EPFL

Thèse n° 8050

Miss-Optimized Memory Systems: Turning Thousands of Outstanding Misses into Reuse Opportunities

Présentée le 3 septembre 2021

Faculté informatique et communications
Laboratoire d'architecture des processeurs
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Mikhail ASIATICI

Acceptée sur proposition du jury

Prof. E. Bugnion, président du jury
Prof. P. Ienne, directeur de thèse
Dr K. Vissers, rapporteur
Prof. V. Prasanna, rapporteur
Prof. G. Alonso, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

Questions you cannot answer
are usually far better for you
than answers you cannot question.
— Yuval Noah Harari

To Maya, Mamiko, and my parents.

Abstract

Even if Dennard scaling came to an end fifteen years ago, Moore's law kept fueling an exponential growth in compute performance through increased parallelization. However, the performance of memory and, in particular, *Dynamic Random Access Memory* (DRAM), has been increasing at a slower pace for decades, making memory system optimization increasingly crucial. Conventional solutions mitigate the issue by shifting as many memory accesses as possible from off-chip DRAM to on-chip *Static RAM* (SRAM) memory, which has higher performance but lower capacity. This is achieved by relying on spatial and temporal locality or on precise compile-time information about the access pattern. However, when the access pattern is irregular and data-dependent, these solutions are ineffective and the processing-memory gap grows even wider as DRAMs themselves are optimized for sequential accesses.

In this thesis, we present a novel memory system for throughput-oriented compute engines that perform irregular read accesses to DRAM. When accesses are irregular, we acknowledge that obtaining a reasonable benefit from on-chip memory may be unrealistic; therefore, we focus on minimizing stalls and reusing each memory response to serve as many misses as possible without relying on long-term data storage. This is the same insight behind nonblocking caches but on a vastly larger scale in terms of outstanding misses, which greatly increases the opportunities for data reuse when accelerators emit a large number of outstanding reads. Because we optimize miss handling rather than increasing hit rate, we call our architecture *miss-optimized memory system* (MOMS).

We first focus on the microarchitectural level to show how a MOMS can support three orders of magnitude more outstanding misses than a traditional nonblocking cache in a way that can be efficiently implemented on *Field-Programmable Gate Arrays* (FPGAs). Once we maximize the reuse of each individual word returned by the DRAM, we introduce two techniques to increase the DRAM throughput. When the DRAM controller is optimized for burst requests, we group incoming requests over multiple words that are requested as a burst. Conversely, when the DRAM controller handles single requests efficiently, our MOMS reorders requests by DRAM bank and row on a much larger scale than general-purpose DRAM controllers. We then discuss techniques to use efficiently the vast amount of resources provided by multi-die FPGAs and introduce two-level architectures which balance reuse maximization and contention of shared hardware. Finally, we develop a graph processing accelerator backed by a MOMS. On three algorithms on graphs with billions of edges and up to a hundred million nodes, our accelerator outperforms the state-of-the-art on FPGAs and achieves higher performance per watt and unit bandwidth than the state-of-the-art on CPUs and GPUs.

Abstract

Memory systems designers face increasing pressure to keep up with the compute engines performance. Our results suggest that miss-optimized memory systems can help to reduce the memory-processing gap where it is largest, that is, when accesses to memory are irregular and difficult to serve from local buffers.
Keywords: DRAMs, nonblocking caches, MSHRs, bandwidth-bound, throughput-oriented, FPGAs

Sunto

Nonostante lo scaling di Dennard sia terminato quindici anni fa, la legge di Moore ha continuato ad alimentare una crescita esponenziale delle prestazioni di calcolo attraverso una maggiore parallelizzazione. Tuttavia, le prestazioni delle Dynamic Random Access Memories (DRAM) è aumentata ad un ritmo più lento per decenni, rendendo l'ottimizzazione del sistema di memoria sempre più cruciale. Le soluzioni convenzionali come le cache e gli scratchpad mitigano il problema spostando il maggior numero possibile di accessi dalla DRAM esterna alla memoria RAM statica (SRAM) interna, che ha prestazioni più elevate ma una capacità inferiore. Ciò fa affidamento sulla località spaziale e temporale o su precise informazioni a tempo di compilazione sulla sequenza di accessi in memoria. Tuttavia, quando tale sequenza è irregolare, queste soluzioni si rivelano inefficaci e il divario elaborazione-memoria cresce ancora di più poiché le DRAM stesse sono ottimizzate per gli accessi sequenziali.

In questa tesi, si presenta un nuovo sistema di memoria per acceleratori orientati al throughput che eseguono accessi irregolari alla DRAM. Quando gli accessi sono irregolari, riconosciamo che ottenere un beneficio ragionevole tramite buffering locale può essere irrealistico; ci si concentra pertanto sulla minimizzazione degli stalli e sul riutilizzo di ogni risposta di memoria per servire il maggior numero possibile di miss senza fare affidamento sullo stoccaggio di dati a lungo termine. Questa è la stessa intuizione alla base delle cache non bloccanti, ma su una scala molto più grande in termini di miss in sospeso, il che aumenta notevolmente le opportunità di riutilizzo dei dati quando gli acceleratori emettono un gran numero di letture in sospeso. Poiché si ottimizza la gestione delle miss piuttosto che aumentare i cache hits, chiamiamo la nostra architettura *sistema di memoria ottimizzato per le miss* (*miss-optimized memory system*, MOMS).

Nella prima parte della tesi si mostra ciò che, a livello microarchitetturale, permette ad un MOMS di gestire tre ordini di grandezza in più di miss in sospeso rispetto a una cache non bloccante tradizionale in una maniera che può essere implementata efficientemente su *Field-Programmable Gate Arrays* (FPGAs). Si introducono quindi due tecniche per aumentare il throughput della DRAM, utilizzando richieste *burst* o riordinando richieste singole su una scala molto più grande rispetto ai controller DRAM *general-purpose*. In seguito, si introducono delle linee guida per utilizzare in modo efficiente le FPGA multi-die e si presentano architetture a due livelli che bilanciano la massimizzazione del riutilizzo dei dati ricevuti dalla DRAM e i conflitti di accesso all'hardware condiviso. Si discute infine un acceleratore per elaborazione dei grafi supportato da un MOMS e si dimostra che raggiunge prestazioni superiori rispetto allo stato dell'arte su FPGA ed ha una maggiore efficienza per watt e per unità di banda passante rispetto

allo stato dell'arte su CPU e GPU nell'esecuzione di tre algoritmi su grafi con miliardi di archi e fino a un centinaio di milioni di nodi.
I progettisti di sistemi di memoria devono affrontare una crescente pressione per tenere il passo con le prestazioni dei motori di calcolo. I risultati esposti in questa tesi suggeriscono che i MOMS possono aiutare a ridurre il divario memoria-elaborazione dove è più grande, ovvero quando gli accessi alla memoria sono irregolari e difficili da soddisfare localmente.
Parole chiave: DRAMs, cache non bloccanti, MSHRs, bandwidth-bound, throughput-oriented, FPGAs

Acknowledgements

I am deeply grateful to all the people that made this long journey possible and enjoyable.

I sincerely thank my supervisor, Paolo Ienne, for entrusting me with a great deal of freedom and at the same time providing continuous guidance, support, and encouragement. I learned a lot from him at the technical, personal, and research level and I feel particularly lucky for having had the chance to work with him. I would also like to thank the members of my thesis committee, Gustavo Alonso, Edouard Bugnion, Viktor Prasanna, and Kees Vissers for finding the time to review my work and for their valuable feedback.

I had the privilege to work with many great minds who helped me shaping my research direction. I am very much obliged to Kristof Denolf, Kees Vissers, and Stephen Neuendorffer for exposing me to interesting open industrial problems, including the application that inspired my research, and for their feedback along the way. I am also grateful to Michael Adler for the insightful discussions about memory-related challenges on FPGAs, and to David Novo, Gabriel Falcao, and Dick Sites for their comments on my research.

During my PhD, I had the chance to do an internship at Microsoft Research in Cambridge, UK, within Project Honeycomb. I would like to thank Junyi Liu and Aleksandar Dragojevic for giving me the opportunity to explore the world of industrial research and for their valuable guidance.

I am very thankful to all of my colleagues at LAP—Ana, André, Andrea, Andrew, Aya, Chantal, Grace, Lana, Nithin, Paolo, René, Sahand, Stefan—for the coffees, lunches, beers, outings, table football games, and the countless other good times we had together. I am particularly grateful to Andrea, Lana, Sahand, and my office mate Stefan for their friendship and support throughout many years at LAP, to Nithin, for introducing me to FPGAs and for his precious guidance, and to Chantal, for always being there to help with any problem, no matter what kind or how big. I would also like to thank Joao and Gabriel, whose short visits were enough to leave many good memories together.

I had great pleasure in working with many excellent interns, including Amna, Andrew, Aya, Kushagra, Sena, Patryk, and my coauthors Damian and Gabor whose work had a significant impact on my research.

A big thank you also to Raffaele, with whom I shared struggles and success since we first set foot in a university back in Italy, all the way to EPFL, as well as Michele and Giuseppe.

Finally, none of this would have been possible without the support and love from my family, to whom I express my deepest gratitude. I am forever indebted to my parents Andrea and Graziella for their unconditional and endless support and for making me the person I am today. I am eternally grateful to my wife Mamiko, for always encouraging me to push beyond my limits and having believed in me even when I did not. Her support made it possible to overcome the darkest moments and made the happiest news even more memorable. Last but not least, I am forever grateful to my daughter Maya, who came into my life halfway through this journey, the day after a paper submission deadline, for turning my world upside down and making me more fulfilled that I could have ever imagined.

Lausanne, May 25, 2021 M. A.

Contents

List of Figures

List of Tables

1 Introduction

Since the inception of electrical computers 80 years ago, the available processing power has been increasing at a tremendous pace. Between 1956 and 2015, the processing power available on a single system has witnessed a 1-trillion fold increase [84] and the number of floating-point operations that an Xbox One X from 2017 can perform in one second would take 190 years on the Atlas, the fastest supercomputer in 1962 [96, 103]. Figure 1.1 shows that the number of transistors in a single *Central Processing Unit* (CPU) has increased by a factor 10^7 in the past 48 years, which led to an exponential increase in single-thread compute performance until the end of Dennard scaling around 2005 and, afterwards, of parallel cores in multicore CPUs.

1.1 Memory-Computation Gap

Memory capacity has also increased exponentially with time [39], although at a slower pace than the number of processing cores [65]. However, the same cannot be said about bandwidth and latency, as shown in Figure 1.2. While the number of transistors between 1999 and 2017 increased by $4,000\times$ and the number of parallel cores between 2005 and 2020 by $100\times$, the bandwidth of main memory only increased by a factor $20\times$ and latency essentially stagnated.

The implications of this gap can be visualized in the roofline model [105], shown in Figure 1.3, which shows the achievable performance in, e.g., *Floating Point Operations per Second* (FLOPS), as a function of the *operational intensity*, i.e., the number of operations per byte read from memory. While at high operational intensities the performance is limited by the processing power of the compute engine, the bottleneck at low operational intensity becomes memory bandwidth. Increasing the processing power shifts the horizontal ceiling up, while increasing the memory bandwidth shifts the sloped memory ceiling leftwards. When the processing power increases more quickly than memory bandwidth as in Figure 1.4, the intersection between the two ceilings (*ridge*) shifts to the right. This means that the minimum arithmetic intensity that is required to fully exploit a compute engine and achieve peak performance increases. Applications where this condition is not satisfied are said to be *memory-bound* and are becoming more and more common [105].

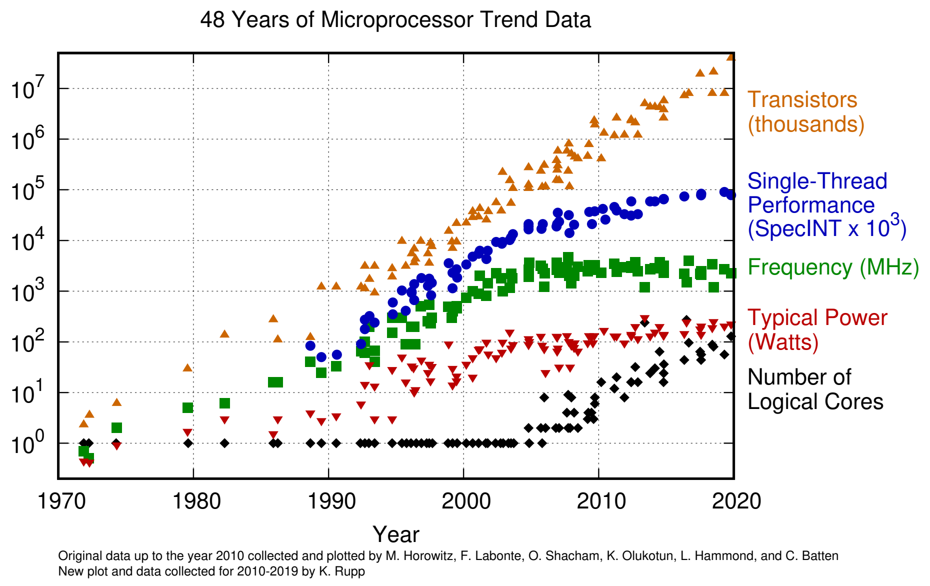


Figure 1.1 – Forty-eight years of microprocessor trends. The number of transistors in a CPU increased by seven orders of magnitude; while the growth in single-thread performance and clock frequency has slowed down or stopped since the mid-2000s, the number of parallel cores took over and increased by two orders of magnitude in 15 years. Reprinted from Karl Rupp's GitHub repository [86].

The examples introduced so far consider a single value for the memory bandwidth; however, in the case of the *Dynamic Random Access Memories* (DRAM) which make up most of the main memory market as of 2020 [92, 1, 100], this is a simplification. The actual DRAM bandwidth is indeed heavily dependent on the *access pattern*, that is, the pattern of addresses that are sent to the memory, as well as the granularity of each memory operation. Therefore, a more complete roofline model defines a region where the memory bandwidth ceiling can be, depending on the properties of the access pattern, rather than a single line, as shown in Figure 1.5. We will show in Section 2.6.4 that the access pattern alone may decrease the memory bandwidth by up to 7×, which can be compounded by another order of magnitude in the case of fine access granularity.[1] As a result, for a broad range of operational intensities, access pattern and granularity will define whether the application will be able to fully exploit the available compute resources or not.

1.2 Limitations of Existing Solutions

The most common way to increase memory bandwidth is to shift as many accesses as possible to *Static Random Access Memories* (SRAMs), whose bandwidth is not only higher than that of DRAMs but also insensitive to the access pattern. Since SRAMs have a larger area and a higher price per bit than DRAM, memories based on the two technologies are usually combined in a *memory hierarchy*. From the perspective of the compute engine, an ideal memory hierarchy behaves as a memory that is as large as the DRAM but as fast as the SRAM: this is achieved by

[1]For example, for a common minimum access granularity of 512 bits, 32-bit irregular accesses can only use $\frac{1}{16}$ of each response

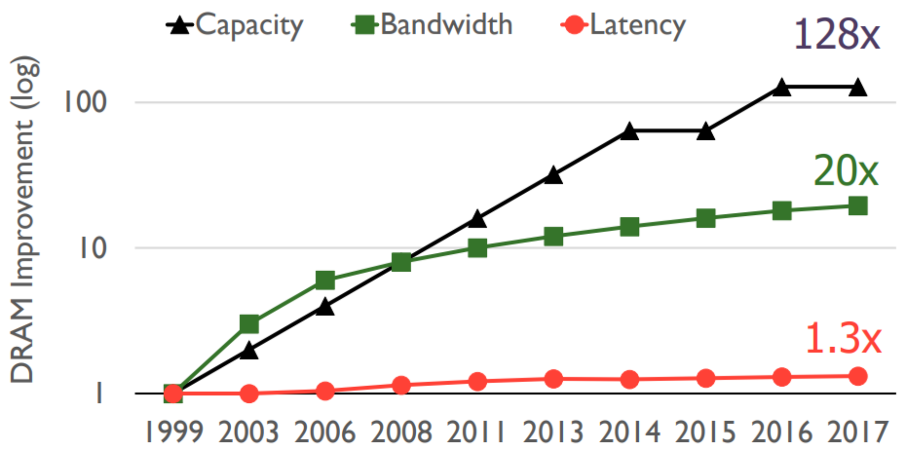


Figure 1.2 – DRAM capacity, bandwidth, and latency trends from 1999 to 2017. Scaling efforts focused on capacity, followed by bandwidth; latency remained mostly the same during 18 years. Reprinted from Onur Mutlu's presentation [72].

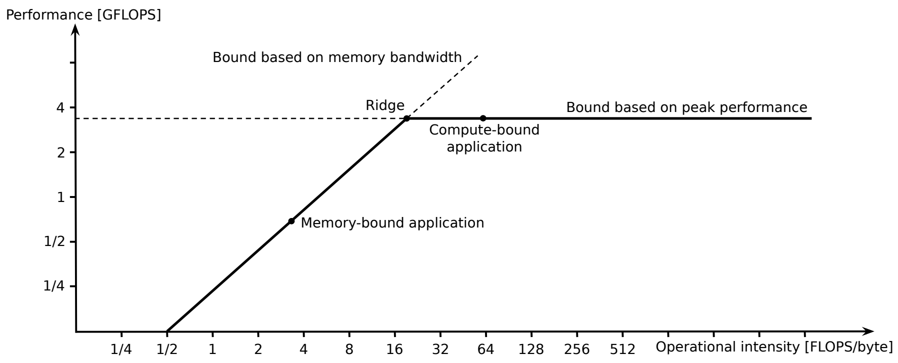


Figure 1.3 – Roofline model definitons. The roofline model provides a visual insight on the bottlenecks of a kernel on a given platform. When the number of operations per transferred byte (*operational intensity*) is high, the performance is limited by compute; conversely, memory bandwidth limits performance at low operational intensities. Reprinted with modifications from Wikipedia [104].

serving as many memory accesses as possible directly from the SRAM while at the same time migrating data from DRAM to SRAM as few times as possible.

Custom memory hierarchy design and automatic generation usually rely on access patterns that are regular (scratchpads), have temporal and spatial locality (scratchpads, caches) or are at least known at compile-time (memory banking and address scrambling) [3, 33, 18, 119]. When access patterns are irregular and data-dependent and rewriting the application is not viable, designers are left with maximizing memory-level parallelism (MLP) by generating enough outstanding memory operations to at least make DRAM access fully pipelined. However, the throughput of the memory system is still capped to one operation per cycle per DRAM channel, at best. This imposes severe limitations on the amount of datapath parallelism that is worth implementing, reducing the advantage of hardware specialization.

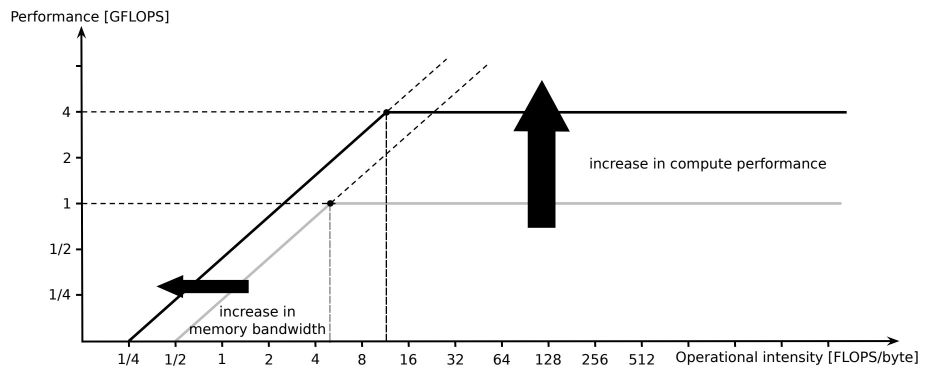


Figure 1.4 – Impact of memory bandwidth and compute power scaling on the roofline model. If the peak compute performance scales faster than the memory bandwidth, the ridge of the model shift rightwards, meaning that the minimum operational intensity that makes the kernel compute-bound increases.

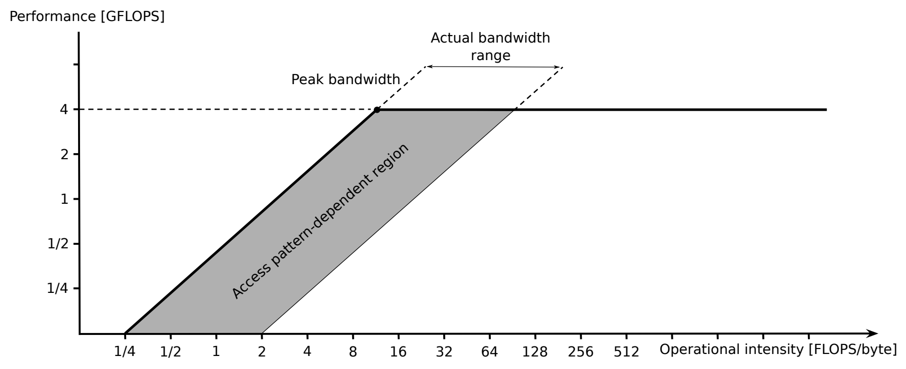


Figure 1.5 – Impact of memory bandwidth variability on the roofline model. The bandwidth in common external memories is heavily dependent on the access pattern; as a result, a particularly unfavorable access pattern may turn a compute-bound kernel into a memory-bound one, wasting compute resources.

In fact, one operation per cycle per DRAM channel is only a theoretical bound: as discussed in Section 1.1, the actual throughput is often significantly lower, especially when accesses are irregular. If accesses are not only irregular but also narrow (such as 32- or 64-bit scalars), the effective bandwidth gets reduced even further. Two separate mechanisms contributes to the bandwidth degradation in those cases. Firstly, both DDR3 and DDR4 operate on bursts of eight beats, normally of 64 bits each [47, 48], which results in a minimum access granularity of a full 512-bit burst. If accesses are narrower than the burst size, the remaining data returned from memory will be discarded, wasting memory bandwidth and energy. In addition, serving uncorrelated streams of requests from multiple accelerators can only be done by time-multiplexing the memory channel, canceling out any benefits due to parallelization when memory bandwidth is the bottleneck. The only way to improve bandwidth utilization, and thus performance, would be to use larger portions of each burst returned from memory.

The second mechanism relates to the organization of bits in DRAM: a few banks (8 and 16 for DDR3 and DDR4 respectively), each consisting of a two-dimensional array of capacitors.

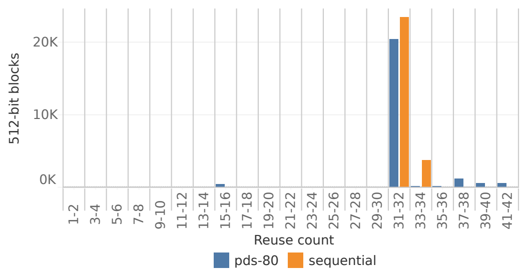


Figure 1.6 – Spatial locality. Reuse count for each 512-bit block of data, for SpMV of pds-80 and for the same number of read operations performed sequentially over the same memory range, wrapping back at the end. Despite showing very different cache hit rates, both memory traces have similar amounts of data reuse across the entire application execution.

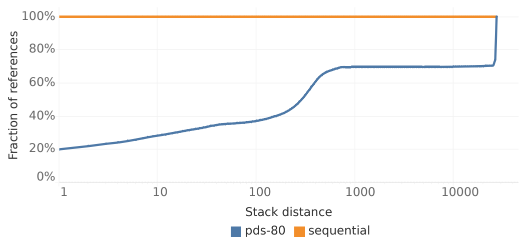


Figure 1.7 – Temporal locality. Fraction of 512-bit block references that have stack distance ≤ x, for SpMV of pds-80 and a sequential memory trace. A large fraction of the reuses that occur in pds-80 are interleaved with references to many different blocks. Blocks can be stored in a cache hoping for future reuse; however, because large stack distances are common, cache lines are likely to be evicted before the next reuse, unless a large cache is used.

Reading data involves first copying the respective row to the *row buffer* [47, 48], an operation that has to be repeated every time a different row is accessed in the same bank. Since this operation is 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 [82]; however, because general-purpose controllers must also minimize latency, the internal request queues are relatively shallow and the optimization is possible only for accesses close in time. More details on DRAM operation and on DRAM controllers are provided in Section 2.3 and 2.4.

1.3 Opportunities for Data Reuse

To discuss the opportunities to increase the utilization of each 512-bit blocks received from memory, Figure 1.6 shows the histogram of the number of reuses of 512-bit blocks for an application with poor locality—accesses to the dense vector of 32-bit integers during sparse matrix-vector multiplication (SpMV) with the pds-80 matrix from SuiteSparse [27] encoded in Compressed Sparse Row (CSR) format. It also illustrates what would happen if the same number of read operations were performed sequentially over the same address span—that is, if instead of reading elements of the dense vector by following the sparsity pattern of the matrix, we simply read the entire vector sequentially, wrapping back to the beginning when reaching the end, until performing the same amount of read operations as in the first case. The sequential access pattern achieves a 15/16 = 94% hit rate on any cache with 512-bit cache lines: the first access to every cache line is a compulsory miss but the following 15 accesses hit on the same cache line which has just been fetched. In contrast, we observed that the SpMV access pattern only achieves a 57% hit rate on a 128 kiB direct-mapped blocking cache, despite having very similar opportunities for reuse across the access pattern as a whole. This is because, in a cache, eviction limits the time window where data reuse could occur. For the same two access patterns, Figure 1.7 shows the cumulative frequency of *stack distances*, i.e., the number of different 512-bit blocks that have been referenced between two consecutive references to the same blocks [20]. For example, for the memory trace: {389, 261, 124, 4938, 261, 389}, the stack distances for the last access to blocks 261 and 389 are 2 and 3 respectively. While the stack distance of the sequential pattern is always zero, the SpMV cumulative histogram grows very slowly, meaning that a large fraction of reuses have large stack distance. With an ideal fully associative cache with N lines and LRU replacement, reaccessing a cache line with stack distances larger than N will always be a miss; on a realistic cache, even reuses with stack distance lower than N could be misses.

1.4 Pushing Nonblocking Caches to the Extreme

The previous example shows that even applications with poor temporal locality may still have some spatial locality, which caches struggle to harness due to large stack distances between reuses. Even worse, a blocking cache actually hampers performance if the hit rate is too low to compensate for the stall cycles due to the misses. Nonblocking caches, described in detail in Section 2.2, reduce stall penalties by handling a small number of misses without stalling. In addition, they also group misses by cache line so that a single cache line request can be used to serve all the respective misses. This is implemented by storing each in-flight cache line in a *miss status holding register* (MSHR), each comprising multiple *subentries* that contain the offset and source of the respective misses. This organization pushes the maximum MLP beyond the DRAM latency and increases bandwidth utilization as long as there are available MSHRs and subentries [68].

Indeed, adding an MSHR with its subentries enlarges the reuse window just like an extra cache line: while a cache stores entire blocks hoping for future reuse, MSHRs and subentries keep track of which portions of the blocks are actually needed. This way, as soon as a block is received, the miss handling logic serves the block requests that have been accumulated and then discards the block. With typical block and address sizes of hundreds and tens of bits respectively, the on-chip memory cost of reusing a block is hundreds of bits if done using the cache or tens of bits *per reuse* using MSHRs and subentries. This means that if the number of block reuses is greater than one but sufficiently small, reusing it via MSHRs and subentries has a lower on-chip memory cost than relying on the cache for the same purpose.

Nonblocking caches are extensively used in processors; however, MSHRs are usually searched in a fully-associative structure to minimize latency, which limits their number to a few tens. In practice, there is often little benefit in increasing the number of MSHRs beyond this limit on realistic CPUs [64, 98]. On FPGAs, fully-associative searches are even less scalable than in ASICs; yet, high-throughput massively parallel FPGA accelerators that generate a large number of outstanding reads to hide memory latency [66, 23] could potentially benefit from an MSHR-rich architecture even more than a general-purpose processor.

1.5 Storing Requests Instead of Data

This thesis is based on the observation that serving a request (1) by using data that is in cache or (2) by reusing a block that has been already requested to serve another miss have the same impact on throughput. This means that, when latency is irrelevant, both are equally beneficial to performance. While both mechanisms can satisfy the request without generating extra memory traffic, thus without adding pressure to the memory bottleneck, the latter requires less on-chip memory when reuse exists but is limited. Indeed, reusing an in-flight cache line only requires storing some miss metadata until the data returns from memory as opposed to keeping the entire cache line as long as possible in the cache array.

1.6 Thesis Outline

After covering some background and related work on nonblocking caches, DRAM memories, and graph processing in **Chapter 2**, we discuss in **Chapter 3** how to scale up the MSHR array by three orders of magnitude. Doing so maximizes the probability for a miss to target a cache line whose request has been already sent out or at least queued.[2] We show that repurposing some on-chip memory from cache to MSHRs generally proves to be beneficial, especially when the DRAM controller exposes the entire DRAM burst through a wide data port and provides enough bandwidth even to single requests.

When DRAM controllers instead have multiple narrow ports, which is common on SoC platforms, or are heavily optimized for bursts and individual accesses cannot be fully pipelined, optimizing the reuse of individual memory requests is generally useful but leaves some performance on the table. To address these scenarios, in **Chapter 4** we show how to extend MSHRs to support bursts of variable length on the memory side. When possible, we make bursts longer and exploit more of a DRAM burst or row without being limited to the data width exposed by the specific memory controller. Conversely, when spatial locality is insufficient, we keep burst short and minimize contention in the controller or avoid requesting unnecessary DRAM bursts. Requesting bursts extends the scope of MOMSes to memory controllers with narrow ports or where single accesses are heavily penalized compared to burst accesses. In addition, bursts make accesses locally more sequential which, based on our memory characterization in Section 2.6.4, generally increases the total bandwidth that the memory can deliver.

Extending MSHRs to support bursts only requires minor modifications to the MOMS architecture; however, this simplicity comes with limitations in the way the generated bursts cover the data requested by the accelerators. In particular, the MOMS may end up requesting data that is not needed, which may account for up to 40% of the data received from memory, a fraction that is sometimes high enough to offset the bandwidth gains provided by bursts. To overcome these limitations, we observe that in order to gain visibility over a large number of incoming

[2] In other words, to increase the probability that new misses will be secondary rather than primary, using terms that will be introduced in Section 2.2.

requests with minimal stalls, MOMSes also need a deep output queue for memory requests. As a byproduct, this deep output queue also gives MOMSes access to thousands of future memory requests. In **Chapter 5**, we discuss how to use this information to explicitly reorder individual memory requests across a window that is three orders of magnitude larger than that of a typical DRAM controller. We show that, on controllers where single accesses are not significantly penalized over bursts and where we have control on the DRAM command scheduling policy, we can reduce DRAM row conflicts without incurring in the data wastage associated with bursts.

After maximizing the achievable DRAM bandwidth on a single DRAM channel, in **Chapter 6** we address the challenges associated to scaling our MOMS over multi-die FPGAs with larger DRAM bandwidth, allowing us to achieve high resource utilization and operating frequency on large FPGAs. We then tackle the main internal bottleneck of our original MOMS architecture, where all requests are handled by a single shared MOMS, which provides high reuse but is susceptible to high contention. Therefore, we additionally explore private MOMSes, which group requests on a per-accelerator basis with no interaccelerator conflict, and two-level MOMSes, which combine the advantages of private and shared MOMSes.

With MOMSes now able to maximize both DRAM bandwidth and FPGA resource utilization, in **Chapter 7** we show how they can make FPGAs key enablers for large-scale graph processing in the cloud, achieving higher performance than CPUs and better energy and bandwidth efficiency than GPUs without the high NRE costs and long development times typical of ASICs. Finally, in **Chapter 8** we conclude and outline possible directions for future research.

2 Background and Related Work

After reviewing the main properties of the various on-chip memories available in modern FPGAs in Section 2.1, we provide some background on nonblocking caches in Section 2.2, on DRAM memories and controllers in Sections 2.3 and 2.4 respectively. We then provide some definitions and present the state-of-the-art of graph processing on CPUs, GPUs, FPGAs, and ASICs in Section 2.5 and conclude the chapter with a detailed presentation and characterization of our experimental setup in Section 2.6.

2.1 FPGA On-Chip Memory

Modern FPGAs have at least three types of on-chip memory: flip flops, LUTRAM, and block RAM (BRAM). Each bit of flip flop-based memory is exposed to the FPGA fabric, providing the highest flexibility in terms of number, type, and width of memory ports and the largest bandwidth. However, flip flop bits are the least abundant and some LUTs must be consumed to implement their access logic. LUTRAMs use LUTs to realize single-, dual-, or quad-port memories with medium depth (32-64 entries). However, they compete with combinational logic for LUTs. BRAMs are dedicated memory resources implemented as hard logic. They provide higher memory density than LUTRAMs and do not require any soft logic; however, they generally provide only two ports and are optimized for narrow and deep memory arrays (at least 512 entries). The most recent FPGA families offer an additional type of on-chip memory (UltraRAM/URAM [112] and eSRAM [43] in Xilinx and Intel FPGAs respectively) with even higher density and lower design flexibility than block RAMs—for example, Xilinx's URAM blocks can only be configured as 72-bit wide, 4,096-entry deep memories, while Intel's eSRAM has a fixed topology with 8 channels, each with up to 42 72-bit wide, 2,048 entry-deep banks, and have a minimum read latency of 10 cycles as opposed to 1-cycle of the finer memory blocks (M20K and MLABs). Therefore, the challenge is to use URAM/eSRAM as much as possible when available, followed by block RAM, LUTRAM, and flip flops.

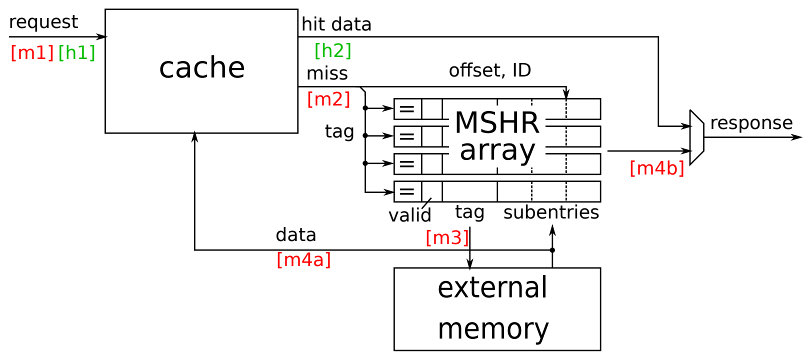


Figure 2.1 – Structure of a nonblocking cache. On a hit (steps [h1]-[h2]), it behaves just like any cache. On a miss (steps [m1] to [m4]), the miss address and source/ID is stored in an MSHR [m2]. Only on the first miss to a particular cache line, a memory request is additionally generated [m3]. When the cache line data is received from memory, it is stored in the cache [m4a] and used to respond to all its pending misses [m4b].

2.2 Nonblocking Caches

The first nonblocking (*lockup-free*) cache, proposed by Kroft in 1981 [56], introduced the idea of using miss status holding registers (MSHRs) to keep track of multiple in-flight misses. Farkas and Jouppi [31] evaluate a number of alternative miss handling architectures (MHA) for loads and observed that (1) nonblocking caches can reduce the miss stall cycles per instruction by a factor 4 to 10 compared to blocking caches, (2) that they are beneficial even for large cache sizes, and (3) that overlapping as many misses as possible allows processors to maximize the benefit provided by nonblocking caches.

Figure 2.1 shows the organization of a typical nonblocking cache based on explicitly-addressed MSHRs, which is among the architectures proposed by Farkas and Jouppi [31] and that today is a *de facto* standard in industry [37]. In addition to the cache array, a nonblocking cache contains an array of MSHRs, which keeps track of the in-flight misses. Each MSHR refers to one missing cache line and contains a valid bit, the tag of the cache line and one or more subentries to handle multiple misses to the same cache line. On the first miss of a cache line (a *primary* miss), the address of the cache line is sent to memory and stored in an MSHR; the offset of the requested word within the cache line, together with the request source/ID, is stored in a *subentry* for that MSHR. Subsequent misses to the same cache line (*secondary* misses) only require the allocation of a subentry on the same MSHR with no additional memory requests. When the missing cache line is received, it is both stored in the cache and used to serve all of its pending misses [31].

Tuck et al. [98] introduced a novel MHA for single processor cores with very large instruction windows. They propose a hierarchical MHA, with a small explicitly-addressed MSHR file for each L1 cache bank and a larger shared MSHR file. MSHRs are explicitly-addressed and shared MSHRs have more subentries than the dedicated ones. On a number of SPEC2000 benchmarks running on a 512-entry instruction window superscalar single-core processor, dedicated files with 16 MSHRs and 8 subentries and a shared file with 30 MSHRs and 32 subentries achieve speedups that are close to those provided by an unlimited MHA. However, we believe that a set of parallel accelerators is fundamentally different from a single-core processor even with a large instruction window for two reasons: a) parallel accelerators with, for instance, decoupled access/execution architectures [66, 23] could generate even more requests per cycle with no fundamental limitations on the total number of in-flight operations, and b) unlike multicore CPUs, requests to be merged can come from the same as well as a different accelerator, so it is important to have a shared MHA to maximize the merging opportunities. Our results throughout the thesis indeed show that, for parallel accelerators with massive MLP, thousands of MSHRs paired to a small cache array or even no cache at all can achieve similar or even better performance of larger caches with few MSHRs.

Gu and Chen [37] discussed the advantages of storing subentries in linked lists to accommodate workloads with different ratios of primary and secondary misses in GPUs. We independently and almost simultaneously proposed the same idea, which is discussed in detail in Section 3.1.2, as a fundamental ingredient to massively scale up the number of MSHRs. While our implementation is different as it is optimized for FPGAs and for a large number of MSHRs and subentries, we essentially reach the same conclusions concerning the advantages of arranging subentries in linked lists as opposed to statically allocating a fixed number of subentries to all MSHRs.

2.3 DRAM Structure and Operation

Dynamic random-access memories (DRAMs) use a single transistor-capacitor pair to store a bit of information, as shown in Figure 2.2. This makes DRAMs more area-efficient, and thus cheaper per bit, than SRAMs, where each bit usually requires a six-transistor (6T) cell. However, the capacitor is not perfect and its charge leaks over time. Therefore, each memory cell must be periodically read and rewritten, or *refreshed* [46]. We focus here on *synchronous* DRAMs and on read operations as currently MOMSes do not optimize writes.

Alongside the schematic of a single cell, Figure 2.2 shows the structure of a typical DRAM *bank*. Memory cells are organized in two-dimensional arrays, where each bit is identified by a *row* and a *column* address. Reading a bit first involves *precharging* the bit lines to $V_{dd}/2$, which is the average voltage between that of logic 0 and logic 1. Then, the row to be read is *activated*: the row address decoder raises the voltage on the wordline corresponding to the row address, which enables the access transistors of that row and connects each capacitor of the row to the respective bit line. This results in a small voltage change $\pm\delta$ on each bit line due to charge sharing between the cell capacitor and the bit line capacitance, where the sign of the voltage change depends on the bit that was stored on the capacitor. Such voltage changes are amplified by the *sense amplifiers* in the *row buffer*, which latch the resulting logic levels and restore the charge in each capacitor. Once a row is in the row buffer, individual bits can be accessed randomly with high throughput and low latency using column accesses, essentially using the row buffer as a cache. On the other hand, accessing a different row requires disabling the current word line, followed by a new precharge and activation; all of these operations require extra time during which no data transfers may occur from the DRAM array [46].

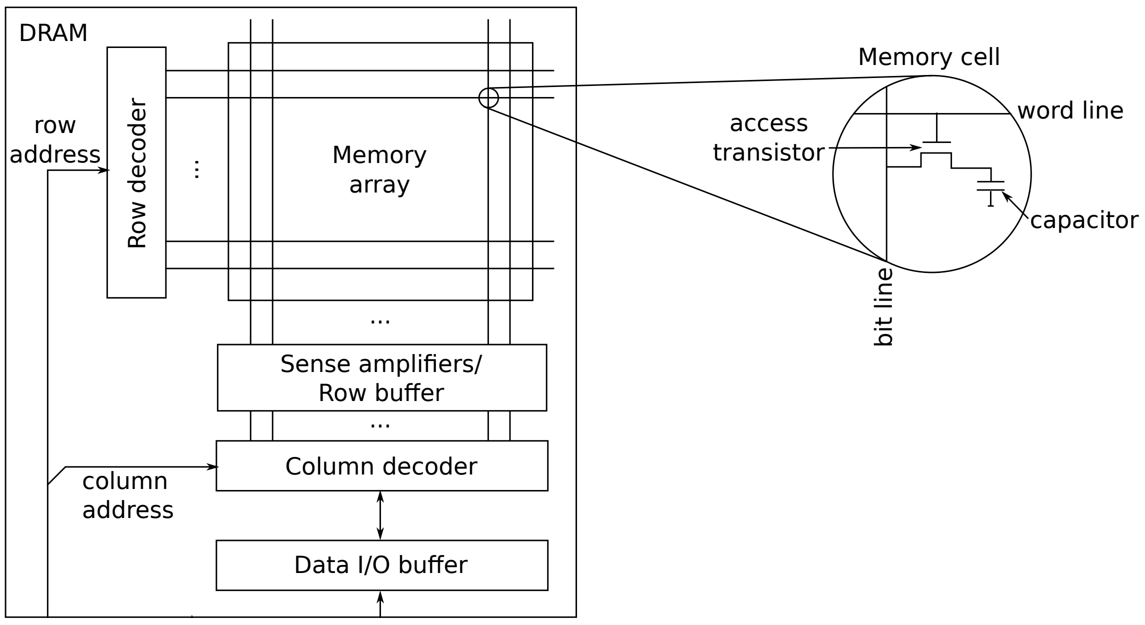


Figure 2.2 – Structure of a DRAM bank with a single memory array. One bit of data is stored at every intersection between bit (columns) and word (row) lines. A single row is first activated and latched by the sense amplifiers; once there, column accesses can be performed to read individual bits. Reprinted with modifications from the book of Jacob et al. [46].

