In Search of Lost Bandwidth: Extensive Reordering of DRAM Accesses on FPGA

Gabor Csordas, Mikhail Asiatici, and Paolo Ienne
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
gabor.csordas@epfl.ch, mikhail.asiatici@epfl.ch, paolo.ienne@epfl.ch

***Abstract*—For efficient acceleration on FPGA, it is essential for external memory to match the throughput of the processing pipelines. However, the usable DRAM bandwidth decreases significantly if the access pattern causes frequent row conflicts. Memory controllers reorder DRAM commands to minimize row conflicts; however, general-purpose controllers must also minimize latency, which limits the depth of the internal queues over which reordering can occur. For latency-insensitive applications with irregular access pattern, nonblocking caches that support thousands of in-flight misses (miss-optimized memory systems) improve bandwidth utilization by reusing the same memory response to serve as many incoming requests as possible. However, they do not improve the irregularity of the access pattern sent to the memory, meaning that row conflicts will still be an issue. Sending out bursts instead of single memory requests makes the access pattern more sequential; however, realistic implementations trade high throughput for some unnecessary data in the bursts, leading to bandwidth wastage that cancels out part of the gains from regularization. In this paper, we present an alternative approach to extend the scope of DRAM row conflict minimization beyond the possibilities of general-purpose DRAM controllers. We use the thousands of future memory requests that spontaneously accumulate inside the miss-optimized memory system to implement an efficient large-scale reordering mechanism. By reordering single requests instead of sending bursts, we regularize the memory access pattern in a way that increases bandwidth utilization without incurring in any data wastage. Our solution outperforms the baseline miss-optimized memory system by up to 81% and has better worst, average, and best performance than DynaBurst across 15 benchmarks and 30 architectures.**

I. Introduction

FPGAs are an attractive platform for accelerating applications that can easily be implemented with massively parallel data paths. However, this parallelism can be exploited only if the memory systems attached to the execution units can feed the pipelines with data fast enough. Whenever the memory access pattern is regular or predictable, techniques such as caching, application-specific buffering and prefetching are highly effective in ensuring that the accelerators retrieve most of their input data from high-bandwidth and low-latency local buffers instead of external memory (often DRAM). When the access pattern is irregular and frequent accesses to external memory are unavoidable, our prior miss-optimized nonblocking caches [1] have been shown to be effective to dynamically increase the utilization of the limited DRAM bandwidth. The key insight is to aggregate requests for words that are close to each other in the memory space, minimizing the number of memory requests. By monitoring windows of tens of thousands of outstanding misses, miss-optimized memory systems can harness spatial locality that would be out of reach to traditional caches of realistic size. However, the bandwidth of the DRAM depends heavily on the received access pattern, which a miss-optimized memory system does not optimize at all.

A. Limitations of Using DRAM as External Memory

DRAMs are organized in multiple banks (8 and 16 for DDR3 and DDR4 respectively), which are independent from each other and can operate in parallel. Each bank contains a two-dimensional array of cells, each storing one bit of data. To access a memory cell the memory controller has to open (*activate*) the corresponding row by copying it to the 1 KB row buffer. After the row is open, a column can be accessed in each clock cycle. A row conflict occurs when a different row is requested: in this case, the previous row has to be written back to memory (*precharge*) in order for the new one to be activated. The process is time consuming and degrades the bandwidth significantly. Maximizing the effective DRAM bandwidth involves (1) changing row as seldom as possible and, once a row is open, access it as much as possible and (2) exploiting bank parallelism to hide precharge and activation latencies as much as possible. General purpose memory controllers apply memory access reordering to reduce the row conflicts but, because they aim to keep access latency low as well, their employ shallow request queues that are searched associatively [2], [3], which limits the ability to reorder and minimize row conflicts.

B. State of the Art: Improving the Effective Bandwidth with Variable-Length Bursts

DynaBurst [4] represents a first attempt to reorder memory accesses in a miss-optimized architecture. Our original system groups incoming requests that can be served by the same, often wide, memory request (e.g., 512 bit) [1]. However, memory requests are sent out in no particular order, which may cause frequent row conflicts when accesses are irregular (frequent gaps in Fig.1a). By organizing incoming requests according to grouping regions spanning 4–8 memory requests, DynaBurst [4] sends bursts of (contiguous) memory requests instead. To minimize memory traffic, DynaBurst assembles all incoming

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00030

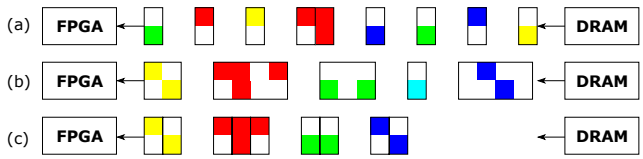

Fig. 1. Impact of different miss-optimized architectures on the memory access pattern and thus on DRAM bandwidth. In this example, the DRAM has one 64-bit port and the FPGA requests 32-bit words; colors identify the DRAM row that was read. The original miss-optimized system [1] requests single words without applying reordering (a), which may lead to frequent row conflicts that reduce the available bandwidth. DynaBurst [4] (b) sends variable-length bursts that cause fewer row conflicts but that may include data that is not needed or request some data twice (e.g., the cyan request). The proposed architecture (c) explicitly reorders individual requests, minimizing row conflicts without data wastage.

requests hitting a grouping region into a single burst that is as short as possible. As these bursts are aligned with the rows of DRAM, they significantly reduce the number of row conflicts. This only requires minor modifications to the miss-optimized architecture; however, it has two important limitations. Firstly, the burst may include inner data blocks that are not needed (e.g., red and green requests in Fig. 1b). Secondly, once the burst request has been sent to memory, its bounds cannot be updated any more; subsequent requests for words that are not covered by the in-flight burst trigger the invalidation of the current burst (i.e., its data will be discarded) and a new request for a burst covering the whole grouping region will be sent out (cyan and blue requests respectively in Fig. 1b). Due to both mechanisms, up to 40% of the data received from memory is wasted [4].

