FReaC Cache: Folded-logic Reconfigurable Computing in the Last Level Cache

Ashutosh Dhar*, Xiaohao Wang*, Hubertus Franke†, Jinjun Xiong†,
Jian Huang*, Wen-mei Hwu*, Nam Sung Kim*, Deming Chen*
*University of Illinois, Urbana-Champaign, †IBM Research,
{adhar2, xwang165, jianh, w-hwu, nskim, dchen}@illinois.edu {frankeh, jinjun}@us.ibm.com

***Abstract*—The need for higher energy efficiency has resulted in the proliferation of accelerators across platforms, with custom and reconfigurable accelerators adopted in both edge devices and cloud servers. However, existing solutions fall short in providing accelerators with low-latency, high-bandwidth access to the working set and suffer from the high latency and energy cost of data transfers. Such costs can severely limit the smallest granularity of the tasks that can be accelerated and thus the applicability of the accelerators. In this work, we present FReaC Cache, a novel architecture that natively supports reconfigurable computing in the last level cache (LLC), thereby giving energy-efficient accelerators low-latency, high-bandwidth access to the working set. By leveraging the cache's existing dense memory arrays, buses, and *logic folding*, we construct a reconfigurable fabric in the LLC with minimal changes to the system, processor, cache, and memory architecture. FReaC Cache is a low-latency, low-cost, and low-power alternative to off-die/off-chip accelerators, and a flexible, and low-cost alternative to fixed function accelerators. We demonstrate an average speedup of** 3**X and Perf/W improvements of** 6.1**X over an edge-class multi-core CPU, and add** 3.5% **to** 15.3% **area overhead per cache slice.**

***Index Terms*—Reconfigurable Computing, Near Memory Acceleration, In Cache Computing, Logic Folding**

I. Introduction

The end of Dennard scaling [1]–[4] has prompted a paradigm shift in processor design, with architects increasingly looking to specialized accelerators for improved performance and energy efficiency [5]–[13]. The use of off-die, and off-chip accelerators, in both data centers and edge devices [10], [14], [15], continues to grow, as exemplified by the increasing adoption of GPUs, FPGAs, and ASICs. However, where to place these accelerators and how to deliver data to them remain as research questions. For example, PCIe-attached accelerators access data in the system memory at limited bandwidth, e.g. 16GB/s in PCIe 3.0 x16, and PCIe system drivers incur tens of thousands of instructions per accelerator transaction [16], translating into longer latency and lost bandwidth. As a result, each DMA transfer can cost between 1µs and 160µs [17]. Furthermore, PCIe-attached cards draw additional power, with a recent study noting that a PCIe-attached FPGA drew 12 W when idle [18]. In edge computing scenarios, working set sizes can be small enough that the time and energy spent shuttling data back and forth render off-chip and off-die accelerators undesirable for many applications.

The widening gap between on-chip and off-chip memories is another important factor. For example, fetching data from off-chip DRAM takes 56ns [19] and consumes 28 to 45 pJ/bit (40nm). In contrast, reading 16bits from an on-chip 32K-word SRAM array costs 11pJ [20]. The widening gap has prompted the exploration of placing accelerators closer to memory [21]–[26]. However, these near memory accelerators (NMA) have traditionally faced many challenges, including: DRAM technology process limits, program transparency, flexibility and sheer design complexity.

When we consider the fast evolving nature of workloads, and the rising design, engineering and verification costs that often make custom ASIC/SoC with on-chip accelerators undesirable, we see that there is a need for a fast, energy-efficient, and programmable reconfigurable computing architecture.

To address these challenges, we seek to provide a middle ground between energy efficiency, cost, and performance, in a manner that introduces limited changes to existing system, processor, and memory architectures. We present FReaC Cache (Folded-logic Reconfigurable Computing in Cache), a reconfigurable computing architecture that leverages existing structures in the last-level cache to build accelerators, thereby bringing flexible, configurable, cheap, and energy-efficient computation closer to memory. FReaC Cache partitions the last level cache (LLC) and uses a portion of the LLC's dense memory arrays to build reconfigurable logic in the form of look-up tables (LUTs). Along with limited logic structures, FReaC Cache uses the LUT arrays with *logic folding* [27] to implement several *micro compute clusters*, each of which is a very small, but dense, computational unit. In doing so, FReaC Cache is able to provide on-demand reconfigurable accelerators, with access to the large bandwidth available at the LLC, yet without the need to move data and configuration off-chip. Thus FReaC Cache is able to provide better *end-to-end* performance and power, at much lower cost, when compared to off-chip accelerators. FReaC Cache partitions are flexible, allowing the user to choose how much of LLC to use for computation, with the rest remaining as a cache.

In this work, we focus on introducing the ideas and principles of FReaC Cache by incorporating it into the LLC of existing processor architectures, without significant modifications to other parts of the processor, and propose its use for edge devices that require low-cost and low-power acceleration for fine-grained tasks across a diverse group of applications. Our goal is to offload small but important kernels that would benefit from customized accelerator logic, and the high throughput

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00021

and high bandwidth of FReaC Cache. FReaC Cache can be incorporated into existing or new processor architectures and is not limited to any particular core micro-architecture, instruction set, or application domain.

In contrast to processing in memory (PIM) and NMA, FReaC Cache provides a less invasive solution, while still having ample bandwidth available at the LLC. Where PIM architectures attempt to perform in-situ processing of data in memory [22]–[26], [28], [29] by either adding additional computational units or manipulating data within the memory arrays, FReaC Cache uses SRAM structures in the LLC to build computational units, without modifying the memory arrays. Thus, FReaC Cache preserves the density and timing of the existing memory arrays, and also limits design costs. In contrast to NMA systems that seek to place computation adjacent to DRAM banks, chips or even in the DRAM controller, FReaC Cache sits farther away from main memory. However, most NMA and PIM approaches require some change to the host processor in order to communicate with the added accelerator via custom instructions in the ISA [21], [23], [24], [30]. ISA changes are not trivial, and changes to the core frontend are often accompanied by a large amount of design verification effort. In FReaC Cache we avoid the addition of any custom instructions or ISA modifications. Rather, we only use loads and stores to reserved address ranges to communicate with the in-cache accelerator. This style of communication is possible due to the core's proximity and relatively low latency access to the LLC.

The need for flexible, low-cost, low-power, and high-performance accelerators is further exacerbated by growing demands for processing at the edge,[1] and the evolving nature of workloads makes programmability of prime importance. While many accelerator solutions are *programmable* [21]–[23], [31], most of them are still *domain specific* [8], [24], [30], [32]. Such accelerators cannot be modified easily to adapt to the fast-evolving algorithms. Take, for example, computer vision. The winning team of ImageNet 2010 [33] used SIFT (Scale-invariant feature transform) and LBP (local binary patterns), whereas the introduction of AlexNet [34] in 2012 prompted a dramatic shift in the domain towards deep neural networks. However, within the sub-domain of deep neural networks, we observe a constant evolution in the algorithms and their requirements. Thus, we submit that in this environment of changing algorithms, a reconfigurable accelerator fabric is ideal, and FReaC Cache strives to provide this flexible fabric.

We position FReaC Cache as a low-cost, low-latency, and low-power alternative to off-die/off-chip accelerators, and a flexible and low-cost alternative to on-chip fixed function accelerators. We use cheap structures to provide acceleration on-demand, and focus on cost and energy sensitive scenarios such as edge-computing.

[1]While FReac Cache is highly energy-efficient, we do not consider ultra-low power devices, such as wearables, in this work. Rather, we focus on processors such as the Samsung Exynos 9810, Apple A12X, Nvidia Tegra Xavier, and Intel desktop/mobile SoCs, with 4 to 8 cores, operating at clock frequencies of 2 to 4GHz, and running machine learning, data processing, and security apps at the edge.

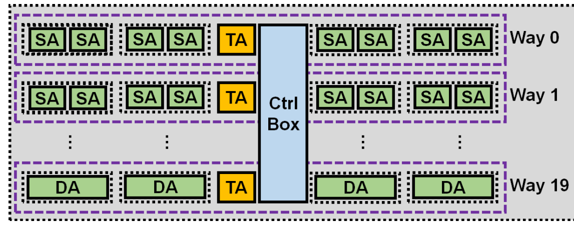

Fig. 1. Cache slice architecture. Data arrays (DA) shown in green, and Tag/State arrays shown in yellow.

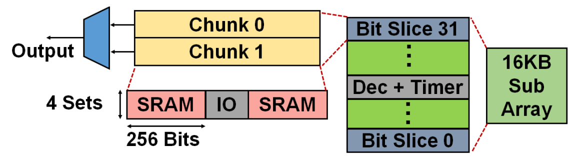

Fig. 2. Sub-array (SA) organization

Our key contributions are:

- We introduce a cache-based reconfigurable computing architecture, FReaC Cache.
- We describe the circuit and architectural components needed to build look-up tables (LUTs) using the existing memory arrays in the LLC.
- Our architecture provides very high logic density, when compared to modern FPGAs, and is capable of high-speed dynamic reconfiguration.
- We demonstrate that FReaC Cache achieves average speedups of 3X, and average Perf/W improvements of 6.1X, when compared to a modern CPUs, and adds only 3.5% to 15.3% of area overhead to the LLC slice.

Our paper begins with a background discussion in Sec. II. We then present FReaC Cache in Sec. III and IV, after which we present our evaluation in Sec. V. Sec. VI provides additional insight, before we discuss related work in Sec. VII, and conclude in Sec. VIII.

II. Background

Last Level Cache Design: Modern last level caches are designed for very large capacities. Hence, most modern chip multi-processors use a distributed LLC NUCA (Non-Uniform Cache Architecture) design [35], with the cache being split into many *slices*, one per-core or per-core-cluster. These slices are organized around a central interconnect that provides high bandwidth between the cores and all the slices [36]–[38]. Memory addresses are interleaved across slices, and cores may access any slice. However, cores may experience non-uniform latency depending on the slice's distance, due to the use of interconnects, such as ring busses [36]. Over the last few generations, LLC architecture has not changed much [36], [38], [39], maintaining its distributed slice architecture, from older bulk designs [40]. We demonstrate our design using the sliced LLC architecture described by Huang et al. [36].

