Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs

Mikhail Asiatici and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH−1015 Lausanne, Switzerland

ABSTRACT

FPGAs rely on massive datapath parallelism to accelerate applications even with a low clock frequency. However, applications such as sparse linear algebra and graph analytics have their throughput limited by irregular accesses to external memory for which typical caches provides little benefit because of very frequent misses. Non-blocking caches are widely used on CPUs to reduce the negative impact of misses and thus increase performance of applications with low cache hit rate; however, they rely on associative lookup for handling multiple outstanding misses, which limits their scalability, especially on FPGAs. This results in frequent stalls whenever the application has a very low hit rate. In this paper, we show that by handling thousands of outstanding misses without stalling we can achieve a massive increase of memory-level parallelism, which can significantly speed up irregular memory-bound latency-insensitive applications. By storing miss information in cuckoo hash tables in block RAM instead of associative memory, we show how a non-blocking cache can be modified to support up to three orders of magnitude more misses. The resulting miss-optimized architecture provides new Pareto-optimal and even Pareto-dominant design points in the area-delay space for twelve large sparse matrix-vector multiplication benchmarks, providing up to 25% speedup with 24× area reduction or to 2× speedup with similar area compared to traditional hit-optimized architectures.

ACM Reference Format:
Mikhail Asiatici and Paolo Ienne. 2019. Stop Crying Over Your Cache Miss Rate:, Handling Efficiently Thousands of Outstanding Misses in FPGAs. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293901

1 INTRODUCTION

FPGAs can accelerate compute-intensive applications by implementing massively parallel datapaths. FPGAs also provide spatially distributed SRAM memories with low latency and large aggregate bandwidth, which can store small datasets or implement custom memory hierarchies backing the external DRAM. However, important application classes such as sparse linear algebra and graph analytics are embarrassingly parallel and yet their memory access pattern is such that, with caches that can be realistically implemented on FPGA, most of the accesses are cache misses. In this paper, we will radically rethink the balancing between cache and miss handling logic to optimize the throughput of *read* operations when a large fraction of cache misses is inevitable. We introduce a generic approach, orthogonal to application-specific optimizations, that can provide significant speedup with little design effort. Thanks to its generality, it might be particularly valuable for solutions generated by high-level synthesis tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA '19, February 24–26, 2019, Seaside, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293901

1.1 The Curse of Sparse Narrow Data

Custom memory hierarchy design and automatic generation usually rely on access patterns that have temporal and spatial locality (caches), are regular (scratchpads) or are at least known at compile-time (memory banking and address scrambling) [1, 4, 12, 30]. When access patterns have poor locality and are irregular and data-dependent, one can at best maximize memory-level parallelism (MLP) by emitting enough outstanding memory operations to fully exploit the DRAM latency; however, the throughput of the memory system is still limited to one operation per cycle per DRAM channel, at most. This imposes severe limitations on the amount of datapath parallelism that is worth implementing, limiting the advantage of using an FPGA.

The effective bandwidth gets even more limited if the operands are narrow, such as in sparse linear algebra and graph applications, which often involve irregular accesses to 32- or 64-bit scalars. This relates to the architecture of DRAM memory controllers on FPGA, which expose the memory through wide data interfaces in order to give access to the full DRAM bandwidth despite the slow FPGA clock. For example, exploiting the full 12.8 GB/s DDR3 bandwidth at 200 MHz requires transferring 512 bits per cycle. In this case, an accelerator that operates on 64-bit inputs could at best exploit one eighth of the peak DDR3 bandwidth, and most of the 512 bits returned by the DRAM controller would be discarded. Multiple accelerators whose requests are mutually uncorrelated can only be served by time-multiplexing the memory channel, canceling out any benefits due to parallelization. The only way to improve bandwidth utilization, and thus performance, would be to use larger portions of each block returned from memory.

