A Service-Oriented Memory Architecture for FPGA Computing

Joseph Melber
Carnegie Mellon University
Pittsburgh, Pennsylvania
jmelber@cmu.edu

James C. Hoe
Carnegie Mellon University
Pittsburgh, Pennsylvania
jhoe@cmu.edu

***Abstract*—Memory access is an essential aspect of FPGA compute accelerator design. Current development environments pay much more attention to high-level compute abstraction while holding on to the familiar basic load-store memory paradigm. This paper proposes a service-oriented memory architecture where, instead of operating in terms of loads, stores and addresses, a compute accelerator design interacts with abstracted memory services that present high-level, semantic-rich operations—both compute and data transfers—on encapsulated data objects. The support for a memory service, realized as a soft-logic module or a composition of modules, is developed by domain experts and available to the accelerator design in a reusable catalog collection. This paper sets forth a service-oriented memory architecture and provides a development framework to specify and generate a customized service-oriented memory system. We evaluate the proposed abstraction and design framework through a case study of a breadth-first search accelerator. We demonstrate that a service-oriented memory paradigm increases development convenience while simplfiying an accelerator design without negatively impacting performance or resource utilization.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) with abundant concurrent processing elements promise a phenomenal level of processing throughput. However, to deliver actual compute performance, FPGA compute accelerator designers must manage to move data to/from the processing elements at a data rate commensurate with the processing throughput. Current FPGA programming environments give disproportional emphasis to lowering the design effort on processing-centric kernels than to the memory access side of the design task. Common design methodologies today support a designer with middleware interfaces (e.g., AXI and Avalon) for read and write operations to insulate the designer from interfacing directly with DRAM at the wire and cycle level [1], [2]. However, the designer must still build all the datapaths for on-chip buffering and data movements, as well as the state machines to coordinate these datapath activities.

The cloud computing community developed service-oriented architecture to enable flexibility and extensibility within a distributed computing environment [3]. Additionally, service-oriented architecture provides policies and practices to compose software services into a larger application. This reduces integration complexity and development time. Recently, entire FPGA accelerators have been deployed as a service [4]–[6]. This work proposes a service-level view of FPGA memory in which the designer does not think in terms of load/store/address, rather their designs interact with abstracted memory services that encapsulate high-level data objects and memory operations behind equally high-level messaging interfaces.

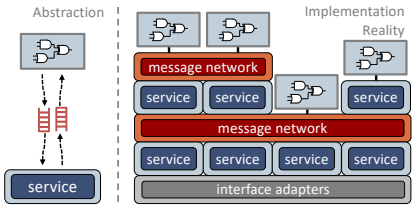

Fig. 1. The service-oriented memory architecture in which processing kernels interact with abstracted memory services that present high-level, semantic-rich operations on encapsulated data objects. The abstraction hides the implementation complexity of services and the required infrastructure.

The service-oriented memory architecture introduces a high-level abstraction for using memory (Fig. 1) and provides designers with starting points higher than standard bus interfaces and DMA IPs, or even explicit notions of memory as an array of locations. Memory services are implemented as soft-logic service modules that abstract platform interfaces and data layout, and encapsulate high-level semantic-rich operations on data in memory. Services themselves are portable functionalities that are agnostic of the platform and memory device where the data resides. Individual services are implemented as separate components within a system—each service can be shared and share resources such as memory devices. This allows the system to enable multiple functions under the abstraction implemented spatially. Furthermore, complex memory services may be a composition of simpler services.

A system generation framework hides implementation complexity by providing facilities to specify memory services and construct a memory system through catalog and registry files respectively. This enables FPGA accelerator designers to build a customized memory system that presents high-level operations through a collection of expertly crafted memory service modules.

In the following sections, we motivate the service abstraction for FPGA memory. We present our framework to support and enable service-oriented design for FPGA memory systems. Finally, we evaluate this abstraction and framework in a case study of a breath-first search graph accelerator.

We demonstrate that the service-oriented memory architecture increases convenience while reducing development complexity without resource or performance overheads.

