

SecBCache: Evaluating Banked Interleaved Word-Granularity Sector Cache for gather/scatter Operations on Sparse Data

Nazmus Sakib

nsakib6@nmsu.edu

New Mexico State University

John Shalf

Lawrence Berkeley National Laboratory

Abdel-Hameed A. Badawy

Florida International University

Research Article

Keywords:

Posted Date: October 30th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11543536/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

SecBCache: Evaluating Banked Interleaved Word-Granularity Sector Cache for gather/scatter Operations on Sparse Data

Nazmus Sakib^{1*}, John Shalf² and Abdel-Hameed A. Badawy³

¹ Klipsch School of ECE, New Mexico State University, Las Cruces, NM, 88003, USA.

² Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA.

³Knight Foundation School of Computing and Information Sciences, Florida International University, Miami, FL, 33199, USA.

*Corresponding author(s). E-mail(s): nsakib6@nmsu.edu;
Contributing authors: jshalf@lbl.gov; abadawy@fiu.edu ;

Abstract

Sparse workloads in scientific computing, graph analytics, and machine learning rely on indirect memory accesses ($A[B[i]]$). These accesses have little spatial locality, so caches use few words of each line they fetch. We present SecBCache, a word-granularity sector cache whose interleaved banks can serve different elements of a vector gather/scatter. It fetches and evicts 8 B subsectors that share one tag per sector. We evaluate SecBCache in gem5 with breadth-first search (BFS) and sparse general matrix-matrix multiplication (SpGEMM) on 16 SuiteSparse datasets. Compared with an 8-way baseline cache, SecBCache reduces SRAM data-array writes by a median of 63% for BFS and 80% for SpGEMM. It also lowers L1 dynamic energy for every dataset. Its area is at most 2.46% of the cache hierarchy. An analytical bank-conflict model conservatively estimates median memory-access delay reductions of 26% for BFS and 50% for SpGEMM. These are modeled delays, not application speedups.

Keywords: Sector Cache, Sparse Memory, Gather/Scatter, Banked Memory, Vector Architecture

1 Introduction

Across high-performance computing clusters and consumer edge devices alike, computation has become fundamentally data bound. While hardware designs vary widely across these domains, both face the same core imperative: maximizing throughput while minimizing power consumption. This shift has intensified the memory-wall problem [1], as system performance and energy efficiency now depend far more on data movement than on raw arithmetic throughput across machine learning, graph analytics, and scientific simulations [2–5].

In addition, a key issue is the growing prevalence of sparse data structures utilized in scientific and engineering simulations [6–8], graph analytics [9–12], and machine learning [13–15] applications, where the accessed memory does not exhibit a predictable pattern. These datasets are often accessed indirectly, with one load depending on the result of another ($A[B[i]]$). In such accesses, $B[i]$ is the independent load that stores the column index or pointer (contiguous in memory), while $A[B[i]]$ is the dependent load that retrieves the actual non-zero data value without a predictable access pattern [16].

These workloads contain mostly zero values that do not contribute to the computation. Compression algorithms such as Compressed Sparse Row (**CSR**) [17, 18] and Compressed Sparse Column (**CSC**) [19, 20] transform these sparse structures into a dense format comprising non-zero elements, which can be processed by CPUs with indirect memory access (*e.g.*, $A[B[i]]$).

Similar concepts apply to graph traversal [11]. We note that indirect memory accesses are common in many HPC applications where the underlying data structure is not sparse, but the access patterns are often complex and irregular [2]. In fact, prior studies have shown that indirect memory access instructions dominate the execution time and the total number of instructions in numerous HPC simulations [2, 8, 21, 22]. Because these addresses are data-dependent, they show little spatial locality, so a conventional multi-word cache line often delivers only a fraction of useful words, wasting bandwidth and energy in a memory system that is already the bottleneck [1, 2, 23].

Traditionally, major CPU vendors, such as Intel, AMD, and ARM, have utilized multi-word cache lines to exploit the spatial and temporal locality inherent in the memory access patterns of common applications [24]. This cache design optimization, combined with various algorithms and compiler-level optimizations, has helped reduce memory access latency while maintaining low cost and energy consumption. Additionally, Out-of-Order CPUs have enough issue width and load/store units to sustain multiple in-flight memory requests to extract memory-level parallelism (MLP). However, this design principle is being challenged by an increasing number of modern workloads that exhibit unpredictable access patterns. Indirect memory accesses often suffer from poor temporal and spatial locality, and this incurs large cache misses that result in poor performance.

Vectorization is the process by which compute units exploit parallelism by executing a single instruction on multiple data (SIMD). Most modern CPUs have vector ISA extensions, and popular open-source compilers [25] can emit vector assembly instructions. Although SIMD is particularly useful for streaming memory with a

known stride, it can also boost performance for indirect memory access via vectorized gather(load)/scatter(store) instructions. However, traditional memory, specifically cache, still suffers from underutilized bandwidth and unnecessary energy consumption from redundant data movement due to its multi-word cache line design, where very few words that are fetched are actually utilized in arithmetic computations [2]. For example, even though Intel’s Skylake processors can load an entire cache line into their vector registers [26] in a single cycle, there are computational kernels with sparse data and irregular memory access, where vector gather/scatter instructions use less than 50% of the words in a cache line on average [23].

To reduce data movement and memory-system power while maintaining high bandwidth utilization, especially at the first level of the memory hierarchy, the simplest solution is to load and store only the words needed for computation. However, a fine-grained (word-granularity) design either adds to the area of a cache (more tags and metadata) or places a burden on the programmer (*e.g.*, explicit memory allocation).

A scratchpad, which is a memory unit used as temporary storage for frequently used data, often operates at word granularity in special-purpose and embedded systems [27, 28]. In recent years, scratchpads for general-purpose computing have been proposed [29], along with their potential benefits for sparse data processing [24]. Nevertheless, they need carefully written special-purpose code for data placement and hardware circuitry for lookup, eviction, etc. In addition, they cannot sustain the full bandwidth of a wide vector load.

A sector cache is an architecture in which each *sector* (a large cache line) is divided into smaller subsectors that share the same tag [30]. This allows flexibility in the granularity of traffic data to and from lower levels of memory without increasing the tag array size, as only the necessary subsectors are placed in the cache (and evicted when needed). Unlike a regular cache line with distinct *tag* bits along with *dirty* and *valid* bits, in a sector cache, subsectors have their own *dirty* and *valid* bits but common *tag* bits.

A sector cache does not need to fetch (or replace) large cache lines from lower-level memory. Instead, it can fetch (or evict) only the necessary subsectors, reducing the data transferred and written. Unlike a scratchpad, it does not require individual *tag* bits for individual words (if subsectors are word-sized), so the *tag* array size remains smaller (similar to a normal cache if designed properly). This also alleviates the burden on programmers as they do not need to manage the allocation and deallocation of words in a scratchpad.

Furthermore, wide vector registers can support multi-word loads in a single cycle, so the memory-level parallelism will not be leveraged solely by accessing single words. In order to achieve maximum bandwidth while keeping traffic and power consumption low for vector processing, interleaved banked memory is a well-established solution [31–33]. Although banked memory may require additional hardware for conflict-free address generation and multi-ported design, interleaved banked memory remains an effective way to exploit SIMD parallelism and sustain high bandwidth for sparse workloads at modest energy cost.

In this paper, we propose SecBCache: a word-granularity sector cache design that is organized as word-interleaved banks for indirect memory access, notably for

the gather/scatter operation. SecBCache combines banked, word-interleaved memory for high bandwidth with a shared-tag sector cache for fine-grained access. Rather than introducing ISA extensions or relying on dedicated off-chip accelerators, SecBCache works with existing vector gather/scatter instructions (*e.g.*, ARM SVE2 [34], Intel AVX-512 [35]). Because gather/scatter instructions generate a clustered burst of addresses over multiple cycles, SecBCache leverages this temporal clustering to demonstrate the potential advantage of concurrent bank access.

We implement SecBCache in gem5 [36, 37], a cycle-level computer architecture simulator, and measure line utilization, SRAM write volume, and evictions directly in simulation. Because gem5 does not let multiple memory requests from the core reach the L1 in the same cycle, it cannot model concurrent accesses to different banks. We therefore estimate the memory-access delay of the banked design with an analytical bank-conflict model. The model groups the simulated request stream into windows of eight requests and computes a Conservative and an Ideal bound on the delay of each window. These delays are model estimates, not measured execution time.

The key contributions of this work are the following:

1. Design a word-granularity sector cache that fetches, fills, and evicts only the requested words.
2. Develop programmer-transparent cache-residency management: unlike a scratch-pad, SecBCache needs no explicit word-level allocation or eviction, uses the same number of tags as a conventional cache of the same size, and keeps a valid and a dirty bit per subsector instead of per line.
3. Use word-interleaved banking so that the elements of a gather or scatter can be served by different banks.
4. Develop an analytical bank-conflict model that estimates the memory-access delay of SecBCache from the simulated request stream under a Conservative and an Ideal bound, and apply it to 16 real-world SuiteSparse datasets.
5. Estimate the area, leakage power, and dynamic energy of the design with CACTI [38]: the area of SecBCache is at most 2.46% of the evaluated cache hierarchy, and its L1 dynamic energy is lower on every dataset.

The rest of this paper is organized as follows: Section 2 reviews prior work on accelerating indirect memory access, vector architectures and banked memory, and sector caches. Section 3 presents the SecBCache design and the analytical bank-conflict model used to bound its memory-access delay. Section 4 describes the gem5 setup, the baseline and SecBCache configurations, and the BFS and SpGEMM benchmarks. Section 5 reports results on cache line utilization, SRAM write volume and evictions, bank conflicts and modeled delay, MSHR demand, and area and energy. Section 6 discusses how SecBCache could be integrated into a real system, and Section 7 concludes.

2 Related Work

Prior work on accelerating indirect and sparse memory access by enhancing existing hardware can be categorized into two groups based on where the proposed modification is placed in the memory hierarchy: on-chip, in or beside the cache (Section 2.1), and

near or inside memory (Section 2.2). Section 2.3 then turns to vector architectures and the banked, interleaved memories that sustain their bandwidth, and Section 2.4 reviews sector caches: the two ideas SecBCache builds on directly.

2.1 On-Chip Architectures

Several proposals modify the on-chip cache to accelerate indirect or sparse access. Compute Caches [39] proposed SRAM bitline circuitry to perform bulk bitwise operations inside the cache array. HTA [40] restructures the layout of hash table data to improve locality within an otherwise conventional cache. Terminus [41] adds dataflow hardware beside the core to support read and update operations on sparse data structures. Neural Cache [42] reconstructs existing cache arrays for inference computation. SortCache [23] builds a sorted key-value structure into the cache to accelerate sparse reductions.

Beyond cache-array restructuring, hardware indirect prefetchers—such as the Indirect Memory Prefetcher (IMP) [43], the Instruction Pointer Classifying Prefetcher (IPCP) [44], and local-delta prefetchers like Berti [45]—track index tables, instruction signatures, or page-level delta patterns to speculatively prefetch non-contiguous targets into lower cache tiers. Although effective at hiding memory latency for predictable index streams, these prefetchers are speculative mechanisms built on conventional 64 B line hierarchies, so every indirect miss still fetches and fills a full line, much of which may go unused. SecBCache instead allocates and fills single words directly at the L1, without speculation.

2.2 Near/In Memory Architectures

The second group places the computation off the critical path of the core, closer to or inside the memory system. DIVA [46] pairs a conventional host processor with processing-in-memory logic distributed across DRAM chips. SPiDRE [47], DMX [48], SpZip [49], GS-DRAM [50], and the Data Rearrangement Engine [51] are off-chip accelerators that support gather and scatter memory operations through dedicated hardware layouts. DX100 [5] extends this direction with an accelerator adjacent to the shared DRAM controller that reorders, coalesces, and interleaves indirect requests to improve bandwidth utilization.