Fig. 1 illustrates the organization of a 20-way associative 2.5MB cache slice [36]. The slice is comprised of multiple data arrays, organized in a tiled fashion in four quadrants. Each way

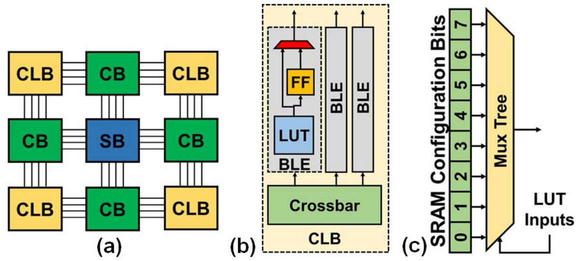

Fig. 3. (a) FPGA architecture. (b) CLB design. (c) SRAM based 3-LUT.

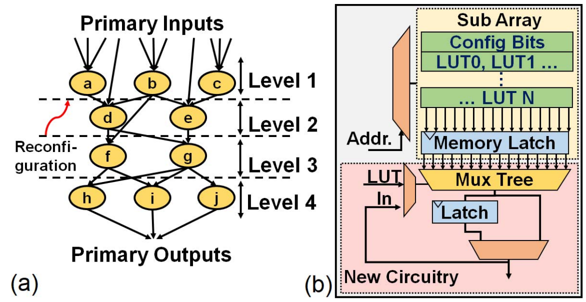

Fig. 4. (a) Logic folding. (b) Compute sub-arrays.

is comprised of a single data array (DA) from each quadrant; i.e. each way is comprised of four data arrays (DA), along with a Tag/State and CV(valid)/LRU array. A control box unit sits in the middle of the cache slice, and is responsible for all control operations, coherence and interconnect interfacing. Each 32KB data array is comprised of two 16KB sub-arrays, each with a 32bit port. Fig. 2 presents the micro-architecture of the sub-array (SA). For simplicity, we do not show redundant rows and columns. The sub-array is comprised of 32 *bit-slices*, each of which contributes 1 bit to the subarray output, and is comprised of two *chunks*. While the sliced LLC architecture described by Huang et al. [36] was presented in the context of a Xeon E5 server processor, the sliced LLC design style is widely used in low-power, edge-class processors such as Samsung Exynos SoCs. That is, the ideas presented in this work lend themselves to any modern LLC architecture that uses a slice-based design, and are not restricted to server-class processors. In this work, we consider an architecture, with sub-arrays of 8KB, for a total slice capacity of 1.25MB.

With this architecture in mind we present four observations. **(1)** the organization of the sub-arrays, makes introducing any new logic inside the cache data arrays very expensive. **(2)** Sub-arrays in a way operate in lock-step, accessing their cells in parallel. **(3)** Since cache lines are not interleaved between data arrays of multiple ways, individual ways can be accessed, modified, and even powered down independently. **(4)** While cache accesses can take several cycles, individual data array operations are only 1 to 2 cycles, and bit line sensing is 1 cycle long [36]. Data arrays in a way share a data bus, thus serializing cache line reads and writes.

Reconfigurable Architectures: Field programmable gate arrays (FPGAs) are virtually synonymous with reconfigurable computing, and can be implemented in several ways. Typically, an FPGA is comprised of an array of logic blocks, i.e. Configurable Logic Blocks (CLBs), organized in an island layout with programmable routing structures, such as switch boxes (SB) and connection boxes (CB), providing the interconnect between each tile, as shown in Fig. 3. The CLBs are comprised of several basic logic elements (BLEs), each of which includes a look-up table (LUT) and a flip-flop. Modern FPGAs include special IO (input/output), DSP, and memory blocks as well. FPGA LUTs are comprised of a *multiplexer tree* or *mux tree*, and SRAM configuration memory that stores the configuration bits for the Boolean function implemented by the LUT. Thus a K-input LUT or a K-LUT would need 2^K SRAM cells to store its function. Fig. 3c illustrates a 3-LUT. FReaC Cache builds on this idea to deploy dense *LUT arrays*, as we shall describe in the next section. Global routing structures that connect CLBs, such as switch boxes and interconnect wires, are responsible for the bulk of the delay in FPGAs, and can occupy nearly 80% of the area [41]. Thus, in this work, we limit the use of global routing structures.

Logic Folding: Logic folding leverages dynamic reconfiguration to allow large circuits to be implemented with limited logical resources by *folding* the circuit over time and sharing the available logic resource across time, i.e. *temporal pipelining* [27], [42]. Thus, relatively large circuits can be implemented in a smaller area, albeit with a longer latency. In Fig. 4a, each node in the graph is a look-up table (LUT) in a combinational circuit. By partitioning the graph into four levels, we can now implement each level as a state of the temporal pipeline, thereby requiring only three LUTs rather than ten, but increasing the latency to four timesteps. At each timestep, the three LUTs must be reconfigured to implement the next level's operations. Thus, the circuit can be realized in 4 cycles if we can reconfigure every cycle. Dependencies between levels are handled by latching outputs.

III. FReaC Cache

FReaC Cache partitions the last level cache (LLC), and uses a portion of it to build reconfigurable logic. In order to do so, FReaC Cache builds on two key ideas: (1) Dense reconfigurable logic can be realized by leveraging the LLC's SRAMs for LUT configuration memory, and by minimizing complex global routing. (2) Logic Folding, as described in Sec. II, and illustrated in Fig. 4a, allows us to trade latency (clock cycles) for reduced resource requirement per cycle in order to map circuits. The high frequency offsets latency penalties incurred during the folding process.

In this section we describe the individual components of FReaC Cache, and the steps involved in transforming and running accelerators in the LLC. Fig. 5 presents a high-level overview of the end-to-end operation of FReaC Cache in six steps, for a single LLC slice. ①: In order to leverage FReac Cache as an accelerator, a portion of the LLC must be selected to operate as an accelerator. ②: Since the entire way is consumed to form compute logic, dirty lines in the selected ways must be flushed. ③: Next, the selected ways

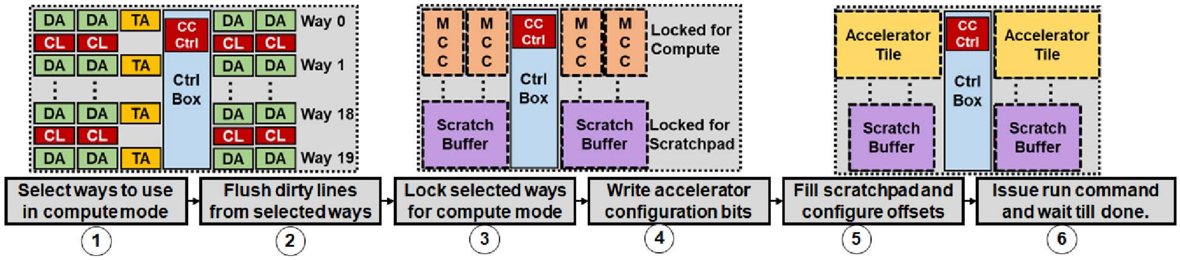

Fig. 5. End-to-end operation of accelerators in FReaC Cache, in a single LLC slice.

are locked for *compute mode*. Note that in order to flush and lock cache ways, we leverage the existing cache control box by introducing our own compute cluster controller (CC Ctrl). The host interacts with the CC Ctrl unit via native load and store (LD/ST) instructions only. ④: With the ways ready for compute mode, we write the accelerator configuration bits. We will discuss the mapping of accelerators and generating logic folding schedules in more detail in Sec. IV. ⑤: The host can fill scratchpads and configure any offsets if needed before beginning computation. ⑥: Finally, the host issues a run command, via LD/STs to the CC Ctrl unit, and waits for the operations to complete. Once the accelerators have completed, a new set of accelerators can be programmed or new data can be provided to the existing set of accelerators by repeating steps ④ and ⑤.

In FReaC Cache, we transform the slice's existing sub-arrays into *Dense Compute Sub-Arrays* and then group the compute sub-arrays with additional logic structures into *Micro Compute Clusters*. We call this slice a *Reconfigurable Compute Slice* . Fig. 6a,b and Fig. 4b illustrate these components, and we will now discuss them in detail.

A. Dense Compute Sub-Arrays

As described in Sec. II, a look-up table (LUT) is comprised of configuration memory and a mux-tree, and a LLC sub-array is capable of providing a fixed number of bits on each access. Thus, each row of the sub-array can hold the configuration bits of one or more LUTs, and by *cycling* through the rows of the sub-array, one by one, we can implement a different LUT(s), and hence a different logical operation, on each access. That is, each row of a sub-array can implement a temporal pipeline stage in logic folding. In order to realize this the sub-array is paired with a *mux tree*, via a memory latch, as shown in Fig. 4b. The memory latch, in conjunction with the mux tree, has thus formed a single look-up table. Note that the inputs to the mux tree are the LUT inputs. Upon reading a new row, the LUT is reconfigured to perform a new operation. Since the sub-array is relatively small, each access can take place in one cycle [36]. Thus, we can dynamically reconfigure the LUT on every cycle. Since a single LUT may not be enough to realize a Boolean circuit, the output of the LUT may be stored in the *state latch*, to be fed back into the input of another LUT at a later time step. Crucially, the mux trees, latches, and other extra logic are external to the sub-array, and do not disturb the existing memory design.

In Fig. 4b we show a single mux tree. This pairing would result in a single 5-input LUT for each row of the sub-array, since each sub-array in a slice is capable of providing 32 bits per access (Sec. II). However, several smaller mux trees can be used to create two 4-input LUTs, or four 3-input LUTs, and so on. The size of LUTs is a well studied topic, and LUT sizes of 4 to 6 are considered the most optimal [43], and used in commercial FPGAs [44], [45].

