Request, Coalesce, Serve, and Forget: Miss-Optimized Memory Systems for Bandwidth-Bound Cache-Unfriendly Applications on FPGAs

MIKHAIL ASIATICI and PAOLO IENNE, School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Applications such as large-scale sparse linear algebra and graph analytics are challenging to accelerate on FPGAs due to the short irregular memory accesses, resulting in low cache hit rates. Nonblocking caches reduce the bandwidth required by misses by requesting each cache line only once, even when there are multiple misses corresponding to it. However, such reuse mechanism is traditionally implemented using an associative lookup. This limits the number of misses that are considered for reuse to a few tens, at most. In this article, we present an efficient pipeline that can process and store thousands of outstanding misses in cuckoo hash tables in on-chip SRAM with minimal stalls. This brings the same bandwidth advantage as a larger cache for a fraction of the area budget, because outstanding misses do not need a data array, which can significantly speed up irregular memory-bound latency-insensitive applications. In addition, we extend nonblocking caches to generate variable-length bursts to memory, which increases the bandwidth delivered by DRAMs and their controllers. The resulting miss-optimized memory system provides up to 25% speedup with 24× area reduction on 15 large sparse matrix-vector multiplication benchmarks evaluated on an embedded and a datacenter FPGA system.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; • **Hardware → Reconfigurable logic and FPGAs;**

Additional Key Words and Phrases: High performance computing, reconfigurable computing, nonblocking caches, DRAM, cuckoo hashing, irregular memory accesses

ACM Reference format:
Mikhail Asiatici and Paolo Ienne. 2021. Request, Coalesce, Serve, and Forget: Miss-Optimized Memory Systems for Bandwidth-Bound Cache-Unfriendly Applications on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 13 (November 2021), 33 pages.
https://doi.org/10.1145/3466823

1 INTRODUCTION

FPGAs rely on hardware specialization and massive parallelism to provide acceleration despite the lower maximum clock frequency compared to CPUs, GPUs, and ASICs. However, this is feasible only when the throughput of the memory system matches that of the datapath. Important classes of applications such as sparse linear algebra and graph analytics are particularly challenging to

Authors' address: M. Asiatici and P. Ienne, School of Computer and Communication Sciences Ecole Polytechnique Fédérale de Lausanne (EPFL), EPFL IC IINFCOM LAP - INF 137 (Bâtiment INF) -Station 14 - CH-1015, Lausanne, Switzerland 1015; emails: {mikhail.asiatici, paolo.ienne}@epfl.ch.

1936-7406/2021/11-ART13 $15.00
https://doi.org/10.1145/3466823

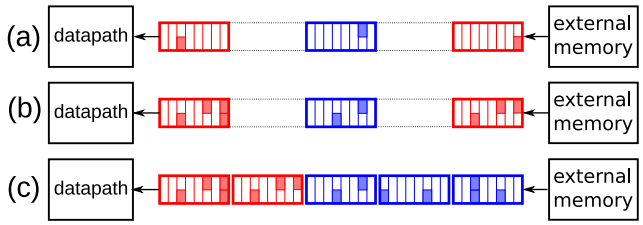

Fig. 1. Total availability and utilization of DRAM-based external memory bandwidth under short irregular access patterns. Thick big rectangles: eight-beat bursts transferred from external memory to the datapath on the FPGA, color-coded by DRAM row. Shaded smaller rectangles: portions of data actually used by the accelerators. Dashed lines: cycles where no transfers occur due to a DRAM row conflict. If requests from the accelerators are forwarded directly to the memory (a), then most of the burst content will be discarded and frequent row conflicts hamper the available DRAM bandwidth. Miss-optimized memory systems (b) improve the utilization of each burst. In addition, sending variable-length bursts on the memory side (c) reduce DRAM row conflicts, which further increase the effective bandwidth available to the accelerators.

accelerate precisely, because their memory access pattern is such that external memories operate at a very low performance point. This results in underutilization of the parallel datapath and low performance, reducing the attractiveness of hardware acceleration on FPGAs.

To tackle the problem, FPGAs provide local SRAM memories with large aggregate bandwidth and low latency, which can implement custom memory hierarchies to mediate access to external memory or store small datasets. When the hit rate of caches that can be realistically implemented on FPGA is still too low, it is often possible to conceive application-specific optimizations. However, such customized solutions are by definition expensive in design time and hard to integrate in generic hardware generation methodologies such as high-level synthesis unless the access pattern is perfectly known at compile-time [4, 10].

In this article, we will radically revisit the balancing between cache and miss handling logic to optimize the throughput of *read* operations when a large fraction of cache misses is inevitable. In addition, we will extend such **miss-optimized memory systems (MOMSes)** to interface with memory using bursts, which better match the ideal access pattern expected by DRAM-based external memories. We introduce a generic approach that can provide significant speedup with little design effort whenever the application can expose its massive parallelism by emitting thousands of outstanding reads and is memory bandwidth bound and latency insensitive. The approach is orthogonal to application-specific optimizations and, thanks to its generality, might be particularly valuable for solutions generated by high-level synthesis tools.

1.1 Irregular Short Accesses Directly on DRAM: Is It Even Worth Trying?

Custom memory hierarchy design and automatic generation usually rely on access patterns that are regular (scratchpads), have temporal and spatial locality (caches) or are at least known at compile-time (memory banking and address scrambling) [1, 4, 14, 45]. When access patterns are irregular, data-dependent and have poor locality, one is left with maximizing **memory-level parallelism (MLP)** by generating enough outstanding memory operations to make DRAM access fully pipelined. However, the throughput of the memory system is still limited to one operation per cycle per DRAM channel, at best. This imposes severe limitations on the amount of datapath parallelism that is worth implementing, limiting the advantage of using an FPGA.

In fact, one operation per cycle per DRAM channel is only a theoretical bound: The throughput that can be realistically achieved is often significantly lower, especially when accesses are irregular. If accesses are not only irregular but also narrow (such as 32- or 64-bit scalars), then

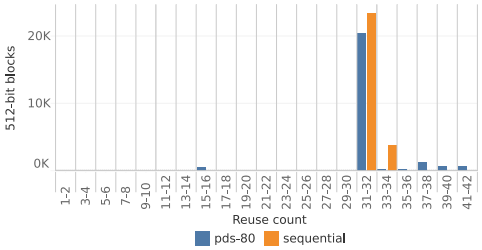

Fig. 2. *Spatial* locality. Reuse count for each 512-bit block of data, for SpMV of pds-80 and for the same number of read operations scanning sequentially the same memory space. Despite showing very different cache hit rates, both memory traces have similar amounts of data reuse across the entire application execution.

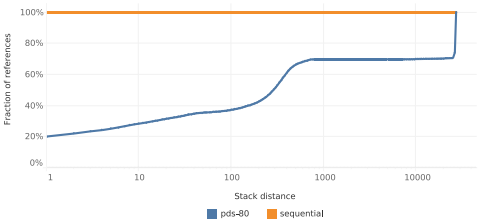

Fig. 3. *Temporal* locality. Fraction of 512-bit block references that have stack distance $\leq x$, for SpMV of pds-80 and a sequential memory trace. A large fraction of the reuses that occur in pds-80 are interleaved with references to many different blocks. Blocks can be stored in a cache hoping for future reuse; however, because large stack distances are common, cache lines are likely to be evicted before the next reuse, unless a large cache is used.

the effective bandwidth gets reduced even further. Two separate mechanisms contributes to the bandwidth degradation in those cases, as suggested in Figure 1(a). First, both DDR3 and DDR4 operate on bursts of eight beats, normally of 64 bits each [20, 21], which results in a minimum access granularity of a full 512-bit burst. If accesses are narrower than the burst size, then the remaining data returned from memory will be discarded, wasting memory bandwidth and energy. In addition, serving uncorrelated streams of requests from multiple accelerators can only be done by time-multiplexing the memory channel, canceling out any benefits due to parallelization when memory bandwidth is the bottleneck. The only way to improve bandwidth utilization, and thus performance, would be to use larger portions of each burst returned from memory.

The second mechanism relates to the organization of bits in DRAM: a few banks (8 and 16 for DDR3 and DDR4, respectively), each consisting of a two-dimensional array of capacitors. Reading data involves first *precharging* the bank's bit lines to $V_{dd}/2$, which is the average voltage between that of logic 0 and logic 1. Then, the 1 kB row to be read is *activated* by connecting its capacitors to the bank's bit lines [20, 21]. Both operations are time-consuming and must be repeated whenever the bank switches row; therefore, whenever such *row conflicts* are frequent, the actual memory bandwidth decreases significantly. DRAM controllers reorder memory requests to reduce the number of DRAM row conflicts [32]; however, because general-purpose controllers must also minimize latency, the internal request queues are relatively shallow and the optimization is possible only for accesses close in time.

1.2 Exploiting Large-Distance Spatial Locality

To discuss the opportunities to increase the utilization of each 512-bit burst received from memory, Figure 2 shows the histogram of the number of reuses of 512-bit blocks for an application with poor locality—accesses to the dense vector of 32-bit integers during **sparse matrix-vector multiplication (SpMV)** with the pds-80 matrix from SuiteSparse [11] encoded in **Compressed Sparse Row (CSR)**. It also illustrates what would happen if the same number of read operations were performed sequentially over the same address span. While both access patterns offer very similar opportunities for reuse, the sequential access pattern achieves a $\frac{15}{16} = 94\%$ hit rate on any cache with 512-bit cache lines but the hit rate of SpMV on a 128-kB direct-mapped blocking cache is only 57%. In fact,

in a cache, eviction limits the time window where data reuse could occur. For the same two applications, Figure 3 shows the cumulative frequency of *stack distances*, i.e., the number of different 512-bit blocks that have been referenced between two consecutive references to the same blocks [7]. For example, for the memory trace: {389, 261, 124, 4938, 261, 389}, the stack distances for the last access to blocks 261 and 389 are 2 and 3, respectively. While the stack distance of the sequential pattern is always zero, the SpMV cumulative histogram grows very slowly, meaning that a large fraction of reuses have large stack distance. With an ideal fully associative cache with N lines and **Least Recently Used (LRU)** replacement, reaccessing a cache line with stack distances larger than N will always be a miss; on a realistic cache, even reuses with stack distance lower than N could be misses.

1.3 Rediscovering Nonblocking Caches

The previous example shows that even applications with poor temporal locality may still have some spatial locality, which caches struggle to harness due to large stack distances between reuses. Even worst, a blocking cache actually hampers performance if the hit rate is too low to compensate for the stall cycles due to the misses. Nonblocking caches reduce stall penalties by handling one or more misses without stalling. In addition, they also group misses by cache line so that a single cache line request can be used to serve all the respective misses. This is implemented by storing each in-flight cache line in a **miss status holding register (MSHR)**, each comprising multiple *subentries* that contain the offset and source of the respective misses. This organization increases bandwidth utilization and pushes the maximum MLP beyond the DRAM latency as long as there are available MSHRs and subentries [29]. Indeed, the time window where a cache line could be reused now includes the time between the first miss and the arrival of the data. In practice, for the purpose of widening the reuse window, adding an MSHR with its subentries is equivalent to adding one cache line to a fully associative cache; however, if the number of reuses is small, then storing the miss metadata may require less bits than storing the entire cache line.

1.4 Scaling Up Nonblocking Caches

Nonblocking caches are extensively used in processors; however, MSHRs are usually searched associatively to minimize latency, which limits their number to a few tens. In practice, there is often little benefit in increasing the number of MSHRs beyond this limit on realistic CPUs [27, 36]. On FPGAs, associative searches are even less scalable than in ASICs; yet, high-throughput massively parallel FPGA accelerators that generate a large number of outstanding reads to hide memory latency [8, 28] could potentially benefit from an MSHR-rich architecture even more than a general-purpose processor.

In this article, we propose a novel miss handling architecture optimized for bandwidth-bound FPGA accelerators that perform irregular accesses to external memory. The key idea is to reuse the same wide memory response to serve multiple narrow requests from the accelerators (Figure 1(b)) on-the-fly, without relying on long-term storage in cache.

First, we show how we can efficiently implement and access thousands of MSHRs and subentries on FPGA by using the abundant on-chip SRAM. This improves bandwidth utilization and is equivalent to reordering narrow requests such that those that hit on the same DRAM burst are handled together. By supporting thousands of outstanding misses, this reordering occurs across a very large request window, which maximizes the chances of data reuse at a lower area cost than an equivalent cache.

1.5 Beyond Single Memory Requests

Repurposing some on-chip memory from cache to MSHRs generally proves to be beneficial, especially when the DRAM controller supports fully pipelined accesses and exposes the entire DRAM

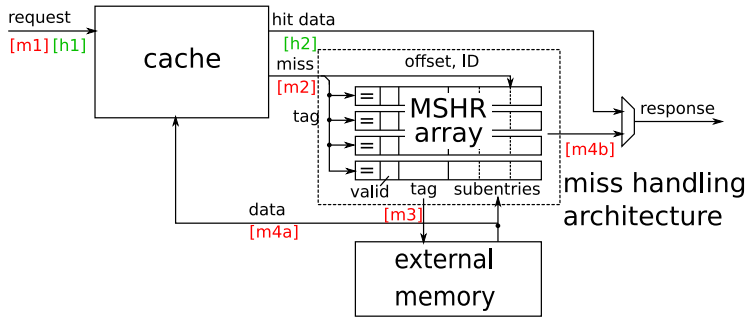

Fig. 4. Structure of a nonblocking cache. On a hit (steps [h1]–[h2]), it behaves just like any cache. On a miss (steps [m1] to [m4]), the miss address and source/ID is stored in an MSHR [m2]. Only on the first miss to a particular cache line, a memory request is additionally generated [m3]. When the cache line data are received from memory [m4], they are stored in the cache and used to respond to all its pending misses.

burst through a wide data port. However, this is not the case on DRAM controllers with multiple narrow ports, which are commonly found on System-on-a-Chip platforms. In those cases, individual memory requests do not provide enough opportunities for reuse and still result in data wastage on the memory side as they are narrower than a DRAM burst. Similarly, when the memory system is heavily optimized for bursts and individual accesses cannot be fully pipelined, optimizing the reuse of individual memory requests is generally useful but leaves some performance on the table.

