Connect On the Fly: Enhancing and Prototyping of Cycle-Reconfigurable Modules

Hao Zhou*, Xinyu Niu†, Junqi Yuan*, Lingli Wang*, Wayne Luk†
†Dept. of Computing, School of Engineering, Imperial College London, UK
*School of Microelectronics, Fudan University, China
Email: {nx210, w.luk}@doc.ic.ac.uk†, {zhouhao, 13210720072, llwang}@fudan.edu.cn*

***Abstract*—This paper introduces cycle-reconfigurable modules that enhance FPGA architectures with efficient support for dynamic data accesses: data accesses with accessed data size and location known only at runtime. The proposed module adopts new reconfiguration strategies based on *dynamic FIFOs*, *dynamic caches*, and *dynamic shared memories* to significantly reduce configuration generation and routing complexity. We develop a prototype FPGA chip with the proposed cycle-reconfigurable module in the SMIC 130-nm technology. The integrated module takes less than the chip area of 39 CLBs, and reconfigures thousands of runtime connections in 1.2 ns. Applications for large-scale sorting, sparse matrix-vector multiplication, and Memcached are developed. The proposed modules enable 1.4 and 11 times reduction in area-delay product compared with those applications mapped to previous architectures and conventional FPGAs.**

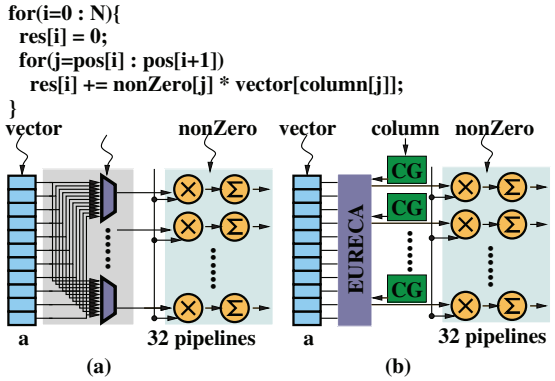

Fig. 1: An example application with dynamic data accesses, implemented with (a) conventional FPGAs, and (b) EURECA architecture.

I. Introduction

There have been many advances in FPGA technology which make them effective for hardware acceleration. However, current FPGAs have not been able to provide resource-efficient acceleration for dynamic operations: operations with execution states known only at runtime. For applications such as stencil computation [1], execution state in each clock cycle is known at design time, and thus the corresponding circuits can be optimized. For applications with dynamic operations, such as Memcached [2], sparse matrix [3], and large-scale sorting [4], an operation has multiple possible execution states every cycle. Designers need to implement extra resources to support all possible execution states. For example, in Figure 1(a), the accessed data position depends on runtime data $column[j]$, and the 32 data-paths read memory data in parallel. With each memory port connected to a specific memory region, a single data-path may connect to any of the memory ports during runtime. Therefore, the 32 data-paths require a crossbar with 1024-to-1024 connections (we assume the design uses 32-bit data). This medium-scale example cannot be routed in a Virtex-6 SX475T FPGA, while recent sparse matrix designs [5] contain 128 parallel data-paths.

The EURECA architecture [6] adopts a new reconfiguration flow that generates configuration on-chip and reconfigures connection network cycle by cycle to support all possible runtime connections with linear area complexity. As shown in Figure 1(b), a EURECA module is inserted between data-paths and memory ports. A Configuration Generator (CG), implemented in user logic, generates configurations based on runtime variable $column[j]$. In each cycle, the generated runtime configurations update the connection network based on the memory port each data-path is accessing. This significantly reduces the resource usage to support parallel dynamic data accesses.

The EURECA architecture uses conventional routing fabrics to connect CG outputs to module configuration input, and implements CGs with user logic. However, the solution shown in Figure 1(b) is only required for the most complex runtime scenarios: the data accesses from parallel data-paths are independent of each other, and each address and data port needs separate configurations. Using the same strategy to support all applications with dynamic data-paths will bring large overhead in resource usage and routing, since more configurations need to be connected and routed to module input. This work examines possible reconfiguration scenarios and proposes novel reconfiguration features for each runtime scenario, to address the shortcomings of the EURECA architecture. The contributions include:

- Categorised runtime data access scenarios and novel runtime reconfiguration strategies for each of the scenarios, see Section III
- With the new strategies, architecture design space exploration to define an optimized cycle-reconfigurable architecture, see Section IV.
- With the optimized architecture, circuits and implementation details for the cycle-reconfigurable modules to support the new strategies, with a prototype 6.44 mm x 7.8 mm chip developed in the SMIC 130-nm technology, see

Section V.

