2019 29th International Conference on Field Programmable Logic and Applications (FPL)

DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses

Mikhail Asiatici
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
mikhail.asiatici@epfl.ch

Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
paolo.ienne@epfl.ch

***Abstract*—The effective bandwidth of the FPGA external memory, usually DRAM, is extremely sensitive to the access pattern. Nonblocking caches that handle thousands of outstanding misses (miss-optimized memory systems) can dynamically improve bandwidth utilization whenever memory accesses are irregular and application-specific optimizations are not available or are too costly in terms of design time. However, they require a memory controller with wide data ports on the FPGA side and cannot fully take advantage of the memory interfaces with multiple narrow ports that are common on SoC FPGAs. Moreover, as their scope is limited to single memory requests, the access pattern they generate may cause frequent DRAM row conflicts, which further reduce DRAM bandwidth. In this paper, we propose DynaBurst, an extension of miss-optimized memory systems that generates variable-length bursts to the memory controller. By making memory accesses locally more sequential, we minimize the number of DRAM row conflicts, and by adapting the burst length on a per-request basis we minimize bandwidth wastage. On a multiple, narrow-ported DDR3 controller, we provide 28% geometric mean and up to 3.4× speedup compared to a traditional nonblocking cache of the same area, while the prior single-request approach would not have been cost-effective. On a controller with a single, wide port, we can further improve the performance of miss-optimized systems by up to 2.4×.**

I. Introduction

FPGAs provide acceleration by implementing massively parallel, application-specific compute engines. However, such an approach breaks down if the performance is limited by the throughput of the memory system rather than that of the datapath—i.e., when the application is memory bandwidth-bound. Applications that perform irregular and narrow memory accesses, such as graph analytics and sparse linear algebra, are especially prone to be bandwidth-bound if accesses are performed directly on DRAM. This happens, for example, whenever the footprint of irregular accesses is too large to be efficiently cached by on-chip SRAM. Usually, problems admit application-specific optimizations, but they are expensive in design time and hard to integrate with accelerators generated from high-level synthesis unless the access pattern is perfectly known at compile-time [1], [2].

A. DRAM Performance Under Irregular Access Patterns

Under irregular access patterns, two mechanisms significantly degrade the DRAM bandwidth, as suggested in Fig. 1a. Firstly, both DDR3 and DDR4 operate on bursts of eight beats, normally of 64 bits each [3], [4]: if accesses are narrower than the burst size (which is common for scalar operands), the remaining data returned from memory will be discarded, wasting memory bandwidth and energy. 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 copying the respective row to the *row buffer* (typically 1 KB [3], [4]), which has to be written back to memory before accessing another row in the same bank. Since both operations are time-consuming, the actual bandwidth decreases when accesses require switching row frequently (row conflicts). DRAM controllers reorder memory requests to reduce the number of DRAM row conflicts [5]; however, general-purpose controllers must also minimize latency, which means that the internal request queues are relatively shallow and the optimization is possible only for accesses close in time.

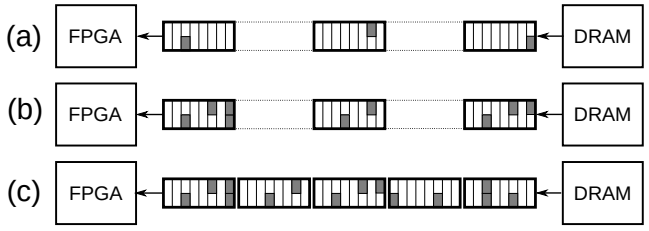

Fig. 1. Total availability and utilization of DRAM bandwidth under short irregular access patterns. White boxes: eight-beat bursts transferred from DRAM to the FPGA. Gray: portions of data actually used by an accelerator on the FPGA. Dashed lines: cycles where no transfers occur due to a DRAM row conflicts. If requests from the accelerators are forwarded directly to the DRAM controller (a), most of the burst beats will be wasted and frequent row conflicts make the interface frequently idle. Miss-optimized memory systems (b) improve the utilization of each bursts but do not reduce row conflicts. Our proposed architecture (c) combines burst reuse with row conflict minimization, which further increase the effective bandwidth accessible to the accelerators.