While near-memory architectures and standalone accelerators deliver impressive simulated speedups, they invariably introduce dedicated custom logic for index reordering and temporary storage. Crucially, they require bespoke programming models, non-standard ISA extensions, or specialized hardware offloading. This high software adoption barrier—combined with the architectural friction of deploying non-standard hardware—has largely prevented these designs from reaching mainstream CPU adoption.

2.3 Vector Architectures and Banked Memory

Vector architectures largely avoid this trade-off, since gather and scatter access has been part of the instruction set since the 1980s in both industrial and academic vector machines [52, 53]. Exploiting the vector ISA extensions does not require a new

programming model. The Tarantula project [33] showed that a banked cache tightly coupled to a vector unit can sustain the bandwidth demanded by the gather and scatter instructions using an address reordering scheme detailed separately by Seznec and Espasa [31]. That evaluation assumed the cost and power budget of a 65nm process [33], and current transistor densities extend well beyond it [2], leaving more area and power available for the memory-side logic required by a banked sectored cache.

Modern ISAs and domain-specific accelerators continue this trajectory by incorporating hardware support for irregular and sparse operations. ARM’s Scalable Vector Extension (SVE/SVE2) provides native gather (LD1D) and scatter (ST1D) primitives [34, 54], though execution remains limited by L1 line serialization and bank conflicts under indirect addressing. In parallel, modern GPU architectures, such as NVIDIA’s Ampere generation, introduce 2:4 structured sparsity in Tensor Cores [55, 56] to double compute throughput. However, structured sparsity requires strict offline data reformatting and does not support arbitrary indirect lookups. SecB-Cache complements these compute-level advances by restructuring the L1 cache to reduce the cost of non-contiguous accesses.

Sustaining vector bandwidth through banked, interleaved memory is itself a long-studied problem. Rau [57] proposed pseudo-random address-to-bank mapping to avoid the conflicts that plain modulo interleaving produces for power-of-two strides. Similarly, Seznec and Lenfant [58] applied a related permutation scheme to vector supercomputer memory. The same problem motivates the conflict-free reordering scheme Seznec and Espasa later applied to a banked cache [31], and continues to motivate recent work on conflict management inside vector register files [32]. Commercial vector processors such as the NEC SX-3 [53] relied on wide interleaved memory for the same reason. These designs also share a known limit case: strides that map onto only a handful of banks rather than spreading evenly across all of them. This phenomenon erodes most of the benefit banking offers, and Tarantula’s own conflict-resolution logic was built to handle exactly this pattern for gather and scatter addresses [33].

Multi-banked vector caches and register files rely on address interleaving and dynamic conflict resolution to sustain wide vector throughput. However, early vector cache designs like Tarantula retained multi-word cache lines, causing severe bandwidth pollution when vector lanes accessed sparse, non-contiguous elements. SecBCache pairs word-interleaved banking with per-subsector valid bits, spreading the elements of a gather or scatter across banks while filling only the requested words.

2.4 Sector Caches

Once bandwidth is available for banked memory, the second issue is fetching only the words actually needed rather than an entire cache line. The sector cache [30] addresses this by sharing one address tag across several subblocks, each with its own valid and dirty bits, an idea first described for the cache of the IBM System/360 Model 85 [59]. Later refinements, including Seznec’s interleaved and decoupled sectored caches [60, 61], and more recently the Micro Sector cache for DRAM [62], mostly focused on recovering space wasted by partially valid subblocks rather than on shrinking the subblocks, since a tag array sized for word-sized subblocks was impractical at the transistor densities available at the time. Under current VLSI scaling [63], that same

metadata occupies a small fraction of the cache array, which is what lets SecBCache return to word-sized subsectors instead of pursuing a more elaborate allocation policy.

3 Methodology

This section describes the SecBCache design and how its performance is estimated. Section 3.1 presents the architecture: how a physical address is divided into different fields and how consecutive words are interleaved across eight banks. Section 3.2 then introduces the analytical bank-conflict model and derives Conservative and Ideal bounds on the memory-access delay of a banked SecBCache.

3.1 SecBCache Design

This section details the microarchitectural modifications made to the gem5 sector cache baseline, including the subsector address breakdown and banked organization.

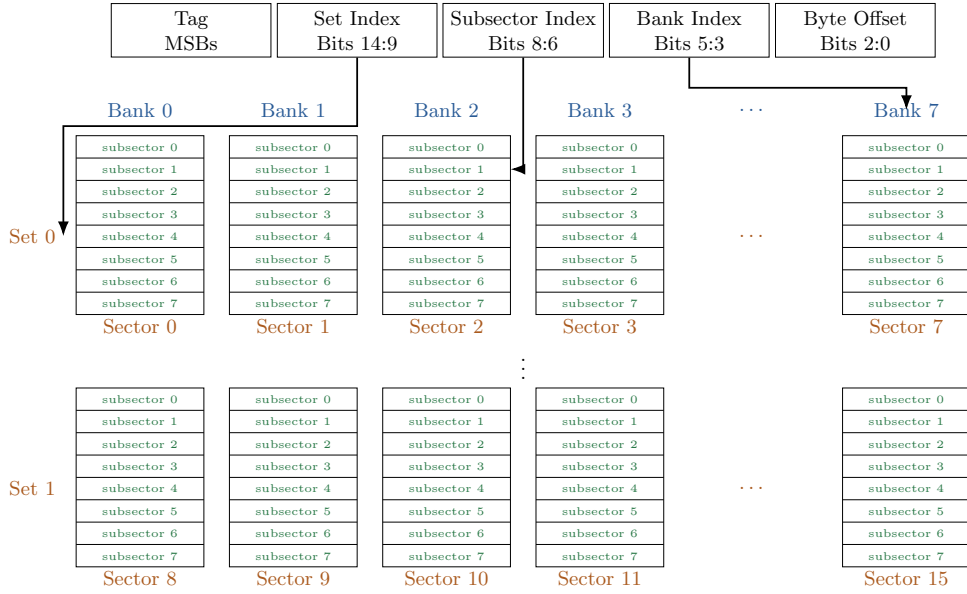


Fig. 1 SecBCache address organization and word-interleaved layout. The address is divided into tag, set index, subsector index, bank index, and byte offset fields. Each sector holds eight 8 B subsectors, which are 64 B apart in the address space, and consecutive words are interleaved across the eight banks.

Figure 1 shows the architectural overview of the SecBCache design with eight banks. The address bits are separated into five fields:

1. *Byte Offset*: SecBCache stores data in 8 B subsectors, and an 8 B subsector is considered a *word* in this design and in the simulation. Therefore, the least significant three bits are the byte offset within a word.

2. *Bank Index*: The number of banks should match the number of words in a vector register so that every element of a vector can be served by a different bank. Popular commodity CPUs for HPC, such as those implementing Intel AVX-512 and ARM SVE2, support 64 B vector registers. gem5 supports several SVE vector register widths (one per simulation). We use 64 B (512 bits) which holds eight 8 B words. Thus, SecBCache has eight banks, selected by three address bits (bits 3 to 5).
3. *Subsector Index*: The purpose of this work is to show the benefits of fine-grained access without increasing the number of *tag* comparisons. A conventional cache with 64 B lines holds eight 8 B words per tag. Therefore, a SecBCache sector has eight subsectors, so there is one tag per eight words as in a conventional cache and no extra *tag* comparisons are required. The subsector index (bits 6 to 8) selects one of the eight subsectors of a sector. Although a sector cache needs additional metadata (a valid and a dirty bit per subsector), the tag array stays close in size to that of a commodity L1 cache.
4. *Set Index and Tag*: The rest of the bits in the address field are reserved for *Set Index* and *Tag*.

Note that bits 6 to 8, rather than bits 3 to 5, are used as the subsector index, so that consecutive words are assigned to consecutive banks. A sector is identified by its tag, set index, and bank index. For example, address 0x00 maps to subsector 0 of the sector in Set 0 of Bank 0, address 0x08 to subsector 0 in Set 0 of Bank 1, and address 0x40 to subsector 1 in Set 0 of Bank 0.

Although SecBCache is particularly suitable for sparse data, this word-interleaved design ensures that contiguous data accesses (*e.g.*, stream array accesses) benefit from SecBCache. Note that a conventional sector cache would hold consecutive words as the subsectors of a sector, whereas in this design the subsectors of a sector are 64 B apart (the number of banks times the word size). Thus, addresses 0x00 and 0x40 which differ only in bit 6, have the same tag, set index, and bank index and belong to the same sector while the eight words of one 64 B line are held in the same set and subsector position of the eight banks. Other features of a sector cache remain the same, except that access and replacement select a bank in addition to the set and subsector. The multiplexer and comparator circuits are not shown for brevity.

Each SRAM bank of SecBCache is direct-mapped: it holds one sector per set, so a set holds eight sectors, one in each bank. The baseline cache and SecBCache therefore have the same capacity (32 KB) and the same number of tags (512). They differ in where data can go: a baseline line can be placed in any of the eight ways of its set, while a SecBCache word can only go to the bank given by its bank index (bits 3 to 5), and replacing a sector removes all of its subsectors.

3.2 Analytical Bank-Conflict Model

Before presenting the model used to estimate bank conflicts and the delay of SecBCache, we distinguish two terms used in this work: *design* and *model*. In the previous section, the *design* of SecBCache and the data layout were discussed, which is the process being simulated in gem5: a *read* or *write* request issued by the CPU for a word

is searched inside SecBCache according to the design. The *hit* and *miss* functionality of the SecBCache is similar to that of a current sector cache implementation in gem5.

Algorithm 1 Bank-Conflict Model

```

1: ReqList  $\leftarrow$  request
2: if ReqList.size() == WindowSize then
3:   ReqList.UniqueSort() // Sort ReqList and remove duplicates
4:   for all request  $x$  in ReqList do
5:     bank  $\leftarrow$  GetBankNumber( $x$ )
6:     BankPlacement[bank]++
7:   end for
8:   MaxBank  $\leftarrow$  max(BankPlacement) // Identify bottleneck bank
9:   //  $\sum \text{latency}(\text{MaxBank}) = \sum \text{hit\_latency} + \sum \text{miss\_latency}$ 
10:  // 1. Conservative bound (no overlap)
11:  BankLatency_conservative  $\leftarrow \sum \text{latency}(\text{MaxBank})$ 
12:  // 2. Ideal Bound (full overlap of misses)
13:  BankLatency_ideal  $\leftarrow \sum \text{hit\_latency}(\text{MaxBank}) + \max(\text{miss\_latency}(\text{MaxBank}))$ 
14:  // max(miss_latency) = 0 if MaxBank has no misses
15:  ReqList.Clear()
16: end if

```

However, gem5 does not allow multiple memory requests from the CPU to reach the memory system within a single cycle. A vector unit with banked memory should be able to concurrently access the banks with non-conflicting memory addresses mapped to them. To resolve this, a *model* was developed that demonstrates the possible benefit of a banked cache operating at word granularity. In subsequent sections, we will discuss how the parameters chosen for the simulation and the data gathered as output address this hybrid method of architectural design and analytical processing, to ensure no unfair benefit of the proposed SecBCache is presented.

Algorithm 1 shows a simplified pseudo-code of the model. This analytical approach is required because cycle-level simulators such as gem5 do not support concurrent word-granularity accesses to the L1 within a single cycle. Each memory request arriving at SecBCache is stored until the number of requests reaches the window size (*WindowSize* in Algorithm 1), which is the theoretical maximum number of requests that the cache can process simultaneously if there were no bank conflict. Windows are formed from consecutive requests in arrival order and do not overlap.

Vector extensions in modern ISAs (*e.g.*, Intel AVX-512 *vgather*, ARM SVE2 LD1D) are capable of generating up to 8 unique addresses per instruction for indirect memory addressing. This address generation capability, coupled with large load/store queues (LSQs) and reorder buffers (ROBs), enables superscalar, out-of-order processors to issue multiple concurrent memory requests within a narrow execution window. This does not require the crossbar to expose eight simultaneous CPU-facing ports; a conventional 2–3 port L1 interface (*e.g.*, two loads, one store per cycle) suffices, since the

eight addresses generated by a single gather/scatter instruction arrive at SecBCache clustered within a narrow temporal window rather than in the same cycle, and the crossbar routes each to its bank as it arrives.