To improve performance, DRAMs implement parallelism at different levels, summarized in Figure 2.3. Multiple arrays can either share address and control signals or operate more or less independently from each other. The first solution is typically used to increase throughput by reading multiple bits with each column access: this has low area overhead but coarsens the granularity of each column operation. Typical DRAMs also implement multiple banks, which is the entire structure shown in Figure 2.2 (possibly with multiple memory arrays sharing the address and control signals). Banks within a DRAM chip are identified by a *bank address* and while they share the other I/O lines (row/column address, data, and control), they can mostly operate independently from each other. This allows to pipeline memory operations: for example, one bank may service a column access on a row that is already activated while another bank is precharging. A chip with one or more banks can be organize in a *rank* to operate in lockstep and further increase the size of each transfer by concatenating the respective data lines while sharing all the other signals, similarly to having multiple arrays in a bank. Multiple ranks, each identified by a dedicated *chip select*, can coexist within the same *DRAM channel*, which is the highest level in the DRAM hierarchy. While the I/O pins of devices in a DRAM channel are either shared or related to each other (such as the data pins in a rank, which are obtained by concatenating the data I/Os from each chip), different DRAM channels have no common signals and are completely independent from each other [46].

The complexity of DRAMs is not restricted to topology but also on its interface as even the simplest memory operations, such as reading or writing some data, require multiple commands such as bank precharge, row activation, and column access, while also periodically refreshing

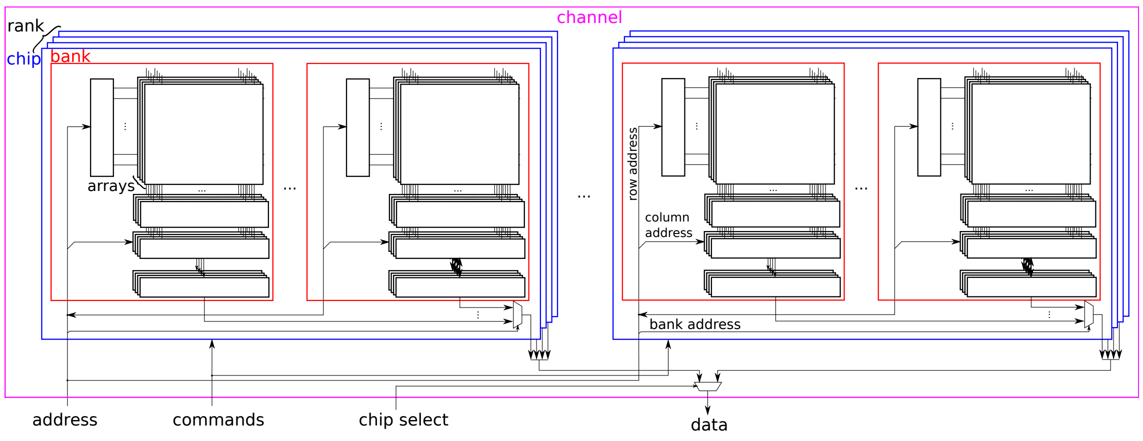


Figure 2.3 – Overview of the DRAM hierarchy together with the multiplexers that are used during read operations [46, 72]. Each *channel* has an independent address, command, and data bus and can be composed of one or multiple *ranks*, which are enabled based on the chip select signals. A rank consists of multiple *chips* that operate in lockstep; the data output of a rank is obtained by concatenating the output of each chip. Each chip contains multiple *banks*, addressed by the *bank address* portion of the address, which operate independently from each other. Each bank in turn typically contains multiple *arrays* so that each column access retrieves multiple bits.

the memory array. Such commands must obey tens of device-dependent timing constraints to ensure proper operation: for example, activation cannot be started until the bit lines reach the proper precharge voltage. Therefore, a *DRAM controller* is typically used to translate the simple memory commands generated by the CPU, GPU, or accelerator into a sequence of legal, and ideally efficient, DRAM commands [46]. Each DRAM controller is typically assigned to a single DRAM channel; modern CPUs [97, 9] and GPUs [77] use on-chip DRAM controllers while, on FPGAs, controllers can be either hardened [109] or in soft logic [108].

Fabrication processes for DRAM are optimized for density of memory cells, whose leakage must be low enough and capacitance large enough to ensure data retention and integrity [46]. Unlike in logic-optimized processes, high operating frequency is a secondary concern and indeed the DRAM core frequency has not increased significantly since the first *single data rate* (SDR) synchronous DRAMs standardized by JEDEC in the mid-90s. Memory bandwidth has been increased by instead 1) transferring a new data on both clock edges (*double data rate*, DDR) and 2) increasing the I/O clock frequency from the 100–200 MHz of the first DDR standard to the 667–1600 MHz of DDR4. To match the throughput of the DRAM core to that of the I/Os, all DDR standards use *prefetching*: each column access inside the DRAM is wider than the I/O data interface and the data read is serialized out in multiple transfers. The prefetch depth has been doubled at every DDR generation until DDR3, where it reached the value of eight, meaning that each column access results in eight transfers across four I/O clock cycles on both clock edges. Note that, on all DDR generations until DDR3, the minimum time between column commands (tCCD) is such that multiple column accesses can be perfectly pipelined as long as they are all *row hits* (i.e., they target a row that is currently active in one of the banks), leading to the ideal throughput of one transfer per clock edge. If the trend continued, DDR4 would have needed a prefetch depth of 16; however, considering that most DDR modules have a data width of 64 bit, this would have meant that the minimum access granularity (*burst length*) would have become 1,024 bits. On many processors, including x86 and ARM, this would have corresponded to two cache lines, potentially leading to large data and bandwidth wastage when memory accesses are irregular. Therefore, DDR4 maintained a prefetch depth of eight like DDR3 but introduced the concept of *bank group* instead. While DDR3 has eight fully independent banks, DDR4 has 16 banks organized in four bank groups: column accesses on a given bank group take longer than four I/O cycles and cannot be perfectly pipelined but column accesses on different bank groups can be partially overlapped and thus achieve ideal throughput [94]. This allows DDR4 to preserve the same access granularity as DDR3 at the expense of a higher complexity of the policies that the DRAM controller must implement to achieve high performance.

In general, prefetching has no significant drawbacks as long as the burst size does not exceed the minimum access granularity which, for the CPUs that make up the majority of the market, corresponds to a single 512-bit cache line. However, for application-specific accelerators that perform short irregular memory operations (such as in sparse linear algebra or graph processing), prefetching may cause significant data wastage if only a small fraction of each DRAM burst is used. The key insight behind our MOMS is to use as much of each DRAM burst as possible, even in those cases. We do so in a dynamic, transparent, and application-agnostic manner by maintaining a large pool of short accesses and serve as many of them as possible with every DRAM burst returned by the memory.

2.4 Reordering Memory Controllers

Due to the high complexity of DRAMs, generating a sequence of commands that is both legal and efficient for an arbitrary input memory operation sequence is an active field of research. Rixner et al. [82] first observed that processing memory operations in the same order as they are received is often suboptimal and does not properly exploit bank-level parallelism, as exemplified in Figure 2.4. Their first-ready, first-come, first-serve policy (FR-FCFS) prioritizes ready operations such as column accesses to an active row (row hits) over older operations that would require resources that are not available, such as a bank that is currently being precharged, or that would make some resources idle. FR-FCFS is now a de-facto standard implemented by most DRAM controllers [46], including those used in our experiments [109, 108]. A plethora of other scheduling policies have been proposed [87, 83, 44, 73, 70, 46]; for instance, they try to balance aggregated throughput and fairness in a multi-processor [54] or heterogeneous systems [99] or to maximize bandwidth under real-time latency guarantees [36]. 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 MOMS is application-agnostic but targets throughput-oriented systems that can trade more cycles on the miss path and a longer worst-case latency for greater bandwidth through deeper request reordering. In Chapter 4, we propose a mechanism to extend MSHRs to handle multiple consecutive cache lines, which will be requested as a burst. This can be seen as an implicit memory request reordering at

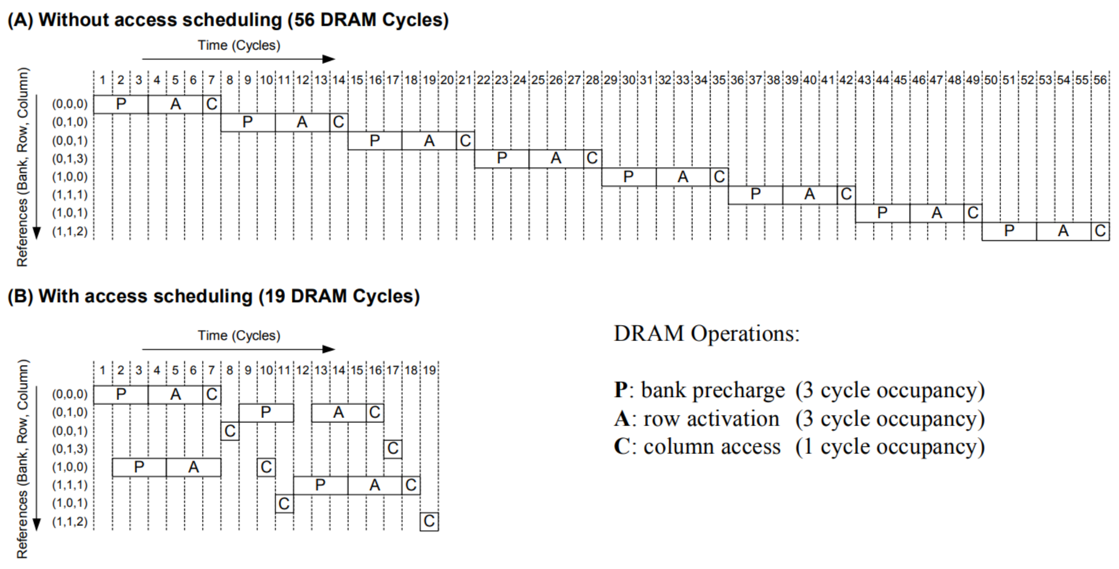

(A) Without access scheduling (56 DRAM Cycles)
Time (Cycles)
References (Bank, Row, Column)
(B) With access scheduling (19 DRAM Cycles)
DRAM Operations:
P: bank precharge (3 cycle occupancy)
A: row activation (3 cycle occupancy)
C: column access (1 cycle occupancy)

Figure 2.4 – Example of memory command schedule required to serve the trace shown on the two vertical axis, (A) with strictly in-order operation and (B) with memory command reordering. Reordering memory commands results in a 3× lower latency thanks to a better utilization of the DRAM resources. Reprinted from the publication of Rixner et al. [82].

the granularity of the number of cache lines handled by each MSHR (2–16) but across the tens of thousands of simultaneous misses handled by the MOMS. In Chapter 5, we propose a more explicit policy that groups single requests by DRAM row, effectively implementing a very aggressive row hit-first policy over the thousands of requests naturally exposed by the MOMS.

2.5 Graph Processing

A graph G consists of a node or vertex set V and an edge set E of sizes N and M respectively. We consider G to be a directed graph; undirected graphs can be handled by duplicating each edge. In addition, we focus here on graph algorithms that associate a value to every node and iteratively update them for a fixed number of iterations or until convergence.

Because graphs are the most effective data representation in a wealth of domains, including social networks [59, 81], drug discovery, [95], genomics [17], and robot navigation [16], high-performance graph processing is a very active area of research. While graph problems are usually embarrassingly parallel, the main challenge of graph processing is overcoming the memory bottleneck, which is particularly tight as memory access patterns are often irregular. Indeed, graph processing frameworks typically iterate sequentially over either vertices or edges (and are accordingly called either *vertex-* or *edge-centric*) but access the other set irregularly [28, 29]. In Chapter 7, we show how MOMS can make FPGAs an attractive platform for large-scale graph processing by tackling the memory bandwidth bottleneck with a lightweight preprocessing that has linear complexity with respect to the number of edges. In the remainder of this section, we review the most representative approaches that have been proposed on CPUs, GPUs, FPGAs, and ASICs.

On CPUs, GraphChi [60] is an out-of-core graph processing system that first introduced the concept of shards to confine random accesses to a smaller range that can be cached in main memory. X-Stream [85] introduced the edge-centric scatter-gather model, where edges are streamed and do not need to be sorted but only partitioned. Frameworks for in-core processing include GraphMat [93], Galois [75], and Totem [34], which also supports hybrid CPU-GPU systems. On a dual-socket Intel Xeon E5-2695 v3 with 28 cores, 240 W TDP, and 136 GB/s of memory bandwidth combined, Aasawat et al. [2] reported 1.3, 1.8, and 9.0 GTEPS for PageRank on RMAT-24 for Galois, GraphMat, and Totem respectively. We achieve 1.8 GTEPS with half the DRAM bandwidth and a 15× lower power. Both Galois and Totem only support graphs in CSR format, where edges are sorted by source node. During preprocessing, Totem also sorts edges by vertex degree. Our approach does not need any edge sorting but a faster linear-time partitioning. GPOP [62] is a cache-, work-, and memory-efficient framework that also does not require any edge sorting and achieves significant speedup over Ligra and GraphMat. When running PageRank, SSSP, and SCC on the RV and FR benchmarks presented in Table 7.2, our best architecture is 0.22–5.0× faster and 2.2–49× more energy efficient.

GPUs have a memory bandwidth (and a power budget) that is at least an order of magnitude larger than that of FPGAs; however, it is challenging to fit the irregular workload and memory accesses typical of graphs into the GPU SIMD execution model. One solution, adopted for example by CuSha [53], is to rely on offline preprocessing to balance the workload and group memory accesses, which can constitute a relevant overhead on graphs that are dynamic or used only a few times [69]. While Gunrock [101] shifts this overhead to runtime, Tigr [76] uses a lighter offline preprocessing to convert irregular graphs into equivalent, more regular ones. On a 21M- and a 59M-node graphs, Tigr achieves at most 10% speedup on PageRank compared to Gunrock, which is much lower than the 1.5–12× speedup that our system sports against Gunrock on the same application.

ASICs offer an order of magnitude higher clock rate, density, and energy efficiency than FPGAs [58] at the cost of significantly higher NRE costs and fabrication times. For example, Graphicionado [38] achieves 4.5 GTEPS for PageRank and 0.2 GTEPS for SSSP on the RV graph with a similar memory bandwidth as ours (we achieve 1.5 and 0.7 GTEPS respectively), while GraphDynS [116] achieves more than 85 GTEPS on RMAT-26 on an HBM whose bandwidth is only 8× larger than that of our DDR4. Both solutions have a clock that is 4–5× faster than ours and use significantly more on-chip memory than us: 64 MB and 32 MB compared to, at most, 9 MB.

On FPGA, FabGraph [88] represents the state-of-the-art of large-scale graph processing on a single FPGA that only needs a linear-complexity preprocessing. Its extension, FabGraph+ [89], focuses on optimizing the PCIe transfer when the FPGA DRAM cannot store the entire graph, a problem also tackled by FPGP [25] that is orthogonal to efficiently processing the graph once in its dedicated DRAM. ForeGraph [26] uses a very similar model as FabGraph but also

Algorithm 1 Sparse matrix-vector multiplication (SpMV)

```
1: for $r \leftarrow 0$ to ROWS $- 1$ do
2:     $out[r] \leftarrow 0$
3:     for $i \leftarrow idx[r]$ to $idx[r+1]$ do
4:         $out[r] \leftarrow out[r] + val[i] \times vect[col[i]]$
```

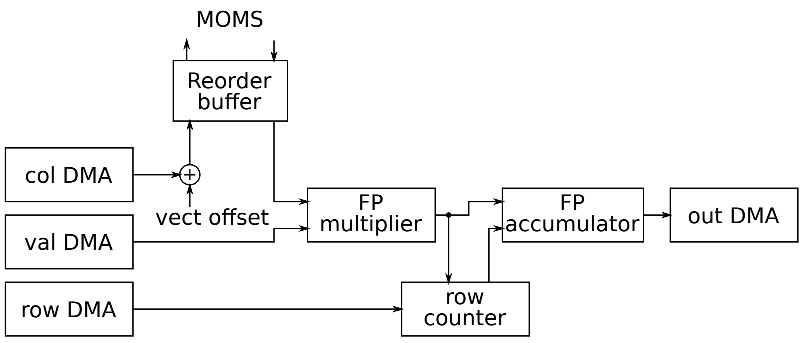


Figure 2.5 – Structure of our benchmark sparse matrix-vector multiplication accelerator. Xilinx AXI DMAs are used to stream all CSR vectors accessed sequentially. The values of the `col` array are used to compute the addresses of the vector elements that are retrieved through our memory controller.

supports multi-FPGA processing. HitGraph [118] outperforms our system on RMAT-24 but not on extremely sparse graphs like WT and, in addition, requires edges to be sorted by destination node. Except for FPGP [25], whose BFS performance on TW is 1.7× less bandwidth efficient than our more complex SSSP on the same graph, all of these works have only been tested in simulation and do not address the challenges related to multi-die partitioning that affect modern large FPGAs.

2.6 Experimental Setup

Our MOMSa is written in Chisel 3 and is fully parametric in terms of, e.g., number of inputs, banks, number and organization of MSHRs and subentries, cache size and associativity, input and output data size. Even though it has been evaluated on two Xilinx platforms (described more in detail in Section 2.6.2), it is written in platform-independent RTL. We compiled it using Vivado 2017.4 for the experiments on the ZC706 platform and Vivado 2019.1 for the AWS F1 FPGA. In the remainder of this section we will introduce the sparse matrix-vector accelerators and the matrices that we used as a benchmark (Section 2.6.1) and then present the FPGA boards where we ran our analysis, with special emphasis on the properties of the memory systems (external DDR memories and respective controllers; Section 2.6.2) which we benchmark in Section 2.6.4.

2.6.1 SpMV Accelerators

As a benchmark, we implemented a simple accelerator for sparse matrix-vector multiplication (SpMV), an important kernel in a broad range of scientific applications [11] and to which many

Table 2.1 – Properties of the benchmark matrices we used. We found the stack percentiles [20] to be a better predictor of performance than, e.g., sparsity. All vectors are of single-precision floating point values. The number of columns corresponds to the vector size divided by 4 bytes and, except for pds-80 and rail4284, it corresponds to the number of rows.

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

sparse graphs algorithms can be mapped [51]. Moreover, SpMV can easily generate a wide range of access patterns depending on the matrix sparsity pattern. Our accelerator, shown in Figure 2.5, is an almost direct implementation of Algorithm 1 for SpMV of a CSR-encoded sparse matrix; we do not include any SpMV-specific optimizations as our controller aims for a generic architectural solution for any applications with irregular memory access pattern. Indices are 32-bit unsigned integers while values are single-precision floating point values. All CSR vectors, accessed sequentially, are provided via AXI4-Stream through Xilinx AXI DMA IPs; the dense vector, accessed randomly, is read through an AXI4-MM port connected to our memory controller. The 8192-entry reorder buffer provides the vector values to the multiply-accumulation pipeline, which is based on floating-point Xilinx IPs. We use the index vector to clear the accumulator every time a new row begins, and the output vector is streamed to DRAM through a DMA. Each accelerator can process one non-zero matrix element (NZ) per cycle; we parallelize the SpMV by interleaving rows across multiple accelerators.

Table 2.1 shows the properties of our benchmark matrices, which are essentially the largest benchmarks used in prior work on SpMV [11]. All benchmarks are available on SuiteSparse [27]. We use the stack distance, introduced in Section 1.3, to characterize the regularity of the access pattern to the dense vector.

2.6.2 FPGA Boards and Memories Specifications

Table 2.2 and 2.3 shows the main properties of the FPGAs and the memory systems used in our experiments. The ZC706 is an embedded board that contains a Zynq-7000 SoC with an FPGA and two ARM cores. The SoC is connected to 1 GB of DDR3 on the processing sys-

Table 2.2 – Specifications of the FPGAs used in our experiments.

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

Table 2.3 – Specifications of the external memory systems used in our experiments. Section 2.6.4 provides more information on the peak bandwidth measurements.

Platform	ZC706		AWS F1
Memory controller	PS	PL	
Memory technology and speed	DDR3-1066	DDR3-1600	DDR4-2133
Total size (GiB)	1	1	64
DDR IO data width (bits)	32	64	64
Memory channels	1	1	4
Controller ports	5 (4 HP, 1 ACP)	1	4 (1 per channel)
Controller data width (bits)	64	512	512
Controller clock frequency (MHz)	150	200	250
Peak measured bandwidth (single accesses, GiB/s)	0.93	8.3	29.8
Peak measured bandwidth (any burst length, GiB/s)	3.9	11.7	55.7

tem (PS) side through the ARM hardened memory controller and 1 GB of DDR3 on the programmable logic (PL) side. 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 [108], which exposes a single, 512-bit wide port that sends full DDR3 eight-beat bursts. We measured a peak bandwidth of 11.7 GiB/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 [109], although our measurements showed that the four HP ports running at 150 MHz are enough to saturate its 3.9 GiB/s bandwidth.

Both memory controllers use bank interleaving [46] and implement memory access reordering [109, 108]. On the PL controller, requests targeting the same bank and row must be within eight or less requests apart in order to inhibit bank precharge [108]. 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 [109]. On the other hand, the FPGA system available on the Amazon AWS F1 instances consists of a Virtex UltraScale+ FPGA connected to four DDR4-2133 memories controlled by Xilinx MIG controllers beyond an undocumented memory system in the encrypted AWS shell [5]. Because the maximum number of outstanding reads per channel is limited to about half of the average number of cycles of latency, the quoted peak bandwidth of 55.7 GiB/s can only be achieved using bursts of length two or larger; if only single requests are used, the maximum measured bandwidth drops to 29.8 GiB/s.

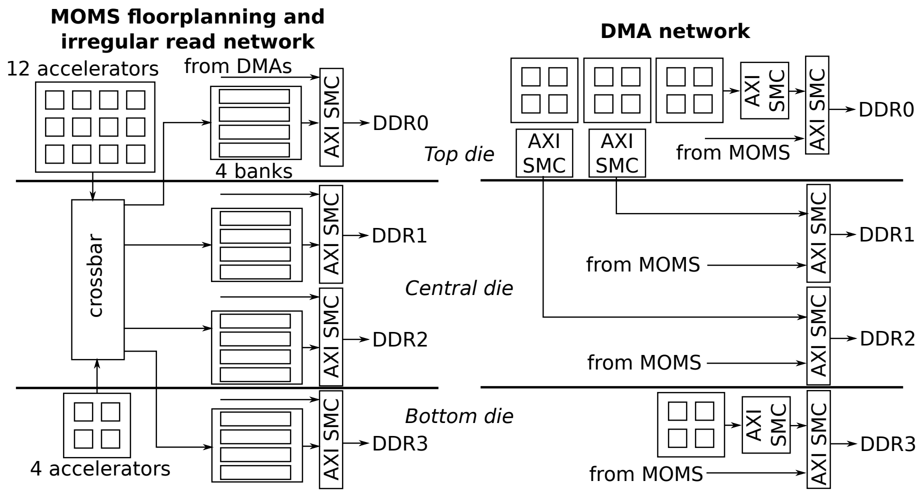


Figure 2.6 – Multi-die partitioning of SpMV accelerator and MOMS on the AWS F1 FPGA. The thick lines represent the die boundaries. For the sake of clarity, the irregular read and DMA networks are shown separately even though they are implemented simultaneously side by side. Each memory channel is shared among four MOMS banks and the DMAs of four accelerators using an AXI SmartConnect.

2.6.3 Top-Level System Organization

On the ZC706 board, we consider two different configurations, which we take as representative of realistic use cases in commercial FPGA systems. In the *PL system*, the dense vector is stored in the PL DDR while sequential vectors are read from the PS memory; the opposite is done in the *PS system*. In the PL system, it is the highest performing memory, exposed through a single, wide port, that is accessed irregularly. Since the PL DDR is often the system bottleneck due to the irregular accesses, we maximize its bandwidth by operating at 200 MHz. Ultimately, the system throughput is limited to ≈2.4 multiply-accumulations (MACC) per cycle by the bandwidth of the PS memory that hosts the sequential vectors. Therefore, four accelerators and four banks are enough to saturate it. On the PS system, the sequential accesses on the PL DDR allow up to ≈8 MACC/cycle, while the PS DDR limits the throughput to ≈6.5 (150 MHz) or ≈4.9 (200 MHz) MACC/cycle if each 32-bit word returned by the PS DDR is used exactly once (which, we found, is often an optimistic assumption). Since the performance is always limited by the PS DRAM 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.

Compared to the ZC706 FPGA, the AWS F1 FPGA contains 5.4× more CLBs (LUTs and FFs), 4.0× more BRAM blocks, and 33.8 MiB of URAM. In addition, the external memory bandwidth is 3.5× larger that of the ZC706. However, 25% of those resources are locked in FPGA regions reserved to the AWS shell and thus unavailable to the designer. The remaining 75% are spread unevenly among three dices (or Super Logic Regions in Xilinx's terminology) as only the central and bottom dices are partially occupied by the shell. The DDR4 controllers are also spread among dices, with the top and bottom die hosting one controller each and two controllers in the central die [6]. Because inter-die interconnects (Xilinx's Super Long Lines or SLLs) are particularly scarce and slow, achieving high resource utilization on multi-die devices is especially challenging and needs to be explicitly taken care of during system design. In Chapter 6 we illustrate how the MOMS modules and the accelerators have been assigned to memory controllers and dices, as well as how we handled inter-die crossings, in order to achieve high performance and resource utilization even on high-end multi-die FPGAs such as the one available on the AWS F1 instances.

As discussed in Section 2.6.2, the external memory bandwidth of the AWS F1 board is 3.5× larger than that of the ZC706. Considering that the four memory channels are symmetric, we do not artificially introduce any asymmetry and share all of them between DMAs and MOMS. Since each MACC requires at least 12 bytes—one 32-bit value from each of the vectors `val`, `col`, and `vec`, neglecting the 8 bytes per row of `row` and `out`—the maximum theoretical performance with a DDR4 bandwidth of 55.7 GiB/s is of 19.9 MACC/cycle at the shell frequency of 250 MHz. However, even though the AWS F1 FPGA offers more resources than the ZC706 one, they are harder to exploit as they are spread unevenly among three dices and about 25% of them are locked in FPGA regions reserved to the AWS shell. The DDR4 controllers are also spread among dices, with the top and bottom die hosting one controller each and two controllers in the central die [6]. In practice, achieving high performance and resource utilization requires to (1) spread the logic as uniformly as possible among dices while (2) minimizing the number of die crossings, under the constraints that (a) the top die has about 60% more resources than the other two and (b) the central die has two memory controllers while the others have only one. In addition, the sequential accesses performed by the DMAs (Figure 2.5) should be as balanced as possible among memory channels.

Figure 2.6 shows the system organization that allowed the implementation of 16 accelerators and 16 banks running at 250 MHz. This is the highest performing system that we could implement, which makes our experimental setup on the AWS F1 FPGA resource-bound—mostly due to the AXI DMAs and fixed infrastructure, as discussed in Section 3.3.5—unlike the ZC706 that was bandwidth-bound. We assign four accelerators' DMAs to each of the four memory channels; as a general rule, we generally assigned accelerators to the same die as the respective memory channel, except for those that would end up in the central die, which have been moved to the least congested top die. We discuss the generic multi-die guidelines that we elaborated for the MOMS, as well as how we handled the inter-die crossings, in Section 6.1.

2.6.4 Platform Characterization

We characterize the three memory systems presented in Table 2.3 using strided access patterns, whose properties are summarized in Figure 2.8. For a controller data width or word size W, the burst length B defines the number of consecutive words that are read with every access. Since all of our controllers expose an AXI interface, W corresponds to the size of the AXI RDATA signal whereas we set the AXI ARLEN as $B-1$. The stride S represents the difference between start addresses of consecutive bursts in words of size W. Setting $S = B$ results in a sequential pattern without gaps, which is the pattern for which memory controllers are generally optimized

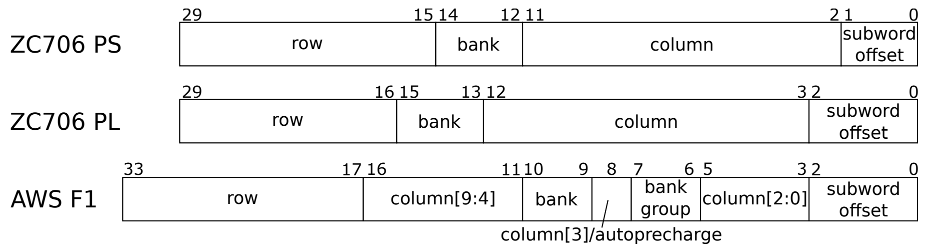


Figure 2.7 – Address mapping in our experimental platforms. We found that the row/bank/column interleaving scheme generally provides the highest performance on our applications; the AWS F1 mapping is set in the AWS shell and cannot be changed.

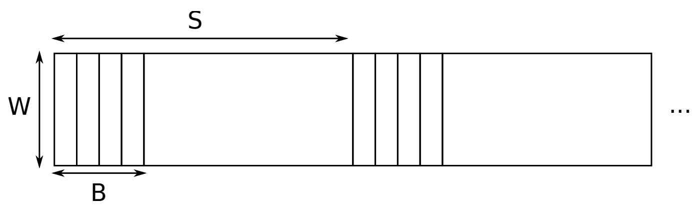


Figure 2.8 – Parameters that define the strided access pattern that we used to characterize the memory systems in our experimental setups. It consists of bursts of B words of size W, which correspond to the controller data width in Table 2.3, each starting at a distance of S words from the beginning of the previous burst.

and through which we therefore expect to reveal the highest achievable throughput for a given B. Increasing the stride produces a non-sequential and yet still intuitive pattern that provides information on the bandwidth degradation as the access sequence deviates from the ideal one.

Figure 2.9a shows the bandwidth of the ZC706 PL memory controller for sequential access patterns ($S = B$) as a function of B. Single accesses with $B = 1$ can only achieve 70% of the theoretical bandwidth: this is due to the AXI SmartConnect between the memory controller and the accelerator which limits the number of outstanding reads to 33 while the average memory latency, at the controller frequency of 200 MHz, is of 45 cycles. As a result, the pipeline between accelerator and memory can never be occupied for more than 73% of the cycles. On the other hand, even bursts of size two are enough to saturate the bandwidth, achieving 97% of the theoretical throughput (11.7 GiB/s or 12.5 GB/s), as 23 requests are now enough to fully utilize the memory pipeline.

Figure 2.9b shows the bandwidth as a function of the stride for $B = 1$ to $B = 4$. For $B \leq 2$, the bandwidth remains stable to the highest value up to $S = 64$ which, given $W = 64$ bytes, corresponds to 4 kiB. Considering the row/bank/column address mapping shown in Figure 2.7, this means that the memory controller can completely hide the activate and precharge overheads as long as there are at least two accesses per (row, bank) pair. For $B = 4$, the high-performance regime extends up to $S = 256$ since every access carries more data that is fetched sequentially at the highest throughput. For larger strides, the bandwidth gradually decreases until saturating at its lowest value for $S \geq 1024$, corresponding to 64 kiB, where only one bank is used and every access is a row conflict. The gap between the two plateaus at low and high S is 7×, 5.4×, and

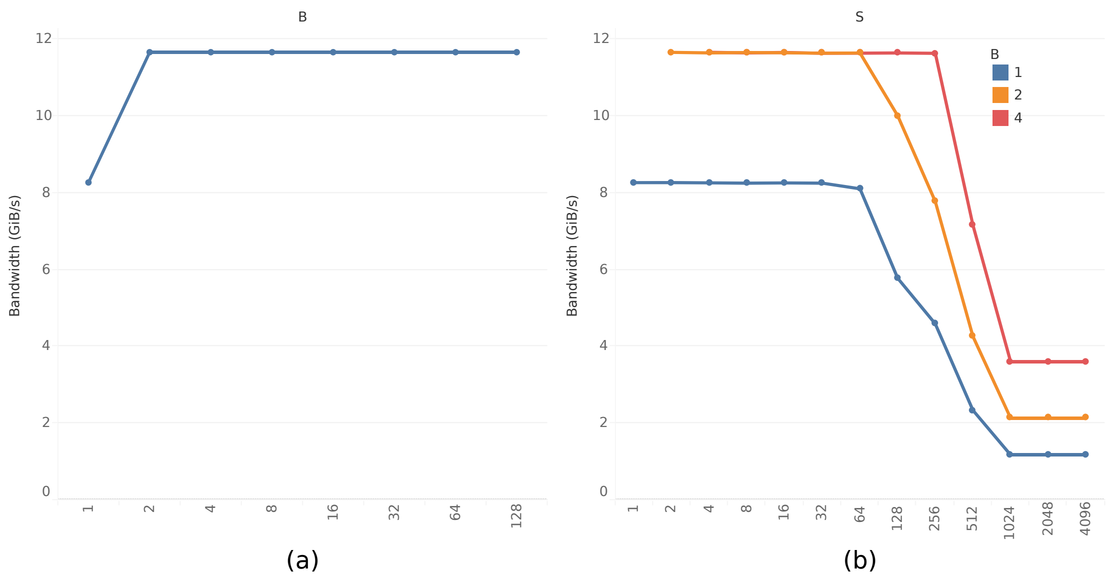


Figure 2.9 – Bandwidth of the ZC706 PL memory system (a) for sequential accesses, as a function of B and (b) for strided accesses, as a function of S. Single accesses can only achieve 70% of the peak bandwidth due to a limitation on the number of outstanding reads. For a given B, the bandwidth is maximum for $S \leq 64$ and minimum for $S \geq 1024$ due to bank parallelism and row conflicts.

3.2× for $B = 1$, $B = 2$, and $B = 4$ respectively and represents the bandwidth variability that can be expected depending on the access pattern.

The multi-ported ZC706 PS memory controller has one more degree of freedom compared to the PL one: the number of ports in use, from one to four. As a result, since all ports share the same memory space on the same physical DDR3 controller and memory, the measured bandwidth is also affected by inter-port conflicts. We set the same S and B on all ports and the results appeared to be the least noisy when the address streams on the four ports started from the same offset. Figure 2.10a shows the sequential bandwidth as a function of B and parametrized by the number of input ports in use. Each 64-bit port has a theoretical bandwidth of 1.14 GiB/s (1.20 GB/s) at 150 MHz, which is essentially saturated when up to three ports are used, whereas on four ports the bottleneck becomes the 3.97 GiB/s (4.26 GB/s) DDR3-1066 bandwidth. The minimum B that achieved peak bandwidth is $B = 4$: indeed, when $B < 4$, the burst length is smaller than the 256-bit DDR3 bursts, which suggests that the PS DDR3 controller does not consider using the same DDR3 burst to serve multiple AXI requests even when they are sequential. The bandwidth versus stride curves in Figure 2.10b when all four input ports are used are more noisy than the corresponding curves for the PL controller due to inter-port conflicts. Nevertheless, the curves become monotonically decreasing starting from $S = 256$ which, given the PS address mapping shown in Figure 2.7, correspond to 2 kiB or half the size of a DRAM row, as it was the case for the PL controller. The bandwidth reaches its minimum for $S = 4096$ and then stabilizes for $S \geq 8192$; since all accesses for $S \geq 4096$ are row conflicts and should be served at an equally low performance irrespective of S, and since this is the only memory system

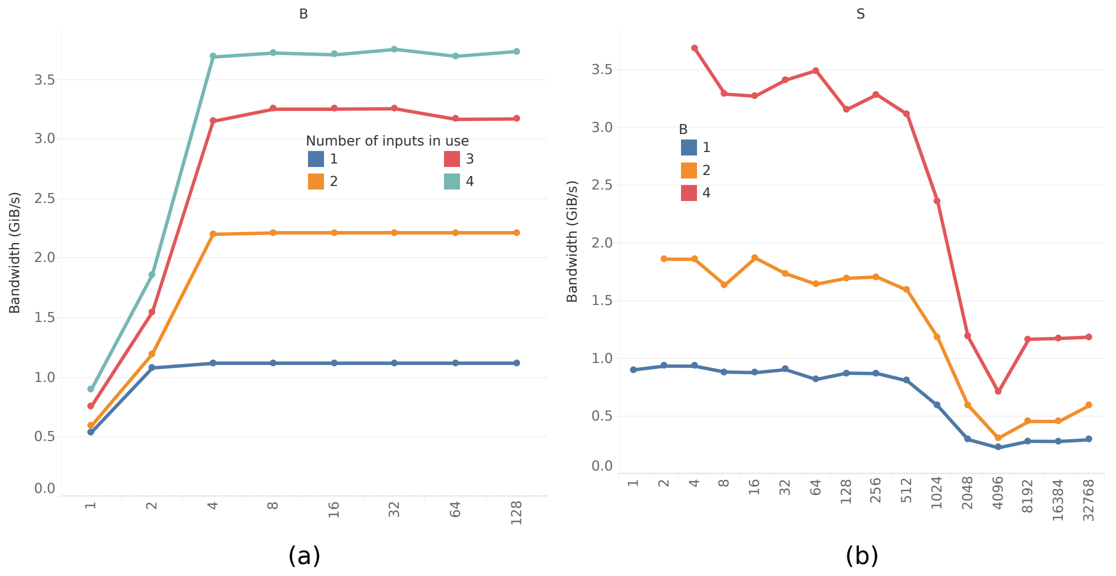
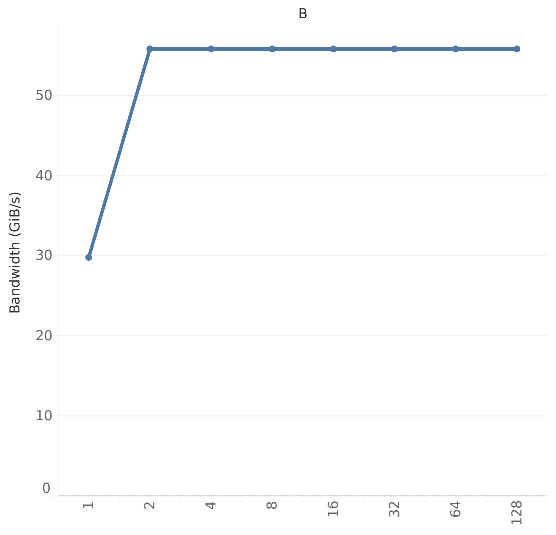


Figure 2.10 – Bandwidth of the ZC706 PS memory system (a) for sequential accesses, as a function of B and of the number of active inputs and (b) for strided accesses from four inputs, as a function of S. The minimum B that achieves peak bandwidth is 4, which corresponds to a full DDR3 burst. While the bandwidth versus stride curves are particularly noisy due to inter-port conflicts, we can still identify a high-performance regime for $S \leq 256$ and a low-performance one for $S \geq 4096$.

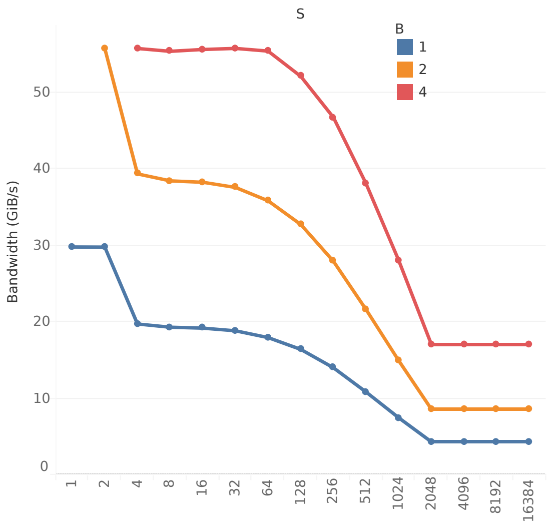
where we observed such a non-monotonous trend, we can only identify inter-port conflicts as a possible explanation for the minimum at $S = 4096$.

The AWS F1 memory system is also multi-ported but, unlike the ZC706 PS memory, each port is connected to an independent memory space and DDR4 channel, meaning that there are no inter-port conflicts. Of the four DDR4 controllers, three are in the user logic and one is in the AWS shell [6]. On all the characterization points, we measured a 5% lower bandwidth from the shell controller compared to the others; however, in our design, we will consider them symmetric as we believe that any possible gain that could be achieved by breaking the symmetry is unlikely to justify the increased design complexity given the minor performance mismatch. Considering that the bandwidth trends with respect to B and S are the same among all controllers, we report here the aggregate bandwidth that we measured across the four memory controllers altogether.

Figure 2.11 shows the bandwidth of the AWS F1 memory system. Like the ZC706 PL, single requests are penalized due to a restriction in the number of outstanding operations, which limits the maximum throughput to an even lower value of 50% of the theoretical one. On the other hand, any $B > 1$ achieves the same peak throughput of 55.7 GiB/s (59.8 GB/s), which corresponds to 93% of the maximum bandwidth of the AXI interfaces of the DDR controllers at 250 MHz, which is slightly lower than theoretical bandwidth of four DDR4-2133 memories. In terms of bandwidth as a function of the stride, shown in Figure 2.11, the degradation starts at $S = 4$ for $B \leq 2$, corresponding to 256 bytes, which is much lower than in the ZC706 memory systems. This is due to the AWS F1 address mapping shown in Figure 2.7: $S > 2$ results in all accessing targeting the same bank group, which results in suboptimal performance. In addition,



(a)



(b)

Figure 2.11 – Bandwidth of the AWS F1 memory system (a) for sequential accesses, as a function of B and (b) for strided accesses, as a function of S. A limitation on the number of outstanding reads penalizes single accesses even more than on the ZC706 PL memory, setting the performance limit to 50% of the peak bandwidth. The transition region between the high-performance and the low-performance The bandwidth as a function of S declines earlier and more gradually than on the ZC706 memory systems due to the more complex address mapping, shown in Figure 2.7.