To address these scenarios, we show how to extend MSHRs to support bursts of variable length on the memory controller side. When possible, we make bursts longer and exploit more of a DRAM burst or row without being limited to the data width exposed by the specific memory controller. Conversely, when spatial locality is insufficient, we keep burst short and minimize contention in the controller or avoid transferring unnecessary DRAM bursts. We will show that supporting bursts (1) makes MOMSes beneficial even behind DRAM controllers with multiple narrow ports and (2) further boosts read throughput behind wide memory ports by increasing DRAM row utilization and, when memory controllers are optimized for bursts, memory-level parallelism.[1]

Without loss of generality, we evaluate our MOMS on a simple parallel SpMV accelerator operating on a set of SuiteSparse matrices [11], which we use as representative of latency-tolerant and bandwidth-bound applications with various degrees of locality. We implemented our solution on two systems that feature three different DRAM memory controller architectures: (1) on the Xilinx ZC706 embedded board, which has two different DDR3 memory controllers and (2) on the datacenter board with DDR4 memory available on the Amazon AWS F1 instances.

2 BACKGROUND

In this section, we provide some background on nonblocking caches (Section 2.1) and review the main properties of the various on-chip memories available in modern FPGAs (Section 2.2).

2.1 Nonblocking Caches

Figure 4 shows the organization of a typical nonblocking cache. In addition to the cache array, a nonblocking cache contains a **miss handling architecture (MHA)** based on an array of MSHRs,

[1]In the reminder of the article, unless otherwise stated, *variable-length bursts* will refer to the requests sent to the DRAM controller—for example, under the form of AXI protocol bursts—rather than the DRAM bursts typical of the DDRx standards. Since we assume that the DRAM controller is given, for example by the FPGA vendor, our MOMS can directly control only the requests sent to the DRAM controller, not those to the DRAM module.

which keeps track of the in-flight misses. On the first miss of a cache line (a *primary* miss), the address of the cache line is sent to memory and stored in an MSHR; the offset of the requested word within the cache line, together with the request source/ID, is stored in a *subentry* for that MSHR. Subsequent misses to the same cache line (*secondary* misses), only require the allocation of a subentry on the same MSHR with no additional memory requests. When the missing cache line is received, it is both stored in the cache and used to serve all of its pending misses [12].

2.2 FPGA On-Chip Memory

Modern FPGAs have at least three types of on-chip memory: flip flops, LUTRAM, and **block RAM (BRAM)**. Each bit of flip flop-based memory is exposed to the FPGA fabric, providing the highest flexibility in terms of number, type, and width of memory ports and the largest bandwidth. However, flip flop bits are the least abundant and some LUTs must be consumed to implement their access logic. LUTRAMs use LUTs to realize single-, dual-, or quad-port memories with medium depth (32–64 entries). However, they compete with combinational logic for LUTs. BRAMs are dedicated memory resources implemented as hard logic. They provide higher memory density than LUTRAMs and do not require any soft logic; however, they generally provide only two ports and are optimized for narrow and deep memory arrays (at least 512 entries).The most recent FPGA families offer an additional type of on-chip memory (UltraRAM/URAM and eSRAM in Xilinx and Intel FPGAs, respectively) with even higher density and lower design flexibility than block RAMs—for example, Xilinx's URAM blocks can only be configured as 72-bit wide, 4,096-entry deep memoriesand Intel's eSRAMs have a 12-cycle read latency and one read and one write port per 72-bit wide, 86,016-entry deep channel. Therefore, the challenge is to use URAM/eSRAM as much as possible when available, followed by block RAM, LUTRAM, and flip flops.

3 KEY IDEAS

Increasing the maximum number of outstanding misses in a nonblocking cache requires scaling up both the number of MSHRs and of subentries, which will be covered in Sections 3.1 and 3.2, respectively. We then discuss, in Section 3.3, how to extend MSHRs from handling single cache lines to a variable number of consecutive cache lines, which will be retrieved from memory using a single burst request. Section 3.4 formalizes how to update the bounds of the burst request generated by each MSHR as new misses are received. To maintain an acceptable circuit complexity and a reasonable operating frequency, we accept that in some corner cases our architecture may send out redundant memory requests. We show, however, in Section 3.5 that such cases are rare under normal operating conditions, which makes handling variable-length bursts of consecutive cache lines overall beneficial in the vast majority of scenarios.

3.1 Scalable MSHR Lookup and Storage

For each additional MSHR, the memory system can handle one more primary miss without stalling; similarly, each additional subentry allows servicing an extra secondary miss with no added traffic to the external memory. Each MSHR has modest storage requirements: ~20–30 bits for the cache line tag and its valid bit, plus ~10–20 bits for offset and request ID for each of the ~4–8 subentries. This is significantly smaller than a 512-bit cache line with its tag. Therefore, within a given on-chip memory budget, bandwidth-bound applications with irregular memory access patterns could benefit more from an increase of the number of MSHRs or subentries, which increase MLP, rather than from an expansion of the cache. In practice, however, scaling up the fully associative MSHR array (Figure 5(a)) also requires additional comparators and a wider multiplexer, which increase area and hurt the critical path. Moreover, on FPGA, associative MSHRs can be mapped efficiently only

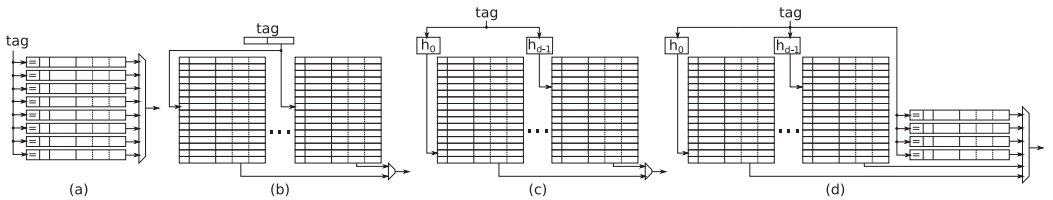

Fig. 5. MSHR-rich architectures for FPGAs. Because of the associative lookup, a traditional architecture (a) does not scale beyond a few MSHRs and MSHRs can only be mapped to flip flops. Using a set-associative memory with a single hash function (b) allows MSHRs to be mapped to block RAM but stalling on every collision results in low load factors. Cuckoo hashing (c) reduces the probability of collision and a stash (d) allows collisions to be handled in the background when the unit is idle.

to the scarce flip flops; LUTRAM- or block RAM-based **Content-Addressable Memory (CAM)** implementations require 20–300 memory bits per CAM bit [18]. However, an n-way set associative cache can be implemented with n block RAM modules.

A set-associative MSHR memory (Figure 5(b)), indexed by the lowest significant bits of the tag, can be easily mapped to block RAM and, as long as there are no collisions, lookups, insertions, and deletions can be performed in a single step. Stalling is the simplest collision handling mechanism; however, we will show in Section 6.2 that this strongly limits the maximum load factor. Using linear probing would result in *expected* constant time lookup, insertion, and deletion, and, whenever any operation cannot be completed in a single step, incoming misses must be stalled.

To overcome these limitations, we propose to store MSHRs using cuckoo hashing (Figure 5(c)). Cuckoo hashing uses d hash tables and d hash functions $h_0, \ldots, h_{d-1}$; each key x can be stored in any hash table in bucket $H_i[h_i(x)]$. Lookups and deletions require *worst case* constant time: Both involve one lookup per hash table, plus one update for deletions. For insertions, key x can be inserted in any hash table whose bucket $h_i(x)$ is empty. If all possible locations $H_i[h_i(x)]$ are occupied, then a *collision* occurs: The new key x displaces an existing entry to one of its alternative locations. If all possible buckets of the displaced entry are also occupied, then the process is repeated recursively until an entry can be inserted into an empty bucket. This means that insertions can still require more than one operation, during which no other misses can be handled. Expected amortized insertion time is constant as long as the load factor is bounded; the bound is 50% for $d = 2$ and grows very quickly with d [13]. To de-amortize insertion, Kirsch et al. proposed to temporarily store displaced entries in a small content-searchable queue (*stash*) [24] (Figure 5(d)). As soon as the input interface is idle, the module tries to insert the oldest entry from the stash; if this results in a collision, then another entry from a different hash table is moved to the stash. By doing so, entry reinsertion effectively happens in the background without slowing down incoming requests; incoming allocations are stalled only when the stash gets full.

3.2 Flexible Subentry Storage

For their explicitly addressed MSHR architecture, Farkas and Jouppi propose to use a fixed number of subentry slots per MSHR (Figure 6(a)) and to stall the miss handling architecture whenever all slots of an MSHR are used. However, waiting for the specific MSHR that is full to be deallocated may take a long time, during which the nonblocking cache may miss opportunities for merging requests to in-flight cache lines to serve them with no extra memory bandwidth cost. Increasing the number of slots per MSHR would reduce the probability of stall at the expense of an increase in area or, in other words, a decrease in load factor due to increased internal fragmentation. To mitigate these drawbacks, we propose a hybrid approach (Figure 6(b)): We store subentries in

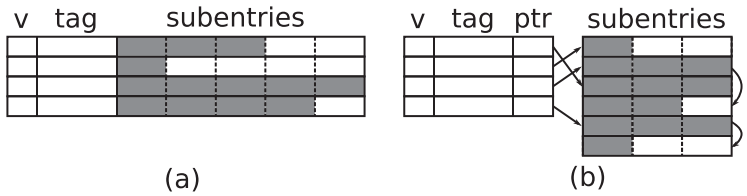

Fig. 6. Subentry organization in memory. Allocating a fixed number of subentries to every MSHR (a) results in a difficult tradeoff between a low maximum load factor and a high probability of stall, especially if there is a large variation in the number of secondary misses per cache line. Using a separate buffer to store blocks of subentries organized as linked lists (b) provides greater flexibility at a modest cost.

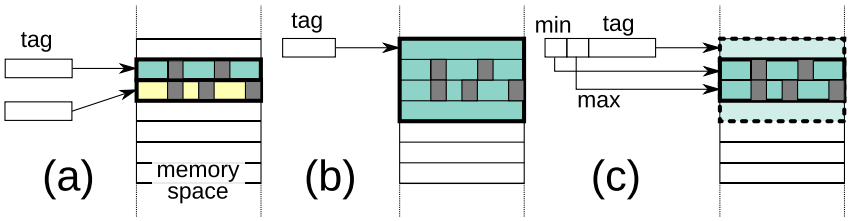

Fig. 7. MSHR memory range and structure. Portions of cache lines that have been requested by some accelerators are shown in gray. MSHRs usually refer to single cache lines (a). Increasing the memory range covered by each MSHR to a set of cache lines that will be requested as a burst (b) reduces DRAM row conflicts but may result in data wastage as the size of the burst increases. By dynamically adjusting the range of the burst (c), we make memory accesses more sequential than in (a) while minimizing data wastage.

a separate buffer and we dynamically allocate blocks of subentries to each MSHR. Specifically, the subentry buffer, mapped to block RAM, contains N_R subentry rows, each comprising n_s slots. Each MSHR is initially assigned one subentry row; whenever a row gets full, an additional row is allocated for that MSHR. Subentry rows are logically organized as a linked list: The head pointer is stored in the MSHR buffer and each subentry row contains a field for the pointer to the next row. We will evaluate the benefits of the linked-list architecture in Section 6.3.

3.3 Generalizing MSHRs from Single Cache Lines to Variable-Length Memory Areas

In nonblocking caches, MSHRs have the granularity of single cache lines (Figure 7(a)). Since cache lines are handled fully independently from each other, there are no guarantees that cache lines that are close in the address space, thus most likely on the same DRAM row, will be requested close to each other in time. If the separation between the requests is larger than the reorder window of the DRAM controller, then unnecessary row conflicts will occur.

A simple way to make use of larger portions of DRAM rows would be to increase the granularity of each MSHR to multiple cache lines (Figure 7(b)). Burst transfers can then be used to request such cache line groups efficiently. However, any cache line within the burst that is not actually needed will cause bandwidth and energy wastage. As we show in Section 6.4, this often results in lower performance than operating with single memory requests.

To strike a balance between DRAM row utilization and bandwidth wastage, we propose to have *each MSHR covering multiple cache lines* but to *dynamically adjust the bounds of the burst requested to memory* based on the cache lines that are actually needed (Figure 7(c)). In particular, each MSHR collects misses to 2^N cache lines, which corresponds to the maximum burst length. Two additional

ALGORITHM 1: MSHR burst offset handling

```
Input  : A miss at address addr = (tag, burstOffset, cacheLineOffset)
Result : Updated MSHR buffer
1  M ← MSHRBuffer.lookup(tag);
2  if M does not exist then
       /* primary miss: allocate new MSHR                                  */
3      M.tag = tag;
4      M.minBurstOffset = burstOffset;
5      M.maxBurstOffset = burstOffset;
6      M.queuePtr = enqPtr;
7      M.ignoreNextResponse = false;
8      MSHRBuffer.add(M);
9      OutputQueue.enq(M);
10 else if M.minBurstOffset ≤ burstOffset ≤ M.maxBurstOffset then
       /* (a) already within the request: do nothing                       */
11 else if deqPtr < M.queuePtr < enqPtr then
       /* (b) adjust request bounds                                        */
12     M.minBurstOffset = min(burstOffset, M.minBurstOffset);
13     M.maxBurstOffset = max(burstOffset, M.maxBurstOffset);
14     MSHRBuffer.update(M);
15     OutputQueue.update(M.queuePtr, M);
16 else
       /* (c) request should be adjusted but has already been sent         */
17     M.ignoreNextResponse = true;
18     M.minBurstOffset = 0;
19     M.maxBurstOffset = maxBurstLength-1;
20     OutputQueue.enq(M);
21     MSHRBuffer.update(M);
22 end
```

fields in the MSHR, `minBurstOffset` and `maxBurstOffset`, store the indexes of the first and last cache line that have at least one pending miss. These indexes define the bounds of the shortest contiguous burst that can serve all the pending misses.

