Automatic Sublining for Efficient Sparse Memory Accesses

WIM HEIRMAN, STIJN EYERMAN, and KRISTOF DU BOIS, Intel Corporation, Belgium
IBRAHIM HUR, Intel Corporation, USA

Sparse memory accesses, which are scattered accesses to single elements of a large data structure, are a challenge for current processor architectures. Their lack of spatial and temporal locality and their irregularity makes caches and traditional stream prefetchers useless. Furthermore, performing standard caching and prefetching on sparse accesses wastes precious memory bandwidth and thrashes caches, deteriorating performance for regular accesses. Bypassing prefetchers and caches for sparse accesses, and fetching only a single element (e.g., 8 bytes) from main memory (subline access), can solve these issues.

Deciding which accesses to handle as sparse accesses and which as regular cached accesses, is a challenging task, with a large potential impact on performance. Not only is performance reduced by treating sparse accesses as regular accesses, not caching accesses that do have locality also negatively impacts performance by significantly increasing their latency and bandwidth consumption. Furthermore, this decision depends on the dynamic environment, such as input set characteristics and system load, making a static decision by the programmer or compiler suboptimal.

We propose the Instruction Spatial Locality Estimator (ISLE), a hardware detector that finds instructions that access isolated words in a sea of unused data. These sparse accesses are dynamically converted into uncached subline accesses, while keeping regular accesses cached. ISLE does not require modifying source code or binaries, and adapts automatically to a changing environment (input data, available bandwidth, etc.). We apply ISLE to a graph analytics processor running sparse graph workloads, and show that ISLE outperforms the performance of no subline accesses, manual sublining, and prior work on detecting sparse accesses.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Special purpose systems**.

Additional Key Words and Phrases: graph analytics, sparse computation

ACM Reference Format:
Wim Heirman, Stijn Eyerman, Kristof Du Bois, and Ibrahim Hur. 2021. Automatic Sublining for Efficient Sparse Memory Accesses. *ACM Trans. Arch. Code Optim.* 18, 3, Article 33 (April 2021), 24 pages. https://doi.org/10.1145/3452141

1 INTRODUCTION

Current general-purpose processors are optimized for operations on dense data: locality and predictability of memory accesses are exploited using caches and prefetchers. However, many emerging workloads, such as graph analytics, operate on sparse data: when represented as

Authors' addresses: Wim Heirman, wim.heirman@intel.com; Stijn Eyerman, stijn.eyerman@intel.com; Kristof Du Bois, kristof.du.bois@intel.com, Intel Corporation, Veldkant 31, Kontich, Belgium; Ibrahim Hur, ibrahim.hur@intel.com, Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR 97124, Oregon, USA.

XXXX-XXXX/2021/4-ART33 $15.00
https://doi.org/10.1145/3452141

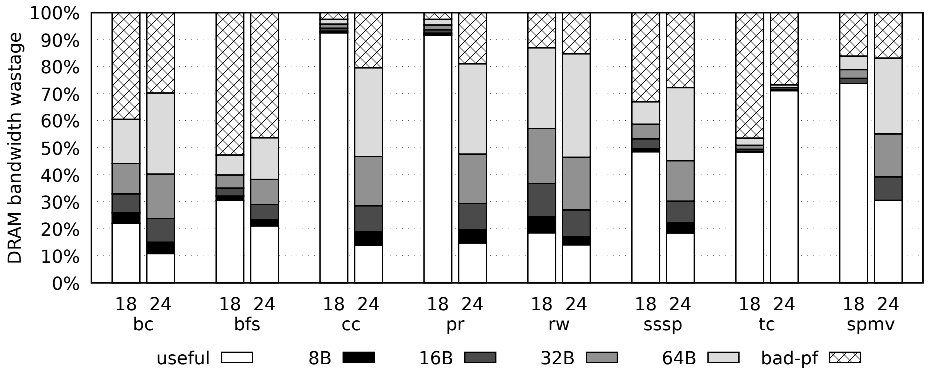

Fig. 1. Bandwidth wastage at different access granularities. When not cache-resident, sparse graph applications can waste up to 90% of memory bandwidth on fetching unused data.

a matrix, most elements are zero. Storing all zeros is inefficient, so instead, the indices of non-zeros are stored. This leads to indirect accesses, such as `A[B[i]]`: array `B` contains the indices to array `A`. Array `B` contains non-consecutive values, so the accesses to `A` are sparse: fetching single elements scattered over the full array.

Sparse accesses do not benefit from caches and traditional stride-based prefetchers. They have no spatial or temporal locality, and no easily predictable access pattern. On large data sets, most accesses need to go all the way to main memory, causing a long latency and core pipeline stall. Large sparse data structures tend to thrash the cache and evict data structures that do show locality. Moreover, they show inefficient use of memory bandwidth: for every access of a single element (e.g., 8 bytes), a full cache line (e.g., 64 bytes) is fetched from memory, leading to low bandwidth efficiency. As a result, sparse accesses not only have low performance, they also degrade the performance of more regular accesses by thrashing caches, triggering useless prefetches and exhausting memory bandwidth [12].

Because sparse accesses have to access main memory anyway, a more efficient solution is to not cache them and to only fetch one element from main memory instead of a full cache line (sub-cache-line access or *subline* access). This reduces the pressure on caches and bandwidth, which increases the performance of regular accesses.[1] However, subline accesses should be used with caution: turning a regular access—an access with locality or a predictable pattern—into a subline access can significantly *reduce* performance by turning cache and prefetch hits into misses. Relying on the programmer or compiler to detect which accesses should be sublined is therefore risky and requires profound insight into the dynamic behavior of the program. Furthermore, whether an access should be sublined depends on the input set, as smaller or denser inputs will exhibit more locality.

To illustrate the benefit of sublining, Figure 1 shows bandwidth efficiency for the sparse access applications that we use in our evaluation (see Section 4 for our setup; the X-axis shows the application and the size of the input set, corresponding to RMAT scales 18 and 24). We simulate these applications for a conventional architecture, which fetches full 64-byte cache lines from DRAM on a cache miss. The bottom bar is the fraction of the

[1]In the results section (Figure 10, "64B" case), we show that not allocating sparse accesses in cache and not training the prefetchers on them can improve performance by up to 25%, even without making use of fine-grained memory accesses.

data fetched from DRAM that is actually used by the application, which can be as low as 10%. The simulator also keeps track of how efficient different subline granularities would be. For example, the 8B component shows how much bandwidth would be wasted when using 8-byte subline accesses, the 16B component shows how much additional useless data is fetched if we fetch 16 bytes for each DRAM access, etc. The top component (*bad-pf*) shows the impact of useless prefetches. The figure demonstrates that when using 8-byte subline accesses, a large fraction of the uselessly consumed bandwidth can be avoided, compared to conventional 64-byte accesses.

In this paper, we discuss options on how to implement a memory subsystem that supports subline accesses and its impact on application performance; and propose ISLE, an automatic subline detector fully implemented in hardware [17]. We evaluate our detector in the context of a specialized graph-analytics architecture, similar to Intel PIUMA [1]. ISLE does not require changing the source code or the binary, and adapts to dynamic behavior, such as a different input set or the memory bandwidth available to the application, which could be impacted by co-running applications. In particular, this paper makes the following contributions:

- A novel subline detection mechanism is presented, added to each core, that automatically turns accesses that show sparse behavior into subline accesses.
- Compared to prior work, our mechanism does not require changing the application binary, and it can dynamically convert accesses to subline accesses and back to regular accesses when beneficial for performance.
- On a many-threaded graph processor running common graph analysis kernels, ISLE improves performance by 33.5% on average and up to 3.5× compared to not sublining, and by up to 30% over prior work that does rely on manual markings.
- We show that our mechanism adapts to changing parameters, such as a change in input set or memory bandwidth.
- We explore the impact of subline granularity (8B, 16B or 32B) on performance, assuming equal total available bandwidth (data and command lines). We show that multiplexed lines perform optimally at the cost of extra logic between the memory bus and DRAM. For dedicated command lines, 16-byte granularity is a sweet spot between exploiting subline accesses and efficient full cache line accesses.

2 BACKGROUND

In this section, we discuss the applications that show sparse behavior, the processor and memory architecture we use to evaluate our proposal, and prior work on detecting sparse accesses and cache optimizations for sparse accesses.

2.1 Applications with sparse accesses

Many big data sets are naturally sparse: connections in a social network, references between tweets or between scientific papers, etc. Sparse data also appears in machine learning applications: in the encoding of categories as a vector with only one non-zero (one-hot encoding), representing a message as a vector with counts of words from a dictionary, etc. In this paper, we focus on graph applications, due to their increasing popularity [30]. Graphs are a natural way of representing relational data, and many algorithms exist to process and analyze data in graph format. Graphs are sparse in the sense that the number of edges between vertices is much smaller than the number of all possible edges, resulting in a sparse connectivity matrix. Furthermore, when graphs represent real-world data (e.g., a social network), there is no locality or predictable pattern in the list of neighbors of a vertex [29].