B. Reorder and Reuse is the Key

Our previous work [6] showed that miss-optimized memory systems represents a general, dynamic solution to boost performance of latency-insensitive, bandwidth-bound applications that read data irregularly. The key idea is to reuse the same wide memory response to serve multiple narrow requests from

1946-1488/19/$31.00 ©2019 IEEE
DOI 10.1109/FPL.2019.00049

the accelerators (Fig. 1b) on-the-fly, without relying on long-term storage in cache. This improves bandwidth utilization and is equivalent to reordering narrow requests such that those that hit on the same wide memory response 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. However, the approach relies on memory controllers exposing the DRAM through a single wide interface and is not directly applicable to controllers with multiple narrow ports, common on FPGA SoC platforms [7], [8]. Moreover, the approach operates only at the granularity of single memory requests: such requests are sent to the memory controller in an arbitrary order, with no special care given to minimize DRAM row conflicts.

C. Increasing the Scope for Dynamic Memory Access Optimization

In this paper, we propose DynaBurst, which builds on top of our miss-optimized memory system to handle bursts of variable length on the memory side. When possible, we make bursts longer and exploit more of a DRAM row without being limited to the controller width; else, when spatial locality is insufficient, we keep burst short and minimize contention in the controller. Without loss of generality, we used a set of sparse matrix-vector multiplication benchmarks to evaluate the performance of our system under vastly different access patterns that can occur in realistic latency-insensitive, bandwidth-bound applications. We will show that supporting bursts is required for miss-optimized memory systems to be beneficial behind external memory interfaces with multiple narrow ports and can further boost read throughput when behind a single wide memory port.

II. Miss-Optimized Memory Systems

Nonblocking caches can handle a given number of outstanding misses without stalling. On the first miss to a given cache line `C` (*primary miss*), the cache 1) allocates a new *miss status holding register* (*MSHR*) which stores the address of `C`, 2) allocates a *subentry* within that MSHR to store the offset of the request within `C`, and 3) sends out a memory request for `C`. Misses on a cache line that has already been requested (*secondary misses*) only require an additional subentry on the respective MSHR. All pending misses are served when the cache line returns from memory. Increasing the number of outstanding misses that can be tolerated has two benefits: (1) by reducing stalls memory-level parallelism (MLP) increases and (2) by having more in-flight cache lines, misses will have more chance to be secondary rather than primary, which increases the average number of requests that will be served with the same data returned from memory. In practice, both factors increase memory bandwidth utilization, especially when the cache hit rate is low.

A miss-optimized or MSHR-rich memory system is a multi-banked nonblocking cache that handles thousands of outstanding misses—two orders of magnitude more than conventional nonblocking caches [6]. It does so by (1) storing MSHRs in cuckoo hash tables in BRAM instead of fully associative register files and (2) dynamically allocating subentries depending on the needs of each MSHR, instead of statically allocating a fixed number of subentries per MSHR and stalling whenever any MSHR runs out of subentries. When applications are latency-insensitive and have a low hit rate, repurposing some BRAMs to MSHRs and subentries often increases throughput [6].

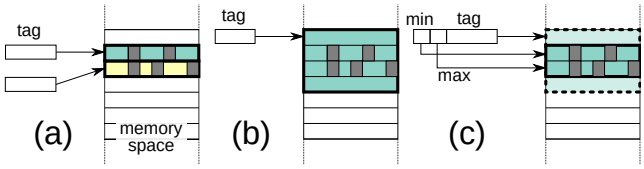

Fig. 2. MSHR memory range and structure. Portions of cache lines that have been requested by some accelerators are shown in gray. MSHRs usually refer to single cache lines (a). Increasing the memory range covered by each MSHR to a set of cache lines that will be requested as a burst (b) reduces DRAM 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.

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

In nonblocking caches, including miss-optimized memory systems [6], MSHRs have the granularity of single cache lines (Fig. 2a). Since cache lines are handled fully independently from each other, there are no guarantees that cache lines that are close in the address space, 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, 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 (Fig. 2b). Burst transfers can then be used to request such cache lines efficiently. However, any cache line within the burst that is not actually needed will cause bandwidth and energy wastage. As we show in Section VIII-A, 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 (Fig. 2c). In particular, each MSHR collects misses to 2^N cache lines, which corresponds to the maximum burst length. Two additional 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.

IV. Dynamically Adjusting Burst Bounds

On a primary miss, a new MSHR is allocated and a memory request is inserted in the output queue; its burst initially

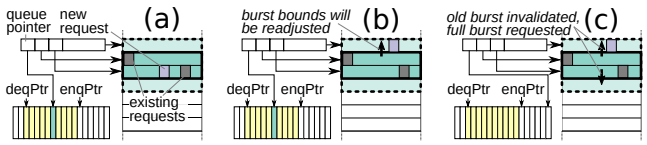

Fig. 3. Burst update policies. Requests that fall within the current burst range (a) do not require any updates; otherwise, burst bounds can be updated if the burst memory request is still in the output queue (b). If the memory request has already left the queue (c), the current burst is invalidated and a new burst of maximum length is requested.