B. Micro Compute Clusters

A single compute sub-array may require a large number of folding cycles in order to realize a logic folded circuit. Thus, we propose organizing the compute sub-arrays into *micro compute clusters* (MCC) by grouping every two adjacent data arrays. i.e., four sub-arrays, into a micro compute cluster, as shown in Fig. 6b. Within a micro compute cluster, each sub-array activates one or more LUTs per cycle, with the help of the latch and the mux-tree placed outside the sub-arrays. In order for the LUTs realized by each compute sub-array to operate together, we add an operand crossbar, similar to the kind found in the CLB of an FPGA (Sec. II), in the cluster as well. Next, we provide a small bank of registers to store intermediate states from the folded circuit and implement sequential logic in the original design. Finally, since arithmetic operations such as multiplication are expensive to implement with LUTs, we add a dedicated integer multiply-accumulate (MAC) unit as well. The additional logic structures introduced are placed outside the sub-arrays, and are spaced between the two data arrays. Thus, we do not affect the area or timing of the memories.

Operation: A logic folded circuit can now be realized within a micro compute cluster by implementing each level of the circuit, cycle by cycle, by loading a new configuration from each sub-array. In order to simplify this, we store the configuration bits of each level in sequential addresses in the sub-arrays, and reuse the existing address busses to step through addresses. The number of steps (folding levels) is determined by the logic folding schedule (Sec. IV). The schedule is executed and managed by a micro compute cluster controller (CC Ctrl) unit that we add to the cache's control box. Next, at each time step, an operand may be buffered in a register or LUT or fetched from the bus. This movement is facilitated by the operand crossbar, which must be configured for each time step, as determined by the schedule. Hence, the crossbar requires configuration bits as well, which are stored in the way's Tag/State arrays, which are not in use while the way

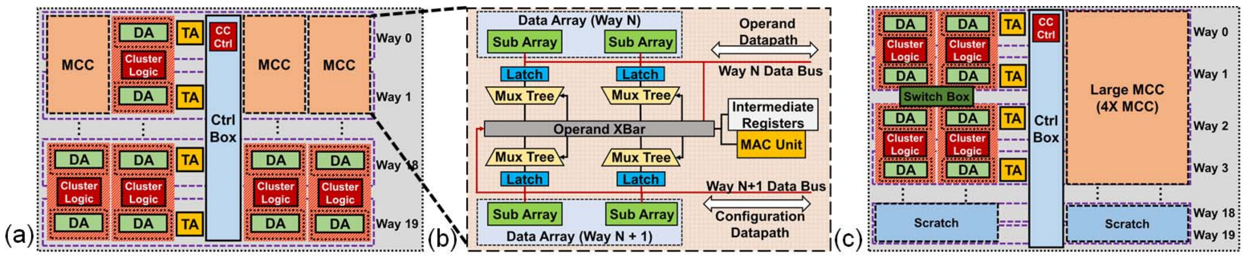

Fig. 6. (a) Reconfigurable compute slice. (b) Micro Compute Cluster (MCC) architecture. (c) Large MCCs and scratchpads.

is being used for compute. Thus, we eliminate the need for additional configuration memory.

Physical Design Considerations: We chose data arrays in adjacent ways, rather than neighboring data arrays, when creating micro compute clusters, for two reasons: First, data arrays are laid out in quadrants (Sec. II) running across all ways. Thus, arrays in the same quadrant are easier to group, without crossing any layout boundaries. Second, interconnecting data arrays within a way would require longer wires, increasing power and latency. Next, we introduce a latch between the data bus and the mux tree. This prevents a long timing path between the sub array output, through the mux and to the crossbar, and does not require the sense amplifiers to be redesigned. The crossbar outputs that drive the data bus are buffered as well. Note that we have chosen to group 2 data arrays into a micro compute cluster because it lends itself easily to the baseline slice architecture [36].

To further minimize overheads, we reuse the existing data busses from both ways. As shown in Fig. 6b, we dedicate one data bus for external operand movement, and the other for moving crossbar configuration data. During compute operations, LUT configurations are read from their respective sub-arrays, while the CC Ctrl unit fetches the crossbar configuration from tag arrays and broadcasts it to clusters, prior to the scheduled time step. The CC Ctrl unit also loads configurations into the arrays via the existing data busses through simple memory store operations.

C. Reconfigurable Compute Slices Operation

Fig. 6a illustrates the LLC slice with micro compute clusters. For illustrative purposes, we present eight CC tiles, and five tiles are shown with all of their logical components - two data arrays and the cluster logic (CL). Cluster logic includes the latches, mux trees, MAC units, registers and crossbar, as in Fig. 6b. Note that since micro compute clusters are built using data arrays across two ways, two ways are completely consumed at a time, such that four compute cluster (MCC) tiles are formed in their place. For maximum flexibility, we add cluster logic to all data array pairs. This allows us to consume an entire cache slice or just a fraction of the slice on-demand, effectively partitioning and reconfiguring the slice on the fly in order to enable computational logic. We also introduce a compute cluster controller (CC Ctrl) unit to the slice's control box to assist with locking and flushing ways, and the control and coordination of clusters. The CC Ctrl leverages the cache controller's existing features and mechanisms to accomplish its tasks. Incoming requests from the cores are serviced by the LLC controller, and the CC Ctrl Unit does not interfere even if a portion of the cache is being used for compute. If the entire LLC is consumed for compute, then core requests are treated as misses, and forwarded to memory.

Host Interface: FReaC Cache does not require custom instructions. The host interacts with the accelerators and CC Ctrl unit via load and store (LD/ST) operations. A range of addresses per slice is reserved for FReaC Cache operations, such that control registers for the CC Ctrl unit are exposed to the host core. The host sets up the LLC slice for compute by writing to control registers in the CC Ctrl units. This setup includes selecting, flushing, and locking ways for compute (Steps ①, ②, and ③, in Fig. 5). In order to configure the accelerator, the host writes micro compute cluster configuration data to a specified address in CC Ctrl Unit, which in turn writes the configuration data to the cluster sub-arrays (Step ④). The host can then fill scratchpad buffers (to be discussed later), and set up accelerator address offsets, by writing to another range of addresses (Step ⑤). Again, the CC Ctrl unit is responsible for forwarding the data into the corresponding sub-arrays. Finally, a *run* register is allocated as well (Step ⑥). These control and data registers are unique to a cache slice, and the setup and configuration must be performed once per slice. The address space and the control registers can be exposed to user code via limited OS support. With the help of a kernel driver, the physical address range can be assigned virtual addresses (`ioremap()` operations), and then exposed to user space via a character device driver, which can be accessed from the user program via an `mmap()` operation.

Setup and Configuration: Steps ①, ②, and ③, in Fig. 5 outline how an LLC slice is set up for compute. First, ways must be selected and flushed, and then locked for compute mode. The mechanisms for enabling this are already available in modern LLCs, and are leveraged by the CC Ctrl unit. As described in Sec. II, ways in the cache are independent of each other, and thus it is possible to instruct the cache control logic to ignore a group of ways. LLCs already include sleep logic [40] [36] to save power, as well as fuse bits to turn off ways in case of poor yield or manufacturing defects. Existing LLCs can also exclusively allocate cache ways to individual cores, thereby modifying the effective LLC seen by the other cores [46]. However, prior to configuring the ways as compute, the ways must be flushed of dirty cache lines. The overhead of flushing out the ways depends on several factors, including: inclusion policies, cache hierarchy, memory bandwidth, and how many lines are dirty. In the worst case, if all lines in the LLC must be flushed, then flush speed is limited by off-chip memory bandwidth. For a 10MB LLC, this can be in the order of hundreds of microseconds[2]. Once the ways are flushed and locked into compute mode, they do not participate in caching. The remaining ways continue to operate as a part of the LLC. The host can then write configuration bits (Step ④ in Fig. 5) into the micro compute clusters, via the CC Ctrl unit in the slice. Once configuration bits for an accelerator have been loaded, they needn't be fetched again unless the configurations are evicted or overwritten.

D. Accelerator Operation

FReaC Cache is a tiled architecture, wherein each micro compute cluster (CC) can operate an independent computation unit (accelerator tile) of its own, by mapping accelerator circuits to it, as shown in Fig. 5. In order to do so, the accelerator circuit is folded and scheduled, with each time step being mapped to LUTs, MACs, and flip-flops (Sec. IV). On each time step the cluster can access up to four 5-LUTs or eight 4-LUTs, one MAC, and one bus operation.

Micro compute clusters in a way share an address bus, and thus operate in lock-step. To further simplify the design, and reuse as many structures as possible, we tie the address lines of all clusters. Since all clusters run the same accelerator and have the same schedules, all accelerator tiles operate in lock-step. As discussed earlier, the CC Ctrl unit is responsible for stepping through the schedule and broadcasting the next address for the clusters on the address bus.

Operand Movement: In order to provide access to external operands, we propose using one of the data busses as the operand datapath (Fig 6b). The cluster first places the address of the operand on the bus, which delivers the address to the CC Ctrl Unit. The CC Ctrl unit processes the address, applies any offsets if needed, and hands it over to the cache controller to be serviced. If the cache slice registers a hit locally, the operand is forwarded back to the clusters via the same data bus. A similar process is followed for write requests. Since the clusters operate in lock-step, multiple requests may be received at once, and the clusters will stall till all requests are serviced. Unlike the CPU core, the clusters wait for the writebacks to complete. In both read and write cases, the cache is responsible for coalescing requests, if it has the capability. Since data arrays share a bus, requests and responses may need to be serialized across multiple cycles.