2.2 Graph analytics processor architecture

Because of the importance of graph analytics, researchers have proposed several graph-specific processor architectures and accelerators [21, 31, 37]. The Cray Urika-GD graph processor [10] was a commercially available graph analytics processor. It contains multiple ThreadStorm/XMT CPUs, connected by a memory-coherent network. Each ThreadStorm core [22] has 128 threads to hide memory latency, and the system has no large caches, because of the sparse memory access pattern. More recent implementations include the Emu architecture [11] and Intel's PIUMA [1], both of which consist of many multi-threaded simple cores and support narrow main memory accesses. We use a baseline architecture for our study inspired by these examples, in addition to evaluating a more traditional out-of-order processor, see Section 4.

On large graph workloads, the performance of conventional out-of-order processors is often latency-bound as the cores wait for (hard-to-predict) memory operations to return [12]. In contrast, most specialized graph processors employ massive multithreading to keep many independent memory accesses outstanding. This allows them to largely forego on typical latency-reducing techniques such as caching and prefetching, and—assuming enough hardware threads are available—makes them bandwidth-bound on these applications. Increasing performance is therefore done by reducing bandwidth wastage, rather than hiding latency.

2.3 Cache design for sparse accesses

Sparse accesses, or more generally, accesses with low or no locality, have been recognized as an issue for conventional cache behavior, i.e., keeping the most recently accessed data. Several papers propose to add low locality accesses in the least recently used (LRU) position, instead of the default most recently used (MRU) position [25, 33], so they are less likely to evict useful data. RRIP [20] uses re-reference interval prediction to detect low locality accesses. In GPUs, cache bypassing is a popular technique to avoid allocating data streams with low temporal reuse [27, 39]. However, these techniques do not consider the effect of spatial locality and its interaction with memory bandwidth efficiency since neither CPUs nor GPUs typically support subline memory accesses. Our proposal can be used concurrently with these cache management techniques by optimizing cache efficiency for accesses that have spatial but no temporal locality (e.g., streaming accesses).

Subline accesses can be cached using a sector cache [35], which can store partial cache lines. Sector caches underutilize the available cache capacity, because empty bytes in a partial cache line cannot be used by other cache lines. Reducing the cache line size improves utilization, but it reduces hit rate for accesses with spatial locality and it increases the overhead of keeping tags. A Line Distillation Cache [34] reserves one way of the cache for fine-grained blocks (e.g., 8-byte blocks), while the other ways use the default cache line size (e.g., 64 bytes), balancing tag overhead and spatial reuse. In an Amoeba Cache [23], all cache storage can be used as either tag or data, adapting the cache block size to the needs of the application. Sector caches and the proposed alternatives involve a major redesign of the cache and related mechanisms such as coherency. Our proposal uses conventional caches, but simply bypasses them on accesses without locality.

Table 1. Configuration and efficiency of the memory bus for different subline granularities.

	64B-only	Dedicated			Muxed
Subline size	—	8	16	32	8
Command bus (packet)	8	8	8	8	8
Data bus (packet)	64	8	16	32	8/64
Efficiency @ 8B subline	—	50.0%	66.7%	80.0%	50.0%
Efficiency @ 64B	88.9%	50.0%	66.7%	80.0%	88.9%

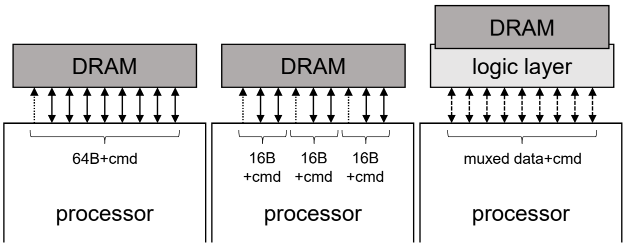

Fig. 2. Three options with equal bandwidth: 64B only, three channels of 16B each, or multiplexed bus. Each arrow represents the bandwidth equivalent of an 8B wire group.

2.4 Sub-cache-line memory and bus design

As shown in Figure 1, a lot of memory bandwidth is wasted on fetching full cache lines of which only a fraction is used. Subline accesses can more efficiently utilize the available bandwidth. However, current DRAM and memory controllers are designed and optimized for cache line sized transactions (and even bigger). In this section, we discuss the changes needed to support subline accesses, and possible design options.

First, memory needs to have many banks to support many concurrent accesses that have no locality (no page hits). The current trend in DRAM design is to increase the number of banks to increase the data rate: from 8 banks per rank in DDR3 to 16 for DDR4 [14] to 32 for DDR5 [19]. The hybrid memory cube (HMC) [9] further increases parallelism, by using vaults that consist of multiple banks, resulting in 256 banks per 4-GB rank and 256-byte rows. Therefore, we assume that memory has enough bank parallelism to support fine-grained accesses from all cores.

The memory bus consists of two types of links: command lines and data lines. The width and frequency of both types are balanced for a certain access granularity. For example, for 64B accesses and assuming 8B commands (address plus read/write bits), the data channels need to have 8× higher bandwidth than the command lines. If we decrease the access granularity, the command to data ratio increases, meaning that the command lines consume more of the available bandwidth—which is in the end limited by physical constraints such as pin count, power budget, or the chip area available for through-silicon vias when 3-D stacking. For example, for 8B accesses, one 8B command bus is needed for each 8B data bus, resulting in a 50% effective data bandwidth. In contrast, 64B accesses have a $8/9 = 89\%$ efficiency. The larger the subline granularity, the higher the data versus command ratio, and thus the higher the efficiency.

Table 1 illustrates the efficiency of multiple access granularities (8—64B). We make a difference between two possible implementations: one with dedicated command lines (Dedicated) and one with multiplexed command/data lines (Muxed). The former has one

8B command bus per channel. For example, for 16B granularity, we have one 8B command bus per two 8B busses of data, see Figure 2 (middle). Note that instead of using multiple physical dedicated command lines, existing command lines can also be clocked higher to provide higher bandwidth, combined with a subranked memory design [40]. With dedicated command lines, loading full 64B cache lines does not make use of the bandwidth dedicated to the extra command lines, limiting the efficiency of 64B accesses (see the last row of Table 1).

A more efficient, but more complex, implementation is a setup where wires can be used for both commands and data (Muxed, Figure 2 right). This requires either the use of an alternative memory standard, or a logic interface die between the memory bus and the DRAM chips. In this case we assume the DRAM chips are stacked on top of the logic die, using a higher-bandwidth connection that is not bottlenecked by the additional command busses. A similar setup is used for new, stacked memory standards such as Multi-Channel DRAM (MCDRAM) and High-Bandwidth Memory (HBM) to connect multiple channels to one or more stacked DRAM devices [2, 36]. In the Muxed setup, an 8B access consumes a total of 16B of bus bandwidth (8B command and 8B data, 50% efficiency), while a 64B access consumes 72B of bus bandwidth (89% efficiency). We evaluate this configuration to quantify the best performance that can be achieved within a limited bandwidth budget.

For simplicity, error correcting codes (ECC) are ignored in Table 1. Traditional SECDED ECC uses 8B of ECC per 64B of data, which can be divided into 1B per 8B of data [41]. This means that ECC for subline 8B accesses does not require more storage than for 64B accesses. However, storing ECC in a separate DRAM chip—as is done in the regular memory DIMM standard—might require fetching 8B of ECC for each access, including 8B accesses [40]. To avoid this overhead, ECC bits can be included in each DRAM chip, and ECC checking can be done in the memory chip itself [7, 24]. Alternatively, ECC checking can be disabled for 8B accesses and enabled for 64B accesses. In this case, our subline predictor could be reset periodically, such that all data is accessed in 64B chunks and error-corrected from time to time. Efficient error checking codes for fine-granularity accesses have been proposed [16, 28], and are orthogonal to this work.

While there are many options and still some outstanding issues in the implementation of a subline access capable memory subsystem, the ideas and results of this paper are not specific for a certain main memory implementation (DDR, HMC, HBM, etc.). Instead, we want to show the potential of automatic and dynamic selective sublining, provided the memory supports efficient subline accesses.

2.5 Prior work on sparse access detection and prefetching

Yoon et al. [41] propose DGMS, a mechanism to detect accesses with low spatial locality and to issue fine-grained (subline) memory accesses to save on bandwidth. Differently from our proposal, they store data that was fetched using a subline access in a sector cache. By storing this data in cache, they can record its reuse for future predictions. A sector cache does not solve the cache thrashing problem and adds extra complexity to the cache. Our proposal does not cache the subline access, because it has no temporal locality either, leaving more cache space for accesses that do have locality. Furthermore, conventional caches can be used. We compare our proposal to DGMS in the evaluation section.