$S = 4$ toggles bit 8, which is mapped to the autoprecharge signal, resulting in banks precharging and activating the same row every other access. Strides larger than 16 always target the same bank and, as S grows further, row conflicts become more and more common until, for $S \geq 4096$, all accesses are row conflicts. On the other hand, with $B \geq 4$, each access is long enough to mitigate the negative impact of these phenomena.

3 From Tens to Tens of Thousands Outstanding Misses

In this chapter, we delve in the details of how to scale up dramatically the number of MSHRs and subentries to turn a nonblocking cache into a MOMS, with a special focus on doing this efficiently on FPGAs. This is based on the properties and availability of the different on-chip memory resources provided by modern FPGAs which were discussed in Section 2.1. After introducing the key ideas at the microarchitectural level in Section 3.1, we discuss more practical implementation choices in Section 3.2 and present detailed experimental results in Section 3.3.

3.1 Key Ideas

Increasing the maximum number of outstanding misses in a nonblocking cache based on explicitly-addressed MSHRs (introduced in Section 2.2) requires scaling up both the number of MSHRs and of subentries, which handle primary and secondary misses respectively. This section presents the main intuitions behind our scaling approach.

3.1.1 Scalable MSHR Lookup and Storage

For each additional MSHR, the memory system can handle one more primary miss without stalling; similarly, each additional subentry allows servicing an extra secondary miss with no added traffic to the external memory. Each MSHR has modest storage requirements: ~20-30 bits for the cache line tag and its valid bit, plus ~10-20 bits for offset and request ID for each of the ~4-8 subentries. This is significantly smaller than a 512-bit cache line with its tag. Therefore, within a given on-chip memory budget, bandwidth-bound applications with irregular memory access patterns could benefit more from an increase of the number of MSHRs or subentries, which increase MLP, rather than from an expansion of the cache. In practice, however, scaling up the fully associative MSHR array (Figure 3.1a) also requires additional comparators and a wider multiplexer, which increase area and hurt the critical path. Moreover, on FPGA, associative

This chapter is based on the work published at the *27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2019 [13] and on *Transactions on Reconfigurable Technology and Systems*, 2021 [15].

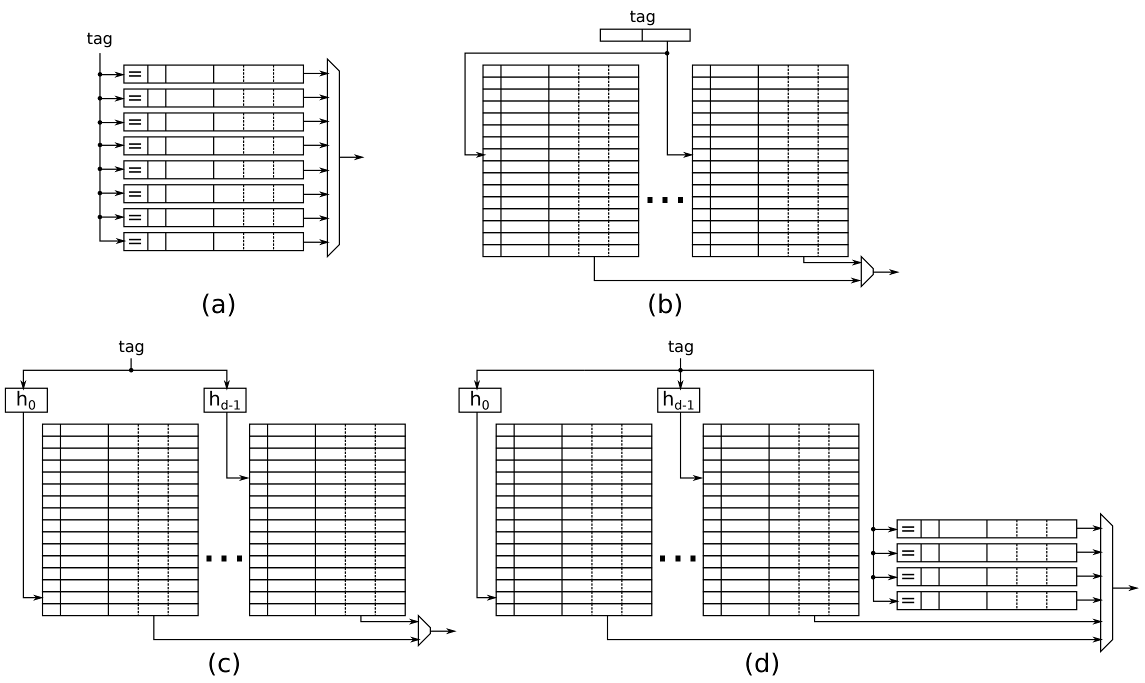


Figure 3.1 – MSHR-rich architectures for FPGAs. Because of the associative lookup, a traditional architecture (a) does not scale beyond a few MSHRs and MSHRs can only be mapped to flip flops. Using a set-associative memory with a single hash function (b) allows MSHRs to be mapped to block RAM but stalling on every collision results in low load factors. Cuckoo hashing (c) reduces the probability of collision and a stash (d) allows collisions to be handled in the background when the unit is idle.

MSHRs can be mapped efficiently only to the scarce flip flops; LUTRAM- or block RAM-based CAM implementations require 20–300 memory bits per CAM bit [45].

A set-associative MSHR memory (Figure 3.1b), indexed by the lowest significant bits of the tag, can be efficiently mapped to block RAM and, as long as there are no collisions, lookups, insertions, and deletions can be performed in a single step. Stalling is the simplest collision handling mechanism; however, we will show in Section 3.3.2 that this strongly limits the maximum load factor. Using linear probing would result in *expected* constant time lookup, insertion, and deletion and, whenever any operation cannot be completed in a single step, incoming misses must be stalled, hurting throughput.

To overcome these limitations, we propose to store MSHRs using *cuckoo hashing* (Figure 3.1c). Cuckoo hashing uses d hash tables and d hash functions $h_0, \ldots, h_{d-1}$; each key x can be stored in any hash table in bucket $H_i[h_i(x)]$. Lookups and deletions require *worst case* constant time: both involve one lookup per hash table, plus one update for deletions. For insertions, key x can be inserted in any hash table whose bucket $h_i(x)$ is empty. If all possible locations $H_i[h_i(x)]$ are occupied, a *collision* occurs: the new key x displaces an existing entry to one of its alternative locations. If all possible buckets of the displaced entry are also occupied, the process is repeated recursively until an entry can be inserted into an empty bucket. This means that insertions can still require more than one operation, during which no other misses can be handled. Expected

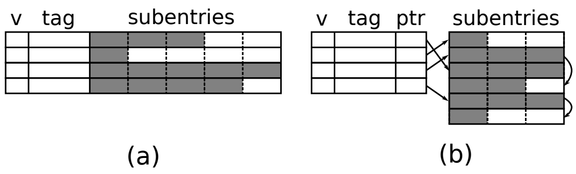


Figure 3.2 – Subentry organization in memory. Allocating a fixed number of subentries to every MSHR (a) results in a difficult tradeoff between a low maximum load factor and a high probability of stall, especially if there is a large variation in the number of secondary misses per cache line. Using a separate buffer to store blocks of subentries organized as linked lists (b) provides greater flexibility at a modest cost.

amortized insertion time is constant as long as the load factor is bounded; the bound is 50% for $d = 2$ and grows very quickly with d [32]. To reduce the average insertion time, Kirsch et al. proposed to temporarily store displaced entries in a small content-searchable queue (*stash*) [55] (Figure 3.1d). As soon as the input interface is idle, the module tries to insert the oldest entry from the stash; if this results in a collision, another entry from a different hash table is moved to the stash. By doing so, entry reinsertion effectively happens in the background without slowing down incoming requests; incoming allocations are stalled only when the stash gets full.

3.1.2 Flexible Subentry Storage

For their explicitly addressed MSHR architecture, Farkas and Jouppi propose to use a fixed number of subentry slots per MSHR (Figure 3.2a) and to stall the miss handling architecture whenever all slots of an MSHR are used. However, waiting for the specific MSHR that is full to be deallocated may take a long time, during which the nonblocking cache may miss opportunities for merging requests to in-flight cache lines to serve them with no extra memory bandwidth cost. Increasing the number of slots per MSHR would reduce the probability of stall at the expense of an increase in area or, in other words, a decrease in load factor due to increased internal fragmentation. To mitigate these drawbacks, we propose a hybrid approach (Figure 3.2b): we store subentries in a separate buffer and we dynamically allocate blocks of subentries to each MSHR. Specifically, the subentry buffer, mapped to block RAM, contains multiple subentry rows, each comprising N_s subentry slots. Each MSHR is initially assigned one subentry row; whenever a row gets full, an additional row is allocated for that MSHR. Subentry rows are logically organized as a linked list: the head pointer is stored in the MSHR buffer and each subentry row contains a field for the pointer to the next row.

3.2 Detailed Architecture

Figure 3.3 shows the top-level view of our miss-optimized memory system. To simplify the design and to maximize the scope for memory access optimization, our MOMS can return responses out of order, which is not unusual among high performance memory systems [41, 40]. Therefore, requests must be tagged with an ID, which will be used to match it with the corresponding response. Optional reorder buffers can be instantiated for compatibility with accelerators that

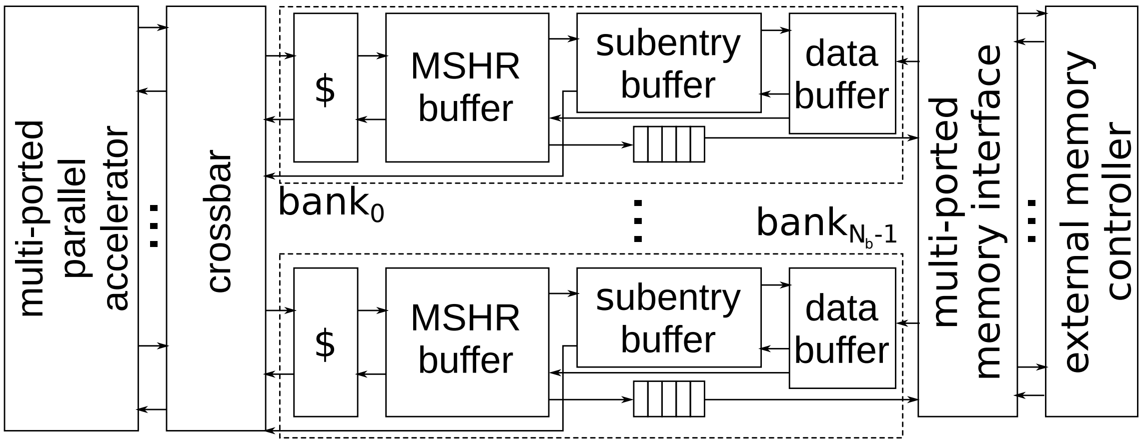

multi-ported parallel accelerator
crossbar
$
MSHR buffer
subentry buffer
data buffer
bank$_0$
bank$_{N_b-1}$
multi-ported memory interface
external memory controller

Figure 3.3 – Top-level view of our MOMS. A crossbar steers memory requests from N_i accelerators to N_b banks according to their address. Each bank consists of a cache, an MSHR buffer, a subentry buffer, and a data buffer. The multi-ported memory interface multiplexes each memory interface among banks.

expect in-order responses. The reorder buffers append an ID to each memory request, which will be used to match the corresponding response from the MOMS and to return the responses in order. Requests received from each of the N_i input channels are redistributed across N_b banks by means of a crossbar. We use a multi-banked structure in order to handle multiple requests and responses per cycle. We maximize workload balancing among banks by interleaving requests among them in the finest possible way: requests pertaining to consecutive cache lines are served by different banks. The crossbar also appends additional bits to the requests' ID to allow responses to be routed to the respective source port. Each bank consists of a set-associative cache, an MSHR buffer, a subentry buffer, and a data buffer. Data for requests that hit in the cache are immediately returned to the crossbar, while misses are handled and stored by the MSHR and subentry buffer. Since each occupied MSHR generates a memory request, the queue depth corresponds to the size of the MSHR buffer. This ensures that the queue will not become the bottleneck and allow the MSHR buffer to be always fully utilized irrespective of the amount of buffering available in the external memory.

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

3.2.1 MSHR Buffer

For the MSHR buffer, shown in Figure 3.4, we use one block RAM per hash table, with the address of the cache line (tag) as key. We use universal hash functions in the form $h_a(x) =$

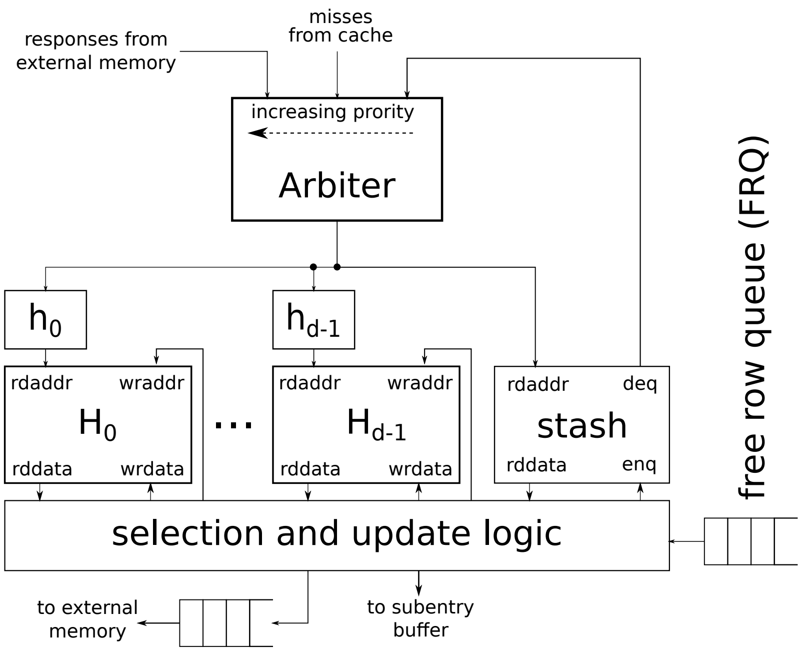


Figure 3.4 – Block diagram of the MSHR buffer. We multiplex the cuckoo hashing pipeline between memory responses, new misses, and entries that have been displaced due to a conflict in one of the cuckoo hash tables.

$(ax \bmod 2^{w_t}) \operatorname{div} 2^{w_t - w_M}$ with w_t being the number of bits of the tag, $w_M = \log_2(M)$ where M is the number of buckets per hash table, and a is a random positive odd integer with $a < 2^w_t$ [106]. Each bucket contains a valid bit, the tag of the missing cache line, and the address of the first subentry row in the subentry buffer as described in Section 3.1.2. The stash is a content-associative memory made of flip-flops. To integrate the stash in the pipeline, we include the stash entries among the locations that are searched during lookups or that can be deallocated when a response is received. Because most operations require a read-modify-write on the dual-ported hash table BRAMs, we use a single pipeline that is multiplexed, with decreasing priority, among memory responses, new misses, and entries from the stash. We will show in Section 3.2.4 that this is generally an acceptable compromise.

3.2.2 Subentry Buffer

Figure 3.5 shows implementation and operation of the subentry buffer. A subentry consists of an (ID, offset) pair; a subentry row contains a) N_s subentry slots, b) the number of allocated subentries, and c) a pointer to the next subentry row with its valid bit. To allocate a subentry, the first subentry row is retrieved from the buffer. If the row is not full (1), the new entry is appended and the row is written back to the buffer. If the row is full (2), a new row must also be allocated. We use a FIFO (free row queue, FRQ) to store the addresses of the empty rows, and allocating a row simply means extracting the first element of the FRQ.

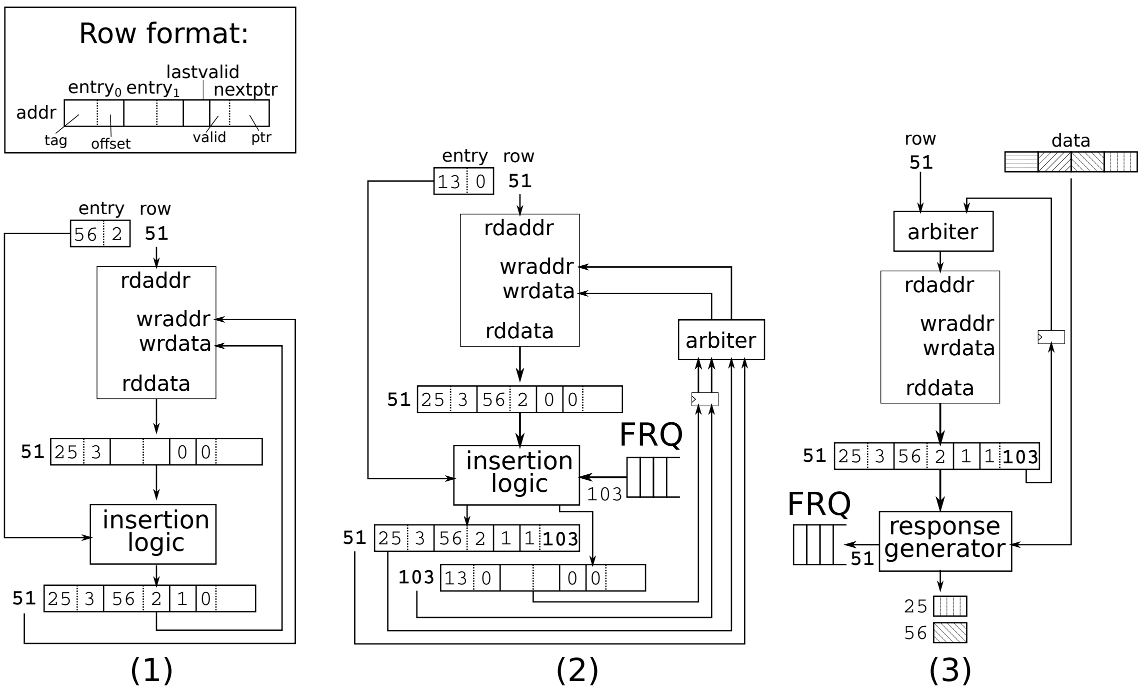


Figure 3.5 – Block diagram and operation of the subentry buffer. For requests, the subentry buffer receives ID, offset, and the address of the first subentry row (head row) from the respective MSHR. The head row is firstly retrieved from the buffer. If it is not full (1), the row is updated with the new entry and written back to the buffer. If the row is full (2), the new entry is inserted in a new row, whose address is stored in the previous row. When a response is received (3), all subentries are retrieved by traversing the subentry row list. After all subentries have been forwarded to the response generator, the row is deallocated by pushing its address to the free row queue.

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

When a cache line is received (3), the corresponding MSHR is deallocated from the MSHR buffer and its subentry rows retrieved from the buffer. The response generator parses the subentry rows, retrieves the data from the data buffer, and emits one response per allocated subentry. The row is then recycled by inserting its address into the FRQ and the process is repeated for the entire linked list of rows.

3.2.3 Data Buffer

The data buffer stores the data received from memory until all of its misses have been served. Because responses are treated in-order inside the pipeline, the data buffer can be implemented as a simple circular buffer in LUTRAM or block RAM. After storing the response in the circular buffer, the module forwards its base address (pointer) and tag to the pipeline. The MSHR buffer will use the tag to deallocate the MSHR pertaining to the response and retrieve its subentries; the base address is then used by the subentry buffer to generate the responses as shown in

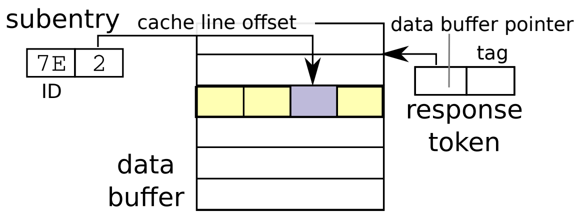


Figure 3.6 – Retrieval of responses from the data buffer. The tag is used to retrieve the MSHR and the pointer to the first row of subentries. The data buffer pointer is used by the subentry buffer to retrieve the cache line from the subentry buffer (yellow). Responses to the individual pending misses can be generated by iterating over all subentries and extracting the relevant word (purple) based on the cache line offsets.

Figure 3.5. When all pending misses have been served, the base pointer is used to deallocate the response data inside the data buffer.

3.2.4 Pipeline Efficiency and Throughput

As long as an MSHR has a single subentry row, the primary and all secondary misses can be handled without stalling the pipeline as they require no more than one read and one write per dual-ported block RAM: lookup in the MSHR buffer, allocation of the MSHR for primary misses, lookup in the subentry buffer for secondary misses, and row update in the subentry buffer. Each block RAM has a data forwarding circuit to ensure that we always read the most up-to-date data despite reads having two-cycle latency. MSHR collisions are handled transparently when the unit is idle, as long as there are free entries in the stash. Allocating an additional subentry row requires stalling the pipeline for one cycle to perform two writes: 1) inserting the pointer of the newly allocated row into the tail of the list and 2) writing the new subentry into the newly allocated row. Allocating a subentry on an MSHR that has more than one row requires traversing the linked list, which costs an extra read per additional row. The traversal cost can be significant for MSHRs with many subentries: to mitigate it, we use an 8-entry fully-associative cache indexed by the head pointer of the subentry list to jump directly to the tail whenever possible. In our subentry architecture, the tradeoff between internal fragmentation and stall cycles, which depend on the number of subentries per row n_s, remains; however, the cost of a full subentry row is reduced from completely stalling the pipeline until the full MSHR is deallocated to a few bubbles in the pipeline. Responses whose MSHR has a single subentry row can also be handled without stalls; each additional subentry row costs one stall cycle. Most of the operations are therefore fully pipelined, with the caveat that a single pipeline is shared between accelerator requests and memory responses. However, the more secondary misses we can merge to the same memory request, the fewer memory responses we will have to handle, reducing the cost of pipeline sharing. Ultimately, the miss handling logic contained in N_b fully-pipelined banks can supply up to $N_b - n_{mem}$ responses per cycle, where n_{mem} is the average number of memory responses per cycle.

3.3 Experimental Results

The innovations presented in this chapter aim at increasing the reuse of individual responses from the DRAM controller by serving as many incoming requests as possible with the same data requested only once. Because this reuse is largely independent from the properties of the DRAM controller and memory, we focus on a single system configuration, the PL system on ZC706, discussed in Section 2.6.3. Indeed, on the ZC706, MSHRs and subentries compete for the same kind of resource as the cache: block RAM. Therefore, we can quantitatively explore the benefit of reallocating some of the resources normally allocated to the cache to implement instead more MSHRs and subentries, which are the new design points introduced by MOMSes which will be discussed in Section 3.3.1. On the other hand, the AWS F1 system uses both BRAM and URAM for cache, MSHRs, and subentries, which makes it harder to visualize and quantitatively analyze the tradeoff. We complement the analysis by showing the advantages of cuckoo hashing for MSHR storage and linked-list architecture for subentries in Sections 3.3.2 and 3.3.3 respectively. Finally, in Section 3.3.4 we show the relation between number of outstanding reads and performance and in Section 3.3.5 we provide details on the resource utilization of our MOMSes.

3.3.1 More Cache or More MSHRs?

We ran our benchmarks on a set of different traditional associative nonblocking caches and MOMS. We used 4-way set associative caches except in the smallest caches due to the limited minimum block RAM depth (see Section 2.1). For the traditional nonblocking caches, we only consider the best architecture that can run at 200 MHz, with 16 MSHRs with 8 subentries each. For MOMSes, we fixed the number of subentries per row to three since, due to the finite choice of block RAM port widths, they occupy the same amount of block RAMs as two and provide a good compromise between utilization and stall cycles, as we will demonstrate in Section 3.3.3. We also fixed the stash size to two entries, which provides timing closure in all cases. We explored the number and depth of MSHR hash tables, as well as the depth of the subentry buffer.

Figure 3.7 summarizes the results. Our MOMSes provide the highest performance benefit to the benchmarks with the highest stack distance percentile (90% and 95%), i.e. the most challenging ones for caches. With rail4284, misses to multiple cache lines are so frequent that even the smallest MOMS with no cache at all performs 25% better than the largest traditional nonblocking cache, which has a 24× larger area. On mawi1234, a small cache is enough to capture any existing temporal locality; after that, investing 2% of block RAMs for a single MSHR cuckoo hash table provides higher returns than any further increase in cache size. pds-80, flickr, youtube, and ljournal offer a more gradual area-delay tradeoff and can benefit from the largest MSHR solutions, which constitute most of the Pareto-dominant points. On these benchmarks, we achieve 10% to 25% throughput increase with the same area or 35% to 60% area reduction at constant throughput. dblp-2010, eu-2005, in-2004, and webbase_1M have higher locality and thus benefit more than other benchmarks from larger caches; however, the simplest MOMS with no cache, which uses 3× fewer BRAMs than the smallest cache, is enough to saturate

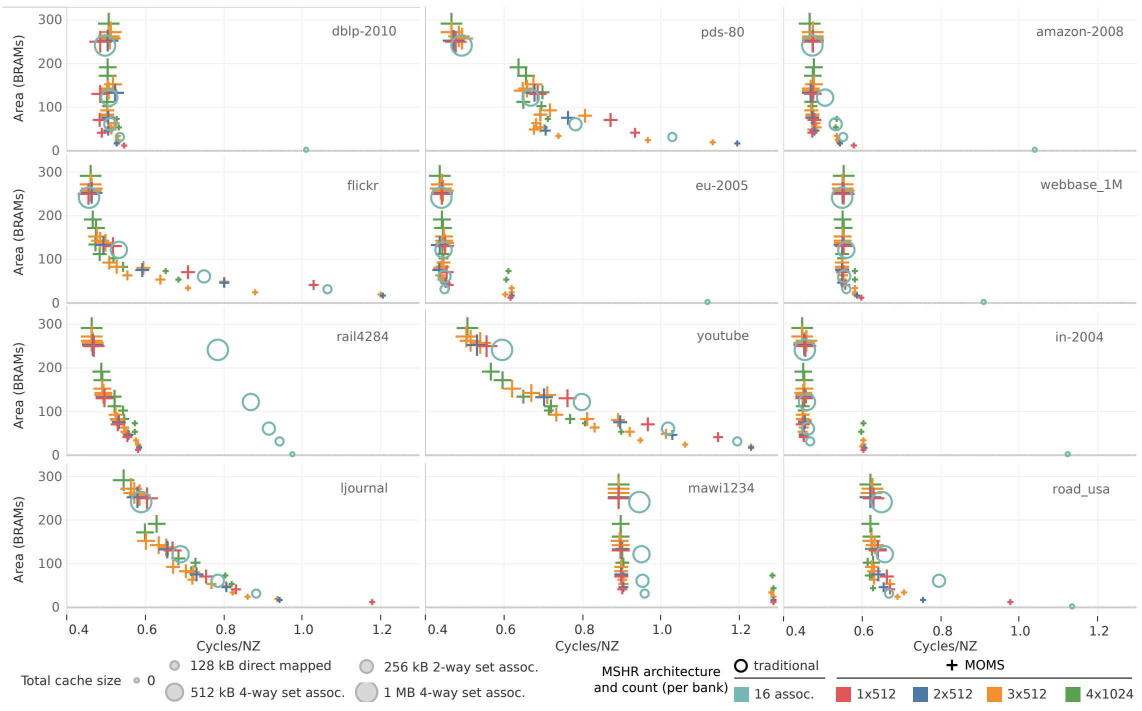


Figure 3.7 – Area of the memory system and normalized execution time for all benchmarks and a broad range of nonblocking cache architectures. For the MOMS architectures, we indicate the number and depth of cuckoo hash tables in each of the four banks, whereas the cache size refers to the entire multi-banked structure. Charts are sorted by increasing vector size and have been truncated at 1.3 cycles/NZ. On half of the benchmarks, all the Pareto-optimal designs are MOMSes, except for the smallest possible but low-performing design with no cache and associative MSHRs. For the other benchmarks, our MOMS provides additional Pareto-optimal designs, especially on the low area side.

the PS DRAM bandwidth only by merging memory requests. On eu-2005 and in-2004, the performance gain provided by the cache-less MOMSes is limited by handling the subentry linked lists. Applications with higher temporal locality may thus benefit from an increase of subentries per row. Benchmarks with few non-zero elements per row such as mawi1234 and road_usa have a lower maximum performance due to the higher bandwidth requirements for the sequential vectors; however, they are among the eight benchmarks that do not saturate the PS DRAM bandwidth without a MOMS.

3.3.2 Number of MSHR Hash Tables and Stash Size

Figure 3.8 analyzes the performance of the MSHR storage architectures described in Section 2.1. For each architecture, we measure average and peak utilization of the MSHR storage space. To make sure the benchmark always uses all of the available MSHRs, we use a synthetic 1M×1M matrix with 5M uniformly distributed non-zero elements, no cache, and each bank contains 4,096 subentry rows with 3 subentries each. All architectures have 2,048 MSHRs per bank or the closest possible value.

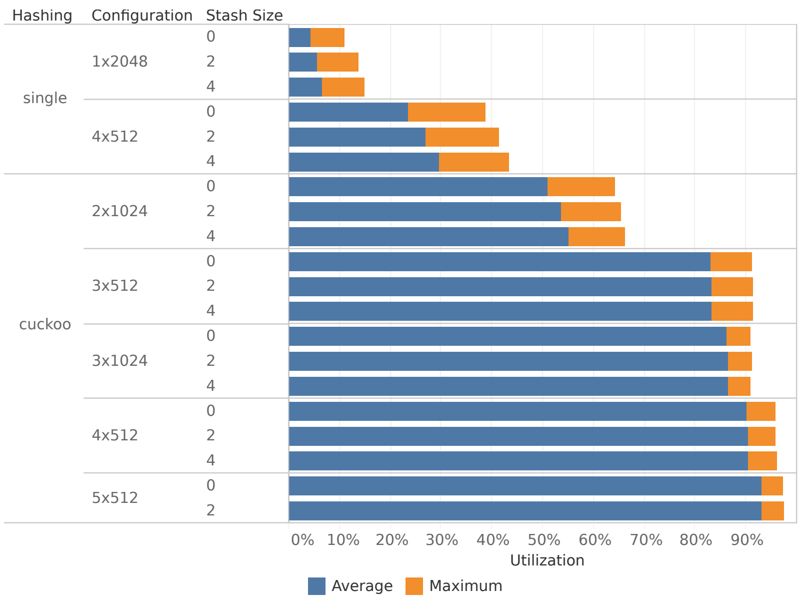


Figure 3.8 – Achievable MSHR storage load factor for several MSHR architectures. The 5×512 system with a 4-entry stash did not meet timing constraints. Single-hash architectures cannot utilize more than 40% of the storage space. Cuckoo hashing can handle collisions more efficiently and three hash tables are enough to achieve more than 80% average and 90% peak load factors, even without stash.

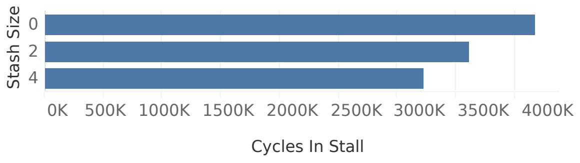


Figure 3.9 – Number of cycles lost due to stalls for collision resolution during the execution of a uniformly distributed benchmark. A 4-entry stash, which uses a few hundreds of LUTs and FFs, reduces the number of stall cycles by 30%.

Because any collision results in a stall that lasts until one of the colliding MSHRs is deallocated, all of the single-hash architectures achieve poor utilization: even by introducing a stash to tolerate up to four collisions, a 4-way set-associative architecture does not go beyond 30% average and 45% peak load factors. Even a simple two-way cuckoo hash table achieves 50% average and 70% peak utilization, and three ways enough to reach more than 80% average utilization, which is consistent with prior findings on cuckoo hashing [32]. Interestingly, using a 3-way 512-entry architecture (1,536 MHSRs) has higher absolute utilization than a 2-way, 1,024-entry organization (2,048 MSHRs). For three or more ways, adding a stash does not affect MSHR utilization but decreases the number of stall cycles by up to 30% with a 4-entry stash (Figure 3.9), which is the largest stash that we could implement within the 200 MHz constraint.

3.3.3 Subentry Organization

We performed a similar analysis for the memory organization of the subentries, as described in Section 3.1.2. We use the ljournal benchmark, which has a large number of secondary misses,

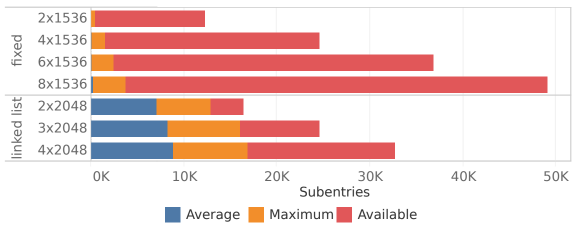


Figure 3.10 – Average and maximum subentry utilization during the execution of ljournal with a 3 × 512 cuckoo MSHR. Allocating a fixed number of subentries per MSHR results in less than 1% average utilization and resource waste. Linked-list architectures provide a more efficient usage of the subentry memory.

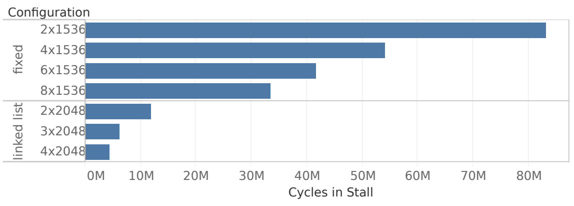


Figure 3.11 – Number of cycles lost due to subentry-related stalls. Stalls occur when (a) filling all subentries of an MSHR for the fixed architectures or (b) handling the linked list or running out of subentry rows for the linked list architectures. The smallest linked list architecture has three times fewer stall cycles than the largest fixed architecture despite having three times fewer subentries.

and a MOMS with no cache and a 3×512 cuckoo MSHR buffer per bank. As shown in Figure 3.10, with a fixed number of subentries per MSHR, stalls are so frequent (Figure 3.11) that they prevent misses from accumulating in the buffers, resulting in very low utilization but also fewer opportunities for request merging and thus a higher traffic to external memory (Figure 3.12). We believe this problem is more pronounced in a MOMS than in a traditional nonblocking cache because it is far more likely to encounter at least one MSHR that needs more than a given number of subentries when handling thousands of misses rather than a few tens of them. Our linked list-based architectures provide much higher average and maximum buffer utilization, fewer stall cycles and decrease the number of external memory requests by a factor 1.3× to 2×.

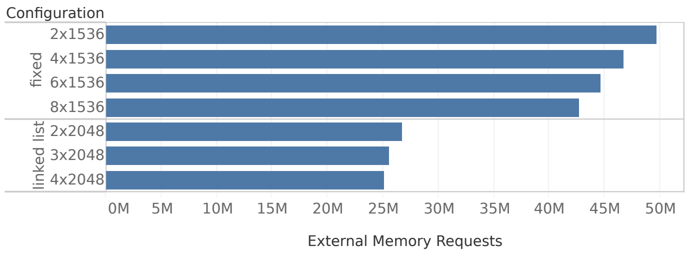


Figure 3.12 – Number of external memory requests during the execution of ljournal with a 3 × 512 cuckoo MSHR and no cache. By increasing subentry utilization, linked list architectures increase the number of accelerator requests that can be served by the same external memory request, resulting in a 37% decrease of external memory traffic.

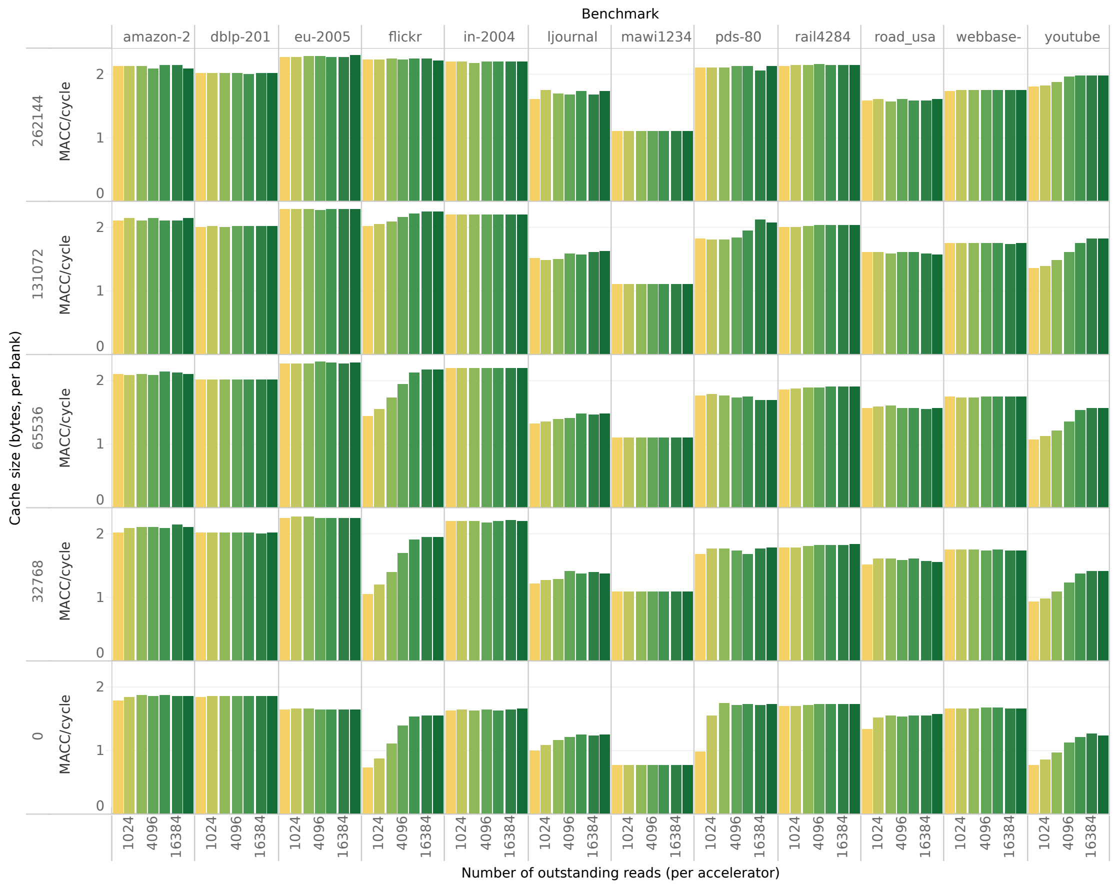


Figure 3.13 – Throughput as a function of ROB size—and thus number of outstanding requests—cache size, and benchmark. The effectiveness of MSHRs and subentries in supplying bandwidth increases with the number of outstanding reads, meaning that MSHRs and subentries not only need to be available but also to be used in order for MOMSes to operate efficiently. This is particularly evident whenever the cache array is too small or the benchmark too irregular to achieve high hit rate, which makes the overall performance more dependent on efficient miss handling.

3.3.4 Number of Outstanding Memory Requests

While all the results shown so far have been obtained using accelerators that can send up to 8,192 outstanding reads, Figure 3.13 shows the impact of changing the size of the SpMV reorder buffer, and thus the maximum number of outstanding reads, from 512 to 32,768. We consider the systems with the largest number of MSHRs and subentries among those presented in Figure 3.7 (4×1024 MSHRs and 3×4096 subentries per bank) and the same five cache sizes (0, 32 kB, 64 kB, 128 kB, and 256 kB per bank).

The benchmarks with the most gradual area/performance curve in Figure 3.7—flickr, ljournal, pds-80, and youtube—show a similar trend with respect to the number of outstanding reads, saturating at around 8k or 16k requests as the bottleneck becomes the number of MSHRs and subentries. On the other hand, benchmarks small or regular enough to essentially achieve peak

Table 3.1 – Resource utilization of MOMSes and traditional cache with 16 MSHRs with 8 subentries each per bank, compared to the resource utilization of the rest of the experimental system. *Fixed infrastructure*, which uses 40–50% of LUTs and FFs of the entire system, includes AXI SmartConnects and the MIG soft memory controller. For MOMSes, we report ranges corresponding to the configurations discussed in Section 3.3.1. The smallest MOMSes has very similar LUT and FF utilization than the traditional cache baseline while consuming 90–95% fewer BRAMs.

	LUT	FF	DSP	BRAM
A) 4 accelerators	19k	23k	32	62
B) Fixed infrastructure	39k	37k	0	1
C) MOMS	21k–23k	29k–31k	4–16	12–314.5
D) Traditional cache	20k	30k	0	253
A + B + C	78k–80k	91k–93k	36–48	75–377.5
Utilization of **A + B + C**	36–37%	21%	4–5%	14–69%

Table 3.2 – Resource utilization of the four-bank MOMS with 256 kiB cache as a function of the number of MSHRs. The number of MSHRs mostly affects BRAM and DSP utilization, which are used for MSHR storage and cuckoo hashing respectively. On the other hand, LUTs and FFs, used in the logic that handles MSHR lookup and update, change by at most 6% and 5% respectively.

	LUT	FF	DSP	BRAM
1 × 512 MSHRs, 3 × 512 subentries	21.3k	29.0k	4	260.5
3 × 512 MSHRs, 3 × 2048 subentries	22.2k	29.0k	12	282.5
4 × 1024 MSHRs, 3 × 4096 subentries	22.6k	30.5k	16	314.5

performance even with small traditional nonblocking caches do not benefit from increasing the maximum number of outstanding reads. Overall, the importance of increasing the size of the reorder buffer is higher whenever the cache is smaller. This confirms the intuition that, whenever MOMSes are crucial for achieving high performance—that is, when caches are absent, small, or the workloads are too irregular to benefit from them—their effectiveness increases with the size of the pool of incoming requests that can be considered for merging to the same memory request.