Prior studies of multi-banked data caches have shown that a relatively small number of processor-to-cache ports can exploit most of the performance potential of a multi-banked organization [64, 65]. Accordingly, we evaluate SecBCache using eight-request windows corresponding to the eight 8 B elements of the 64 B vector width, rather than assuming 8 requests are issued in the same cycle.

Afterwards, only unique addresses are chosen for the next step because a typical CPU will be able to forward (for loads) or handle consistency (for stores). Once we have the unique addresses for that window, the bank index of each address is calculated to build the histogram of bank occupancy for that window. In a banked memory, the limiting factor for performance is the bank with the maximum address conflicts, $\max(\text{BankPlacement})$. The latency is computed separately for hits and misses. Each hit, read or write, costs the hit latency of the cache: five cycles for SecBCache and three for the baseline cache. Each miss costs its latency in gem5, from the time it reaches the cache until its data returns.

Within each window, the SRAM bank with the maximum number of conflicts is responsible for the performance bottleneck. The Conservative bound (*BankLatency.conservative*) assumes a worst-case blocking scenario where strict serialization of all requests in that bank is enforced, meaning that no coalescing or memory-level parallelism is available to hide stalls. Therefore, it is calculated as the sum of the latencies for all hits and misses within that bank with maximum conflicts. The Ideal bound (*BankLatency.ideal*) represents a system in which a CPU can sustain multiple in-flight memory requests (*e.g.*, using non-blocking miss status holding registers (MSHRs) and a multi-ported LSQ) and the longest-stalling miss completely overlaps and hides the latencies of other misses within the most conflicted bank. If the most conflicted bank has no misses then the maximum miss latency is zero, and the Ideal bound equals the Conservative bound.

Both bounds evaluate static windows in isolation, so they do not dynamically model the cycle-by-cycle queuing and structural backpressure that occurs between the CPU execution pipeline and the cache input port (such as reorder buffer stalls). These effects interact with bank conflicts: sustained conflicts at one bank lengthen load latency, which can fill the LSQ and block the ROB. The Conservative bound charges this serialization in full, whereas the Ideal bound assumes the longest miss hides all others; the two bounds are meant to bracket the effect of out-of-order overlap without modeling it explicitly.

Therefore, our delay evaluation isolates the memory subsystem, relying only on hit and miss latencies: CPU-to-L1 crossbar traversal is not included. To ensure a fair comparison, the baseline is evaluated with the same memory-centric metric, the total hit and miss latency observed at the cache, but without bank-level overlap, since a conventional cache has no banked access path.

4 Experimental Setup

This section describes how SecBCache is evaluated. Section 4.1 explains how gather/scatter instructions are generated in gem5 and how the indirectly accessed array is routed to reach SecBCache. Section 4.2 shows where SecBCache sits in the simulated cache hierarchy. Section 4.3 lists the system, baseline, and SecBCache parameters and explains the choices that favor the baseline. Section 4.4 introduces the BFS and SpGEMM kernels and the 16 SuiteSparse datasets used in the evaluation.

4.1 Gather/Scatter in gem5

The Scalable Vector Extension (SVE) of the ARM ISA is supported by gem5. The gather/scatter assembly instructions are decoded to generate individual micro-ops, which are processed by the CPU as individual load/store instructions.

```
double (*restrict A) = (double*) 0x800000000;
for (int i = 0; i < N; i++){
    C[i] = A[B[i]];
}
```

Fig. 2 C code with pointer to fixed memory

Figure 2 shows the core of the gather operation. To isolate indirect accesses from instruction fetches and regular data traffic, the array `A[]` is placed at a fixed address (`0x800000000`) in the source code. The simulated system has two non-overlapping physical memory ranges: one reserved for data accessed indirectly and one for everything else. Before the program starts running, the gem5 configuration script maps a virtual region beginning at `0x800000000` one-to-one onto the indirect-access range, calling gem5’s `map()` once per 4 KB page. Because the mapping is one-to-one, the address in the C code is also the physical address presented to the L1 caches.

The crossbar between the CPU and the L1 caches decodes the physical address of each request and routes it to the cache that owns that range: the indirect-access range goes to SecBCache, and all other addresses, including the index array `B[]`, go to the L1 D-cache. The baseline configuration uses the same memory map. Apart from this address-based routing, SecBCache operates as a standard, hardware-managed L1 cache in gem5. The `restrict` qualifier rules out compiler aliasing. This memory map is a simulation device for isolating indirect accesses, so the results characterize the memory subsystem on these access streams and not the performance of a complete system. Section 6 discusses how such a mapping could be realized in a real system.

Although prior studies using gem5 [5, 23, 29] introduced custom instructions (assembly and intrinsics) and specialized compile-time and decode-stage processing to present a programmer’s view of their designs, our methodology uses direct memory mapping to isolate and evaluate the underlying hardware behavior of indirect accesses.

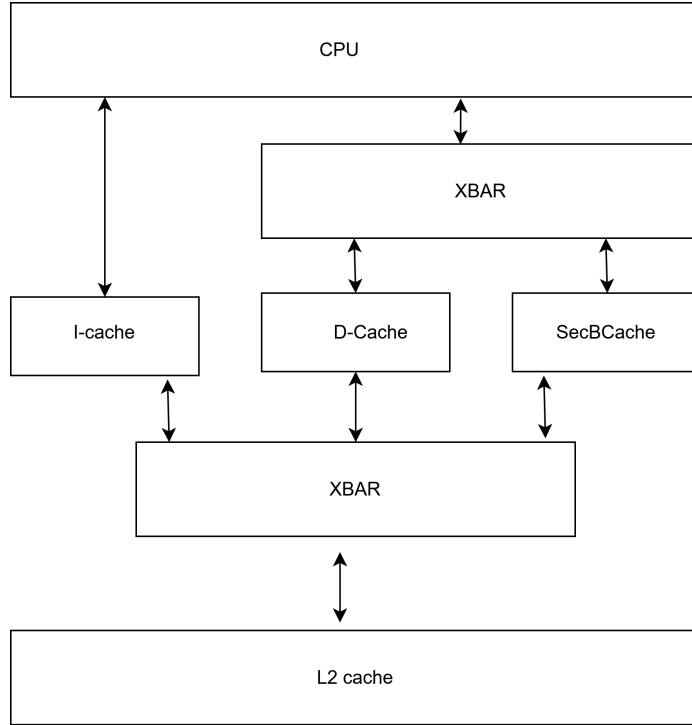


Fig. 3 CPU and cache configuration

4.2 CPU Configuration

Figure 3 shows the top-level view of the CPU configuration. SecBCache, a word-granularity banked sector cache, sits at the same level as the L1 I-cache and L1 D-cache. SecBCache and the L1 D-cache have separate address ranges. Only memory that is accessed indirectly is mapped to SecBCache; all other traffic including the data needed to compute the indirect addresses goes through the L1 D-cache. This lets the CPU access SecBCache in parallel with the two other L1 caches.

The crossbar connecting the CPU to the L1 D-cache and SecBCache routes each request to the right cache. This crossbar is required in gem5 because the CPU object exposes only two memory ports (*instruction* and *data*), so connecting more than one data-side cache requires a crossbar. It also reflects a realistic configuration, since a core-to-L1 interconnect with many ports would greatly increase area and power. The L2 cache is shared by all three L1 caches. All ports shown are bidirectional.

4.3 Baseline and SecBCache Configurations

Table 1 describes the system parameters to evaluate SecBCache performance. The *Vector Length* denotes the vector register width in *quadwords*. A single core was used in the experiments. In Figure 3, the crossbar (XBAR) connecting the `dcache_port` of the CPU to the L1 D-cache and SecBCache is 512 bits wide. The L1 D-cache, baseline

Table 1 System Configuration

Component	Parameter
Processor Core	ARM ISA (AArch64), vector length = 4 quadwords, out-of-order, frequency = 2 GHz
L1 D-cache	Size = 64 KB, associativity = 8, access latency = 3 cycles
L2 Cache	Size = 2 MB, associativity = 16, access latency = 10 cycles
Window Size	8 requests (matching SVE2 gather/scatter vector width)

cache, and L2 latencies were set from CACTI [38] access times, rounded up to whole cycles with margin. The window size was set to 8, matching the number of addresses generated per SVE2 gather/scatter instruction.

Recall from Section 4.1, only the source array of a gather/scatter computation is mapped to the SecBCache; all other data, including the index array, communicate with memory via the L1 D-cache.

Table 2 Baseline and SecBCache Configuration

Component	Parameter
Baseline Cache	Size = 32 KB, associativity = 8, latency = 2 cycles, line size = 64 B
SecBCache	Size = 32 KB, 8 banks, associativity = 1 per bank (direct-mapped), access latency = 4 cycles, subsector size = 8 B

In both configurations the core has three L1 caches: an instruction cache, the 64 KB L1 D-cache of Table 1 for all regular data, and a 32 KB cache for the indirect-access range of Section 4.1. In the SecBCache configuration, this third cache is SecBCache whereas in the baseline configuration, it is a conventional cache of the same size, which we call the *baseline cache*. All other components, including the L1 D-cache and the L2, are identical in the two configurations.

Table 2 lists the two caches. They differ in associativity, latency, line or subsector size. The associativity of SecBCache applies to each bank.

Since CACTI cannot model the demultiplexing and routing overhead of a banked sector cache, a 4-cycle latency (double the baseline) was explicitly chosen for SecBCache, which accounts for (conservatively) both the bank demultiplexing and subsector decoding. In addition, the baseline configuration is modeled with an aggressive 8-way associativity, ensuring fewer conflict misses. These choices, although unfavorable to SecBCache, ensure that the delay reductions reported in Section 5 are not inflated by a weak baseline.

4.4 Benchmark Applications

SecBCache and the bank-conflict model were evaluated using two application kernels on 16 sparse datasets from various domains in the SuiteSparse Matrix Collection.

1. *BFS*: Breadth-first search is a common algorithm for graph analytics applications. Graph topologies are inherently sparse, and compression algorithms such as Compressed Sparse Row (CSR) are often utilized to store them. While compression reduces static memory footprint by eliminating the zero elements, it introduces indirect memory addressing (*e.g.*, $A[B[i]]$). This transforms the frontier expansion of BFS into a gather-update-scatter operation. We used ARM SVE2 intrinsics in the source code to ensure vectorization.

Although there are no strict rules regarding the starting point (the root) of the traversal, our approach explicitly takes the vertex with the largest number of outgoing edges as the root. This stresses the memory hierarchy and ensures that the design and timing model of SecBCache are tested under high contention.

Table 3 BFS Datasets

Name	Domain	# of rows/columns	# of non-zeros	Density (%)
ca-CondMat	Condensed Matter Physics	23,133	93,497	0.0175
email-Enron	Email Network	36,692	183,831	0.0137
wiki-Vote	Wiki Vote Network	8,297	103,689	0.1506
TSOPF_RS_b39_c7	Power Network	14,098	252,446	0.1270
ca-GrQc	Collaboration Network	5,242	14,496	0.0528
cit-HepTh	Citation Network	27,770	352,807	0.0457
soc-sign-bitcoin-otc	Trade Network	5,881	35,592	0.1029
p2p-Gnutella25	P2P Network	22,687	54,705	0.0106

2. *SpGEMM*: Sparse general matrix-matrix multiplication is a fundamental kernel utilized in scientific simulations, high-performance computing (HPC), and machine learning applications. Similar to BFS, SpGEMM can stress the memory hierarchy through an unpredictable access pattern. To implement an outer-product formulation of SpGEMM ($C=A \times B$), matrix A was converted to Compressed Sparse Column (CSC) format and matrix B to Compressed Sparse Row (CSR) format, which contain only the non-zero elements. For each dataset, B is taken equal to A (*i.e.*, we evaluate $C = A^2$). The compression transforms the multiplication process into a sequence of vectorized gather (loading sparse elements), fused multiply-add, and scatter (storing accumulated partial products) operations.