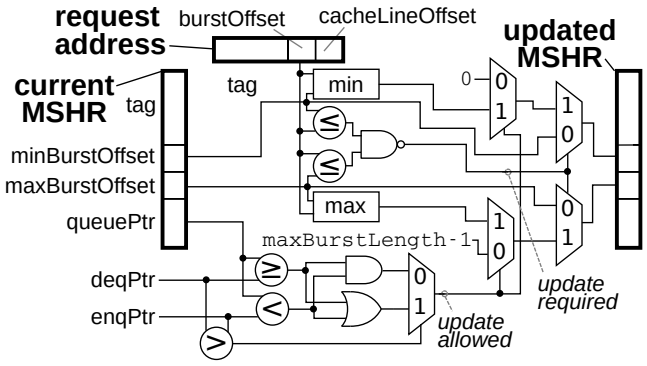

Fig. 4. Burst bounds update circuit. The updated MSHR on the right overwrites the current MSHR on the left in the following cycle; tag and queuePtr are never modified after MSHR allocation. Considering that realistic burst offsets and queue pointers are on 1–4 bits and 9–12 bits respectively (c.f. Section VIII), the policies shown in Fig. 3 can be implemented with a relatively lightweight circuit.

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 misses handling is described in Fig. 3 and implemented by the circuit in Fig. 4. Secondary misses that are covered by the current burst bounds (Fig. 3a) require no updates to the MSHR. If the current burst does not cover the new miss, 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 (Fig. 3b). 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 (Fig. 3c). To handle secondary misses that fall in this case, we could, in principle, request an additional burst only for the new cache line. However, if the second burst is not guaranteed to cover the entire burst range, the problem may appear again once the second burst has also been sent out to memory. In practice, 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 Fig. 4, often already on the critical path of the entire system, would become even more complex.

To handle the cases shown in Fig. 3c 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 VII, this policy allows us to achieve the same operating frequency as our previous work [6]. 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, this happens naturally, as the next section will show.

V. 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, 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, 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, it will start to stall incoming requests: this reduces n_{acc} to $n_{acc}\prime$ and moves the equilibrium point to $n_{s,eq}\prime = n_{acc}\prime - n_{mem} < n_{s,eq}$. The larger the MSHR and subentry buffers, the closer $n_{s,eq}\prime$ 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.

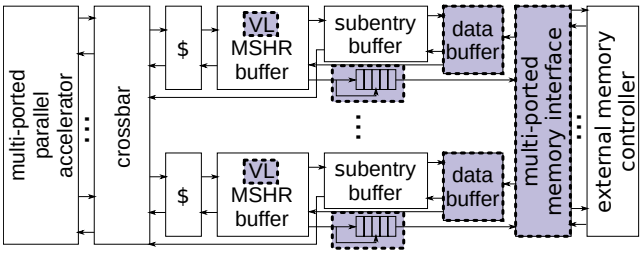

Fig. 5. Top level view of the proposed memory system. We highlighted the main differences compared to our previous memory system [6]: variable-length (VL) MSHR buffer, updatable queue, data buffer, and multi-ported memory interface.

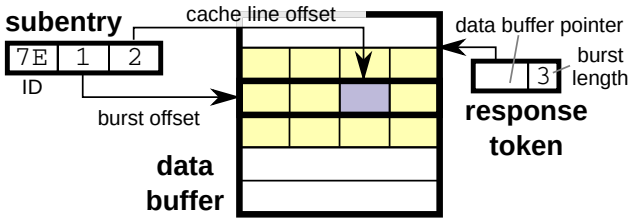

Fig. 6. Retrieval of responses from the data buffer. The response token, generated once the entire burst has been received, locates the response data inside the data buffer (yellow). Responses to the individual pending misses can be generated by iterating over all subentries and extracting the relevant word (purple) based on burst and cache line offsets.

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.