FReaC Cache Scratchpads: To fully exploit the capabilities of FReaC Cache, we introduce support for scratchpads. By locking-out ways in the cache, we allow the CC Ctrl to route accelerator loads and stores to the sub-arrays in the ways reserved for the scratchpad. Using the existing cache line mapping, a total of 32 bytes may be loaded at a time from each way. However, due to the shared data bus between sub-arrays and the narrow datapath in the cache's control box, delivery of words is serialized. We use the processor core to fill the scratchpads, and thus enable the cores to *initialize* data directly into the scratchpads. In doing so, we avoid the need to flush data out of upper level caches, and the overhead of copying data into the scratchpads (Step ⑤ in Fig. 5). Scratchpads are not necessary for FReaC Cache, but most accelerators use local scratchpads for improved performance and power. In addition, scratchpads help address the LLC's inability to access the TLB, which can add overheads. Without scratchpads or access to the TLB, FReaC Cache would require: (1) Working set is flushed out of upper level caches, (2) Cores do not touch the data while the accelerator is operating, (3) Core provides the physical address, and (4) Data is contiguous and pages are pinned in the host memory.

E. Large Micro Compute Clusters and Multi-Cores

Enabling Larger Compute Clusters: By restricting accelerators to a single micro compute cluster, we limit them to four to eight LUTs per cycle (assuming 5-input or 4-input LUTs). For control or logic heavy applications, that may have large LUT-based circuits, this may result in a large number of fold steps and hurt performance. Thus, we propose the addition of lightweight FPGA-styled switch boxes, where each switch box performs *static routing*, with segments connecting neighboring micro compute clusters. We can now group 4, 8, 16, or up to 32 compute clusters to form a single large accelerator tile, with more LUTs available per cycle. Fig. 6c presents the final slice overview, and illustrates an example where an accelerator tile is formed by using four MCC, and two ways are used to form a scratchpad.

Due to the density of the compute clusters, the limited number of LUTs per cluster, and the short distances between clusters, supporting such global routing structures is not as expensive as on traditional FPGAs. Moreover, a single cache slice is significantly smaller than an FPGA, making routing bits from one end to another possible within a single clock cycle. We explore overheads and timing in more detail in Section V.

FReaC Cache in Multi-Core Systems: In FReaC Cache, accelerators implemented in each slice operate independently of each other. Communication between accelerators is performed via the global address space, as in other data-parallel architectures like GPUs. In the case of large compute requirements, the problem can be broken down into smaller independent problems, which are worked on by each slice's accelerator(s). Thus, FReaC Cache is very amenable to data-parallel problems. Note that the switching infrastructure that interconnects compute clusters is limited to a single slice as well. Thus, the size of an accelerator tile is limited by the size and associativity of an LLC slice. Overall performance is determined by the associativity, number of LLC slices, and the total number of MAC units in some cases.

IV. Mapping Accelerators

FReaC Cache is a flexible architecture; a portion of the LLC slice can be used for computation, while the rest can be used

[2]Assumes four channels of DDR4.

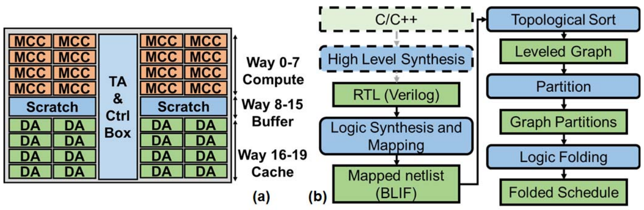

Fig. 7. (a) Slice partition example. (b) Mapping flow.

as cache, as illustrated in Fig. 7a. Assuming 8KB sub-arrays, the slice provides: a 256KB cache, a 512KB scratchpad, and 16 MCC tiles which can be used to implement one or more accelerator tiles per slice. Using multiple slices will increase the number of accelerators and scratchpads.

Logic Folding: In order to map the accelerator to a slice, we must construct a folding schedule. Scheduling and mapping for logic folding has been explored in previous work for FPGA styled designs [42], [47]–[49]. In order to study the efficacy of FReaC Cache's architecture, we created a mapping flow, based on previous work, that provides a schedule that describes how many cycles the circuit is spread over, as well as the resources used in each cycle within a cluster. Recall that the schedule is common to all compute cluster tiles used to form an accelerator. Fig. 7(b) illustrates our proposed flow.

We begin with an RTL design of the accelerator. In this work we use high-level synthesis (HLS), but we are agnostic to the source of the RTL. As with any new acceleration paradigm, accelerators in FReaC Cache must be tuned for the architecture. Since logic folding already performs temporal pipelining, traditional performance *optimizations*, such as pipelining, may result in longer folding schedules and hurt performance. In addition, micro compute clusters only have one operand path, the data bus, and scratchpad buffers are outside the clusters. Thus, an accelerator tile should be designed with a single memory port, and no internal memory buffers. Note that multiple tiles are instantiated to leverage all the available internal bandwidth in the LLC slice. Once we have the accelerator RTL, we use the open-source VTR toolchain [50] to perform logic synthesis and technology mapping, in order to map the circuit into a netlist of look-up tables, flip-flops, adders, and multipliers. The synthesized circuit, or netlist, is a directed graph (DAG), where the primary inputs (PI) and primary outputs (PO) of the DAG are the accelerator module's ports - memory, enable, start, done. Next, our folding algorithm begins by performing a topological sort of the input DAG, which is then used to produce a *leveled graph*, as shown in Fig. 4a. The leveled graph is a re-organization of the input DAG into levels, where each level consists of nodes (vertices) with no dependence on each other, but with incoming edges (dependencies) from nodes in a higher level [42], [47]–[49]. Next, we partition the DAG based on resource constraints derived from how many micro compute clusters are used per accelerator tile. A graph partition is formed by grouping one or more levels together until a resource constraint is violated. We then perform logic folding within the partition to ensure all constraints are met. Since logic folding is performing temporal pipelining, each schedule step realizes a combinational logic path [49]. The final schedule is then the sequence of each partition's schedule[3]. The schedule determines the number of times the circuit is folded (folding cycles or steps), which in turn determines the *effective clock rate* of the circuit. A circuit folded N times would require N cache clock cycles to implement the entire circuit, making the effective clock rate of the circuit as $CacheClock/N$. Hence, minimizing the number of logic folding steps improves performance.

V. Evaluation

Our evaluation makes use of the gem5 [51] simulator. We have implemented a cycle-accurate timing model within gem5 to model FReaC Cache's performance by accounting for: the folding schedule of each benchmark accelerator' synthesized circuit, the contention on the cache buses when moving operands from scratchpad to clusters, cluster IO bandwidths, and loading of operands to the accelerator tiles. For each benchmark accelerator, we perform an RTL simulation to generate traces of memory accesses used, and the exact number of cycles it runs for. The system we simulate is an 8-core ARM micro-architecture, similar to the A15s in an Exynos-5 SoC, and is described in Table I. We use McPat [52] and Cacti 6.5 [35] to generate the size, power, and latency of the memory arrays (Table II). For our simulation, we consider the latency and power of reading a word from a subarray, not the latency of fetching an entire cache line from the L3. Thus, we see that the latency of reading a single word from a subarray allows us to perform one read per cycle, thereby allowing us to *reconfigure* our subarray every cycle as well. Similarly, movement of data from subarrays in one way to another requires movement along a shared databus within the cache control box and is also serialized. We estimate the total leakage power of the LLC is 1.125W via McPat.

For our evaluation, we selected a mix of benchmarks from MachSuite [53], and a few handwritten, that were well suited for FReaC Cache's intended use case, and represented compute, memory, and logic (LUT) bound apps. We excluded benchmarks such as n-body molecular dynamics (KNN, GRID) and DNN training (backprop), as we targeted edge processing in this paper. FReaC Cache is capable of accelerating small but important kernels that would benefit from the low-latency, high-throughput, and high-bandwidth data access of FReaC Cache. Hence we focus on kernels, rather than large multi-phase applications. Since the original benchmark datasets were very small, we scaled the problem by a factor of $256X$ in a batched fashion. Work is divided evenly across all available accelerator tiles/CPU threads in a data parallel fashion.

[3]The small size of the CC tiles allows us to design the crossbars and switches such that data and configuration bits can move to LUTs and MACs in the time it takes to reconfigure the next level (1 cycle). Thus, our folding solution does not need to create multi-cycle paths.

TABLE I
SYSTEM SIMULATION PARAMETERS

Parameter	Value
ISA/Num Cores	ARM/8 cores
Fetch/Decode Width	3/3
Dispatch/Issue/Commit Width	6/8/8
Clock	4GHz
L1D Cache Size/Ways/Latency	32KB/2-way/2cycle
L2D Cache Size/Ways/Latency	256KB/8-way/10cycle
L3D Cache Size/Ways/Latency	10MB/20-way/27cycle
L3D Cache Slice Number/Size	8/1.25MB
Memory Controller	4 channels, DDR4-2400

TABLE II
MEMORY PARAMETERS (32NM)

SRAM Subarray			
Size	Dimensions	AccessTime	AccessEnergy
8KB	0.136 X 0.096mm	0.12ns	0.00369nJ

L3 Cache Slice			
Size	Height	Width	Data Subarrays
1.25MB	1.63mm	1.92mm	160

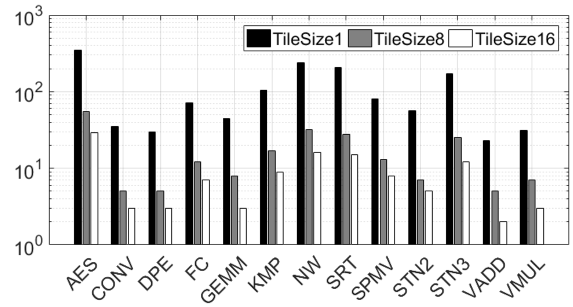

Fig. 8. Folding cycles needed by accelerators. Data is shown on a log scale.