Unlike *BFS*, *SpGEMM* does not require conditional updates. However, both rely on indirect addressing for compressed data, making them ideal candidates to evaluate the SecBCache architecture.

Table 3 and Table 4 list the datasets used for the BFS and SpGEMM kernels: name, domain, number of rows (all matrices are square), number of non-zeros, and density (non-zeros divided by rows $\times$ columns in percent).

Table 4 SpGEMM Datasets

Name	Domain	# of rows/columns	# of non-zeros	Density (%)
LeGresley_4908	Power Network	4,908	30,482	0.1265
rotor2	Structural Problem	791	10,685	1.7077
Trefethen_700	Combinatorial Problem	700	6,677	1.3627
filter2D	Model Reduction Problem	1,668	6,209	0.2232
t2dalE	Duplicate Model Reduction	4,257	4,257	0.0235
S80PL_n1	Eigenvalue/Model Reduction	4,028	9,927	0.0612
Pd	Counter Example Problem	8,081	13,036	0.0200
EX1	Combinatorial Problem	560	4,368	1.3929

5 Experimental Results

This section evaluates SecBCache against the 32 KB, 8-way, 64 B line baseline of Table 2 on the 16 datasets of Section 4.4. Section 5.1 measures how much of each fetched 64 B line the baseline cache actually uses and how much SRAM write volume and evicted data SecBCache saves. Section 5.2 examines how requests are distributed across the eight banks and reports the resulting Conservative and Ideal memory-access delay bounds. Section 5.3 estimates how many 64 B MSHRs each eight-request window needs, and Section 5.4 reports the area, leakage power, and dynamic energy. Line utilization, SRAM write volume, and evictions are gem5 statistics. Bank placement and delay come from the bank-conflict model of Section 3.2, applied to the request stream simulated in gem5; the delays describe the memory subsystem, not end-to-end execution time. Area and energy are CACTI-based estimates. The percentages in the text are computed from unrounded measurements, and bar labels are rounded to two decimals.

These terms are common, but in a conventional cache they all refer to the same 64 B line, whereas SecBCache fetches and fills 8 B subsectors and evicts whole sectors, so we define each term as it is used in this section:

- A *request* is one load or store of an 8 B word that the core sends to the L1; each element of a gather or scatter is one request.
- On a miss, the cache *fetches* the missing data from the L2: a 64 B line for the baseline cache and an 8 B subsector for SecBCache. The data returned is a *transfer* over the L1–L2 interconnect.
- Writing the fetched data into the L1 data array is a *fill* (8 B for SecBCache, 64 B for baseline).
- The baseline cache evicts one 64 B line, and SecBCache evicts one sector with all of its valid subsectors.
- *Line utilization* (Figures 4 and 5) is, for each period a 64 B line spends in the baseline cache, the number of distinct words the program accesses divided by eight, averaged over all such periods.

- *SRAM write volume* (Figures 6 and 7) is the number of bytes written into the data array by fills. Store hits are not included, and a word filled again after an eviction is counted again.

5.1 Line Utilization, SRAM Write Volume, and Evictions

This subsection measures how much of each fetched 64 B line the baseline cache uses, and how much SRAM write volume and evicted data a word-granularity sector cache saves as a result. All metrics in this subsection are gem5 statistics and not derived from the bank-conflict model.

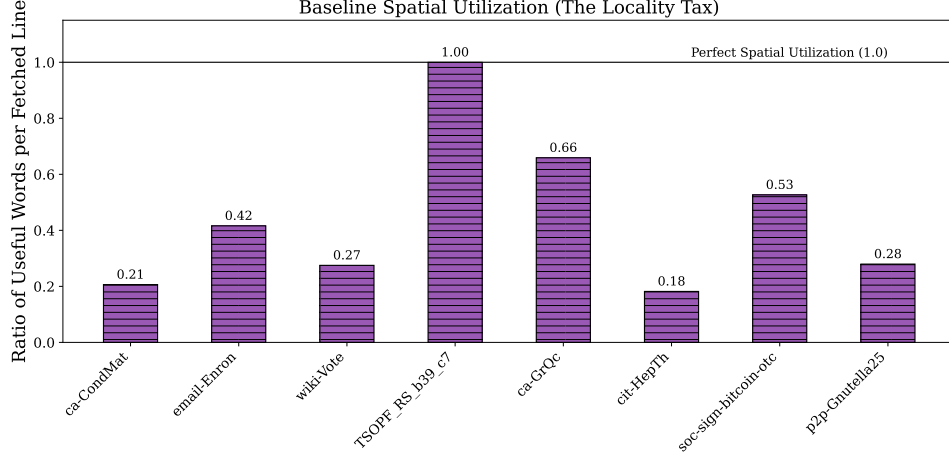


Fig. 4 Cache line utilization (BFS)

Figure 4 shows, for each BFS dataset, the fraction of the eight words in a fetched 64 B line that the baseline accesses before the line is evicted. Utilization ranges from 18.2% (cit-HepTh) to 99.9% (TSOPF_RS.b39.c7), with a median of 34.8%, and five of the eight datasets use fewer than half of the words they fetch. Only TSOPF_RS.b39.c7 (99.9%), ca-GrQc (66%), and soc-sign-bitcoin-otc (53%) use half or more of each line, as connected vertices in these graphs often reside within the same 64 B boundary.

Figure 5 shows the same metric for the SpGEMM kernel. Utilization for SpGEMM ranges from 12.5% (t2dal_e) to 57.7% (rotor2), with a median of 18.5%, approximately half the BFS median of 34.8%. t2dal_e uses exactly one word from every 64 B line it fetches. The gap comes from the datasets: in the BFS graphs, neighboring vertices often have nearby IDs, and BFS also scans the visited array in order, so more of each 64 B line is used, while the SpGEMM output entries accessed together usually do not share a line. In both kernels, most of each fetched line is never used, which is the redundant data movement SecBCache targets to alleviate.

Figure 6 compares the SRAM write volume of SecBCache with that of the baseline cache for the BFS kernel. Since SecBCache fills only the requested 8 B subsector instead of a whole 64 B line, it writes 52.0% (email-Enron) to 80.6% (cit-HepTh)

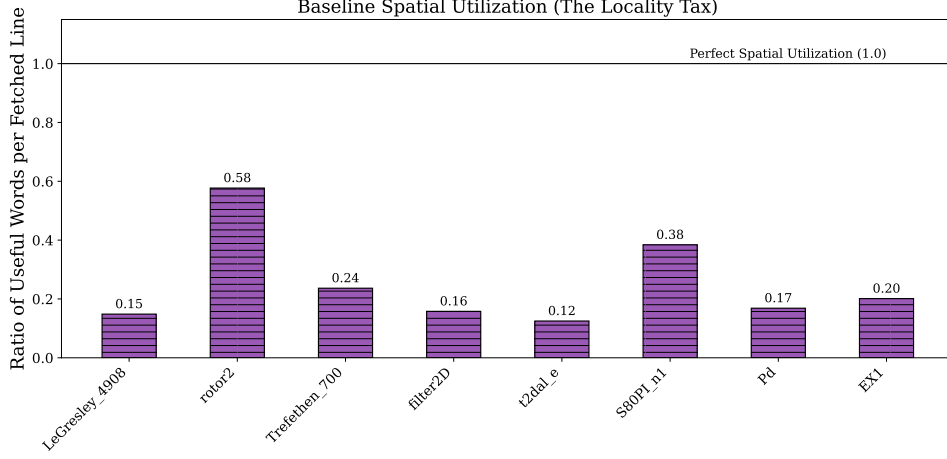


Fig. 5 Cache line utilization (SpGEMM)

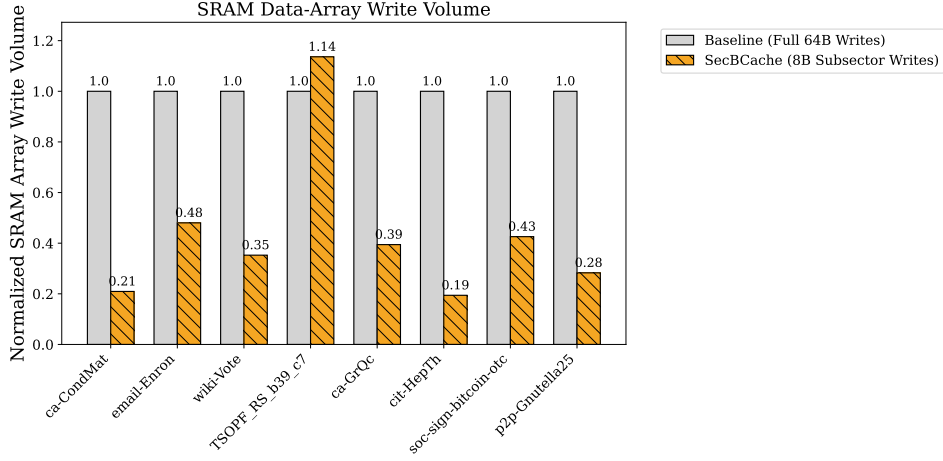


Fig. 6 Normalized SRAM write volume (BFS)

less data on seven of the eight datasets, with a median reduction of 63%. The highest reductions occur among the least-utilized datasets shown in Figure 4 (cit-HepTh, ca-CondMat, and p2p-Gnutella25, which use at most 28% of each line), whereas TSOPF_RS_b39_c7, whose baseline uses 99.9% of each line, writes 13.6% more data than the baseline.

This metric counts data written into the data array, not data transferred from the L2. In our gem5 setup, SecBCache uses 8 B MSHRs, so each fetch returns an 8 B transfer, and the miss latencies in the delay results reflect 8 B transfers. With 64 B MSHRs (Section 5.3), each fetch would return a 64 B transfer, of which SecBCache would still fill only the requested words.

Line utilization and SRAM write volume count different events, so they are not interchangeable. Utilization counts the distinct words used while a line is in the baseline cache. Write volume counts every fill, so a word that is evicted and later filled again is counted again.

In the baseline cache, one 64 B fetch brings in all neighboring words at once. SecBCache fills each word separately when it is requested, and replacing a sector removes all of its subsectors, so a word used again after a replacement must be filled again. This is why TSOPF_RS.b39.c7, whose baseline cache uses nearly every word of each line, writes 13.6% more data with SecBCache than with the baseline.

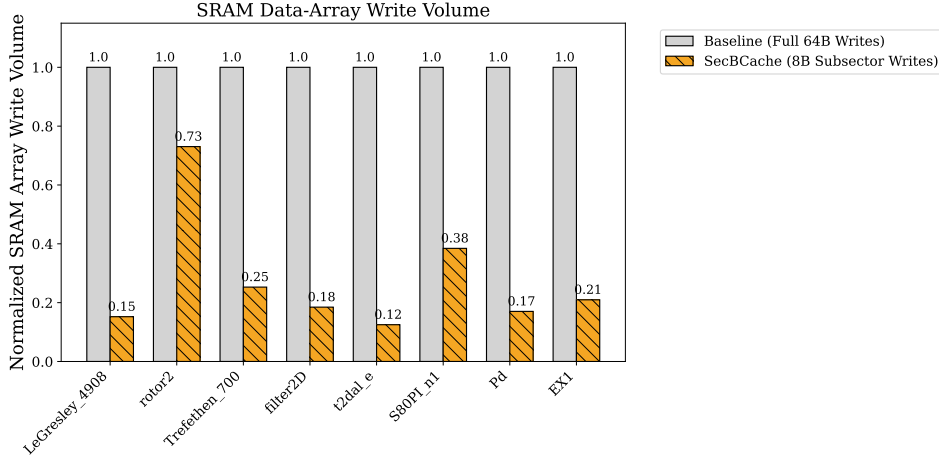


Fig. 7 Normalized SRAM write volume (SpGEMM)

Figure 7 shows the same comparison for SpGEMM. SecBCache writes 27.0% (rotor2) to 87.5% (t2dal_e) less data, with a median reduction of 80%. The ordering follows utilization (Figure 5): t2dal_e writes exactly one-eighth of the baseline volume, while rotor2, the only dataset that uses more than half of each line, saves the least.