3.4 Dynamically Adjusting Burst Bounds

Algorithm 1 describes how we implement miss handling with variable-length MSHRs. On a primary miss, a new MSHR is allocated and a memory request is inserted in the output queue; its burst initially covers only the primary miss' cache line. To enable future updates of the request, we store its address in the output queue (`queuePtr`) in the MSHR; `queuePtr` is initialized to the queue's enqueue pointer, `enqPtr`. Secondary miss handling is described in Figure 8 and implemented by the circuit in Figure 9. Secondary misses that are covered by the current burst bounds (Figure 8(a)) require no updates to the MSHR. If the current burst does not cover the new miss, then burst offsets can be adjusted as long as the memory request is still in the output queue—i.e., it has not been sent to memory yet (Figure 8(b)). We compare `queuePtr` to the current `enqPtr` and `deqPtr` to determine whether the request can still be updated.

Once the request has been sent out to memory, its burst bounds cannot be updated any more (Figure 8(c)). To handle secondary misses that fall in this case, we could, in principle, request an

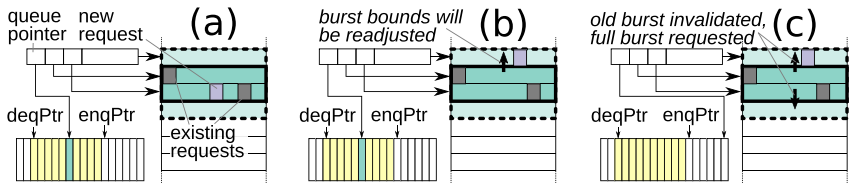

Fig. 8. Burst update policies. Requests that fall within the current burst range (a) do not require any updates; otherwise, burst bounds can be updated if the burst memory request is still in the output queue (b). If the memory request has already left the queue (c), then the current burst is invalidated and a new burst of maximum length is requested.

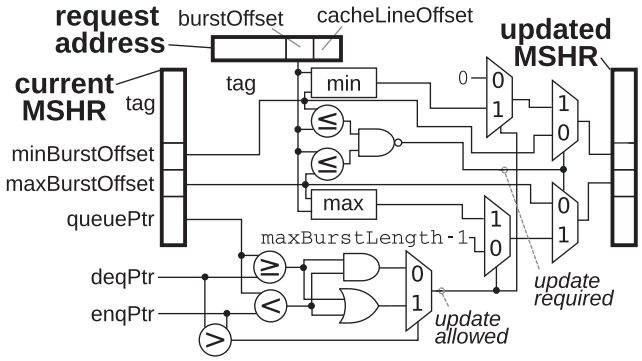

Fig. 9. Burst bounds update circuit. The updated MSHR on the right overwrites the current MSHR on the left in the following cycle; tag and queuePtr are never modified after MSHR allocation. Considering that realistic burst offsets and queue pointers are on 1–4 bits and 9–12 bits, respectively (see Sections 6.4 and 6.5), the policies shown in Figure 8 can be implemented with a relatively lightweight circuit.

additional burst only for the new cache line. However, if the second burst is not guaranteed to cover the entire burst range, then the problem may appear again once the second burst has also been sent out to memory. In the worst case, up to 2^N memory requests per MSHR may be needed. Since each burst would need a separate `minBurstOffset` and `maxBurstOffset`, the size of each MSHR would dramatically increase. Moreover, since we would also need to look up all the bursts associated to a given MSHR to determine whether any of them covers the new secondary miss or whether any of them can still be updated, the circuit in Figure 9, often already on the critical path of the entire system, would become even more complex.

To handle the cases shown in Figure 8(c) with an acceptable impact on the critical path, we take the pragmatic tradeoff of marking the in-flight request as invalid and we ask again for the full memory region—essentially, we take this for an indication of sufficiently high spatial locality. As discussed in Section 5, this policy allows us to achieve the same operating frequency as single-request MSHRs. However, discarding responses cause bandwidth wastage and should be reduced to a minimum, which is achieved by having MSHRs spend the largest fraction of their lifetime in the output queue rather than in the memory controller. If (1) accelerators generate more memory requests than the memory controller can sustain and (2) there are more MSHRs than maximum in-flight requests in the memory controller, then this happens naturally, as the next section will show.

3.5 Minimizing Burst Invalidations

Consider a memory controller that can sustain n_{mem} memory requests per cycle, accelerators that overall can generate n_{acc} requests per cycle and a memory system that has N_b banks to handle n_b requests per cycle, with $n_b \geq n_{acc} > n_{mem}$. Without loss of generality, we consider the hit rate to be negligible, thus all requests will be misses: If not, then n_{acc} is replaced by $n_{miss} = (1 - H)n_{acc}$, where H is the hit rate. At startup, the MSHR buffer is empty; therefore, all requests are primary misses. This means that each accelerator request will allocate an MSHR and generate a memory request; therefore, the number of allocated MSHRs will increase by $n_{acc} - n_{mem}$ per cycle. In other words, as long as $n_{acc} > n_{mem}$, accelerator requests naturally tend to accumulate inside the MSHR and subentry buffers without having to forcefully stall them. As the number of allocated MSHRs grows, so does the probability for future misses to be secondary rather than primary, which in turn increases the average number of accelerator requests that each memory response will serve. As a result, the MSHR allocation rate decreases to $(n_{acc} - n_s) - n_{mem}$ per cycle, n_s being the secondary misses per cycle. If MSHRs and subentries were unlimited, then the system will tend to $n_{s,eq} = n_{acc} - n_{mem}$, i.e., each memory response is reused $\frac{n_{acc}}{n_{mem}}$ times on average and the number of MSHRs remains constant at some value $N_{MSHR,eq}$. If the system runs out of MSHRs or subentries before reaching equilibrium, then it will start to stall incoming requests: This reduces n_{acc} to $n_{acc'}$ and moves the equilibrium point to $n_{s,eq'} = n_{acc'} - n_{mem} < n_{s,eq}$. The larger the MSHR and subentry buffers, the closer $n_{s,eq'}$ will be to the ideal $n_{s,eq}$.

An application with good locality will reach $n_{s,eq}$ very quickly with few MSHRs; the poorer the locality, the higher $N_{MSHR,eq}$. If $N_{mem,IF}$ is the total number of in-flight requests that the memory controller can sustain, then each memory request will spend $\frac{N_{mem,IF}}{N_{MSHR,eq}}$ of its lifetime inside the memory controller and the rest inside the MSHR buffer output queue. Therefore, the higher $N_{MSHR,eq}$, the more likely the burst bounds of an MSHR can still be adjusted without having to invalidate the first burst, and $N_{MSHR,eq}$ will naturally tend to be higher for applications with poor locality where most of the full burst will likely not be used.

To reduce invalidations on regular applications that tend to have a low $N_{MSHR,eq}$, we tried to artificially stall memory requests until a minimum number of used MSHRs was reached or a timeout since the last received request expired. In practice, excessive stalling was usually more harmful than invalidations unless extensive application-specific fine tuning of the minimum MSHR occupation and the timeout were performed, which is incompatible with the desired generality of the proposed memory system.

4 DETAILED ARCHITECTURE

Figure 10 shows the top-level view of our miss-optimized memory system. To simplify the design and to maximize the scope for memory access optimization, our controller can return responses out of order, which is not unusual among high performance memory systems [15, 16]. Therefore, requests must be tagged with an ID, which will be used to match it with the corresponding response. Requests received from each of the N_i input channels are redistributed across N_b banks by means of a crossbar. We use a multi-banked structure to handle multiple requests and responses per cycle. We maximize workload balancing among banks by interleaving requests among them in the finest possible way: Requests pertaining to consecutive MSHR memory ranges (i.e., the dashed area in Figure 7(c)) are served by different banks. Each bank consists of a set-associative cache, an MSHR buffer, a subentry buffer, and a data buffer. Data for requests that hit in the cache are immediately returned to the crossbar, while misses are handled and stored by the MSHR and subentry buffer. The MSHR buffer has been extended to handle bursts by implementing Algorithm 1, which defines how to allocate and update MSHRs and how to manage the output queue. The queue depth

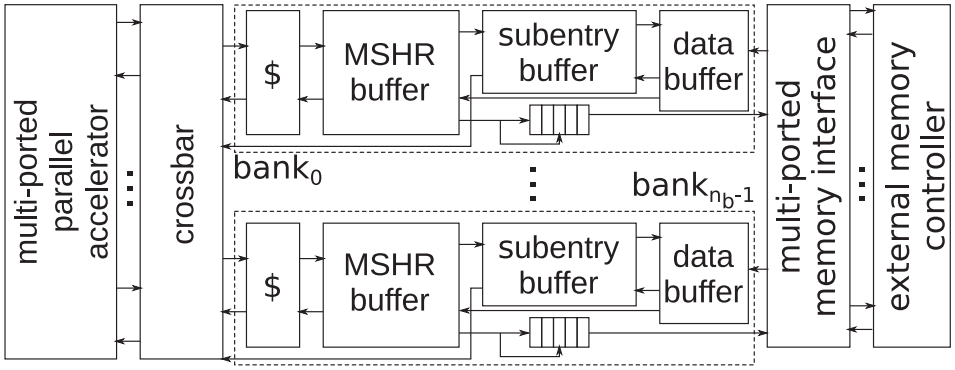

Fig. 10. Top-level view of our MOMS. A crossbar steers memory requests from N_i accelerators to N_b banks according to their address. Each bank consists of a cache, an MSHR buffer with updatable queue, a subentry buffer, and a data buffer. The multi-ported memory interface multiplexes each memory interface among banks.

corresponds to the size of the MSHR buffer: Even if each MSHR can generate an additional memory request with the full burst, it does so only if the partial burst has already left the queue, so the queue will never host more than one request per MSHR at a time.

The external memory interface can handle $n_{mem} \geq 1$ memory ports, where n_{mem} is a divisor of n_b. It includes one arbiter/demultiplexer per memory port, each connected to $\frac{n_b}{n_{mem}}$ banks. Therefore, each bank is statically assigned to a memory port, each of which currently uses a round-robin arbiter to pick requests from its banks. This simple solution avoids having to instantiate another crossbar and works well if ports are symmetric (as in our experimental system) and requests are reasonably well distributed among banks. The memory interface can be easily modified when these assumptions do not hold.

4.1 MSHR Buffer

For the MSHR buffer, we use one block RAM per hash table, with the address of the MSHR memory range (*burst tag*) as key. We use universal hash functions in the form $h_a(x) = (ax \bmod 2^{w_t}) \operatorname{div} 2^{w_t - w_M}$ with w_t being the number of bits of the tag, $w_M = \log_2(M)$ where M is the number of buckets per hash table, and a is a random positive odd integer with $a < 2_t^w$ [42]. Each bucket contains a valid bit, the missing burst tag, and the address of the first subentry row in the subentry buffer as described in Section 3.2. The stash is a content-associative memory made of flip-flops. To integrate the stash in the pipeline, we include the stash entries among the locations that are searched and updated during lookups or that can be deallocated when a response is received. On an MSHR hit (i.e., secondary misses), we use the circuit in Figure 9 to update the burst bounds if necessary and possible

4.2 Subentry Buffer

Figure 11 shows implementation and operation of the subentry buffer. A subentry consists of an (ID, burst offset, cache line offset) tuple; a subentry row contains (a) n_s subentry slots, (b) the number of allocated subentries, and (c) a pointer to the next subentry row with its valid bit. To allocate a subentry, the first subentry row is retrieved from the buffer. If the row is not full (1), then the new entry is appended and the row is written back to the buffer. If the row is full (2), then a new row must also be allocated. We use a FIFO (*free row queue*, FRQ) to store the addresses of the empty rows, and allocating a row simply means extracting the first element of the FRQ. The FRQ

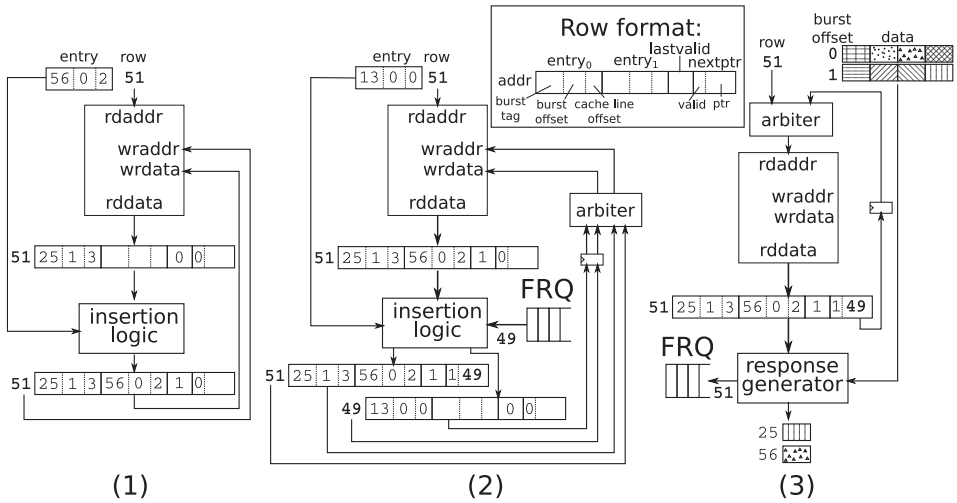

Fig. 11. Block diagram and operation of the subentry buffer. For requests, the subentry buffer receives ID, burst offset, cache line offset, and the address of the first subentry row (head row) from the respective MSHR. The head row is first retrieved from the buffer. If it is not full (1), then the row is updated with the new entry and written back to the buffer. If the row is full (2), then the new entry is inserted in a new row, whose address is stored in the previous row. When a response is received (3), all subentries are retrieved by traversing the subentry row list. After all subentries have been forwarded to the response generator, the row is deallocated by pushing its address to the free row queue.