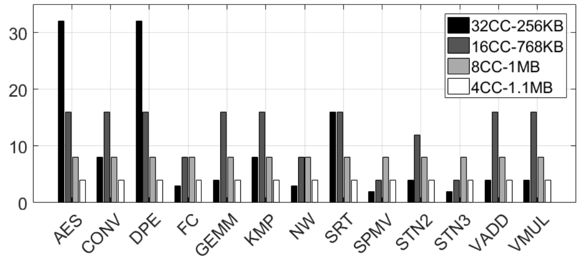

Fig. 9. Impact of compute to memory allocation ratio on number of accelerator tiles. Accelerator tile size 1.

A. Evaluating Area and Timing Overheads

In order to evaluate the overheads involved in FReaC Cache, we use Cacti [35], McPat [52], DSENT [54], and RTL synthesis with a 45nm library that is scaled to 32nm. FreaC Cache adds the following components to form a micro compute cluster (MCC): Mux Trees, Operand XBars, Intermediate Registers, and MACs. The organization and functionality of MCCs are far less complex than those of a processor, and we take special consideration to minimize impact on cache timing: (1) By adding buffers, we avoid loading existing buses. (2) We do not modify the memory arrays themselves. (3) The majority of the key components are wire heavy but localized to a cluster, and modern process nodes have high wire density. (4) As we shall demonstrate, the new components have negligible area, thus their addition doesn't significantly impact critical wire paths.

We take a conservative approach to our area and delay modeling, and consider the worst case. Since we add new wires in short segments, we can reasonably estimate these via Cacti and DSENT, as in [55]. In particular, Cacti version 6+ was developed to be cognizant of wire delays in large caches [56]. First, we consider micro compute clusters. Four components have been added: an operand xbar, mux-trees, intermediate registers, and a MAC unit. We use RTL models to estimate the cost of a 32bit MAC unit and 256 intermediate value holding flip-flops to be $1011\,\mu m^2$ and $1086\,\mu m^2$ respectively. Next, we use DSENT to estimate the cost of a 32X1 Mux tree to be $45\,\mu m^2$ and the operand crossbar to be $1239\,\mu m^2$. Thus, the total area added per cluster is $0.0034\,mm^2$. If we enable 32 clusters in the slice, using 16 ways, the total overhead is $0.109\,mm^2$, which is just 3.5% of the total area of the LLC slice described in Table II. This enables the basic FReaC Cache mode of 32 independent accelerator tiles per slice.

However, as we described in Sec. III-E, there is a potential benefit in enabling larger clusters. In order to do so, we consider FPGA-style island routing as shown in Fig. 3. To do so, we place a switch box in-between groups of four micro compute clusters, and an additional switch box to cross the tag arrays and control box, to enable X-Y routing. Hence, we have a total of 28 (7X4) switch boxes, placed across 16 ways of the cache, creating an interconnect fabric between the 8X4 micro compute cluster tiles. Note that FPGA routing structures and interconnects can be designed to be placed on top of buffers and logic [57]. Hence, once we determined the area of logic blocks, we could determine the lengths of wires and interconnects. We then swept the frequency of our model with DSent and CACTI until timing wasn't violated in the worst case, and thus settled on 3GHz for large compute clusters, and 4GHz for small compute clusters. The longest path possible is the Manhattan distance between two switches at opposite corners of the slice. We found this to be $2.864\,mm$, based on the geometry of the cache slice and subarrays, which must be completed over 10 links between the switches, and must meet a delay of $0.3\,ns$ to complete within a cycle. We consider 32 bit links, and compute the total area of global routing and links to be $3469\,\mu m^2$. Finally, the switch boxes will require configurations as well, and we add a wide output 8KB memory for every four micro compute cluster. This adds a total overhead of $0.35\,mm^2$. Note that this is only necessary if we need to operate very large accelerator tiles at 3GHz. Thus, we add a total of $0.48\,mm^2$ or 15.3% overhead to the slice. This is a conservative estimate as we opted for short links and more switch boxes, and hence added more configuration memory for the switches.

B. Accelerator Design Space

We begin by exploring the design space of accelerators being mapped to FReaC Cache. We synthesized the benchmarks using Xilinx Vivado HLS. First, we explore the impact on the

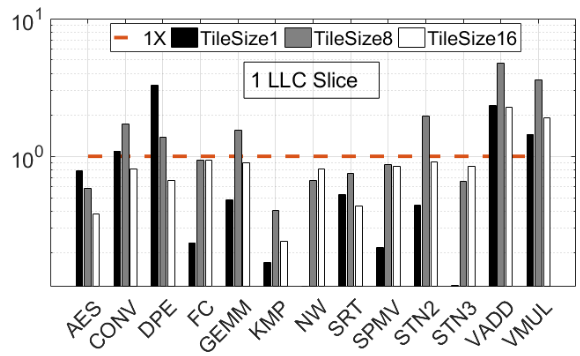

Fig. 10. Speedup as a function of accelerator tile size. Data is shown on a log scale.

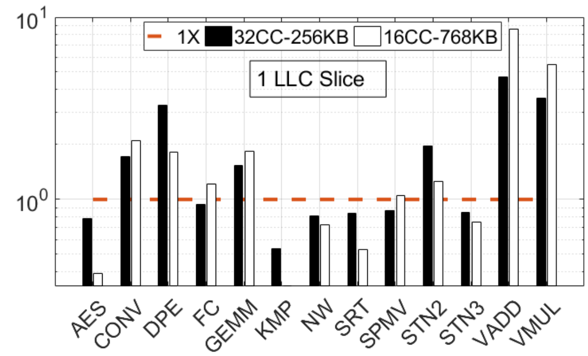

Fig. 11. Speedup as a function of MCC to mem ratio. Data is shown on a log scale.

number of compute cluster tiles used to realize an accelerator tile. The more MCCs that are available per accelerator, the more resources can be allocated per folding step, and thus fewer folding cycles. We present the number of folding cycles for each of the benchmarks in Fig. 8 across different tile sizes. While allocating more MCCs per accelerator tile reduces the number of folds, there is a trade-off with the number of concurrent accelerator tiles per slice.

Thus, there is trade-off between the latency of an accelerator tile and net throughput. This trade-off also requires considering the impact of working set proximity. In order to maximize the performance and efficiency, the working set must be made available in the scratchpad buffers. Thus, the number of concurrent accelerator tiles is also limited by the working set of each accelerator tile. To illustrate this, we consider different ratios of the LLC being partitioned to compute and memory. We start with 16 ways for compute and 4 for memory, creating 32 MCCs and a 256KB scratchpad, and sweep down to 2 ways for compute and 18 for memory, creating 4 MCCs and a 1.1MB scratchpad. Fig. 9 presents the max number of accelerator tiles, of tile size 1 (1 MCC per tile), that can be fit in a single slice. Accelerators with smaller working sets, such as AES and dot product engines, are able to fill all 32 MCC tiles with accelerators. However, both computational and memory-bound kernels such as GEMM, KMP, Sorting, and Stencil reach maximum number of tiles (and hence, throughput) by allocating more of the LLC to scratchpads. Note that this is a function of the accelerator's working set and the number of slices available. Overall, we observe that an organization of 32 MCC and 256KB scratchpad, and 16 MCCs with 768KB scratchpad, allow for the most accelerator tiles to be instantiated in a single slice.

C. *FReaC Cache Performance and Efficiency*

We compare FReaC Cache to its base system's eight ARM cores, a large PCIe-attached FPGA, Xilinx ZCU102 FPGA, and a standalone Ultra 96 SoC FPGA system as well. We use OpenMP to parallelize the baseline benchmarks, in a data-parallel fashion, across all available physical cores. We modeled the power of the ARM cores via McPat [52] with a 32nm low-power library. To evaluate the FPGA, we synthesized the benchmark circuits with all optimization directives enabled. Next, we attempt to instantiate 256 copies of the benchmark IPs, to reflect maximum data parallelism. If all copies do not fit, then we batch the workloads, and scale latency accordingly. We include a 160 μs latency for DMA and configuration overheads [17], and also include the cost of transferring data to the FPGA via PCIe3.0 x16 for the ZCU102, and cost of transferring over AXI busses in the Ultra 96 (U96). We then use the Xilinx Power Estimator (XPE) [58] to estimate power, and account for the board idle and leakage power as 12W [18] for the ZCU102. The latency of benchmarks on FReaC Cache was provided by our Gem5 simulator. To provide the best performance, and consistency with our FPGA comparison, we move data into the scratchpad buffers. We measure the latency of the kernel's operation in FReaC Cache accelerators, and the latency to transfer the datasets to the scratchpad buffers. When loading data into the buffers, we load LLC slices in parallel, thereby making full use of the LLC's bandwidth. Kernel latency also includes the time to write the configuration data. We estimate the power of FReaC Cache by accounting for the number of reads from the compute clusters and scratchpads. We also assume the links between the switch boxes run at 100% load and consume about 9 mW per link, and we add leakage power. We present all data relative to a single thread of the A15 host.

In order to better understand FReaC Cache, we first examine a single cache slice and the impact of accelerator tile sizing. We consider a slice with a 32MCC-256KB partitioning, thus consuming all 20 ways of the slice. We then sweep across accelerator tile sizes, allocating 1, 8, and 16 MCCs per accelerator, and measure the speedup of kernel execution over a single host core (A15). We also constrain our exploration with the max number of accelerators per slice, as shown in Fig. 9. In Fig. 10, with the exception of AES, we see that increasing tile size improves performance. However, we see a reduction in performance with tile size 16, since tiles of 16 or more MCCs require a reduction in clock speed. As we can see in Fig. 8 and Fig. 9, AES has a very high folding overhead but can fit several copies in a single slice. Thus, it is better suited for multiple tiles per slice, with few MCCs per tile. As we noted earlier, there is a trade-off between allocating the LLC to memory versus compute. We examine this more closely in Fig. 11. We present the best performance possible, across all accelerator tile sizes, for two different compute-to-memory partitions in a single slice. Once again, we find that AES strongly prefers more compute clusters over buffer memory, along with other computational kernels such as dot product engines, fully connected layers, and GEMM. Note however that we are restricted to a single slice here. The optimal compute-to-memory tradeoff is a function of the total number of slices as well, and we observed that the 16MCC-768KB split proved to be more useful with increasing number of slices participating in the acceleration.