1.2 Misses Are a Fact of Life

Figure 1 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 CSR-encoded pds-80 matrix from SuiteSparse [8]—and if the same number of read operations were performed

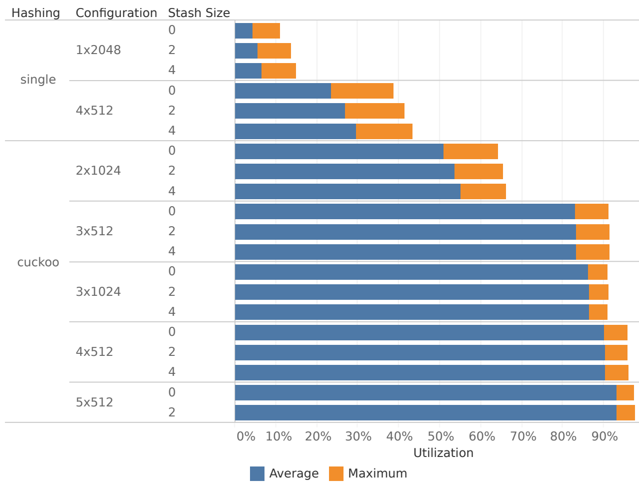

Figure 10: Achievable MSHR storage load factor for several MSHR architectures. The 5×512 system with a 4-entry stash did not meet timing constraints. Single-hash architectures cannot utilize more than 40% of the storage space. Cuckoo hashing can handle collisions more efficiently and three hash tables are enough to achieve more than 80% average and 90% peak load factors, even without stash.

block RAMs for the smallest proposed MHA 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 proposed MHA with no cache, which uses 3× fewer BRAMs than the smallest cache, is enough to saturate the PS DRAM bandwidth only by merging memory requests. On eu-2005 and in-2004, the performance gain provided by the cache-less MHAs 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 an MSHR-rich MHA.

6.3 Number of MSHR Hash Tables and Stash Size

Figure 10 analyzes the performance of the MSHR storage architectures described in Section 2.2. For each architecture, we measure average and peak utilization of the MSHR storage space. To make sure the benchmark always uses all of the available MSHRs, we use a synthetic 1M×1M matrix with 5M uniformly distributed non-zero elements generated with the Python function `scipy.sparse.random(1e6, 1e6, 5e-6)`, no cache, and each bank contains 4096 subentry rows with 3 subentries each. All architectures have 2048 MSHRs per bank or the closest possible value.

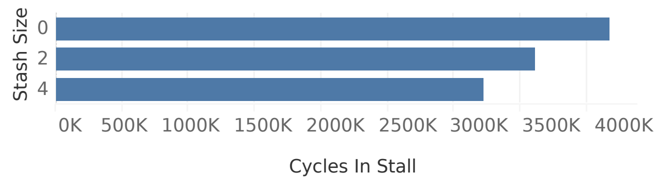

Figure 11: Number of cycles lost due to stalls for collision resolution during the execution of a uniformly distributed benchmark. A 4-entry stash, which occupies less than 0.1% of LUTs and FFs, reduces the number of stall cycles by 30%.

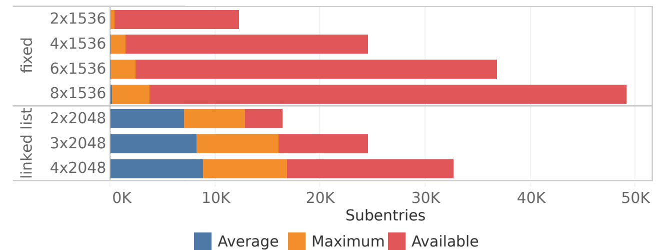

Figure 12: Average and maximum subentry utilization during the execution of ljournal with a 3x512 cuckoo MSHR. Allocating a fixed number of subentries per MSHR results in less than 1% average utilization and resource waste. Linked-list architectures provide a more efficient usage of the subentry memory.

Because any collisions result in a stall that lasts until one of the colliding MSHRs is deallocated, all of the single-hash architectures achieve poor utilization: even by introducing a stash to tolerate up to four collisions, a 4-way set-associative architecture does not go beyond 30% average and 45% peak load factors. Even a simple 2-way cuckoo hash table achieves 50% average and 70% peak utilization, and three ways enough to reach more than 80% average utilization, which is consistent with prior findings on cuckoo hashing [11]. Interestingly, using a 3-way 512-entry architecture (1536 MHSRs) has higher absolute utilization than a 2-way, 1024-entry organization (2048 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 11), which is the largest stash that we could implement within the 200 MHz constraint.