C. This Paper: Extensive Reordering of Memory Accesses

Miss-optimized memory systems accommodate thousands of requests from throughput-oriented accelerators in order to maximize the number of incoming requests that can be served by the same memory response, and they do so without artificially increasing latency [4]. To process a large window of incoming requests without stalling, a deep queue is used to buffer the outgoing memory requests as they are generated. Miss-optimized systems, so far, focused on gaining visibility of thousands of incoming requests; however, the output queue also gives them access to thousands of future memory requests. In this paper, we use this information to minimize DRAM row conflicts by explicitly reordering individual memory requests across a window that is three orders of magnitude larger than that of a typical DRAM controller downstream. Similarly to the way incoming requests are grouped based on the respective cache line [1] or burst [4], we group memory requests based on their DRAM bank and row, and memory accesses belonging to the same group will be sent out contiguously to the DRAM controller. Doing so reduces DRAM row conflicts without incurring in the data wastage associated with bursts [4] (Fig. 1c).

II. Miss-Optimized Nonblocking Cache

Our original miss-optimized memory system [1] is shown in Fig. 2. The accelerators are connected to multiple banks

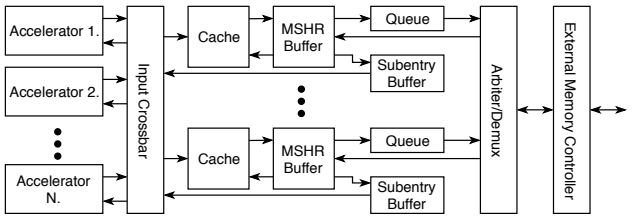

Fig. 2. Top-level architecture of the miss-optimized memory system [1]. Variable number of accelerators send their requests to multiple cache banks capable of storing thousands of in-flight misses. The output of the memory system is connected to the external memory through a wide data bus (e.g. 512 bits).

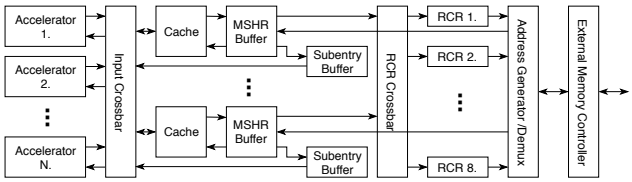

Fig. 3. Top-level architecture of the proposed extension of the miss-optimized memory system. The RCRs are inserted between the MSHR Buffers and the External Memory Controller. Each of them is assigned to a bank of the DRAM. The RCR Crossbar steers between the cache banks and the RCR banks. The responses bypass the RCRs and go directly to the MSHR Buffers.

by the input crossbar which steers requests according to some bits of the address. Cache hits will be immediately served just as in a multi-banked cache. On the first miss to a given cache line (primary miss), a new memory request is placed in the bank output queue, its tag is stored into an miss status holding register (MSHR), whereas the offset of the requested word is stored into one of the MSHR's subentries. An arbiter handles memory requests from all banks and forwards them to the DRAM. Requests to cache lines whose tag is not in the cache but for which an MSHR has been already allocated (secondary misses) can be served by a pre-existing memory request and only need their word offset to be stored in a subentry. When the cache line returns from the DRAM, its tag is used to retrieve the respective MSHR and subentries in order to serve all its pending misses. Conventional nonblocking caches store a few tens of MSHRs into fully associative memories and each MSHR is attributed a fixed number of subentry slots; the cache stalls whenever it runs out of MSHRs or any MSHR runs out of subentry slots. Our miss-optimized memory system is a nonblocking cache with some key implementation differences:

- Instead of using fully-associative CAMs that map poorly to FPGA resources, MSHRs are stored into cuckoo hash tables in BRAM. This improves the critical path and increases the MSHR capacity from tens to tens of thousands while maintaining constant time lookup and deletion; a stash [5] is used to minimize the impact of collisions.
- Instead of statically allocating a fixed number of subentries to each MSHR, subentries are stored in a separate buffer and are dynamically allocated to MSHRs. Subentries are organized in blocks (rows) that are allocated at once. Each

MSHR contains a pointer to its first subentry row inside the subentry buffer; more rows can be appended as a linked list, if needed. This strongly reduces the occurrence of subentry-related stalls, now limited to the case where no subentry rows are available.

As the number of outstanding misses increases, so does the probability for misses to be secondary rather than primary, meaning that more and more incoming requests can be served without generating extra memory requests.

III. Where to Reorder?

As mentioned in Section I-A, in order to maximize the effective DRAM bandwidth we must 1) minimize row changes within each bank and 2) exploit the fact that banks can operate in parallel. Both requirements can be satisfied if 1) once a row has been opened, we send as many requests (column accesses) to that row as possible and 2) afterwards, instead of opening a new row in the same bank, we move to another bank. The latter allows the memory controller to at least overlap precharge of the previous row and activation of the following row in the other bank; modern reordering DDR controllers may even anticipate the next row activation to completely hide its latency [6].

To achieve these goals, we extend the original miss-optimized memory system [1] as shown in Fig. 3. The Row Conflict Reducers (RCRs) are responsible for the memory access reordering. Since the miss-optimized memory system was already returning responses out-of-order [1], further reordering of accesses on the output of nonblocking cache can be applied without any impact on the functionality of the system. Each DRAM bank, being completely independent from the others, has a corresponding RCR that organizes its requests. A crossbar is inserted between the MSHR buffers and the RCRs as the number of cache banks, as well as the policy that is used to partition requests among them, are not necessary the same as for DRAM banks. To maximize bank parallelism, the address generator picks groups of requests that target a given row in round-robin order from each RCR.