Akin et al. [4] propose a sparse access detector without using sector caches, similar to our proposal. They assume that the programmer (or compiler) is able to mark memory operations that could benefit from subline accesses. In fact, their mechanism is targeted at filtering out marked subline operations that do show locality, and turning them back into conventional cached accesses. Thereto, they keep a table (sparse access buffer or SAB) at the memory controller, which keeps the data addresses of the last sublined accesses. Whenever a subline access hits in this table, a reuse is detected, and this access is overruled into a cached access instead of a subline access. Because this data element is now cached, it can be removed from the SAB, leaving room for tracking other subline accesses. Additionally, subline accesses are overridden when LLC MPKI is low or when the SAB has a high promotion rate. The main differences with our proposal are that we do not require input from the programmer, operating on unmodified binaries. Our proposal also turns conventional accesses into subline accesses *and* back, whatever is optimal for performance, while Akin's mechanism only performs the subline to conventional conversion. We compare our proposal to this prior work in Section 5.

Several works propose prefetchers for indirect or other irregular access patterns. Yu et al. [42] propose the indirect memory prefetcher (IMP) to prefetch sparse data that is accessed indirectly using an array with indices. Other techniques expose propose programmable prefetch engines to software [3, 13, 43]. These techniques hide the latency of sparse accesses, but do not solve the cache thrashing and bandwidth wastage if the prefetches are cached. Memory accesses made by any of these prefetchers could also be done using subline accesses, putting them into a special prefetch buffer instead of in the cache. IMP only detects and exploits indirect access patterns, while the other proposals require program changes. In contrast, our proposal works for all kinds of sparse accesses (e.g., pointer chasing code) on unmodified binaries. Nevertheless, several of these techniques can be combined, sharing the subline memory access infrastructure.

Finally, other proposals address algorithmic changes [15] or specialized hardware [32, 38] for particular graph operations such as breadth-first search (BFS) or sparse matrix multiplication, to hide or reduce the effect of sparse accesses. While these techniques can significantly speed up specific graph operation kernels, they cannot match the flexibility of a graph analytics processor based on fully programmable cores.

3 AUTOMATIC SUBLINE DETECTION

Rather than relying on the programmer or compiler to decide which accesses should (potentially) be sublined, we propose the Instruction Spatial Locality Estimator (ISLE) which automatically identifies load and store instructions that exhibit poor spatial (and temporal) locality and promotes them to subline accesses at runtime. ISLE looks for loads and stores that often miss in cache as a first indication, but contains additional mechanisms to exclude the following cases from subline conversion to prevent turning future cache hits into misses:

(a) Initial accesses to a data structure cause a burst of cache misses (cold misses) but the structure later becomes cache-resident.
(b) An instruction can bring data in cache that is subsequently reused by a later instruction (short-term reuse, e.g., when using tiling to stream through a larger data set).

3.1 Hardware implementation of ISLE

ISLE is integrated in the processor core, see Figure 3. It consists of two main components: the subline instruction table (SIT) and the subline address table (SAT), accompanied by the logic to interface with the processor pipeline and update the tables. After loads and stores are decoded, the subline detector is interrogated by the allocation stage to determine whether, for this particular execution of the instruction, it should be converted into a subline access. At the commit stage, runtime information about the execution is sent back to the detector to update its internal state, which can impact conversion of future instructions.

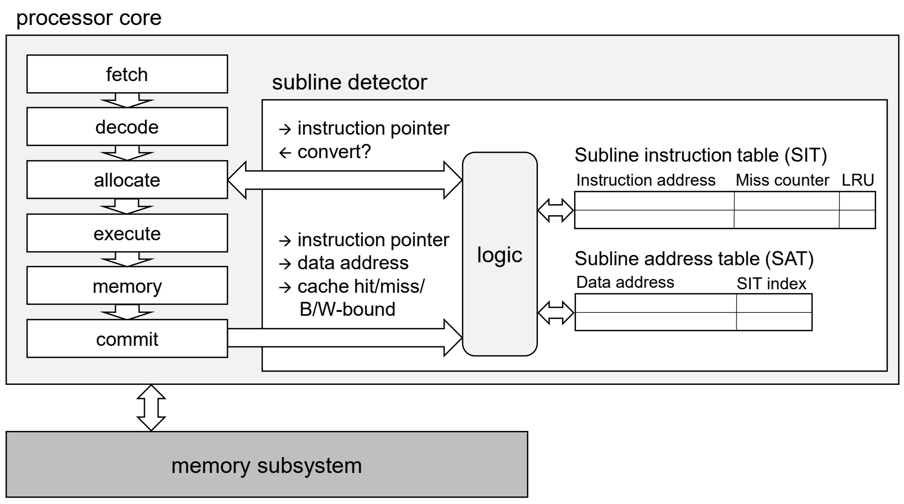

Fig. 3. ISLE is integrated into the processor core and interfaces with the execution pipeline at the allocate and commit stages.

The subline instruction table (SIT) is a set-associative structure, indexed by the instruction pointer of load and store instructions. Each entry contains an instruction pointer, the counter value for a saturating counter, and some bits for managing the replacement policy. The subline address table (SAT) is a first-in first-out (FIFO) circular queue that, for each entry, contains a data address (at cache line granularity) and an index that points to one of the entries in the SIT.

When the processor commits a load or store instruction it updates the SIT and SAT structures according to the following rules (see Figure 4). First, the instruction pointer is looked up in the SIT. If there is no match, and the instruction caused a cache miss, an entry is selected for replacement using (pseudo) LRU replacement policy. Selection may also be biased towards entries with a low counter value. A newly allocated entry is initialized by filling in its instruction pointer and resetting the counter to zero. When there was a match in the SIT, the counter value is updated as follows (saturating at zero and at a positive constant C):

(1) cache hits decrement the counter value by H,
(2) cache misses that were bandwidth bound increment the counter by B, and
(3) other cache misses increment the counter by M.

Also, if the counter value is above a threshold $T1$, the data address (at cache line granularity) is pushed onto the SAT together with a pointer value that uniquely defines this instruction's SIT entry, while the oldest entry leaves the SAT.

In addition, for all memory accesses, the data address is compared to all entries in the SAT. If there are one or more matches, the SIT index field of the oldest matching entry is used to look up the entry in the SIT, and its counter value is reduced by S.

Conversion of loads and stores to subline accesses happens in the allocate (dispatch) pipeline stage, when new loads and stores are allocated into the memory access unit before execution. The memory access' instruction pointer is looked up in the SIT. If there is a match, and the counter value is larger than the threshold $T2$, the memory access is executed using a subline access instead of a regular cached access.

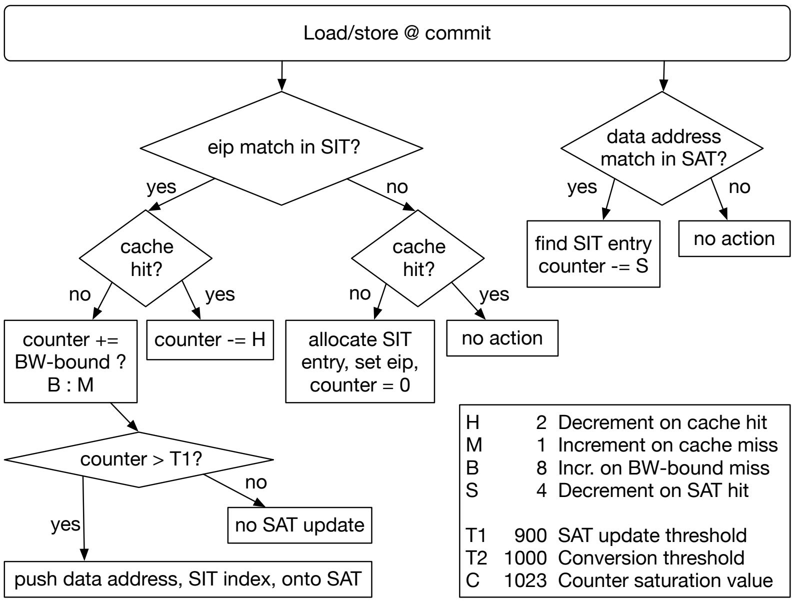
Load/store @ commit
eip match in SIT?
yes
no
cache hit?
no
yes
counter += BW-bound ? B : M
counter -= H
cache hit?
no
yes
allocate SIT entry, set eip, counter = 0
no action
data address match in SAT?
yes
no
find SIT entry counter -= S
no action
counter > T1?
yes
no
no SAT update
push data address, SIT index, onto SAT
H 2 Decrement on cache hit
M 1 Increment on cache miss
B 8 Incr. on BW-bound miss
S 4 Decrement on SAT hit
T1 900 SAT update threshold
T2 1000 Conversion threshold
C 1023 Counter saturation value

Fig. 4. The SIT and SAT tables of ISLE are updated at commit of load and store instructions.