VI. System Architecture

Fig. 5 shows the top-level organization of DynaBurst, which we implemented by extending our previous memory system [9]. Within each bank, the variable-length (VL) MSHR buffer extends the previous buffer based on cuckoo hashing with stash by including the logic to maintain burst offsets and response invalidation discussed in Section III. Entries in the new output queue include burst bounds along with the tag and the queue must now allow updates of existing entries. In practice, the queue remains a dual-ported BRAM except that the write address is not restricted to the enqueue counter. Its depth 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. Subentries, which previously contained request ID and offset within the cache line, are augmented with a burst offset, i.e., the offset of the corresponding cache line within the burst. To handle memory controllers with multiple narrow ports, we generalized the memory system to handle up to n_{mem} memory ports, where n_{mem} is a divisor of n_b. We did so by replacing the single output arbiter/demultiplexer (for requests/responses respectively) with 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 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.

On the response path, we introduced a data buffer to store the data read from memory. In our previous work, the cache lines were sent directly to the MSHR and subentry buffer pipelines alongside their tag. This approach is not viable any more as each MSHR can refer to a variable number of cache lines and sizing each pipeline register for the maximum burst length (up to a few kilobits) results in large area wastage. Instead, we store the burst in the data buffer and only send its starting address (pointer), size (burst length), and tag to the pipeline as a single token for the entire burst. The MSHR buffer will first use the tag to retrieve its subentries, which contain all the information required to serve the respective pending miss as shown in Fig. 6. When all pending misses have been served, burst pointer and size are used to deallocate the burst data inside the data buffer. Because responses are treated in-order inside the pipeline, the data buffer can be implemented as a simple circular buffer. Moreover, as the data is accessed via pointers and used in a single place at the end of the pipeline, it can be packed more efficiently into BRAM or LUTRAM instead of spreading it into the more scarce flip-flops in the pipeline.

VII. Experimental Setup

DynaBurst has been implemented in Chisel 3 with Vivado 2017.4 and evaluated on a Xilinx ZC706 board with an xc7z045 FPGA, 1 GB of DDR3 memory on the processing system (PS) side, and 1 GB of DDR3 on the programmable logic (PL) side. The PS DDR is connected to the ARM's hard memory controller and is exposed to the FPGA through five 64-bit ports (HP0–HP3 and ACP); the PL DDR is accessed through the single 512 bit AXI interface of the MIG soft controller. Both controllers perform access reordering [10], [11]. Based on our measurements, the PS DDR achieves its peak bandwidth of 3.9 GB/s at 150 MHz with the four HP ports, while the PL DDR provides 9.0 GB/s at 150 MHz and 12.0 GB/s at 200 MHz.

We use the single-precision floating point compressed sparse row SpMV accelerators (CSR SpMV) from our previous work [6]. Such accelerators read three vectors sequentially—column index and value of the non-zero matrix entries and row pointers—and the dense vector irregularly, behind a 8192-entry reorder buffer and our memory system. Each multiply-accumulation (MACC) consumes two 32-bit words from sequential vectors (non-zero column index and value)

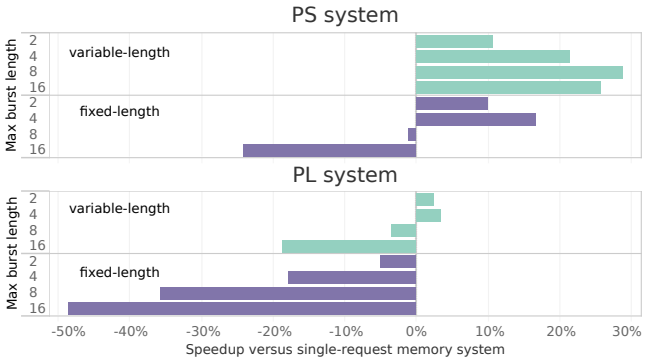

Fig. 7. Speedup of proposed architecture compared to our prior work [6], both with dynamically adjusted burst bounds (c.f. Section IV) and always requesting full bursts (geometric mean across all benchmarks and all configurations). Using bursts of a suitable size is beneficial to both systems and minimizing each burst's length is always better than using bursts of fixed length.

and one irregularly accessed word from the dense vector. One row pointer (sequential) is additionally consumed at the end of each row.

We consider two different configurations, which we take as representative of realistic use cases in commercial FPGA systems. In the *PL system*, the same used in our previous work [6], 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.

We use a superset of the SuiteSparse [12] sparse matrices used in our previous work [6] as benchmarks. We summarize their properties in Table I. Note that we propose a generic architecture that can dynamically extract locality without any assumptions on the application, hence the absence of SpMV-specific optimizations. For this reason, even though we use a single type of accelerator, we striven to cover a broad range of possible workloads by picking matrices from different domains (e.g., web/transport/social networks, linear programming) that have very different sparsity patterns.