6.4 Subentry Organization

We performed a similar analysis for the memory organization of the subentries, as described in Section 3.2. We use the ljournal benchmark, which has a large number of secondary misses, and an MHA with no cache and a 3×512 cuckoo MSHR buffer per bank. As shown in Figure 12, with a fixed number of subentries per MSHR, stalls are so frequent (Figure 13) 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 14). We believe this problem is more pronounced in an MSHR-rich architecture than in an associative

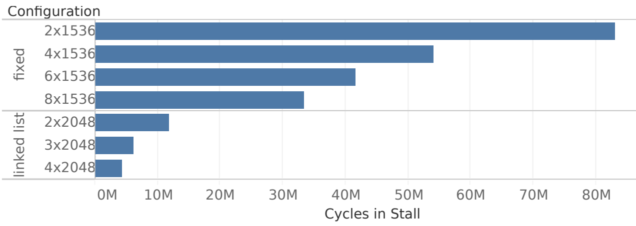

Figure 13: Number of cycles lost due to subentry-related stalls. Stalls occur when (a) filling all subentries of an MSHR for the fixed architectures or (b) handling the linked list or running out of subentry rows for the linked list architectures. The smallest linked list architecture has three times fewer stall cycles than the largest fixed architecture despite having three times fewer subentries.

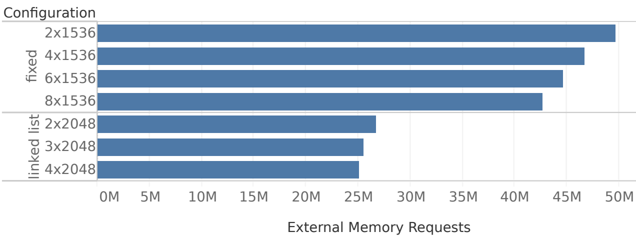

Figure 14: Number of external memory requests during the execution of ljournal with a 3x512 cuckoo MSHR and no cache. By increasing subentry utilization, linked list architectures increase the number of accelerator requests that can be served by the same external memory request, resulting in a 37% decrease of external memory traffic.

MHA 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 DDR memory requests by a factor 1.3× to 2×, with evident great energy impact.

7 RELATED WORK

7.1 Miss Handling Architectures

The first non-blocking cache was proposed by Kroft in 1981 [20]. Farkas and Jouppi [10] evaluate a number of alternative MHA organizations, including the explicitly-addressed MSHRs that inspired our MHA. They observed that non-blocking caches can reduce the miss stall cycles per instruction by a factor 4 to 10 compared to blocking caches, that the most aggressive architectures (such as explicitly-addressed) are beneficial even for large cache sizes, and that overlapping as many misses as possible allows processors to maximize the benefit provided by non-blocking caches.

Tuck et al. [25] 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 [7, 22] could generate even more requests per cycle with no fundamental limitations on the total number of in-flight operations, and b) requests to be merged can come from the same as well as a different accelerator, so it is important to have a shared MHA to maximize the merging opportunities. Our results indeed showed that, for parallel accelerators with massive MLP, small caches with thousands of MSHRs can achieve similar or even better performance of larger caches with few MSHRs.

7.2 Memory Systems for Irregular Memory Accesses

Several pieces of work aimed at improving the efficiency of traditional caches on non-contiguous memory accesses. Impulse [5] introduces an additional address translation stage to remap data that is sparse in the physical/virtual memory space into contiguous locations in a shadow address space. However, it is a processor-centric system which relies on the intervention from the OS to manage the shadow address space. Traversal caches [24] optimize repeated accesses to pointer-based data structures on FPGA. Such approach is however limited to pointer-based data structures that are repeatedly accessed and that can fit entirely in the FPGA block RAM.

Another line of work explored the automatic generation of application-specific memory systems. Bayliss et al. [4] proposed a methodology that automatically generates reuse buffers for affine loop nests, which reduce the amount of memory requests and of DRAM row conflicts. However, the approach is restricted to kernels consisting of an affine loop nest whose bounds are known at compile time. TraceBanking [30] does not rely on static compiler analysis and uses a memory trace to generate a banking scheme that is provably conflict-free. It also supports non-affine loop nests but requires the dataset to fit entirely in the block RAM. ConGen [16] focuses on optimizing DRAM accesses without relying on any local buffering on FPGA. It uses a memory trace to generate a mapping from the addresses generated by the application to DRAM addresses such that the number of row conflicts is minimized. All of these solutions rely on exact information about the application's memory access pattern at hardware compile time. Our approach is application-agnostic, fully dynamic and does not make any assumptions on the access pattern properties.