3.2 Impact of ISLE parameters

Using the algorithm and structures described above, a load or store will be converted to use subline accesses after seeing $T2/M$ consecutive cache misses. Assuming $M = 1$ and $T2$ is near C, the maximum value of the counter C should be dimensioned such that it is larger than the number of expected initial cache misses that a cache-resident structure would see. This prevents an initial burst of cold misses to result in subline conversion—which could wrongly prevent a data structure from allocating in cache. The value of C is therefore dependent on the per-core cache capacity. On the other hand, it takes (close to) C accesses to turn a sparse access into subline, meaning that a large C value increases the warmup time of the predictor. Our evaluated setup (see Section 4) has 512 KB of cache per core, for which we experimentally found $C = 1{,}024$ as a good balance between detecting cache residency and predictor warmup.

The ratio H/M allows some selectable fraction of cache hits to be tolerated. Even large data structures that are accessed in random fashion tend to have some cache hits on a purely probabilistic basis, or there could be limited spatial locality on sparse matrices with clustered non-zeroes. With a value of $H = M = 1$, the counter value will converge towards C only when the hit rate is below 50%. Other combinations of H/M can be used to tolerate different hit ratios.

By detecting when main memory bandwidth is saturated and using a different constant for bandwidth-bound misses, B, we can dynamically adjust the hit ratio that is allowed in bandwidth-bound vs. latency-bound scenarios. Tagging individual memory accesses as bandwidth-bound can be done cheaply by detecting if they are inserted into the memory controller's queues beyond some fixed position, and propagating a bit back with the data response. Setting $B > M$ ensures that, when main memory bandwidth is saturated, subline conversion is more aggressive—hence fewer wasted bytes are transferred over a heavily loaded memory bus. In contrast, when bandwidth is not saturated the extra bytes do not impact performance yet prefetching them from main memory and allocating them in cache can improve performance even when their hit rate is relatively low.

The size of the SIT should be such that it can contain all loads and stores in the hot code path of the application, excluding those that are mostly cache-resident (remember that the SIT only allocates on cache misses). In most workloads, there are less than a few hundred instructions that need to be tracked. By biasing the replacement policy towards entries with a low counter value, those entries that are actively causing subline conversions (or are more likely to do so in the near future) are maintained over those entries that describe loads and stores with good locality, for which we need no further information. Subline conversion will happen on those memory accesses that have a SIT entry with a counter value larger than $T2$. Typically, $T2$ is near C, but could be made slightly smaller (say 90% of C) to tolerate a small number of cache or SAT hits without immediately disabling the subline conversion.

Potential reuse across instructions is detected by the SAT; onto which all data addresses for potentially subline-converted accesses are pushed. If another instruction accesses a matching data address (at cache line granularity, to detect both temporal and spatial reuse), we reduce its counter by S. This potentially reduces the counter value below the $T2$ threshold, disabling subline conversion for the instruction that originally loaded this address. The ratio of S/M (or S/B when bandwidth-bound) determines how many SAT hits are allowed for an instruction to still be marked as subline. Many workloads with random access patterns often see small amounts of spatial or temporal locality, converting them to subline will make it impossible to exploit this locality but this cost—especially in highly bandwidth-constrained scenarios—is often more than offset by eliminating the added bandwidth pressure of fetching wasted bytes. Setting S to a small value tolerates some reuse of sublined accesses, while a large value of S disables subline conversion for an instruction as soon as any reuse is detected.

Typically, the cross-instruction reuse happens quickly (e.g., spatial reuse when loop unrolling) so the SAT can be relatively small, e.g., 8 or 16 entries, which makes it cheap to check the SAT for a match. A larger SAT can be implemented by splitting it into sets indexed by the data address, and having a FIFO structure per set, at a small cost in tracking accuracy (but enabling much larger tracking coverage).

$T1$ determines when we start using the SAT to detect cross-instruction locality: a value slightly below $T2$ (say 90% of $T2$) will enable address tracking for all subline-converted accesses (as those have a counter value $> T2$), but also those accesses that may become subline in the near future. This way cross-instruction locality can be detected even before an instruction is (wrongly) converted, while simultaneously preventing the SAT from being flooded by accesses made by instructions that are not likely to be subline-converted.

Note that we used only one set of ISLE parameters for all applications evaluated in Section 5, so no workload-dependent tuning is necessary—the optimum parameter settings depend solely on hardware characteristics.

3.3 Hardware overhead

The storage budget of ISLE is modest. Assuming the configuration used in the evaluation section (see Section 4 for details), each line in the SIT has 48 bits for the instruction address, a 10-bit saturating counter and 1 bit for NRU replacement, so 59 bits in total. With 64 entries, this results in 472 bytes per core. The SAT has 16 entries, each with a 42-bit cache line address and a 6-bit SIT pointer, for a total of 98 B per core. Storage overhead can be further reduced by keeping only a part of the address bits, at the cost of some false matches and lower accuracy.

Subline conversion happens at allocate, this requires one SIT lookup and a comparison of the counter to T2 so we do not expect this to materially impact frequency targets or chip area. Updates to the SIT and SAT are performed off the critical path, at commit of load and store instructions. The implementation of the algorithm shown in Figure 4 therefore does not impact the frequency of the core, nor does it require use of specialized high-speed area and power-hungry structures.

3.4 Integration of ISLE into cache-coherent memory hierarchies

While we envision ISLE to be most beneficial in the context of specialized, highly threaded but simple graph processing cores, subline accessing can also be implemented as part of a traditional cache-coherent design with out-of-order superscalar cores. We illustrate the performance of this combination in the results section.

Coherency. Subline reads act as regular reads w.r.t. coherency: they check their own private caches, send back-snoops, use data from an on-chip copy when available, potentially cause modified or exclusive to shared transitions in other caches, and set their core-valid bit in the tag directory, to indicate that this core has used data from this cache line. If the cache line is available in its own private cache or in another on-chip cache, the cache line is not invalidated. Only when the cache line is not available on the chip, the data is not cached and the read is a truly uncached subline DRAM access. This makes sure that cache lines that are loaded by normal, non-subline loads are not invalidated, and that there is no performance hit. If the cache line is not in the requester's private cache, but it is available in a cache of another core, the cache line is not loaded in the requester's private cache. However, the core-valid bit is set in the tag directory, to ensure proper memory ordering (see below). The only exception on the normal coherence protocol is that the core doing the subline load is never set as the owner: because this core does not have a full copy of the cache line, it cannot provide data to other cores. If the core that did the subline load is the only remaining valid core for a cache line, later accesses by other cores to this line always fetch from DRAM (using a normal or subline access, depending on the type of the new load). Important to note is that subline memory operations never result in incomplete cache lines (i.e., cache lines where only part of the cache line is valid, like in sector caches). If the full cache line is available on-chip, its value is used; if not, the partial data is not cached at all.

Subline stores are handled similar to write combining stores. This means they will trigger back-invalidations into any cache that has a copy of the line. If a modified copy is available anywhere, its content is combined with the subline value being written and the result is written to DRAM, using a regular write rather than a subline write as now new data is available for the full cache line. The cache line is not kept in any core cache afterwards, which does not cause a performance hit for the cores that originally held the cache line, as a write to a cache line in a core always causes the invalidation of the copies in other cores.

Some optimizations can be made to this basic scheme, for instance, a subline read that hits a line in exclusive or modified state in another cache may choose to retain ownership of the line in the remote cache—assuming some mechanism is in place that guarantees consistency in these cases. However, in most applications the data sets accessed with sparse vs. dense patterns are distinct, so cached accesses and subline accesses do not often operate on the same addresses. Hence, such optimizations are expected to have little real-world performance

Table 2. Benchmark applications used in this study

Application	Abbr	Short explanation
Betweenness centrality	bc	Calculates the centrality of a vertex, determined by how many shortest paths go through it
Breadth first search	bfs	Walks through the graph in breadth first order
Connected components	cc	Splits the graph into subgraphs with no edges between them
Pagerank	pr	Calculates the popularity of a vertex by aggregating the popularity of its neighbors
Random walks	rw	Performs a walk by selecting a random neighbor at each step
Single source shortest path	sssp	Calculates the shortest path from one vertex to all others
Triangle count	tc	Calculates the number of triangles: three vertices that are fully connected
Sparse matrix vector multiplication	spmv	Multiplies a sparse matrix with a dense vector

benefits. In our experiments, we implemented the simple option where all subline reads downgrade remote copies to shared state.

Ordering rules. During execution inside the core, subline converted loads and stores need to remain ordered just as if they still were regular loads and stores. Because subline loads set their core-valid bit in the tag directory, the core executing a subline load will receive invalidation messages when other cores write to this line, which can be checked against the memory order queue, as normal (e.g., restarting execution from a not yet committed load for which the data on its load address has been invalidated).

Memory types. For simplicity, we apply automatic conversion only to loads and stores to cached write-back memory ranges (WB memory type in x86). Accesses to other regions (e.g., uncached (UC) or I/O ranges) are less common and usually not performance critical, and should be left unmodified. Since in x86 the memory type depends on the address, it is only known at the execute stage. Loads/stores that commit and accessed a memory type other than WB do not allocate into the SIT, and invalidate their SIT entry if they have one. Usually, the memory type accessed is constant for each instruction, e.g., an instruction that at some point accessed WB memory and got converted to subline will unlikely access UC memory at some later point. If this does happen, a subline-converted instruction accessing non-WB memory is nuked at the execute stage (after address generation and the memory type is known), its SIT entry invalidated, and execution restarts at this instruction now without converting it to a subline access.