- Three benchmark applications targeting the prototype chip, showing large improvements compared with applications mapped to FPGAs and EURECA architectures, see Section VI.

II. Related Work

Support for communication operations has been explored in reconfigurable architectures. Coarse-grained architectures such as Matrix [7], Tilera [8] and Ambric [9] implement distributed general-purpose processors and dedicated communication networks on-chip. General-purpose processors can support dynamic data accesses, enabled by local caches and global communication network. To coordinate multiple processors, hardware designs executed on these architectures need new programming models, and often cannot exploit fine-grained parallelism in applications. For existing FPGA architectures, previous work proposes memory abstractions such as CoRAM [10], and optimization tools such as polyhedral models [11], to improve data access efficiency. However, these approaches are limited by the underlying hardware architectures, and cannot efficiently support applications with dynamic data accesses, such as the motivating example in Section I. In contrast, this work enhances existing FPGA architectures with cycle-reconfigurable modules, providing efficient support for dynamic accesses while preserving fine-grained parallelism in existing reconfigurable designs.

Since not all possible connections of dynamic accesses are active at the same time, reconfigurable designs can use runtime reconfiguration to only implement the active connections. In [12], partial reconfiguration is applied to update a wide crossbar by reusing the routing multiplexers. It takes 220 μs to reconfigure a crossbar running at 150MHz. As discussed in the motivating example, dynamic data accesses often require reconfiguration within each clock cycle, so the 220 μs reconfiguration time reduces the effective clock frequency to 4.5 KHz. DPGA [13] and time-multiplexed FPGAs [14] store multiple configuration sets on-chip to reduce reconfiguration time to a single clock cycle, at the expense of replicating on-chip configuration memories. The 3D programmable architecture from Tabula [15] replicates the configuration of logic blocks as well as interconnect. The replicated configuration memories, however, introduce large area and power overhead. The EURECA architecture [6] generates configurations on-chip, demonstrating the potential to efficiently support dynamic data accesses with small area overhead. However, the experiments are based on simulation, and on-chip configuration requires a large amount of user logic and routing fabric to process the generated configuration cycle by cycle. In this work, we propose new reconfiguration strategies to remove these limitations, and derive optimized architecture organisation with a prototype cycle-reconfigurable chip.

III. Reconfiguration Strategies

In this section, we categorise data access operations based on the changes in accessed locations during runtime, and propose the corresponding reconfiguration strategies for each access pattern.

A. Data Access Patterns

Data access patterns reflect the regularity of data access operations. We express a data access operation as a mapping from loop indices to memory locations. For the example application in Figure 1, the vector and matrix data access operations can be expressed as:

$$loop\{i,j\} -> accessSet_{vector}\{f(j)\} \quad (1)$$

$$loop\{i,j\} -> accessSet_{nonZero}\{j\} \quad (2)$$

Implemented in hardware, loop indices indicate clock cycles, and the corresponding data access set contains the accessed data location in each clock cycle. Within the same data access set, the distance between two consecutive data access operations indicates the stride value of this access set. In this work, as shown in Figure 2, we use the stride value to divide data access operations into four categories:

- Static: accesses with fixed strides.
- Dynamic size: linear accesses with variable vector size.
- Dynamic offset: vector access with dynamic offsets.
- Random: each access with a dynamic offset.

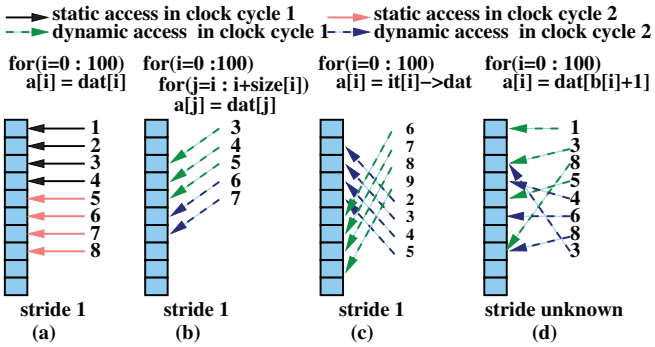

Fig. 2: (a) static-access, (b) dynamic-size, (c) dynamic-offset, and (d) random-access patterns.

Challenge 1: variable vector size. In each clock cycle, the increase in accessed locations depends on runtime variables. Figure 2(b) shows an example with dynamic-size patterns. While data are accessed in streaming manner, due to the non-deterministic data size, the mapping between memory ports and data-paths becomes dynamic. During runtime, the challenges include: (1) Data re-alignment. Runtime connections need to be adapted to the dynamic mapping between memory ports and data-paths. For the example in Figure 2(b) and Figure 3(1), the data mapping changes in the second clock cycle. (2) Data management. Accessing data with dynamic mapping and non-deterministic size requires generating proper memory control signals.