IV. Row Conflict Reducer

The goal of the Row Conflict Reducer is to group the incoming requests by row address. The architecture of the RCR can be found in Fig. 4 and consists of three main modules:

- Row Address Buffer,
- Column Address Buffer, and
- Row Address Deallocation Queue.

The logic between accelerators and RCRs interprets the request address according to Fig. 5a. and sends only the tag to the RCRs. The way the tag address is decomposed into DRAM bank, row, and column address (Fig. 5b.) depends on the addressing scheme used by the DRAM controller; we assume here that it contains the entire row and bank addresses, and part of the column address as the lowest significant bits, often with the highest entropy, are usually assigned to parts of the column address [7].

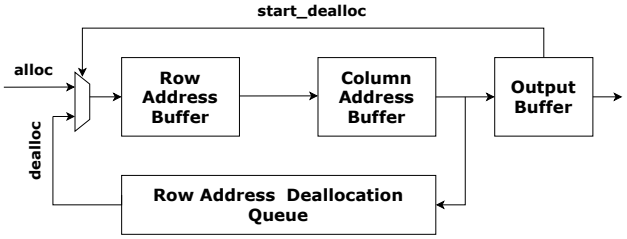

Fig. 4. Row Conflict Reducers responsible for reordering accesses to make the access patter more regular. Incoming tags are decomposed into row and column addresses; the row is searched and, if needed, allocated into the Row Address Buffer. Each entry in the Row Address Buffer includes a pointer to a region into the Column Address Buffer that contain all column addresses that have been received for that row. The Deallocation Queue contains a copy of all received row addresses in FIFO order. Whenever the output buffer is free, it triggers the deallocation and readback of the oldest row and all the respective received column addresses.

The bank address is used by the crossbar to determine which RCR bank will handle the request and is thus implicit inside each RCR. Inside each RCR, memory requests are handled very similarly to the way incoming requests are processed by the miss handling logic, with row and column address analogous to cache line tag/MSHR and offset/subentry. When an RCR receives a request, it first searches the corresponding row address inside the row address buffer. If it is the first request to a given row, an entry is allocated in the Row Address Buffer for the row address, together with some storage for the column addresses inside the Column Address Buffer. The column address of the request is stored into the Column Address Buffer and finally the row address is placed into the Row Address Deallocation Queue which will be used to easily iterate over existing Row Address Buffer entries. If the row address of an incoming request has already been stored in the Row Address Buffer, only a column address allocation for the respective row occurs.

Two important differences between miss handling logic and RCRs are the way 1) outputs are sent out and 2) entry deallocation occurs. The output of the miss handling logic does not need any information from the subentries, which are used only after the memory response returns to assemble the responses to individual pending misses; for this reason, both MSHR and subentries cannot be deallocated until the arrival of the memory response. The output of an RCR, instead, needs information from both the row and column address buffers as the whole incoming address must be forwarded to memory; moreover, both row and column addresses can be immediately deallocated as soon as the respective memory requests have been sent out.

For the reason outlined above, when the output buffer is empty, the Row Address Deallocation Queue issues the deallocation of the oldest row address. The deallocation triggers retrieval and deletion of the respective Row Address Buffer and Column Address Buffer entries, which are forwarded to the external memory controller through the output buffer. As long as the rate of production of memory requests by the miss

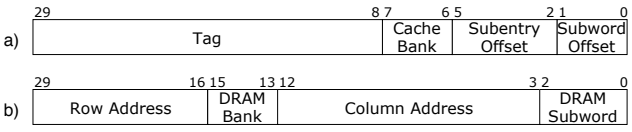

Fig. 5. Decomposition of memory addresses by the miss-handling logic (a) and DDR memory (b). The MSHR decomposes the address as the tag of the cache block, the address of the cache bank, the subentry offset (the word offset in the cache block) and the subword offset (a); and the external memory controller as row address, bank address, column address and DRAM subword offset (b). The numbers refer, without loss of generality, to the PL system described in Section V, where accelerators request 32-bit words, the memory system implements 4 cache banks, the external memory controller has a 512-bit wide bus and uses bank interleaving address scheme [7], the DRAM has 8 banks (e.g. DDR3), each bank returns 64-bit words and 10 bits are used as column address. Because memory requests are 512-bit wide, the six least significant bits do not need to be stored into the RCRs as they are are always zero.

handling logic is higher than the rate of consumption of rows by the memory controller, rows will accumulate inside the Row Address Deallocation Queue, extending the lifetime of rows inside the Row Address Buffer and thus the time window during which they can collect column accesses. This is analogous to the conditions that ensure that incoming requests naturally accumulate inside the miss handling logic without having to forcefully increase their latency [4].

A. Row Address Buffer and Deallocation Queue

Just like the MSHR buffer, the Row Address Buffer must support scalable content-addressable lookup and deletion in constant time, as well as fast insertion. Therefore, the Row Address Buffer is also implemented as a cuckoo hash table with stash [1] which stores row addresses together with a pointer to an entry in the column address buffer (analogous to the MSHR's pointer to the subentry buffer). The Row Address Deallocation Queue stores a copy of the row addresses and is implemented as a circular buffer. To be able to utilize the entire Row Address Buffer, the size of the Row Address Queue matches that of the address buffer.

B. Column Address Buffer

The Column Address Buffer stores the requested column addresses grouped by their row address. We could, in principle, store the column addresses in linked lists just like the analogous subentry buffer stores the subentries. The subentry buffer used linked lists because there is no theoretical upper bound for the number of allocated subentries per MSHR as the same word can be requested multiple times. Linked lists provide a good compromise between area efficiency and flexibility but their efficient handling requires complex control logic [1].

