dongarra, Michael Heroux, and Piotr Luszczek. A new metric for ranking high performance computing systems. *National Science Review*, 3(01), 2016.

[63] Onur Mutlu. Intelligent architectures for intelligent computing systems. In *Design, Automation and Test in Europe Conference (DATE)*, 2021.

[64] Nandita Vijaykumar, Ataberk Olgun, Konstantinos Kanellopoulos, F Nisa Bostanci, Hasan Hassan, Mehrshad Lotfi, Phillip B Gibbons, and Onur Mutlu. Metasys: A practical open-source metadata management system to implement and evaluate cross-layer optimizations. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(2), 2022.

[65] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *ACM/IEEE 43rd Annual International Symposium on Microarchitecture (MICRO)*, 2010.

[66] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers. *IEEE Micro*, 29(1), 2009.

[67] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *ACM/IEEE 40th Annual International Symposium on Microarchitecture (MICRO)*, 2007.

[68] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *ACM/IEEE 16th International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[69] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ACM/IEEE 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[70] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *ACM/IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[71] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4), 2016.

[72] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2008.