Challenge 2: dynamic offset. The accessed vector data have an internal stride value of 1, with the starting address changing randomly. Dynamic pointers typically have dynamic-offset patterns, as shown in Figure 2(c) and Figure 3(2). Implemented in hardware, the dynamic data accesses often span multiple

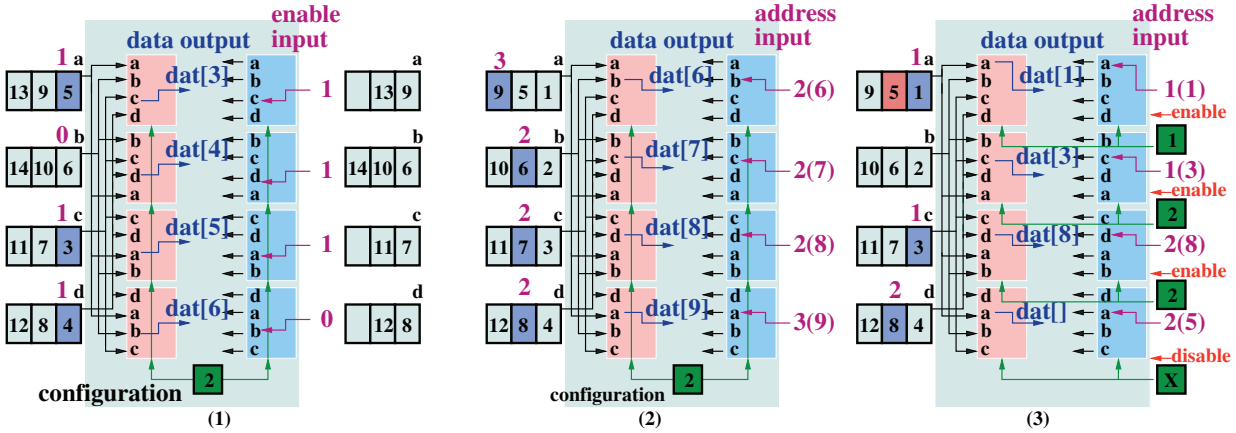

Fig. 3: A CRM operates as (1) dynamic FIFOs, (2) dynamic caches, and (3) dynamic shared memories, in correspondence to data access patterns (b), (c), (d) in Figure 2, respectively.

columns. Besides data re-alignment, the challenge to efficiently support dynamic-offset patterns includes the mapping of input address data.

Challenge 3: random accesses. In a data access set, each data access operation depends on different runtime variables. Therefore, the stride value of the data access set remains unknown, as shown in Figure 2(d). For the dynamic-size and dynamic-offset patterns, while runtime reconfiguration is required to map the dynamic connections, the fixed stride values ensure the parallel data access operations will not conflict during runtime. However, for the random-access patterns, there are possibilities that multiple data access operations point to the same memory port with different addresses. This leads to invalid hardware configurations during runtime. Supporting the random-access patterns requires (1) providing independent configurations for each access operations, and (2) resolving runtime data access conflicts.

B. Runtime Reconfiguration Strategies

A Cycle-Reconfigurable Module (CRM) supports the dynamic data access scenarios with optimized reconfiguration strategies. Previously, EURECA architecture relies on user logic to support all different data access patterns. This increase the design resource usage. Furthermore, since a large number of configurations need to be updated during runtime, the routing overhead to connect EURECA's CG outputs and configuration input of CRMs limits design scalability. In this work, we enhance the CRMs to support a new runtime reconfiguration strategy for each of the challenges listed above. This enables applications to efficiently support dynamic data accesses while minimizing resource usage and routing complexity.

Dynamic FIFOs support dynamic-size patterns. As shown in Figure 3(1), memory blocks grouped with CRMs are configured as FIFOs. To address **Challenge 1**, we add two features in the CRM. (1) Reconfigurable connections inside the CRM are pre-aligned such that vector access with fixed stride value can be supported with the same runtime configuration, instead of preparing configurations for each memory ports. (2) Connections for FIFO enable signals are also runtime reconfigurable, such that variable size of data can be fetched from FIFOs every clock cycle.