Next, we consider the end-to-end performance of FReaC Cache in our evaluation system. Consuming the entire LLC may not be feasible in practice, thus we reserve two ways, 128KB, per slice as cache. This leaves 10%, or 1MB, of the LLC in place, while allocating the remaining 18 ways for compute and scratchpad. We consider a 16MCC-640KB compute-scratchpad split per slice, and sweep across all possible accelerator tile sizes and cache slices. For the sake of brevity, we report the best performance (speedup) possible for a given number of slices, and also report the corresponding performance per watt (throughput per watt), and power. We present our data in Fig. 12 on a log scale. Speedup is measured over the *end-to-end* latency of the application, and we use a single A15 thread as the baseline. The end-to-end latency includes the cost of initializing arrays, moving them into the scratchpad buffers, and back to the core. For comparison, we include the fully-parallelized eight thread A15 implementation, along with the ZCU102 and the Ultra96 (U96) FPGAs. For the FGPAs, we also include the cost of moving data into their buffers.

As we can see, with increasing number of cache slices, FReaC Cache's end-to-end performance increases. Across benchmarks, FReaC Cache outperforms the ARM cores at a fraction of power. On average, when using all eight slices, FReaC Cache is 8.2X and 3X faster than the single and multi-threaded implementations, respectively. Additionally, FReaC Cache is 6.1X more efficient (Perf/Watt) than the multi-core CPU, on average. FReaC Cache proves to be especially good with memory-bound and computational kernels such as convolutions, dot products, vector add/mults, fully-connected layers, and GEMMs, showing up to 14.5X speedup over single-thread implementations. Logic-heavy apps like AES and sorting (SRT) suffer a higher penalty due to folding. Thus, while they are faster than a single CPU thread, the multi-threaded implementation outpaces them, but at nearly twice the power. However, the large ZCU102 FPGA outperforms FReaC Cache, the A15, and the U96 on most benchmarks. This comes at the cost of a massive increase in power, and we note that the ZCU102 chip is much larger than the LLC as well as the entire A15 chip. The edge-centric lower-power Ultra 96 is bested by FReaC Cache in both computational and memory-sensitive benchmarks. FReaC Cache also proves to be more energy efficient than both FPGA solutions as well. Thus, FReaC Cache proves itself to be highly performant, flexible, and efficient, across benchmarks and domains.

Finally, moving data to and from accelerators can cost time and energy. In the case of NMA and PIM, this may also require additional mechanisms in the core and host OS. To mitigate the cost of copying data to and from buffers, improve performance, and avoid pinning pages to physical addresses, FReaC Cache uses the cores to initialize data directly into the scratchpad buffers. This effectively eliminates a copy operation, but still costs time to do so. In our evaluation in Fig. 12 we considered the end-to-end latency of the application, including the cost of initializing and copy data. In Fig. 13 we present the end-to-end speedup versus the speedup of the kernel only, on a log scale. For reference, we also provide the multi-threaded implementation as well. We observe that depending on the benchmark, copying and initialization can have negligible to 60% overhead. Thus, in some cases, our end-to-end speedup is a fraction of the peak kernel speedup. This is in part due to the size of the working sets, and even the CPU suffers from reduced end-to-end performance due to it. FReaC Cache still manages to outperform the CPU and even the FPGAs. Note that despite the massive bandwidth of the LLC and initializing data directly in the buffers, we are affected by memory initialization delays. However, off-chip accelerators like the ZCU102 and the U96 require a full copy over much slower channels, after initialization is completed, adding even more overhead. Thus, FReaC Cache provides cost-effective, energy-efficient, and flexible acceleration, with a high-bandwidth and low-latency path to the user's working sets, which places FReaC Cache in a highly unique position.

VI. Discussion

In this section we present a discussion to clarify FReaC Cache's motivation, positioning, and potential limitations.

Portability: In this work, we illustrate the ideas and principles of FreaC Cache by incorporating it into an existing architecture. While we use the detailed description of the Intel Xeon LLC provided by Huang et al. [36] as the base of the example FReac Cache design, FReaC Cache is not limited to Intel LLC architectures, and does not rely on any special Intel enterprise features. Instead, our focus is on processors for edge computing in this work. Changing the underlying cache slice architecture may affect the number of effective LUTs per cycle, the size of the micro compute cluster, number of clusters, and number of clusters grouped across ways, etc. For example, if a cache way is comprised of twice as many subarrays, we can have twice as many micro compute clusters, or the same number of clusters, with twice as many LUTs available per cycle. For consistency, we used the architecture described by Huang et al. [36] throughout this paper. Total memory capacity only limits scratchpads sizes and the number of configurations we can store, but not performance.

FPGA-Based Architectures: FReaC Cache is not a new FPGA architecture. Rather, it is a highly cost-effective and power-efficient solution for edge scenarios, where small throughput or memory-bandwidth intensive kernels can be occasionally offloaded. Rather than re-designing reconfigurable

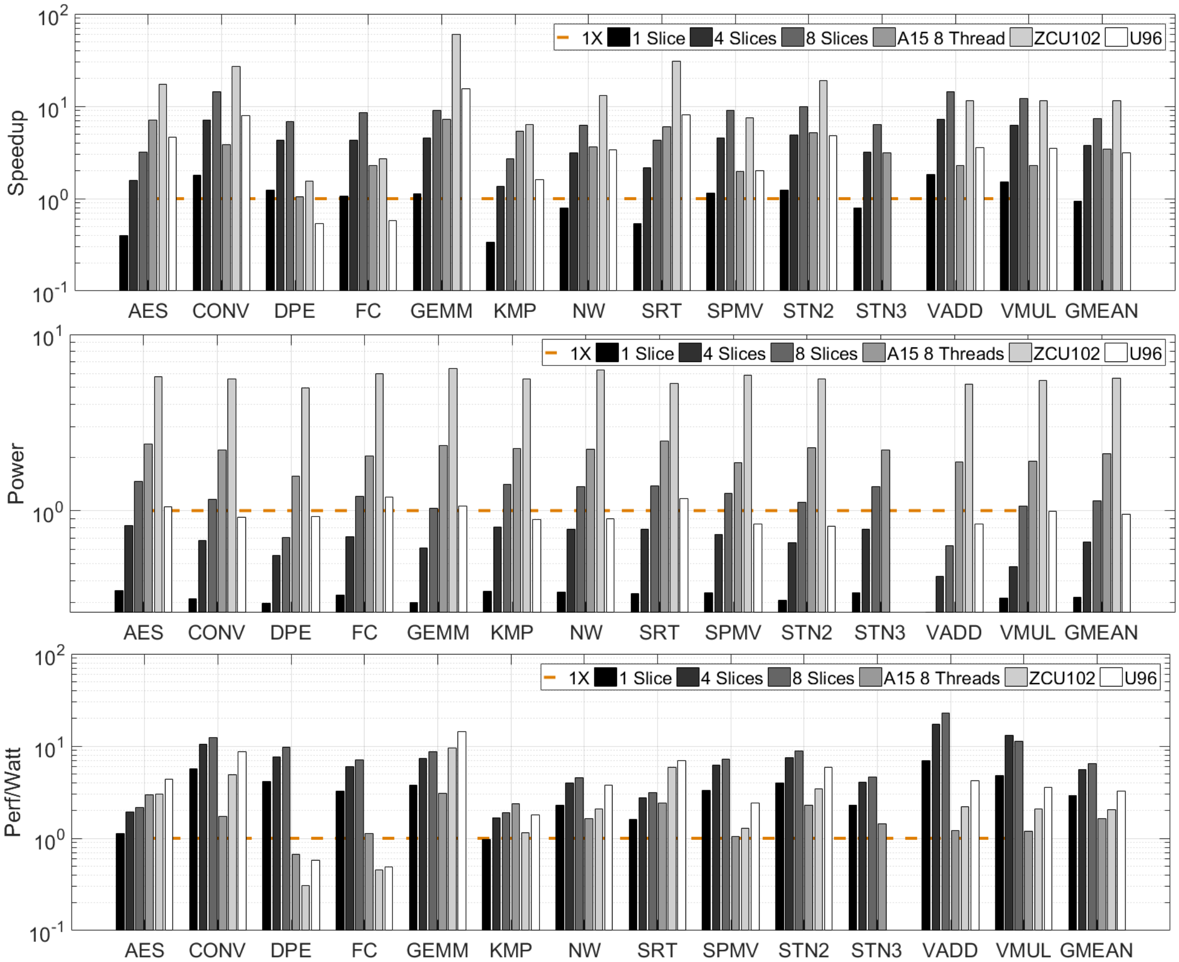

Fig. 12. Relative Speedup, Power, and Perf/Watt, versus number of LLC slices. Data is shown on a log scale.

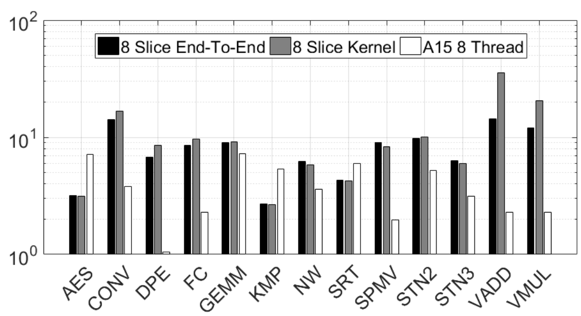

Fig. 13. End-to-end versus kernel speedup.Data is shown on a log scale.

computing logic, our focus is on the best way to reconfigure the LLC and convert it into a customized accelerator that can leveraging near-memory computing. It is not designed to handle general-purpose reconfigurable computing or provide glue logic as typically targeted by generic FPGA devices. While we do use LUTs, like an FPGA, our design and organization is very different from an FPGA, and we do not consider many features of generic FPGAs, including embedded BRAM, rich FF-based control logic, high I/O capabilities, multi-clock domains, etc. Thus, FReaC Cache is not the perfect solution for every application. Where the application is well understood and mission-critical, an FPGA might be a better choice. For example, circuits with large and complex control circuits would be better suited to FPGAs, where the immediate access to thousands of LUTs is more critical. However, FReaC Cache has a significant area advantage over FPGAs as: (1) LLC subarrays occupy the majority of the area, and the additional logic has area fractional overheads, and (2) 80% of FPGA area is spent on routing structures and their configuration bits [41], which FReaC cache avoids. Also, FPGAs have a limited configuration bandwidth of just 400MB/s[4]. FReaC Cache configuration is limited by LLC-DRAM bandwidth and the LLC's internal bandwidth (10s to 100s of GB/s).