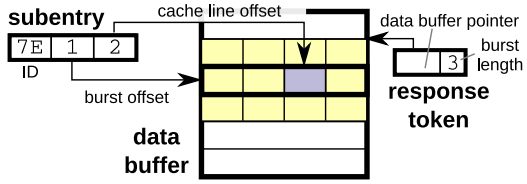

Fig. 12. Retrieval of responses from the data buffer. The response token, generated once the entire burst has been received, locates the response data inside the data buffer (yellow). Responses to the individual pending misses can be generated by iterating over all subentries and extracting the relevant word (purple) based on burst and cache line offsets.

is also shared with the MSHR buffer to allow the allocation of the first subentry row for newly allocated MSHRs. When the FRQ gets empty, further allocations are stalled.

When a response is received, the corresponding MSHR is deallocated from the MSHR buffer and its subentry rows retrieved from the buffer. The response generator parses the subentry rows, retrieves the data from the data buffer, and emits one response per allocated subentry. The row is then recycled by inserting its address into the FRQ and the process is repeated for the entire linked list of rows.

4.3 Data Buffer

The data buffer store bursts received from memory. Because responses are treated in-order inside the pipeline, the data buffer can be implemented as a simple circular buffer in LUTRAM or block RAM. After storing the burst in the circular buffer, the module forwards its base address (pointer), size (burst length), and tag to the pipeline. The MSHR buffer will use the tag to deallocate the

MSHR pertaining to the burst and retrieve its subentries; base address and size are then used by the subentry buffer to generate the responses as shown in Figure 12. When all pending misses have been served, burst pointer and size are used to deallocate the burst data inside the data buffer.

4.4 Pipeline Efficiency and Throughput

As long as an MSHR has a single subentry row, the primary and all secondary misses can be handled without stalling the pipeline as they require no more than one read and one write per dual-ported block RAM: lookup in the MSHR buffer, allocation of the MSHR for primary misses, lookup in the subentry buffer for secondary misses, and row update in the subentry buffer. Each block RAM has a data forwarding circuit to ensure that we always read the most up-to-date data despite reads having two-cycle latency. MSHR collisions are handled transparently when the unit is idle, as long as there are free entries in the stash. Allocating an additional subentry row requires stalling the pipeline for one cycle to perform two writes: (1) inserting the pointer of the newly allocated row into the tail of the list and (2) writing the new subentry into the newly allocated row. Allocating a subentry on an MSHR that has more than one row requires traversing the linked list, which costs an extra read per additional row. The traversal cost can be significant for MSHRs with many subentries: To mitigate it, we use an 8-entry fully associative cache indexed by the head pointer of the subentry list to jump directly to the tail whenever possible. In our subentry architecture, the tradeoff between internal fragmentation and stall cycles, which depend on the number of subentries per row, remains; however, the cost of a full subentry row is reduced from completely stalling the pipeline until the full MSHR is deallocated to a few bubbles in the pipeline. Responses whose MSHR has a single subentry row can also be handled without stalls; each additional subentry row costs one stall cycle.

Most of the operations are therefore fully pipelined, with the caveat that a single pipeline is shared between accelerator requests and memory responses. However, the more secondary misses we can merge to the same memory burst and the longer the bursts, the fewer independent memory responses we will have to handle, reducing the cost of pipeline sharing. Ultimately, N_b fully pipelined banks can supply up to $N_b - n_{bursts,mem}$ responses per cycle, where $n_{bursts,mem}$ is the average number of bursts per cycle returned by the external memory.

5 EXPERIMENTAL SETUP

Our memory controller is written in Chisel 3 and is fully parametric in terms of, e.g., number of inputs, banks, MSHR cuckoo hash tables, MSHRs per hash table, subentry rows and number of subentries per row, cache size and associativity, input and output data size. Even though it has been evaluated on two Xilinx platforms (described more in detail in Section 5.2), it is written in platform-independent RTL. We compiled it using Vivado 2017.4 for the experiments on the ZC706 platform and Vivado 2019.1 for the AWS F1 FPGA. In the remainder of this section, we will introduce the sparse matrix-vector accelerators and the matrices that we used as a benchmark (Section 5.1) and then present the FPGA boards where we ran our analysis, with special emphasis on the properties of the memory systems (external DRAM memories and respective controllers) (Section 5.2). We conclude in Section 5.3 by discussing the top-level system design and illustrating how the MOMS can be floorplanned among multiple FPGA dies.

5.1 SpMV Accelerators

As a benchmark, we implemented a simple accelerator for SpMV, an important kernel in a broad range of scientific applications [3] and to which many sparse graphs algorithms can be mapped [23]. Moreover, SpMV can easily generate a wide range of access patterns depending on the matrix sparsity pattern. Our accelerator, shown in Figure 13, is an almost direct implementation of

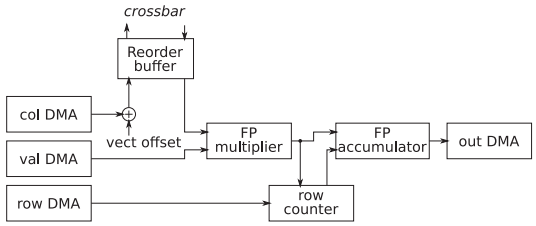

Fig. 13. Structure of our benchmark sparse matrix-vector multiplication accelerator. Xilinx AXI DMAs are used to stream all CSR vectors accessed sequentially. The values of the `col` array are used to compute the addresses of the vector elements that are retrieved through our memory controller.

ALGORITHM 2: Sparse matrix-vector multiplication (SpMV)

```
1: for r ← 0 to ROWS − 1 do
2:     out[r] ← 0
3:     for i ← idx[r] to idx[r + 1] do
4:         out[r] ← out[r] + val[i] × vect[col[i]]
5:     end for
6: end for
```

Algorithm 2 for SpMV of a CSR-encoded sparse matrix; we do not include any SpMV-specific optimizations as our controller aims for a generic architectural solution for any applications with irregular memory access pattern. Indices are 32-bit unsigned integers while values are single-precision floating point values. All CSR vectors, accessed sequentially, are provided via AXI4-Stream through Xilinx AXI DMA IPs; the dense vector, accessed randomly, is read through an AXI4-MM port connected to our memory controller. The 8,192-entry reorder buffer provides the vector values to the multiply-accumulation pipeline, which is based on floating-point Xilinx IPs. We use the index vector to clear the accumulator every time a new row begins, and the output vector is streamed to DRAM through a DMA. Each accelerator can process one **non-zero matrix element (NZ)** per cycle; we parallelize the SpMV by interleaving rows across multiple accelerators.

Table 1 shows the properties of our benchmark matrices, which are essentially the largest benchmarks used in prior work on SpMV [3]. All benchmarks are available on SuiteSparse [11]. We use the stack distance, introduced in Section 1.2, to characterize the regularity of the access pattern to the dense vector.

5.2 FPGA Boards

Tables 2 and 3 show the main properties of the FPGAs and the memory systems used in our experiments. The ZC706 is an embedded board that contains a Zynq-7000 SoC with an FPGA and two ARM cores. The SoC is connected to 1 GB of DDR3 on the **processing system (PS)** side through the ARM hardened memory controller and 1 GB of DDR3 on the **programmable logic (PL)** side through the Xilinx MIG soft controller. For the PS memory, we only used the four HP ports as we found them to be enough to achieve the peak memory bandwidth: Using the ACP port brings no performance advantage and complicates the design as the number of ports is not anymore a power of two. However, the FPGA system available on the Amazon AWS F1 instances consists of a Virtex UltraScale+ FPGA connected to four DDR4 channels driven by Xilinx MIG controllers. Because the maximum number of outstanding reads per channel is limited to about half of the average number of cycles of latency, the quoted peak bandwidth of 59.9 GiB/s can only be achieved using bursts of

Table 1. Properties of the Benchmark Matrices We Used

Benchmark	Vector size (MB)	Rows (M)	Non-zero elements (M)	Stack distance percentiles		
				75%	90%	95%
amazon-2008	2.81	0.735	5.16	6	6.63k	19.3k
cit-Patents	14.4	3.78	16.5	91.1k	129k	151k
cont11_i	7.48	1.47	5.38	2	2	3
dblp-2010	1.24	0.326	1.62	2	348	4.68k
eu-2005	3.29	0.863	19.2	5	26	69
flickr	3.13	0.821	9.84	3.29k	8.26k	14.5k
in-2004	5.28	1.38	16.9	0	4	11
ljournal	20.5	5.36	79.0	19.3k	120k	184k
mawi1234	70.8	18.6	38.0	20.9k	176k	609k
pds-80	1.66	0.129	0.928	26.3k	26.6k	26.6k
rail4284	4.18	0.004	11.3	0	13.3k	35.4k
road_usa	91.4	23.9	57.7	31	601	158k
webbase_1M	3.81	1.00	3.10	2	19	323
wikipedia-20061104	12.0	3.15	39.4	47.3k	105k	137k
youtube	4.33	1.13	5.97	5.8k	20.6k	32.6k

We found the stack percentiles [7] to be a better predictor of performance than, e.g., sparsity. All vectors are of single-precision floating point values. The number of columns corresponds to the vector size divided by 4 bytes and, except for pds-80 and rail4284, it corresponds to the number of rows.

Table 2. Specifications of the FPGAs Used in Our Experiments

Platform	ZC706	Amazon AWS F1
FPGA	Zynq-7000 xc7z045	Virtex UltraScale+ xcvu9p
LUTs	218,600	1,182,000
Flip-Flops	437,200	2,364,000
DSPs	900	6,840
36 kib BRAMs	545 (2.4 MiB)	2,160 (9.5 MiB)
URAMs	0	960 (33.8 MiB)
SLRs	1	3

Table 3. Specifications of the External Memory Systems Used in Our Experiments

Platform	ZC706		AWS F1
Memory controller	PS	PL	
Memory technology	DDR3	DDR3	DDR4
Total size (GiB)	1	1	64
DDR IO data width (bits)	32	64	64
Memory channels	1	1	4
Controller ports	5 (4 HP, 1 ACP)	1	4 (1 per channel)
Controller data width (bits)	64	512	512
Controller clock frequency (MHz)	150	200	250
Peak measured bandwidth (GiB/s)	3.9	12.0	59.9

length two or larger; if only single requests are used, the maximum measured bandwidth drops to 32.0 GiB/s.

Compared to the ZC706 FPGA, the AWS F1 FPGA contains 5.4× more CLBs (LUTs and FFs), 4.0× more BRAM blocks, and 33.8 MiB of URAM. However, 25% of those resources are locked in

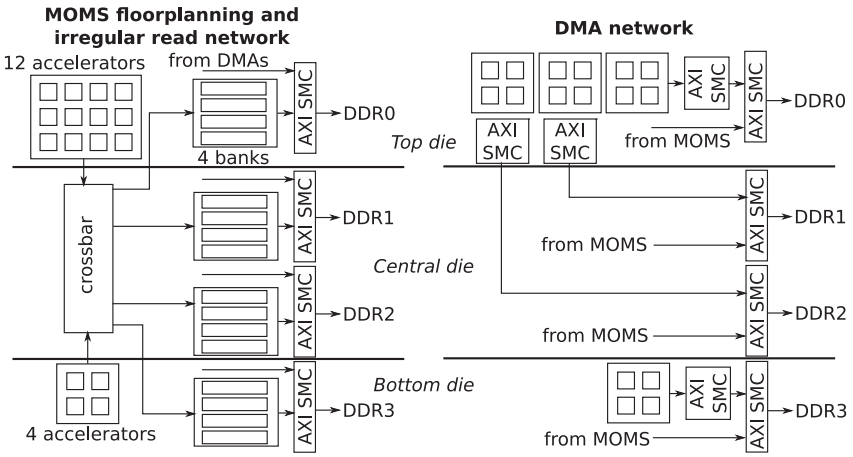

Fig. 14. SLR partitioning of SpMV accelerator and MOMS on the AWS F1 FPGA. The thick lines represent the SLR boundaries. For the sake of clarity, the irregular read and DMA networks are shown separately even though they are implemented simultaneously side by side. Each memory channel is shared among four MOMS banks and the DMAs of four accelerators using an AXI SmartConnect.

FPGA regions reserved to the AWS shell and thus unavailable to the designer. The remaining 75% are spread unevenly among three dies (or Super Logic Regions, SLRs, in Xilinx's terminology) as only the central and bottom SLR are partially occupied by the shell. The DDR4 controllers are also spread among SLRs, with the top and bottom SLR hosting one controller each and two controllers in the central SLR [2]. Because inter-die interconnects (Xilinx's Super Long Lines or SLLs) are particularly scarce and slow, achieving high resource utilization on multi-SLR devices is especially challenging and needs to be explicitly taken care of during system design. In the next section, we will illustrate how the MOMS modules and the accelerators have been assigned to memory controllers and, in the case of the AWS F1 FPGA, SLRs to achieve high performance and resource utilization even on high-end multi-SLR FPGAs.

5.3 Top-level System Organization

On the ZC706 board, we consider two different configurations, which we take as representative of realistic use cases in commercial FPGA systems. In the *PL system*, the dense vector is stored in the PL DDR while sequential vectors are read from the PS memory; the opposite is done in the *PS system*. In the PL system, it is the highest performing memory, exposed through a single, wide port, that is accessed irregularly. Since the PL DDR is often the system bottleneck due to the irregular accesses, we maximize its bandwidth by operating at 200 MHz. Ultimately, the system throughput is limited to ≈2.4 **multiply-accumulations (MACC)** per cycle by the bandwidth of the PS memory that hosts the sequential vectors. Therefore, four accelerators and four banks are enough to saturate it. On the PS system, the sequential accesses on the PL DDR allow up to ≈8 MACC/cycle, while the PS DDR limits the throughput to ≈6.5 (150 MHz) or ≈4.9 (200 MHz) MACC/cycle if each 32-bit word returned by the PS DDR is used exactly once (which, we found, is often an optimistic assumption). Since the performance is always limited by the PS DDR whose bandwidth does not increase past 150 MHz, we ran the system at 150 MHz. This eases timing closure and allowed us to implement eight accelerators and eight banks connected to the four 64-bit HP ports.