4 EXPERIMENTAL SETUP

Workloads. We evaluate our automatic subline detector on graph and sparse matrix applications. The graph applications are taken from the GAP benchmark suite [5]. Next to the six benchmarks in the GAP suite, we implemented random walks in the GAP infrastructure. Random walks is a frequently used kernel for sampling a graph. The sparse matrix application is sparse matrix dense vector multiplication (SpMV) using a compressed sparse row (CSR) storage format, a very common kernel in sparse algebra computations. Table 2 shows an overview of the evaluated applications.

As input to the applications, we use both synthetic and real-world graphs (see Table 3). The real-world graphs were obtained from the Stanford Network Analysis Platform (SNAP) Network Data Sets [26], where we selected the largest graph from each application domain. The synthetic matrices are recursive matrices (RMAT) [8] constructed using the graph

Table 3. Input graphs

Input name	nodes	edges	avg. degree
Synthetic graphs			
RMAT-18	262,143	3,805,448	14.5
RMAT-20	1,048,575	15,699,687	15.0
RMAT-22	4,194,303	64,155,718	15.3
RMAT-24	16,777,216	260,376,709	15.5
Real-world graphs			
com-DBLP	317,080	2,099,732	6.6
web-Google	916,428	5,105,039	5.6
roadNet-CA	1,971,281	5,533,214	2.8
soc-Pokec	1,632,803	30,622,564	18.8
sx-stackoverflow	2,601,977	36,233,450	13.9
cit-Patents	3,774,768	16,518,948	4.4
com-LiveJournal	3,997,962	69,362,378	17.3

Table 4. Graph processor configuration.

Parameter	Baseline	Scaled down
Core (IO)	In-order single issue	
(OOO)	2-wide out-of-order	
L1 cache	16 KB data, 16 KB instruction	
L2 cache	512 KB unified, stride prefetcher	
Frequency	1 GHz	
# cores	512	8
DDR bandwidth	460 GB/s	7.2 GB/s

generator tool in the GAP suite. RMAT matrices have an exponential degree distribution: few vertices have many neighbors, while many vertices have few or no neighbors. We evaluate all applications on graphs with scale 18, 20, 22 and 24 (256K to 16M vertices), and an average degree of 15.

Simulation setup. To evaluate ISLE's performance, we use a modified version of the Sniper multi-core simulator [6]. Following the recommendations by Eyerman et al. [12], and similar to both PIUMA and Emu, we use a baseline architecture consisting of many in-order cores and high-bandwidth memory, but no large shared caches, see Table 4. For some experiments, we use an out-of-order core similar to Intel's Knights Landing (KNL) [36]. Each core has a 512 KB unified L2 cache with write-allocate policy, and a stream prefetcher that is trained on all cached accesses (so not on subline accesses). To limit simulation time, we use a scaled down configuration (8 out of 512 cores) for sensitivity studies and comparison with prior work. Because there is not much sharing between cores and no shared cache, this approach of scaling down a configuration is legitimate, which we also verified by comparing the results of the simulations of the full and scaled down configurations. Only with small graphs, for which there is not enough work to keep the full 512-core configuration busy, can there be a difference in terms of load imbalance. We assume that real-world graphs are large enough to avoid this issue.

We added sub-cache-line access granularity to the memory model in our simulator. Our baseline configuration uses 8-byte accesses with dedicated command lines (one 8-byte command bus per 8-byte data bus). The memory bandwidth in Table 4 is the effective data bandwidth, so the total bandwidth (command plus data lines) is double of that. We also implemented other granularities (16 and 32 B) and the multiplexed approach (see Section 2), and compare their performance in the results section. Furthermore, we added infrastructure to add markers to the source code of the applications that inform the simulator which data structures should be accessed using subline loads and stores. We use this for evaluating manual sublining and for evaluating the sparse access buffer (SAB) proposed by Akin et al. [4], which requires these markers. The markers are ignored in experiments using ILSE's automatic conversion.

To determine which data structure to mark for sublining, we run an initial simulation with all cached accesses, and record the cache efficiency for each structure (how many bytes of a cache line are effectively used). For each data structure with low cache efficiency, we add subline markers, and evaluate the impact on performance, keeping those markers that result in performance improvement for at least one input set. The markers are statically inserted into the binary, independent of the input size. For `bfs`, no subline candidate improved performance for the evaluated input sizes. We do convert one data structure with the lowest cache efficiency, to indicate that manual sublining is not able to improve performance for this benchmark.

Automatic subline detection. We implemented ISLE in the simulator, as well as DGMS [41] and SAB [4]. For both DGMS and SAB, we use the settings reported in their paper that result in the best performance. ISLE is set up with 64 SIT entries, 16 SAT entries, and configured with the parameters shown in Figure 4. As replacement policy in the SIT, we use a 1-bit non-recently used (NRU) policy.

Some loads and stores are never converted to subline accesses, these do not allocate into the SIT. This includes AVX3 vector loads and stores that access a full cache line, as well as special accesses such as uncacheable or memory-mapped I/O operations. Streaming loads and stores (`movnt` in x86) are uncached by design, we always convert them to subline accesses.

To detect which cache misses are bandwidth bound, we tag those memory requests that are inserted into the memory controller's pending read buffer when this queue is more than half full (including memory operations from other threads and applications). Once requests make it back to the core, if they result in a SIT counter increment, tagged accesses use the alternate increment value (B rather than M, see Figure 4). Many variations on this scheme exist that look at different queues or use different insertion positions as the threshold, but these do not significantly affect our results.

5 RESULTS AND DISCUSSION

We start the discussion of the results with quantifying the performance improvement obtained by ISLE and a comparison to prior work. Next, we investigate how sensitive this performance benefit is to memory bandwidth and detector parameters. Finally, we compare different implementations of the memory bus to support subline accesses.

5.1 Performance evaluation over prior work

Figure 5 plots the speedup of using uncached subline memory accesses over a baseline that implements all loads and stores as cached, 64B transactions. The *manual* variant uses static subline markers inserted by the expert programmer to decide which data structures should use subline accesses. For some benchmarks and graph sizes this leads to performance

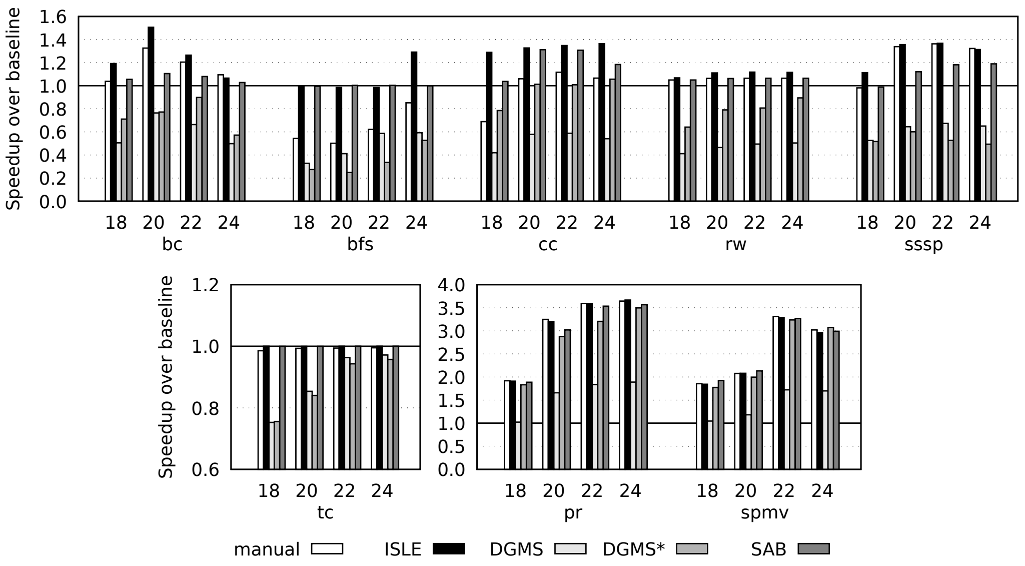

Fig. 5. Speedup over a baseline without sublining; for manual markings, our proposal (ISLE), and prior art (DGMS and SAB). ISLE is able to provide equal or higher performance than existing methods, without the need for source code changes.

degradation, because we selected at least one data structure, to make the difference with the baseline. In these cases, sublining nothing or only part of the accesses to a certain data structure is more optimal. Also, some data structures are accessed by multiple instructions with different access patterns, and would require per-instruction, rather than a per-data structure markings.