3.3.5 Resource Utilization

Table 3.1 shows the resource utilization of the entire system, with MOMS and baseline traditional nonblocking cache described in Section 3.3.1. For MOMSes, we provide ranges that correspond to the configurations presented in Section 4.3.1.

Indicatively, the minimum cache that is worth implementing due to the minimum block RAM depth—a single 32 kB way (512 lines × 512 data bits)—has similar block RAM requirements as 3×512 MSHRs with 3×2048 subentries. In general, on the ZC706, the cache requires 8.5 block RAMs per 32 kB per cache way, the MSHR buffer requires 0.5 block RAMs per 512 MSHRs per cuckoo hash table for storage plus 0.5 block RAMs per 512 MSHRs for the request queue to the external memory arbiter, and the subentry buffer requires 1 block RAM per 512 subentry rows of up to 3 subentries each, plus 1 block RAM every 1,024 subentry rows for the FRQ. Each cuckoo hash function also uses 1 DSP block.

BRAMs are the dominant resource in MOMSes and traditional caches and, especially the former, has large variations depending on the number of MSHRs, subentries, and cache size. Cache-less MOMSes have up to 90–95% fewer BRAMs than traditional caches. The FF utilization of MOMSes is comparable and generally slightly lower than that of the traditional cache, as FFs are repurposed from MSHR and subentry storage to, mostly, pipeline registers. The LUT utilization is 4–10% higher in MOMSes due to more complex logic. Overall, the LUT and FF utilization of MOMSes is comparable to that of accelerators and has minimal variations depending on the number of MSHRs and subentries, as shown in Table 3.2. In addition to traditional caches, MOMSes use at most 5% of the available DSPs for cuckoo hashing.

3.4 Conclusion

Conventional wisdom has it that some form of local buffering such as caching is the best way to optimize the access to external memory, hence the vast effort in maximizing the hit rate under all possible scenarios. Nonblocking caches are one of the few architectures for *miss* optimization instead. In this chapter, we took the key idea behind nonblocking caches to the extreme: we designed a scheme to handle three orders of magnitude more misses without stalls compared to classic nonblocking caches based on fully-associative MSHRs. We presented an efficient FPGA implementation of such MSHR-rich cache, where we map tens of thousands of MSHRs and subentries to the abundant FPGA block RAM and all stages of miss handling are pipelined with minimal stalls. On twelve sparse matrix-vector multiplication benchmarks, we showed that, under a limited block RAM budget, repurposing some block RAMs from cache to MSHRs can provide higher performance gains when the access pattern is such that a relevant amount of misses cannot be avoided. This is especially true for the benchmarks with the lowest temporal locality, but even on more regular access patterns, MSHRs can complement caches by optimizing long-distance reuse, providing similar performance gains as a larger cache at lower area costs. Having discussed how to increase the reuse of every memory response, in the next chapter we will introduce a technique to increase the available bandwidth by optimizing the memory access pattern that is sent to the memory controller.

4 Increasing Available Bandwidth by Using Bursts

In Chapter 3, we 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 the accelerators (Figure 4.1b) on-the-fly, without relying on long-term storage in cache. We showed that repurposing some on-chip memory from cache to MSHRs generally proves to be beneficial, especially when the DRAM controller exposes the entire DRAM burst through a wide data port.

However, this is not the case on DRAM controllers with multiple narrow ports, which are commonly found on SoC platforms. In those cases, individual memory requests do not provide enough opportunities for reuse and still result in data wastage on the memory side as they are narrower than a DRAM burst. Similarly, when individual accesses cannot possibly exploit the entire memory bandwidth due to limitations on the maximum number of outstanding reads—which is the case for all of our memory systems—optimizing the reuse of individual memory requests is generally useful but leaves some performance on the table. Finally, 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.

To address these scenarios, we show in this chapter how to extend MSHRs to support bursts of variable length on the memory side (Figure 4.1c). When possible, we make bursts longer and exploit more of a DRAM burst or row without being limited to the data width exposed by the specific memory controller. Conversely, when spatial locality is insufficient, we keep burst short and minimize contention in the controller or avoid requesting unnecessary DRAM bursts. We will show that supporting bursts (1) makes MOMSes beneficial even behind DRAM controllers with multiple narrow ports and (2) further boosts read throughput behind wide memory ports by increasing DRAM row utilization and, when memory controllers are optimized for bursts, memory-level parallelism. In practice, with respect to the memory system characterization we

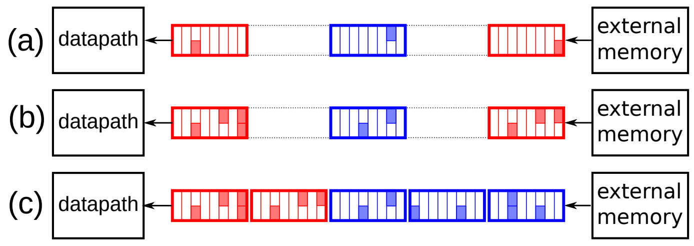


Figure 4.1 – Total availability and utilization of DRAM-based external memory bandwidth under short irregular access patterns. Thick big rectangles: eight-beat bursts transferred from external memory to the datapath on the FPGA, color-coded by DRAM row. Shaded smaller rectangles: portions of data actually used by the accelerators. Dashed lines: cycles where no transfers occur due to a DRAM row conflict. If requests from the accelerators are forwarded directly to the memory (a), most of the burst content will be discarded and frequent row conflicts hamper the available DRAM bandwidth. Miss-optimized memory systems (b) improve the utilization of each burst. In addition, sending variable-length bursts on the memory side (c) reduce DRAM row conflicts, which further increase the effective bandwidth available to the accelerators.

performed in Section 2.6.4, we make use of the higher performing curves corresponding to burst lengths greater than one.

4.1 Key Ideas

In this section, we first define the new mapping between MSHRs and memory regions of variable length (Section 4.1.1) and present its practical implementation in Section 4.1.2. To avoid reducing the efficiency of the pipeline in the MSHR buffer and to maintain a reasonable area overhead, we had to accept the possibility for the new MSHR buffer to send out redundant requests in some circumstances. However, in Section 4.1.3 we use a qualitative model to demonstrate that those circumstances are expected to be rare, especially when the incoming access pattern is irregular, which is the target scenario for MOMSes.

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

In nonblocking caches, MSHRs have the granularity of single cache lines (Figure 4.2a). Since cache lines are handled fully independently from each other, there are no guarantees that cache lines that are close in the address space, thus most likely on the same DRAM row, will be requested close to each other in time. If the separation between the requests is larger than the reorder window of the DRAM controller, unnecessary row conflicts will occur.

A simple way to make use of larger portions of DRAM rows would be to increase the granularity of each MSHR to multiple cache lines (Figure 4.2b). Burst transfers can then be used to request

This chapter is based on the work published at the *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications*, 2019 [12] and on *Transactions on Reconfigurable Technology and Systems*, 2021 [15].

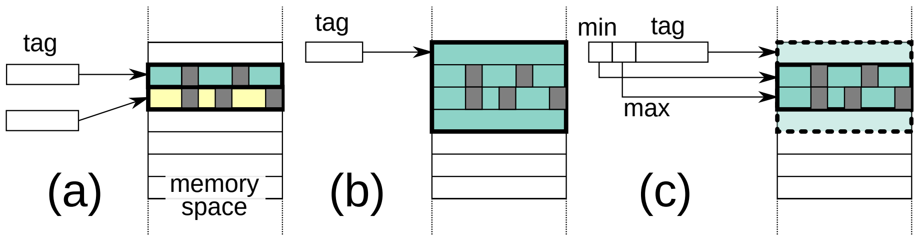


Figure 4.2 – MSHR memory range and structure. Portions of cache lines that have been requested by some accelerators are shown in gray. MSHRs usually refer to single cache lines (a). Increasing the memory range covered by each MSHR to a set of cache lines that will be requested as a burst (b) reduces 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.

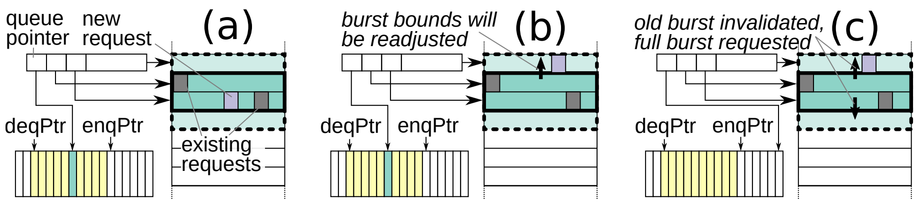


Figure 4.3 – Burst update policies. Requests that fall within the current burst range (a) do not require any updates; otherwise, burst bounds can be updated if the burst memory request is still in the output queue (b). If the memory request has already left the queue (c), the current burst is invalidated and a new burst of maximum length is requested.

such cache line groups efficiently. However, any cache line within the burst that is not actually needed will cause bandwidth and energy wastage. As we show in Section 4.3.1, this often results in lower performance than operating with single memory requests.

To strike a balance between DRAM row utilization and bandwidth wastage, we propose to have *each MSHR covering multiple cache lines* but to *dynamically adjust the bounds of the burst requested to memory* based on the cache lines that are actually needed (Figure 4.2c). 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.

4.1.2 Dynamically Adjusting Burst Bounds

Algorithm 2 describes how we implement miss handling with variable-length MSHRs. On a primary miss, a new MSHR is allocated and a memory request is inserted in the output queue; its burst initially covers only the primary miss' cache line. To enable future updates of the request, we store its address in the output queue (`queuePtr`) in the MSHR; `queuePtr` is initialized to the queue's enqueue pointer, `enqPtr`. Secondary miss handling is described in Figure 4.3 and implemented by the circuit in Figure 4.4. Secondary misses that are covered by the current burst bounds (Figure 4.3a) require no updates to the MSHR. If the current burst does not cover