II. Background

Current standard RTL-level design methodologies provide middleware layers with bus-like read/write interfaces [1], [2]. High-level synthesis design methodologies support a C-to-hardware abstraction primarily in the design of the processing kernels [7], [8]. RCMW and GAScore provide a standard set of memory interfaces tailored to streaming and block copy access patterns [9], [10]. LEAP provides load-store interfaces to scratchpad memories and extends their capacity virtually with the backing of off-chip DRAM [11], [12]. CoRAM restricts a processing kernel's data access to on-chip SRAM scratchpads and provides a software-like control-thread API to manage data transfer to/from off-chip DRAM [13]. In all of these, the FPGA accelerator designer cannot avoid thinking explicitly about how a data structure is laid out in a linearly addressed space and how a complex operation expands into discrete loads and stores. CoRAM++ extended the CoRAM API to include data-structure specific data transfers [14]. CoRAM++ data structure specific support is an early precursor to our current thinking to develop a complete memory as a service abstraction. This work adopts a service-oriented memory abstraction providing useful operations on data as well as transferring data to/from memory.

III. Memory Service Abstraction

A. Motivating Example: Shared Counters

Consider an example accelerator with many processing kernels distributed on an FPGA fabric. The kernels need to increment the entries of a shared table of counters in DRAM. Following today's common approach (Fig. 2.top), each processing kernel needs to understand the table explicitly as an array in DRAM and manipulate it through load and store primitives via a middleware layer like an AXI or Avalon bus [1], [2]. In addition to following a correct locking discipline, for each increment operation, a processing kernel needs to load the table entry from DRAM and then store the updated value back to DRAM. While straightforward, this is not the best way to think about hardware design given FPGAs' programmability and hardware concurrency.

Service-oriented memory architecture insulates designers from implementation complexity while maintaining FPGAs' inherent ability to specialize. Consider an alternative scenario (Fig. 2.bottom) where a centralized service module serves to provide the abstraction and implementation of the counter table. The processing-focused kernels only need to send simple, fire-and-forget requests to the proxy module to effect the table increments. Only the service module needs to understand the table layout and DRAM specifics (e.g., how to update a single word when the granularity of DRAM accesses is larger). As a centralized service, atomicity of the increment is trivially observed without the need for locking or cross-kernel communication. Performance is improved because the read-increment-write sequence is contained within the service module, and can be placed near to the physical DRAM interface. The service could even introduce caching and merging optimizations without affecting the design and operation of the processing kernels.

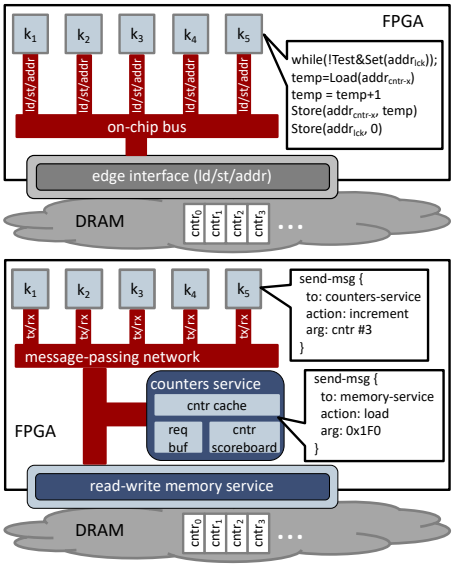

Fig. 2. Top: Processing kernels incrementing a shared table of counters in DRAM. Each kernel must be aware of the interface and locking semantics, as well as the table data structure details. Bottom: A service module acts on the kernel's behalf to manipulate counters in memory in response to a message request.

B. A Case for Memory as a Service

Specialized memory operations available as convenient services reduce the development burden on compute accelerator designers. Good designers do practice modular design, but as ad hoc efforts that are not generally reusable. Imposing a service framework and interface can help force those efforts into a reusable and portable form. The memory as a service model balances abstraction and customization proficiently to encapsulate in-memory data objects behind high-level, intelligent access and compute operations. The lowest level services enable the portability of processing kernels and other higher-level services by virtualizing the differences across diverse platforms and memory technologies. Extending convenience and portability even further, services themselves can be composed of simpler services hierarchically in the service-oriented memory architecture.

Decoupling memory operations as services simplifies the logic of processing-focused kernels. Services range from simple memory engines supporting atomic read-modify-writes, to traversing and modifying large pointer-based data structures. Furthermore, abstraction overhead in hardware is low—without services, the kernel modules would have to implement these operations internally if not provided as a service. Memory service abstraction can actually improve design results if high quality, expert-designed service modules are available. Below this abstraction, designers can even selectively instantiate performance improvements through parameters.