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

TABLE I
BENCHMARK MATRICES USED IN THE EXPERIMENTS [12]. STACK DISTANCE PERCENTILES ARE A METRIC FOR THE TEMPORAL LOCALITY OF A MEMORY TRACE: HIGH STACK DISTANCE PERCENTILES MEANS THAT THE ACCESS PATTERN DURING SPMV WITH A GIVEN VECTOR HAS POOR TEMPORAL LOCALITY [13]. WE FOUND THESE PERCENTILES TO BE A BETTER PREDICTOR OF PERFORMANCE THAN, E.G., SPARSITY. ALL VECTORS ARE OF SINGLE-PRECISION FLOATING POINT VALUES.

VIII. RESULTS

For both PS and PL system we consider three configurations in terms of MSHRs and subentries. All systems have six subentries per row.

The 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 KB of cache per bank (4-way set associative, except for the 2-way 8 KB), 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 PL systems have (1) one 512-, (2) three 512-, and (3) four 1024-entry MSHR cuckoo hash tables per bank; 32–256 KB 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.

We will compare each of these architectures to alternative generic memory systems: (1) our previous memory system [6]—with unit-length burst—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. Systems (2) are the same baselines used in our previous work [6] and contain the maximum number of MSHRs and subentries that ensure timing closure at 200 MHz (PL systems) and that result in a slice utilization similar to the miss-optimized architectures (both systems).

A. Benefits of Dynamically Adjusting the Burst Length and Impact of Maximum Burst Length

Fig. 7 shows the speedup of DynaBurst compared to our prior single request memory system [6]. Adjusting burst bounds

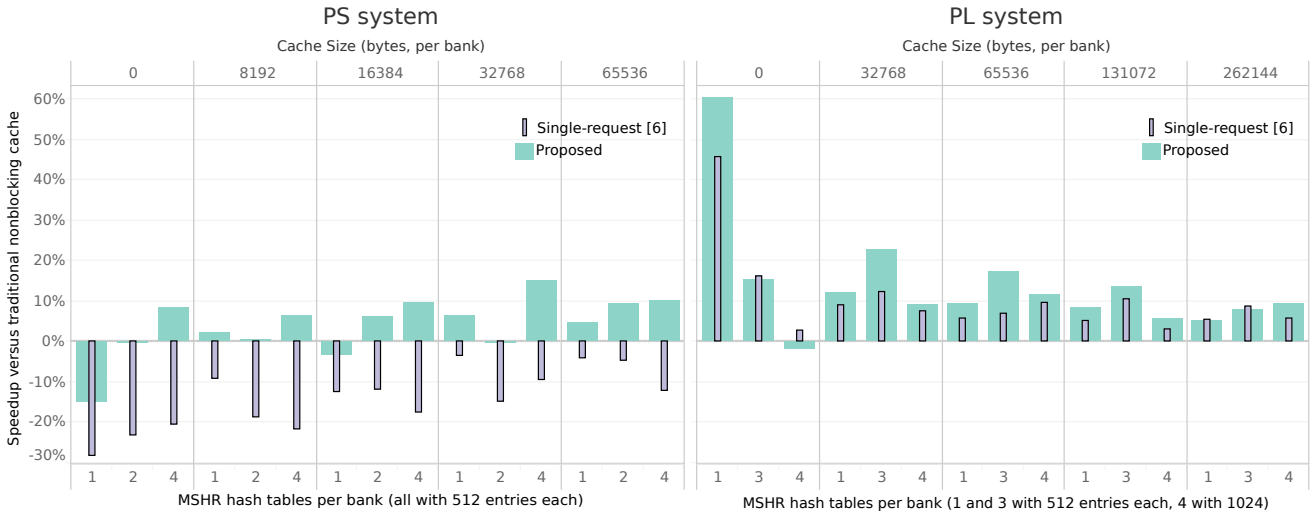

Fig. 8. Speedup of proposed systems (with maximum burst length of 8 and 4 for the PS and PL systems respectively) and single-request systems compared to the nonblocking cache with 16 associative MSHRs/bank with closest BRAM utilization (geometric mean across all benchmarks). Bursts are key enablers for miss-optimized architectures on memory controllers with multiple narrow ports (PS) where our previous system [6] performs worse than the traditional nonblocking cache with the closest area. Moreover, when memory interfaces are wide (PL), bursts bring further speedups to most data points where the single-request system was already reasonably effective.

is always useful, on all design points. On the 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 PL system 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 (c.f. Fig. 2 and 3). This tradeoff explains the bitonic speedup curve on both systems.