However, since only primary misses trigger the generation of a memory request, and because the miss handling logic guarantees that there is only one primary miss per cache line tag (as described in Section II), the RCR is guaranteed not to receive the same tag multiple times. This means that the maximum number of DRAM column addresses associated to a given DRAM row can never be larger than the number of all the possible column addresses in a row. Moreover, since each possible column address in a row can only have two states (requested or not), the column information associated to each row is a simple bit mask where each bit is set to 1 only if the corresponding column has been requested. This greatly simplifies the update logic which only needs to either set a bit or clear all of them when the column mask is initialized.

benchmark	vector size (MB)	rows (M)	non-zero elements (M)	sparsity ($\times 10^{-6}$)	stack distance 75%	stack distance 95%
amazon-2008	2.81	0.735	5.16	9.5	6	19.3k
cit-Patents	14.4	3.78	16.5	1.2	91.1k	151k
cont11_i	7.48	1.47	5.38	1.9	2	3
dblp-2010	1.24	0.326	1.62	15	2	4.68k
eu-2005	3.29	0.863	19.2	26	5	69
flickr	3.13	0.821	9.84	15	3.29k	14.5k
in-2004	5.28	1.38	16.9	8.8	0	11
ljournal	20.5	5.36	79.0	2.7	19.3k	184k
mawi1234	70.8	18.6	38.0	0.1	20.9k	609k
pds-80	1.66	0.129	0.928	16	26.3k	26.6k
rail4284	4.18	0.004	11.3	2400	0	35.4k
road_usa	91.4	23.9	57.7	0.1	31	158k
webbase_1M	3.81	1.00	3.10	3.1	2	323
wikipedia-20061104	12.0	3.15	39.4	4.2	47.3k	137k
youtube	4.33	1.13	5.97	4.6	5.8k	32.6k

TABLE I
SUITESPARSE MATRICES USED IN OUR EXPERIMENTS [8]. HIGH STACK DISTANCE PERCENTILES CORRESPOND TO POORER TEMPORAL LOCALITY DURING SPMV [1], [9]. VECTORS ARE OF SINGLE-PRECISION FLOATING POINTS NUMBERS.

C. Output Buffer and Address Generator

The output buffer inside each RCR triggers the row address deallocations when it is empty. It takes a row address and a column bit mask and forwards them together to the Address Generator (Fig. 3). The Address Generator contains a round-robin arbiter and the logic to reconstruct the original sequence of tags by iterating over the ones in the column mask. The tags are then forwarded to the DRAM controller. This ensures that the access pattern at the output of the Address Generator complies with the requirements discussed in Section III.

V. EVALUATION

We used Chisel 3 to implement the modifications outlined in Section IV in the original miss-optimized memory system code [1] . We used Vivado 2017.4 and targeted a Xilinx ZC706 board, which contains an xc7z045 FPGA, 1 GB of DDR3 on the programmable logic side (PL), and 1 GB of DDR3 on the processor system side (PS) [10].

Being DDR3, both memories have eight banks: the PL memory is a Micron MT8JTF12864HZ-1G6G1 with 64-bit datapath, while the PS memory is a Micron MT41J256M8HX-15E with 32-bit datapath. The PL memory controller is a Xilinx MIG [11], which exposes a single, 512-bit wide port that sends full DDR3 8-beat bursts. We measured a peak bandwidth of 12.0 GB/s with burst sequential accesses at 200 MHz. The PS memory uses the ARM memory controller, which exposes it to the FPGA through five 64-bit ports [3], although our

measurements showed that the four HP ports running at 150 MHz are enough to saturate its 3.9 GB/s bandwidth. Both memory controllers use bank interleaving [7] and implement memory access reordering [3], [11]. On the PL controller, requests targeting the same bank and row must be within eight or less requests apart in order to inhibit row precharge and activation [11]. The PS controller, instead, picks requests from a 32-entry CAM "to maximize DDR memory access efficiency" and keeps rows open until another row from the same bank is required [3].

Our natural baselines being our original miss-optimized system [1] and its burst-based extension, DynaBurst, [4], we evaluated our approach using the same sparse matrix-vector (SpMV) accelerators and matrices from SuiteSparse, both freely available [1], [8]. The accelerators [1] multiply a CSR-encoded sparse matrix with a dense vector; single-precision floating point matrix values, as well as their column indices and row pointers (32-bit integers) are streamed using Xilinx AXI DMA IPs and the column indices are used to access the dense vector. Therefore, the three CSR vectors are accessed sequentially while the column indices define the access pattern on the dense vector. Such accesses are performed through our memory system and behind an 8192-entry reorder buffer. Just like the baselines, our approach is also fully generic and application-agnostic, which explains the absence of SpMV-specific optimizations. We used benchmarks from very different application domains (including linear programming, social, web, and transport networks) to evaluate our system on a large variety of access patterns. Their properties are summarized in Table I.

As in our original memory system [1], we used the PS memory to host the CSR vectors accessed sequentially and perform the irregular accesses on the PL memory. The PL memory, despite being the highest performing memory available, is often the system bottleneck when no miss-optimized architectures are used, followed by the sequential accesses to the PS memory, whose bandwidth can be saturated by four accelerators running at 200 MHz [1]. To match the accelerators throughput, we implemented four banks in our memory system and ran the entire design at 200 MHz. We also tested the dual configuration, where the irregular accesses are performed on the PS memory instead, where DynaBurst excels [4]. With that configuration, however, the proposed system performs worse than both baselines on most of the data points. This suggests that an access pattern where accesses to the same row are sent consecutively, normally beneficial to the PL controller (as shown in the following subsections), is instead pathological for the PS controller. Since we could not identify a way to explicitly expose row hit opportunities through the access pattern sent to the PS controller, we will not discuss this configuration further and focus on the optimization of the irregular accesses on the PL memory instead. Ultimately, this will not be an obstacle once a designer has full control on the memory controller.