As discussed in Section 5.2, the external memory bandwidth of the AWS F1 board is about 4× larger than that of the ZC706. Considering that the four memory channels are symmetric, we do not artificially introduce any asymmetry and share all of them between DMAs and MOMS. Since each MACC requires at least 12 bytes—one 32-bit value from each of the vectors `val`, `col`, and `vec`, neglecting the 8 bytes per row of `row` and `out`—the maximum theoretical performance with a DDR4 bandwidth of 59.9 GiB/s is of 21.4 MACC/cycle. However, even though the AWS F1 FPGA offers more resources than the ZC706 one, they are harder to exploit as they are scattered among three SLRs. It is indeed required to (1) spread the logic as uniformly as possible among SLRs while (2) minimizing the number of SLR crossings, under the constraints that (a) the top SLR has about 60% more resources than the other two and (b) the central SLR has two memory controllers while the others have only one. In addition, the sequential accesses performed by the DMAs (Figure 13) should be as balanced as possible among memory channels.

Figure 14 shows the system organization that allowed the implementation of 16 accelerators and 16 banks running at 250 MHz. This is the highest performing system that we could implement, which makes our experimental setup on the AWS F1 FPGA resource-bound—mostly due to the AXI DMAs and fixed infrastructure, as discussed in Section 6.8—unlike the ZC706 that was bandwidth-bound.

To minimize SLR crossings, we implement the crossbar in the central SLR and MOMS banks in the same SLR as the respective memory controller. Therefore, the central SLR hosts the crossbar and eight banks while the top and bottom SLR have four banks each. As for the sequential accesses, we assign four accelerators' DMAs per memory channel. Also for SLR crossing minimization, we keep accelerators in the same SLR as the respective DMAs' memory channel whenever possible. We make an exception for the accelerators connected to the central SLR's channels, which are moved to the least congested top SLR. All of the SLR crossings use a pair of AXI Register Slices configured as fully registered on the source SLR and registered input on the destination SLR: This makes all SLLs buffered on both ends with no combinational logic in between, which gives the highest performance [43].

6 EXPERIMENTAL RESULTS

As discussed in Section 1, our MOMS increases DRAM bandwidth utilization by increasing reuse at two levels: (1) of *individual responses from the DRAM controller* by serving as many incoming requests as possible with the same data requested only once and (2) of *DRAM bursts and rows* by organizing requests to the DRAM controller in contiguous bursts.

In Sections 6.1 to 6.3, we first characterize the reuse mechanism (1) alone. Because it is largely independent from the properties of the DRAM controller and memory, we focus on a single system configuration, the PL system on ZC706. Indeed, on the ZC706, MSHRs and subentries compete for the same kind of resource as the cache: block RAM. Therefore, we can quantitatively explore the benefit of reallocating some of the resources normally allocated to the cache to implement instead more MSHRs and subentries, which are the new design points introduced by MOMSes. However, the AWS F1 system uses both BRAM and URAM for cache, MSHRs, and subentries, which makes it harder to visualize and quantitatively analyze the tradeoff. We complement the analysis by showing the advantages of cuckoo hashing for MSHR storage and linked-list architecture for subentries in Sections 6.2 and 6.3, respectively.

In Sections 6.4 to 6.7, we introduce the reuse mechanism (2) and evaluate it on the three memory systems presented in Section 5.2. In particular, in Section 6.4 we discuss the tradeoffs involved in choosing the maximum burst length and the advantages of using variable-length bursts instead of always requesting bursts of the maximum length. We then analyze the performance impact of adding bursts on MOMSes with different amounts of MSHRs and subentries (Section 6.5) and on

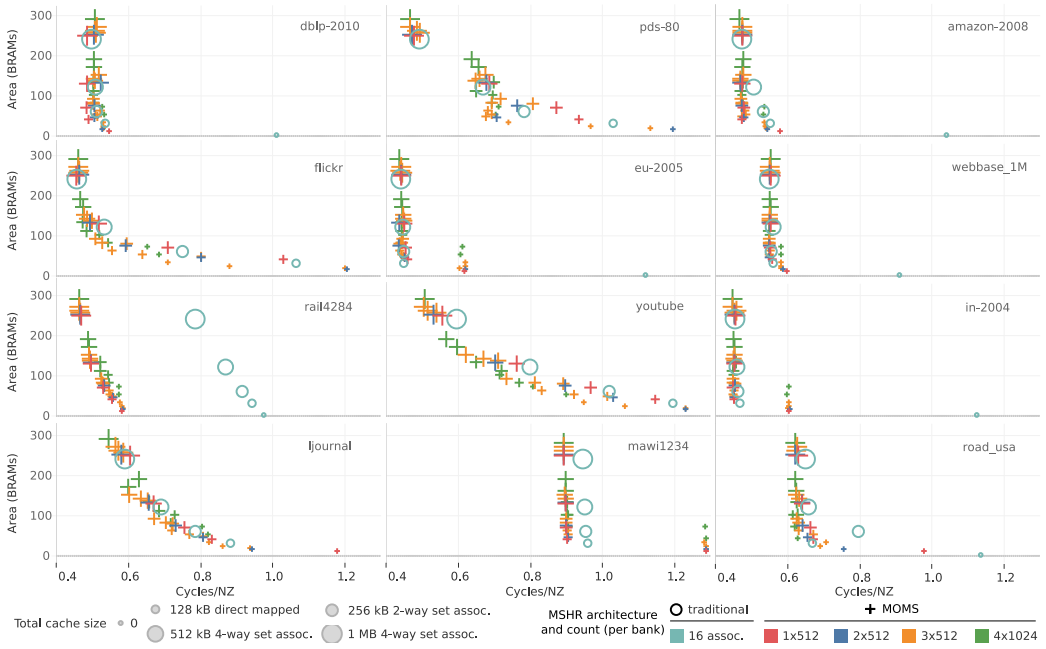

Fig. 15. Area of the memory system and normalized execution time for all benchmarks and a broad range of nonblocking cache architectures. For the MOMS architectures, we indicate the number and depth of cuckoo hash tables in each of the four banks, whereas the cache size refers to the entire multi-banked structure. Charts are sorted by increasing vector size and have been truncated at 1.3 cycles/NZ. On half of the benchmarks, all the Pareto-optimal designs are MOMSes, except for the smallest possible but low-performing design with no cache and associative MSHRs. For the other benchmarks, our MOMS provides additional Pareto-optimal designs, especially on the low area side.

specific benchmarks (Section 6.6) and present insights on how bursts improve (and, in a few cases, harm) performance (Section 6.7).

We conclude with Section 6.8 by discussing the area cost of MOMSes and the trends with respect to the number of MSHR and subentries and to the maximum burst length.

6.1 More Cache or More MSHRs?

We ran our benchmarks on a set of different traditional associative nonblocking caches and MOMS. We used 4-way set associative caches except in the smallest caches due to the limited minimum block RAM depth (see Section 6.8). For the traditional nonblocking caches, we only consider the best architecture that can run at 200 MHz, with 16 MSHRs with 8 subentries each. For MOMSes, we fixed the number of subentries per row to three, since, due to the finite choice of block RAM port widths, they occupy the same amount of block RAMs as two and provide a good compromise between utilization and stall cycles (see Section 6.3). We also fixed the stash size to two entries, which provides timing closure in all cases. We explored the number and depth of MSHR hash tables, as well as the depth of the subentry buffer.

Figure 15 summarizes the results. Our MOMSes provide the highest performance benefit to the benchmarks with the highest stack distance percentile (90% and 95%), i.e., the most challenging ones for caches. With rail4284, misses to multiple cache lines are so frequent that even the smallest MOMS with no cache at all performs 25% better than the largest traditional nonblocking cache,

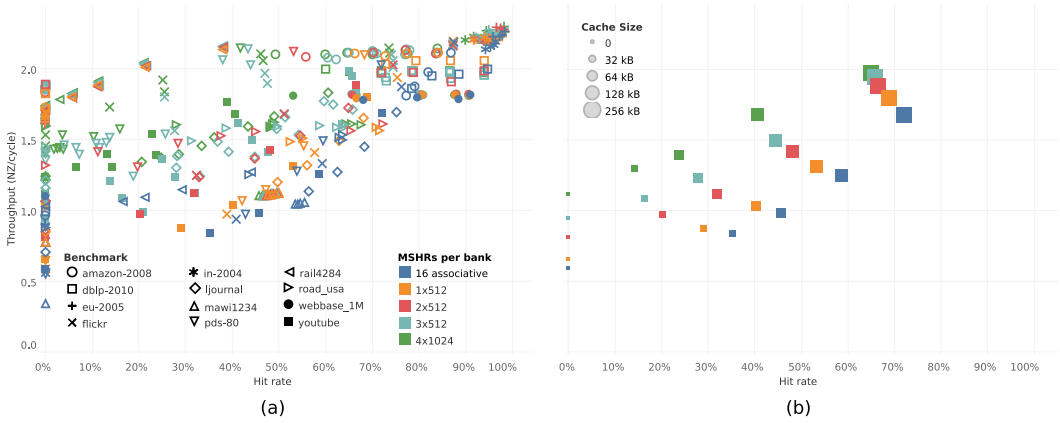

Fig. 16. Throughput as a function of cache hit rate, colored by number of MSHRs per bank, for all cache sizes (a) on all benchmarks and (b) highlighting a single benchmark (youtube). While traditional caches need high hit rates to achieve high throughputs, adding MSHRs and subentries shift the throughput/hit rate curves upwards and reduce their sensitivity on the cache hit rate, which improves performance especially at low cache hit rates.

which has a 24× larger area. On mawi1234, a small cache is enough to capture any existing temporal locality; after that, investing 2% of block RAMs for a single MSHR cuckoo hash table provides higher returns than any further increase in cache size. Pds-80, flickr, youtube, and ljournal offer a more gradual area-delay tradeoff and can benefit from the largest MSHR solutions, which constitute most of the Pareto-dominant points. On these benchmarks, we achieve 10% to 25% throughput increase with the same area or 35% to 60% area reduction at constant throughput. Dblp-2010, eu-2005, in-2004, and webbase_1M have higher locality and thus benefit more than other benchmarks from larger caches; however, the simplest MOMS with no cache, which uses 3× fewer BRAMs than the smallest cache, is enough to saturate the PS DRAM bandwidth only by merging memory requests. On eu-2005 and in-2004, the performance gain provided by the cache-less MOMSes is limited by handling the subentry linked lists. Applications with higher temporal locality may thus benefit from an increase of subentries per row. Benchmarks with few non-zero elements per row such as mawi1234 and road_usa have a lower maximum performance due to the higher bandwidth requirements for the sequential vectors; however, they are among the eight benchmarks that do not saturate the PS DRAM bandwidth without a MOMS.

Figure 16(a) shows the relation between cache hit rate and throughput (thus the inverse of the horizontal axis of Figure 15) as a function of the number of MSHRs per bank, for all benchmarks and cache sizes. In traditional caches, the throughput grows essentially linearly with the cache hit rate and only benchmarks that achieve high hit rate can reach peak throughput. The more MSHRs are added, the more the throughput increases at low cache hit rates, making the cache array less critical. This is particularly evident on benchmarks with gradual area-performance tradeoff in Figure 15 such as youtube, shown in Figure 16(b): Increasing the number of MSHRs shifts the throughput/hit rate curve upwards, increasing performance especially when the hit rate is low.

Generally, for a given benchmark and cache size, we observed no significant differences in DRAM and cache bandwidth utilization between traditional caches and MOMSes. In contrast, there is a significant difference in the average number of accelerator requests that are served with each memory response, which explains the vast majority of the performance differences that appear in Figure 15. This means that the main contribution of MOMSes to throughput is the increase of

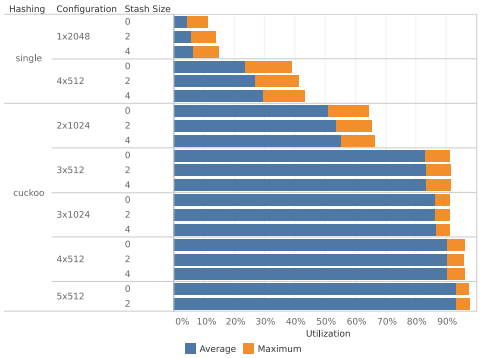

Fig. 17. Achievable MSHR storage load factor for several MSHR architectures on the ZC706 PL system. The 5 × 512 system with a 4-entry stash did not meet timing constraints. Single-hash architectures cannot utilize more than 40% of the storage space. Cuckoo hashing can handle collisions more efficiently and three hash tables are enough to achieve more than 80% average and 90% peak load factors, even without stash.

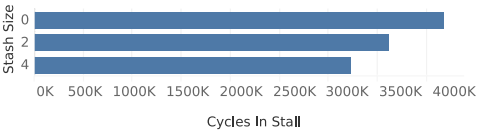

Fig. 18. Number of cycles lost due to stalls for collision resolution during the execution of a uniformly distributed benchmark. A 4-entry stash, which occupies less than 0.1% of LUTs and FFs, reduces the number of stall cycles by 30%.

the opportunities for data reuse: Indeed, the 64 MSHRs of the traditional cache (16 per bank) are generally enough to fill the memory request pipeline, whose latency is about 45 cycles.

There are only two scenarios where MOMSes have a visible impact also on the DRAM bandwidth utilization: on designs without cache array and on the rail4284 benchmark. Without cache array, even the smallest 1 × 512 MOMS has a 16% geometric mean and up to 44% higher DRAM bandwidth utilization compared to the traditional architecture. On rail4284, the DRAM bandwidth mismatch between MOMSes and traditional architectures remains within 41–48% even when the cache array is present, while it is negligible on all the other benchmarks. We believe that both cases are due to more frequent stalls resulting from one of the MSHRs running out of subentries in the traditional architecture. This reduces the MSHR utilization below the threshold required to fill the memory request pipeline. Except on rail4284, even a small cache array can take over most of the workload associated to cache lines with high locality, making it less likely for MSHRs to use all of their eight subentries. Cache-less MOMSes are affected much less by high-locality cache lines thanks to our dynamic subentry allocation.