Overall, the 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. PL systems benefit from bursts only through DRAM row conflict minimization, which still brings significant speedup on specific design points as discussed in the next section.

B. Architectural Exploration

We further analyze the architectures with the ideal maximum burst length for the respective system—4 and 8 for PL and PS systems respectively. Fig. 8 explores the benefit of DynaBurst for different cache sizes and MSHR count.

If the memory controller has multiple narrow ports (PS system), 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 our original results [6] 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.

C. Detailed Speedup Profile

Fig. 9 shows the speedup on individual benchmarks provided by the best-performing architecture on each system (excluding the peculiar case with no cache and 1×512 MSHRs/bank on the PL 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 MSHR-rich architectures useful also on the PS system, achieving up to 3.4× speedup. On the PL system, burst architectures improve the performance of miss-optimized systems 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 traditional nonblocking cache was performing worse.

D. 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, we simulated a bad and a good performing benchmark on the PS and PL systems analyzed in Section VIII-C and analyzed how many of the cache lines requested from memory are actually used. More specifically, Fig. 10 shows, for each burst length, how many of the requested cache lines have been actually used at least once and how many

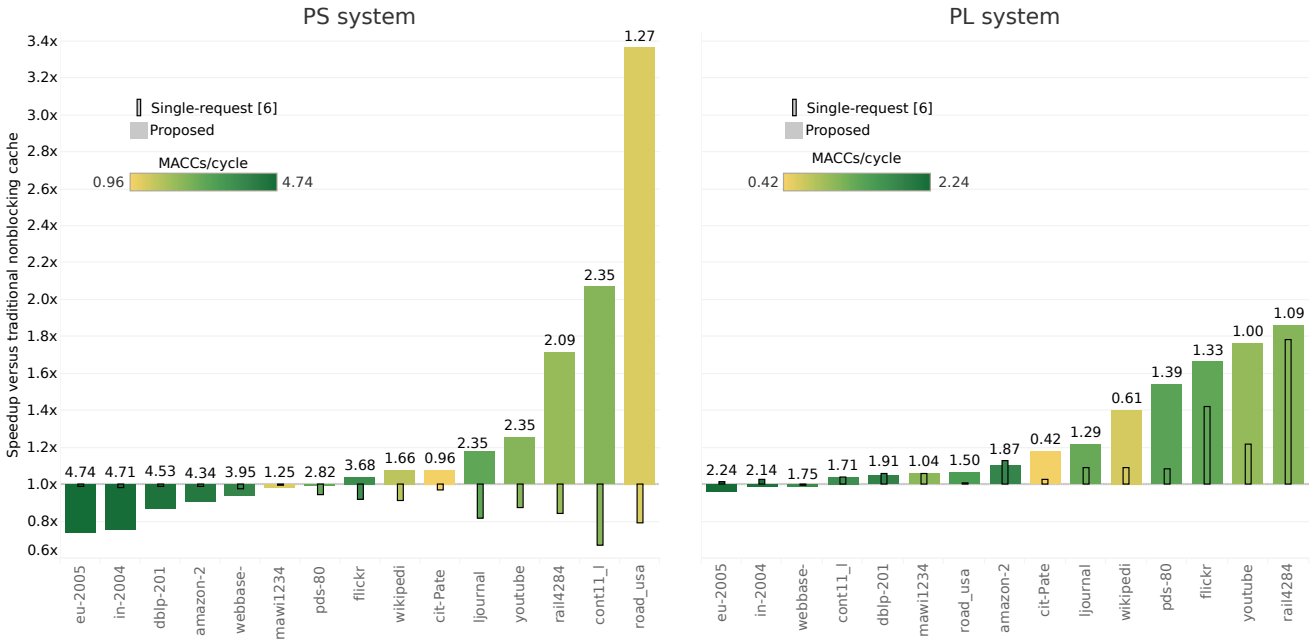

Fig. 9. Speedup provided by the proposed and single-request architecture on individual benchmarks compared to traditional nonblocking cache, at 32 KB of cache and 4×512 (3×512) MSHRs per bank on PS (PL) system. Bars are color-coded based on the absolute performance, in MACCs/cycle, achieved by the traditional nonblocking cache baseline, also displayed on top of each bar. Our burst-based miss-optimized architecture is beneficial to most of the largest and/or irregular benchmarks, where the baseline has the lowest performance.

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. 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 though the total share of wasted data is similar in both PS benchmarks, and higher than in the PL system, the speedup provided by DynaBurst is significantly higher in road_usa than in eu-2005.