We used the following MSHR and subentry configurations (per bank) to evaluate our architecture: (1) *small (S)*: one 512-entry MSHR hash table and 512 subentry rows, (2) *medium (M)*: three 512-entry hash tables and 2,048 subentry rows, and (3) *large (L)*: four 1,024-entry hash tables and 4,096 subentry rows. Subentry rows have 6 subentries each. Each of these configurations was paired to five different cache sizes: 0 KB, 32 KB, 64 KB, 128 KB, and 256 KB (per bank), resulting in 15 configurations that, we think, representatively span every parameter. Each of them has been compared to an original miss-optimized system [1] and a DynaBurst system with maximum burst length of 4 (which was shown to provide the best performance [4]) with the same configuration.

We implemented each of the configurations described above using two different RCR variations. The smaller system uses Row Address Buffers with two 64-entry hash tables, whereas the larger RCRs use two 512-entry hash tables.

A. Global Speedup and Impact of RCR Size

Fig. 6 shows the speedup provided by the proposed architecture for all the configurations (geometric mean across all benchmarks). The speedup tends to be higher on smaller caches as the number of memory accesses is larger and the system is more sensitive to the memory performance. However, the RCRs achieve positive geometric mean speedup on all configurations, compared not only to the original miss-optimized system [1] but also to DynaBurst [4].

On systems (S), the number of outstanding memory requests is always limited by the small number of MSHRs rather than the RCR slots. Therefore, there are no significant differences in performance between the two RCR architectures. Conversely, systems (M) and (L) have enough MSHRs to saturate the 2×64 RCRs; with the larger RCRs, they can accept more requests before stalling and they can reorder the memory accesses more efficiently, resulting in greater speedups.

B. Memory Bandwidth Utilization

To validate the hypothesis that the RCRs provide speedup by increasing the available DRAM bandwidth, we analyzed usage and availability of the memory interface, with and without 2×512 RCR. On every cycle during the execution of a benchmark, the memory address (input) channel will be either (1) idle (valid is low: no request is being sent to memory), (2) active (valid and ready are both high: a new request is being accepted), or (3) busy (valid is high but ready is low: the memory is applying backpressure). Ideally, the channel will always be either idle or active according to the system's needs (depending on benchmark, cache hit rate, and fraction of secondary misses). However, as the fraction of idle cycles decreases, busy cycles also appear due to DRAM refresh overlapping non-idle cycles and, especially, row conflicts. In practice, busy cycles represent unavailable DRAM bandwidth.

Fig. 7 shows the fraction of active cycles as a function of the fraction of non-idle cycles (active and busy) during the execution of each benchmark, on each configuration—in other words, it shows the obtained bandwidth as a function of requested bandwidth. Without RCRs, the obtained bandwidth essentially matches the requested one as long as it is below 10% of the peak; beyond this point, it depends on the access

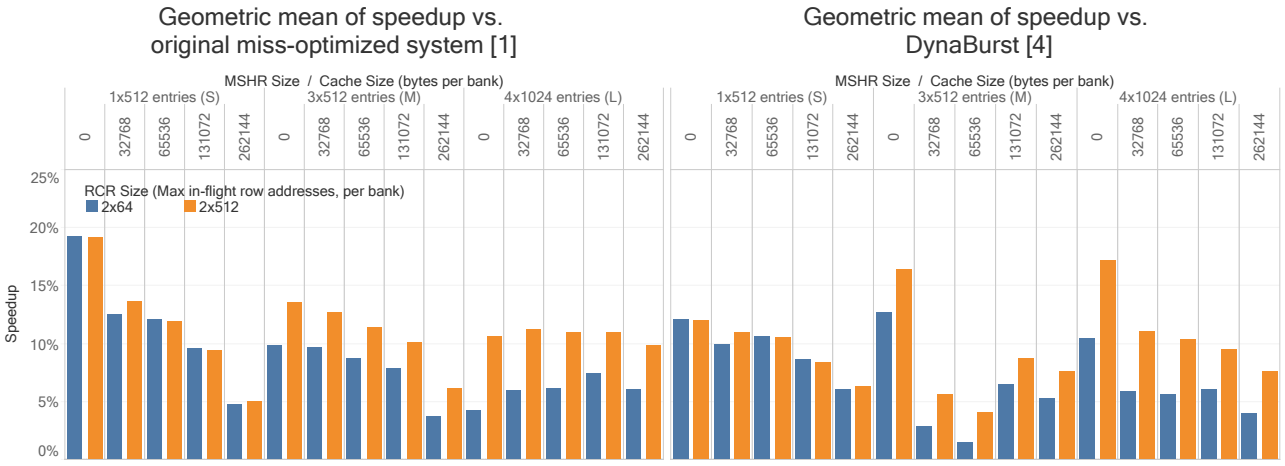

Fig. 6. Geometric mean of the speedup of the proposed memory system using the smaller and the larger RCRs across all the benchmarks, compared to the original miss-optimized system [1] and to DynaBurst [4]. RCRs provide positive geomean speedup on all architectures and compared to both baselines. They are especially useful when paired to smaller caches or no cache as the performance becomes more sensitive to the memory bandwidth. Systems (M) and (L) have enough MSHRs to saturate the smallest RCRs and thus need the larger RCRs to achieve their full potential.