Figure 8 classifies every L1 replacement in the BFS datasets, that is, every miss that allocates space, by the amount of valid data it evicts. The baseline always evicts a full 64 B line. SecBCache evicts nothing in a Zero Eviction, where the missing word goes into an empty subsector of a sector that is already resident. SecBCache evicts 1 to 7 valid subsectors in a Partial Eviction, and all eight in a Full Eviction. Full Evictions are rare, from 1.3% (cit-HepTh) to 11.2% (ca-GrQc) of replacements. The Zero Eviction share rises with utilization, from 28.9% for cit-HepTh, the least-utilized dataset, to 80.7% for TSOPF_RS.b39.c7, where successive misses fill empty subsectors of a resident sector.

Figure 9 shows the same breakdown for SpGEMM. Zero Evictions are rare due to lower spatial locality, at most 3.3% for LeGresley_4908, t2dal_e, S80PI_n1, and Pd. Maximum Zero Eviction is demonstrated by rotor2 (41.5%), which is the dataset with the highest utilization. In six of the eight datasets, most replacements (from

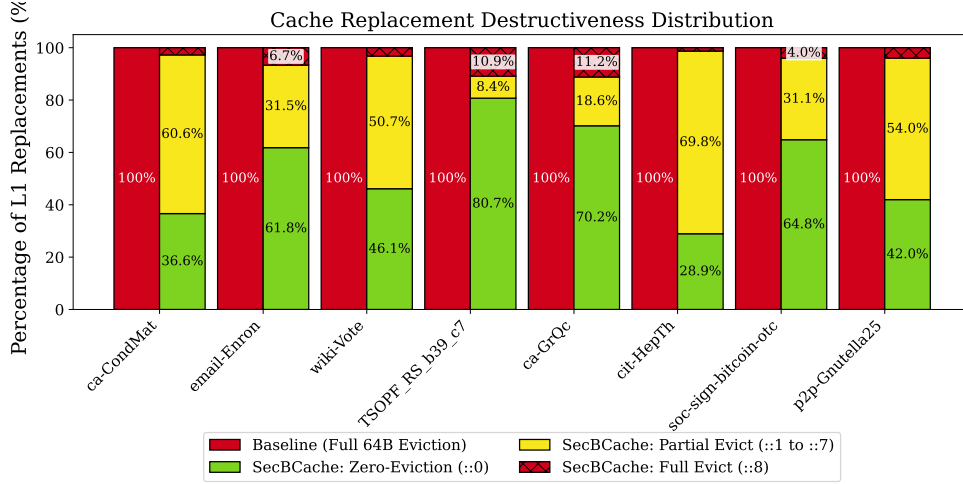


Fig. 8 Data evicted per replacement (BFS)

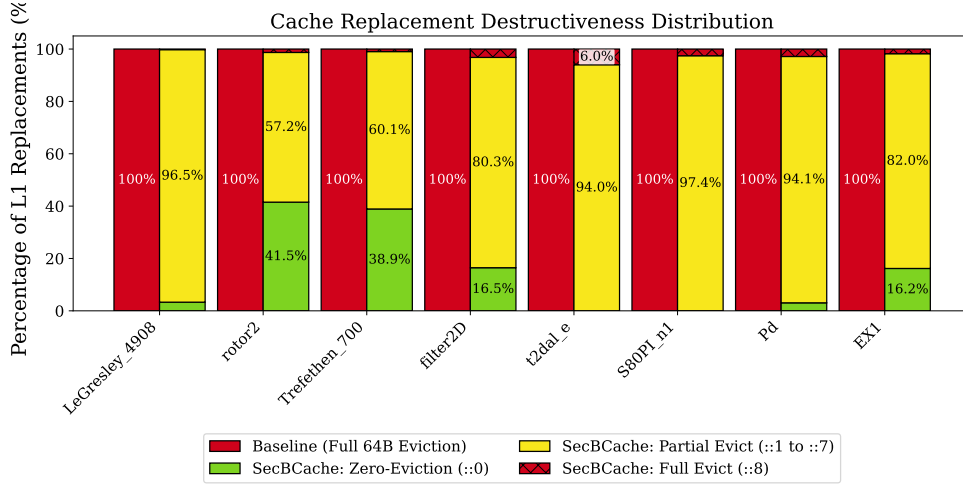


Fig. 9 Data evicted per replacement (SpGEMM)

66.1% to 97.4%) evict exactly one subsector (the Partial Eviction segment in Figure 9 also counts replacements of 2 to 7 subsectors), and Full Evictions never exceed 6.0% (t2dal_e).

5.2 Bank Conflicts and Memory-Access Delay

This subsection first shows how the requests of each eight-request window are distributed across the eight banks and then how this distribution determines the Conservative and Ideal delay bounds.

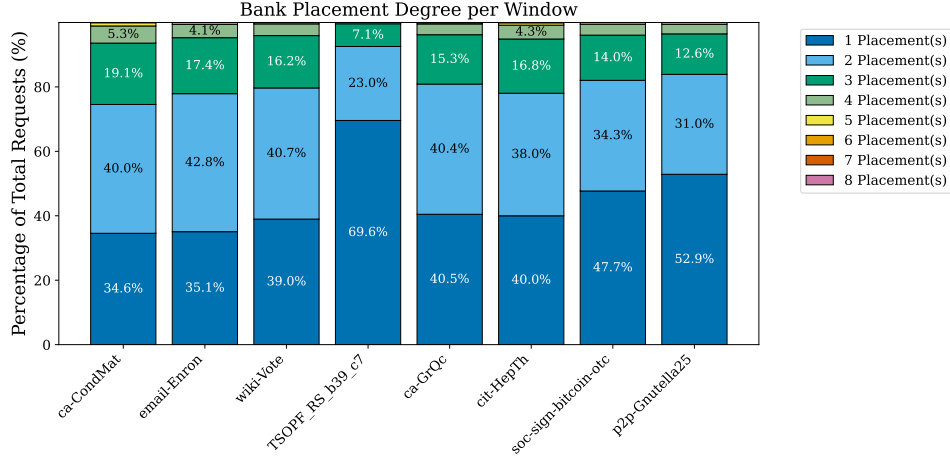


Fig. 10 Bank Placement Distribution (BFS)

Figure 10 shows, for the BFS datasets, how requests are distributed by placement degree: a request has degree k when k requests of its window map to the same bank and are serialized there, so degree 1 means no bank conflict. TSOPF_RS.b39.c7 serves 69.6% of its requests without conflict and 23.0% at degree 2, consistent with its high spatial locality (Figure 4). ca-CondMat and email-Enron have the lowest conflict-free shares (34.6% and 35.1%). Across all BFS datasets, degrees 1 and 2 cover 74.5% to 92.5% of requests, and degree 5 or higher at most 1.1%, demonstrating the potential for word-interleaved banking for sparse data structures.

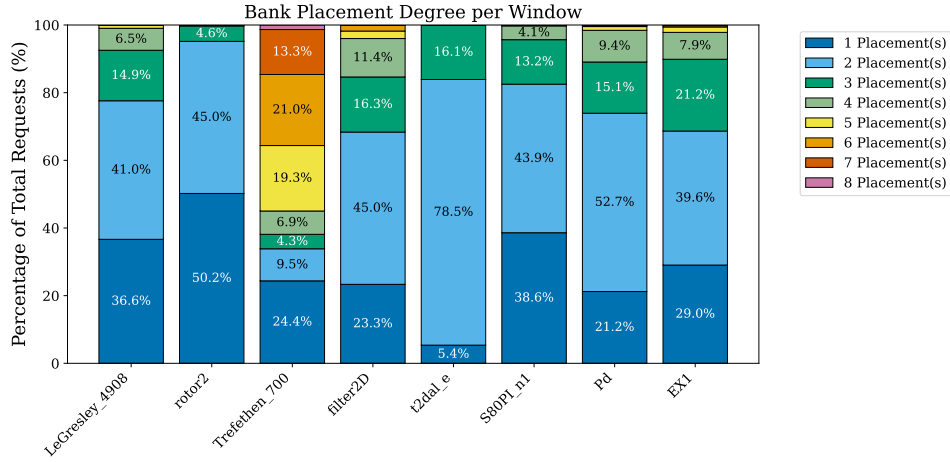


Fig. 11 Bank Placement Distribution (SpGEMM)

Figure 11 shows the distribution for SpGEMM. Seven of the eight datasets again concentrate on degrees 1 and 2 (68.4% for filter2D to 95.2% for rotor2), although the median conflict-free share is lower than for BFS (26.7% versus 40.2%). t2dal_e has only 5.4% conflict-free requests, but 78.5% are at degree 2, and none exceed degree 3. Trefethen.700 is the outlier: only 33.9% of its requests are at degree 1 or 2, and 55.0% are at degree 5 or higher. This is a consequence of the power-of-two stride for Trefethen.700 [66], which results in high bank conflict, a known weakness of interleaved memory discussed in Section 2.3.

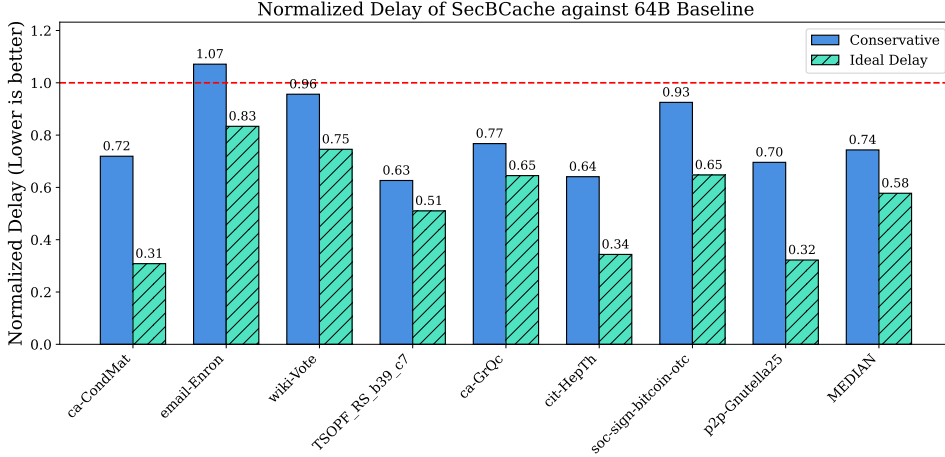


Fig. 12 Modeled memory-access delay under the Conservative and Ideal bounds, normalized to the baseline (BFS)

Figure 12 shows the memory-access delay of SecBCache under the Conservative and Ideal bounds, normalized to the baseline (values below 1.0 favor SecBCache). Under the Ideal bound, the delay drops for every dataset by 16.6% (email-Enron) to 69.2% (ca-CondMat), with a median of 42%. Under the Conservative bound, which charges the full bank serialization penalty, delay drops for seven of the eight datasets by 4.4% (wiki-Vote) to 37.4% (TSOPF_RS_b39_c7) with a median of 26%, and only email-Enron is slower (7.1%).

The results reflect a trade-off of word-granularity fills. SecBCache serves the requests of a window in parallel across its banks, but because it fills only the requested word, it gives up the reuse the baseline gets when later requests hit other words of a 64 B line.

Bank conflicts alone do not explain the differences in delay: ca-CondMat and email-Enron have nearly identical bank placement (Figure 10), yet their delays differ by 35 percentage points under the Conservative bound and 52 under the Ideal bound. email-Enron uses twice as many words per fetched line as ca-CondMat (Figure 4), so the baseline reuses more of each line and SecBCache gains little (17% under the Ideal bound, and a 7% regression under the Conservative bound). ca-CondMat uses only 20.6% of each line, so SecBCache is able to outperform the baseline configuration.