Algorithm 2 MSHR burst offset handling

```
Input: a miss at address addr = (tag, burstOffset, cacheLineOffset)
Result: Updated MSHR buffer and queue

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

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 (Figure 4.3b). 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 (Figure 4.3c). 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 the worst case, up to 2^N memory requests per MSHR may be needed. Since each burst would need a separate `minBurstOffset` and `maxBurstOffset`, the size of each MSHR would dramatically increase. Moreover, since we would also need to look up all the bursts associated to a given MSHR to determine whether any of them covers the new secondary miss or whether any of them can still be updated, the circuit in Figure 4.4, often already on the critical path of the entire system, would become even more complex.

To handle the cases shown in Figure 4.3c 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

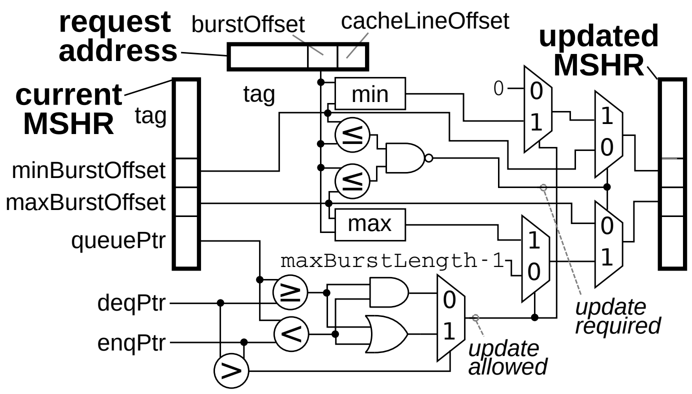


Figure 4.4 – Burst bounds update circuit. The updated MSHR on the right overwrites the current MSHR on the left in the following cycle; `tag` and `queuePtr` are never modified after MSHR allocation. Considering that realistic burst offsets and queue pointers are on 1–4 bits and 9–12 bits respectively (see Section 4.3), the policies shown in Figure 4.3 can be implemented with a relatively lightweight circuit.

memory region—essentially, we take this for an indication of sufficiently high spatial locality. As discussed in Section 2.6, this policy allows us to achieve the same operating frequency as single-request MSHRs. However, discarding responses cause bandwidth wastage and should be reduced to a minimum, which is achieved by having MSHRs spend the largest fraction of their lifetime in the output queue rather than in the memory controller. If (1) accelerators generate more memory requests than the memory controller can sustain and (2) there are more MSHRs than maximum in-flight requests in the memory controller, this happens naturally, as the next section will show.

4.1.3 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:

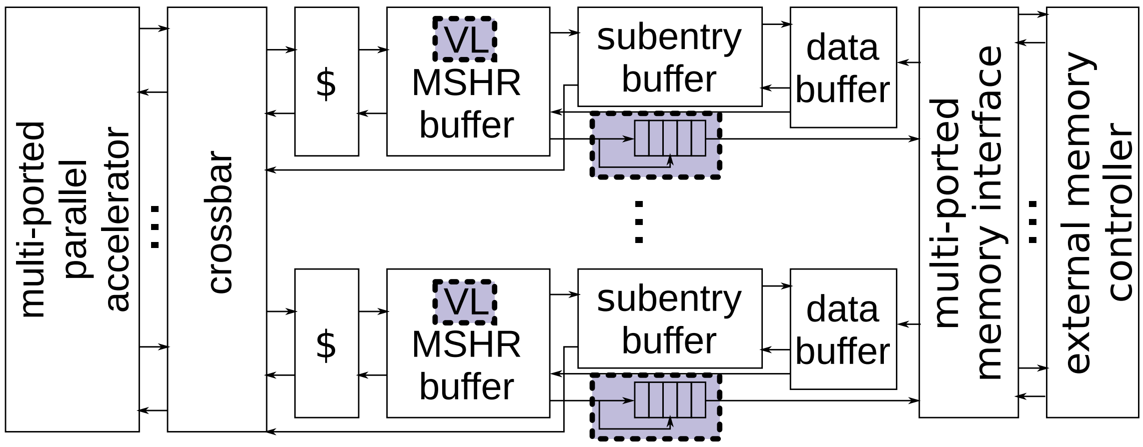

multi-ported parallel accelerator
crossbar
$
VL
MSHR buffer
subentry buffer
data buffer
multi-ported memory interface
external memory controller

Figure 4.5 – Top-level view of the burst-based MOMS. We highlighted the main differences compared to the single-request MOMS: variable-length (VL) MSHR buffer and updatable queue.

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.

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

4.2 System Architecture

Figure 4.5 shows the top-level organization of DynaBurst, which extends the single-request MOMS discussed in Chapter 3. 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 4.1. 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 still 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

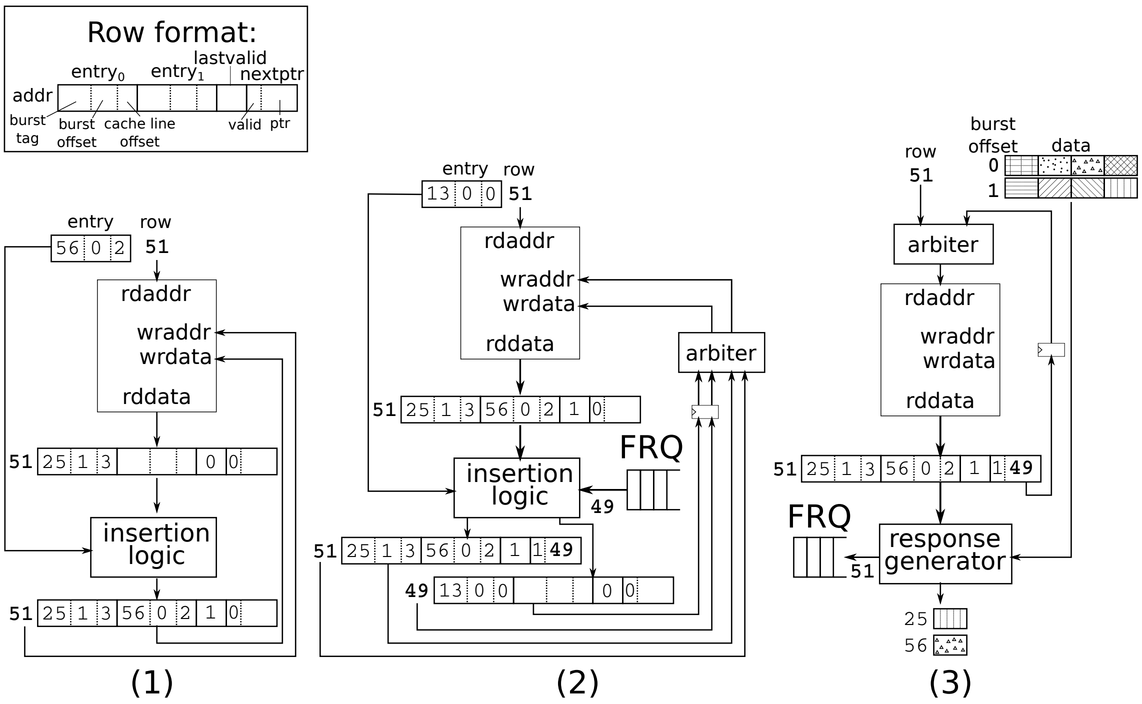


Figure 4.6 – Block diagram and operation of the subentry buffer with variable-length bursts: allocation of a subentry when the row is (1) not full or (2) full and (3) deallocation of all subentries and response generation. The operation is very similar to the case of single requests shown in Figure 3.5 except for the burst offset that has to be stored and used to retrieve the cache lines during response generation.

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, which will be used to retrieve the corresponding cache line once the response returns. The response token generated by the data buffer now also contains the burst length so that, once all the responses have been generated, the entire burst can be deallocated from the buffer. Figure 4.6 and 4.7 demonstrate the operation of the new subentry buffer and data buffer.

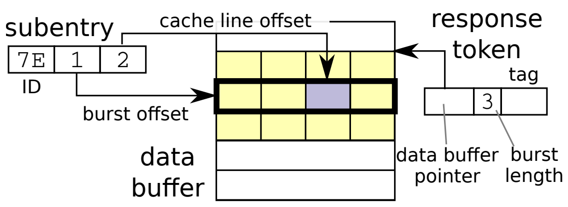


Figure 4.7 – Retrieval of responses from the data buffer. The tag is used to retrieve the MSHR and the pointer to the first row of subentries. The data buffer pointer is used to retrieve the cache line from the subentry buffer (yellow). Responses to the individual pending misses can be generated by iterating over all subentries and extracting the relevant word (purple) based on the burst and cache line offsets. Once all responses have been generated, the burst length is used to deallocate the burst from the data buffer.

Finally, most of the considerations about pipeline efficiency discussed in Section 3.2.4 still apply except that the minimum penalty due to responses improves from one cycle per response to one cycle *per burst*.

4.3 Evaluation

For all systems we consider three configurations in terms of MSHRs and subentries. All configurations on the ZC706 have six subentries per row, which is twice the number of subentries per row that we considered when evaluating our single-request architecture in Section 3.3.1. Indeed, with MSHRs covering multiple cache lines, we expect an increase of secondary misses. The configurations on the AWS F1 have eight subentries per row since the two extra subentries can be implemented using the same amount of URAMs for the subentry buffer, given the larger width of URAM primitives compared to BRAM ones.

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

Similarly, the 60 ZC706 PL systems have (1) one 512-, (2) three 512-, and (3) four 1024-entry MSHR cuckoo hash tables with (1) 512, (2) 2,048, and (3) 4,096 subentry rows per bank; 32–256 kiB of cache per bank or no cache, and the maximum burst lengths from 2 to 16. PL systems have a 32-entry, 512-bit wide data buffer per bank.

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

We will compare each of these architectures to alternative generic memory systems: (1) single-request MOMSes—with same amount of MSHRs, subentry rows, and cache and (2) a traditional nonblocking cache with 16 associatively-searched MSHRs, each with 8 subentries, with the closest BRAM utilization on ZC706 and with 256 kiB of cache per bank on AWS. Systems (2) are the same baselines used in Section 3.3.1 and contain the maximum number of MSHRs and subentries that ensure timing closure at 200 MHz (ZC706 PL) or 250 MHz (AWS F1) and that result in a FF utilization similar to the MOMS architectures (all systems; see Section 4.3.5).

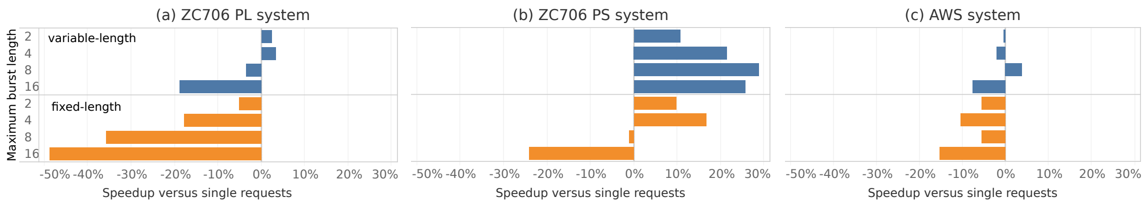


Figure 4.8 – Speedup obtained in MOMSes by sending bursts of memory requests compared to single-request MOMSes, both with dynamically adjusted burst bounds and always requesting full bursts (geometric mean across all benchmarks and all configurations). Using bursts of a suitable size is beneficial to all systems and minimizing each burst's length is always better than using bursts of fixed length.

4.3.1 Benefits of Dynamically Adjusting the Burst Length and Impact of Maximum Burst Length

Figure 4.8 shows the speedup of using bursts compared to restricting to single memory requests. Adjusting burst bounds is always useful, on all design points. On the ZC706 PS system, four beats of 64 bits (256 bits) corresponds to the PS DRAM 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. In addition, bursts of size 4 are the shortest ones that can possibly saturate the ZC706 PS DRAM bandwidth, as shown in Section 2.6.4. Still, trimming bursts yields even higher speedups as contention among the memory controller ports is minimized. This effect does not appear on the ZC706 PL and AWS systems as single responses already consist of full DRAM bursts.

The maximum burst length controls the tradeoff between using larger parts of DRAM bursts/rows and wasting bandwidth due to requesting unnecessary data, either between pending misses on distant cache lines or due to frequent burst invalidations (see Figures 4.2 and 4.3). This tradeoff explains the bitonic speedup curve on both ZC706 systems and the single useful maximum burst length on the AWS system.

Overall, the ZC706 PS system gains the most from using bursts. Indeed, restricting to single 64-bit memory requests leaves few opportunities for reuse among 32-bit accelerator requests. The ZC706 PL systems benefit from bursts only by better exploiting the memory access pipeline and through DRAM row conflict minimization, which still brings significant speedup on specific design points as discussed in Section 4.3.3. The AWS memory system benefits from bursts even more than the ZC706 PL one as the single-request bandwidth is capped to 50% of the maximum instead of 70%. However, the larger available bandwidth per PE makes the bandwidth optimizations between MOMS and DRAM controller generally less critical than on the ZC706.

4.3.2 Architectural Exploration

We further analyze the architectures with the ideal maximum burst length for the respective system—4 for ZC706 PL systems and 8 for ZC706 PS and AWS systems respectively. Figure 4.9

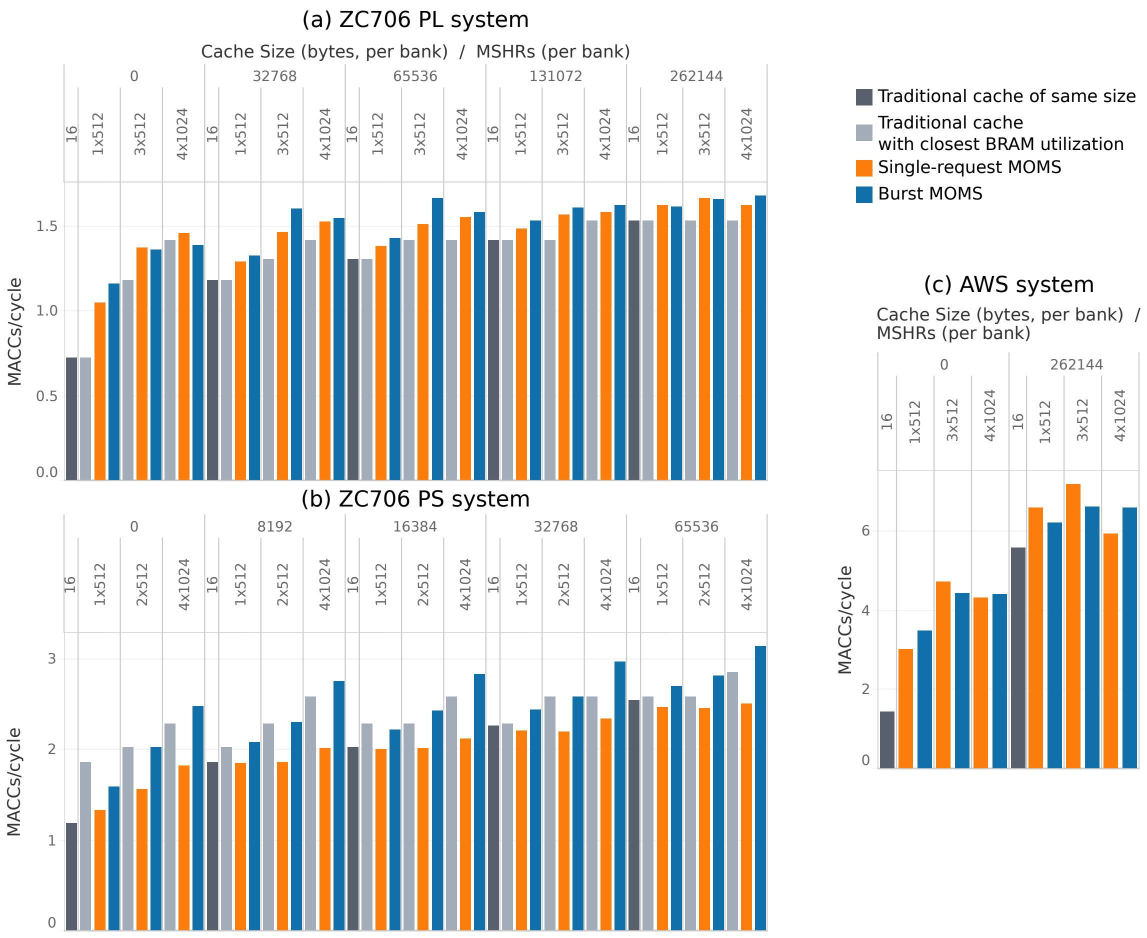


Figure 4.9 – Throughput, in geometric mean across all benchmarks, of a traditional nonblocking cache, a single-request MOMS and variable-length burst MOMS with maximum burst length of 4 for the ZC706 PL system and 8 for the ZC706 PS and AWS systems. For the ZC706 systems, without URAM, we additionally compare each MOMS to the traditional nonblocking cache with the closest BRAM utilization to analyze performance within a fixed area budget.

compares the throughput of traditional caches, single-request and burst MOMSes for different cache sizes and MSHR count.

If the memory controller has multiple narrow ports (ZC706 PS system), repurposing some BRAMs from cache to MSHRs/subentries never pays off unless bursts are used. The speedup increases with the number of MSHRs and at four cuckoo hash tables becomes comparable to the single-request results on the PL system. Even there, bursts provide additional speedup on most of the architectures, especially where single-request architectures were the most useful. This includes the most lightweight system, whose baseline with the closest area has no cache at all, and on intermediate configurations with 3×512 MSHRs/bank and moderate cache size. On AWS, we cannot compare traditional caches at constant BRAM utilization since both caches and MSHRs/subentries use a mix of BRAM and URAM. Complementing caches with MOMSes is always useful compared to being limited to a few tens of associative MSHRs but there is no clear winner between single-request and burst MOMSes. Burst MOMSes have a significant

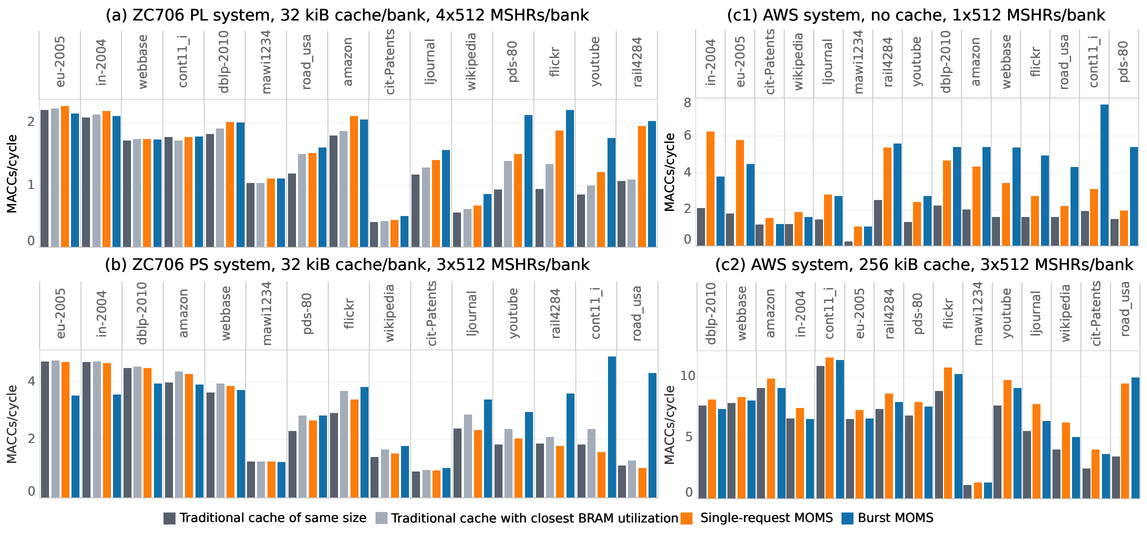


Figure 4.10 – Throughput of traditional nonblocking cache, single-request MOMS and burst MOMS on individual benchmarks for architectures where burst MOMSes have the largest speedup compared to traditional caches. Benchmark are sorted by increasing burst MOMS speedup compared to traditional caches. For AWS, we include an architecture where the single-request MOMS performs particularly well (c2). On ZC706, burst MOMSes are beneficial to most of the largest and/or irregular benchmarks, where traditional caches have the lowest performance. On AWS, burst MOMSes are particularly useful where memory bandwidth is more critical (c1); when the memory bottleneck is less evident, single-request MOMSes introduce less caches pollution and perform slightly better than both traditional caches and burst MOMSes (c2).

advantage with no cache and few MSHRs, where bandwidth maximization is more critical and, for a fixed number of MSHRs, burst MSHRs can handle more cache lines than single-request ones. Single-request MOMSes appear to be more beneficial than burst MOMSes when paired to a cache, which suggests that more sporadic misses are better handled individually; however, they also seem to interfere more unpredictably with DMAs as the 4 × 1024 single-request MOMSes starved accelerators of sequential data more often than 3 × 512, resulting in lower performance, which was not the case for the burst MOMSes. This phenomenon can only occur on AWS as on the ZC706 systems DMAs and MOMSes were connected to different memories.

4.3.3 Detailed Speedup Profile

Figure 4.10 shows the throughput on individual benchmarks provided by the best-performing MOMS on each system. Small and/or regular benchmarks, characterized by high cache hit rate, benefit more from a larger cache than from more MSHRs, which is reasonable. Where caches are less effective, bursts make MOMSes useful also on the PS system, achieving up to 3.4× speedup. On the ZC706 PL system, burst architectures improve the performance of MOMSes on 10 benchmarks out of 15, in six cases by more than twice. The trend is confirmed by the absolute performance on the traditional nonblocking cache: the speedup is the highest on the benchmarks where the traditional nonblocking cache was performing worse. On AWS, the smallest cache-less burst MOMS (c1) generally performs better than the single-request one,

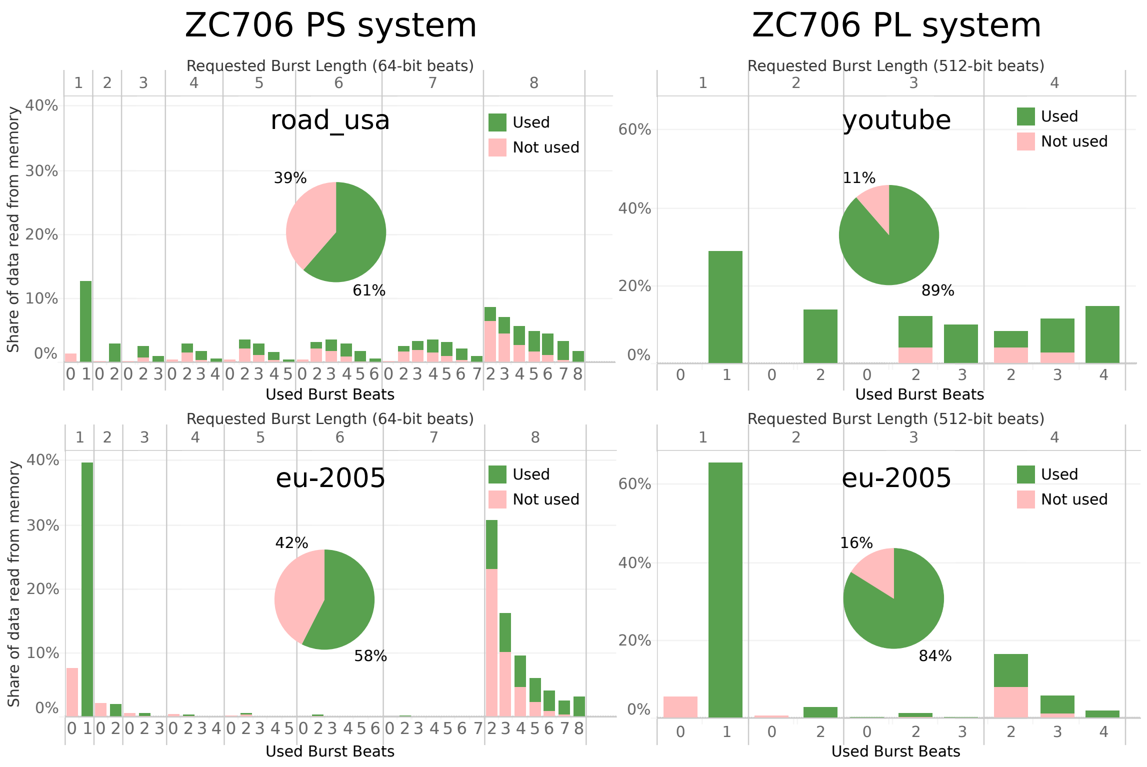


Figure 4.11 – Distribution of requested, used, and wasted cache lines per burst as a function of the burst length, normalized by the total number of cache lines requested from memory, for the same ZC706 systems evaluated in Figure 4.10. Pie charts: used and wasted cache lines, aggregated over all burst lengths. Top (bottom) row: benchmarks where introducing bursts is advantageous (harmful) compared to single-request systems. Benchmarks that get the highest speedup from bursts obtain a large share of useful data through bursts of all possible sizes. When the performance is poor, invalidations and single requests are more frequent.

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

4.3.4 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 ZC706 PS and PL systems and analyzed how many of the cache lines requested from memory are actually used. More specifically, Figure 4.11 shows, for each burst length, how many of the requested cache lines have been actually used at least once and how many have been wasted, normalized by the total number of requested cache lines.

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

In the high-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 as we showed in Section 2.6.4. Indeed, even if the total share of wasted data is similar in both ZC706 PS benchmarks, and higher than in the ZC706 PL system, the speedup provided by bursts is significantly higher in road_usa than in eu-2005.

Where burst MOMSes are particularly effective, most of the bursts converge to their optimal length by the time requests are sent to memory as invalidations are almost non-existing. Conversely, on the regular benchmarks, single requests are more common 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.

4.3.5 Resource Utilization

Table 4.1 shows the resource utilization of the entire system, with MOMS and baseline traditional nonblocking cache described in Section 3.3.1. For MOMSes, we provide ranges that correspond to the configurations presented at the beginning of this section.

On the AWS F1 FPGA, we used URAM for the cache data and the subentry buffer. While the FPGA has 4× more memory bits in URAMs than BRAMs, the larger minimum depth of each block, 4,096 entries instead of 512, increases the minimum size of the cache and subentry buffer that are worth implementing to 256 kiB and 4,096 rows respectively, per bank. Such cache requires 8 URAMs and 4 BRAMs (for tag and valid arrays) instead of 68 BRAMs if URAMs were not available. As for the subentry buffer, 4,096 rows of eight subentries each require 3 URAMs for all the maximum burst lengths that we considered, instead of 25–29 BRAMs.

BRAMs and URAMs are the dominant resource in MOMSes and traditional caches and, especially the former, has large variations depending on the number of MSHRs, subentries, and cache size. On ZC706, cache-less MOMSes have up to 90–95% fewer BRAMs than traditional caches; on AWS, 60% fewer URAMs. The FF utilization of MOMSes remains comparable and generally slightly lower than that of the traditional cache despite the burst handling logic. The LUT utilization is 5–85% higher in MOMSes due to more complex logic. Still, even the largest MOMS uses at most 16% of FPGA LUTs, which is dwarfed by the 30–50% of LUTs locked in fixed infrastructure. Overall, the LUT and FF utilization of MOMSes is comparable to that of accelerators. In addition to traditional caches, MOMSes use at most 5% of the available DSPs for cuckoo hashing. On

Table 4.1 – Resource utilization of MOMSes and traditional cache with 16 MSHRs with 8 subentries each per bank, compared to the resource utilization of the rest of the experimental system. *Fixed infrastructure*, which uses 50–70% of LUTs and FFs of the entire system, includes AXI SmartConnects, soft memory controllers and, in (c), AWS shell. For MOMSes, we report ranges corresponding to the configurations discussed in Section 4.3.1. Accelerators on the ZC706 systems use DMAs with 64-bit data ports as they are connected to 64-bit memory controllers (PL) or to be able to fit 8 accelerators (PS), while on the AWS system they use more resources as they are 512-bit wide as the memory controllers. On ZC706, the smallest MOMSes has very similar LUT and FF utilization than the traditional cache baseline while consuming 90–95% fewer BRAMs. On AWS, the cache-less architecture save 60% of the URAMs.

(a) PL system on ZC706

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

(b) PS system on ZC706

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

(c) AWS F1

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

Table 4.2 – Resource utilization of the 16-bank MOMS on AWS F1 with 256 kiB cache, 3 × 512 MSHR, 8 × 4096 subentries per bank as a function of the maximum burst length. The overhead of burst handling is mostly due to the logic shown in Figure 4.4 and to the burst offset bits in each MSHR and is within 21% for LUTs, 7% for FFs, and 15% for BRAM.

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

AWS, despite multiple SLRs, we achieve timing closure at 250 MHz with around 70% LUT and 60% BRAM utilization across the entire device and 80–90% on the top SLR.

Table 4.2 shows the impact of maximum burst length on the resource utilization. The burst handling logic shown in Figure 4.4 has at most a 21% LUT and 7% FF overhead, while the minimum and maximum burst offsets and queue pointer in each MSHR account for a worst-case 15% BRAM overhead.

4.4 Conclusion

Irregular memory access patterns bring DRAM memories far from their optimal operating point, which reduce the benefit of datapath parallelization. We showed in the previous chapter that MOMSes 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. In this chapter, we extended MOMSes 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 DRAM row buffer and by increasing memory-level parallelism. Using bursts makes MOMSes beneficial also behind DRAM controllers with multiple narrow ports, commonly found on SoC platforms, and further increases their usefulness when memory ports are wide, especially when single requests alone cannot achieve peak bandwidth utilization. In the next chapter, we will present an alternative technique to increase DRAM bandwidth when single requests are not excessively penalized over bursts which does not incur in data wastage associated to burst handling.

5 Increasing Available Bandwidth by Large-Scale Request Reordering

The original MOMS described in Chapter 3 groups incoming requests that can be served by the same, often wide, memory request (e.g., 512 bit). However, memory requests are sent out in no particular order, which may cause frequent row conflicts when accesses are irregular (frequent gaps in Figure 5.1a). By organizing incoming requests according to grouping regions spanning 4–8 memory requests, the variable-length burst extension discussed in Chapter 4 sends bursts of (contiguous) memory requests instead. To minimize memory traffic, we assemble all incoming 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 Figure 5.1b). 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 Figure 5.1b). We showed in Section 4.3.4 that both mechanisms result in wasting up to 40% of the data received from memory.

The main idea of MOMSes is to accommodate thousands of requests to maximize the number of incoming requests that can be served by the same memory response. To accumulate those incoming requests without stalling, we use a deep queue to buffer the outgoing memory requests as they are generated. So far, we 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 chapter, we discuss how to 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 (Chapter 3) or burst (Chapter 4), we group memory requests based on their DRAM bank and row, and memory accesses belonging to the same group

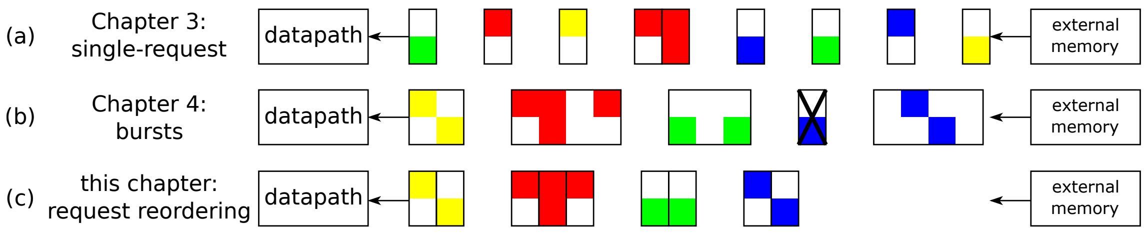


Figure 5.1 – Impact of different miss-optimized architectures on the memory access pattern and thus on DRAM bandwidth. In this example, incoming requests are half the size of the DRAM controller word size; colors identify the DRAM row that was read. The single-request MOMS (Chapter 3) requests single words without applying reordering (a), which may lead to frequent row conflicts that reduce the available bandwidth. The improvement presented in Chapter 4 (b) sends variable-length bursts that cause fewer row conflicts but that may include data that is not needed (the columns fully white) or request some data twice (the crossed blue request, which has been requested again in the following full burst). The architecture proposed in this chapter (c) explicitly reorders individual requests, minimizing row conflicts without data wastage.

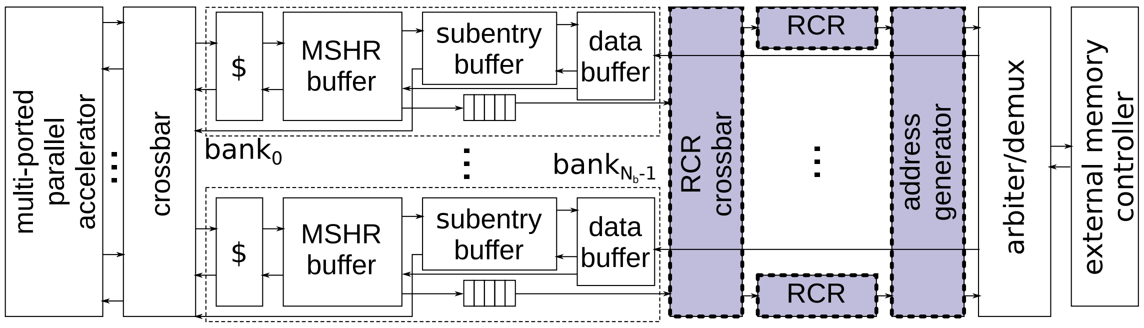


Figure 5.2 – Top-level architecture of the MOMS with row conflict reducers (RCRs), where we highlighted the main differences from the single-request MOMS presented in Chapter 3. The RCRs are inserted between the MSHR buffers and the external memory controller. Each of them is assigned to a bank of the DRAM. The RCR crossbar steers requests between MOMS banks and RCRs. The responses bypass the RCRs and reach the MSHR buffers directly.

will be sent out contiguously to the DRAM controller. Doing so reduces DRAM row conflicts without incurring in the data wastage associated with bursts (Figure 5.1c).

5.1 Key Idea: Where and How to Reorder?

As mentioned in Section 2.3, 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 overlap precharge and activation in two different banks; modern reordering DRAM controllers may even anticipate the next row activation to completely hide its latency [82].

This chapter is based on the work published at the *Proceedings of the IEEE International Conference on Field Programmable Technology*, 2019 [24].

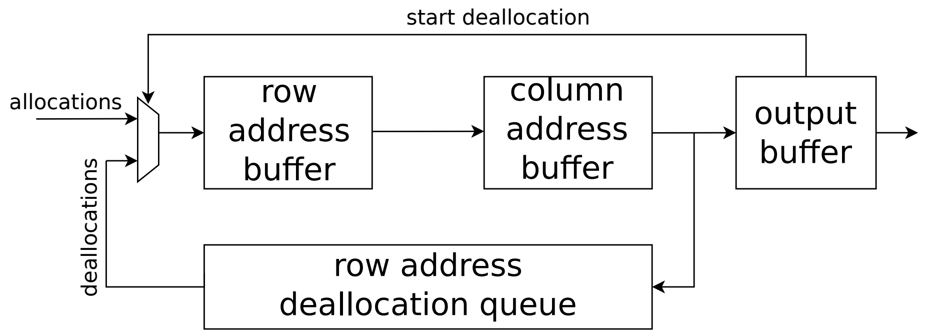


Figure 5.3 – Architecture of an RCR. Incoming tags are decomposed into row and column addresses; the row is searched and, if needed, allocated into the row address buffer. Each entry in the row address buffer includes a pointer to a region into the column address buffer that contain all column addresses 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.

To achieve these goals, we extend the original MOMS as shown in Figure 5.2. The Row Conflict Reducers (RCRs) are responsible for the memory access reordering. Since MOMSes already return responses out-of-order, further reordering of output requests 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 MOMS 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.

5.2 Row Conflict Reducer

The goal of the row conflict reducer is to group the incoming requests by row address. The architecture of the RCR is shown in Figure 5.3 and consists of 1) row address buffer, 2) column address buffer, and 3) row address deallocation queue.

The miss handling logic between accelerators and RCRs interprets the request address according to Figure 5.4a and sends only the tag to the RCRs. The way the tag address is decomposed into DRAM bank, row, and column address (Figure 5.4b) 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 [46].

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, as summarized in Table 5.1(a). When an RCR receives a request, it first searches the corresponding row address inside

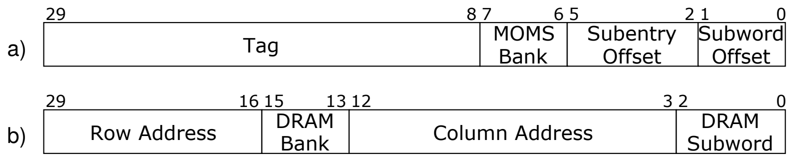


Figure 5.4 – Interpretation of memory addresses by the miss-handling logic (a) and DRAM memory (b). The miss handling logic decomposes the incoming byte address as the tag of the cache line, the address of the MOMS bank, the subentry offset (the word offset in the cache line) and the subword offset (a); 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 ZC706 PL system described in Section 2.6.2, 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 [46], the DRAM has eight 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.

Table 5.1 – Analogies and differences between MOMS miss handling and RCR memory request handling. The two processes operate at different scales but have large conceptual overlaps.

(a) analogies

	miss handling	memory request handling
primary grouping key	cache line tag	DRAM row address
secondary address	cache line offset	DRAM column address
bank address	MOMS bank address	DRAM bank address
primary storage	MSHR buffer	row address buffer
secondary storage	subentry buffer	column address buffer

(b) differences

	miss handling	memory request handling
output address dependencies	primary grouping key	primary grouping key and secondary address
deallocation time	after receiving a response	after sending out the request
uniqueness of secondary addresses within a group	no	yes

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, as summarized in Table 5.1(b). 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 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, discussed in Section 4.1.3.

5.2.1 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 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.

5.2.2 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.

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, 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.

5.2.3 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 (Figure 5.2). 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 5.1.

5.3 Evaluation

We compare our RCR-based MOMS to the original MOMS discussed in Chapter 3 and to its burst-based extension presented in Chapter 4 on the ZC706 PL system introduced in Section 2.6.2.

We also tested the dual ZC706 PS configuration, where the irregular accesses are performed on the PS memory instead, where the burst MOMS excels. With that configuration, however, the RCR-based 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.

On the other hand, the AWS memory system has two important differences compared to the ZC706 PL one. Firstly, as discussed in Section 2.3, DDR4 memories achieve ideal throughput only if consecutive requests hit different bank groups. This requirement could be easily satisfied by grouping RCRs by bank group and changing the arbitration policy of the address generator. However, the second difference has more profound implications and make the memory system unsuitable to cleanly demonstrate the usefulness of row conflict reduction. The Xilinx UltraScale+ MIG controller exposes an *autoprecharge* signal that can be used to control, on a per-transaction basis, whether to immediately precharge a bank following a column access or to leave the row open [113]. If this signal were exposed by the AWS shell to the user logic, then it would be possible to immediately precharge a bank following the last access to a given row, making row conflict reduction even more effective. Instead, the autoprecharge signal is tied inside the encrypted shell to bit 8 of the incoming byte address [7], as shown in Figure 2.7. While this is optimal for sequential accesses when coupled with the specific address mapping used [7], it results in an address-dependent open/closed row policy in a general case, which would require an overly complicated ad-hoc adaptation of the RCRs to make them useful on this specific platform. Our goal is to propose solutions that are instead as platform-independent as possible and we believe the approach would be perfectly viable if we had direct access to the

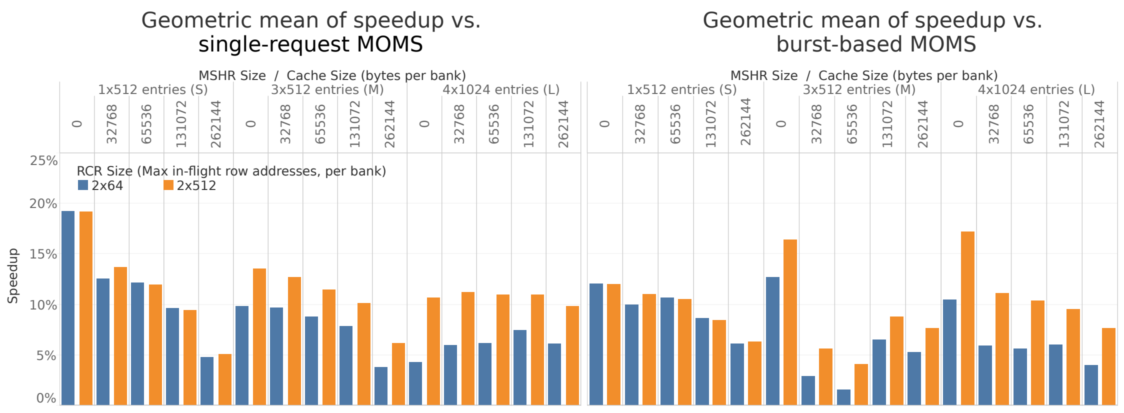


Figure 5.5 – Geometric mean of the speedup of the RCR-based MOMS using the smaller and the larger RCRs across all the benchmarks, compared to the single-request and variable-length burst MOMS. RCRs provide positive geomean speedup on all architectures and compared to both baselines. They are especially 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.

DDR4 controller as we do for the ZC706 PL DDR3 controller, with the possibility of being even more effective if we could control the row policy on a per-transaction basis. Given that this is not possible at the moment, we will focus on the optimization of the irregular accesses on the ZC706 PL memory instead.

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 six 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 a single-request MOMS and one with a maximum burst length of 4 (which provided the best performance on the ZC706 PL system) 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.

5.3.1 Global Speedup and Impact of RCR Size

Figure 5.5 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 to both baselines.

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.

5.3.2 Memory Bandwidth Utilization

To validate the hypothesis that the RCRs provide speedup by increasing the available DRAM bandwidth, we compared usage and availability of the memory interface using single-request MOMSes, burst MOMSes, and MOMSes with 2×512 RCRs. 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 nonidle cycles and, especially, row conflicts. In practice, busy cycles represent unavailable DRAM bandwidth.

Figure 5.6 shows the fraction of active cycles as a function of the fraction of nonidle 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: in the ideal case they would be identical, meaning that there are no busy cycles. 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 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%, hitting the throughput limitation of single requests (see Section 2.6.4). Bursts have a similar benefit and can be even better than RCRs at raw bandwidth saturation: however, if we only consider the 512-bit data beats that are used at least once, RCRs outperform burst-based MOMSes on most data points, which correspond to all benchmarks except for pds-80. This explains the average positive speedup achieved by the RCRs even compared to the burst-based MOMSes.

5.3.3 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 (Figure 5.7) and lowest (Figure 5.8) mean speedup with respect to the corresponding burst MOMS. Even on individual

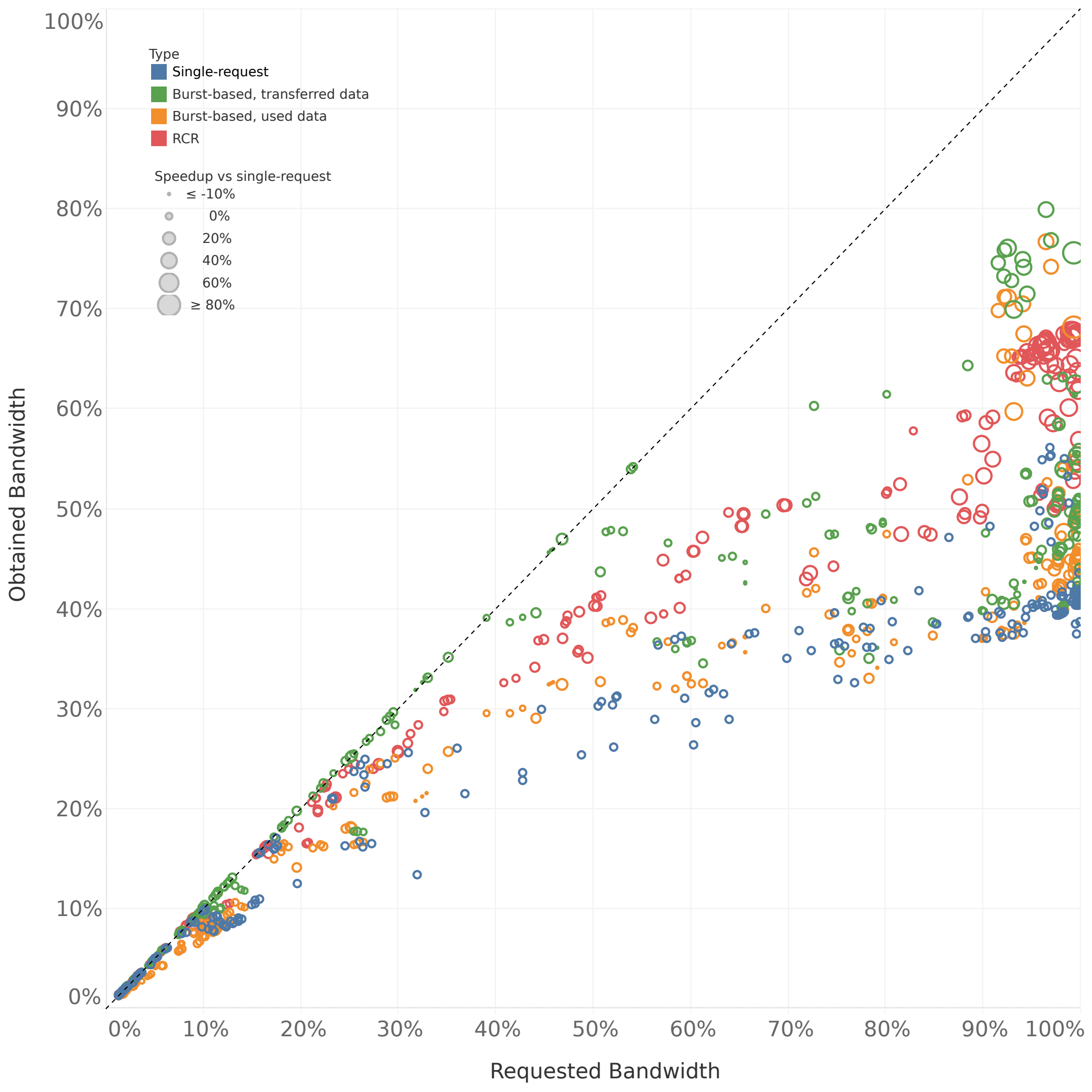


Figure 5.6 – Fraction of requested and obtained DRAM bandwidth during the execution of all benchmarks on all configurations, color-coded by MOMS architecture. The circle size is proportional to the speedup with respect to the single-request MOMS. Backpressure from the DRAM controller, a symptom of row conflicts, severely limits the DRAM bandwidth available to the single-request MOMSes. Both bursts and RCRs provide their highest speedups to data points that need the largest bandwidth. Burst MOMSes have higher bandwidth utilization but waste part of it internally. RCRs increase bandwidth utilization without incurring in data wastage and outperform the burst MOMSes on most data points in terms of bandwidth actually used.

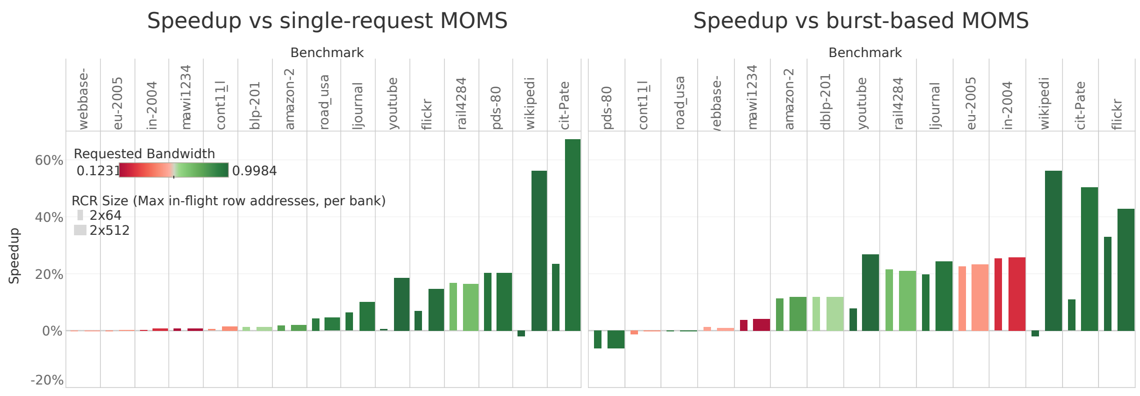


Figure 5.7 – Speedup provided by the RCRs on individual benchmarks, on a configuration particularly favorable to RCRs (system (L) with no cache). The proposed solution never slows down the single-request MOMS; moreover, it outperforms the burst MOMS on most of the memory-bound benchmarks and on two regular benchmarks where bursts, due to data wastage, have a net negative impact on performance.

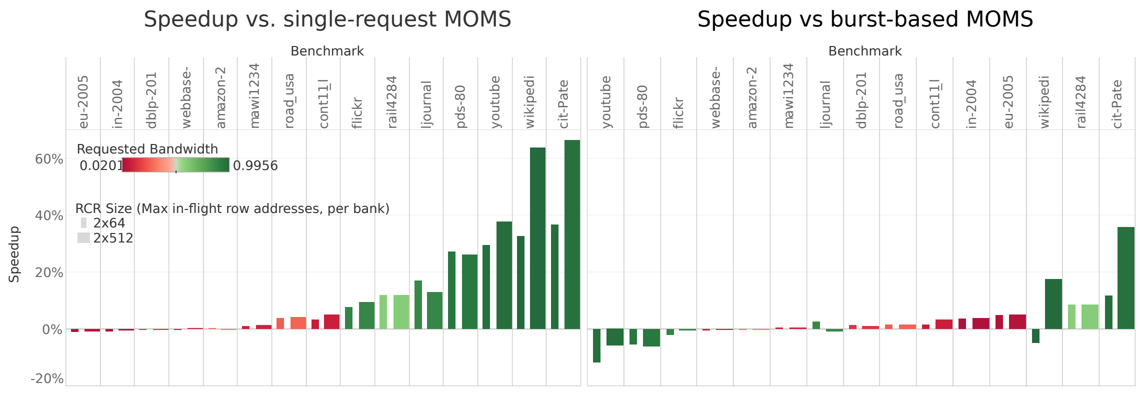


Figure 5.8 – Speedup provided by the RCRs on individual benchmarks, on a configuration where the RCR-based MOMS is moderately competitive to the burst-based MOMS (system (M) with 64 kB of cache per bank). RCRs still have neutral or positive impact compared to the single-request MOMS; while bursts provide 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).

benchmarks, the larger RCR has generally neutral or positive impact, depending on how often the smaller RCR gets saturated. Compared to the single-request MOMS, 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 the burst-based MOMS is more complex: on a few bandwidth-hungry benchmarks, bursts are slightly more efficient than RCRs at increasing performance. As shown in Figure 5.6, 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 Figure 5.8. However, data wastage seems to be more common than useful phenomena as most of the bandwidth-bound benchmarks benefit more from RCRs than from

Table 5.2 – Area utilization of the proposed solution and overhead compared to the baselines (MOMS alone, not counting accelerators and fixed infrastructure). 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.

Configuration	**(S)**		**(M)**		**(L)**	
RCR	**2×64**	**2×512**	**2×64**	**2×512**	**2×64**	**2×512**
LUT						
count	29k	32k	30k	33k	32k	35k
vs. single-request MOMS	+30%	+45%	+36%	+51%	+41%	+56%
vs. burst MOMS	+23%	+37%	+18%	+30%	+20%	+33%
FF						
count	40k	40k	41k	41k	41k	42k
vs. single-request MOMS	+37%	+40%	+36%	+39%	+35%	+38%
vs. burst MOMS	+16%	+18%	+14%	+16%	+13%	+15%
BRAM						
count	272.5	276.5	300.5	304.5	340.5	344.5
vs. single-request MOMS	+4%	+6%	+2%	+3%	-3%	-2%
vs. burst MOMS	+3%	+5%	+1%	+2%	-5%	-4%
DSP						
count	20	20	28	28	32	32
vs. single-request MOMS	+16	+16	+16	+16	+16	+16
vs. burst MOMS	+16	+16	+16	+16	+16	+16

bursts, sometimes by a significant margin, with the notable exception of pds-80 which is indeed the only benchmark where burst-based MOMSes achieve higher used bandwidth than RCRs even after factoring in data wastage. At the other end of the spectrum, most regular benchmarks are insensitive to both RCRs and bursts as the single-request MOMS already provides them with enough bandwidth. However, in the case of eu-2005 and in-2004 in Figure 5.7, data wastage is such that bursts 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.

5.3.4 Resource Utilization

Table 5.2 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 (L). 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 5.2.2), 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, which are still a negligible amount compared to the 900 DSPs available on the ZC706 FPGA.

5.4 Conclusion

As discussed in Chapter 1, compute engines have consistently outperformed memory in throughput for decades, and FPGAs are no exception. To make things worse, Section 2.3 showed that the DRAM bandwidth decreases even further when applications perform irregular memory accesses, hindering most of the benefits of hardware acceleration. Our MOMSes 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, single-request MOMSes discussed in Chapter 3 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, as presented in Chapter 4, mitigates the problem but incurs in data wastage, which reduces or may even cancel out any advantage. In this chapter, 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 thanks to the deep memory request queue exposed by the miss-optimized memory system. This maximizes the opportunities for DRAM row reuse and therefore the available bandwidth without incurring in data wastage, extending the advantage of using MOMSes to support throughput-oriented parallel accelerators with irregular memory access pattern. After having discussed techniques to increase the utilization of the memory channel, in the next chapter we will discuss how to maximize the utilization of FPGA resources when scattered across multiple dices and how to reduce contention on the MOMS resources that are shared among multiple accelerators.

6 Going Large: Multi-Die and Multi-Level Architectures

In Chapter 4 we demonstrated our burst-based approach on a 16-bank MOMS, which is the largest system that we could implement on the AWS F1 FPGA at the maximum clock frequency of the memory controllers, 250 MHz. While for up to eight inputs and banks we could simply scale up the system shown in Figure 4.5 with no architectural modifications, doubling the number of inputs and banks to 16 without frequency degradation was not as straightforward. This is despite the AWS F1 FPGA has $4--5\times$ more resources than the ZC706 FPGA and is two generations ahead (Xilinx UltraScale+ versus Zynq-7000). In addition, the maximum throughput on all systems is never greater than 60% of the theoretical peak of one response per bank per cycle, even on the highest performing benchmarks.

In this chapter, we discuss two design enhancements that improve MOMSes scalability, allowing them to fully take advantage of large multi-die FPGAs and to get closer to the theoretical throughput upper bound. Firstly, we describe how to floorplan a MOMS over a multi-die FPGA and how to handle the die crossings to achieve high resource utilization without clock frequency degradation. Secondly, we discuss alternative MOMS configurations that reduce the impact of bank conflicts: analogous to cache hierarchies, we evaluate private and two-level MOMSes alongside with the shared MOMSes considered so far.

6.1 Spanning Over Multiple Dies Without Sacrificing Performance

When we tried to implement 16 accelerators and banks on the AWS F1 FPGA, the clock frequency dropped from 250 MHz to 189 MHz despite the fact that the utilization of all FPGA resources remained below 30–50%. Looking at the physical layout produced by the placer for this larger system, shown in Figure 6.1, we observe that at least the ten combinational paths with the lowest worst negative slack happen to cross the boundary of a die (or Super Logic Region, SLR, in Xilinx's terminology). Indeed, even though FPGA CAD tools expose multi-die FPGAs as single devices, special care is needed to handle die crossings as they are particularly scarce and slow

This chapter is partly based on the work published at the *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021 [14].

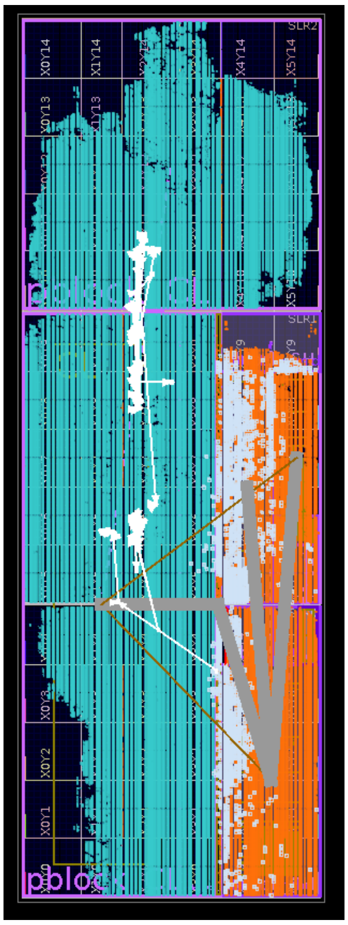


Figure 6.1 – Layout of the placed-and-routed MOMS with 16 inputs and banks on the AWS F1 FPGA when no multi-die-aware design modifications and constrains are used. The three dies are enclosed in purple boxes; the logic belonging to the shell is shown in orange, while the user logic is in blue. The ten paths with the lowest worst negative slack are shown in white: all of them happen to cross a die boundary.

compared to intra-die interconnections. As a result, the presence of multiple dies must be taken into account early in the design process by (1) making sure that all inter-die connections are registered on both ends and do not include any combinational components, (2) minimizing the number of inter-die connections [110], and (3) spreading the logic as much as possible among dies to maximize the total resource availability and minimize routing congestion.