6.2 Number of MSHR Hash Tables and Stash Size

Figure 17 analyzes the performance of the MSHR storage architectures described in Section 2.2. We focus on the PL system on ZC706 as trends on the other systems are similar. For each architecture, we measure average and peak utilization of the MSHR storage space. To make sure the benchmark always uses all of the available MSHRs, we use a synthetic 1M × 1M matrix with 5M uniformly distributed non-zero elements generated with the Python function `scipy.sparse.random(1e6, 1e6, 5e-6)`, no cache, and each bank contains 4,096 subentry rows with three subentries each. All architectures have 2048 MSHRs per bank or the closest possible value.

Because any collisions result in a stall that lasts until one of the colliding MSHRs is deallocated, all of the single-hash architectures achieve poor utilization: Even by introducing a stash to tolerate up to four collisions, a 4-way set-associative architecture does not go beyond 30% average

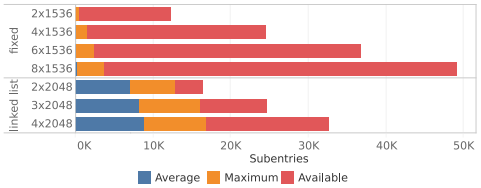

Fig. 19. Average and maximum subentry utilization during the execution of ljournal with a 3 × 512 cuckoo MSHR on the ZC706 PL system. Allocating a fixed number of subentries per MSHR results in less than 1% average utilization and resource waste. Linked-list architectures provide a more efficient usage of the subentry memory.

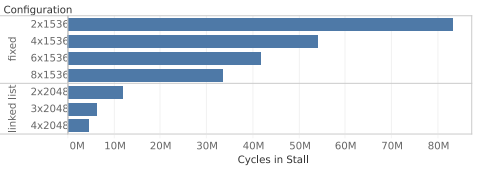

Fig. 20. Number of cycles lost due to subentry-related stalls. Stalls occur when (a) filling all subentries of an MSHR for the fixed architectures or (b) handling the linked list or running out of subentry rows for the linked list architectures. The smallest linked list architecture has three times fewer stall cycles than the largest fixed architecture despite having three times fewer subentries.

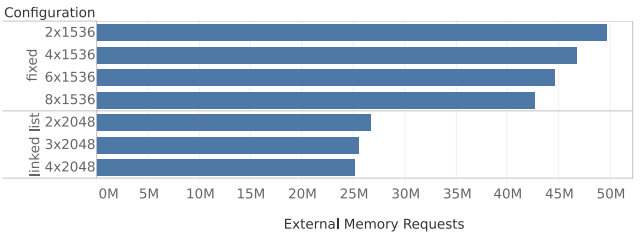

Fig. 21. Number of external memory requests during the execution of ljournal with a 3 × 512 cuckoo MSHR and no cache. By increasing subentry utilization, linked list architectures increase the number of accelerator requests that can be served by the same external memory request, resulting in a 37% decrease of external memory traffic.

and 45% peak load factors. Even a simple 2-way cuckoo hash table achieves 50% average and 70% peak utilization, and three ways enough to reach more than 80% average utilization, which is consistent with prior findings on cuckoo hashing [13]. Interestingly, using a 3-way 512-entry architecture (1,536 MHSRs) has higher absolute utilization than a 2-way, 1,024-entry organization (2,048 MSHRs). For three or more ways, adding a stash does not affect MSHR utilization but decreases the number of stall cycles by up to 30% with a 4-entry stash (Figure 18), which is the largest stash that we could implement within the 200 MHz constraint.

6.3 Subentry Organization

We performed a similar analysis for the memory organization of the subentries, as described in Section 3.2. We use the ljournal benchmark, which has a large number of secondary misses, and a MOMS with no cache and a 3 × 512 cuckoo MSHR buffer per bank on the PL system on ZC706. As shown in Figure 19, with a fixed number of subentries per MSHR, stalls are so frequent (Figure 20) that they prevent misses from accumulating in the buffers, resulting in very low utilization but also fewer opportunities for request merging and thus a higher traffic to external memory (Figure 21). We believe this problem is more pronounced in a MOMS-rich architecture than in a traditional nonblocking cache, because it is far more likely to encounter at least one MSHR that needs more than a given number of subentries when handling thousands of misses rather than a few tens of them. Our linked list-based architectures provide much higher average and maximum buffer

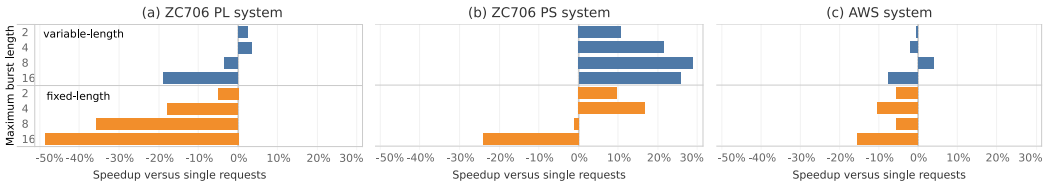

Fig. 22. Speedup obtained in MOMSes by sending bursts of memory requests compared to single-request MOMSes, both with dynamically adjusted burst bounds (cf. Section 3.4) and always requesting full bursts (geometric mean across all benchmarks and all configurations). Using bursts of a suitable size is beneficial to all systems and minimizing each burst's length is always better than using bursts of fixed length.

utilization, fewer stall cycles and decrease the number of DRAM memory requests by a factor 1.3× to 2×, with evident great energy impact.

6.4 Introducing Variable-Length Bursts

For all systems, we consider three configurations in terms of MSHRs and subentries. All configurations on the ZC706 have six subentries per row, while those on the AWS F1 have eight, since the two extra subentries can be implemented using the same amount of URAMs for the subentry buffer given the large width of URAM primitives.

The ZC706 PS systems use a 512-entry, 64-bit wide data buffer per bank. We considered (1) one, (2) two, and (3) four 512-entry MSHR cuckoo hash tables with (1) 512, (2) 1,024, and (3) 2,048 subentry rows per bank. To each configuration, we add 8, 16, 32, 64 kiB of cache per bank (4-way set associative, except for the 2-way 8 kiB), or no cache. Finally, variants with maximum burst length of (i) 2, (ii) 4, (iii) 8, and (iv) 16 beats are generated for each of those 15 architectures.

Similarly, the 60 ZC706 PL systems have (1) one 512-, (2) three 512-, and (3) four 1,024-entry MSHR cuckoo hash tables with (1) 512, (2) 2,048, and (3) 4,096 subentry rows per bank; 32–256 kiB of cache per bank or no cache, and the maximum burst lengths from 2 to 16. PL systems have a 32-entry, 512-bit wide data buffer per bank.

The AWS F1 systems are similar to the ZC706 PL systems as they both connect to 512-bit wide memory ports. The main differences arise from the use of URAM for the subentry buffer and the cache: Because of the larger minimum width and depth of URAM compared to BRAM ($72 \times 4{,}096$ vs 36×512), we only consider 256 kiB of cache per bank or no cache and subentry buffers always have 4,096 rows.

We will compare each of these architectures to alternative generic memory systems: (1) single-request MOMSes—with same amount of MSHRs, subentry rows, and cache and (2) a traditional nonblocking cache with 16 associatively-searched MSHRs, each with 8 subentries, with the closest BRAM utilization on ZC706 and with 256 kiB of cache per bank on AWS. Systems (2) are the same baselines used in Section 6.1 and contain the maximum number of MSHRs and subentries that ensure timing closure at 200 MHz (ZC706 PL) or 250 MHz (AWS F1) and that result in a FF utilization similar to the MOMS architectures (all systems; see Section 6.8).

Figure 22 shows the speedup of using bursts compared to restricting to single memory requests. Adjusting burst bounds is always useful, on all design points. On the ZC706 PS system, four beats of 64 bits (256 bits) corresponds to the PS DDR burst size (8×32 bits), which makes even fixed bursts of up to four beats beneficial compared to single requests which waste 75% of the burst content. Still, trimming bursts yields even higher speedups as contention among the memory controller ports is minimized. This effect does not appear on the ZC706 PL and AWS systems as single responses already consist of full DRAM bursts.

The maximum burst length controls the tradeoff between using larger parts of DRAM bursts/rows and wasting bandwidth due to requesting unnecessary data, either between pending misses on distant cache lines or due to frequent burst invalidations (cf. Figure 7 and 8). This tradeoff explains the bitonic speedup curve on both ZC706 systems and the single useful maximum burst length on the AWS system.

Overall, the ZC706 PS system gains the most from using bursts. Indeed, restricting to single 64-bit memory requests leaves few opportunities for reuse among 32-bit accelerator requests. The ZC706 PL systems benefit from bursts only through DRAM row conflict minimization, which still brings significant speedup on specific design points as discussed in Section 6.6. In addition to higher DRAM row utilization, the AWS memory system benefits from bursts through an increase of MLP, since single requests are not sufficient to fully pipeline memory accesses and can only exploit about 50% of the peak bandwidth. However, the larger available bandwidth per PE makes the bandwidth optimizations between MOMS and DRAM controller generally less critical than on the ZC706.

6.5 Impact of Bursts across the MOMS Design Space

We further analyze the architectures with the ideal maximum burst length for the respective system—4 for ZC706 PL systems and 8 for ZC706 PS and AWS systems, respectively. Figure 23 compares the throughput of traditional caches, single-request and burst MOMSes for different cache sizes and MSHR count.

If the memory controller has multiple narrow ports (ZC706 PS system), then repurposing some BRAMs from cache to MSHRs/subentries never pays off unless bursts are used. The speedup increases with the number of MSHRs and at four cuckoo hash tables becomes comparable to the single-request results on the PL system. Even there, bursts provide additional speedup on most of the architectures, especially where single-request architectures were the most useful. This includes the most lightweight system, whose baseline with the closest area has no cache at all, and on intermediate configurations with 3×512 MSHRs/bank and moderate cache size. On AWS, we cannot compare traditional caches at constant BRAM utilization, since both caches and MSHRs/subentries use both BRAM and URAM. Complementing caches with MOMSes is always useful compared to being limited to a few tens of associative MSHRs and there is no clear best architecture between single-request and burst MOMSes. Burst MOMSes have a significant advantage with no cache and few MSHRs, where bandwidth maximization is more critical and, for a fixed number of MSHRs, burst MSHRs can handle more cache lines than single-request ones. Single-request MOMSes appear to be more beneficial than burst MOMSes when paired to a cache, which suggests that more sporadic misses are better handled individually; however, they also seem to interfere more unpredictably with DMAs as the $4 \times 1{,}024$ single-request MOMSes starved accelerators of sequential data more often than 3×512, resulting in lower performance, which was not the case for the burst MOMSes. This phenomenon can only occur on AWS as on the ZC706 systems DMAs and MOMSes were connected to different memories.

6.6 Benchmark-related Trends due to Bursts

Figure 24 shows the throughput on individual benchmarks provided by the best-performing MOMS on each system. Small and/or regular benchmarks, characterized by high cache hit rate, benefit more from a larger cache than from more MSHRs, which is reasonable. Where caches are less effective, bursts make MOMSes useful also on the PS system, achieving up to 3.4× speedup. On the ZC706 PL system, burst architectures improve the performance of MOMSes on 10 benchmarks out of 15, in six cases by more than twice. The trend is confirmed by the absolute performance on the traditional nonblocking cache: The speedup is the highest on the benchmarks where the

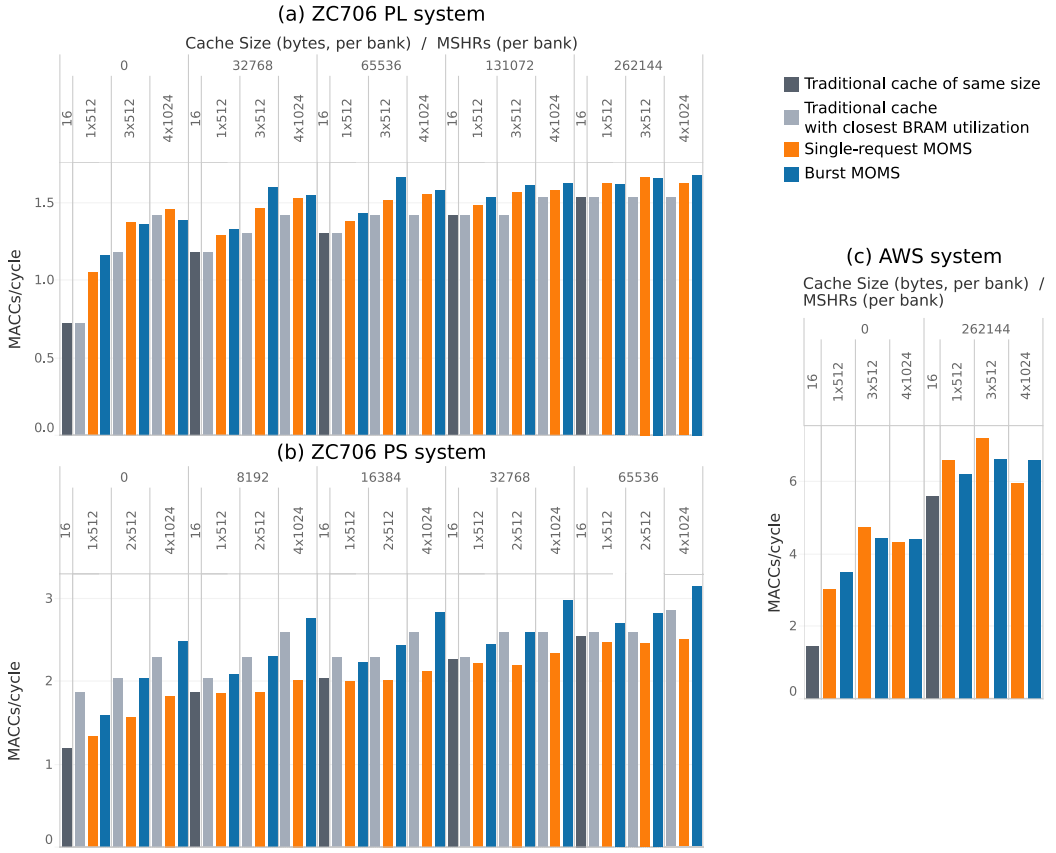

Fig. 23. Throughput, in geometric mean across all benchmarks, of a traditional nonblocking cache, a single-request MOMS and variable-length burst MOMS with maximum burst length of 4 for the ZC706 PL system and 8 for the ZC706 PS and AWS systems. For the ZC706 systems, without URAM, we additionally compare each MOMS to the traditional nonblocking cache with the closest BRAM utilization to analyze performance within a fixed area budget. Bursts are key enablers for MOMSes on memory controllers with multiple narrow ports (ZC706 PS) where single-request MOMSes performs worse than the traditional nonblocking cache with the closest area. On the ZC706 PL system, bursts bring further speedups to most data points where the single-request system was already reasonably effective. On AWS, single request MOMS generally perform better except with no cache and few MSHRs, where bandwidth is critical and burst MSHRs can handle more cache lines, or with cache and many MSHRs, where single request MOMSes have more interference with DMAs.

traditional nonblocking cache was performing worse. On AWS, the smallest cache-less burst MOMS (c1) generally performs better than the single-request one, by up to a factor 2.8×. On the same design point, the best of the two MOMSes—which uses at most 7% and 5% of the available BRAMs and URAMs, respectively—achieves 49% to 124% (average 72%) of the performance of the traditional cache that uses 2.3× more on-chip memory bits. When it includes a large cache (c2), the single-request MOMS generally handles the fewer misses better than the burst MOMS, which may introduce unnecessary cache lines that pollute the cache.