C. Curating Services

Service-oriented memory architecture enables services and entire memory systems to be developed as loosely connected components thus simplifying development without relinquishing flexibility. A service comprises a logical operation and its physical implementation. In our work, services are implemented as service modules developed as soft-logic in Verilog or Bluespec [15]. Domain experts craft efficient memory operations inside service modules abstracting data layout, interface semantics, data modification, et cetera. Each service must have a corresponding catalog entry in JSON that describes its functionality, interface channels, operations ascribed to each channel, parameters, and message structure. The catalog entry is a binding contract between users and the service developer. The listing below sketches the entry for the "counters" service:

```
{ "service-type": "counters",
  "channels": [{ "ifc-name": "count",
                 "ifc-name": "table_mem" }],
  "provides": [{ "service-type": "counters",
                 "ifc-name": "count" }],
  "requires": [{ "service-type": "read-write",
                 "ifc-name": "table_mem" }],
  "message-structure": { ... } }
```

Service composition extends service module designers the same ease in development that user-level designers are afforded through service-oriented architecture. Not only can services make use of abstract portable operations through other services, this freedom to build complex service hierarchies enables rich customization without redevelopment cost or knowledge of the implementation complexity. Returning to our previous example, the counters service requires a read-write service. The read-write service provides virtualized access to the the table of counters in memory. The counters service does not need to be aware of the location of the table in memory nor the hardware details of the physical memory device.

D. Communicating with Services

The design of a standardized application programming interface for memory services affects many of our objectives. Abstracting communication decouples services and clients simplifying development while maintaining implementation flexibility for optimization. We support interactions between client and service modules through request-response message passing over logically point-to-point latency-insensitive [16] channels. A message comprises of a defined number of argument words and optionally data. The service module designer defines the semantics of a message to their service through its catalog entry.

When a processing kernel of an FPGA compute accelerator wants to operate on a service-abstracted data object, it invokes the memory service as a client through its associated channel interface. Using SystemVerilog's *interface* construct [17], a channel interface is defined as a pair of message queues—for request and response, respectively—running in opposite directions between the client module and the service module. The point-to-point channel connection is a logical construction; our framework later elaborates each logical connection by name. During generation, depending on the mapping strategy, a connection may be rewritten and multiplexed on to a shared physical path transparent to the client and the service modules' perspectives.

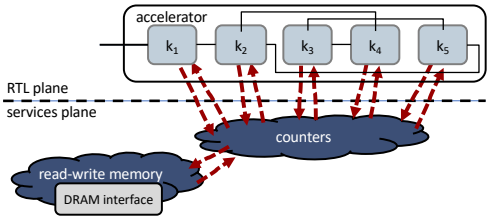

Fig. 3. A service-level sketch of the design from Fig. 2.bottom showing the normal accelerator design module hierarchy and the out-of-band service channel connections. The registry file is used to express these logical connections.

IV. System Development Framework

A. Instantiating Services in a Design

A memory service module is not instantiated in the traditional sense. Designers express abstract connections to services, through channel interfaces, in their kernels but not the service's physical implementation. This development model enables convenient and flexible system design. In the service-oriented memory architecture, service modules live "extracorporeally"—outside of the normal compute accelerator's module hierarchy. A client module indicates the desire to access a service module by declaring a channel interface in its port list (e.g., "cntrs" below). This interface is left dangling only to be completed later during post-processing rewriting.

```
module kernel(input clk, channel cntrs);
  ...
  always_ff @(posedge clk) begin
    cntrs.txPush <= do_inc && !cntrs.txFull;
    cntrs.txMsg.arg0 <= cntr_id;
    cntrs.txMsg.arg1 <= how_much;
  end
endmodule
```

Client and service module instantiations and connections are expressed by the designer using a JSON registry file separate from the Verilog design files. A registry file for a given service-oriented memory system specifies the logical organization of connections between modules. Fig. 3 shows a logical sketch of the counters accelerator design from Fig. 2.bottom. These connections are expressed by the designer in the registry file. The listing below sketches one kernel and the counters service captured in the registry file for the counters accelerator:

```
{ "platform": { ... },
  "connected-components": [ ...
  { "type": "kernel", "name": "k5",
    "connections":
    [ { "channel-ifc-name": "cntrs",
        "connection": "counters_svc" }]},
  { "type": "counters", "name": "counters_svc",
    "connections":
    [ { "channel-ifc-name": "count",
        "connection": "counters_svc" },
      { "channel-ifc-name": "table_mem",
        "connection": "localMem" }]}, ... ]}
```

The registry file specifies the connection named "counters_svc" to link the channel interfaces between the counters and kernel modules. The post processing generator: 1) parses the registry file to construct a memory system module with appropriate external interfaces, 2) instantiates modules, and 3) implements logical connections as physical messaging channels. Designers also have the option to specify control and status registers, through the registry file, to be implemented.