TSOPF_RS_b39_c7 is the opposite case: it uses 99.9% of each line, yet it gains the most under the Conservative bound because 69.6% of its requests are conflict-free.

Another major factor affecting the memory-access delay reduction is the working-set size. The three smallest graphs, ca-GrQc, soc-sign-bitcoin-otc, and wiki-Vote (Table 3) have working sets closest to the cache size (both the baseline and the SecB-Cache), so the baseline reuses more of its resident lines and they show the smallest gains under the Conservative bound among the datasets that improve (23.3%, 7.5%, and 4.4%). Finally, the Conservative bound essentially represents a blocking architecture: aggregated delay due to serialization at the bank with maximum structural hazard. This is why the Conservative bound gives the smaller delay reduction: realistic out-of-order memory-level parallelism would overlap many of these bank stalls.

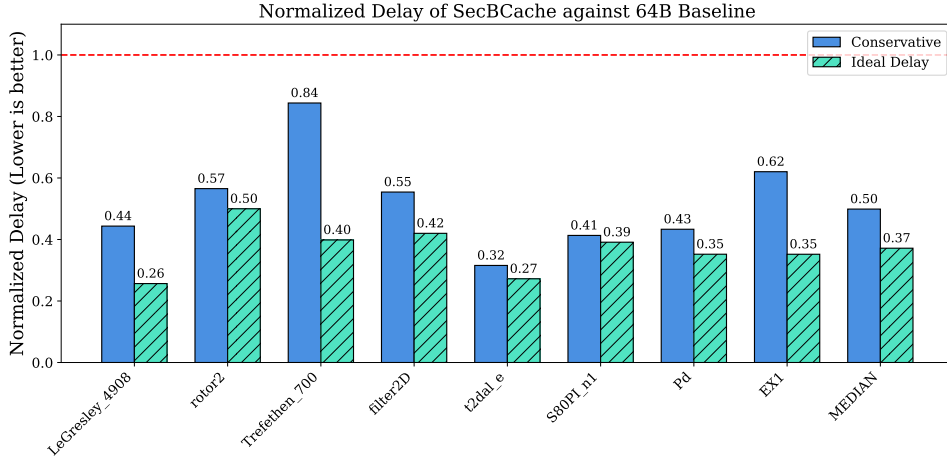


Fig. 13 Modeled memory-access delay under the Conservative and Ideal bounds, normalized to the baseline (SpGEMM)

Figure 13 shows the same bounds for SpGEMM. Delay drops for every dataset under both models: by 15.6% (Trefethen_700) to 68.4% (t2dal_e) under the Conservative bound with a median of 50%, and by 50.0% (rotor2) to 74.3% (LeGresley_4908) under the Ideal bound with a median of 63%.

The median reduction under the Conservative bound is 24 percentage points larger than for BFS, consistent with the lower line utilization of SpGEMM (median 18.5% versus 34.8%): the baseline gets little reuse from its 64 B lines. t2dal_e uses exactly one word from every fetched line and gains the most under the Conservative bound. Trefethen_700 gains the least under the Conservative bound because of its severe bank conflicts (Figure 11), but its delay under the Ideal bound is 60% lower, which shows that most of its penalty comes from serialization at one bank. rotor2 gains less than t2dal_e despite fewer bank conflicts because it has the highest baseline utilization (57.7%), so SecBCache gives up the most reuse.

5.3 Miss Status Holding Registers (MSHRs)

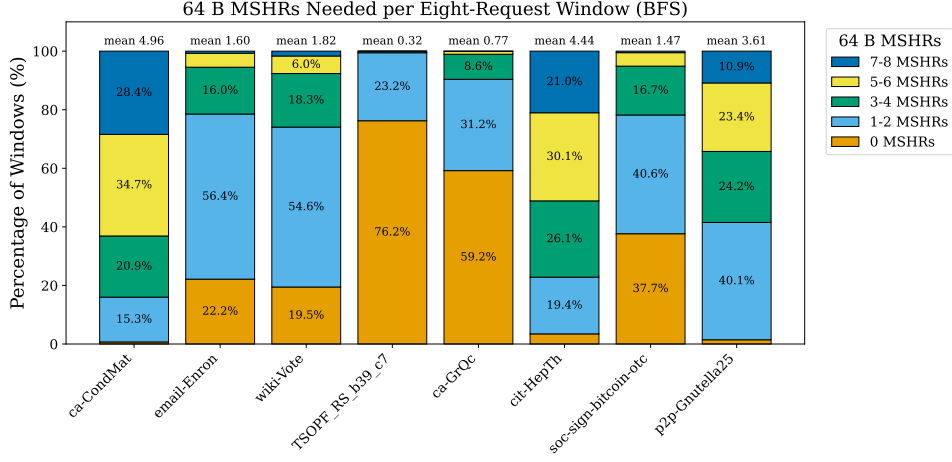


Fig. 14 Estimated number of 64 B MSHRs needed per eight-request window (BFS)

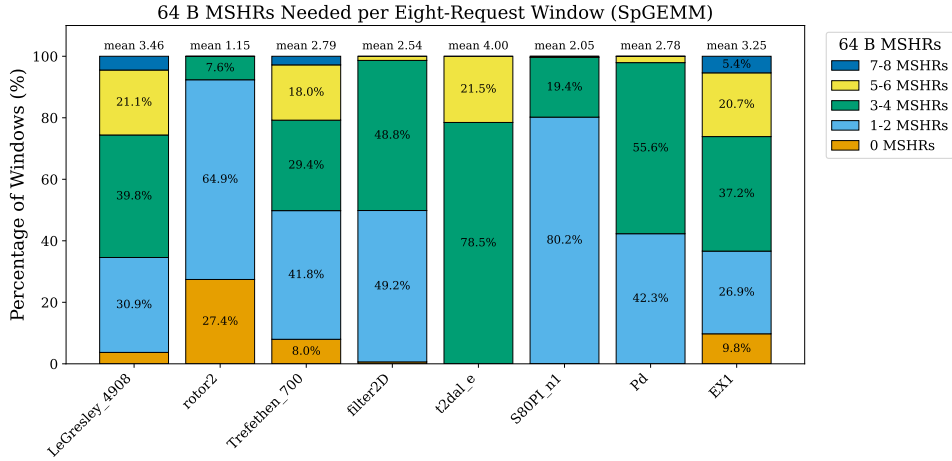


Fig. 15 Estimated number of 64 B MSHRs needed per eight-request window (SpGEMM)

Since gem5 enforces a uniform width for cache lines and Miss Status Holding Registers (MSHRs), simulating SecBCache required allocating artificially high 128 ‘8 B’ MSHRs compared to the baseline’s 32 64 B MSHRs. This is a simulator workaround rather than a design parameter.

With 64 B MSHRs, the misses of an eight-request window need one MSHR per distinct 64 B line they touch, and Figures 14 and 15 show this count for every window. By construction, a window never needs more than eight. On average, a window needs between 0.32 (TSOPF_RS_b39_c7, where most windows need none) and 4.96 (ca-CondMat) MSHRs for BFS, and between 1.15 (rotor2) and 4.00 (t2dal_e) for SpGEMM. Datasets with low line utilization need the most, because their misses rarely share a line: ca-CondMat, cit-HepTh, and p2p-Gnutella25 need five to eight MSHRs in 63.1%, 51.1%, and 34.2% of their windows. In contrast, TSOPF_RS_b39_c7 needs none in 76.2% of its windows. Even if two windows overlap, we estimate that sixteen 64 B MSHRs (half of the baseline’s 32) would suffice.

5.4 Area and Power

In this subsection, we evaluate the area, the overhead of leakage power, and the dynamic energy consumption of the SecBCache. All hardware parameters were modeled using CACTI. As established in the Methodology section, our evaluation was conducted by isolating the memory subsystem, and this power analysis strictly evaluates the cache hierarchy without incorporating CPU stall-induced leakage.

5.4.1 Area and Leakage Power

As described in Section 3.1, SecBCache requires no additional computational hardware in the CPU. Its additions are a bank-routing network and per-subsector state: one tag per sector, with a valid and a dirty bit for each of its eight 8 B subsectors. Including this metadata and the H-tree wiring between the eight banks, SecBCache occupies 0.14 mm², at most 2.46% of the combined area of the L1 I-cache, L1 D-cache, and L2, and adds 2.12% to their leakage power. These estimates do not include the CPU-to-L1 crossbar or the MSHRs. The metadata grows linearly with the number of subsectors, and the routing wires grow with the number of banks, which stays small in an L1 cache (eight in this work).

5.4.2 Dynamic Energy

Figures 16 and 17 compare the L1 dynamic energy of SecBCache with that of the baseline cache of Table 2, normalized to the baseline based on CACTI and gem5 data. The estimate covers only the two L1 caches being compared, not the L2, DRAM, or the CPU-to-L1 crossbar. SecBCache reduces L1 dynamic energy for every dataset. For BFS, the median reduction is 63%, and for SpGEMM it is 75%.

The largest reductions occur where the baseline uses the fewest words of each fetched line: cit-HepTh (18% utilization) and ca-CondMat (21%) save 83% and 82%, and t2dal_e, which uses exactly one word of every line (Figure 5), saves the most in SpGEMM. The smallest reductions occur where the baseline already uses much of each line: TSOPF_RS_b39_c7 (99.9% utilization, Figure 4) and rotor2 (57.7%, Figure 5). These two datasets also have the smallest write-volume reductions (TSOPF_RS_b39_c7 writes 14% more data and rotor2 27% less, Figures 6 and 7). Overall, the energy reductions track the SRAM write-volume reductions (medians of 63% and 80%): the datasets that write the fewest bytes into the data array also spend the least L1 energy.

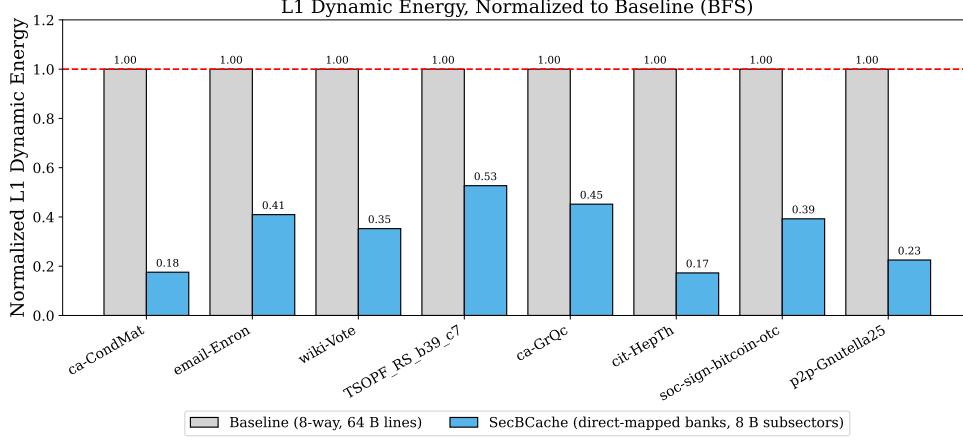


Fig. 16 L1 dynamic energy, normalized to the baseline (BFS)

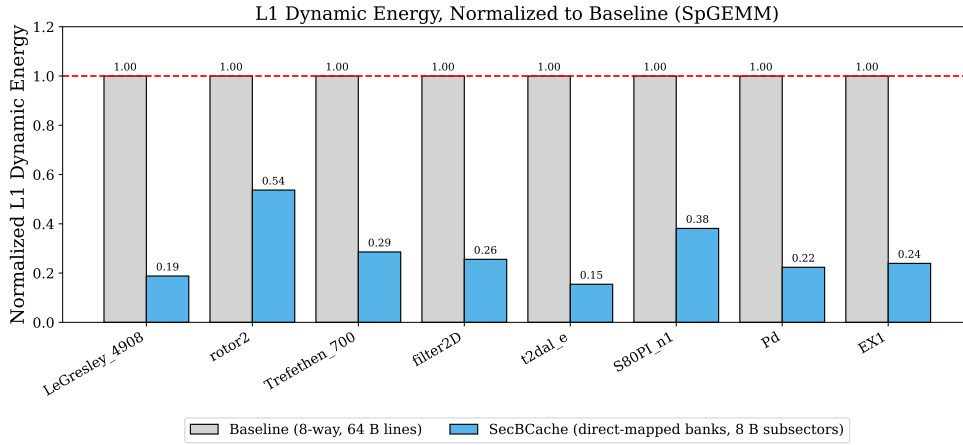


Fig. 17 L1 dynamic energy, normalized to the baseline (SpGEMM)

6 System Integration

In a practical implementation, software integration could use a lightweight library call (*e.g.*, `secmalloc()`). Developers would mark the data structures used for indirect lookups ($A[B[i]]$) as SecBCache-eligible, while subsector allocation, replacement, and eviction remain managed by hardware. The OS allocator tags these pages using page-level memory-type attributes—similar to x86 PAT or MTRR mechanisms—allowing the load/store unit to steer accesses automatically via TLB lookups.