Interestingly, the benefit of manual sublining depends on the graph size. For example, for `cc`, manual sublining reduces performance by 30% for graph size 18, while performance is improved for larger graph sizes, with exactly the same markers. For small graphs, more data fits into the cache, increasing cache hit rates, hence accesses should not be sublined. For large graphs, most of the data is not in cache, increasing the benefit of sublining. This shows the need for a dynamic mechanism to efficiently make use of subline accesses.

Our approach does not need manual markings, and in all cases is able to match or outperform the performance of the best manual markers.[2] Even in those cases where manual marking was excessive (`bfs`) our technique can find the optimum—and even improve performance over the baseline—without any programmer intervention. In workloads where sublining is not useful, such as the compute-bound `tc`, ISLE does not perform any conversions so it does not cause any overhead.

Comparing with existing work, DGMS [41] builds on a sector cache implementation and tracks the set of words used in each cache line, switching between fine-grained (subline) and coarse-grained accesses depending on the average utilization. DGMS also tries to prefetch certain words inside a cache line based on historical usage, on a per instruction basis. However, for our workloads, the accesses are sufficiently random for DGMS's spatial pattern

[2]The slight performance degradation compared to manual markers in some cases, such as `spmv`, is because ISLE starts off with doing cached accesses only, and learns the workload's behavior during the first few iterations. The difference approaches zero on longer-running workloads.

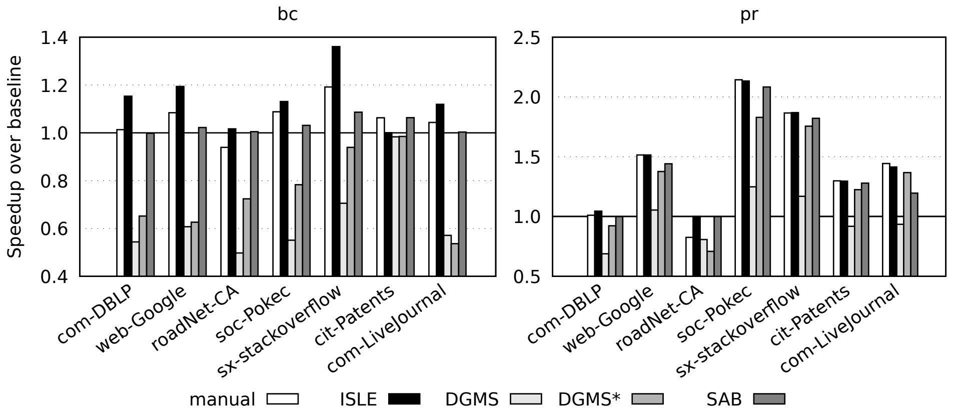

Fig. 6. Real-world graphs show diverse behavior, to which ISLE can adapt.

predictor to not fetch any useful data—worse, it degrades performance by prefetching more useless data. We therefore implemented an alternative, DGMS*, which uses DGMS's fine-grain/coarse-grain prediction but does not prefetch additional words for fine-grained accesses. Using this modification, DGMS* works well in those cases where there is a clear benefit of using subline accesses for most loads, such as `pr` and `spmv`. For other cases that are not bandwidth-bound and where cached accesses are optimal to exploit spatial locality, DGMS* is still too aggressive in its conversion to subline accesses. The reason here is that DGMS remembers only one pattern per instruction. Very often this pattern has only one or two words used, hence DGMS uses subline accesses. Spatial locality however occurs only on some cache lines touched by the same instruction, the probability of this being indicated in the single stored pattern is too low for DGMS to switch these instructions to cached mode. In contrast, ISLE maintains a global view of all accesses made by a specific instruction, and can much more accurately revert the instruction back to cached accesses if significant spatial locality is detected (either via cache hits, or SAT hits). Note that we evaluate DGMS on a different set of applications than Yoon et al. [41], and the results do not preclude that DGMS performs better on other types of applications.

SAB [4] works by relying on the programmer to mark potentially sublined loads while the hardware detects locality to undo sublining on some accesses. For the straightforward wins of `pr` and `spmv`, SAB correctly retains the subline accesses that were manually marked and yields identical performance improvements as the other techniques. In those cases where manual markings were applied excessively (`bfs` and `cc`), SAB is able to recover some performance and avoid a slowdown over not using sublining. Yet, in other cases (`cc`), SAB converts too many subline accesses back into cached accesses, reducing the performance improvement.

Figure 6 shows the results for two applications using the real-world graphs: `bc` as an example of an application with limited performance benefit, and `pr`, which has substantial benefit. The other applications have similar results. In general, the performance benefits are similar to that of the synthetic graphs. Small graphs (e.g., `com-DBLP`) mostly fit in cache, so sublining does not improve performance. Graphs with a low average degree (few neighbors per vertex, e.g., `roadNet-CA` and `cit-Patents`) also show limited benefit: less data needs to be fetched at each step of the algorithm, meaning that the bandwidth demand is low and

Table 5. Subline converted instructions (static and dynamic) for each workload.

	18		24	
	stat.	dyn.	stat.	dyn.
`bc`	6	1.9%	6	14.4%
`bfs`	0	0.0%	4	4.3%
`cc`	3	22.6%	3	34.4%
`pr`	4	8.5%	4	9.5%
`rw`	1	5.7%	1	5.8%
`sssp`	3	10.9%	4	14.7%
`tc`	0	0.0%	0	0.0%
`spmv`	4	27.6%	4	26.0%

```
 1  for (NodeID u=0; u < g.num_nodes(); u++) {
 2    NodeID comp_u = comp[u];
 3    NodeID *x = g.out_neigh(u).begin();
 4    for (int i=0; i<g.out_neigh(u).size(); i++) {
 5      NodeID comp_v = comp[x[i]];
 6      if ((comp_u < comp_v)
 7          && (comp_v == comp[comp_v])) {
 8        change = true;
 9        comp[comp_v] = comp_u;
10  } } }
```

Fig. 7. Code snippet of `cc` with sublined accesses highlighted. Only some accesses to the *comp* structure should be sublined.

fetching full cache lines does not stress bandwidth limits. Larger graphs that do stress the memory hierarchy show corresponding performance improvements; again ISLE is able to match or outperform the other techniques without relying on manual markings.

Analysis of sublined instructions. Table 5 lists the number of load and store instructions that are converted by ISLE to subline accesses. The static column lists the number of instructions, out of those instructions that make up at least 1‰ of the workload's total instruction count, the ones that are converted to subline at least 10% of the time. The dynamic column indicates how many dynamic loads and stores used a subline rather than a cached access. Typically, only a handful of static instructions need to be converted inside each application kernel, although these can represent over one third (in the case of `cc-24`) of the total number of dynamic loads and stores. More instructions are converted for large graphs than for small graphs: for instance, `bfs` has no subline conversions for the size 18 input set; while the inner loop has four static instructions, corresponding to 4.3% of all dynamic loads and stores, that are converted when running the size 24 input. A larger fraction of data fits into cache for small graphs, increasing the locality and hit rates. ISLE is able to identify this behavior, adapting the sublining fraction to the input size.

Figure 7 shows in more detail which accesses are converted in the case of `cc`. The outer loop iterates over all vertices u, while the inner loop at line 4 iterates over all neighbors v of node u. The *comp* array holds a temporary data structure that indicates for each node which cluster it belongs to. According to Figure 1, for large inputs, there is significant wastage of cache capacity and main memory bandwidth when using cached accesses. The *manual* variant marked the *comp* data structure for subline conversion as it is accessed randomly by

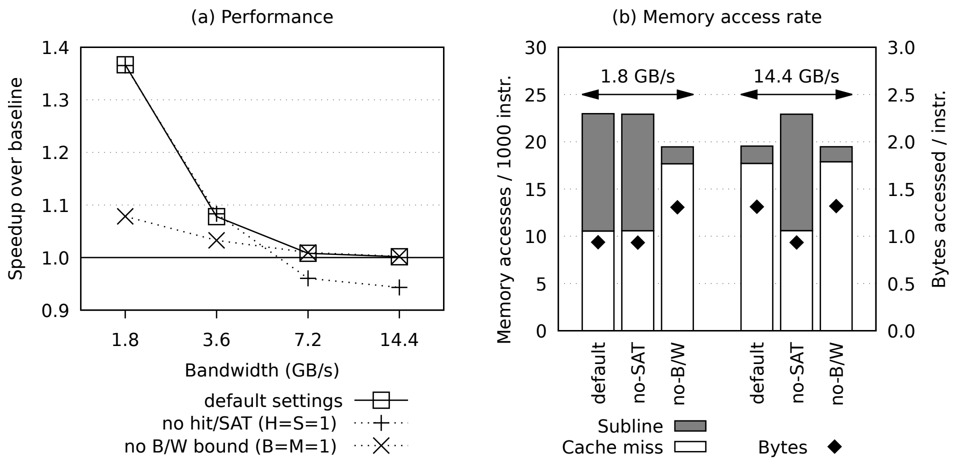

Fig. 8. Effect on performance when changing the subline detector parameters (a), and a comparison of the distribution of memory access types (b) when sweeping over total bandwidth for `bfs-22`.

the code of line 5. However, as we saw in Figure 5, the *manual* variant caused a performance degradation of up to 30% for small inputs. This is because other accesses to *comp* (notably those on lines 2 and 7) do have spatial and temporal locality. ISLE is aware of this, and only marks the array access to *comp* (but not the `x[i]`) from line 5 for conversion. This combination of cached and subline accesses even outperforms SAB which is not able to detect the spatial locality that benefits lines 2 and 7 and converts these instructions as well.