Where DynaBurst performs well, most of the bursts converge to their optimal length (according to the policy described in Section IV) 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.

E. Resource Utilization

Fig. 11 shows the area of all the PS and PL architectures with ideal maximum burst length (8 and 4 respectively). Due to the logic for burst handling, slice overhead is 30–40% compared to single request systems, which brings slice utilization with one and two hash tables per bank close to that of traditional caches where associative MSHRs are stored in flip flops. BRAM overhead is more modest (0–15%) and is due to the additional burst offset bits in each subentry and to the data buffer. Trends in the PL system are very similar but overheads are even smaller (10–15% slices, 2–15% BRAM) thanks to the wide pipeline registers for 512-bit cache lines, previously in flip-flops, that are now packed more efficiently into LUTRAM in the data buffer.

IX. Related Work

All modern DRAM controllers implement some form of memory operation reordering such as the first-ready first-come first-serve (FRFCFS) policy which prioritizes row hits [5] or one of the many alternatives [14]–[20]. 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. This was also the focus of our previous work [6], which has been extensively discussed in Sections I and II.

Several works automatically generate application-specific memory systems. TraceBanking [21] uses a memory trace to produce efficient on-chip memory banking schemes. Cong et al. [1] restructure local buffers in HLS applications to make DRAM-BRAM memory transfers more efficient. EASY [22] uses an SMT solver to minimize the number of BRAM

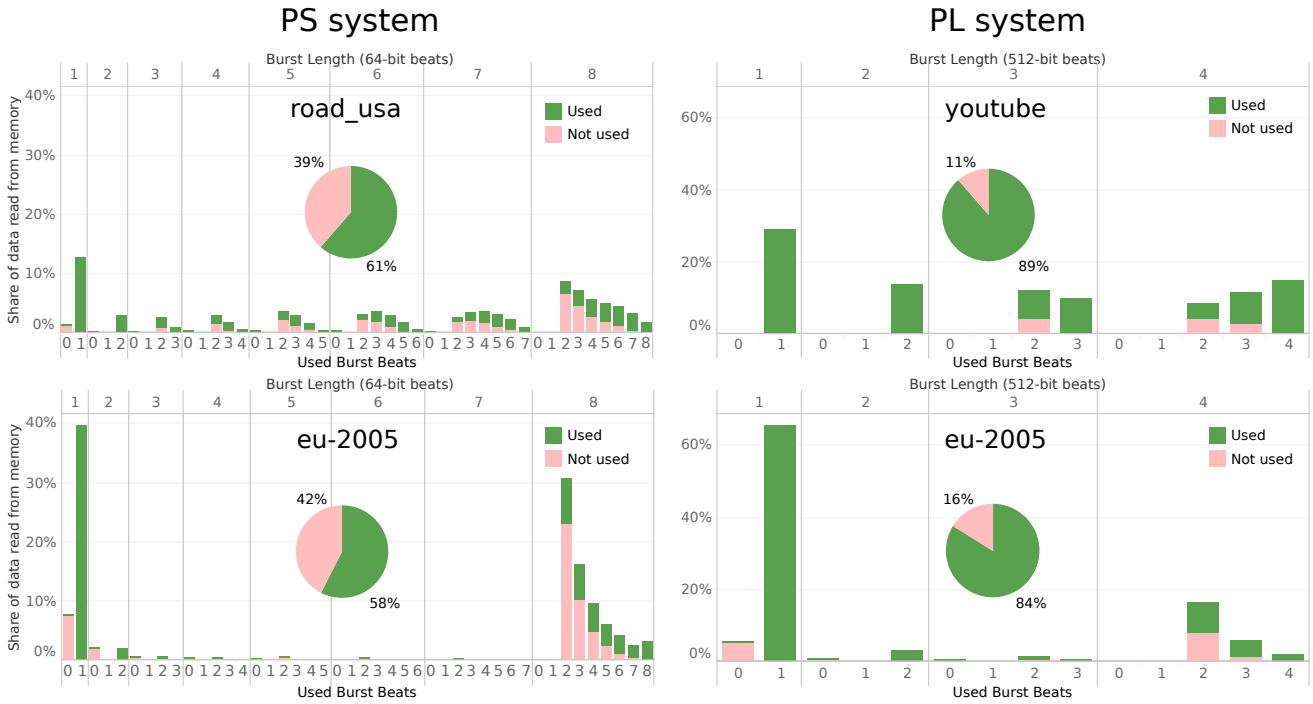

Fig. 10. Distributions of requested, used, and wasted cache lines per burst as a function of the burst length, normalized by the total number of cache lines requested from memory, for the same systems evaluated in Fig. 9. Pie charts: used and wasted cache lines, aggregated over all burst lengths. Top (bottom) row: benchmarks where DynaBurst performs well (poorly) compared to baseline and single-request systems. Benchmarks that get the highest speedup from the proposed architecture obtain a large share of useful data through bursts of all possible sizes. On the low end of the performance spectrum, invalidations and single requests are more frequent.