This is consistent with recent systems research on user-level allocators that require no OS or ISA changes [67, 68] and on allocators for specialized memory technologies such as processing-in-memory [69], as well as compiler techniques that automatically identify irregular memory-access regions in a program [70].

In a multicore system, SecBCache would need to take part in the same coherence protocol as the L1 D-cache. Because valid and dirty state are already tracked per subsector, coherence state could also be kept per 8 B word, although coherence messages (*e.g.*, invalidations) would then need to address individual subsectors.

Per-word state could reduce false sharing between different words of the same 64 B line, but only if the shared L2 or directory also tracked sharers per word, at the cost of more directory state and coherence messages. We leave such a protocol to future work.

7 Conclusion

This paper presented SecBCache, a word-granularity sector cache organized as word-interleaved banks that targets the data-movement and energy costs of indirect memory access in sparse workloads. Conventional L1 caches fetch and evict data at 64 B granularity, an assumption well suited to dense contiguous access but a poor fit for the gather/scatter patterns common in sparse machine-learning, graph, and scientific-computing kernels, where our results show only a fraction of each fetched cache line is ever used. By sharing a single tag across eight word-sized subsectors, SecBCache retains hardware-managed residency and avoids the metadata overhead of a fully word-addressable cache or scratchpad, while its banked, interleaved layout lets the elements of a gather or scatter be served by different banks.

Because SecBCache works with standard vector gather/scatter instructions, it needs no new ISA extensions, standalone accelerators, or major code changes; only the data accessed indirectly has to be marked as SecBCache-eligible (Section 6). This lowers the software adoption barrier that has limited prior sparse acceleration proposals. By combining word-interleaved subsectors with hardware-managed residency, SecBCache reduces the L1 data-array writes and SRAM energy spent on words that a fixed 64 B line fetches but the program never uses.

Across 16 SuiteSparse datasets with BFS and SpGEMM kernels, gem5 measurements show that SecBCache reduces the SRAM data-array write volume by a median of 63% and 80%, and CACTI-based estimates show lower L1 dynamic energy on every dataset, with an area of at most 2.46% of the evaluated cache hierarchy. The analytical bank-conflict model estimates median memory-access delay reductions of 26% for BFS and 50% for SpGEMM under the Conservative bound; these are modeled memory-subsystem delays, not end-to-end speedups.

SecBCache reduces unnecessary L1 data-array writes for irregular accesses and shows potential for reducing memory-access delay under the evaluated analytical model. The next steps are to validate these delay estimates with a cycle-level model of concurrent bank access and to evaluate SecBCache in a full multicore system.

Acknowledgements. The authors acknowledge Dr. Tarun Laxman Prabhu from Los Alamos National Lab for helpful discussions and feedback.

Declarations

Funding

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Contract No. DE-AC02-05CH11231 and ASCR Competitive Portfolios for Computer Science.

Competing interests

The authors have no competing interests to declare that are relevant to the content of this article.

Data availability

The sparse matrices used in this study are publicly available through the SuiteSparse Matrix Collection. The simulation results and data derived to support the findings of this study are available from the corresponding author upon a reasonable request.

Code availability

The custom code used during the current study is available from the corresponding author upon reasonable request.

Ethics approval

Not applicable. This study does not involve human participants, human data, or animals.

Author Contributions

Nazmus Sakib: Conceptualization, methodology, software, investigation, data analysis, and writing the first draft. John Shalf: Conceptualization, methodology, and supervision. Abdel-Hameed A. Badawy: Supervision, conceptualization, methodology, writing review, and extensive editing. All authors reviewed and approved the final manuscript.

References

- [1] McKee, S.: Reflections on the memory wall. (2004). <https://doi.org/10.1145/977091.977115>
- [2] Rodrigues, A., Gokhale, M., Voskuilen, G.: Towards a scatter-gather architecture: hardware and software issues. In: Proceedings of the International Symposium on Memory Systems. MEMSYS '19, pp. 261–271. Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3357526.3357571>
- [3] Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P., Mutlu, O.: Google workloads for consumer devices: Mitigating data movement bottlenecks. SIGPLAN Not. **53**(2), 316–331 (2018) <https://doi.org/10.1145/3296957.3173177>
- [4] Mutlu, O., Olgun, A., Yüksel, İ.E.: Memory-centric computing: Solving computing’s memory problem. In: 2025 IEEE International Memory Workshop (IMW), pp. 1–4 (2025). <https://doi.org/10.1109/IMW61990.2025.11026935>
- [5] Khadem, A., Kamalakkannan, K., Zhu, Z., Poptani, A., Gu, Y., Dominguez-Trujillo, J.B., Talati, N., Fujiki, D., Mahlke, S., Shipman, G., Das, R.: DX100: Programmable data access accelerator for indirection. In: Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA). ISCA’25, pp. 1641–1658. Association for Computing Machinery, New York, NY, USA (2025). <https://doi.org/10.1145/3695053.3731015>
- [6] Hapla, V., Horák, D., Merta, M.: Use of direct solvers in tfeti massively parallel implementation. In: Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing. PARA’12, pp. 192–205. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-36803-5_14
- [7] Itoh, S., Ordejón, P., Martin, R.M.: Order-n tight-binding molecular dynamics on parallel computers. Computer Physics Communications **88**(2), 173–185 (1995) [https://doi.org/10.1016/0010-4655\(95\)00031-A](https://doi.org/10.1016/0010-4655(95)00031-A)
- [8] Shipman, G.M., Dominguez-Trujillo, J., Sheridan, K., Swaminarayan, S.: Assessing the Memory Wall in Complex Codes . In: 2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), pp. 30–35. IEEE Computer Society, Los Alamitos, CA, USA (2022). <https://doi.org/10.1109/MCHPC56545.2022.00009> . <https://doi.ieeecomputersociety.org/10.1109/MCHPC56545.2022.00009>
- [9] Gilbert, J.R., Reinhardt, S., Shah, V.B.: High-performance graph algorithms from parallel sparse matrices. In: Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing. PARA’06, pp. 260–269. Springer, Berlin, Heidelberg (2006)

- [10] Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12. IEEE Computer Society Press, Washington, DC, USA (2012)
- [11] Xing, W., Ghorbani, A.: Weighted PageRank Algorithm . In: Proceedings. Second Annual Conference on Communication Networks and Services Research, pp. 305–314. IEEE Computer Society, Los Alamitos, CA, USA (2004). <https://doi.org/10.1109/DNSR.2004.1344743> . <https://doi.ieeecomputersociety.org/10.1109/DNSR.2004.1344743>
- [12] Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh International Conference on World Wide Web 7. WWW7, pp. 107–117. Elsevier Science Publishers B. V., NLD (1998)
- [13] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: Eie: efficient inference engine on compressed deep neural network. In: Proceedings of the 43rd International Symposium on Computer Architecture (ISCA). ISCA'16, pp. 243–254 (2016). <https://doi.org/10.1109/ISCA.2016.30>
- [14] Anwar, S., Hwang, K., Sung, W.: Structured pruning of deep convolutional neural networks. J. Emerg. Technol. Comput. Syst. **13**(3) (2017) <https://doi.org/10.1145/3005348>
- [15] Park, J., Li, S., Wen, W., Tang, P.T.P., Li, H., Chen, Y., Dubey, P.: Faster CNNs with Direct Sparse Convolutions and Guided Pruning (2017). <https://arxiv.org/abs/1608.01409>
- [16] Vasireddy, P., Kavi, K., Weaver, A., Mehta, G.: Streaming sparse data on architectures with vector extensions using near data processing. In: Proceedings of the International Symposium on Memory Systems. MEMSYS '23. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3631882.3631898> . <https://doi.org/10.1145/3631882.3631898>
- [17] Gustavson, F.G.: Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM Trans. Math. Softw. **4**(3), 250–269 (1978) <https://doi.org/10.1145/355791.355796>
- [18] Eisenstat, S.C., Gursky, M.C., Schultz, M.H., Sherman, A.H.: Yale sparse matrix package I: The symmetric codes. International Journal for Numerical Methods in Engineering **18**(8), 1145–1151 (1982) <https://doi.org/10.1002/nme.1620180804>
- [19] Duff, I.S., Grimes, R.G., Lewis, J.G.: Sparse matrix test problems. ACM Trans. Math. Softw. **15**(1), 1–14 (1989) <https://doi.org/10.1145/62038.62043>
- [20] Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA (2003). <https://doi.org/>

- [21] Murphy, R., Rodrigues, A., Kogge, P., Underwood, K.: The implications of working set analysis on supercomputing memory hierarchy design. In: Proceedings of the 19th Annual International Conference on Supercomputing. ICS '05, pp. 332–340. Association for Computing Machinery, New York, NY, USA (2005). <https://doi.org/10.1145/1088149.1088193> . <https://doi.org/10.1145/1088149.1088193>
- [22] Rupnow, K., Rodrigues, A., Underwood, K., Compton, K.: Scientific applications vs. spec-fp: a comparison of program behavior. In: Proceedings of the 20th Annual International Conference on Supercomputing. ICS '06, pp. 66–74. Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1183401.1183413> . <https://doi.org/10.1145/1183401.1183413>
- [23] Srikanth, S., Jain, A., Conte, T.M., DeBenedictis, E.P., Cook, J.: Sortcache: Intelligent cache management for accelerating sparse data workloads. *ACM Trans. Archit. Code Optim.* **18**(4) (2021) <https://doi.org/10.1145/3473332>
- [24] Seager, K.O., Tiwari, A., Laurenzano, M.A., Peraza, J., Cicotti, P., Carrington, L.: Efficient hpc data motion via scratchpad memory. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 801–805 (2012). <https://doi.org/10.1109/SC.Companion.2012.111>
- [25] Sakib, N., Prabhu, T., Santhi, N., Shalf, J., Badawy, A.-H.A.: Comparison of vectorization capabilities of different compilers for x86 and arm cpus. In: 2024 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2024). IEEE
- [26] Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number 248966-042. <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf> (2019)
- [27] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: Proceedings of the Tenth International Symposium on Hardware/Software Code-sign. CODES '02, pp. 73–78. Association for Computing Machinery, New York, NY, USA (2002). <https://doi.org/10.1145/774789.774805>
- [28] Gschwind, M.: The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.* **35**(3), 233–262 (2007) <https://doi.org/10.1007/s10766-007-0035-4>
- [29] Singh, A., Dave, S., Zardoshti, P., Brotzman, R., Zhang, C., Guo, X., Shrivastava, A., Tan, G., Spear, M.: Spx64: A scratchpad memory for general-purpose microprocessors. *ACM Trans. Archit. Code Optim.* **18**(1) (2021) <https://doi.org/10.1145/3436730>