5.2 Sensitivity to memory bandwidth and detector parameters

To evaluate the effect of the different components of our subline detector, we ran experiments with different values of the detector parameters and made a sweep over total bandwidth. Figure 8 shows the results for the `bfs-22` workload, other applications and input sets show similar behavior. In Figure 8(a), we compare ISLE using the default parameter settings of Figure 4 to two alternative detectors where we reduce the impact of cache and SAT hits (by setting $H = S = 1$), or where we disable the distinction between bandwidth-bound and other cache misses (by setting $M = B = 1$).

Remember from Section 3.2 that the saturating counter will trend upwards, so sublining will be enabled, when the ratio of cache hits over cache misses is smaller than H/M (for a particular memory access instruction). In `bfs`, there is a moderate amount of spatial locality so cache hit rates are relatively high. The default detector parameters set $H/M = 2/1$, so a cache hit rate of 33% (one hit for every two misses) is enough to keep the counter value low and prevent conversion. When there is ample bandwidth available, this is the right thing to do as the latency hiding that is caused by prefetching whole cache lines leads to moderate performance improvement while spending extra memory bandwidth on unused data bytes is harmless. In contrast, the no-hit/SAT version of ISLE will subline more aggressively (with $H = S = 1$, until a 50% hit rate). This works well when bandwidth is saturated, but fails to exploit spatial locality in high-bandwidth scenarios.

Ignoring bandwidth pressure to make subline decisions, on the other hand, fails to make this trade-off by erring on the side of caution. Disabling the bandwidth-bound detection by

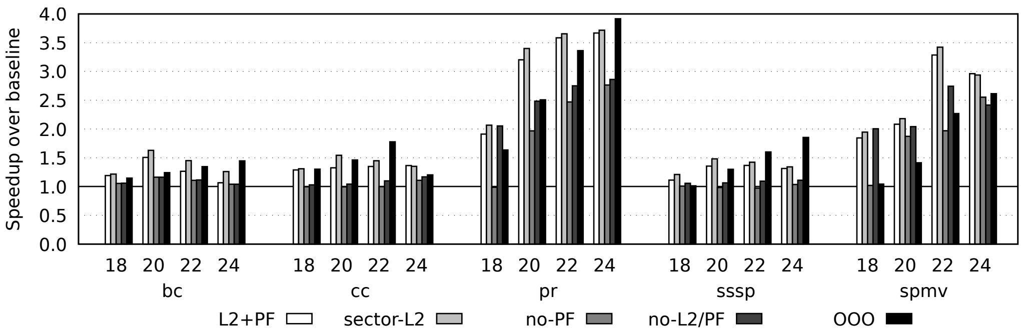

Fig. 9. Effect of subline accesses for architectures with varying core types and cache subsystems.

setting $B = M = 1$ will, on applications with moderate spatial locality such as `bfs`, convert very few instructions to subline accesses, even in low-bandwidth scenarios where the extra gain from exploiting spatial reuse is wiped out by the added bandwidth pressure of wasted bytes. The default settings of the predictor clearly perform best in all cases: converting fewer instructions when bandwidth is abundant to exploit the (limited) available locality, while converting more instructions when bandwidth is constrained.

Figure 8(b) plots the amount of main memory accesses made in each of the above cases, for the two most extreme memory bandwidth settings. At high bandwidth (low bandwidth pressure), ISLE converts few instructions, so most main memory accesses are 64-byte cache misses. At low bandwidth (high bandwidth pressure), more instructions are converted and around half the memory accesses are made with 8-byte granularity. Somewhat surprising perhaps is that when bandwidth is constrained, best performance is obtained by making *less* use of the cache, which leads to *more* memory accesses—albeit of smaller granularity so that the total memory traffic volume is lower (see the 'Bytes' markers in Figure 8(b)). This is of course due to the fact that because of limited spatial and temporal locality of these workloads, caches act as bandwidth multipliers rather than serving their usual role as bandwidth filters: for each 8B of useful data, a 64B cache line is fetched from memory.

5.3 ISLE and alternative processor configurations

Our baseline configuration has a 512 KB L2 cache and prefetchers, which improve performance on more complex algorithms with moderate working sets. Eyerman et al. [12] suggested a more extreme architecture with only 16 KB of cache per core and no prefetchers, motivated by the low memory access locality. Figure 9 evaluates the effect of ISLE when applied to a number of different memory subsystem implementations for those workloads that are sensitive to sublining. It plots performance of a system that uses our detector for selecting sublining, over a baseline with the same memory configuration that does cached accesses exclusively. The *L2+PF* option is our baseline architecture used in Figure 5 which has both an L2 cache and a stream prefetcher.

The *sector-L2* option includes a sectored L2 cache, which allocates a copy of all words that were read using a subline access (as before, subline accesses do not trigger the prefetchers). Performance is similar to the default option where the L2 is not sectored and where temporal reuse of sublined words cannot be exploited. This shows that, while ISLE is compatible with sector caches and can exploit them to obtain a small performance improvement, it does not

Table 6. Bandwidth and configuration of the memory bus for different subline granularities.

	64B-only	32B	16B	8B	Muxed
Command bandwidth (GB/s)	0.8	1.44	2.4	3.6	7.2
Data bandwidth (GB/s)	6.4	5.76	4.8	3.6	

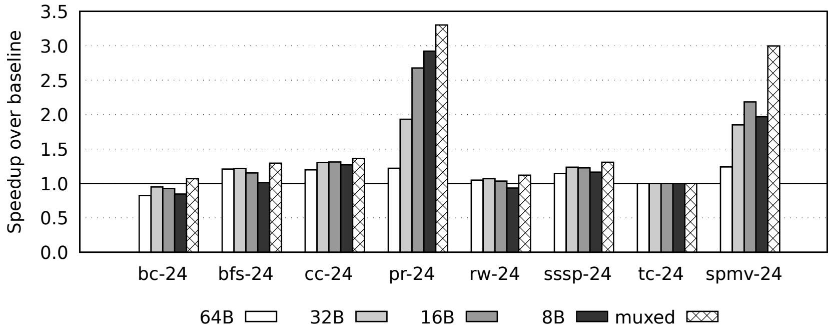

Fig. 10. Speedup over always-cached baseline for various memory configurations.

depend on sector caches either. This is in contrast to DGMS, which requires sector caches to detect lowly used cache lines.

The next option *no-prefetch* disables the L2 prefetcher. Overall, performance improvements for this case are significantly lower—indicating that a large fraction (roughly half) of the performance improvement comes from avoiding useless prefetches. When removing the effect of bad prefetches, the impact of input sizes becomes more clear: for small graph sizes (18 and 20), a larger fraction of the data fits in cache so sublining is not needed. The *no-L2/PF* option represents the extreme no-cache architecture from [12]. In all of these cases, ISLE is able to adapt each workload to the configuration to find significant performance improvements.

Finally, the *OOO* option uses a two-wide superscalar out-of-order core with more modern cache replacement (S-RRIP [20]) and prefetch throttling (NST [18]). On small inputs of `spmv`, the baseline system already throttles useless prefetches so ISLE does not provide further performance improvement; but on most workloads and input set combinations the use of automatic sublining using ISLE again provides significant performance improvement. Also note that due to the irregular and dependent main memory accesses, the out-of-order configuration does not outperform the in-order architecture, while the latter has significantly better area-normalized performance.

5.4 Implications for memory system design

In our experiments thus far we assumed that the extra command busses to support subline accesses are available, so that we could evaluate the potential speedup based on the reduction of data bandwidth in isolation. Now, we consider what happens when we keep the total off-chip bandwidth constant, and vary the subline granularity or use dedicated versus multiplexed command lines. Table 6 lists the different options, all add up to an off-chip bandwidth of 7.2 GB/s (including both command and data busses) for every eight cores. Figure 10 plots performance for all of these variations, with subline decisions made by ISLE.

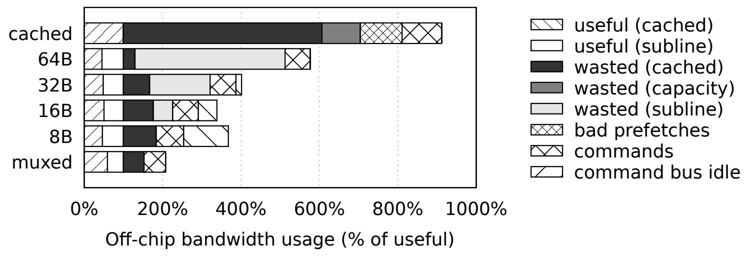

Fig. 11. Bandwidth usage for spmv-24.