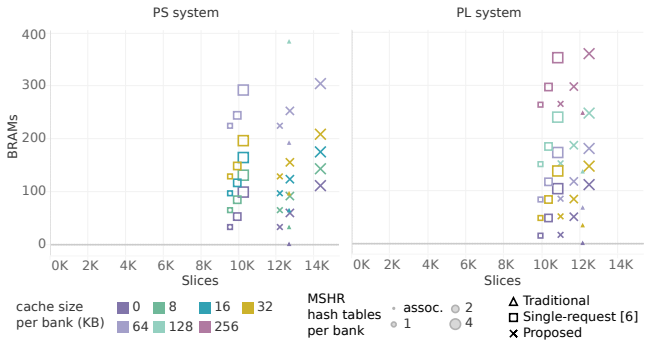

Fig. 11. Area of all systems considered in the exploration in Section VIII-B. On the PS systems, burst handling has a 30–40% slice and 0–15% BRAM overhead compared to single-request miss-optimized architectures and similar slice utilization as traditional nonblocking caches. The overhead is even smaller on the PL systems (10–15% slices, 2–15% BRAM) as it is partially compensated by a more efficient storage of the wide cache lines, from flip-flops to LUTRAM.

bank arbiters for multi-threaded HLS accelerators. All these contributions rely on precise compile-time information on the access pattern and need the entire data set to fit in on-chip memory or at least to be processable in a tiled fashion—two limitations that do not concern our system.

The methodology from Bayliss et al. [2] generates application-specific reuse buffers that minimize memory traffic and row conflicts, but is restricted to affine loop nests. ConGen [23] minimizes row conflicts by scrambling DRAM address bits by analyzing at compile-time the application's full memory trace. MATCHUP [24] and LMC [25] 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.

X. Conclusion

Irregular memory access patterns bring DRAM memories far from their optimal operating point, which reduce the benefit of datapath parallelization. When application-specific solutions are not available, miss-optimized memory systems improve throughput of latency-insensitive applications by dynamically reusing the data returned from DRAM as much as possible and often more efficiently than a traditional nonblocking cache with the same area. DynaBurst extends such systems by requesting variable-length bursts from memory, which increase the absolute amount of DRAM bandwidth available to the FPGA by using larger portions of DRAM bursts and of the DRAM row buffer. This makes miss-optimized systems beneficial also behind DRAM controllers with multiple narrow ports, commonly found on SoC platforms, and further increase their usefulness when memory ports are wide. Our memory system can be downloaded as an open-source project from `https://github.com/m-asiatici/dynaburst`.

REFERENCES

[1] 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)*. IEEE, 2017, pp. 1–6.

[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] JEDEC, *DDR3 SDRAM Standard JESD79-3F*. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd-79-3d

[4] ——, *DDR4 SDRAM Standard JESD79-4B*. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4a

[5] 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.

[6] 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.

[7] Xilinx Inc., *Zynq UltraScale+ MPSoC Processing System v3.1 (PG201)*, Oct. 2017.

[8] Intel Corp., *Intel® Stratix® 10 SoC Device Design Guidelines (AN-802)*, Sep. 2018.

[9] M. Asiatici and P. Ienne. Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs (public repository. [Online]. Available: https://github.com/m-asiatici/MSHR-rich

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

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

[12] 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.

[13] 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.

[14] J. Shao and B. T. Davis, "A burst scheduling access reordering mechanism," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 285–294.

[15] H. G. Rotithor, R. B. Osborne, and N. Aboulenein, "Method and apparatus for out of order memory scheduling," US Patent US7127574, Oct. 2006.

[16] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Computer Architecture News*, vol. 36, no. 3. ACM, 2008, pp. 39–50.

[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," vol. 12, no. 4, p. 65, 2016.

[18] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Computer Architecture News*, vol. 36, no. 3. ACM, 2008, pp. 63–74.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, "MATCHUP: memory abstractions for heap manipulating programs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 136–145.

[25] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer, "LMC: Automatic resource-aware program-optimized memory partitioning," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 128–137.