Fig. 7. Fraction of requested and obtained DDR bandwidth during the execution of all benchmarks on all configurations, clustered by miss-optimized architecture. The circle size is proportional to the speedup with respect to the original miss-optimized system [1]. Backpressure from the DDR controller, a symptom of row conflicts, severely limits DDR bandwidth availability to the original system [1]. Both bursts and RCRs provide their highest speedups to data points that need the largest bandwidth. DynaBurst [4] has higher bandwidth utilization but wastes part of it internally. RCRs increase bandwidth utilization without incurring in data wastage.

pattern but it is never greater than 55% and can be as low as 38% of the requested one. The RCRs shift the majority of the points upwards, increasing the obtained bandwidth especially to the benchmarks that need it the most: close to 100% requested bandwidth, where the RCRs provide the larger speedups, the obtained bandwidth range shifts from 38–55% to 50–68%. Bursts have a similar benefit and can be even better than RCRs at bandwidth saturation: however, Fig. 7 does not capture data wastage which, on our same board and DDR memory, has been shown to affect up to 16% of the received data [4]. This explains the average positive speedup achieved by the RCRs even compared to DynaBurst [4].

C. Speedup on Individual Benchmarks

To further investigate the relationship between benchmark properties, requested bandwidth, and RCR performance, we explore the speedup of the proposed solution on individual benchmarks. We analyze an architecture where RCRs have the highest (Fig. 8) and lowest (Fig. 9) mean speedup with respect to DynaBurst [4]. Even on individual benchmarks, the larger RCR has generally neutral or positive impact, depending on how often the smaller RCR gets saturated. Compared to the original miss-optimized system [1], there is a clear separation between benchmarks that have low requested bandwidth, where the RCRs practically have no impact, and those that use most of the available bandwidth, where speedups are consistently positive (with the larger RCR). Considering all architectures, the speedup can be as high as 81% (cit-Pate on system (L) with 256 kB of cache per bank).

The trend versus DynaBurst [4] is more complex: on a few bandwidth-hungry benchmarks, bursts are slightly more efficient than RCRs at increasing performance. As shown in Fig. 7, bursts can be more efficient than RCRs at increasing bandwidth; moreover, bursts that contain more data than necessary may act as a prefetching mechanism when coupled with a cache as in Fig. 9. However, data wastage seems to be more common than useful phenomena as most of the bandwidth-bound benchmarks benefit more from RCRs than from bursts, sometimes by a significant margin. On the other end of the spectrum, most regular benchmarks are insensitive to both RCRs and bursts as the original baseline [1] already provides

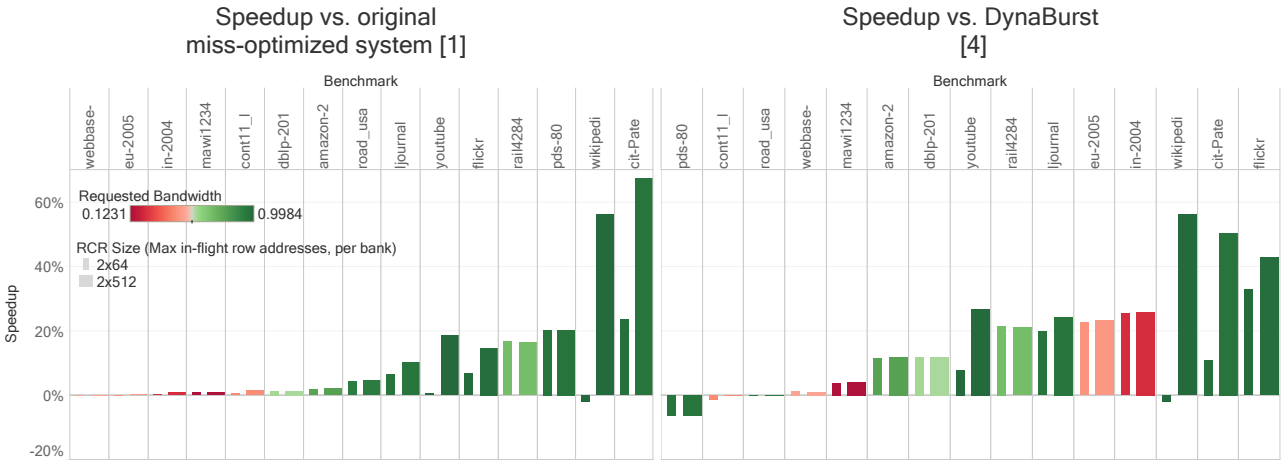

Fig. 8. Speedup provided by the RCRs on individual benchmarks, on a configuration favourable to our system (system (L) with no cache). The proposed solution never slows down the original miss-optimized system [1]; moreover, it outperforms bursts on most of the memory-bound benchmarks and on two regular benchmarks where bursts, due to data wastage, have a net negative impact on performance.