To satisfy (1), on all the signals that connect logic on different dies and do not use handshake signals we inserted two registers back-to-back per inter-die crossing and applied the appropriate RTL attributes to instruct the CAD tool to map them to flip flops rather in LUT-based shift registers that cannot be fractured across different dies (`(* shreg_extract = "no" *)`, in the case of Xilinx). For inter-die connections that have handshake signals, we used the crossing logic shown in Figure 6.2 either described in RTL or, when the crossing occurs between two distinct IPs instantiated in the Xilinx IP Integrator, as a pair of AXI Register Slice Xilinx IPs configured with registered inputs on the destination endpoint as shown in Figure 6.3.

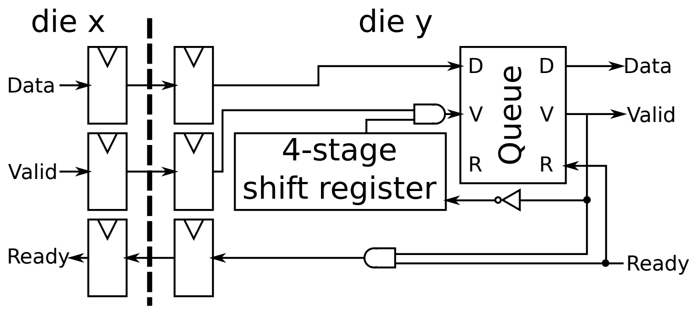
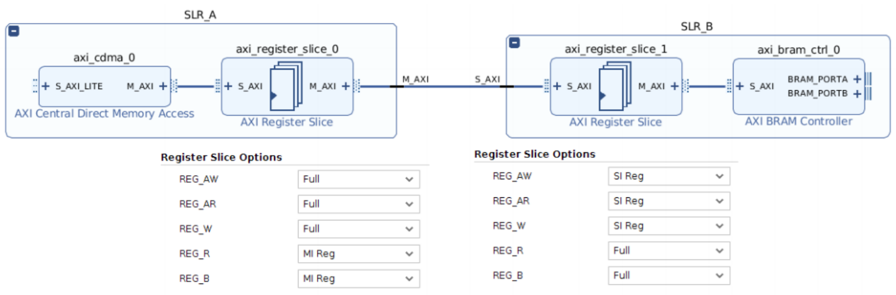


Figure 6.2 – Inter-die crossing circuit for signals with handshake. All the crossings are buffered on both ends with no combinational logic in between. Since it takes two cycles for the ready signal on die y to propagate to die x and, by that time, there may be up to two tokens in the crossing registers, the queue needs at least four slots to buffer all of them.



Register Slice Options

REG_AW	Full
REG_AR	Full
REG_W	Full
REG_R	MI Reg
REG_B	MI Reg

Register Slice Options

REG_AW	SI Reg
REG_AR	SI Reg
REG_W	SI Reg
REG_R	Full
REG_B	Full

Figure 6.3 – Circuit that allow high-performance AXI4 connections that cross a die boundary at the level of the Xilinx IP Integrator (reprinted from the AXI Register Slice documentation [111]). This is equivalent to the custom-built circuit in Figure 6.2 that is used when the crossing occurs inside an IP described in RTL.

As for (2), we statically assign MOMS banks to memory channels and assign banks to the same die as the respective channel. This ensures that the 512-bit data bus from memory channel to MOMS banks does not cross any die boundary. Note that the input ports see a single memory space, which is obtained by interleaving the address space of the multiple memory channels. This provides workload balancing among channels in a manner that is completely transparent to the input ports. Since requests are already assigned to MOMS banks in an interleaved fashion, grouping together MOMS banks that are interleaved consecutively automatically interleaves requests among memory channels without requiring a second crossbar between MOMS banks and memory channels.

Since the AWS F1 FPGA has two DDR4 channels in the central die and one in each of the other dies, the central die hosts twice as many banks than each of the top and bottom dies (eight and four respectively in our final 16-bank configuration). To minimize the number of inter-die crossings to and from the MOMS crossbar, we assigned also the crossbar to the central die. Besides the irregular reads that occur through the MOMS, each SpMV accelerator also consumes three vectors that are read sequentially and produces an output stream of data: as discussed in Section 2.6.3, each memory channel serve the sequential requests of four accelerators. While, as a general rule, we assigned the SpMV accelerators to the same die as the respective sequential

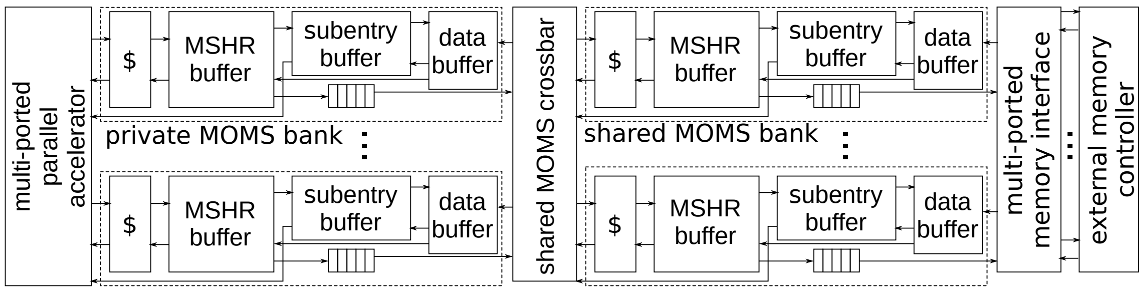


Figure 6.4 – Structure of a two-level MOMS. Private MOMS banks essentially correspond to a single shared MOMS bank that are dedicated exclusively to a single input port. Private-only MOMSes lack the shared MOMS crossbar and banks and are directly connected to the memory interface.

request memory channel, we move to the top die all the accelerators that, following this rule, should be allocated to the central die. Indeed, the central die is already the most congested as it hosts twice as many MOMS banks and has 25–35% of its resources assigned to the AWS shell. Applying these guidelines result in the multi-die-aware system presented in Section 2.6.3 and shown in Figure 2.6.

6.2 Private and Two-Level MOMSes

In the previous section, we showed that ignoring the multi-die structure of the FPGA prevented MOMSes from efficiently utilizing more than 25–40% of the AWS F1 FPGA resources. It appears from Figures 3.7, 3.13, and 4.10 that a similar hard limitation affects the internal bandwidth between accelerators and MOMS banks, on all platforms: indeed, while N_b banks are able to supply up to N_b responses per cycle, the maximum throughput is limited in practice to approximately $0.6N_b$, on all platforms. As explained in Section 3.2.4, sharing the miss handling pipeline between incoming requests and memory responses does reduce the maximum throughput of the miss handling logic. However, in the worst case of zero hit rate and one memory response per channel per cycle (an overly conservative assumption, as shown in Figure 5.6), the throughput would decrease to $N_b - N_{mem}$ requests per cycle. In the case of the PL ZC706 and AWS systems with $\frac{N_b}{N_{mem}} = 4$, this would correspond to 75% of the ideal throughput, which is still significantly higher than the observed 60%. Moreover, even the most regular benchmarks with the largest caches, where the shared miss handling pipeline is rarely used, cannot run at more than 60% of the theoretical throughput.

Rather than in the miss handling logic, we found that this limitation is due to the conflicts among MOMS banks at the level of the MOMS crossbar. Indeed, while the shared MOMS architecture we investigated so far has the advantage of enabling requests from different accelerators to be merged to the same memory request, the main drawback is that all the requests reach the MOMS crossbar and compete for the shared MOMS banks. We will show in Section 6.3.3 that, as a result, as many as 40% of the MOMS banks and accelerators remain unused even in the best case where the bottleneck towards external memory is eliminated.

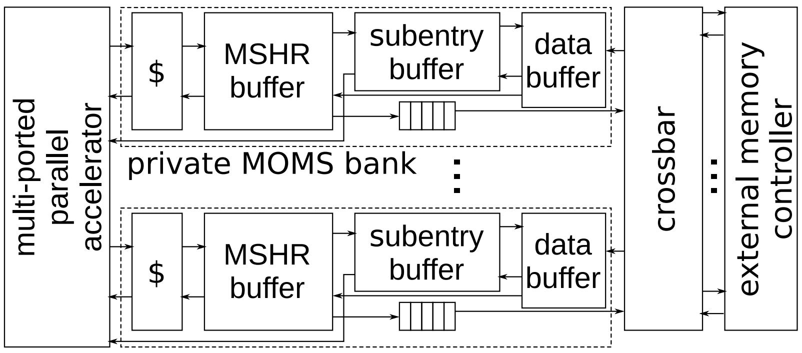


Figure 6.5 – Structure of a private MOMS. The multi-ported memory interface of shared and two-level MOMSes, which contain one round-robin arbiter per output port, is replaced by a full crossbar.

To overcome this limitation, we extend the design space at the system level to include *private* and *two-level* MOMSes. In private MOMSes, banks are moved before the crossbar and become private to each accelerator. This completely eliminates bank conflicts but may result in data duplication inside each private cache and redundant memory requests as only requests from the same accelerators are considered for merging. Two-level MOMSes combine the advantages of private and shared MOMSes: each private MOMS bank processes requests at the same throughput as its accelerator[1] and reduces the number of requests that reach the shared MOMS, resulting in lower contention.

Figure 6.4 shows the resulting system block diagram of two-level MOMSes. Private MOMSes in two-level configurations can process responses out-of-order as they arrive from the shared MOMS; however, the response needs to contain the address of the original request as it is used to retrieve the respective MSHR and to store the response in the private cache. To minimize the amount of information to store in the two-level MOMS as a whole, private MOMS banks do not pair IDs to their requests and the shared MOMS only stores the index of the port that originated the request in the ID field of each subentry. We instead propagate the MSHR tag of the response to the subentry buffers in the shared MOMS, which is normally not necessary in a shared-only MOMS; the subentry buffer will then append the offset of each response and forward the entire address to the private MOMS bank together with the data.

The structure of a private-only MOMS can be derived from that in Figure 6.4 by removing the shared MOMS crossbar and banks and connecting the private MOMS banks directly to the memory channels through a full crossbar, as shown in Figure 6.5. This crossbar is slightly more complex than the set of arbiters used for the same purpose by the shared MOMS as each input must be connected to every output. This is unlike the case of shared MOMS banks, each of which is connected to a single memory channel.

While some of the considerations that apply to private/shared/two-level MOMSes can also be said about caches, there are also important differences related to the different goal of caches and

[1]Neglecting the impact of the miss pipeline sharing inside private MOMS banks; while this is generally a reasonable assumption, it may in fact become the main bottleneck in a few cases that will be presented in Section 6.3.1.

MOMSes. The main goal of caches is to minimize latency, which is possible only if the data is retrieved locally at least in the L2 cache. Incidentally, high hit rate also means high throughput as many requests can be served without extra memory traffic; however, relying solely on the cache for maximizing throughput may require large cache arrays on irregular applications. When latency is essentially irrelevant, secondary misses (or *MSHR hits*) are equivalent to cache hits in that both can be served without extra memory traffic, while requiring less on-chip memory when there is little temporal locality. A two-level MOMS leverages this insight to reduce traffic both towards the memory, as discussed in the previous chapters, and towards the shared MOMS, in a more area-efficient way than if they relied solely on cache hits as a normal cache would. From the perspective of miss handling, two-level MOMSes group misses hierarchically: secondary misses in private MOMS banks are grouped to a single request to the shared MOMS which, in turn, may be a secondary miss in the shared MOMS and thus be grouped with other requests from different ports to the same memory request. This hierarchical grouping has the added benefit of reducing the likelihood of having long linked lists of subentries in any given level, which increases the throughput of the subentry buffers (see Section 3.2.4).

The system-level organization of a MOMS—private, shared, or two-level—is essentially orthogonal to the bandwidth optimization techniques discussed in Chapters 4 and 5, which can still be applied to the *outermost* MOMS level (i.e., the one connected directly to the external memory). Indeed, the effectiveness of both variable-length bursts and RCRs relates the internal architecture of DRAMs and bring no benefit when used behind a second MOMS level.

6.3 Evaluation

After comparing the throughput of various two-level, shared, and traditional architectures in Section 6.3.1, we will assess the contribution of the private and shared caches to the overall performance in Section 6.3.2. We then discuss the impact of private MOMSes on the contention on the shared MOMS in Section 6.3.3 and conclude in Section 6.3.4 with the resource utilization of the new architectures.

6.3.1 Architecture Exploration

Table 6.1 presents the configurations that we evaluated on the AWS F1 system. In general, we targeted the same configuration for private and shared MOMSes, whenever existing, except for configuration 20/8 2L where we had to reduce the number of shared MSHRs and remove the private cache to obtain a routable design. Private-only systems with 20 accelerators were not routable. Like the 16/16 S configuration already used in Chapter 4 and shown in Figure 2.6, two-level and shared systems have a quarter of the accelerators in the bottom die and the rest in the top die. Private systems have instead a quarter of accelerators in the middle die and only half of them to the top die, as the lack of a shared MOMS result in a less congested central die. All MOMSes are burst-based with maximum burst length of 8, which in Chapter 4 we found to be the highest performing on this system and application. In two-level MOMSes, private MOMS banks

Table 6.1 – Configurations evaluated in our experiments. The number of MSHRs, subentries, and the cache size are per private MOMS or per shared MOMS bank.

system	accelerators	private			shared			
		MSHRs	subentries	cache	banks	MSHRs	subentries	cache
16/8 2L	16	4x1024	9x4096	128 kB	8	4x1024	11x4096	256 kB
16/8 2Lt	16	16	8 per MSHR	128 kB	8	16	8 per MSHR	256 kB
16/8 S	16		none		8	4x1024	11x4096	256 kB
16/8 St	16		none		8	16	8 per MSHR	256 kB
16/16 2L	16	4x1024	9x4096	128 kB	16	4x1024	11x4096	256 kB
16/16 2Lt	16	16	8 per MSHR	128 kB	16	16	8 per MSHR	256 kB
16/16 S	16		none		16	4x1024	11x4096	256 kB
16/16 St	16		none		16	16	8 per MSHR	256 kB
20/8 2L	20	4x1024	9x4096	none	8	3x512	11x4096	256 kB
20/8 2Lt	20	16	8 per MSHR	128 kB	8	16	8 per MSHR	256 kB
20/8 S	20		none		8	4x1024	11x4096	256 kB
20/8 St	20		none		8	16	8 per MSHR	256 kB
16 P	16	4x1024	9x4096	256 kB			none	
16 Pt	16	16	8 per MSHR	256 kB			none	

send 64-bit requests to the shared MOMS, which is twice the size of the accelerator request size. Increasing this private-to-shared data size decreases the number of requests hitting the shared MOMS, which reduces contention and the slowdown due to the request/response pipeline sharing inside private MOMS banks (presented in Section 3.2.4). However, it also increases the number of inter-die connections as private MOMSes are on the same die as accelerators and none of them is on the same die as the shared MOMS crossbar. This in practice leads to a sharp drop in clock frequency or even unroutability on most designs above 64 bits.

To widen the design space, we accept timing violations at the target frequency of 250 MHz as long as designs can run at 227 or 218 MHz, which are the immediately lower frequency available from the AWS shell. To account for the frequency differences, we report the performance in GFLOPS instead of multiply-accumulations (MACCs) per cycle as in the previous chapters, where GFLOPS = 2 × MACCs/cycle × f_{clk}.

Figure 6.6 shows the resulting throughput by benchmark and architecture. In geometric mean, the highest performing architecture on most benchmarks is a two-level MOMS; among them, the 16/16 2L is only 10% and 6% faster than the 20/8 2L (with fewer shared MSHRs and no private cache) and 16/8 2L respectively, whereas the 16/16 S is 30% faster than the other shared MOMSes. The 16/8 2L MOMS outperforms the 16/16 S MOMS and the best performing 16/16 2L MOMS is about 10% faster than the 16/16 S MOMS. This confirms that private MOMSes effectively relieve the shared MOMS from part of the workload, which enables using fewer shared MOMS banks than accelerators. Private-only MOMSes are by far the slowest architectures; the gap with the other architectures is smaller on eu-2005 and in-2004 which are the smallest and the most regular benchmarks and thus offer enough private reuse opportunities, as shown in Figure 6.7. These are also the benchmarks where the advantage of two-level MOMSes compared to the shared ones is the largest; more generally, the performance on the private MOMS alone appears to be a good predictor of the benefit of two-level architectures compared to shared-only.

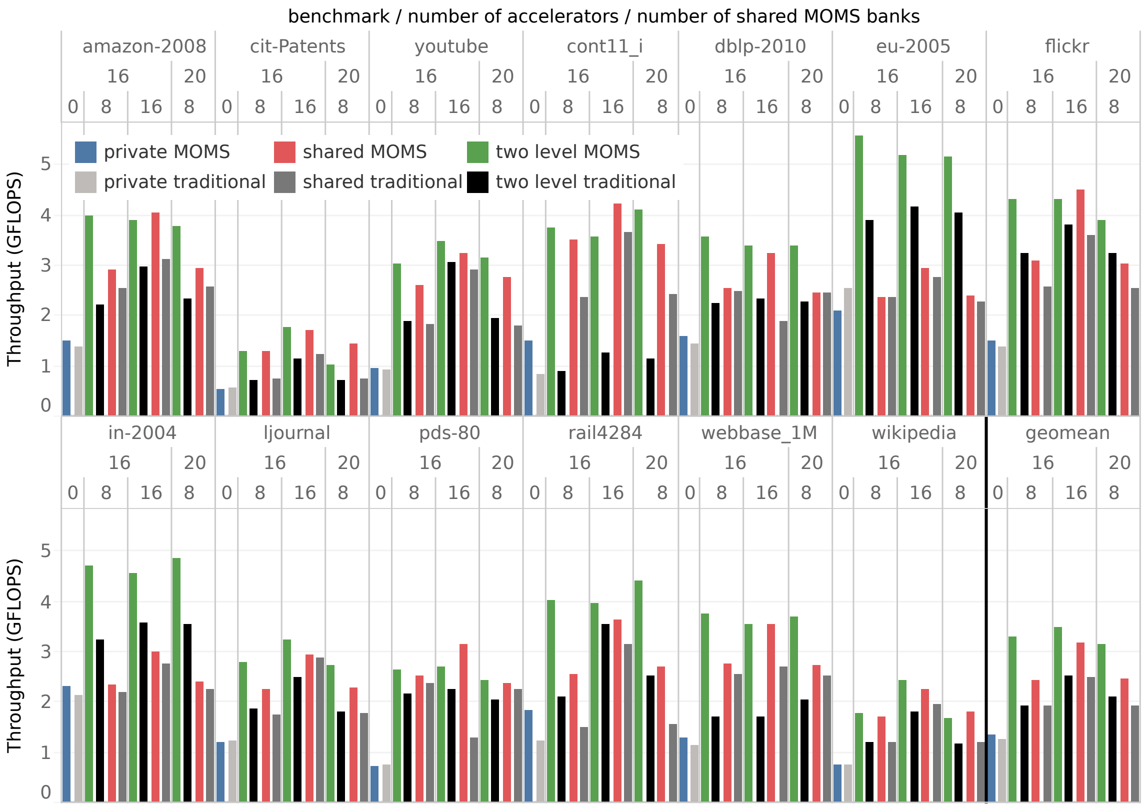


Figure 6.6 – Throughput of the shared, two-level, and private MOMSes and traditional caches presented in Table 6.1. Two-level MOMSes generally outperform shared MOMSes even with fewer shared MOMS banks, whereas two-level traditional caches do not bring significant improvements and are sometimes even slower than shared traditional caches. Private-only systems are always slower as there are not enough private reuse opportunities.

Traditional architectures are generally slower than the corresponding MOMS version and most of the two-level traditional architectures are slower than the shared MOMS, despite the private cache. In fact, on six benchmarks out of 13, two-level traditional caches are even slower than shared ones due to the extra stalls introduced by the private cache when running out of MSHRs or subentries, which throttles the throughput of requests that reach the shared cache.

Figure 6.7 shows that most benchmarks have few private reuse opportunities, which we measure as the ratio of the number of accelerator requests over the number requests generated by the private MOMS. Our workload partitioning scheme, where we interleave consecutive sparse matrix rows across different accelerators, was indeed conceived with shared MOMSes in mind as it requires an inter-accelerator request merging step to exploit any inter-row locality that may exist. Figures 6.8 and 6.9 shows that indeed most of the total reuse can be attributed to the shared MOMS. We tried partitioning the matrix rows in as many blocks as there are accelerators and assigning consecutive rows to the same accelerator: while this increases the performance of private MOMSes and the relative benefit of two-level versus shared MOMSes, it results in lower absolute best performance on most benchmarks due to the fewer reuse opportunities overall, defeating the purpose of maximizing throughput.

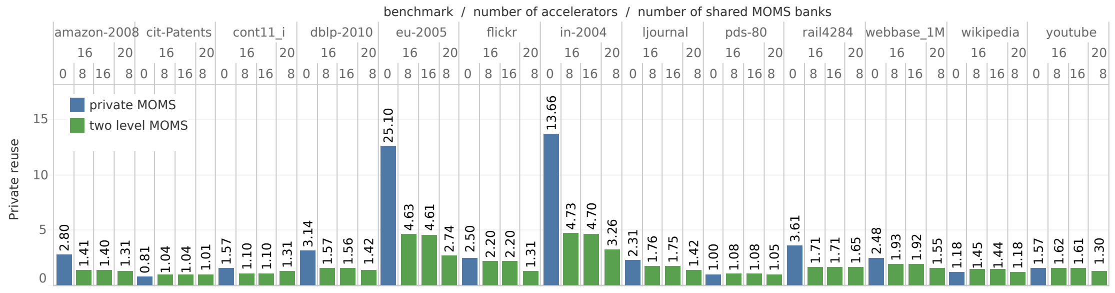


Figure 6.7 – Average number of incoming requests per private MOMS request (private reuse) in the private and two-level architectures. In most benchmarks, the private MOMS of two-level architectures only reduces requests by a factor between 1 and 2; the benchmarks with the highest private reuse are those where the gap between two-level and shared MOMSes is the largest. Private-only architectures achieve a slightly higher reuse thanks to the private cache and the longer output data size (512 bits versus 64 bits). This is however insufficient to outperform shared and two-level MOMSes.

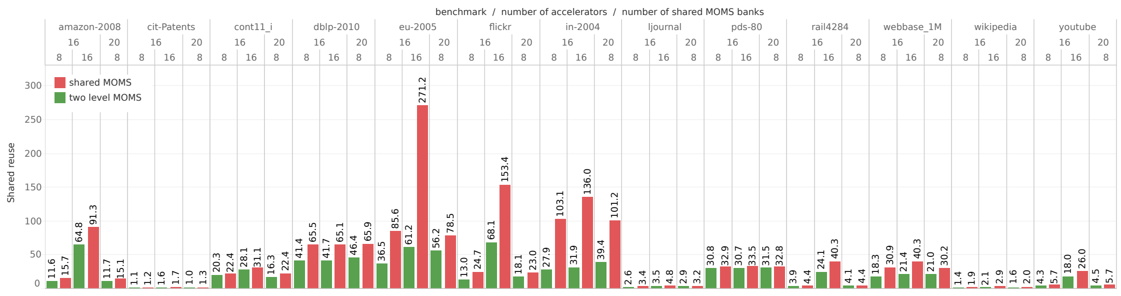


Figure 6.8 – Average number of requests received by the shared MOMS per request generated by the shared MOMS to memory (shared reuse). The shared reuse ranges from a few units to a few hundreds; it is generally lower on the two-level MOMSes as the per-accelerator reuse opportunities have been already harvested by the private MOMS.

A low private reuse can not only make private MOMSes in a two-level architecture useless, but may also introduce a new bottleneck that lowers the performance compared to shared-only MOMSes due to the MSHR/subentry pipeline sharing between requests and responses discussed in Section 3.2.4. While this sharing has usually limited impact on shared MOMSes as most architectures have more banks than memory channels, private MOMSes cannot benefit from a multi-banked structure to share the workload associated to handling the responses. In the worst case where all the requests received by a private MOMS bank are primary misses, requests and responses will be in a 1:1 ratio, meaning that the private MOMS bank will process requests at only 50% of the ideal throughput. This phenomenon is the cause of the low performance of the 16/16 2L system on cont11_i. This is however the only case of performance degradation due to private MOMSes and two-level MOMSes generally outperform, or at least achieve the same performance, as shared MOMSes, usually even with fewer shared MOMS banks.

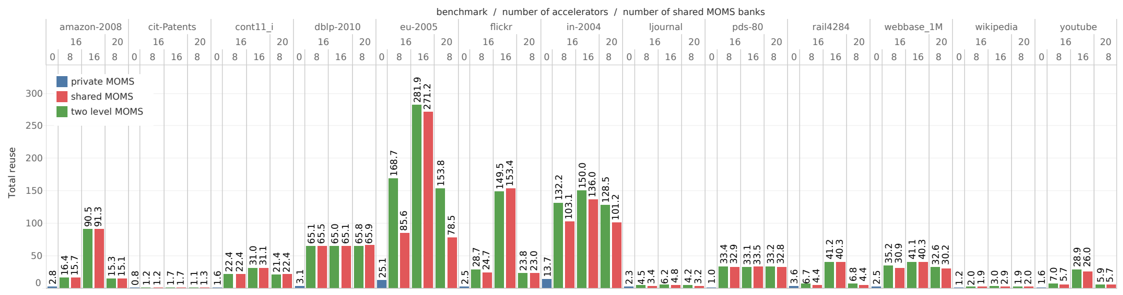


Figure 6.9 – Average number of incoming requests per request sent to memory in all MOMS architectures. Two-level MOMSes generally achieve an overall level of reuse that is higher or comparable to that of shared MOMSes. Lacking a shared grouping stage, the total reuse in private-only MOMSes remain 1–2 orders of magnitude below that of the other architectures.

6.3.2 Impact of Caches

Figure 6.10 shows the throughput only of the MOMSes with and without private and shared caches. In two-level designs, removing the shared cache reduces the throughput by 34% in geometric mean; on the other hand, the impact of the private cache is essentially negligible (3% geometric mean) except on eu-2005 and flickr. In addition, two-level MOMSes without private cache still outperform shared-only MOMSes in most cases and in geometric mean. This suggests that having a large number of MSHRs and subentries at the expenses of a cache is even more useful in private MOMSes than it is in a shared-only MOMS, especially when part of a two-level MOMS. It also confirms our intuition that, when the number of reuses is small, MSHRs and subentries are more effective than caches when it comes to improving throughput.

6.3.3 Contention on the Shared MOMS

To understand the impact of the two-level architecture on contention, Figure 6.11 shows the bandwidth requested from the shared MOMS in 16/16 architectures as a function (a) of the bandwidth requested by the accelerators and (b) of the bandwidth requested at the crossbar input, as summarized at the top of the figure. The meaning of bandwidth *requested* is the same used in Section 5.3.2 for the memory port: it represents the fraction of cycles where the request interface has its valid signal raised, irrespective of whether the downstream logic is applying backpressure or not. The idea is to quantify the amount of contention at the level of the shared MOMS crossbar: in an architecture with as many shared MOMS banks as inputs, when there is no contention, the requested bandwidth at the crossbar output would correspond to that at the input as all the input requests are forwarded to the output at the same throughput. Conversely, in the extreme case where all the crossbar inputs are always active and always targeting the same output among N_b, the requested bandwidth at the crossbar input would be 100% but at the output it would only be $\frac{1}{N_b}$ as $N_b - 1$ banks never receive any request. Therefore, contention for shared MOMS banks introduces a drop in requested bandwidth at the crossbar output as some banks will be inactive while others are serializing requests received from multiple inputs.

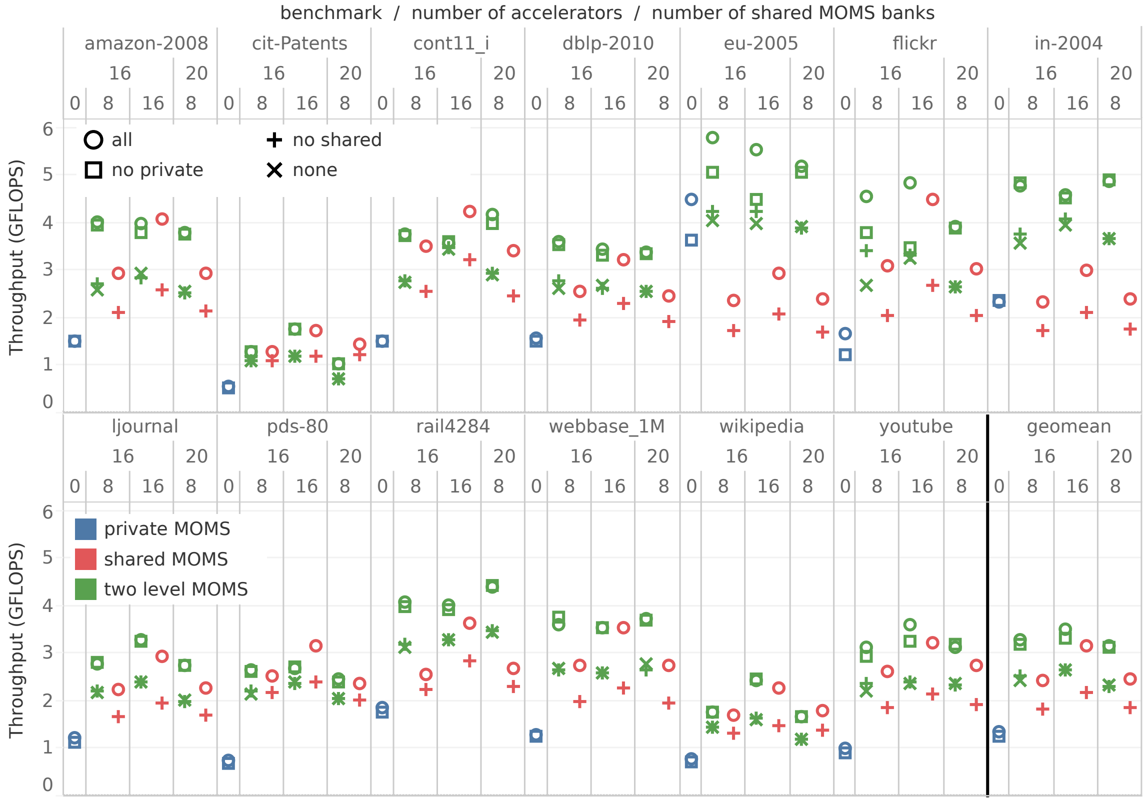


Figure 6.10 – Impact of private and shared caches in private, shared, and two-level MOMSes. While shared caches have a tangible impact on throughput in two-level architectures (34% geometric mean), the benefit of private caches is much smaller (3%) and even cache-less private MOMSes are sufficient for two-level MOMSes to outperform shared MOMSes.

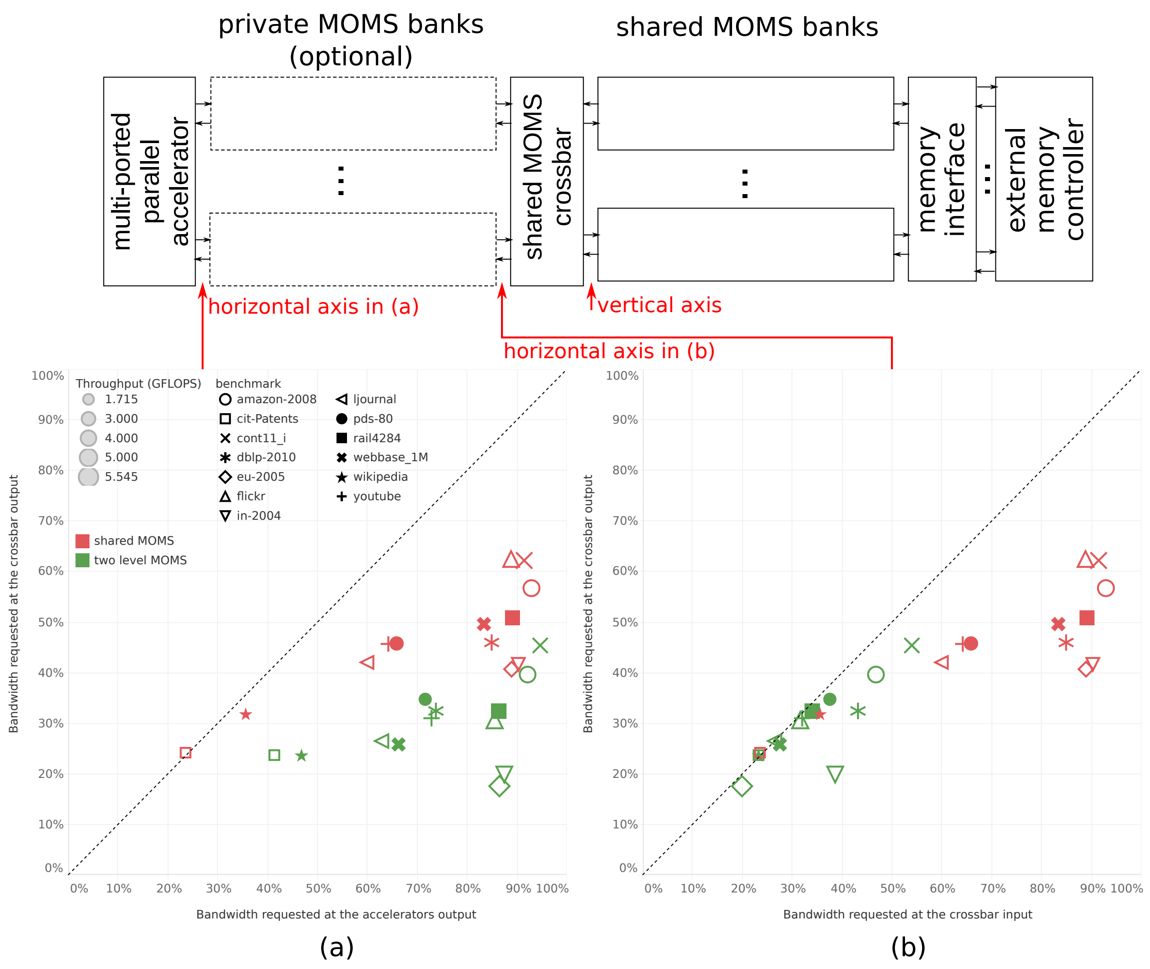


Figure 6.11 – Requested bandwidth at the shared MOMS crossbar output as a function of the requested bandwidth at the accelerators output and at the crossbar input for the 16/16 shared and two-level MOMSes. When the requested bandwidth at the crossbar input is high, the probability for a given output to be requested by multiple inputs, thus leaving some outputs without inputs, increases. In other words, when there is contention, the requested bandwidth at the crossbar output will be lower than that at the crossbar input. In the shared-only configuration, contention is high for most points and there is a large mismatch between the bandwidth requested by the accelerators and that at the input of the shared MOMS. In the two-level configuration, private MOMS banks reduce the bandwidth requirements at the crossbar input and thus contention.

This can be seen graphically in Figure 6.11b as the gap between the ideal line with 45 degrees slope and the actual y coordinate.

In a shared-only architecture, the two representations in Figure 6.11 are identical as the accelerator outputs are connected directly to the crossbar input. On this architecture, two factors can shift points to the left. Very irregular benchmarks with few reuse opportunities such as cit-Pate and wikipedi may reach the maximum number of outstanding reads per accelerator, which corresponds to the reorder buffer (ROB) depth (8,192). Conversely, on high-performing benchmarks such as eu-2005, the DMAs may become the bottleneck if they are unable to match the throughput of the irregular requests. The first factor appears to be the dominating one on the left side of the figure, while the second one for the right side. In the shared-only architecture, only the two leftmost points, which are severely limited by the ROB and by the few reuse opportunities, are essentially on the ideal line. All the other points are significantly below the ideal line and the gap between requested bandwidth at the crossbar input and output is as high as 30%. As a result, the shared MOMS is never requested more than 62% of its available bandwidth, which is consistent with the throughput ceiling that we found in the previous chapters across all benchmarks and on all systems.

In a two-level MOMS, the private MOMS introduces two more mechanisms that reduce the bandwidth requirement at the crossbar input: (3) grouping some of the requests received from the accelerators and (4) throttling the accelerators due to the request/response pipeline sharing. Both mechanisms generally shift all the points down and Figure 6.11a shows that this mostly happens because of a decrease in bandwidth requirement at the crossbar input rather than an increase in contention. Indeed, most points are now on the left side of the chart, where contention is low. In other words, private MOMS banks decrease the bandwidth demand on the shared MOMS to a point where the crossbar can easily handle it without becoming a bottleneck. Mechanism (4), obviously unwanted, is responsible for decreasing the bandwidth at the crossbar input level (i.e., shifting point leftwards from Figure 6.11b to 6.11a for the same benchmark) by a factor that is larger than the amount of private reuse shown in Figure 6.7. This is particularly evident, for example, on cit-Pate.

6.3.4 Resource Utilization and Operating Frequency

Table 6.2 shows the resource utilization of the MOMSes and caches listed in Table 6.1. In the 16/8 and 16/16 systems[2], the net cost[3] of each private MOMS bank is 62%–66% LUTs, 62% FFs, 140% BRAMs and 50% URAMs of that of a shared MOMS bank. The DSP utilization is negligible in all cases as the FPGA contains 6840 DSP blocks and will not be discussed further. The higher BRAM utilization of a private MOMS bank compared to a shared MOMS one is due to the larger

[2]The shared MOMS in the 20/8 2L system has fewer MSHRs than the one in the 20/8 S system, making the comparison less intuitive.

[3]Computed as the difference between the 2L and the S system areas divided by the number of inputs. This accounts for the slight reduction in resource utilization of the shared MOMS in a two-level system compared to a shared-only one due to the narrower ID and offset of each request.

Table 6.2 – Resource utilization and clock frequency of the designs listed in Table 6.1. Each private MOMS bank is equivalent to approximately 60%, 140%, and 50% of a shared MOMS bank in terms of LUTs/FFs, BRAMs, and URAMs utilization.

system	LUTs	FFs	BRAMs	URAMs	DSPs	frequency (MHz)
16/8 2L	138k	170k	598	176	96	250
16/8 2Lt	93k	150k	286	128	0	227
16/8 S	80k	99k	158	96	32	250
16/8 St	56k	92k	22	64	0	250
16/16 2L	193k	236k	754	256	128	227
16/16 2Lt	124k	197k	274	192	0	227
16/16 S	132k	164k	314	176	64	227
16/16 St	76k	155k	42	128	0	227
20/8 2L	154k	180k	356	128	104	250
20/8 2Lt	111k	173k	312	144	0	218
20/8 S	87k	106k	158	104	32	250
20/8 St	65k	100k	22	64	0	250
16 P	149k	227k	356	176	64	227
16 Pt	105k	210k	76	128	0	227

size of the tags in cache and MSHR buffer as a result of the shorter cache line size (64 bits versus 512 bits), which is also the reason for the lower LUT and FF utilization. Removing the private cache, which was shown to have an almost negligible impact of performance, would lower the overhead of each private MOMS bank to 57%–61%, 52%, 59%, and 10% respectively. Therefore, without private cache, the 16/8 2L system would achieve essentially the same performance as the 16/16 S system with 40% fewer on-chip memory bits (+10% BRAMs, -57% URAMs) and similar LUT and FF utilization.

The 16 P MOMS has a slightly higher resource utilization than the 16/16 S one owing to the wide, 512-bit data crossbar that redistributes requests from the four memory channels to the 16 private MOMS banks, which replaces the 32-bit 16/16 input crossbar and the four 4-to-1 512-bit memory arbiters in the 16/16 S system.

6.4 Conclusion

In this chapter, we zoomed out of the MOMS microarchitecture and looked at optimizations opportunities at the system level. Firstly, we discussed how to floorplan MOMSes on multi-die FPGAs to balance the resource utilization across dies and minimize inter-die crossings. This enable MOMSes to actually make use of the large amount of FPGA resources available on multi-die FPGAs with essentially no clock frequency degradation. Secondly, we presented private and two-level MOMSes to eliminate or reduce the impact of contention on the shared MOMS. Owing perhaps to the limited private reuse opportunities, only in eu-2005, flickr, and in-2004 we succeeded in the original goal of bringing the absolute performance above 60%

of the ideal throughput of 16 accelerators, which is 4.8 and 4.35 GFLOPS at 250 and 227 MHz respectively. Nevertheless, we showed that two-level MOMSes outperform shared MOMSes alone by 24% in geometric mean when using the same number of shared banks and that a two-level MOMS with no private cache and half the number of shared MOMS banks has essentially the same performance as a shared-only architecture while using 40% fewer on-chip memory bits. While the advantage of private-only and two-level MOMSes is arguably limited on the SpMV accelerators evaluated so far, we will show in the next chapter that these architectures will be key enablers for FPGAs to compete and even outperform CPUs and GPUs in large scale graph processing.

7 Enabling Efficient Large-Scale Graph Processing

In the previous chapters, we focused on the MOMS architecture and used a variety of SpMV benchmarks that generate widely different access patterns to characterize the impact of every contribution: dramatically scaling up the number of MSHRs and subentries (Chapter 3), sending bursts to memory (Chapter 4), reordering single requests by DRAM row (Chapter 5), and mitigating the impact of internal bottlenecks using two-level architectures (Chapter 6). We showed that MOMSes can bring significant speedups even to very simple accelerators, partly relieving the burden of memory system design on the developer of the accelerator.

In this chapter, we show a concrete example of the impact that MOMSes can have on high-performance computing by showing an example of MOMS-accelerator codesign for large-scale graph processing. We demonstrate that MOMSes can overcome the shortcomings of other memory systems for graph acceleration on FPGA, which are especially evident when dealing with large graphs with tens to a hundred million nodes. We pair the MOMS with a high-throughput parallel accelerator that emits hundreds of thousands of outstanding requests and processes responses out-of-order as they are delivered by the MOMS. The resulting system achieves a 3× geometric mean speedup compared to the state-of-the-art on FPGA, 1.1×–5.8× higher bandwidth efficiency and 3.0×–15.3× better power efficiency than multicore CPUs, and supports much larger graphs than the state-of-the-art on GPUs.

7.1 Limitations of the State-of-the-Art

Graphs are the most effective data representation in a wealth of domains, including social networks [59, 81], drug discovery, [95], genomics [17], and robot navigation [16]; this makes the efficient processing of large graphs crucial in many disciplines.

While graph problems are usually embarrassingly parallel, the performance of graph algorithms on traditional platforms is in practice limited by the bandwidth of the memory system as ei-

This chapter is based on the work published at the *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021 [14].

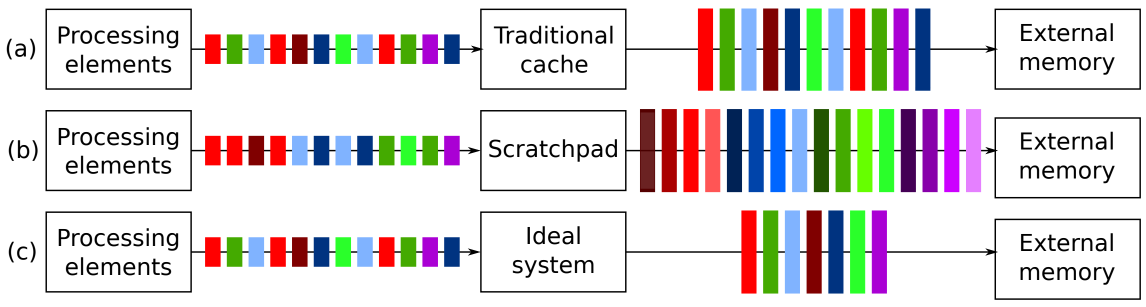


Figure 7.1 – Performance of different memory systems when accesses are irregular. Cache lines belonging to the same tile are identified by different shades of the same color. Traditional caches (a) are effective only when the reuse distance is short enough, which is rarely the case with large-scale graph workloads. Statically-managed scratchpads (b) need strictly ordered accesses and transfer efficiently tiles of data; thus, they guarantee that all accesses are hits, but usually also transfer data that are never used. An ideal, infinite cache (c) would request only useful cache lines and exactly once. Miss-optimized memory systems push caches in this direction for a reasonable area cost.

ther the edge or the node set are typically accessed irregularly [28, 29]. GPUs are also a poor fit for algorithms with irregular control flow and memory access patterns, and require some heavy preprocessing of the graph to achieve good performance [53, 101]. This is problematic on dynamic graphs or in the common scenario where graphs are generated by another application and are used no more than a few times [69]. ASICs provide excellent performance by customizing processing pipelines and the memory system to the access patterns typical of graph processing [38, 116]. However, their fabrication requires months and involves large NRE costs, especially if they are to be implemented on the advanced technology nodes used to evaluate their performance in simulation. While FPGAs cannot reach the performance of ASICs, they are now available in data centers [80, 4, 107]. Today, anyone can deploy immediately and for less than a dollar per hour an FPGA graph accelerator, even tightly integrated in a more complex pipeline directly in the cloud. The accessibility and cost of FPGAs is now comparable to that of GPUs, while retaining the hardware flexibility of ASICs. This makes them an attractive platform to accelerate algorithms with divergent control and irregular memory accesses.

While both vertex- [67] and edge-centric [85] approaches have been proposed on FPGA, the latter choice appears to be more common among recent solutions targeting large-scale graphs [26, 88]. Considering that the edge set is usually larger than the node set, streaming the edges indeed limits the range of the irregular memory accesses to the smallest one of those sets. And, if the entire node set fits in on-chip memory, random accesses to external memory are entirely eliminated. When this is not the case, sorting the edges by source and destination node turns node accesses to sequential; this, however, makes preprocessing more expensive and super linear in the number of edges. Since accesses are generally too irregular to make traditional caches effective (Figure 7.1a), state-of-the-art FPGA accelerators for graph processing [26, 88] mitigate the problem by partitioning the node set in tiles (intervals) and accessing the node set in a tiled fashion (Figure 7.1b). This only requires edges to be partitioned by source and destination interval, which has lower complexity than sorting. However, transferring nodes at the granularity of tiles may cause unnecessary data transfers as not all nodes are always accessed in every iteration. In addition, the number of tile transfers between on- and off-chip memory is quadratic in the number of nodes, leading to even more redundant data transfers. As a result, node transfers dominate the total execution time if the node set is much larger than the amount of on-chip memory.

7.2 Key Ideas

The main idea of this chapter is to use a MOMS to support the irregular read accesses typical of graph processing. Our intuition is that the common skewed edge distribution results in many opportunities for request merging as some nodes are requested orders of magnitude more often than others; however, low-degree nodes are still common enough for traditional caches to stall too frequently and hurt throughput, whereas MOMSes tolerate a large number of misses much better. And, by taking advantage of the dynamic, fine-grained operation of MOMSes, we avoid the redundant data transfers typical of tiling, as suggested in Figure 7.1c. Because the MOMS only handles reads, we still buffer destination nodes in on-chip memory; however, the number of statically scheduled node transfers is now linear in the number of intervals rather than quadratic. This reduces the node transfer overhead for large graphs and, at the same time, lowers the amount of on-chip memory required without increasing the complexity of preprocessing beyond the linear cost of edge partitioning.

Our graph accelerator is adaptable to wide classes of graph algorithms. It is based on multiple out-of-order processing elements (PEs) each handling thousands of hardware threads—one per edge—in a simultaneous multithreading fashion to mask the latency of memory and MOMS. The system has been designed with modern multi-die FPGAs in mind, using the techniques described in Chapter 6. Our graph accelerator has been evaluated on the Xilinx UltraScale+ FPGAs available on the Amazon AWS F1 instances. This sets us apart from prior FPGA solutions which have been tested only in simulation [26, 88] and whose performance is unclear once connected to shells or to DRAM controllers physically constrained to separate dies.

7.3 Graph Processing Model

In this section, we describe the node and edge partitioning that represents the bulk of our lightweight preprocessing (Section 7.3.1; an optional extra node relabeling step will be introduced in Section 7.4.4), the programming model implemented by our accelerator (Section 7.3.2), and discuss how graphs are encoded and stored in the FPGA off-chip memory (Section 7.3.3).

7.3.1 Graph Partitioning

As described more in detail in Section 7.3.2, we adopt an edge-centric model which iterates over the entire edge set and, in principle, may access both source and destination nodes in an arbitrary order. Interval-based partitioning is a common lightweight ($O(M)$) preprocessing

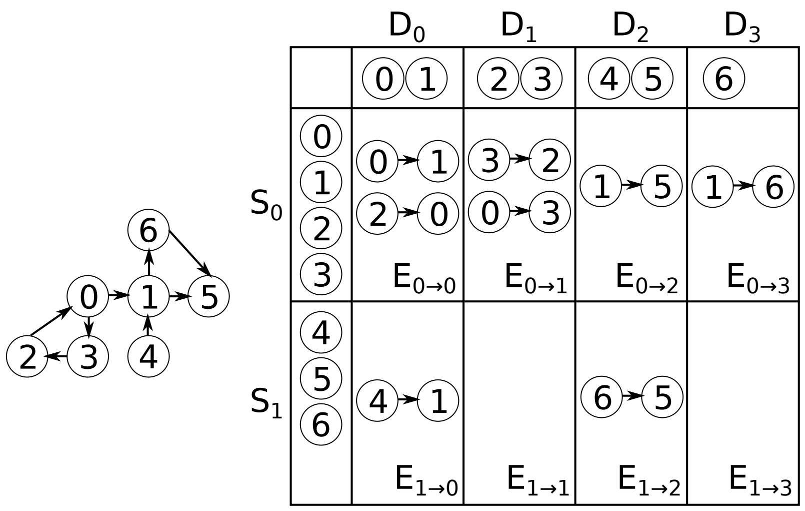


Figure 7.2 – Example of interval-based graph partitioning for the graph shown on the left, assuming N_s = 4 and N_d = 2. Edges are partitioned in shards $E_{s \to d}$ based on the respective source and destination node intervals.

technique that provides an arbitrary degree of locality to the accesses to the node set [60, 85]. Nodes are partitioned in Q disjoint intervals and edges into Q^2 *shards*, where shard $E_{i \to j}$ contains all the edges that have source and destination node in intervals S_i and D_j respectively. Shards can then be streamed while the respective source and destination intervals are both in on-chip memory, which provides high performance irrespective of the access pattern.

Because we use a MOMS to avoid buffering source nodes in on-chip memory, we could, in principle, partition edges in Q shards based on the destination interval alone. However, we keep the source node partitioning for two reasons: (1) to avoid processing edges whose source interval does not contain any node that has been updated in the previous iteration and (2) to apply the edge compression mechanism introduced by ForeGraph [26] and described in Section 7.3.3. Therefore, as shown in Figure 7.2, we partition edges into $Q_s \times Q_d$ shards based on Q_s source and Q_d destination node intervals; such intervals can now have different sizes N_s and N_d since they serve different purposes.

7.3.2 Programming Model

Template 3 presents the programming model implemented by our accelerator. It represents an execution framework that can be configured to implement a variety of graph algorithms by customizing the functions *init*(), *gather*(), *apply*(), the initial node values $V_{DRAM,in}$, a per-node constant vector V_{const}, a global constant scalar const, and two control flags use_local_src and always_active. The model is based on the edge-centric Gather-Sum-Apply-Scatter (GAS) [35, 49] and generalizes the model used by ForeGraph [26] and FabGraph [88]. Table 7.1 shows three examples of graph algorithms implemented using our model. The main purpose of the *init*() function, not present in the original GAS model, is to enable additional optimizations in some algorithms. For example, we can implement PageRank as in ForeGraph [26]: instead of reading

Template 3 Programming Model