B. Quality of Service

Quality of service (QoS) is an important principle in service-oriented architecture. Our framework allows designers to specify QoS constraints through the registry file to flexibly optimize services without modifying their kernel RTL code. For example, the number of requests a service can accept, or the client request arbitration scheme for sharing a service can be specified as parameters. Static priority and round-robin arbitration are supported in the physical channel transport infrastructure. These provide a QoS knob for bandwidth allocation to clients. Specifying the physical implementation of logical channels through the registry file is another QoS optimization. The current framework supports logical channels implemented as point-to-point physical channels, as well channels via a shared network-on-chip [18] to increase scalability.

C. Instrumentation and Introspection

As the number of services grow in a complex service-oriented memory system, it becomes difficult to debug and tune services. To ease this burden, our framework supports instrumentation and introspection features through a statistics interface [19]. We provide a set of simple statistics IP cores to probe and instrument service logic. The current set of statistics cores include counters, samplers, and a protocol checker. These statistics allow designers to gain high-level insights about system behavior and fine-tune their memory systems. Statistics cores are fully synthesizable and therefore can be used in simulation, prototyping, and the final design.

V. Case Study Implementation

We selected a graph algorithm, breadth-first search (BFS), as the application case study for the service-oriented memory architecture. We chose this algorithm as it is memory-intensive, data-centric, and irregular; this provides an opportunity to examine memory services in a stressful and dynamic application. Our baseline accelerator is designed as an elastic pipeline with kernel stages decoupled by memory accesses—this is similar to the baseline FPGA-only accelerator design by Wang et al. [20]. The design aims to exploit available memory-level parallelism, and maximize on-chip data reuse. Our baseline accelerator accesses the input graph data at the platform DRAM interface granularity stored in the compressed sparse row format as shown in Fig. 4.

Our service-oriented accelerator design absorbs complexity from kernel stages and simplifies their code to look much like the algorithm pseudocode. The fully abstracted BFS pipeline is shown in Fig. 5 with worklist and graph services. The

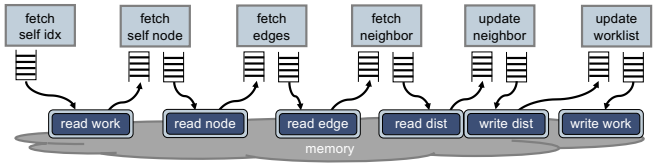

Fig. 4. Baseline BFS elastic pipeline accelerator with read/write services. Each service enables access to a part of the graph data structure in memory.

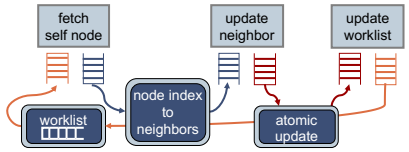

Fig. 5. Final BFS accelerator including graph services—that abstract the graph traversal and atomic neighbor distance update—and a worklist service.

pipeline now has a reduced number of kernel logic stages as the higher-level graph services absorb complexity, and the stages themselves are much cleaner. The "node index to neighbors" service streamlines the algorithm kernel by providing a compound operation that, with one request, returns not just one neighbor but a stream of neighbors and their distances (so the kernel logic does not need to iteratively look-up and query each neighbor). This encapsulation does not add extra logic cost or logic delay.

Service-oriented memory architecture is also effective in addressing hardware-induced difficulties. For example, DRAM interfaces on FPGAs operate on a multi-word granularity as large as 64 bytes. The baseline accelerator needs to account for this inconvenient detail at every turn. When updating a neighbor node's new distance, the baseline accelerator must read an entire 64-byte block, modify the affected sub-word, and write back the entire block. Worse yet, the kernel module must check if back-to-back updates are to the same block to ensure the updates are correctly merged. This kind of complexity arising from the operational and structural specifics of the interface is dealt within the "atomic update" graph service abstraction. Similarly, our service-oriented accelerator abstracts the worklist as a service, and the kernel modules interact with this service at the work-item granularity. As far as the designer is concerned, the worklist is an "unlimited" capacity circular buffer. These services support memory operations at the sub-word granularity regardless of the underlying memory interface, resulting in both lower design effort and increased portability. Furthermore, services can be specified with performance enhancements like a cache. We demonstrate this in the "atomic update" graph service, taking advantage of temporal locality in the neighbor distance memory operations.