Alternative Near and In-Cache computing approaches: We first consider Compute Caches [21], in which the authors propose leveraging bit-line computing to achieve vector computation. Due to the nature of the computation, this approach is limited by the pitch of the sub-arrays, and hence the authors are limited to a simple set of bit operations— AND, OR,

[4]32bit CAP port and 200MHz on Xilinx Ultrascale+ FPGAs [59].

XOR, copy, and compares— which are effective for the data manipulation domain that the authors target, such as string matching, bitmap indexing etc. Crucially, this approach requires significant redesign of the cache and sub-arrays and adds new ISA instructions, which add significant design and verification costs. Also, operands must be placed for sufficient locality in order to perform in-situ processing. The esoteric nature of computation, ISA, benchmarks, and simulation infrastructure in Compute Cache makes it hard to perform an apples-to-apples comparison with FReaC Cache. However, we note that FReaC Cache is much less intrusive. All new logic is placed outside the sub-arrays, existing busses are re-used, and no custom ISAs are used. Thus, we minimize the impact on area, timing, energy, and design of the LLC and the cores. Most importantly, we are not limited to bit-level operations or a restricted domain of applications. While FReaC Cache is general purpose, it is best suited for when the host needs acceleration of memory-bound computational kernels. Where Compute Cache offers average speedups of 1.9X on data-manipulation workloads, FReaC Cache demonstrated an average speedup of 3X across diverse workloads.

Next, we consider near-cache computation, such as BSSync [60], that places computation near the cache, rather than inside the arrays. Instead of placing ALUs in the LLC and adding new ISA instructions, we consider placing lightweight embedded cores (EC), such as an ARM A7, in the LLC. Like FReaC Cache, this provides similar general-purpose capabilities, independent operation from the host cores, and communication between the accelerator cores and host cores can still be done via LD/STs. Each A7 core has an area of about $0.49\,\mathrm{mm}^2$ [61], [62], which is similar to the per-slice overhead of FReaC Cache. Thus, we consider two scenarios: (1) iso-area, where one EC is placed per slice, and (2) two ECs per slice. These provide a total of eight and sixteen cores in the LLC, respectively. For a fair comparison, we allocate 16 ways of the LLC as scratchpad for the cores to use. As we can see in Fig.14, the FReaC Cache based accelerator's customized circuits and effective memory bandwidth utilization enable it to significantly outperform the iso-area 8 EC solution by an average of 4X, and the 16 EC setup by an average of 2X. Thus, FReaC Cache is significantly more area and compute efficient than such near-cache solutions. Note that speedup in Fig.14 is shown relative to a single A15 thread, and the figure includes the performance of all eight A15 cores.

Interference with Host Performance: Since a part of the LLC is dedicated to computation, there is a potential trade-off between CPU and the accelerator performance in FReaC Cache. The solution depends on the application, and whether the CPU and accelerator work cooperatively. The problem, however, is similar to cache-interference [63]–[69] and performance isolation problems seen in chip multi-processors, and multi-tenant cloud scenarios [70]–[72]. In particular, prior work shows that partitioning the LLC can be an effective solution in such mixed-workload and multi-application scenarios. Dedicating a portion of the LLC to accelerators for a process has an effect on the performance of other running processes similar to that

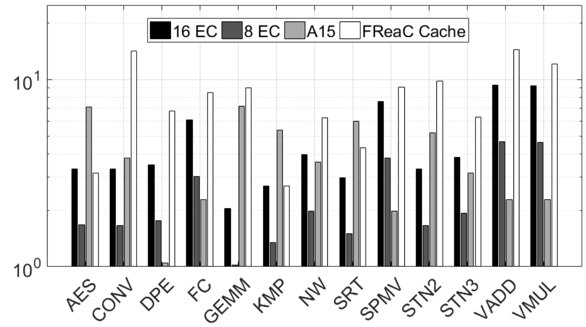

Fig. 14. Kernel speedup for 16 and 8 lightweight embedded cores (EC) in the LLC, versus 8 slices of FReaC Cache accelerators and the 8 cores of the host A15 core. Data is shown on a log scale.

of dedicating a partition of the LLC to that process. A better understanding of how much cache to allocate for compute requires a detailed study to clearly define the level of multi-tenancy and the interactions among concurrent processes, and previous work on cache QoS and interference provides a good foundation for this.

As a first-order analysis, we consider two groups of applications: {AES, NW, STN2, and STN3} and {CONV, FC, KMP, and SRT}. Each group contains applications with a mix of compute and memory bound kernels, as well as logic/branch heavy characteristics. We then consider two scenarios - 1MB and 4MB of the LLC is retained for caching, while the remainder is used to accelerate one of the applications in the group. The remaining three applications are allocated two CPU threads, each. Fig. 15 presents our analysis, and all data is normalized to a single threaded baseline, as in the previous figures. Since three applications, out of the group of four, run on the CPU complex, each application will run three times in total[5]. Thus, we consider an average of the three run times, per LLC capacity. Our study reveals two key points: First, we note that the benchmarks do not exhibit sensitivity to the total LLC capacity. This is primarily due to the fact that the core's L1 and L2 caches are capable of holding the per-thread working set. Our benchmarks run in a batched and data-parallel fashion. So, while the total application working set can be up to 32MB, which is greater than the LLC capacity, the per-thread working set (one element of the batch) does not exceed 128KB. Second, we see that allocating more LLC resources to acceleration results in improved performance for the accelerator. This is expected behavior. However, we note that this is largely related to how many ways can be allocated as scratchpad space, in order for the accelerators to function. Thus, we observe that for our given set of benchmarks, allocating up to 90% of the LLC (9MB) to computation/scratchpad to accelerate one application does not hurt the performance of the remaining three applications, as the remaining 1MB is sufficient to back up the per-thread working sets in the L1 and L2 caches. Here, we see that the FReaC Cache based accelerator can provide between 1.8X and 9X of speedup over its CPU run. Note

[5]For example, CONV runs in the following grouping: {CONV, FC, KMP}, {CONV, FC, SRT}, {CONV, KMP, SRT}.

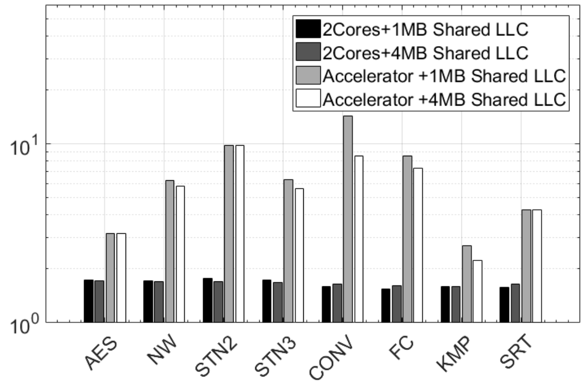

Fig. 15. Cache interference study. Speedup over a single thread under varying shared LLC capacity with concurrently running applications. Data is presented for application running either on 2 CPU cores or an accelerator, with either 1MB or 4MB of the LLC retained. Data is shown on a log scale.

that, whereas in Fig. 12, 13, and 14 we considered up to eight threads of the host CPU per application, here each application is restricted to only two threads while the accelerators leverage all eight slices of the LLC.

Thus, in such a scenario, offloading computational or memory limited applications to the LLC would provide the best overall performance, and would have limited impact on the CPU cores. Should one or more applications be sensitive to LLC capacity, then the user would need to scale back the LLC allocation devoted to computation and/or consider partitioning and allocating the LLC to specific applications [64]–[68]. As our results show, FReaC Cache is still able to deliver acceleration with just 60% of the LLC (6MB). Reducing the amount of LLC allocated for computation, would provide proportional reduction in acceleration. Thus, when possible, FReaC Cache transforms surplus LLC capacity into compute, providing energy-efficient, customized, and cheap acceleration.

VII. Related Work

There have been several recent attempts at Processing in Memory (PIM) [22]–[26], [28], [29]. One strategy towards PIM is to modify the DRAM subarray architecture to enable computation or acceleration within the memory array [24], [25]. Seshadri et al. [25] [24] proposed modifications to the DRAM subarray architectures to enable data copy and bitwise AND and OR operations. Such architectures take advantage of the high bandwidth and low data transfer cost provided by the through silicon vias (TSVs), and do not affect DRAM densities. However, these systems run the risk of breaking coherence, and are expensive. In contrast, FReaC Cache is cost-effective, and the LLC is already the point of coherence in modern multi-core CPUs. In addition, the reconfigurable fabric is not limited to any subset of operations. Li et al. [22] proposed a reconfigurable accelerator architecture by modifying the sense amplifiers in DRAM to support Boolean algebra acceleration. This approach is flexible but requires several changes in the DRAM architecture.