```
1: continue = true
2: iter = 0
3: while iter < max_iter and continue do
4:     active_srcs_next = {false}
5:     continue = false
6:     for $d \in [0, Q_d - 1]$ do                                  ▷ In parallel across multiple PEs
7:         for all $i \in D_d$ do                                    ▷ Transferring $D_d$ from DRAM to BRAM
8:             $V_{BRAM}[i] = init(V_{const}[i], V_{DRAM,in}[i], const)$
9:         for all $s \in [0, Q_s - 1]$ do                           ▷ Streaming edges
10:            if active_srcs[s] then
11:                for all $e$ in $E_{s \to d}$ do
12:                    if $e_{src} \in D_d$ and use_local_src then
13:                        new = $gather(V_{BRAM}[e_{src}], V_{BRAM}[e_{dst}], e_w)$
14:                    else
15:                        new = $gather(V_{DRAM,in}[e_{src}], V_{BRAM}[e_{dst}], e_w)$
16:                    if new $\neq V_{BRAM}[e_{dst}]$ or always_active then
17:                        active_srcs_next[src_interval(d)] = true
18:                        continue = true
19:                    $V_{BRAM}[e_{dst}]$ = new
20:        for $i \in D_d$ do                                        ▷ Transferring $D_d$ from BRAM to DRAM
21:            $V_{DRAM,out}[i] = apply(V_{BRAM}[i])$
22:    active_srcs = active_srcs_next
```

both score (PR) and outdegree (OD) irregularly and recomputing the normalized score $d \times \frac{PR}{OD}$ for each source node, we read the constant OD sequentially once upon BRAM initialization and use it to normalize the score before sending it to DRAM. This reduces the size of each irregular read from 64 to 32 bits and allows to compute the normalized score only once per node; denormalizing the score has negligible overhead as it only requires sequential memory operations and can be done only once after the last iteration.

The model supports both synchronous and asynchronous execution, unlike ForeGraph [26] and FabGraph [88] that only support the latter. For synchronous execution, $V_{DRAM,in}$ and $V_{DRAM,out}$ are swapped after every iteration, meaning that the node values that are read during

Table 7.1 – Examples of algorithm-specific parameters for Template 3.

	PageRank	SCC	SSSP
$V_{const}[i]$	$OD[i]$	*not used*	*not used*
initial $V_{DRAM,in}[i]$	$\frac{0.15}{N \times OD[i]}$	i	$i = source\,?\,0 : \infty$
const	$\frac{0.85}{N}$	*not used*	*not used*
(v_c, v_{DRAM}, c)	(c, v_{const})	v_{DRAM}	
$gather(u, v, w)$	$v[0] + u$	$\min(u, v)$	$\min(u + w, v)$
$apply(v)$	$0.15 \times \frac{v[0]}{v[1]}$	v	v
use_local_src	false	true	true
always_active	true	false	false

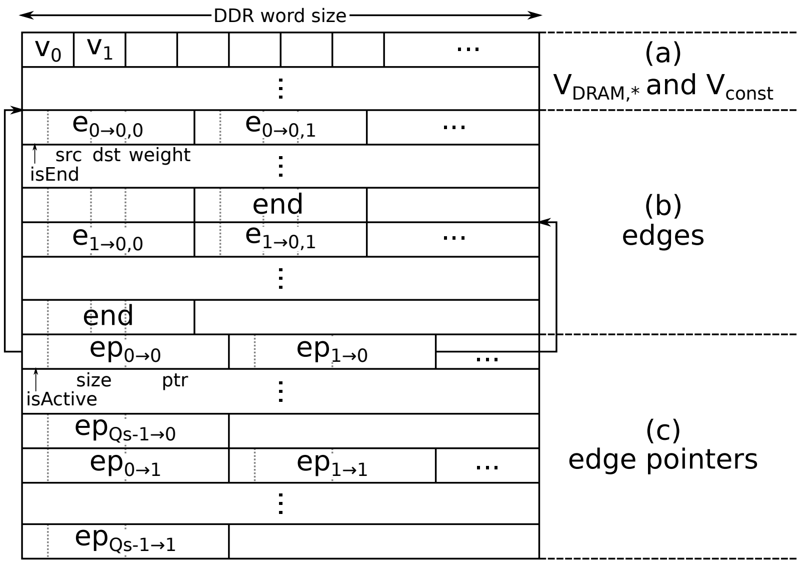


Figure 7.3 – Graph layout in memory, consisting of (a) node initialization values, (b) edges in compressed format and organized by shard, and (c) edge pointers.

execution are updated only at the end of each iteration. For asynchronous execution, $V_{DRAM,in}$ and $V_{DRAM,out}$ point to the same array in memory: as a result, the *gather*() function at line 15 will read updated values as soon as they appear in DRAM. If, in addition, (1) V_{BRAM} and $V_{DRAM,in}$ use the same format and (2) the algorithm remains correct even when *gather*() uses partial node values, use_local_src can be set to true: whenever the source node is in the current destination group, it will be read from the local BRAM, using the most up-to-date version available in the system and reducing the traffic to DRAM. For the examples in Table 7.1, SCC and SSSP satisfy both requirements while PageRank never satisfies (2) as partial scores in BRAM may underestimate the final score.

7.3.3 Graph Encoding and Memory Layout

Our accelerator accepts graphs described in coordinate format (COO): a list of tuples (`src`, `dst`, `weight` (optional)), one per edge. Our preprocessing only requires edges to be partitioned according to their shard; this preprocessing has $O(M)$ complexity as opposed to other approaches [38, 116] that require edges to be sorted at least by source node (sometimes implicitly if graphs have to be converted to CSR format), which has $O(MlogM)$ complexity.

The entire memory layout is shown in Figure 7.3. The first section contains the vertex arrays: the initial $V_{DRAM,in}$, V_{const} (if used by the algorithm) and allocates memory for $V_{DRAM,out}$ if the execution is synchronous. This is followed by all the edges organized by shard. Because the highest bits of source and destination nodes are implicit in the shard, each edge explicitly stores only the offsets (i.e., the lowest bits) within the respective source and destination groups. Since each word retrieved from the DRAM is usually wide enough to contain multiple edges and because the number of edges in a shard is not necessarily a multiple of the number of edges

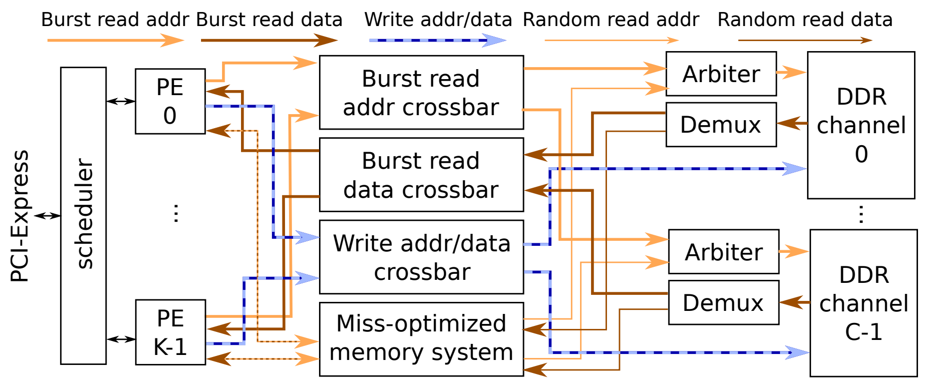


Figure 7.4 – Top-level system architecture. PEs pull jobs from the scheduler, which exposes a single job per destination interval. Burst reads and writes for node initialization, writeback, and edge streaming are forwarded to the respective memory channel, which are interleaved every 2,048 bytes. Irregular short reads to retrieve the source node values are handled by a MOMS.

per DRAM word, we append a special terminating edge at the end of each shard to ensure PEs will ignore any following data in the last DRAM word. It is indeed not possible for PEs to use an edge counter for this purpose as edges may return out-of-order from multiple DRAM channels (see Section 7.4.2). By using 15 bits for the destination node offset, 16 bits for the source node offset, and one bit for the `isTerminatingEdge` flag, we always use 32 bits per unweighted edge even for graphs that have tens of millions of nodes. This is similar to the edge compression technique used in ForeGraph and FabGraph [26, 88] except for the `isTerminatingEdge` flag to support out-of-order responses from multiple DRAM channels. For weighted graphs, source and destination are followed by the edge weight. Because each shard may contain an arbitrary number of edges, we use an array of *edge pointers* to identify starting address and size of each shard, as well as whether the respective source group is enabled or not (thus whether the shard should be streamed in at all or not, to implement line 10 of Template 3). All this fits into 64 bits.

7.4 System Architecture

In this section, we describe the hardware architecture of our accelerator, starting from the top-level system organization (Section 7.4.1) and down to the internals of a PE (Section 7.4.2), highlighting the challenges involved in managing the out-of-orderness of the memory system due to both the MOMS and to the presence of multiple independent memory channels (Section 7.4.3). Having described the system-level structure and the workload partitioning among PEs, we present two optional preprocessing techniques that trade workload balancing for cache line reuse (Section 7.4.4).

7.4.1 Top-Level Architecture

Figure 7.4 shows the top-level system architecture of our system. We target boards that comprise an FPGA connected to one or more external memory channels. In the latter case, we interleave

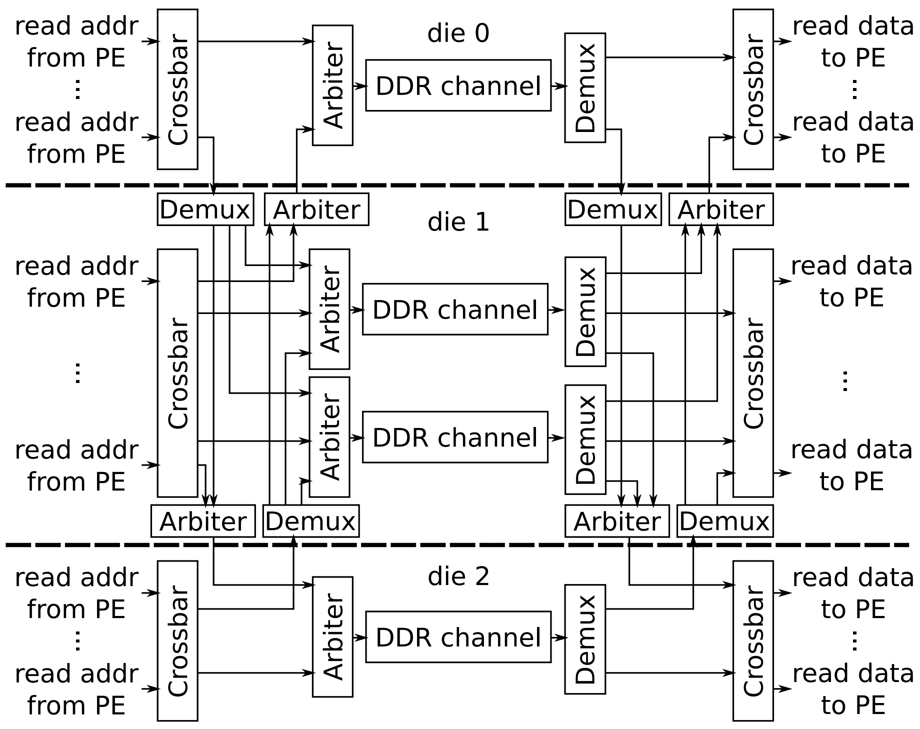


Figure 7.5 – Multi-die aware interconnect architecture for burst reads (the one for burst writes is analogous). Requests and responses are first routed to the target die and then to the target resource within the die.

the addresses of each channel every 2,048 bytes of global address space seen by the PEs to maximize aggregate bandwidth. The interleaving step defines a tradeoff between workload balancing among channels (and thus aggregate available bandwidth) and maximum burst length as we do not split and reassemble bursts across different channels. For our AWS F1 system described in Section 2.6, we found that interleaving the four DDR4 channels every 2,048 bytes, which defines a maximum burst length of 32 64-byte DRAM words, provides a good balance.

The scheduler contains memory-mapped registers that are used to transfer configuration parameters such as (1) the number of node groups and (2) the addresses in memory of the node and edge pointers arrays shown in Figure 7.3. The same interface is also used to start the accelerator and to notify its completion to the main processor. During execution, PEs pull jobs from the scheduler through an arbiter. Each job is associated to a node destination group and consists of (1) the base address from which the PE will read $V_{const}[i]$ and $V_{DRAM,in}$ for its node group (2) the base address in $V_{DRAM,out}$ to which the PE will write the final value of the node group, (3) the base address of the edge pointers for the node group, and (4) the index of the node group, used by the PE to notify the completion of the job. All of these pointers are computed by the scheduler by incrementing the global base pointers by an appropriate stride that depends on the size of the node destination groups and on the number of source and destination node groups.

In order to maximize resource utilization, PEs are scattered across multiple dies. Two distinct paths exist between PEs and DRAM controllers: a multi-die-aware MOMS, used for the random

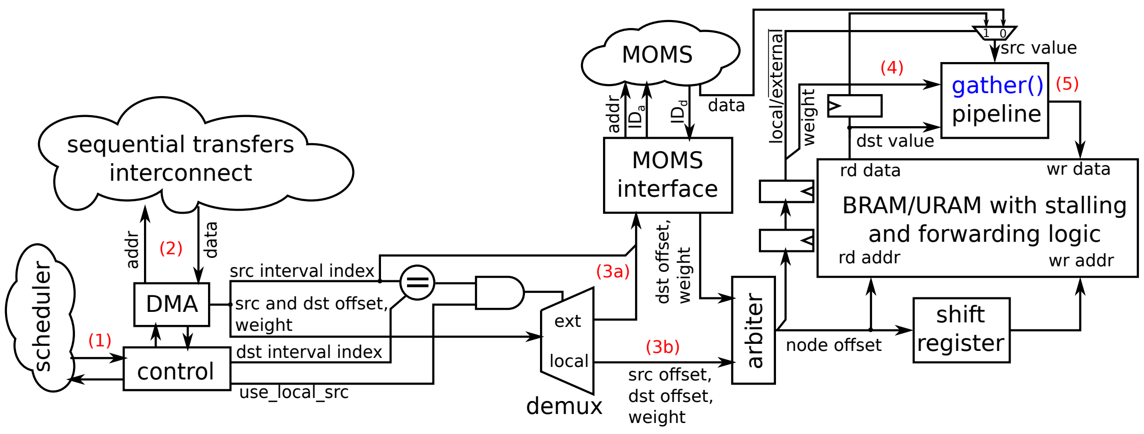


Figure 7.6 – Architecture of a PE. After obtaining a job (1), edges are fetched by the DMA (2) by dereferencing the active edge pointers. Source node values are fetched through the MOMS (3a) unless use_local_src (see Template 3) is enabled (3b). The MOMS interface, shown in Figure 7.7, stores the state associated to each edge while waiting for responses. Once available, node values and edge weight are forwarded to the *gather*() pipeline (4) and the destination node value is updated in BRAM (5). The logic that handles node initialization and writeback is not shown but simply implements direct connections between DMA and BRAM through *init*() and *apply*() respectively.

short reads that retrieve the value of source nodes by dereferencing the edge source indices, and one for burst reads and writes, used in all the other transfers (initial destination node values, edge pointers, edges, and final node values). The multi-die-aware MOMS is described in Section 6.1. The burst network is, from a high-level perspective, equivalent to the bus interconnect logic commonly provided by IP vendors except for being platform independent and multi-die aware. Unlike the interconnect system used by the DMAs of our SpMV accelerators, which was discussed in the previous chapters and is shown in Figure 2.6, every PE must be connected to every other channel, potentially on different dies. This is because jobs are dynamically scheduled to PEs and thus may access data on any channel, unlike the SpMV DMAs, each of which was statically allocated to a single channel. The PE interconnect logic is made multi-die aware by (1) splitting each of the three crossbars (for read address, read data, and write address/data) into a first crossbar per die that routes transactions to the appropriate die through the crossing shown in Figure 6.2 and (2) a set of arbiters per die that forward transactions to the appropriate resource in the same die (DRAM controller or PEs for requests or responses respectively) as shown in Figure 7.5.

7.4.2 PE Architecture

Figure 7.6 shows the internal structure of a PE. Each PE contains a DMA unit that handles all the sequential data transfers: node initialization, edge pointer retrieval, edge streaming (DRAM to PE), and node writeback (PE to DRAM). Upon acceptance of a job, the PE will first read the initial value of all the nodes in the destination group. To minimize the initialization time and because DRAM controllers often expose ports that are much wider than node values, we write four node

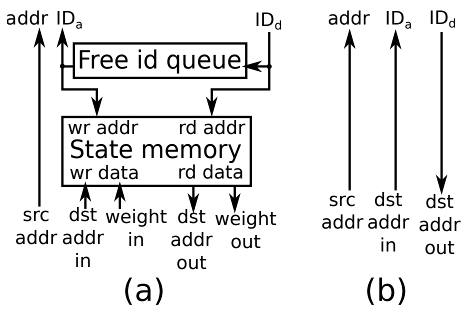


Figure 7.7 – Available MOMS interfaces, which are responsible for retrieving the state associated to each out-of-order MOMS response. For weighted graphs we use the architecture (a), which uses a queue to keep track of the available ids and which stores, for each id, the destination node offset and edge weight in the state memory BRAM. For unweighted graphs, the state reduces to the destination node offset. Considering that its size is comparable to that of the unique ids produced by the architecture (a), we implement the optimized interface (b) that uses the destination node offset directly as an id without duplicating information that is anyway stored in the MOMS.

values per cycle. Once the node initialization is completed, the PE requests edge pointers and, if the respective source group is active, the PE will start requesting edges from that group.

For each received edge, the source node value will be either retrieved from DRAM through the MOMS or from the local BRAM, if use_local_src is active and the source node is in the current destination interval. Once the source node value is provided by either the MOMS or the BRAM, it is sent to the *gather*() pipeline together with the edge state. Because writing the output of the *gather*() pipeline is a read-modify-write operation on the destination node, we use forwarding (whenever possible) or stalling logic to ensure that the *gather*() pipeline always receives the latest version of the destination node value. For algorithms where always_active is false (such as SCC and SSSP), the *gather*() pipeline also returns an `updated` flag, which is set whenever a destination node has been updated and is used to implement line 16 of Template 3.

Once all the edges have been streamed, the destination node memory is written back to DRAM and the PE notifies the completion of the job to the scheduler together with the destination group's `updated` flag, if it exists.

7.4.3 Handling Efficiently Out-of-Order Responses

When the data is interleaved across multiple DRAM channels, responses may return out-of-order whenever they hit multiple channels, even when each individual channel responds in-order. Since nodes must be initialized in a specific order, to prevent out-of-order responses and avoid expensive burst reordering, the PE will never issue more than one outstanding read burst for initial node values. We found this not to be an issue if we use a 64-entry, 512-bit wide DMA queue and issue the next 32-beat burst as soon as the queue has enough space to hold it. Bursts for edges, on the other hand, may be shorter than 32 beats as the number of edges in a shard is not necessarily a multiple of the number of edges in a 32-beat burst and we found that limiting each PE to a single outstanding request for edges results too frequently in an empty read queue.

However, unlike node initial values, edges may be streamed out-of-order, provided that each burst is paired to the corresponding source group as, in our compressed edge format, it defines the high bits of the source node. Therefore, we tag each edge burst request with an ID that is unique for each source group and use the ID that returns with the edge data to stream the right source edge prefix to the downstream logic.

To maximize both MLP and the effectiveness of the MOMS when requesting source nodes from DRAM through the MOMS, the PE must send thousands of outstanding reads to the MOMS. Since each edge can be processed independently, we treat them as separate threads: when the source node data request is sent to the MOMS, we store the thread state (destination node offset and edge weight) and suspend it; when a response returns, we retrieve the respective state and resume the thread execution. This mechanism is implemented by the MOMS interface and the MOMS itself as shown in Figure 7.7. For weighted graphs, we tag each request with a unique ID of size ID_{size} provided by the free ID queue, which we also use to store and retrieve destination node offset and edge weight in the state memory. The BRAM cost per thread of this solution is ID_{size} bits in the free ID queue, ID_{size} bits in the MOMS subentry buffer (see Figure 3.5), plus the 15 bits for the destination node offset and one edge weight in the state memory. For unweighted graphs, the state reduces to the 15-bit destination node offset, whose size is comparable to ID_{size} (as we target thousands of simultaneous threads). Therefore, pairing each destination offset to a unique ID using the circuit in Figure 7.7a would require approximately $3 \times ID_{size}$ bits. We lower this size to ID_{size} bits per thread by using directly the destination offset as an ID, as shown in Figure 7.7b. In other words, we use the MOMS itself to store the entire edge state. By doing so, in addition, the maximum number of threads is only limited by the MOMS capacity instead of the smallest between the capacities of the MOMS and of the state memory.

7.4.4 Node Reordering

Graphs described in coordinate format implicitly assign a unique integer label to each node, which also defines the address of the node value in memory. However, this labeling is, in principle, arbitrary, and while it does not affect algorithm correctness, it has a dramatic impact on performance. Because node values are usually smaller than a cache line, placing nodes that are tightly connected with each other close in the memory space improves the cache hit rate [30] or, in the case of MOMS, the opportunities for memory response reuse. Faldu et al. [30] indeed showed that, in many graph benchmarks, labeling preserves tight clusters.

On the other hand, jobs corresponding to different destination intervals may be processed in parallel; therefore, it is desirable to arrange nodes in destination intervals in such a way that the number of in-edges per interval is as balanced as possible. With respect to Figure 7.2, this means distributing edges as uniformly as possible among columns D_i. Note that, to use bursts to transfer destination intervals between on- and off-chip memory, nodes belonging to the same destination interval should be contiguous in memory as in Figure 7.2. Both ForeGraph [26] and FabGraph [88] statically schedule intervals to PEs and read source nodes from on-chip scratchpads that are private to each PE. Therefore, the workload balancing among PEs is very critical. Because of cluster preservation, paired with the common power-law degree distribution, they found that placing consecutive nodes in the same interval—i.e., computing the destination interval of node i as $n_{i,d} = \left\lfloor \frac{n_i}{N_d} \right\rfloor$—results in a very skewed workload distribution. Therefore, they both propose to use a *hash-based* relabeling such that node i is placed in interval $n_{i,d} = n_i \bmod Q_d$ instead, which results in a more uniform workload distribution.

In our case, however, the PE-level balancing is less critical as jobs are dynamically scheduled to PEs and 1–2 orders of magnitude more numerous than PEs. By letting each PE pull a new job whenever idle instead of forcing each of them to process the same number of jobs, we found that our N_{PE} PEs generally achieve a good workload balance even without hash-based relabeling as long as the largest jobs are smaller than $\frac{M}{N_{PE}}$. In contrast, maximizing cache line reuse becomes more critical. In particular, hash-based partitioning may destroy any cluster that is preserved in the original labeling [30], hurting cache line reuse. Therefore, we keep cache lines as they are and hash entire cache lines among destination intervals.

Orthogonally to cache line reordering, we also evaluate a technique introduced by Faldu et al. called *DBG reordering* [30] prior to cache line hashing to handle graphs whose initial labeling does not preserve tightly connected communities. DBG coarsely partition nodes in 8 groups according to their out-degree, following the intuition that clustering nodes with high out-degree together will lead to higher cache line reuse. This has $O(N)$ complexity, which, for most graphs, is even lower than the $O(M)$ cost of partitioning and cache line reordering. We evaluate the cost and benefit of these techniques in Section 7.5.3.

7.5 Evaluation

After presenting our experimental setup, FPGA-specific die assignments, and benchmarks in Section 2.6, we analyze the impact of the number of PEs and of different MOMS architectures on PageRank, SCC, and SSSP in Section 7.5.2. We then analyze the impact and cost of the various preprocessing techiques (Section 7.5.3), the impact of number of memory channels, and thus bandwidth, on performance (Section 7.5.4), and assess the contribution of the cache arrays to the measured throughput (Section 7.5.5). In Section 7.5.6 we compare our performance with the state-of-the-art on CPUs, GPUs, and FPGAs and we conclude in Section 7.5.7 by presenting the resource utilization and operating frequency of our designs.

7.5.1 Experimental Setup

The system has been written in RTL using Chisel 3 and synthesized using Vivado 2019.1. The code is fully parametric in terms of number of PEs and memory channels and their distribution on the different dies, as well as node size and type, *init*(), *gather*(), and *apply*() functions, MOMS organization, and many other dimensions.

Table 7.2 – Benchmarks properties.

	Benchmark	*N*	*M*
WT	wiki-Talk [63]	2.39M	5.02M
DB	dbpedia-link [57]	18.3M	172M
UK	uk-2005 [27, 19]	39.5M	936M
IT	it-2004 [27, 19]	41.3M	1.15B
SK	sk-2005 [27, 19]	50.6M	1.95B
MP	twitter_mpi [57, 21]	52.6M	1.96B
RV	twitter_rv [59]	61.6M	1.47B
FR	com-friendster [63, 117]	65.6M	1.81B
WB	webbase-2001 [27, 19]	118M	1.02B
24	RMAT-24 [22, 52]	16.8M	268M
25	RMAT-25 [22, 52]	33.6M	537M
26	RMAT-26 [22, 52]	67.1M	1.07B

Our evaluation has been performed on the AWS F1 board described in Section 2.6. The FPGA spans over three dies (*SLR*s in Xilinx terminology), with 25–35 % of the resources of the bottom and central dies reserved for the shell. The central SLR hosts two memory controllers (one of which is in the shell) while the other two dies have one controller each. We assign the shared MOMS crossbar to the central die and each bank to the respective memory channel's die. We found that assigning 30%, 15%, and 55% of the PEs to the bottom, central, and top die respectively provide a good area balancing. Private MOMS, when existing, are assigned to the same die as the respective PE.

Each PE holds 32,768 destination nodes in URAM; each node requires 32 bits in SCC and SSSP and 64 bits in PageRank. For the PageRank PEs, which operate on single-precision floating point, we implemented the *gather*() and *apply*() functions in Table 7.1 using Vivado HLS. Because its *gather*() pipeline has a 4-cycle latency, it may have to be stalled to handle RAW hazards. The *gather*() functions of SCC and SSSP, which operate on 32-bit unsigned integers, are implemented in Chisel and fully combinational, meaning that no stalls are required. The state memory and the free ID queue of the SSSP PEs have 8,192 slots and are implemented in BRAM. We run PageRank for 10 iterations and the other algorithms until convergence.

As benchmarks, we used a set of real world and synthetic large graphs, whose main properties are summarized in Table 7.2. For SSSP, we added random integer weights between 0 and 255 [116]. If not specified, we enable both hashing and DBG.

7.5.2 Architecture Exploration

We performed an extensive design space exploration and we present the most significant design points in Figure 7.8. All MOMSes are of single-request type (as presented in Chapter 3). With RCRs not available on the AWS F1 platform (see Section 5.3), we found the benefit brought by

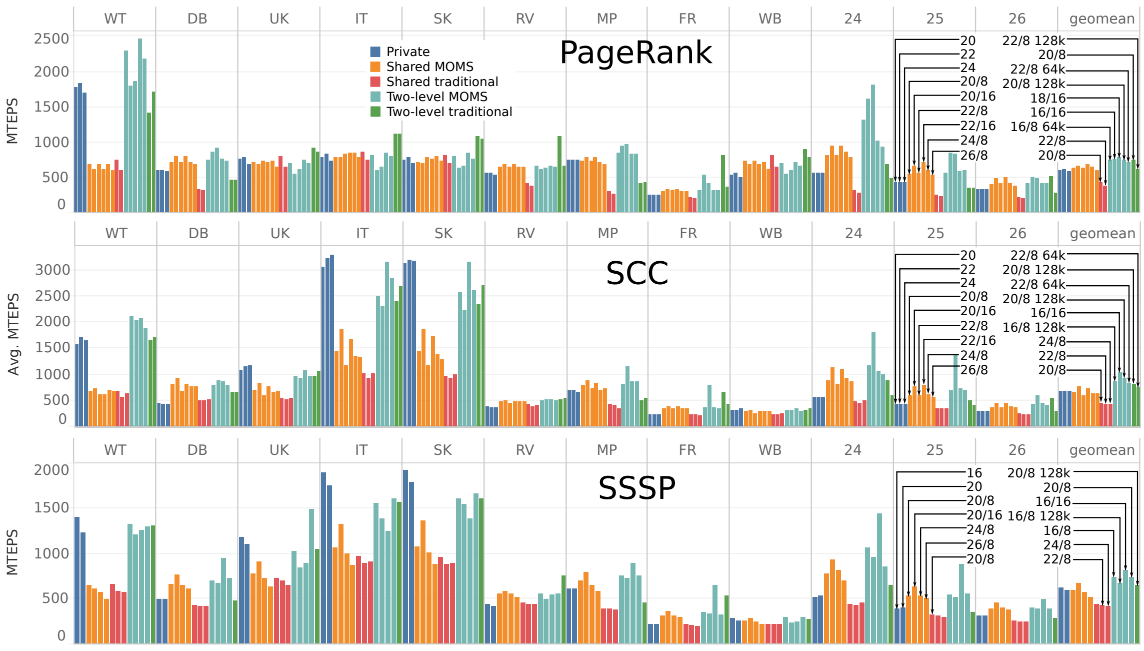


Figure 7.8 – Throughput on PageRank, SCC, and SSSP for different architectures. For shared and two-level architectures, the label **X/Y Zk** indicates X PEs and Y MOMS banks with Z kB of private cache. The two-level architectures with 16 banks generally provide the highest performance, balancing PE peak throughput, amount of conflicts in the shared MOMS, memory efficiency, and routing congestion.

burst-based MOMSes on this application generally insufficient to compensate for the corresponding area and delay increase on this application. We set a target frequency of 250 MHz and discard systems that run at less than 185 MHz.

Each shared MOMS bank contains 256 kB of direct-mapped cache in URAM, 4,096 MSHRs (four 1,024-entry cuckoo hash tables), and 32,768 subentries (4,096 rows with eight subentries each, in URAM). Private MOMSes also have 4,096 MSHRs and 49,152 subentries (4,096 rows with 12 subentries each, in URAM) and have an output data width of 64 bits when part of a two-level MOMS: higher widths dramatically increase the number of inter-die crossing and thus routing congestion, resulting in longer critical paths or routing failures. Private MOMSes have 256 kB of 4-way set-associative cache when there is no shared MOMS; in two-level architectures, we increased the private cache as much as possible until timing degradation. Still, many two-level MOMSes have no private cache at all since they bring little benefit to most benchmarks. Traditional caches have 16 MSHRs and 8 subentries per MSHR per private cache and per shared bank—more associative MSHRs lower the maximum frequency for no performance gain.

As shown in Figure 7.8, two-level architectures provide the highest performance in geometric mean as fewer requests reach the shared MOMS, leading to fewer conflicts, while providing reuse among multiple PEs without extra memory requests. Whenever they can be routed, the architectures with 16 shared banks generally outperform those with more PEs and 8 banks, suggesting that inter-PE conflicts remain critical. This is confirmed by the poor performance of shared MOMSes, which cannot benefit from some filtering from the private MOMS. Private

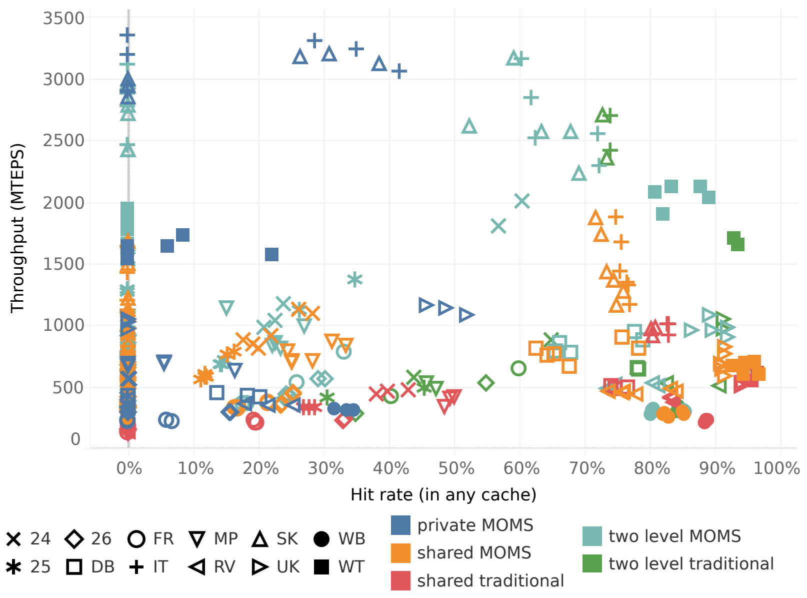


Figure 7.9 – Throughput on SCC versus cache hit rate for the architectures shown in Figure 7.8, including the same architectures completely without caches (with only MSHRs and subentries). Compared to traditional caches, MOMSes achieve higher performance at lower (or even zero) cache hit rate, meaning that caches arrays can be made smaller or even removed altogether with essentially no performance penalty.

MOMSes alone tend to be limited by the excessive amount of redundant requests. There are however important exceptions: IT, SK, and, to a lesser extent, UK and WT perform well also without shared MOMSes as they have higher locality and the benefit from the elimination of conflicts is higher than the increase in memory traffic. They also benefit from traditional two-level caches whenever they leave space for more PEs than two-level MOMSes or run at higher frequency.

SCC achieves the highest throughput among all the applications. PageRank is throttled by RAW stalls due to the 4-cycle *gather*() pipeline, especially frequent on IT, SK, UK, and WB. For SSSP, (1) the overhead of state memory and free ID queue reduces parallelism and operating frequency and (2) weighted edges consume twice the bandwidth than unweighted ones.

Figure 7.9 shows the throughput on SCC as a function of the cache hit rate (in either cache level) for the architectures shown in Figure 7.8. While traditional architectures need high hit rates to reach their peak performance, MOMSes often outperform them despite the lower hit rate, suggesting that caches are less critical in MOMSes. To validate this hypothesis, we also considered the same systems with all the caches deactivated and hence always achieving a 0% hit rate. While traditional caches naturally lose practically all of their performance, MOMSes have little throughput degradation on most benchmarks, meaning that, even in graph processing, thousands of MSHRs can essentially replace caches at a fraction of the area cost when latency is irrelevant.

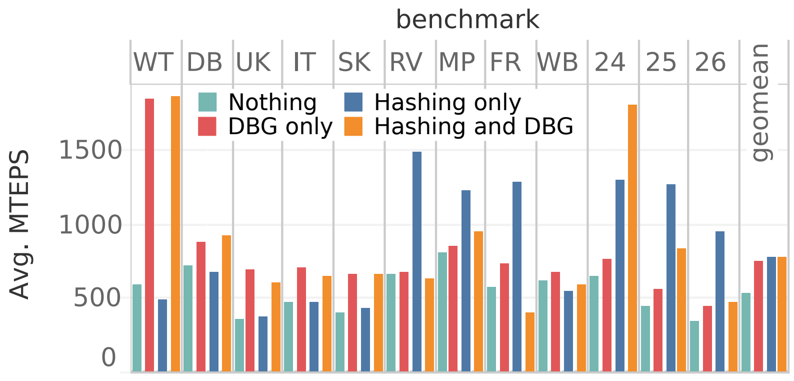


Figure 7.10 – PageRank throughput on the 18/16 two-level MOMS architecture depending on the preprocessing used. Most benchmarks benefit from both hashing and DBG.

Table 7.3 – Preprocessing time in seconds.

	partitioning	hashing	DBG
WT	0.04	0.05	0.03
DB	0.94	1.17	0.31
UK	2.68	6.65	0.99
IT	3.78	6.36	1.32
SK	6.71	12.0	3.40
RV	12.7	10.4	2.76
MP	15.5	14.4	2.55
FR	20.7	18.0	3.06
WB	4.48	4.99	1.43
24	1.58	1.91	0.56
25	3.16	4.66	0.65
26	8.40	7.97	1.58

7.5.3 Preprocessing Cost and Impact

Figure 7.10 shows the PageRank performance on the 18/16 architecture depending on the preprocessing technique used (trends are similar on the other applications). Most benchmarks benefit from hashing, especially the ones with fewer nodes, which results in fewer jobs and a more critical load balancing. In addition, when the labeling does not preserve the original graph communities (FR, MP, RV, and the RMATs) using DBG [30]—thus “breaking" the initial cache lines—provides a significant speedup.

Table 7.3 shows the preprocessing time of all the benchmarks on a 20-core Intel Xeon E5-2698 excluding disk I/O. We use OpenMP to parallelize most of the operations. Our preprocessing is generally lightweight and all the steps besides partitioning are optional, allowing to trade preprocessing time for runtime efficiency or to quickly explore the preprocessing design space to maximize the performance for a given application.

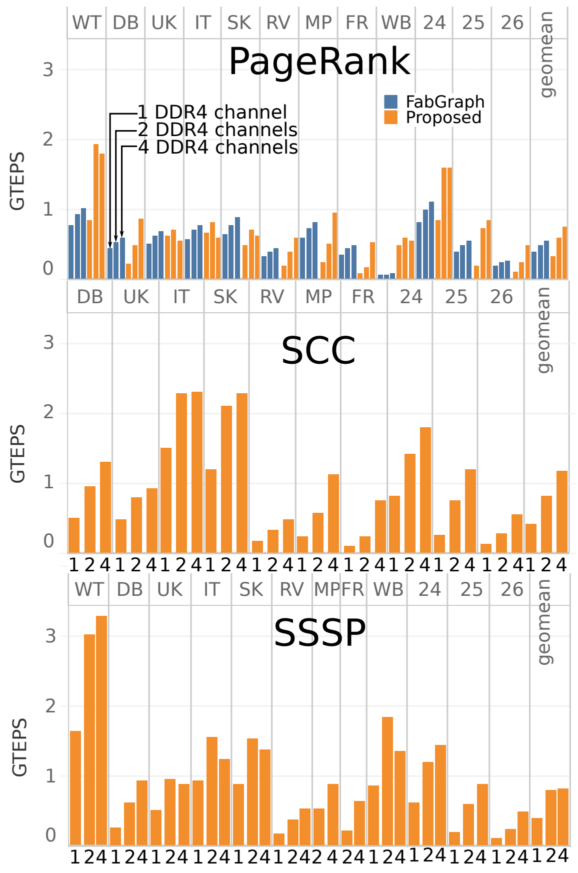


Figure 7.11 – Scalability of throughput as a function of the number of DDR4 channels for the two-level 16/16 MOMS architecture and for PageRank on FabGraph. The 4-channel PageRank and SSSP system have lower operating frequency than the 2-channel ones due to the higher number of die crossings, which increase congestion. For SCC, where the frequency is constant, the throughput of our system generally scales linearly with the available memory bandwidth except for the benchmarks that become compute-bound already with two memory channels (IT, SK, UK, WB, and WT). One 16 GB channel does not have enough memory to run SSSP on FR and MP. FabGraph numbers are optimistic estimations based on the theoretical bandwidth and disregarding any multi-die-related implementation issues and RAW conflicts.

7.5.4 Memory Bandwidth Scalability

Figure 7.11 shows the throughput as a function of the number of DDR4 channels, for the two-level 16/16 MOMS and for PageRank on FabGraph. We used the theoretical model described by Equations (2) to (7) in the FabGraph paper [88] to estimate its performance considering that edges are always active; we compute for it a very optimistic estimation that uses the ideal DRAM bandwidth of 16 GB/s per channel, ignoring the 50% bandwidth limitation imposed by the AWS shell on single request, any other implementation difficulties related to multi-die design, and the handling of RAW conflicts of a floating-point PageRank implementation (FabGraph implements PageRank using integers and thus the initiation interval of its key pipeline is 1 instead of the realistic 4).

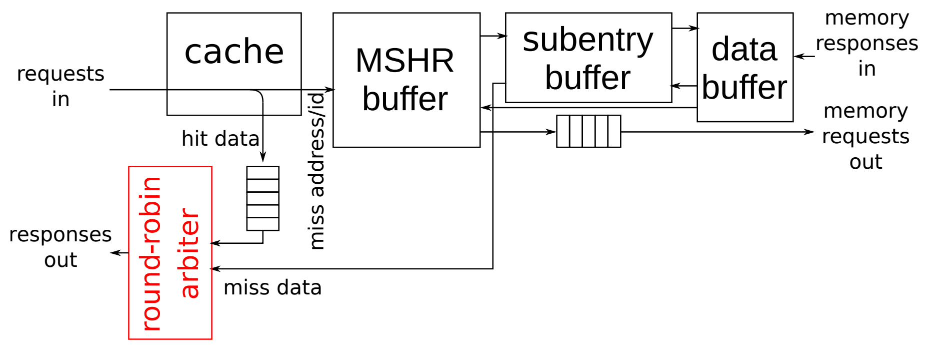


Figure 7.12 – Detail of the hit and miss data paths from MOMS to accelerator. The round-robin arbiter, highlighted in red, becomes the main bottleneck on private MOMSes when there are many reuse opportunities and little contention on the shared MOMS.

We can identify two categories of benchmarks: (1) compute-bound (IT, SK, UK, WB, and WT) and (2) memory-bound (all the others). Benchmarks of the first category have good locality and need less than four channels to achieve peak performance, being rather limited by PE parallelism and, in PageRank, RAW conflicts that occur because our *gather*() function has a 4-cycle latency. These are indeed the benchmarks that benefit the most from private MOMSes or traditional caches in Figure 7.8. On PageRank and SSSP, some compute-bound benchmarks even decrease their performance on 4-channel systems as they operate at lower frequency due to the higher number of die crossings resulting from the use of all dies. The performance of the memory-bound benchmarks, instead, scales essentially linearly with the memory bandwidth. In geometric mean, FabGraph performs better than our system on one memory channel but scales less than ideally because the performance becomes more and more limited by the internal bandwidth between their L1 and L2 cache, between which transfers are particularly numerous on large graphs. In addition, being a simulation-only analysis, it does not take into account how handling multiple dies affects routing congestion.

7.5.5 Impact of Caches

Figure 7.9 already showed that MOMSes do not need high cache hit rates to achieve peak performance and that cache-less MOMSes are often more competitive than traditional caches. Figure 7.13 shows more in detail the impact on performance of adding caches and/or dramatically scaling up the MSHR array by comparing the SCC throughput of a two-level 16/16 MOMS and traditional cache, with and without 4 MiB of shared cache (256 kiB per bank). While the traditional cache has a 2.4× throughput decrease without the cache array, the MOMS maintains essentially the same performance (both in geometric mean), meaning that MSHRs can essentially replace the cache array with little difference in terms of throughput. In addition, the MOMS without cache is 19% faster than the traditional cache with cache array. Both results indicate that, for graph processing, the contribution of MSHRs to throughput is higher than that of cache, despite the area cost of MSHRs and subentries in the MOMS in terms of memory bits is 11% smaller than that of the cache arrays. In fact, IT, SK, and 26 perform even better without

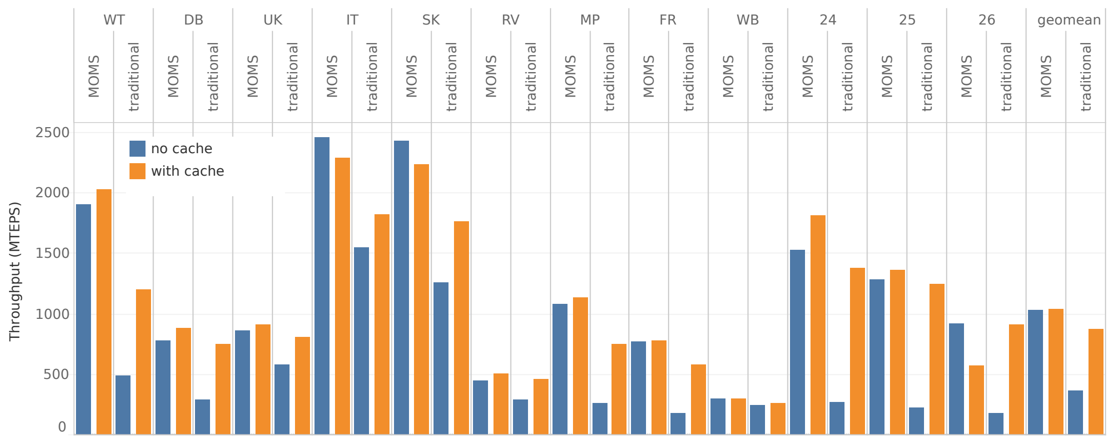


Figure 7.13 – Throughput on SCC for the 16/16 two-level MOMS and traditional cache, with and without shared cache. In geometric mean, the cache array has no impact on the MOMS performance, which is faster than the full traditional cache with cache array. Some benchmarks achieve an even higher throughput without cache because the longer latency allow private MOMSes to accumulate more requests, increasing the private reuse.

shared cache: this happens because shared hits have much lower latency than misses and do not leave enough time for the private MOMS to accumulate enough secondary misses. As a result, the private MOMS need to handle more responses, which steal pipeline slots from requests, as discussed in Section 3.2.4. The solution would be to avoid sharing the pipeline between requests and responses, which we leave as future work.

Figure 7.14 shows the same performance data for the 20/8 two-level system, which has 2.5 MiB and 2 MiB of private and shared cache respectively. The general trends are similar to those discussed for the 16/16 system: large MSHR and subentry buffers are more critical than large caches, and MOMSes without cache have essentially the same performance as the full traditional caches using 25% fewer memory bits. Shared caches are generally more useful than private ones, which is consistent with our findings on SpMV in Section 6.3.2, except for IT and SK which posses significant private reuse opportunities. Perhaps surprisingly, the IT benchmark runs at a slightly higher throughput with no cache at all compared to having a private cache only. In fact, many reuse opportunities at both levels, signaled by the high absolute performance, mean that most of the bottlenecks have been eliminated (memory, shared MOMS contention, request/response pipeline sharing): this uncovers one more point of contention on the return path between responses that are produced by the cache (hits) and by the subentry buffer (misses), which are merged using a round-robin arbiter just before the response output of a MOMS bank, as shown in Figure 7.12. The moderate private hit rate of 37% results in high contention which, due to the high throughput, propagates past the 32-entry queue on the cache hit side and stalls the request input, throttling the incoming requests. Despite this issue, increasing the queue depth resulted in a clock frequency decrease due to the higher congestion and therefore the current queue depth still provides the highest performance.

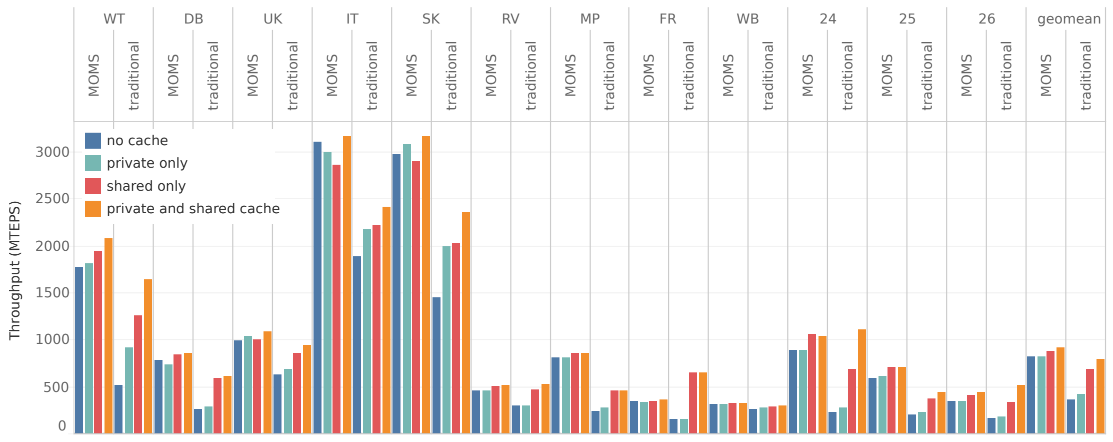


Figure 7.14 – Throughput on SCC for the 20/8 two-level MOMS and traditional cache, with and without private and/or shared cache. As in the 16/16 two-level MOMS, the contribution of caches to the MOMS throughput is essentially negligible, especially for the private cache.

Table 7.4 – Memory bandwidth and power consumption of the platforms considered in Figure 7.15. *The GPU power is overestimated as it includes the power consumed by the entire board.

	Platform	External memory bandwidth	Power
This work, FabGraph	FPGA	64 GB/s	23 W
Gunrock	GPU	900 GB/s	300 W*
Ligra, GraphMat	CPU	233 GB/s	224 W

7.5.6 Comparison with the State of the Art

For each of the three graph applications, we consider (1) the two architecture-preprocessing combinations with the highest geometric mean throughput and (2) the architecture-preprocessing combination with the highest performance on a given benchmark. We consider scenario (1) as representative of a general-purpose, possibly hardened graph processor, while scenario (2) shows the highest possible performance that can be achieved by taking advantage of the reprogrammability of FPGAs and using an architecture that is highly optimized for a specific situation. We compare them to (a) FabGraph [88], (b) Gunrock [101], (c) Ligra [90], and (d) GraphMat [10, 93] whenever the respective graph application is available.

For FabGraph, we use the theoretical model as in Section 7.5.4, which can only be applied to PageRank and ignores RAW stalls. We ran Gunrock on the NVIDIA Tesla V100 with 16 GB of HBM2 memory available on the AWS p3 instances. We evaluated Ligra and GraphMat on a dual socket 2.5 GHz Intel Xeon E5-2680 v3 with 12 cores and 24 logical threads each, connected to 16 1,833 MT/s DDR4 channels. Only Ligra can benefit from DBG and, thus, we added it to the graph.

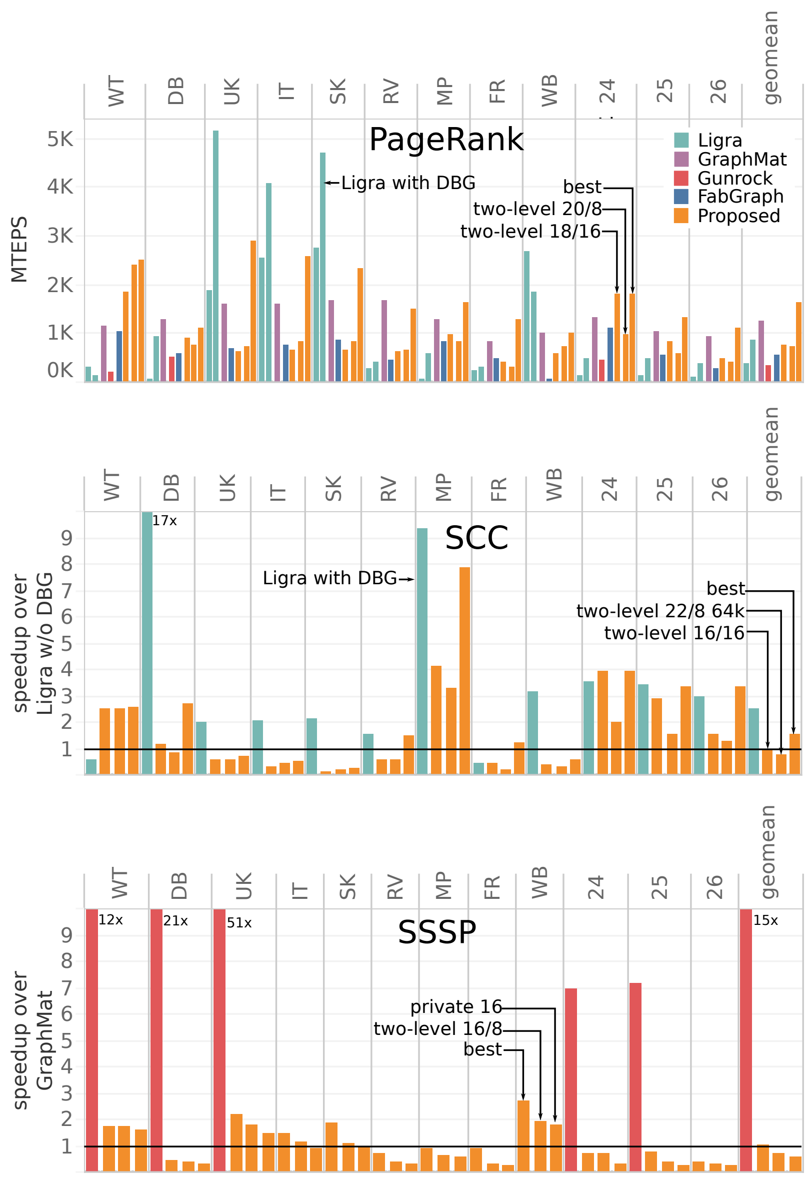


Figure 7.15 – Comparison with state of the art on CPU, GPU, and FPGA. In many cases, our results are, in absolute terms, competitive to or better than those of the best contender. Considering the bandwidth and power gap among the platforms outlined in Table 7.4, our solutions are 1.1×–5.8× more bandwidth-efficient and 3.0×–15.3× more bandwidth-efficient than CPUs and GPUs on PageRank and SCC. Gunrock is more bandwidth- and energy-efficient than our solutions on DB and UK on SSSP but runs only for small benchmarks.

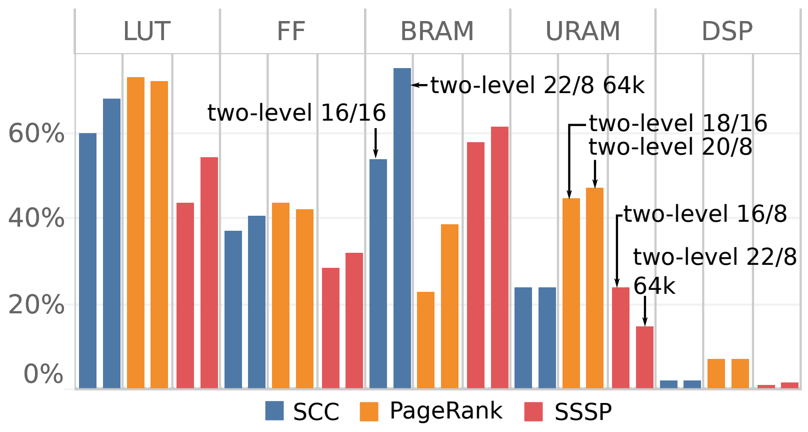


Figure 7.16 – Relative utilization of resources for the top two architectures of each application. Designs are mostly limited by LUTs, used especially in the interconnect networks, and BRAM.

Table 7.4 summarizes bandwidth and power consumption. For the FPGA, we report the maximum power reported by the `fpga-describe-local-image` API [8] while for the CPU we read the Intel RAPL registers [42] using CPU Energy Meter [91]. Both measurements exclude external memory and are therefore comparable. Unfortunately, we could not obtain the same kind of data for the GPU and thus we use the TDP, which corresponds to the absolute maximum consumption of the entire board, including HBM2 memory.

On PageRank, our generic architectures outperform the original Ligra, FabGraph, and Gunrock by 2.1×, 1.4×, and 2.1× (geomean) respectively, while the geomean speedup of the specialized architectures increases to 4.5×, 3.0×, and 4.5× respectively and to 1.3× and 1.9× over GraphMat and Ligra with DBG respectively. On SCC and SSSP, our architectures remain competitive with the CPU baselines in absolute terms and are 1.1–3.5× (generic) and 2.3–5.8× (specialized) more bandwidth-efficient and 3.0–9.4× (generic) and 6.1–15.3× (specialized) more power efficient. On SSSP, Gunrock achieves excellent performance by keeping track of the frontier at the granularity of single nodes as opposed to larger source intervals; however, with only 16 GB of memory, it can only run the five smallest benchmarks, while with the same amount of memory we could run all benchmarks except FR and MP.

7.5.7 Resource Utilization and Operating Frequency

Figure 7.16 shows the resource utilization of the highest performing designs of Section 7.5.6. While LUTs and FFs are mostly used in the interconnect, BRAMs and URAMs are used in both PEs and MOMSes. DSPs are in general underutilized, even in the floating-point PageRank. Note that we report the average utilization across the area not occupied by the shell; the utilization per SLR, which is the main factor that affects routability, is higher and peaks at 90% of LUTs in the central SLR for the two-level 16/16 PageRank system without significantly affecting the operating frequency, which remains between 196 MHz and 227 MHz for all of these designs.