VI. Evaluation

A. Evaluation Setup

For this study, we deploy our prototype framework on the Intel FPGA Programmable Acceleration Card (PAC) D5005 [21] and the Intel Open Programmable Acceleration Engine (OPAE) SDK [22]. We evaluate our framework and architecture through a case study of a BFS graph accelerator. We use a set of synthetic and standard network graphs [23] as inputs which are summarized in Tab. I for this evaluation.

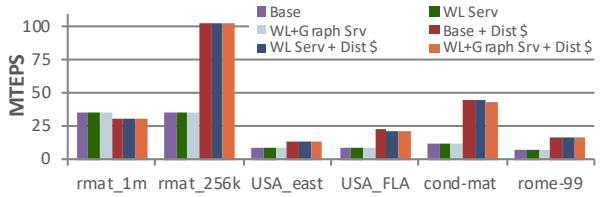

Fig. 6. Performance achieved in millions of traversed edges per second (MTEPS) for the BFS accelerator for the base implementation, worklist service, and graph services implementations showing that the service abstraction does not negatively impact performance.

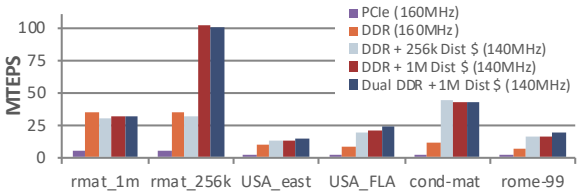

Fig. 7. Performance across memory devices available in the PAC system and node distance caching for the BFS accelerator shown in MTEPS. These results highlight the flexbility of services across diverse memory interfaces and cache configurations.

TABLE I
BFS BENCHMARK GRAPHS AND SIZES

Graph	Nodes	Edges
rome99	3.3k	8.8k
cond-mat	40k	351k
USA_FLA	1.1M	2.7M
USA_east	3.6M	8.7M
rmat_256k	256k	4.2M
rmat_1m	1M	16M

TABLE III
WORK-ITEM BUNDLES SPILLED TO MEMORY

Benchmark	Ave	Max
rome99	0.02	1
cond-mat	699.12	1036
USA_FLA	27.87	113
USA_east	20.38	208
rmat_256k	85.19	11395
rmat_1m	50.63	44891

B. Service Abstraction Overhead

Our first scenario evaluates the overheads introduced by abstracting portions of the BFS pipeline as services. Comparable performance is achieved while greatly reducing the complexity of the accelerator pipeline and kernel code—therefore reducing the overall effort to develop a BFS accelerator. Across the set of benchmark graphs we measured the computation throughput in traversed-edges-per-second (TEPS). Abstracting the worklist and graph operations as services does not reduce performance shown in Fig. 6 nor increase resource utilization as shown in Tab. II. This is unsurprising as the logic required to provide these functionalities has to be included in the baseline user-level kernel logic without services.

C. Memory Device Portability

The second set of experiments evaluates the flexibility of service-oriented memory architecture. Changing the memory device behind services is a trivial adjustment with the support of our framework, allowing designers to tune for performance through modifying the registry file without modifying user-level code. We evaluated the fully abstracted BFS accelerator design across five memory device configurations as shown in Fig. 7. As anticipated, performance improves with each optimization conveniently enabled by making small changes to a registry file rather than RTL redevelopment.

D. Service Instrumentation and Introspection

Throughout our accelerator development, specifically while developing and optimizing the worklist functionality, we noticed a need for runtime introspection [19]. This requirement arose as the worklist's memory demands are runtime and benchmark-dependent. We developed instrumentation features as a core function of the service-oriented memory architecture and monitored the memory requirements of the worklist service. We tracked the average and maximum number of work-item bundles spilled to memory shown in Tab. III. These statistics led us to prefetch ahead of the accelerator to keep up with work item demands, rather than cache or attempt to rely on on-chip SRAM. This demonstrated introspection's value assisting in the development of services with runtime effects on performance, and tuning for efficiency.