6.7 Analysis of Burst Usage

To better understand the mechanisms behind the improvement of memory access performance on most of the benchmarks and investigate the reasons for the slowdown on some benchmarks,

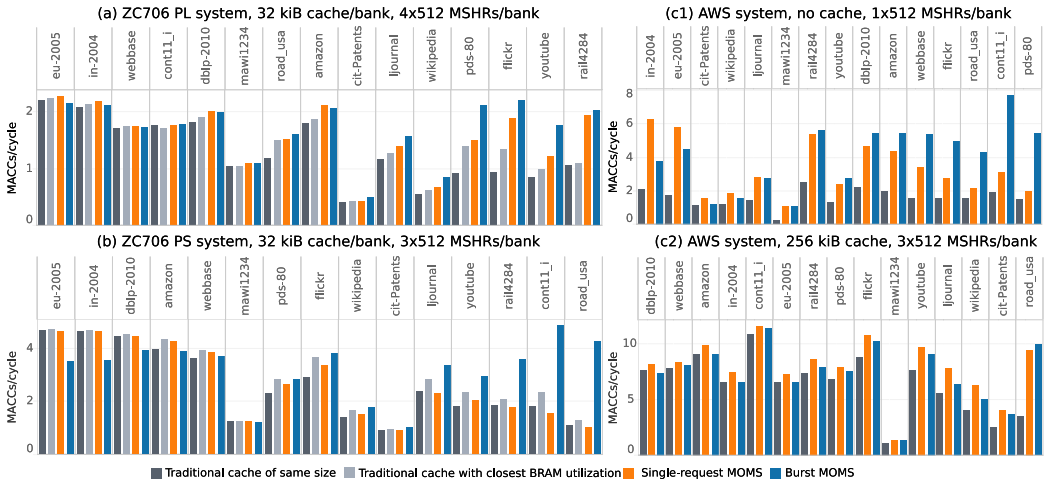

Fig. 24. Throughput of traditional nonblocking cache, single-request MOMS and burst MOMS on individual benchmarks for architectures where burst MOMSes have the largest speedup compared to traditional caches. Benchmark are sorted by increasing burst MOMS speedup compared to traditional caches. For AWS, we include an architecture where the single-request MOMS performs particularly well (c2). On ZC706, burst MOMSes are beneficial to most of the largest and/or irregular benchmarks, where traditional caches have the lowest performance. On AWS, burst MOMSes are particularly useful where memory bandwidth is more critical (c1); when the memory bottleneck is less evident, single-request MOMSes introduce less caches pollution and perform slightly better than both traditional caches and burst MOMSes (c2).

we simulated a bad and a good performing benchmark on the ZC706 PS and PL systems and analyzed how many of the cache lines requested from memory are actually used. More specifically, Figure 25 shows, for each burst length, how many of the requested cache lines have been actually used at least once and how many have been wasted, normalized by the total number of requested cache lines.

By construction, in bursts of two or more beats, at least two distinct cache lines will be always used. Invalidated bursts are completely discarded; hence, the bars corresponding to zero used cache lines count the number of cache lines wasted because of invalidations. Bursts of maximum length can never be invalidated. Data wastage in bursts where two or more cache lines have been used are instead due to requests hitting cache lines covered by the same MSHR but that are not consecutive.

In the well-performing benchmarks, a large share of useful data is retrieved through bursts of all lengths, which the memory controller can serve more efficiently than single requests, especially in the PS system. Indeed, even if the total share of wasted data is similar in both PS benchmarks, and higher than in the ZC706 PL system, the speedup provided by bursts is significantly higher in road_usa than in eu-2005.

Where using bursts is particularly beneficial, most of the bursts converge to their optimal length (according to the policy described in Section 3.4) by the time requests are sent to memory as invalidations are almost non-existing. Conversely, on the regular benchmarks, single requests are more dominant and bursts of maximum length are almost exclusively due to prior invalidations. In those cases, the cache already filters most of the memory accesses and the few remaining misses are better served by single-request architectures.

6.8 Resource Utilization

Table 4 shows the resource utilization of the entire system, with MOMS and baseline traditional nonblocking cache described in Section 6.1. We do not consider the case of a blocking cache,

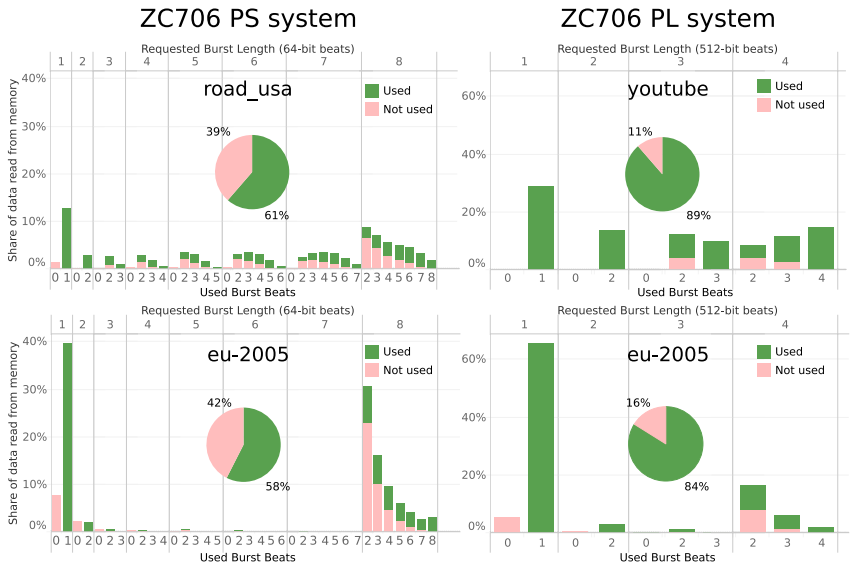

Fig. 25. Distributions of requested, used, and wasted cache lines per burst as a function of the burst length, normalized by the total number of cache lines requested from memory, for the same ZC706 systems evaluated in Figure 24. Pie charts: used and wasted cache lines, aggregated over all burst lengths. Top (bottom) row: benchmarks where introducing bursts is advantageous (harmful) compared to single-request systems. Benchmarks that get the highest speedup from bursts obtain a large share of useful data through bursts of all possible sizes. When the performance is poor, invalidations and single requests are more frequent.

because it performs significantly worse than the nonblocking cache for modest area savings. For MOMSes, we provide ranges that correspond to the configurations presented in Section 6.4.

Indicatively, the minimum cache that is worth implementing due to the minimum block RAM depth—a single 32 kB way (512 lines × 512 data bits)—has similar block RAM requirements as 3 × 512 MSHRs with 3 × 2048 subentries. In general, on the ZC706, the cache requires 8.5 block RAMs per 32 kB per cache way, the MSHR buffer requires 0.5 block RAMs per 512 MSHRs per cuckoo hash table for storage plus 0.5 block RAMs per 512 MSHRs for the request queue to the external memory arbiter, and the subentry buffer requires 1 block RAM per 512 subentry rows of up to 3 subentries each, plus 1 block RAM every 1,024 subentry rows for the FRQ. Each cuckoo hash function also uses 1 DSP block. On the AWS F1 FPGA, we used URAM for the cache data and the subentry buffer. While the FPGA has 4× more memory bits in URAMs than BRAMs, the larger minimum depth of each block, 4,096 entries instead of 512, increases the minimum size of the cache and subentry buffer that are worth implementing to 256 kiB and 4,096 rows, respectively, per bank. Such cache requires 8 URAMs and 4 BRAMs (for tag and valid arrays) instead of 68 BRAMs if URAMs were not available. As for the subentry buffer, 4,096 rows of eight subentries each require 3 URAMs for all the maximum burst lengths that we considered, instead of 25–29 BRAMs.

BRAMs and URAMs are the dominant resource in MOMSes and traditional caches and, especially the former, has large variations depending on the number of MSHRs, subentries, and cache size. On ZC706, cache-less MOMSes have up to 90–95% fewer BRAMs than traditional caches (in Section 6.1, we take the ZC706 PL system as representative to illustrate the trend more in detail); on AWS, 60% fewer URAMs. The FF utilization of MOMSes is comparable and generally slightly lower than that of the traditional cache, as FFs are repurposed from MSHR and subentry storage to, mostly, pipeline registers. The LUT utilization is 5–85% higher in MOMSes due to more

Table 4. Resource Utilization of MOMSes and Traditional Cache with 16 MSHRs with Eight Subentries Each per Bank, Compared to the Resource Utilization of the Rest of the Experimental System

	LUT	FF	DSP	BRAM
A) 4 accelerators	19k	23k	32	62
B) Fixed infrastructure	39k	37k	0	1
C) MOMS	23k–27k	30k–35k	4–16	12–363
D) Traditional cache	20k	30k	0	253
A + B + C	81k–85k	90k–95k	36–48	75–426
Utilization of **A + B + C**	37–39%	21–22%	4–5%	14–78%

(a) PL system on ZC706

	LUT	FF	DSP	BRAM
A) 8 accelerators	38k	51k	64	124
B) Fixed infrastructure	73k	71k	0	1
C) MOMS	20k–35k	26k–36k	8–32	24–322
D) Traditional cache	19k	36k	0	202
A + B + C	131k–146k	148k–158k	72–96	149–447
Utilization of **A + B + C**	60–67%	34–36%	8–11%	27–82%

(b) PS system on ZC706

	LUT	FF	DSP	BRAM	URAM
A) 16 accelerators	142k	140k	128	696	0
B) Fixed infrastructure	575k	694k	12	275	43
C) MOMS	104k–135k	151k–165k	16–64	152–314	48–176
D) Traditional cache	76k	155k	0	42	128
A + B + C	821k–852k	985k–999k	156–204	1123–1285	91–219
Utilization of **A + B + C**	69–72%	44–45%	2.3–3.0%	54–59%	5.0–23%
Bottom SLR	69–71%	42%	1.7–2.2%	55–59%	3.8–14%
Middle SLR	51–55%	36–37%	0.6–1.7%	25–33%	7.5–41%
Top SLR	89–91%	54%	4.5–5.0%	83–87%	3.8–14%

(c) AWS F1

Fixed infrastructure, which uses 50–70% of LUTs and FFs of the entire system, includes AXI SmartConnects, soft memory controllers and, in (c), AWS shell. For MOMSes, we report ranges corresponding to the configurations discussed in Section 6.4. Accelerators on the ZC706 systems use DMAs with 64-bit data ports as they are connected to 64-bit memory controllers (PL) or to be able to fit 8 accelerators (PS), while on the AWS system they use more resources as they are 512-bit wide as the memory controllers. On ZC706, the smallest MOMSes has very similar LUT and FF utilization than the traditional cache baseline while consuming 90–95% fewer BRAMs. On AWS, the cache-less architecture save 60% of the URAMs.

complex logic. Still, even the largest MOMS uses at most 16% of FPGA LUTs, which is dwarfed by the 30–50% of LUTs locked in fixed infrastructure. Overall, the LUT and FF utilization of MOMSes is comparable to that of accelerators. In addition to traditional caches, MOMSes use at most 5% of the available DSPs for cuckoo hashing. On AWS, despite multiple SLRs, we achieve timing closure

Table 5. Resource Utilization of the 16-bank MOMS on AWS F1 with 256 kiB Cache, 3×512 MSHR, 8×4096 Subentries Per Bank as a Function of the Maximum Burst Length

	LUT	FF	DSP	BRAM	URAM
1	109k	153k	48	218	176
2	124k	159k	48	242	176
4	125k	160k	48	250	176
8	129k	161k	48	250	176
16	132k	163k	48	250	176

The overhead of burst handling is mostly due to the logic shown in Figure 9 and to the burst offset bits in each MSHR and is within 21% for LUTs, 7% for FFs, and 15% for BRAM.