- [30] Rothman, J.B., Smith, A.J.: Sector cache design and performance. In: Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728), pp. 124–133 (2000). <https://doi.org/10.1109/MASCOT.2000.876437>
- [31] Seznec, A., Espasa, R.: Conflict-free accesses to strided vectors on a banked cache. *IEEE Transactions on Computers* **54**(7), 913–916 (2005) <https://doi.org/10.1109/TC.2005.110>
- [32] Razilov, V., Gecin, I., Matúš, E., Fettweis, G.: Conflict management in vector register files. *ACM Trans. Archit. Code Optim.* **22**(1) (2025) <https://doi.org/10.1145/3702002>
- [33] Espasa, R., Ardanaz, F., Emer, J., Felix, S., Gago, J., Gramunt, R., Hernandez, I., Juan, T., Lowney, G., Mattina, M., Seznec, A.: Tarantula: a vector extension to the alpha architecture. In: Proceedings 29th Annual International Symposium on Computer Architecture (ISCA), pp. 281–292 (2002). <https://doi.org/10.1109/ISCA.2002.1003586>
- [34] Arm Limited: Introducing SVE2 Guide. <https://support.arm.com/documentation/102340/0100/Introducing-SVE2>. Version 1.0. Learn the Architecture Series. Accessed: 2026-08-22 (2021)
- [35] Intel Corporation: Gather Data Sampling. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/gather-data-sampling.html>. Software Security Guidance Technical Documentation. Accessed: 2026-08-22 (2023)
- [36] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. *SIGARCH Comput. Archit. News* **39**(2), 1–7 (2011) <https://doi.org/10.1145/2024716.2024718>
- [37] Lowe-Power, J., Ahmad, A.M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Beckmann, B., Bharadwaj, S., Black, G., Bloom, G., Bruce, B.R., Carvalho, D.R., Castrillon, J., Chen, L., Derumigny, N., Diestelhorst, S., Elsasser, W., Escuin, C., Fariborz, M., Farmahini-Farahani, A., Fotouhi, P., Gambord, R., Gandhi, J., Gope, D., Grass, T., Gutierrez, A., Hanindhito, B., Hansson, A., Haria, S., Harris, A., Hayes, T., Herrera, A., Horsnell, M., Jafri, S.A.R., Jagtap, R., Jang, H., Jeyapaul, R., Jones, T.M., Jung, M., Kanno, S., Khaleghzadeh, H., Kodama, Y., Krishna, T., Marinelli, T., Menard, C., Mondelli, A., Moreto, M., Mück, T., Naji, O., Nathella, K., Nguyen, H., Nikoleris, N., Olson, L.E., Orr, M., Pham, B., Prieto, P., Reddy, T., Roelke, A., Samani, M., Sandberg, A., Setoain, J., Shingarov, B., Sinclair, M.D., Ta, T., Thakur, R., Travaglini, G., Upton, M., Vaish, N., Vougioukas, I., Wang, W., Wang, Z., Wehn, N., Weis, C., Wood, D.A., Yoon, H., F. Zulian: The gem5 Simulator: Version 20.0+ (2020). <https://arxiv.org/abs/2007.03152>

- [38] Balasubramonian, R., Kahng, A.B., Muralimanohar, N., Shafiee, A., Srinivas, V.: Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Trans. Archit. Code Optim.* **14**(2) (2017) <https://doi.org/10.1145/3085572>
- [39] Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D., Das, R.: Compute caches. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 481–492 (2017). <https://doi.org/10.1109/HPCA.2017.21>
- [40] Zhang, G., Sanchez, D.: Leveraging caches to accelerate hash tables and memoization. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). MICRO-52, pp. 440–452. Association for Computing Machinery, Columbus, OH, USA (2019). <https://doi.org/10.1145/3352460.3358272>
- [41] Lee, H.R., Sanchez, D.: Terminus: A programmable accelerator for read and update operations on sparse data structures. In: Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO). MICRO’24, pp. 1233–1246 (2024). <https://doi.org/10.1109/MICRO61859.2024.00092> . <https://doi.org/10.1109/MICRO61859.2024.00092>
- [42] Eckert, C., Wang, X., Wang, J., Subramaniyan, A., Iyer, R., Sylvester, D., Blaauw, D., Das, R.: Neural cache: bit-serial in-cache acceleration of deep neural networks. In: Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA). ISCA’18, pp. 383–396. IEEE Press, Los Angeles, CA, USA (2018). <https://doi.org/10.1109/ISCA.2018.00040>
- [43] Yu, X., Hughes, C.J., Satish, N., Devadas, S.: IMP: Indirect memory prefetcher. In: Proceedings of the 48th International Symposium on Microarchitecture (MICRO), pp. 178–190. IEEE, Waikiki, HI, USA (2015)
- [44] Pakalapati, S., Panda, B.: Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 118–131 (2020). <https://doi.org/10.1109/ISCA45697.2020.00021>
- [45] Ros, A.: Berti: A per-page best-request-time delta prefetcher. In: Proceedings of the 3rd Data Prefetching Championship (DPC-3) (2022)
- [46] Hall, M., Kogge, P., Koller, J., Diniz, P., Chame, J., Draper, J., LaCoss, J., Granacki, J., Brockman, J., Srivastava, A., Athas, W., Freeh, V., Shin, J., Park, J.: Mapping irregular applications to diva, a pim-based data-intensive architecture. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing. SC ’99, p. 57. Association for Computing Machinery, New York, NY, USA (1999). <https://doi.org/10.1145/331532.331589> . <https://doi.org/10.1145/331532.331589>
- [47] Barredo, A., Beard, J.C., Moretó, M.: Poster: Spidre: Accelerating sparse memory

- access patterns. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 483–484 (2019). <https://doi.org/10.1109/PACT.2019.00056>
- [48] Wang, S.-T., Xu, H., Mamandipoor, A., Mahapatra, R., Ahn, B.H., Ghodrati, S., Kailas, K., Alian, M., Esmailzadeh, H.: Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators . In: 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 1043–1062. IEEE Computer Society, Los Alamitos, CA, USA (2024). <https://doi.org/10.1109/HPCA57654.2024.00083> . <https://doi.ieeecomputersociety.org/10.1109/HPCA57654.2024.00083>
 - [49] Yang, Y., Emer, J.S., Sanchez, D.: Spzip: Architectural support for effective data compression in irregular applications. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 1069–1082 (2021). <https://doi.org/10.1109/ISCA52012.2021.00087>
 - [50] Seshadri, V., Mullins, T., Boroumand, A., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Gather-scatter dram: in-dram address translation to improve the spatial locality of non-unit strided accesses. In: Proceedings of the 48th International Symposium on Microarchitecture. MICRO-48, pp. 267–280. Association for Computing Machinery, New York, NY, USA (2015). <https://doi.org/10.1145/2830772.2830820> . <https://doi.org/10.1145/2830772.2830820>
 - [51] Gokhale, M., Lloyd, S., Hajas, C.: Near memory data structure rearrangement. In: Proceedings of the 2015 International Symposium on Memory Systems. MEMSYS '15, pp. 283–290. Association for Computing Machinery, New York, NY, USA (2015). <https://doi.org/10.1145/2818950.2818986> . <https://doi.org/10.1145/2818950.2818986>
 - [52] Lewis, J.G., Simon, H.D.: The impact of hardware gather/scatter on sparse gaussian elimination. SIAM J. Sci. Stat. Comput. **9**(2), 304–311 (1988)
 - [53] Watanabe, T., Matsumoto, H., Tannenbaum, P.D.: Hardware technology and architecture of the nec sx-3/sx-x supercomputer system. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing. Supercomputing '89, pp. 842–846. Association for Computing Machinery, New York, NY, USA (1989). <https://doi.org/10.1145/76263.1379809> . <https://doi.org/10.1145/76263.1379809>
 - [54] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Khatri, M., Mortridge, A., Collinge, J., Brooks, N., De Sutter, B., Willems, P., *et al.*: The ARM scalable vector extension. IEEE Micro **37**(2), 26–39 (2017)
 - [55] Mishra, A., Latorre, J.A., Pool, J., Stosic, D., Stosic, D., Banerjee, S., Dasika, G., Prabhu, Y., Badua, M., Gold, M., *et al.*: Accelerating sparse deep neural networks with NVIDIA Ampere architecture and TensorRT. arXiv preprint arXiv:2104.08378 (2021)

- [56] NVIDIA Corporation: NVIDIA A100 Tensor Core GPU Architecture Whitepaper. Technical report, NVIDIA (2020)
- [57] Rau, B.R.: Pseudo-randomly interleaved memory. In: Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA). ISCA '91, pp. 74–83. Association for Computing Machinery, New York, NY, USA (1991). <https://doi.org/10.1145/115952.115961>
- [58] Seznec, A., Lenfant, J.: Interleaved parallel schemes: improving memory throughput on supercomputers. In: [1992] Proceedings the 19th Annual International Symposium on Computer Architecture (ISCA), pp. 246–255 (1992). <https://doi.org/10.1109/ISCA.1992.753321>
- [59] Liptay, J.S.: Structural aspects of the system/360 model 85, ii: The cache. IBM Systems Journal **7**(1), 15–21 (1968) <https://doi.org/10.1147/sj.71.0015>
- [60] Seznec, A.: Interleaved sectored caches: reconciling low tag volume and low miss ratio. Research Report RR-2084, INRIA (1993). <https://inria.hal.science/inria-00074588>
- [61] Seznec, A.: Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In: Proceedings of 21 International Symposium on Computer Architecture (ISCA), pp. 384–393 (1994). <https://doi.org/10.1109/ISCA.1994.288133>
- [62] Chaudhuri, M., Agrawal, M., Gaur, J., Subramoney, S.: Micro-sector cache: Improving space utilization in sectored dram caches. ACM Trans. Archit. Code Optim. **14**(1) (2017) <https://doi.org/10.1145/3046680>
- [63] Stillmaker, A., Baas, B.: Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm. Integration, the VLSI Journal **58** (2017) <https://doi.org/10.1016/j.vlsi.2017.02.002>
- [64] Juan, T., Navarro, J.J., Temam, O.: Data caches for superscalar processors. In: Proceedings of the 11th International Conference on Supercomputing. ICS '97, pp. 60–67. Association for Computing Machinery, New York, NY, USA (1997). <https://doi.org/10.1145/263580.263595>
- [65] AMD: AI Engine Kernel Coding Best Practices Guide (UG1079). <https://docs.amd.com/r/2021.1-English/ug1079-ai-engine-kernel-coding/Vector-Processing-Unit> (2021)
- [66] Bornemann, F., Laurie, D., Wagon, S., Waldvogel, J.: The SIAM 100-Digit Challenge. Society for Industrial and Applied Mathematics, Philadelphia, PA (2004). <https://doi.org/10.1137/1.9780898717969> . <https://epubs.siam.org/doi/abs/10.1137/1.9780898717969>

- [67] Li, R., Wu, Q., Kavi, K., Mehta, G., Yadwadkar, N.J., John, L.K.: Nextgen-malloc: Giving memory allocator its own room in the house. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems. HotOS '23, pp. 135–142. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3593856.3595911> . <https://doi.org/10.1145/3593856.3595911>
- [68] Li, R., Wu, Q., Kavi, K., Mehta, G., Beard, J.C., Yadwadkar, N.J., John, L.K.: Speedmalloc: Improving multi-threaded applications via a lightweight core for memory allocation. CoRR **abs/2508.20253** (2025) <https://doi.org/10.48550/ARXIV.2508.20253> [2508.20253](https://arxiv.org/abs/2508.20253)
- [69] Lee, D., Hyun, B., Kwon, Y., Rhu, M.: Pim-malloc: A fast and scalable dynamic memory allocator for processing-in-memory (pim) architectures. In: 2026 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 1–17 (2026). <https://doi.org/10.1109/HPCA68181.2026.11408588>
- [70] Rolinger, T.B., Krieger, C.D., Sussman, A.: Compiler optimization for irregular memory access patterns in pgas programs. In: Mendis, C., Rauchwerger, L. (eds.) Languages and Compilers for Parallel Computing, pp. 3–21. Springer, Cham (2023)