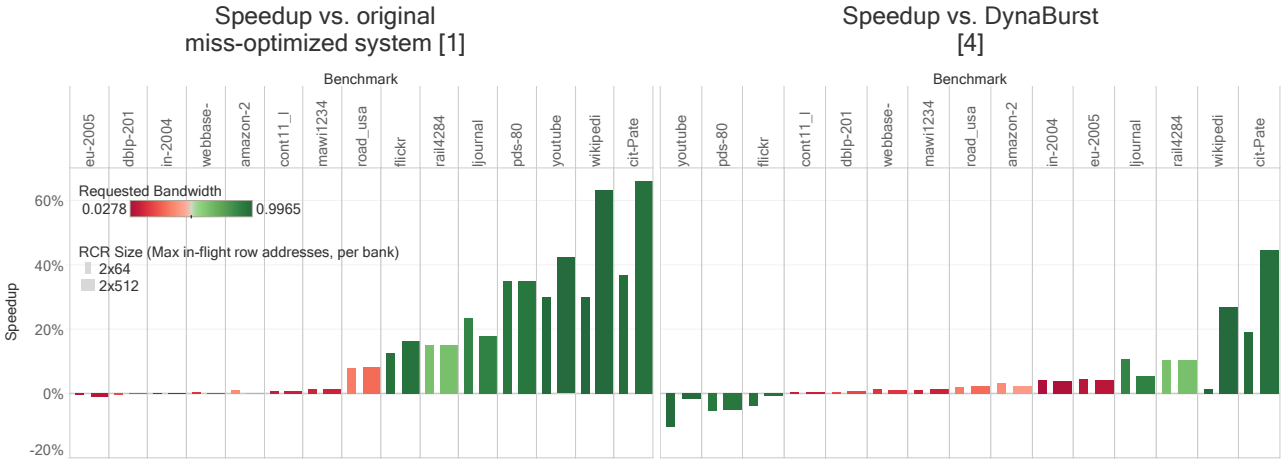

Fig. 9. Speedup provided by the RCRs on individual benchmarks, on a configuration where our architecture is moderately competitive to DynaBurst [4] (system (M) with 64 kB of cache per bank). RCRs still have neutral or positive impact compared to the original system [1]; while bursts provide slightly better performance on three bandwidth-intensive benchmarks, the 2×512 RCRs achieve higher performance on 12 benchmarks out of 15 (up to 45% improvement).

them with enough bandwidth. However, in the case of eu-2005 and in-2004 in Fig. 8, data wastage is such that bursts [4] have a net negative impact on performance. The RCRs, while not providing speedup in some cases, have at least no negative impact on performance—in other words, RCRs offer better worst case performance than bursts [4].

D. Resource Utilization

Table II shows the resource utilization of the systems that have been tested (with the largest cache size). LUT and flip-flop overhead ranges between +13% and +56% and is due to the RCR crossbar and the eight RCRs logic, which is almost a replica of the MSHR and subentry buffers one but for twice as many RCRs than banks. BRAM overhead is much smaller and even negative for systems (3). This is because the MSHR output queue in the baselines, which can contain up to one tag per MSHR, is replaced by row and column address buffers which, through the bit mask (see Section IV-B), encodes the same information more efficiently. Finally, because each cuckoo hash function requires a DSP, the two hash tables in each of the eight RCRs require 16 additional DSPs. Considering that miss-optimized systems tend to be BRAM-limited [1], the RCR overheads mostly affect resources that are less critical. For example, on our target FPGA, system (L) with 256 kB of cache (the largest we tested) has 63% BRAM utilization but only 16%, 10%, and 3.6% LUT, FF, and DSP utilization respectively despite the overheads.

VI. Related Work

Several memory system specialized for irregular access patterns have been proposed in the past. The Stream Memory Controller [12] relies on the detection of streaming computations at compile-time to send memory requests in an optimal order at runtime. The Impulse controller [13] defines a shadow

Config.	(S)		(M)		(L)	
RCR	**2×64**	**2×512**	**2×64**	**2×512**	**2×64**	**2×512**
LUT						
count ($\times 10^3$)	29.0	32.2	30.2	33.3	31.7	35.0
vs [1]	+30%	+45%	+36%	+51%	+41%	+56%
vs [4]	+23%	+37%	+18%	+30%	+20%	+33%
FF						
count ($\times 10^3$)	39.5	40.3	40.6	41.4	41.2	41.9
vs [1]	+37%	+40%	+36%	+39%	+35%	+38%
vs [4]	+16%	+18%	+14%	+16%	+13%	+15%
BRAM						
count	272.5	276.5	300.5	304.5	340.5	344.5
vs [1]	+4%	+6%	+2%	+3%	-3%	-2%
vs [4]	+3%	+5%	+1%	+2%	-5%	-4%
DSP						
count	20	20	28	28	32	32
vs [1]	+16	+16	+16	+16	+16	+16
vs [4]	+16	+16	+16	+16	+16	+16

TABLE II
AREA UTILIZATION OF THE PROPOSED SOLUTION AND OVERHEAD COMPARED TO BASELINES [1], [4] (MEMORY SYSTEM ALONE, NOT COUNTING ACCELERATORS AND GLUE LOGIC [1]). LUT, FF, AND DSP OVERHEADS ARE DUE TO THE RCR CROSSBAR AND LOGIC, WHILE THE BRAM OVERHEAD FOR ROW AND COLUMN ADDRESS BUFFERS IS ESSENTIALLY COMPENSATED BY THE ELIMINATION OF THE MSHR OUTPUT QUEUE. BASED ON THE RESOURCE BLEND OF A TYPICAL FPGA, THE OVERHEADS ARE CONCENTRATED ON THE LEAST CRITICAL RESOURCES.

memory space to perform application-specific remappings to better utilize DRAM-CPU bandwidth and caches when accesses in the physical address space would be strided or irregular. Both approaches require interventions from the compiler and/or the operating system and are not readily applicable to spatial computing. ConGen [14] gains global information about the access pattern through the full memory trace, which is used to generate offline a custom address mapping that minimizes row conflicts. The window seen by our dynamic approach, while not truly global, is still orders of magnitude larger than that of general-purpose memory controllers and does not require the entire access pattern to be available at compile-time. DReAM [15] analyzes toggling rates of address bits at runtime and adjusts address mapping on-the-fly if needed. When this happens, the data in DRAM must be migrated, which involves heavy performance overheads. We use a fixed address mapping, bank interleaving [7], and reorder memory operations instead, with minimal area overhead for request bookkeeping.

The idea of dynamically reordering DRAM operations in a generic memory controller to maximize performance was firstly proposed in 2000 by Rixner et al. [6]. Their first-ready, first-come, first-serve policy (FR-FCFS) prioritizes ready operations such as column accesses to an active row (row hits). FR-FCFS is now a de-facto standard implemented by most DRAM controllers [7], including the Xilinx MIG used in our experiments [11]. A plethora of other scheduling policies have been proposed; for instance, they try to balance aggregated throughput and fairness in a multi-processor [16] or heterogeneous systems [17] or to maximize bandwidth under real-time latency guarantees [18]. The general-purpose scenarios targeted by those policies must optimize bandwidth together with latency and fairness, which limits queues depth and aggressiveness of reordering. Our solution is application-agnostic but targets throughput-oriented systems. As such, the RCRs implement a very aggressive row hit-first policy over the thousands of requests naturally exposed by the miss-optimized memory system.