Coalescing aims at increasing bandwidth utilization between datapath and DRAM by merging multiple narrow memory accesses into fewer, wider ones. Modern GPUs dynamically coalesce accesses from the same instruction executed by different threads [26] in the same warp, and the load-store units instantiated by the Intel FPGA OpenCL compiler can perform both static and dynamic burst coalescing [28]. To increase the opportunities for coalescing and thus the utilization of the bandwidth to the GPU L1 cache, Kloosterman

et al. propose an inter-warp coalescer [19]. Wang et al. [27] proposed a dynamic coalescing unit for HMC memories in a multi-core system, implemented on a small RISC-V core. Incoming requests are stored in a binary tree and forwarded to the HMC after a timeout or after receiving 128 bytes of requests. All of these approaches have a very short window where coalescing can occur, at most a few requests wide. We showed that explicitly-addressed MSHRs also perform coalescing, on wider request windows and over multiple bursts at the same time (one per MSHR).

8 CONCLUSION

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. Non-blocking caches are one of the few architectures for *miss* optimization instead. In this paper, we took the key idea behind non-blocking caches to the extreme: we designed a scheme to handle three orders of magnitude more misses without stalls compared to classic fully-associative MHAs. 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, most of which cannot fit in the FPGA block RAM, 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. Therefore, we believe MSHR-rich MHAs open up new opportunities to increase performance of bandwidth-bound, latency-insensitive applications with irregular memory access patterns. Our MHA, as well as the benchmark SpMV accelerators, can be downloaded as an open-source project from `https://github.com/m-asiatici/MSHR-rich`.

ACKNOWLEDGMENTS

The authors would like to thank Dick Sites, Stephen Neuendorffer, Kristof Denolf, and Kees Vissers for their valuable feedback and suggestions.

REFERENCES

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

[2] Adrian Cosoroaba. 2013. *White Paper 383 - Achieving High Performance DDR3 Data Rates.* Xilinx Inc.

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

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

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

[6] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*. Phoenix, Az., 150–159.

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

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[9] Richard Dorrance, Fengbo Ren, and Dejan Marković. 2014. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs. In *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 161–170.

[10] K. I. Farkas and N. P. Jouppi. 1994. Complexity/Performance Tradeoffs with Non-blocking Loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. Chicago, Ill., 211–222.

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

[12] Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware System Synthesis from Domain-Specific Languages. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*. Munich, 1–8.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*. Seoul, 243–254.

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

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

[16] Matthias Jung, Deepak M. Mathew, Christian Weis, Norbert Wehn, Irene Heinrich, Marco V. Natale, and Sven O. Krumke. 2016. Congen: An application specific dram memory controller generator. In *Proceedings of the 2nd International Symposium on Memory Systems*. Alexandria, Va., 257–267.

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

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

[19] John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. 2015. WarpPool: sharing requests with inter-warp coalescing for throughput processors. In *Proceedings of the 48th Annual International Symposium on Microarchitecture*. 433–444.

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

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

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

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

[24] Greg Stitt, Gaurav Chaudhari, and James Coole. 2008. Traversal caches: A first step towards FPGA acceleration of pointer-based data structures. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis*. Atlanta, Ga., 61–66.

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

[26] Vasily Volkov. 2016. *Understanding latency hiding on GPUs.* Ph.D. Dissertation. UC Berkeley.

[27] Xi Wang, John D. Leidel, and Yong Chen. 2016. Concurrent dynamic memory coalescing on GoblinCore-64 architecture. In *Proceedings of the Second International Symposium on Memory Systems*. 177–187.

[28] Felix Winterstein and George Constantinides. 2017. Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs. In *Proceedings of the 2017 International Conference on Field Programmable Technology*. Melbourne, 104–111.

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

[30] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. A New Approach to Automatic Memory Banking Using Trace-Based Address Mining. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 179–188.