Table 6. Resource Utilization of the 16-bank MOMS on AWS F1 with 256 kiB Cache and 8×4096 Subentries Per Bank, Maximum Burst Length of 8, as a Function of the Number of MSHRs

	LUT	FF	DSP	BRAM	URAM
1×512	121k	157k	16	202	176
3×512	129k	161k	48	250	176
4×1024	132k	164k	64	314	176

The number of MSHRs mostly affects BRAM and DSP utilization, which are used for MSHR storage and cuckoo hashing, respectively. However, LUTs and FFs, used in the logic that handles MSHR lookup and update, change by at most 10% and 4%, respectively. Similar trends are observed on the ZC706 systems when also the number of subentries changes as they are also stored in BRAM.

at 250 MHz with around 70% LUT and 60% BRAM utilization across the entire device and 80–90% on the top SLR.

Table 5 shows the impact of maximum burst length on the resource utilization. The burst handling logic shown in Figure 9 has at most a 21% LUT and 7% FF overhead, while the minimum and maximum burst offset bits in each MSHR account for a worst-case 15% BRAM overhead. Most of the BRAM variability that appears in Table 4 is in fact due to the number of MSHRs and subentries, as illustrated in Table 6, in addition to the cache size.

7 RELATED WORK

We contrast our approach to prior research on miss handling architectures (Section 7.1), memory systems customized for irregular memory access patterns (Section 7.2) or that are automatically generated for a specific applications (Section 7.3), memory request reordering (Section 7.4), and coalescing (Section 7.5).

7.1 Miss Handling Architectures

The first non-blocking cache was proposed by Kroft in 1981 [26]. Farkas and Jouppi [12] evaluate a number of alternative MHA, including the explicitly-addressed MSHRs that inspired our MOMS. They observed that non-blocking caches can reduce the miss stall cycles per instruction by a factor 4 to 10 compared to blocking caches, that the most aggressive architectures (such as explicitly-addressed) are beneficial even for large cache sizes, and that overlapping as many misses as possible allows processors to maximize the benefit provided by non-blocking caches.

Tuck et al. [36] introduced a novel MHA for single processor cores with very large instruction windows. They propose a hierarchical MHA, with a small explicitly-addressed MSHR file for each L1 cache bank and a larger shared MSHR file. MSHRs are explicitly-addressed and shared MSHRs have more subentries than the dedicated ones. On a number of SPEC2000 benchmarks running on a 512-entry instruction window superscalar single-core processor, dedicated files with 16 MSHRs and 8 subentries and a shared file with 30 MSHRs and 32 subentries achieve speedups that are close to those provided by an unlimited MHA. However, we believe that a set of parallel accelerators is fundamentally different from a single-core processor even with a large instruction window for two reasons: (a) parallel accelerators with, for instance, decoupled access/execution architectures [8, 28] could generate even more requests per cycle with no fundamental limitations on the total number of in-flight operations, and (b) requests to be merged can come from the same as well as a different accelerator, so it is important to have a shared MHA to maximize the merging opportunities. Our results indeed showed that, for parallel accelerators with massive MLP, small caches with thousands of MSHRs can achieve similar or even better performance of larger caches with few MSHRs.

7.2 Memory Systems for Irregular Memory Accesses

Several pieces of work aimed at improving the efficiency of traditional caches on non-contiguous memory accesses. Impulse [6] introduces an additional address translation stage to remap data that is sparse in the physical/virtual memory space into contiguous locations in a shadow address space. However, it is a processor-centric system that relies on the intervention from the OS to manage the shadow address space. Traversal caches [35] optimize repeated accesses to pointer-based data structures on FPGA. Such approach is however limited to pointer-based data structures that are repeatedly accessed and that can fit entirely in the FPGA block RAM. Brugger et al. [5] present an ASIC-based accelerator for link assessment, a graph algorithm. Since the accesses to DRAM are irregular and short, they propose to reorganize the DRAM chips to expose a data width that is 8× narrower, which results in 8× lower data wastage with every memory access. We tackle the same problem by using each wide data block returned from DRAM to serve multiple incoming requests, which is directly applicable to commercial DRAM modules and controllers.

7.3 Automatic Generation of Application-Specific Memory Systems

Another line of work explored the automatic generation of application-specific memory systems. Bayliss et al. [4] proposed a methodology that automatically generates reuse buffers for affine loop nests, which reduce the amount of memory requests and of DRAM row conflicts. However, the approach is restricted to kernels consisting of an affine loop nest whose bounds are known at compile time. TraceBanking [45] does not rely on static compiler analysis and uses a memory trace to generate a banking scheme that is provably conflict-free. It also supports non-affine loop nests but requires the dataset to fit entirely in the block RAM. ConGen [22] focuses on optimizing DRAM accesses without relying on any local buffering on FPGA. It uses a memory trace to generate a mapping from the addresses generated by the application to DRAM addresses such that the

number of row conflicts is minimized. Cong et al. [10] restructure local buffers in HLS applications to make DRAM-BRAM memory transfers more efficient. EASY [9] uses an SMT solver to minimize the number of BRAM bank arbiters for multi-threaded HLS accelerators. All of these contributions rely on exact information about the application's memory access pattern at hardware compile time or need the entire dataset to fit in on-chip memory or at least to be processable in a tiled fashion. Our approach is application-agnostic, fully dynamic, does not require that the dataset accessed irregularly resides in local buffers and does not make any assumptions on the access pattern properties. MATCHUP [41] and LMC [44] use static analysis on HLS code and runtime profiling, respectively, to generate application-specific cache systems. Those generators could instantiate our architecture behind their caches to transparently boost read bandwidth towards external memory when the hit rate remains low and applications are latency-tolerant.

7.4 Request Reordering in Memory Controllers

All modern DRAM controllers implement some form of memory operation reordering such as the first-ready first-come first-serve policy that prioritizes row hits [32] or one of the many alternatives [17, 19, 30, 31, 33, 34, 37]. All these approaches must also minimize latency and thus rely on associative lookups over shallow request queues that provide only a local view of the memory accesses. We target throughput-oriented applications that can trade a few more cycles on the miss path for greater bandwidth through deeper request reordering.

7.5 Request Coalescing

Coalescing aims at increasing bandwidth utilization between datapath and DRAM by merging multiple narrow memory accesses into fewer, wider ones. Modern GPUs dynamically coalesce accesses from the same instruction executed by different threads [38] in the same warp, and the load-store units instantiated by the Intel FPGA OpenCL compiler can perform both static and dynamic burst coalescing [40]. To increase the opportunities for coalescing and thus the utilization of the bandwidth to the GPU L1 cache, Kloosterman et al. propose an inter-warp coalescer [25]. Wang et al. [39] proposed a dynamic coalescing unit for HMC memories in a multi-core system, implemented on a small RISC-V core. Incoming requests are stored in a binary tree and forwarded to the HMC after a timeout or after receiving 128 bytes of requests. All of these approaches have a very short window where coalescing can occur, at most a few requests wide. We showed that explicitly-addressed MSHRs also perform coalescing, on wider request windows and over multiple bursts at the same time (one per MSHR).

8 CONCLUSION

It is commonly assumed that some form of local buffering such as caching is the only way to optimize random accesses to external memory, which is testified by the vast effort in maximizing the hit rate under all possible scenarios. Nonblocking caches are one of the few architectures that attempt to at least mitigate the impact of misses: Notably, they reduce pipeline stalls and make a single cache line request to serve as many misses as possible, avoiding redundant requests. MOMSes scale up the concept to three orders of magnitude more misses, greatly increasing the average number of cache line reuses. We do so by efficiently mapping tens of thousands of MSHRs and subentries to the abundant FPGA on-chip RAM and by fully pipelining all stages of miss handling with minimal stalls. Besides scaling up the number of MSHRs and subentries, we additionally show how to extend MSHRs from handling a single to a few contiguous cache lines, which can be requested from memory as a burst. This requires minimal changes to the architecture and results in a higher memory bandwidth available to the MOMS as DRAM memories and controllers serve bursts more efficiently than single irregular requests. We evaluate our MOMS on an embedded and on a datacenter FPGA

system with three types of DDR3 and DDR4 memory controllers using 15 sparse matrix-vector multiplication benchmarks. Our findings suggest that large MSHR arrays can either be paired to a cache or replace the cache altogether, bringing the same advantages on read bandwidth as a (larger) cache but at a lower area cost. Handling bursts has a limited area overhead and results in significant speedups, especially behind DRAM controllers with multiple narrow ports, commonly found on SoC platforms, and on memory controllers that can be fully pipelined only using burst requests. Therefore, we believe MOMSes open up new opportunities to increase performance of bandwidth-bound, latency-insensitive applications with irregular memory access patterns.

REFERENCES

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. LEAP scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 25–28.

[2] Amazon.com, Inc. 2020. AWS Shell Interface Specification. Retrieved from https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md.

[3] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 781–792.

[4] Samuel Bayliss and George A. Constantinides. 2011. Application specific memory access, reuse and reordering for SDRAM. In *Proceedings of the 7th International Symposium on Applied Reconfigurable Computing*. 41–52.

[5] Christian Brugger, Valentin Grigorovici, Matthias Jung, Christian De Schryver, Christian Weis, Norbert Wehn, and Katharina Anna Zweig. 2017. A memory centric architecture of the link assessment algorithm in large graphs. *IEEE Des. Test* 35, 1 (2017), 7–15.

[6] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*. 70–79.

[7] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing*. 150–159.

[8] Tao Chen and G. Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. Taipei, Taiwan, 46.

[9] Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason H. Anderson, and George A. Constantinides. 2019. EASY: Efficient Arbiter SYnthesis from Multi-threaded Code. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 142–151.

[10] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2017. Bandwidth optimization through on-chip memory restructuring for HLS. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. IEEE, 1–6.

[11] Timothy A. Davis and Yifan Hu. 2011. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1.

[12] K. I. Farkas and N. P. Jouppi. 1994. Complexity/Performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. 211–222.

[13] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.* 38, 2 (2005), 229–248.

[14] Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware system synthesis from domain-specific languages. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*. Munich, 1–8.

[15] Intel Inc. 2016. *Hybrid Memory Cube Controller IP Core User Guide*. Intel Inc.

[16] Intel Inc. 2018. *Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*. Intel Inc.

[17] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture News*, Vol. 36. ACM, 39–50.

[18] Muhammad Irfan, Zahid Ullah, and Ray C. C. Cheung. 2019. Zi-CAM: A power and resource efficient binary content-addressable memory on FPGAs. *Electronics* 8, 5 (2019), 584.

[19] Bruce Jacob, Spencer Ng, and David Wang. 2010. *Memory Systems: Cache, DRAM, disk*. Morgan Kaufmann.

[20] JEDEC. 2012. DDR3 SDRAM Standard JESD79-3F. Retrieved from https://www.jedec.org/standards-documents/docs/jesd-79-3d.

[21] JEDEC. 2017. DDR4 SDRAM Standard JESD79-4B. Retrieved from https://www.jedec.org/standards-documents/docs/jesd79-4a.
[22] Matthias Jung, Deepak M. Mathew, Christian Weis, Norbert Wehn, Irene Heinrich, Marco V. Natale, and Sven O. Krumke. 2016. Congen: An application specific dram memory controller generator. In *Proceedings of the 2nd International Symposium on Memory Systems*. 257–267.
[23] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. SIAM.
[24] Adam Kirsch and Michael Mitzenmacher. 2007. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the 45th Annual Allerton Conference on Communication, Control, and Computing*, Vol. 75. 751–758.
[25] John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. 2015. WarpPool: Aharing requests with inter-warp coalescing for throughput processors. In *Proceedings of the 48th Annual International Symposium on Microarchitecture*. 433–444.
[26] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*. 81–87.
[27] Sheng Li, Ke Chen, Jay B. Brockman, and Norman P. Jouppi. 2011. *Performance Impacts of Non-blocking Caches in Out-of-order Processors*. HPL Tech Report.
[28] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. 2014. CGPA: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Design Automation Conference*. 1–6.
[29] Mario D. Marino and Kuan-Ching Li. 2017. System implications of LLC MSHRs in scalable memory systems. *Microprocess. Microsyst.* 52 (2017), 355–364.
[30] Sally A. McKee, Assaji Aluwihare, Benjamin H. Clark, Robert H. Klenke, Trevor C. Landon, Christopher W. Oliver, Maximo H. Salinas, Adam E. Szymkowiak, Kenneth L. Wright, William A. Wulf, et al. 1996. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the International Conference on Supercomputing*. 125–132.
[31] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Computer Architecture News*, Vol. 36. ACM, 63–74.
[32] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*. 128–138. https://doi.org/10.1109/ISCA.2000.854384
[33] Hemant G. Rotithor, Randy B. Osborne, and Nagi Aboulenein. 2006. Method and apparatus for out of order memory scheduling. US Patent US7127574.
[34] Jun Shao and Brian T. Davis. 2007. A burst scheduling access reordering mechanism. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*. 285–294.
[35] Greg Stitt, Gaurav Chaudhari, and James Coole. 2008. Traversal caches: A first step towards FPGA acceleration of pointer-based data structures. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis*. 61–66.
[36] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*. 409–422.
[37] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Arch. Code Optimiz.* 12, 4 (2016), 1–28.
[38] Vasily Volkov. 2016. *Understanding Latency Hiding on GPUs*. Ph.D. Dissertation. UC Berkeley.
[39] Xi Wang, John D. Leidel, and Yong Chen. 2016. Concurrent dynamic memory coalescing on GoblinCore-64 architecture. In *Proceedings of the 2nd International Symposium on Memory Systems*. 177–187.
[40] Felix Winterstein and George Constantinides. 2017. Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs. In *Proceedings of the International Conference on Field Programmable Technology*. 104–111.
[41] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. 2015. MATCHUP: memory abstractions for heap manipulating programs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 136–145.
[42] Philipp Woelfel. 1999. Efficient strongly universal and optimally universal hashing. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*. 262–272.
[43] Xilinx Inc. 2020. AXI Register Slice v2.1 (PG373).
[44] Hsin-Jung Yang, Kermin Fleming, Michael Adler, Felix Winterstein, and Joel Emer. 2016. LMC: Automatic resource-aware program-optimized memory partitioning. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 128–137.
[45] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. A new approach to automatic memory banking using trace-based address mining. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 179–188.

Received January 2021; revised April 2021; accepted May 2021