7.6 Conclusion

Graph processing is a key building block of applications in very different domains; yet, achieving good performance is challenging due to the irregular workload distribution, control flow, and memory accesses, especially when graphs are large. We observed that many graph accelerators, especially on FPGA, are based on the assumption that performance practically only comes from statically managed local buffers and/or extensive preprocessing. We challenge this prejudice and show that throughput-oriented out-of-order PEs can instead benefit from the dynamic fine-grained operation of caches and, even more, of MOMSes, without expensive preprocessing. We demonstrate our approach on PageRank, SCC, and SSSP, achieving 3× geometric mean speedup compared to state-of-the-art on FPGAs, 1.1×–5.8× higher bandwidth efficiency and 3.0×–15.3× power efficiency than multicore CPUs, and the ability to scale to very large-scale graphs compared to reference GPU implementations. To the best of our knowledge, our system is the first that can run graph processing on multi-die FPGAs, pushing the boundary of efficient large-scale graph analytics on a single node in the cloud. This chapter also demonstrates the versatility of MOMSes which increase the available bandwidth, and thus speed up, simple sparse matrix-vector accelerators as well as out-of-order graph accelerators using the same exact architecture.

8 Conclusions

With memories consistently lagging behind compute engines in terms of throughput, more and more applications are or will soon become memory-bound. Irregular memory access patterns exacerbate the problem as traditional memory hierarchies based on caches become ineffective and the DRAM bandwidth drops by another order of magnitude. Common solutions generally follow the principles of increasing local data reuse and making external memory accesses sequential; however, concrete implementations of such principles may be hard to obtain and are often application-specific. Therefore, a general solution to alleviate the problem is in high need whenever application-specific techniques are not available or are unacceptably expensive in terms of design effort. The latter aspect is especially important today as FPGAs aspire to become as attractive to software programmers as CPUs and GPUs. While high-level synthesis is making it easier to build efficient compute pipelines using software-like high-level languages, very limited automation is available when it comes to the development of custom memory systems, which still requires extensive computer and memory architecture knowledge.

8.1 A Cache for Throughput-Oriented Applications with Irregular Access Pattern

In this thesis, we present miss-optimized memory systems (MOMSes), a general architecture that increases the throughput that a parallel accelerator can obtain from DRAM memory. Just like caches are the *go-to* architecture when latency is important and when the access pattern has spatial and temporal locality, MOMSes represent an application-agnostic solution when the access pattern is too irregular for traditional caches to be effective, throughput is crucial but latency is irrelevant, and the application is parallel enough to generate a large number of outstanding read memory requests. While this set of requirements may sound restrictive, it includes important applications such as sparse linear algebra, graph processing, and ray tracing.

At its essence, a MOMS is an extreme form of nonblocking cache that can tolerate hundreds of thousands of outstanding misses instead of a few tens. Like traditional nonblocking caches, missing cache lines are requested only once and then used to serve all their pending misses.

The number of misses that can be handled is limited by the number of miss status holding registers (MSHRs) and of subentries, where each MSHR keeps track of one missing cache line and each subentry handles one miss. By dramatically increasing the number of misses that are handled simultaneously, we increase the probability for a miss to target a cache line that has been already requested to serve another miss, meaning that it will be served without generating extra memory traffic. In Chapter 3, we showed that the key to achieve this is to leverage the abundant and high-throughput BRAMs to store the miss bookkeeping information and to make subentry allocation more flexible. Firstly, we moved MSHRs from fully-associative registers to cuckoo hash tables. Secondly, instead of statically allocating a fixed number of subentries per MSHR, we moved all subentries to a central buffer and dynamically allocate them to MSHRs. This avoids stalling the entire miss handling logic if one of the thousands of MSHRs runs out of subentries as a traditional cache would.

By promoting the miss-handling logic from a patch to mitigate the impact of the (ideally, few) misses to an element that is as central as the cache array itself, we introduced new Pareto-optimal points in the area-throughput design space of generic memory systems for throughput-oriented applications. In other words, repurposing some of the BRAMs from cache arrays to MSHRs and subentries can provide significant throughput boosts when the access pattern is too irregular to achieve a hit rate that is high enough for traditional caches to be effective.

8.2 Exploiting Every Bit of Memory Bandwidth and FPGA Resources

The baseline MOMS described in Chapter 3 tackles one of the properties that make DRAMs a poor fit for many applications with irregular access pattern: the mismatch between memory access granularity and the size of the data types that are accessed irregularly. Our MOMSes combine thousands of unrelated short memory accesses, even from different accelerators, to use as much as possible of the long responses returned by the memory.

After focusing on *individual* memory requests, we looked at optimization opportunities *across* memory requests. We indeed observed that many DRAM controllers on FPGA are optimized for burst accesses and do not expose the full DRAM bandwidth when only single accesses are used. Using bursts also increases the reuse of DRAM rows, further increasing the DRAM throughput. In Chapter 4, we extended MOMSes to group incoming requests by small sets of contiguous cache lines. Those cache lines will be retrieved using a single burst request of the shortest possible length, in order to minimize memory traffic. This requires only minimal changes to the MOMS architecture, which however conceptually sets apart MOMSes from traditional nonblocking caches as now MSHRs are allocated at the granularity of bursts instead of cache lines. Using bursts makes MOMSes cost-effective also when the external memory is exposed through narrow ports and improve the performance on most benchmark on all of our experimental platforms, sometimes by a significant margin.

To handle bursts with minimal area overhead, we accept that the MOMS may send some redundant requests or request data that is not used to serve any miss. While the increased bandwidth obtained through bursts usually outweighs the impact of such inefficiencies, they nonetheless mask some of the achievable speedup and may even result in a net slowdown when single-request MOMSes are already very effective. For DRAM controllers that do not excessively penalize single accesses and expose enough details on their DRAM command scheduling policy, Chapter 5 presented an alternative bandwidth optimization technique that does not introduce any bandwidth wastage. Our *row conflict reducers* (RCRs) are to out-of-order DRAM controllers what baseline MOMSes are to traditional nonblocking caches: where general-purpose DRAM controllers reorder some tens of requests to minimize DRAM row conflicts, we operate on the thousands of memory requests that are in the output queue of the MOMS. This maximizes DRAM row reuse, just like the baseline MOMS maximizes cache line reuse, while still operating on individual requests and thus without being affected by the limitations of burst handling. Row conflict reducers provide the greatest speedup where burst-based MOMSes were performing worse and never slow down the baseline MOMS.

Finally, in Chapter 6 we addressed the bottlenecks that are inside the MOMS and the limitations that MOMSes face when scaled to large FPGAs. We discussed how to floorplan a MOMS on a multi-die FPGA and how to handle die crossings to use the large amount of resources available on those FPGAs without clock frequency degradation. In addition to the shared MOMS architecture, where all the accelerator requests are handled by a centralized set of MOMS banks, we also introduced private MOMSes, which handle requests on a per-accelerator basis, and two-level MOMSes which combine private and shared MOMSes. While accelerators face contention when accessing the shared MOMS, they can benefit exclusively from their own private MOMS bank, which can therefore handle requests at the same throughput as they are generated. Private MOMSes alone outperform shared MOMSes when each individual accelerator exposes enough opportunities for reuse, while two-level MOMSes are the best general-purpose architecture as they allow memory requests to be reused by multiple accelerators while being less limited by contention on the shared MOMS.

8.3 Revisiting Caches for Graph Processing

Even though cache parameters such as associativity, number of lines, line size, and replacement policy may require some fine-tuning to bring a given application to its absolute highest performance performance point, their architecture remains essentially the same irrespective of the application, and even a suboptimal cache generally provides some latency and throughput gain as long as the access pattern has temporal and spatial locality. Similarly, while the set of optimal MOMS parameters is often application-specific, the MOMS architecture remains essentially the same across applications and simply implementing the key concept of dramatically scaling up the number of MSHRs is generally beneficial as long as the application is latency-insensitive and can expose a large number of outstanding reads. In Chapters 3 to 6, we proved this for a broad range of sparse matrix-vector multiplication benchmarks; in Chapter 7, we demonstrate another important example of generality by designing a MOMS-based accelerator for graph processing. The accelerator is based on parallel processing elements (PEs) that implement an instance of the commonly used gather-apply-scatter programming model. They can generate thousands of outstanding read requests to fetch source nodes directly from DRAM through the MOMS and process the responses out-of-order. By getting rid of statically scheduled scratchpads that are commonly used in state-of-the-art graph processing accelerators, they obviate the data transfer overheads that become significant on large graphs as the fine-grained operation of MOMSes results in fewer redundant transfers. This ultimately results in higher performance than state-of-the-art solutions on FPGAs and higher bandwidth and energy efficiency than those on CPUs and GPUs. By only requiring a lightweight preprocessing whose cost is linear in the number of edges, our MOMS-based solution is also suitable for graphs that are dynamically updated or that, once generated, are reused only a few times [69].

8.4 Future Perspectives

More applications. We showed that the same MOMS architecture is beneficial to two accelerators that are very different but share the following properties: (1) being bandwidth-bound and latency-insensitive, (2) producing a data-dependent irregular memory access pattern to a large memory space, which makes traditional solutions—caches, on-chip buffering, or data restructuring—either ineffective, impractical, or too expensive, and (3) processing a massively parallel workload and exposing it to the memory system through a large number of outstanding read requests. There are a number of other applications that possess such properties: as a result, they have been particularly challenging to accelerate so far and it may be worth investigating whether MOMSes can alleviate their memory bottleneck. For example, *ray tracing* is a computer graphics application that involves computing the intersection between millions of rays and the millions of triangles in a 3D scene that is the closest to the ray origin. Rays can be processed independently from each other, which provides huge opportunities for parallelism, but quickly diverge in space as they are reflected or refracted, which results in irregular accesses to the data structure that stores the scene [79]. A MOMS could increase the access bandwidth of this data structure in the common case where it is too large to be entirely stored in on-chip memory. From the computer vision domain, one of the kernels of *simultaneous location and mapping* (SLAM) [74] involves thousands of independent irregular accesses to a large 3D array which represents a reconstruction of the environment where a camera is located. *Remapping*, an OpenCV function [78], takes pixels from one place in a 2D image and locates them in another position in a new image: every pixel in the output image can be handled independently from the others and involves one or more reads from the input image, with an access pattern that is as arbitrary as the transformation that is implemented. Xilinx provides an HLS implementation of remapping based on line buffers, whose cost in terms of BRAMs/URAMs can be significant for large offsets between input and output pixel position [115], whereas accessing the input image directly from DRAM through a MOMS may require fewer on-chip memory resources.

More memory technologies and network. There are two factors that make DRAMs a poor fit for applications that perform irregular short memory accesses: (1) their minimum access granularity is large compared to the application's native data size, resulting in high data wastage on every access and (2) only specific access patterns (such as fully sequential or bursts that are long enough) can achieve peak bandwidth. MOMSes mitigate the impact of both by (1) maintaining a pool of thousands of short memory requests to maximize the opportunities to reuse each long memory response and (2) transforming the access pattern generated naturally by the accelerators into another one that the memory can serve with higher bandwidth. While the scales can differ, those factors are not unique to the DDR3 and DDR4 memories that have been considered in our experiments. Therefore, it may be worth investigating MOMS-like architectures to increase the available bandwidth when applications of the kind discussed earlier in this subsection are connected to other memories such as high-bandwidth memories (HBMs) [102], disks, or even network.

Avoid sharing the request-response pipeline. Throughout the chapters we eliminated a number of internal and external bottlenecks that limit the performance of MOMSes. One last internal bottleneck is represented by the pipeline sharing between requests and responses discussed in Section 3.2.4. This is the main bottleneck of private or two-level MOMSes when the private reuse is high and was particularly evident in the scenarios where we measured the highest absolute performance on graph processing (see Figure 7.8). Addressing this issue would require duplicating the ports of the MSHR and subentry buffers by either replicating the memories [61] or double-pumping them. The most challenging aspect would probably be to carefully solve any conflicts that may occur between the two pipelines, a completely new problem as requests and responses are currently serialized into a single pipeline at the input of the MSHR buffer. This would include, for example, handling a new request that reaches the MSHR buffer in the same exact cycle as the data for the same cache line, which would mandate appending a subentry to an MSHR that is going to be deallocated in the same cycle, or traversing a subentry linked list to append a new request while the same linked list is being deallocated.

Improving the input crossbar. The two-level architectures proposed in Chapter 6 overcome the bottleneck introduced by the shared MOMS by reducing the traffic that reach the shared crossbar to a point where it can be handled without slowdowns. An orthogonal solution would be to replace the current shared crossbar with another interconnect that can actually sustain a 100% throughput. The current crossbar consists of one pipelined round-robin arbiter per output, each with a two-slot FIFO in front of every input. While it is formally a crossbar with input queues and virtual output queues (IQ-VOQ), we suspect that the VOQs are too shallow and the scheduling algorithm too simple to achieve a throughput that is tangibly higher than that of a simple IQ crossbar, which is about 58% when requests are independent and identically distributed among a large number of inputs and outputs [50]. Therefore, it may be worth evaluating the benefit of deeper queues, more complex scheduling algorithms (some of which are theoretically capable of achieving 100% throughput [71]), and to replace the crossbar with a network-on-chip, such as the hardened ones available on recent Xilinx Versal FPGAs [114].

Application-specific parameter tuning. We showed that, on SpMV and graph processing, most MOMSes provide some speedup on most benchmarks compared to a traditional nonblocking cache of similar area; however, there is no single MOMS configuration that provides the highest throughput to all benchmarks and the performance differences between MOMSes on the same benchmark can be significant. While we emphasized the generality of MOMSes, it may be advantageous to fine-tune the parameters of a MOMS once the application is defined, especially on FPGAs. We believe machine learning could be a promising direction to explore to address this problem, for example by training a model that, given a relevant set of features of the applications, can enable exhaustive design-space explorations by estimating the performance of every possible MOMS architecture in a matter of seconds instead of the hours required for a full synthesis, place, and route.

8.5 Final Remarks

Compute performance has been improving at a tremendous pace for decades; unfortunately, memory performance has not been following as quickly. While a number of ingenious solutions manage to mitigate or close the gap in many scenarios, notably in the common case where both temporal and spatial locality are present or when memory accesses are known at design time, others still lacked a general solution or methodology to bring the throughput of the memory closer to that of the compute resources. This thesis enlarges the set of applications that can benefit from generic memory system optimizations, widening the scope and decreasing the cost of hardware acceleration.

Bibliography

[1] 360 Research Reports. Hybrid Memory Cube (HMC) and High-bandwidth Memory (HBM) Market 2020. https://www.wfmj.com/story/42593932/hybrid-memory-cube-hmc-and-high-bandwidth-memory-hbm-market-2020-cagr-of-276-with-top-countries-data-latest-trends-market-size-share-global-industry, 2020.

[2] Tanuj Kr Aasawat, Tahsin Reza, and Matei Ripeanu. How well do CPU, GPU and hybrid graph processing frameworks perform? In *Proceedings of the 2018 International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 458–466, Vancouver, BC, Canada, 2018.

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

[4] Amazon.com, Inc. AWS announces seven new compute services and capabilities to support an even wider range of workloads. https://press.aboutamazon.com/news-releases/news-release-details/aws-announces-seven-new-compute-services-and-capabilities.

[5] Amazon.com, Inc. What's the spec of the FPGA on-board DDR in F1—AWS FPGA Development Discussion Forum. https://forums.aws.amazon.com/thread.jspa?messageID=816012, 2017.

[6] Amazon.com, Inc. AWS Shell Interface Specification. https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md, 2018.

[7] Amazon.com, Inc. F1 DRAM random access bandwidth—AWS FPGA development discussion forum. https://forums.aws.amazon.com/thread.jspa?messageID=897290, 2019.

[8] Amazon.com, Inc. Does fpga-describe-local-image -S 0 -M include DRAM? https://forums.aws.amazon.com/thread.jspa?messageID=950323&#950323, 2020.

[9] AnandTech. AMD Opteron Coverage - Part 1: Intro to Opteron/K8 Architecture. https://www.anandtech.com/show/1098/6, 2003.

[10] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *Proceedings of the 2016 International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322, Chicago, Illinois, USA, 2016.

[11] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the 2014 International Conference for High-Performance Computing, Networking, Storage and Analysis*, pages 781–792, New Orleans, Louisiana, USA, November 2014.

[12] Mikhail Asiatici and Paolo 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*, pages 254–262, 2019.

[13] Mikhail Asiatici and Paolo 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*, pages 310–319, February 2019.

[14] Mikhail Asiatici and Paolo Ienne. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, page to appear, Valencia, Spain, June 2021.

[15] Mikhail Asiatici and Paolo Ienne. Request, coalesce, serve, and forget: Miss-optimized memory systems for bandwidth-bound cache-unfriendly applications on FPGAs. *Transactions on Reconfigurable Technology and Systems (TRETS)*, page to appear, 2021.

[16] Tim Bailey, Eduardo Mario Nebot, JK Rosenblatt, and Hugh F Durrant-Whyte. Data association for mobile robot navigation: A graph theoretic approach. In *Proceedings of the 2000 International Conference on Robotics and Automation*, volume 3, pages 2512–2517, San Francisco, California, USA, 2000.

[17] Raji Balasubramanian, Thomas LaFramboise, Denise Scholtens, and Robert Gentleman. A graph-theoretic approach to testing associations between disparate sources of functional genomics data. *Bioinformatics*, 20(18):3353–3362, 2004.

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

[19] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[20] Calin Cascaval and David A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual International Conference on Supercomputing*, pages 150–159, Phoenix, Arizona, USA, November 2003.

[21] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the 4th International Conference on Weblogs and Social Media (ICWSM)*, Washington DC, USA, May 2010.

[22] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 International Conference on Data Mining*, pages 442–446, Brighton, United Kingdom, 2004.

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

[24] Gabor Csordas, Mikhail Asiatici, and Paolo Ienne. In search of lost bandwidth: Extensive reordering of DRAM accesses on FPGA. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 188–196. IEEE, 2019.

[25] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the 24th International Symposium on Field-Programmable Gate Arrays*, pages 105–110, Monterey, California, USA, 2016.

[26] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 217–226, Monterey, California, USA, 2017.

[27] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[28] Assaf Eisenman, Lucy Cherkasova, Guilherme Magalhaes, Qiong Cai, and Sachin Katti. Parallel graph processing on modern multi-core servers: New findings and remaining challenges. In *Proceedings of the 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 49–58, London, United Kingdom, 2016.

[29] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. Parallel graph processing: Prejudice and state of the art. In *Proceedings of the 7th International Conference on Performance Engineering*, pages 85–90, 2016.

[30] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *Proceedings of the 2019 International Symposium on Workload Characterization (IISWC)*, pages 1–13, Orlando, Florida, USA, 2019.

[31] Keith I. Farkas and Norman P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Chicago, Illinois, USA, 1994.

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

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

[34] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT)*, pages 345–354, 2012.

[35] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Bellevue, Washington, USA, 2012.

[36] Sven Goossens, Benny Akesson, and Kees Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Proceedings of the Design, Automation and Test in Europe*, pages 525–530, Grenoble, France, 2013.

[37] Yongbin Gu and Lizhong Chen. Dynamically linked MSHRs for adaptive miss handling in GPUs. In *Proceedings of the International Conference on Supercomputing*, pages 510–521, 2019.

[38] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual International Symposium on Microarchitecture*, pages 1–13, Lanzarote, Spain, 2016.

[39] hblok.net. Historical Cost of Computer Memory and Storage. https://hblok.net/blog/posts/2017/12/17/historical-cost-of-computer-memory-and-storage-4/, 2017.

[40] Intel Corp. *Hybrid Memory Cube Controller IP Core User Guide*, May 2016.

[41] Intel Corp. *Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*, August 2018.

[42] Intel Corp. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm, 2020.

[43] Intel Corp. Intel® Stratix 10 Embedded Memory User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-memory.pdf, 2021.

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

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

[46] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[47] JEDEC. DDR3 SDRAM standard JESD79-3F. https://www.jedec.org/standards-documents/docs/jesd-79-3d, 2012.

[48] JEDEC. DDR4 SDRAM standard JESD79-4B. https://www.jedec.org/standards-documents/docs/jesd79-4a, 2017.

[49] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. *Transactions on Knowledge and Data Engineering*, 30(2):305–324, 2018.

[50] Mark Karol, Michael Hluchyj, and Samuel Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, 35(12):1347–1356, 1987.

[51] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[52] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable SIMD-efficient graph processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, pages 39–50, San Francisco, California, USA, 2015.

[53] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–252, Vancouver, BC, Canada, 2014.

[54] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd International Symposium on Microarchitecture*, Atlanta, Georgia, USA, December 2010.

[55] Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the 45th Annual Allerton Conference on Communication, Control, and Computing*, volume 75, pages 751–758, Monticello, Illinois, USA, September 2007.

[56] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, Minneapolis, Minnesota, USA, May 1981.

[57] Jérôme Kunegis. KONECT: The Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, page 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery.

[58] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[59] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, 2010.

[60] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Bellevue, Washington, USA, 2012.

[61] Charles Eric Laforest, Zimo Li, Tristan O'rourke, Ming G Liu, and J Gregory Steffan. Composing multi-ported memories on FPGAs. *Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):1–23, 2014.

[62] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. GPOP: A scalable cache-and memory-efficient framework for graph processing over parts. *ACM Transactions on Parallel Computing (TOPC)*, 7(1):1–24, 2020.

[63] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[64] Sheng Li, Ke Chen, Jay B. Brockman, and Norman P. Jouppi. Performance impacts of non-blocking caches in out-of-order processors. HPL Tech Report, Hewlett Packard Labs, 2011.

[65] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH Computer Architecture News*, 37(3):267–278, 2009.

[66] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. CGPA: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Design Automation Conference*, pages 1–6, San Francisco, California, USA, June 2014.

[67] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 International Conference on Management of Data*, pages 135–146, Indianapolis, Indiana, 2010.

[68] Mario D. Marino and Kuan-Ching Li. System implications of LLC MSHRs in scalable memory systems. *Microprocessors and Microsystems*, 52:355–364, 2017.

[69] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[70] Sally A McKee, Assaji Aluwihare, Benjamin H Clark, Robert H Klenke, Trevor C Landon, Christopher W Oliver, Maximo H Salinas, Adam E Szymkowiak, Kenneth L Wright, William A Wulf, and James H Aylor. Design and evaluation of dynamic access ordering hardware. In *International Conference on Supercomputing*, pages 125–132, 1996.

[71] Nick McKeown, Adisak Mekkittikul, Venkat Anantharam, and Jean Walrand. Achieving 100% throughput in an input-queued switch. *IEEE Transactions on Communications*, 47(8):1260–1267, 1999.

[72] Onur Mutlu. Memory Systems and Memory-Centric Computing Systems — Lecture 1: Memory Trends and Basics. https://people.inf.ethz.ch/omutlu/pub/onur-ACACES2018-Lecture1-Topic1-MemoryTrendsAndBasics-July-9-2018-afterlecture-with-backup.pdf, 2018.

[73] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Computer Architecture News*, volume 36, pages 63–74. ACM, 2008.

[74] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *Proceedings of the 2015 International Conference on Robotics and Automation (ICRA)*, pages 5783–5790, Seattle, Washington, USA, 2015.

[75] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th Symposium on Operating Systems Principles*, pages 456–471, 2013.

[76] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for GPU-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[77] NVIDIA Corp. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[78] OpenCV. Remapping tutorial. https://docs.opencv.org/3.4/d1/da0/tutorial_remap.html, 2021.

[79] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

[80] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 13–24, Minneapolis, Minnesota, USA, 2014.

[81] Louise Quick, Paul Wilkinson, and David Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining*, pages 457–463, Istanbul, Turkey, 2012.

[82] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 128–138, Vancouver, BC, Canada, 2000.

[83] Hemant G Rotithor, Randy B Osborne, and Nagi Aboulenein. Method and apparatus for out of order memory scheduling. US Patent US7127574, October 2006.

[84] Nick Routley. Visualizing the Trillion-Fold Increase in Computing Power. https://www.visualcapitalist.com/visualizing-trillion-fold-increase-computing-power/, 2017.

[85] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th Symposium on Operating Systems Principles*, pages 472–488, Farmington, Pennsylvania, USA, 2013.

[86] Karl Rupp. Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data, 2021.

[87] Jun Shao and Brian T Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 285–294, Phoenix, Arizona, 2007.

[88] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. Improving performance of graph processing on FPGA-DRAM platform by two-level vertex caching. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 320–329, Seaside, California, USA, 2019.

[89] Zhiyuan Shao, Chenhao Liu, Ruoshi Li, Xiaofei Liao, and Hai Jin. Processing grid-format real-world graphs on DRAM-based FPGA accelerators with application-specific caching mechanisms. *Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(3):1–33, 2020.

[90] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming*, pages 135–146, Shenzen, China, 2013.

[91] SoSy Lab - LMU Munich. CPU Energy Meter. https://github.com/sosy-lab/cpu-energy-meter, 2020.

[92] Statista. Distribution of DRAM market revenue worldwide from 2010 to 2020, by architecture. https://www.statista.com/statistics/553383/worldwide-dram-market-share-by-architecture/, 2020.

[93] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11), 2015.

[94] Synopsys, Inc. DDR4 Bank Groups in Embedded Applications - DesignWare Technical Bulletin. https://www.synopsys.com/designware-ip/technical-bulletin/ddr4-bank-groups.html, 2012.

[95] Ichigaku Takigawa and Hiroshi Mamitsuka. Graph mining: procedure, application to drug discovery and recent advances. *Drug discovery today*, 18(1-2):50–57, 2013.

[96] TechPowerUp. AMD Xbox One X GPU. https://www.techpowerup.com/gpu-specs/xbox-one-x-gpu.c2977, 2021.

[97] Michael Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. *JFE Technical Report*, 03 2011.

[98] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 409–422, Orlando, Florida, USA, December 2006.

[99] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *Transactions on Architecture and Code Optimization (TACO)*, 12(4):65, 2016.

[100] Verified Market Research. DRAM Market Size And Forecast. https://www.verifiedmarketresearch.com/product/global-dram-market-size-and-forecast-to-2025/, 2020.

[101] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st Symposium on Principles and Practice of Parallel Programming*, pages 1–12, Barcelona, Spain, 2016.

[102] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking high bandwidth memory on FPGAs. In *Proceedings of the 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119, Fayetteville, Arizona, USA, 2020.

[103] Wikipedia. List of fastest computers. https://en.wikipedia.org/wiki/List_of_fastest_computers, 2021.

[104] Wikipedia. Roofline model. https://en.wikipedia.org/wiki/Roofline_model, 2021.

[105] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[106] Philipp Woelfel. Efficient strongly universal and optimally universal hashing. In *International Symposium on Mathematical Foundations of Computer Science*, pages 262–272, Szklarska Poreba, Poland, September 1999.

[107] Xilinx, Inc. Baidu deploys Xilinx FPGAs in new public cloud acceleration services. https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html.

[108] Xilinx, Inc. *7 Series FPGAs Memory Interface Solutions (UG586)*, March 2011.

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

[110] Xilinx, Inc. *UltraFast Design Methodology Guide for the Vivado Design Suite (UG949)*, 2019.

[111] Xilinx, Inc. AXI Register Slice v2.1. https://www.xilinx.com/support/documentation/ip_documentation/axi_register_slice/v2_1/pg373-axi-register-slice.pdf, 2020.

[112] Xilinx, Inc. *UltraScale Architecture Memory Resources (UG573)*, 2020.

[113] Xilinx, Inc. Ultrascale architecture-based fpgas memory ip v1.4 (pg150). https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf, 2021.

[114] Xilinx, Inc. *Versal Architecture and Product Data Sheet: Overview (DS950)*, February 2021.

[115] Xilinx, Inc. Vitis Vision Library. https://github.com/Xilinx/Vitis_Libraries/tree/master/vision, 2021.

[116] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual International Symposium on Microarchitecture*, pages 615–628, Columbus, Ohio, USA, 2019.

[117] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[118] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. HitGraph: High-throughput graph processing framework on FPGA. *Transactions on Parallel and Distributed Systems*, 30(10):2249–2264, 2019.

[119] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru 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*, pages 179–188, Monterey, California, USA, February 2017.

MIKHAIL ASIATICI

EXPERIENCE

Research Intern April 2021 – Present
Microsoft Research *Cambridge, United Kingdom*
- Accelerating B-Tree operations on Catapult FPGAs within Project Honeycomb

PhD Student in Computer and Communication Sciences September 2016 – Present
École Polytechnique Fédérale de Lausanne (EPFL) *Lausanne, Switzerland*
- Developed the first generic memory system to increase read bandwidth of throughput-oriented hardware accelerators with irregular memory access patterns

Intern and Scientific Assistant September 2015 – August 2016
École Polytechnique Fédérale de Lausanne (EPFL) *Lausanne, Switzerland*
- Adapted an automated methodology to generate complete high-performance hardware systems from algorithms written in a high-level Domain-Specific Language to Intel and Xilinx Zynq FPGAs
- Developed a methodology and the infrastructures required to integrate FPGAs in a cloud computing ecosystem by virtualizing FPGA resources and supporting dynamic partial reconfiguration

PhD Student in Electrical Engineering (Micro and Nanosystems) November 2014 – August 2015
Royal Institute of Technology (KTH) *Stockholm, Sweden*
- Modelled and characterized a capacitive MEMS accelerometer at temperatures of up to 400 °C
- Developed Through Silicon Vias (TSV) to enable 2.5D and 3D integration in circuits for high temperature (200 °C) applications
- Awarded additional research funding from the Program of Excellence in Electrical Engineering

Cooperation Associate March 2014 – August 2014
European Organization for Nuclear Research (CERN) *Geneva, Switzerland*
- Designed and implemented a readout circuit for SiPM photodetectors, a LabView software interface for acquisition and signal processing and a trigger system for microfluidic scintillation detectors

Research Intern June 2013 – August 2013
University of Glasgow *Glasgow, United Kingdom*
- Designed and simulated an acoustic sensor based on a phononic crystal for surface acoustic waves for liquid sensing and particle counting

Hardware Engineering Intern March 2012 – July 2012
National Institute of Metrological Research (INRiM) *Turin, Italy*
- Designed and implemented an acquisition system for ambient temperature based on Pt100 sensors with higher accuracy than state-of-the-art commercial systems

EDUCATION

MSc in Nanotechnologies for ICT — September 2012 – October 2014
Politecnico di Torino – INPG – EPFL — *Turin, Italy – Grenoble, France – Lausanne, Switzerland*

BSc in Electronic Engineering — September 2009 – July 2012
Politecnico di Torino — *Turin, Italy*

LANGUAGES

- Italian: native
- English: fluent
- French: fluent

PUBLICATIONS

- **M.Asiatici**, P.Ienne, "Request, Coalesce, Serve, and Forget: Miss-Optimized Memory Systems for Bandwidth-Bound Cache-Unfriendly Applications on FPGAs" – Transactions on Reconfigurable Technology and Systems (TRETS) (2021) – to appear
- **M.Asiatici**, P.Ienne, "Large-Scale Graph Processing on FPGAs with Caches for Thousands of Outstanding Misses" – 48th International Symposium on Computer Architecture (ISCA 2021) – to appear
- **M. Asiatici**, D. Maiorano, P. Ienne, "FPGAs in the Datacenters: the Case of Parallel Hybrid Super Scalar String Sample Sort" - 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP 2020)
- G. Csordas, **M.Asiatici**, P.Ienne, "In Search of Lost Bandwidth: Extensive Reordering of DRAM Accesses on FPGA" –2019 International Conference on Field-Programmable Technology (FPT 2019)
- **M.Asiatici**, P.Ienne, "DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses" – 29th International Conference on Field Programmable Logic and Applications (FPL 2019)
- A. Guerrieri, S. Kashani-Akhavan, **M. Asiatici**, P. Ienne, "Snap-On User-Space Manager for Dynamically Reconfigurable System-on-Chips", IEEE Access, vol. 7, p. 103938-103947 (2019)
- **M.Asiatici**, P.Ienne, "Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs" – 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19),
- M. J. Laakso, S. J. Bleiker, J. Liljeholm, G. E. Mårtensson, **M. Asiatici**, A. C. Fischer, G. Stemme, T. Ebefors, F. Niklaus , "Through-glass vias for glass interposers and MEMS packaging applications fabricated using magnetic assembly of microscale metal wires", IEEE Access, vol. 6, p. 44306-44317 (2018)
- M. J. Laakso, S. J. Bleiker, J. Liljeholm, G. E. Mårtensson, **M. Asiatici**, A. C. Fischer, G. Stemme, T. Ebefors, F. Niklaus, "Through-Glass Vias for MEMS Packaging", Micronano System Workshop (MSW), (2018)
- **M. Asiatici**, N. George, K. Vipin, S. A. Fahmy, P. Ienne, "Virtualized Execution Runtime for FPGA Accelerators in the Cloud" – IEEE Access, vol. 5, p. 1900-1910 (2017)
- **M. Asiatici**, M. Laakso, A. C. Fischer, F. Niklaus, "Through Silicon Vias With Invar Metal Conductor for High-Temperature Applications" – J. Microelectromech. Syst., vol. 26, issue 1, p. 158-168 (2016)
- **M. Asiatici**, A. C. Fischer, H. Rödjegård, S. Haasl, G. Stemme, F. Niklaus "Capacitive inertial sensing at high temperatures of up to 400° C" , Sensors and Actuators A: Physical, vol. 238, p. 361-368 (2016)

Ce document a été imprimé au Centre d’impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.