The results are normalized to 64B on a memory bus optimized for 64B accesses (i.e., one 8B command line for 64B of data lines, all accesses cached).

Even though 8B/dedicated reduces the total available data bandwidth by 44%, it still provides a performance benefit for those workloads where 8B accesses cause significant savings (pr and spmv). When very few 8B accesses can be made, the dedicated 8B option can cause degradation as it also reduces efficiency for 64B accesses, the worst case being bc which degrades by 20% over the 64B-only baseline. In those cases, the muxed option is better, although at extra hardware cost.

As to access granularity, while 16B and 32B accesses still waste bandwidth by fetching unneeded data from memory, both still provide some benefit. These options also reduce the degradation of other workloads by wasting less on protocol overhead in case of dedicated command buses. If a muxed bus is too complex, a dedicated bus with 16B granularity is the best option: it does not significantly degrade performance for applications that do not benefit from sublining, while achieving close to optimal performance for applications that do benefit. The 64B option has a 64B-only memory subsystem identical to that of the baseline, but uses ISLE for cache bypassing. This yields performance improvements of up to 25%, isolating the effect of avoiding cache trashing and useless prefetching.

Figure 11 breaks down what the off-chip memory bandwidth is used for (both data and commands), normalized to useful data bandwidth. Spmv with input scale 24 is an example where sublining is very useful. The *cached* option uses a traditional memory subsystem where all accesses are cached, which wastes a lot of bandwidth on useless data bytes—because not all words inside each cache line are used (*wasted (cached)* component), because of cache capacity misses that do not occur when low-locality data is not allocated in cache (*wasted (capacity)* component), and because of useless prefetches. The 64B option uses ISLE to bypass caches and prefetchers for sparse accesses, reducing bandwidth requirements by 35%.

When sublining, most remaining cached accesses are to structures that are accessed linearly, so much less bandwidth wastage remains from cache fills that are incompletely used, or not used at all in the case of wrong prefetches. For the random access component, the application makes accesses at 8B granularity, so sublining at 16B or 32B wastes some data bytes ($^1/_2$ and $^3/_4$ of total subline traffic, respectively). With a dedicated command bus, some command bandwidth is unutilized while doing 64B accesses. In fact, the 16B version performs better than the 8B version since the time wasted on extra bytes loaded by 16B subline accesses is gained back by the fact that less bandwidth is dedicated to command busses, hence speeding up the 64B accesses. The multiplexed option does not have this disadvantage, at the cost of extra hardware complexity.

6 CONCLUSIONS

Applications with sparse memory accesses can benefit from judicious use of non-cached sub-cache-line memory accesses, saving on memory bandwidth and cache capacity. We propose the Instruction Spatial Locality Estimator (ISLE), a novel hardware subline detector that differentiates between memory operations that benefit from caching and those that should be sublined. ISLE works on unmodified binaries, requires no input from the programmer or compiler, and matches or outperforms manual subline marking and prior work; improving performance by 33.5% on average and up to 3.5× for bandwidth-constrained configurations. We also investigate the impact of choosing the subline granularity (8B, 16B or 32B) on performance and efficiency, and conclude that a configuration with multiplexed command and data busses is optimal in terms of performance but the most complex, while memory subsystems with 16B subline accesses and dedicated command lines can provide a good tradeoff between performance and complexity.

REFERENCES

[1] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, Hans-Christian Hoppe, Jason Howard, Ibrahim Hur, MidhunChandra Kodiyath, Samkit Jain, Daniel S. Klowden, Marek M. Landowski, Laurent Montigny, Ankit More, Przemyslaw Ossowski, Robert Pawlowski, Nick Pepperling, Fabrizio Petrini, Mariusz Sikora, Balasubramanian Seshasayee, Shaden Smith, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche, and Izajasz P. Wrosz. 2020. PIUMA: Programmable Integrated Unified Memory Architecture. arXiv:2010.06277 [cs.AR]

[2] Advanced Micro Devices, Inc. 2013. High Bandwidth Memory (HBM) DRAM.

[3] Sam Ainsworth and Timothy M Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. *ACM SIGPLAN Notices* 53, 2 (2018), 578–592.

[4] Berkin Akin, Chiachen Chou, Jongsoo Park, Christopher J. Hughes, and Rajat Agarwal. 2018. Dynamic Fine-grained Sparse Memory Accesses. In *International Symposium on Memory Systems (MEMSYS)*.

[5] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[6] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[7] Sanguhn Cha, O Seongil, Hyunsung Shin, Sangjoon Hwang, Kwangil Park, Seong Jin Jang, Joo Sun Choi, Gyo Young Jin, Young Hoon Son, Hyunyoon Cho, Jung Ho Ahn, and Nam Sung Kim. 2017. Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining*. 442–446.

[9] Hybrid Memory Cube Consortium. 2015. Hybrid Memory Cube Specification 2.1.

[10] Cray. 2014. Urika GD. https://www.cray.com/sites/default/files/resources/Urika-GD-TechSpecs.pdf

[11] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly scalable near memory processing with migrating threads on the Emu system architecture. In *Workshop on Irregular Applications: Architectures and Algorithms*.

[12] Stijn Eyerman, Wim Heirman, Kristof Du Bois, Joshua B. Fryman, and Ibrahim Hur. 2018. Many-core Graph Workload Analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[13] Stijn Eyerman, Wim Heirman, Kristof Du Bois, Ibrahim Hur, and Joshua B. Fryman. 2020. Indirect Memory Fetcher. US patent 10,684,858.

[14] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. 2019. Understanding the Interactions of Workloads and DRAM Types: A Comprehensive Experimental Study. arXiv:1902.07609 [cs.AR]

[15] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2008. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering* 10, 2 (2008), 20–25.

[16] Seong-Lyong Gong, Minsoo Rhu, Jungrae Kim, Jinsuk Chung, and Mattan Erez. 2015. CLEAN-ECC: High reliability ECC for adaptive granularity memory system. In *IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[17] Wim Heirman, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, and Ibrahim Hur. 2018. System, Apparatus and Method for Dynamic Automatic Sub-Cacheline Granularity Memory Access Control. US patent application 16/203,891.

[18] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[19] SK Hynix. 2018. SK Hynix Inc. Announces 1Ynm 16Gb DDR5 DRAM.

[20] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *International Symposium on Computer Architecture (ISCA).*

[21] Hai Jin, Pengcheng Yao, Xiaofei Liao, Long Zheng, and Xianliang Li. 2017. Towards Dataflow-Based Graph Accelerator. In *International Conference on Distributed Computing Systems (ICDCS).*

[22] Andrew Kopser and Dennis Vollrath. 2011. Overview of the next generation Cray XMT. In *Cray User Group Proceedings.*

[23] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. 2012. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[24] Sanghyuk Kwon, Young Hoon Son, and Jung Ho Ahn. 2014. Understanding DDR4 in pursuit of In-DRAM ECC. In *International SoC Design Conference.*

[25] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 12 (2001), 1352–1361.

[26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[27] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *International Conference on Supercomputing (ICS).*

[28] Sheng Li, Doe Hyun Yoon, Ke Chen, Jishen Zhao, Jung Ho Ahn, Jay B. Brockman, Yuan Xie, and Norman P. Jouppi. 2012. MAGE: Adaptive Granularity and ECC for resilient and power efficient memory systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC).*

[29] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[30] M. Usman Nisar, Arash Fard, and John A. Miller. 2013. Techniques for Graph Analytics on Big Data. In *IEEE International Congress on Big Data.*

[31] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *International Symposium on Computer Architecture (ISCA).*

[32] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *International Symposium on High Performance Computer Architecture (HPCA).*

[33] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *International Symposium on Computer Architecture (ISCA).*

[34] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. 2007. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *International Symposium on High Performance Computer Architecture (HPCA).*

[35] Jeffrey B. Rothman and Alan Jay Smith. 2000. Sector cache design and performance. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.*

[36] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing (KNL): Second-Generation

Intel Xeon Phi Product. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[37] William S. Song, Vitaliy Gleyzer, Alexei Lomakin, and Jeremy Kepner. 2016. Novel graph processor architecture, prototype system, and results. In *IEEE High Performance Extreme Computing Conference (HPEC)*.

[38] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[39] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU Cache Bypassing. In *Workshop on General Purpose Processing Using GPUs (GPGPU)*.

[40] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. 2011. Adaptive Granularity Memory Systems: A Tradeoff Between Storage Efficiency and Throughput. In *International Symposium on Computer Architecture (ISCA)*.

[41] Doe Hyun Yoon, Min Kyu Jeong, Michael Sullivan, and Mattan Erez. 2012. The Dynamic Granularity Memory System. In *International Symposium on Computer Architecture (ISCA)*.

[42] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[43] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. *ACM SIGPLAN Notices* 53, 2 (2018), 593–607.