The automatic generation of application-specific memory systems based on on-chip memory represents another active area of research. Several methodologies focused on generating efficient multi-banked buffers for local reuse or to accelerate DRAM-BRAM data transfers within the context of high-level synthesis [19]–[22]. Other works target specific classes of applications such as those involving stencil operations [23], [24] or sparse matrix-vector multiplication [25]–[27]. By leveraging the huge bandwidth gap between external and local memory (tens of GB/s versus TB/s), these approaches can achieve absolute performance levels that are beyond our reach, but only on specific applications, when the dataset is small enough to entirely fit in on-chip memory, or when its processing can be performed in a tiled fashion. We target applications that do not fit within these constraints, as well as scenarios where the design effort for the development of a custom memory system would be unacceptably high, and try to get as much bandwidth as possible out of the external memory.

VII. CONCLUSION

For decades, compute engines have consistently outperformed memory in throughput, and FPGAs are no exception. To make things worse, the DRAM bandwidth decreases even further when applications perform irregular memory accesses, hindering most of the benefits of hardware acceleration. Common solutions involve shifting as many accesses as possible from DRAM to on-chip memory through caches or application-specific memory systems. Miss-optimized memory systems take a different approach and strive to use the limited DRAM bandwidth as efficiently as possible, which is beneficial when frequent accesses to DRAM are unavoidable. While they maximize reuse of individual memory responses, they do not tackle the reduction in available memory bandwidth caused by frequent DRAM row conflicts due to the irregular access pattern. Sending bursts of memory requests mitigates the problem but incurs data wastage, which reduces or may even cancel out any advantage. In this paper, we show how memory requests generated by miss-optimized memory systems can be efficiently reordered to reduce DRAM row conflicts without requesting any unnecessary data. This is similar to what modern DRAM controllers do, but on three orders of magnitude more requests. This maximizes the opportunities for DRAM row reuse and therefore the available bandwidth, extending the advantage of using miss-optimized memory systems to support throughput-oriented parallel accelerators with irregular memory access pattern.

REFERENCES

[1] M. Asiatici and P. Ienne, "Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs," in

Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Feb. 2019, pp. 310–319.

[2] S. Bayliss and G. A. Constantinides, "Application specific memory access, reuse and reordering for SDRAM," in *Proceedings of the 7th International Symposium on Applied Reconfigurable Computing*, Belfast, Mar. 2011, pp. 41–52.

[3] Xilinx Inc., *Zynq-7000 SoC Technical Reference Manual (UG585)*, Jul. 2018.

[4] M. Asiatici and P. Ienne, "DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses," in *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications*, Sep. 2019.

[5] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," in *European Symposium on Algorithms*, Karlsruhe, Sep. 2008, pp. 611–622.

[6] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens, "Memory access scheduling," in *27th International Symposium on Computer Architecture (ISCA 2000), June 10-14, 2000, Vancouver, BC, Canada*, 2000, pp. 128–138.

[7] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[8] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[9] C. Cascaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th annual international conference on Supercomputing*, Phoenix, Az., Nov. 2003, pp. 150–159.

[10] Xilinx Inc., *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC (UG954)*, Jul. 2018.

[11] ——, *7 Series FPGAs Memory Interface Solutions (UG586)*, Mar. 2011.

[12] S. A. McKee, A. Aluwihare, B. H. Clark, R. H. Klenke, T. C. Landon, C. W. Oliver, M. H. Salinas, A. E. Szymkowiak, K. L. Wright, W. A. Wulf *et al.*, "Design and evaluation of dynamic access ordering hardware," in *International Conference on Supercomputing*, 1996, pp. 125–132.

[13] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, "The Impulse memory controller," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1117–1132, 2001.

[14] M. Jung, D. M. Mathew, C. Weis, N. Wehn, I. Heinrich, M. V. Natale, and S. O. Krumke, "ConGen: An application specific DRAM memory controller generator," in *Proceedings of the 2nd International Symposium on Memory Systems*, Alexandria, Va., Oct. 2016, pp. 257–267.

[15] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "DReAM: Dynamic re-arrangement of address mapping to improve the performance of DRAMs," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 362–373.

[16] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, Ga., Dec. 2010.

[17] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 65, 2016.

[18] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *Proceedings of the Design, Automation and Test in Europe*, 2013, pp. 525–530.

[19] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "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*, Monterey, Calif., Feb. 2017, pp. 179–188.

[20] J. Cheng, S. T. Fleming, Y. T. Chen, J. H. Anderson, and G. A. Constantinides, "EASY: Efficient Arbiter SYnthesis from Multi-threaded Code," in *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2019, pp. 142–151.

[21] Y. T. Chen and J. H. Anderson, "Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software," in *Proceedings of the 27th International Conference on Field-Programmable Logic and Applications*, Ghent, Belgium, Sep. 2017, pp. 1–8.

[22] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for HLS," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[23] W. Li, F. Yang, H. Zhu, X. Zeng, and D. Zhou, "An efficient data reuse strategy for multi-pattern data access," in *Proceedings of the International Conference on Computer Aided Design*, 2018, p. 118.

[24] J. Escobedo and M. Lin, "Graph-theoretically optimal memory banking for stencil-based computing kernels," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 199–208.

[25] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *Proceedings of the 22nd IEEE Symposium on Field-Programmable Custom Computing Machines*, Boston, Mass., May 2014, pp. 36–43.

[26] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2014, pp. 161–170.

[27] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43th Annual International Symposium on Computer Architecture*, Seoul, Jun. 2016, pp. 243–254.