TABLE II
RESOURCE UTILIZATION FOR SERVICE ABSTRACTIONS

Configuration	ALM	Registers	BRAM
Base	141495	163643	10590080
Worklist Service	141598	164131	10590080
Worklist + Graph Services	138817	164895	10590080

VII. CONCLUSION

FPGA's inherently reconfigurable fabric provides immense freedom to hardware accelerator designers to generate highly optimized and efficient compute architectures. Furthermore, modern FPGAs continue to grow in size and attach to rich and varied memory devices. This same development freedom complicates memory system design as interconnects and hierarchies must be developed from raw memory interfaces to support highly optimized compute modules. We have argued that the familiar general-purpose load-store memory access paradigm is not an efficient abstraction to ease FPGA development. Rather, the advantages of FPGAs as a reconfigurable fabric should be extended into a memory architecture design. We demonstrated a service-oriented memory architecture for FPGA computing, where services provide customized access to data and absorb complexity from processing kernel modules. Through well defined interfaces and flexible composition, services simplify application development without reducing performance or adding resource overheads, and present a successful abstraction for FPGA memory.

ACKNOWLEDGMENTS

This work was supported by the Intel Strategic Research Alliance (FPGA Programming Optimization). The authors thank Intel and Bluespec for tools, hardware, and support.

References

[1] Xilinx, *AXI Reference Guide*, https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[2] Intel Corporation, *Avalon Interface Specifications*, https://www.altera.com/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[3] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy, survey, and issues of cloud computing ecosystems," in *Cloud Computing: Principles, Systems and Applications*. London: Springer London, 2010, pp. 21–46, ISBN: 978-1-84996-241-4. DOI: 10.1007/978-1-84996-241-4_2. [Online]. Available: https://doi.org/10.1007/978-1-84996-241-4_2.

[4] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783710.

[5] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16, Santa Clara, CA, USA: Association for Computing Machinery, 2016, pp. 456–469, ISBN: 9781450345255. DOI: 10.1145/2987550.2987569. [Online]. Available: https://doi.org/10.1145/2987550.2987569.

[6] C. Wang, X. Li, Y. Chen, Y. Zhang, O. Diessel, and X. Zhou, "Service-oriented architecture on FPGA-based MPSoC," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2993–3006, Oct. 2017, ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2701828.

[7] Xilinx, *Vivado HLx*, https://www.xilinx.com/products/design-tools/vivado.html.

[8] Intel Corporation, *Intel HLS Compiler*, https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html.

[9] R. Kirchgessner, A. D. George, and H. Lam, "Reconfigurable computing middleware for application portability and productivity," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, Jun. 2013, pp. 211–218. DOI: 10.1109/ASAP.2013.6567577.

[10] R. Willenberg and P. Chow, "A Remote Memory Access Infrastructure for Global Address Space Programming Models in FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13, Monterey, California, USA: ACM, 2013, pp. 211–220, ISBN: 978-1-4503-1887-7. DOI: 10.1145/2435264.2435301. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435301.

[11] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. 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*, 2011, pp. 25–28.

[12] K. Fleming, H. J. Yang, M. Adler, and J. Emer, "The LEAP FPGA operating system," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927488.

[13] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2011, pp. 97–106.

[14] G. Weisz and J. C. Hoe, "CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[15] Bluespec, Inc., *Bluespec System Verilog*, http://www.bluespec.com/products/bsc.htm.

[16] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001, ISSN: 1937-4151. DOI: 10.1109/43.945302.

[17] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, Feb. 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[18] M. K. Papamichael and J. C. Hoe, "CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ACM, 2012, pp. 37–46.

[19] M. K. Papamichael, "Pandora: Facilitating ip development for hardware specialization," PhD thesis, Carnegie Mellon University, 2015.

[20] Y. Wang, J. C. Hoe, and E. Nurvitadhi, "Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2019, pp. 136–144. DOI: 10.1109/FCCM.2019.00028.

[21] Intel Corporation, *Intel FPGA Programmable Acceleration Card D5005 Data Sheet*, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-d5005.pdf, DS-1058, Nov. 2019.

[22] ——, *Open Programmable Acceleration Engine - Documentation*, https://opae.github.io/latest/index.html, version 1.4.0.

[23] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663.