In DRAF [31], the authors explored the idea of a reconfigurable architecture that uses DRAM for implementing an FPGA, in order to provide higher density and lower power. They leverage the large DRAM capacity to support multiple configurations, and the focus is not on providing in-cache or near-memory computing. In contrast, FReaC Cache is not looking to build a new FPGA architecture. Rather, we focus on utilizing logic folding to pack a large amount of logic into existing SRAM arrays in the LLC. This enables us to aggressively reconfigure the compute clusters to achieve the illusion of a larger FPGA and thus provide cheap and easy processing near data. Another similar direction for building reconfigurable architectures in memory leverages CGRA-like architectures [26], [28]. Gao et al. [26] proposed Heterogeneous Reconfigurable Logic (HRL) that resembles both FPGA and CGRA. While this approach targets PIM/NMA, it is a more specialized solution, looking at CGRAs stacked with DRAMs.

Finally, FReaC Cache and Piperench [73] have some similarities in that they try to leverage reconfiguration to cope with limited resources. In FReaC Cache, we perform fine-grained reconfiguration at a gate (LUT) level and focus on realizing any circuit in a limited area. In contrast, Piperench uses a PE-based architecture, and is focused on more pipelined and compute styled workloads. FReaC Cache is more tightly integrated into the memory hierarchy and more area-efficient.

VIII. Conclusion

In this work we presented a novel architecture, FReaC Cache, that leverages the existing subarrays of the LLC and that, without changing the sub-arrays, is able to create dense reconfigurable computing clusters. FReaC Cache can be implemented with minimal overheads of 3.5% to 15.3% in area, and shows average speedups of 3X, and average Perf/W improvements of 6.1X over an edge-class multi-core processor when 90% of the LLC is consumed for acceleration. We also demonstrated FReaC Cache's competitive advantage over modern FPGAs, where FReaC Cache is much more area and power efficient. Finally, we acknowledge that using the LLC for computation can degrade cache performance, but certain applications do not use up the entire LLC. By converting the LLC for computation, we achieve two goals: (1) avoidance of wasted LLC capacity, and (2) near data computation. Hence FReaC Cache provides a cost- and power-efficient solution for acceleration in edge devices.

Acknowledgment

This work was supported in part by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network, the Xilinx Center of Excellence and Xilinx Adaptive Compute Clusters (XACC) program at the University of Illinois Urbana-Champaign, and by NSF award CNS-1705047.

References

[1] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 365–376.

[3] M. B. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *DAC Design Automation Conference 2012*, June 2012, pp. 1131–1136.

[4] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sept 2011.

[5] M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-based In-Line Accelerator for Memcached," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 57–60, July 2014.

[6] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "SCALEDEEP: A scalable compute architecture for learning and evaluating deep networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 13–26.

[7] J. Kung, Y. Long, D. Kim, and S. Mukhopadhyay, "A programmable hardware accelerator for simulating dynamical systems," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 403–415.

[8] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 27–40.

[9] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 553–564.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 1–12.

[11] S. Mashimo, T. V. Chu, and K. Kise, "High-performance hardware merge sorter," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 1–8.

[12] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[13] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May 2015.

[15] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. 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.

[16] IBM, "Coherent Accelerator Processor Proxy (CAPI) on POWER8," 2014. [Online]. Available: http://www.nallatech.com/wp-content/uploads/Ent2014-CAPI-on-Power8.pdf

[17] Y. k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[18] H. Giefers, R. Polig, and C. Hagleitner, "Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, June 2014, pp. 92–99.

[19] S. Saini, R. Hood, J. Chang, and J. Baron, "Performance Evaluation of an Intel Haswell-and Ivy Bridge-Based Supercomputer Using Scientific and Engineering Applications," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2016, pp. 1196–1203.

[20] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era," *IEEE Design Test*, vol. 34, no. 2, pp. 39–50, April 2017.

[21] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 481–492.

[22] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based Reconfigurable In-Situ Accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017, pp. 288–301.

[23] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 336–348.

[24] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2013, pp. 185–197.

[25] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, July 2015.

[26] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 126–137.

[27] W. Zhang, N. K. Jha, and L. Shang, "NATURE: A hybrid nanotube/CMOS dynamically reconfigurable architecture," in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 711–716.

[28] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 283–295.

[29] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 113–124.

[30] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache Automaton: Repurposing Caches for Automata Processing," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2017, pp. 373–373.

[31] M. Gao, C. Delimitrou, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and C. Kozyrakis, "DRAF: A Low-power DRAM-based Reconfigurable Acceleration Fabric," in *Proc. of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 506–518.

[32] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda, "Hardware-software codesign of accurate, multiplier-free deep neural networks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[33] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *ArXiv e-prints*, Sep. 2014.

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, 2012, pp. 1097–1105.

[35] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 3–14.

[36] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel Xeon Processor E5 Family," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1954–1962, Aug 2013.

[37] G. K. Konstadinidis, H. P. Li, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. P. Masleid, C. Zheng, Y. D. Lin, P. Loewenstein, H. Park, V. Srinivasan, D. Huang, C. Hwang, W. Hsu, C. McAllister, J. Brooks, H. Pham, S. Turullols, Y. Yanggong, R. Golla, A. P. Smith, and A. Vahidsafa, "SPARC M7: A 20 nm 32-Core 64 MB L3 Cache Processor," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 79–91, Jan 2016.

[38] J. Chang, S. L. Chen, W. Chen, S. Chiu, R. Faber, R. Ganesan, M. Grgek, V. Lukka, W. W. Mar, J. Vash, S. Rusu, and K. Zhang, "A 45nm 24MB on-die L3 cache for the 8-core multi-threaded Xeon Processor," in *2009 Symposium on VLSI Circuits*, June 2009, pp. 152–153.

[39] W. Chen, S. L. Chen, S. Chiu, R. Ganesan, V. Lukka, W. W. Mar, and S. Rusu, "A 22nm 2.5MB slice on-die L3 cache for the next generation Xeon processor," in *2013 Symposium on VLSI Technology*, June 2013, pp. C132–C133.

[40] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, R. D. Limaye, and S. Vora, "A 65-nm Dual-Core Multithreaded Xeon Processor With 16-MB L3 Cache," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 1, pp. 17–25, Jan 2007.

[41] M. Lin, A. E. Gamal, Y. C. Lu, and S. Wong, "Performance Benefits of Monolithically Stacked 3-D FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 216–229, Feb 2007.

[42] T. J. Lin, W. Zhang, and N. K. Jha, "A Fine-Grain Dynamically Reconfigurable Architecture Aimed at Reducing the FPGA-ASIC Gaps," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2607–2620, Dec 2014.

[43] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 3, pp. 288–298, March 2004.

[44] Altera, "Altera Whitepaper FPGA Architecture, WP-01003-1.0," Tech. Rep., 2006. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf

[45] Xilinx, "UG474 7 Series FPGAs Configurable Logic Block," Tech. Rep., 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[46] Intel, "Improving real-time performance by utilizing cache allocation technology," 2015. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[47] W. Zhang, L. Shang, and N. K. Jha, "NanoMap: An Integrated Design Optimization Flow for a Hybrid Nanotube/CMOS Dynamically Reconfigurable Architecture," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 300–305.

[48] T. J. Lin, W. Zhang, and N. K. Jha, "FDR 2.0: A Low-Power Dynamically Reconfigurable Architecture and Its FinFET Implementation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 10, pp. 1987–2000, Oct 2015.

[49] W. Zhang, N. K. Jha, and L. Shang, "A Hybrid Nano/CMOS Dynamically Reconfigurable System Part II: Design Optimization Flow," *J. Emerg. Technol. Comput. Syst.*, vol. 5, no. 3, pp. 13:1–13:31, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1568485.1568487

[50] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, June 2014.

[51] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[52] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[53] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.

[54] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic, "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, May 2012, pp. 201–210.

[55] T. Zhang and X. Liang, "Dynamic front-end sharing in graphics processing units," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 286–291.

[56] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. USA: IEEE Computer Society, 2007, p. 3–14. [Online]. Available: https://doi.org/10.1109/MICRO.2007.30

[57] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. USA: Kluwer Academic Publishers, 1999.

[58] Xilinx, "Xilinx Power Estimator User Guide," Tech. Rep., 2017. [Online]. Available: https://china.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug440-xilinx-power-estimator.pdf

[59] Xilinx, "DS925 Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics," Tech. Rep., 2020. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf

[60] J. H. Lee, J. Sim, and H. Kim, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 241–252.

[61] H. Mair, G. Gammie, A. Wang, S. Gururajarao, I. Lin, H. Chen, W. Kuo, A. Rajagopalan, W. Ge, R. Lagerquist, S. Rahman, C. J. Chung, S. Wang, L. Wong, Y. Zhuang, K. Li, J. Wang, M. Chau, Y. Liu, D. Dia, M. Peng, and U. Ko, "23.3 a highly integrated smartphone soc featuring a 2.5ghz octa-core cpu with advanced high-performance and low-power techniques," in *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, 2015, pp. 1–3.

[62] Wikichip, "ARM Cortex A7 Microarchitecture." [Online]. Available: https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a7#Memory_Hierarchy

[63] O. Temam, C. Fricker, and W. Jalby, "Cache Interference Phenomena," *SIGMETRICS Perform. Eval. Rev.*, vol. 22, no. 1, p. 261–271, May 1994. [Online]. Available: https://doi.org/10.1145/183019.183047

[64] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 257–266. [Online]. Available: https://doi.org/10.1145/1006209.1006246

[65] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 657–668.

[66] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the Most out of Last Level Cache in Intel Processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303977

[67] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, p. 25–36, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1269899.1254886

[68] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads," *SIGPLAN Not.*, vol. 49, no. 4, p. 729–742, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2644865.2541944

[69] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 340–351.

[70] R. S. Kannan, M. Laurenzano, J. Ahn, J. Mars, and L. Tang, "Caliper: Interference Estimator for Multi-Tenant Environments Sharing Architectural Resources," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3323090

[71] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying Effects of Shared on-Chip Resource Interference for Consolidated Virtual Machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2038916.2038938

[72] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt, "CloudScope: Diagnosing and Managing Performance Interference in Multi-tenant Clouds," in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015, pp. 164–173.

[73] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, April 2000.