We use the example in Figure 2(b) and 3(1) to demonstrate the use case of *dynamic FIFOs*. For the example problem, in the first cycle, data-paths read ($dat[3]$, $dat[4]$, $dat[5]$), which start from the third FIFO memory ports. As the module internal connections are pre-aligned, the runtime reconfiguration value 2 is applied to all the reconfigurable connections. The reconfigured connections resolve the data re-alignment issue discussed above. In this example, ($dat[3]$, $dat[4]$, $dat[5]$, $dat[6]$) appear at the CRM output ports, while FIFO outputs are ($dat[5]$, $dat[6]$, $dat[3]$, $dat[4]$). The enable signals share the same reconfiguration. As shown in Figure 3(1), data-paths set the enable signals to be (1, 1, 1, 0) as three data are read from FIFOs. The same offset applies to the enable signals since the read starts from the third memory ports. After reconfiguration, the enable signals are properly mapped into the FIFOs.

Dynamic caches support dynamic-offset patterns. The grouped memory blocks are configured as a shared memory architecture, where each port accesses a certain region in memory space. For dynamic-offset patterns, both starting address and accessed data size could vary from cycle to cycle. As shown in Figure 3(2), the replicated data-paths access four data starting from $dat[6]$. The accessed data span the second and the third columns of the shared memory. To address **Challenge 2**, (1) the accessed data can use the same strategy to be re-aligned. (2) To support more flexible data access, address inputs need to be connected to the grouped memory blocks, with each memory port connecting to one address input. While runtime configurations are calculated as the modulo of address offset, the address input is calculated as the depth in each memory region. For example, the address for the first data $dat[6]$ is 2 ($6/4$). As shown in Figure 3(2), the CRM pre-aligns the address connections similar to the data connections, and share the same configuration at each cycle. Due to the one-cycle delay between address input and data output, the configuration is buffered internally to provide one-cycle delay in connection

reconfiguration. In this work, we limit the number of supported memory ports to be powers of 2, such that the modulo and division operations can be implemented as shifting operations.

Dynamic shared memories support random-access patterns. With unknown stride value, each data access operation in a data access set is independent, and needs to be supported separately. To be consistent with dynamic FIFOs and dynamic caches, the shared memories adapt pre-aligned connections. Figure 3(3) shows the arrangement of data and address connections for a shared memory. To address **Challenge 3**, we add two module features. (1) Inside a CRM, we add control units to enable configurations to be distributed from the same configuration input, or updated in parallel from independent configuration inputs. (2) Enable signals are added for address connection blocks to prevent data access conflicts, which occur when two or more data-paths try to access the same memory port. As shown in Figure 3(3), the first and the forth data-paths try to access the first memory port within the same cycle, with address 1 and 5 respectively. An access scheduler is implemented in user logic to decide which address connection will be enabled, and buffer the address inputs that are not enabled. We do not harden this scheduler in the CRM so that various scheduling strategies can be customized based on application requirements.

IV. Architecture Exploration

In this section, we explore the design space of a CRM, including network implementation and memory group size, where memory group size M defines the number of BRAMs coupled with a CRM. For the examples in Figure 3, $M = 4$ and each BRAM has one data output port. In this work, we use dual-port BRAMs. The explored results are verified with application performance in the following section.

Connection network in a CRM includes runtime reconfigurable connections for data, address, and enable signals. In hardware, this can be implemented with either multiplexers or permutation network [16]. While multiplexers enable simple reconfiguration strategies, permutation network leads to smaller CRM area. Given a CRM with memory group size M, we model the impact on CRM area, additional module pins to input configurations, and CG complexity in Table I.

Multiplexer-based connections have $O(M^2)$ area complexity, since the underlying multiplexers provide all possible connections in a single step. As a consequence, the runtime connections are highly regular, and can share runtime configurations extensively as discussed above. This leads to $O(Mlog_2M)$ complexity for the required additional configuration input pins and CG logic. In Table I, $log_2 8M$ indicates the maximum number of configuration bits required to define one runtime connection.

Permutation network is an all-to-all network with multiple layers. Each layer only needs to cover parts of possible runtime connections. Given $8M$-to-$8M$ connections, a permutation network requires $2log_2 8M - 1$ layers, with each layer containing $4M$ 2-to-2 selection units. This reduces area complexity to $O(Mlog_2M)$. However, such multi-layer network makes it difficult to share configurations among the selection units. In order to generate runtime connections within a clock cycle, we consider $8M$-to-$8M$ connections as logic input, and configurations to each selection unit as logic output. This leads to a CG complexity of $O(M^3)$. Figure 4 shows CRM properties as memory group size increases. For a CRM based on permutation network, its number of configuration pins and CG area rapidly increase with M.

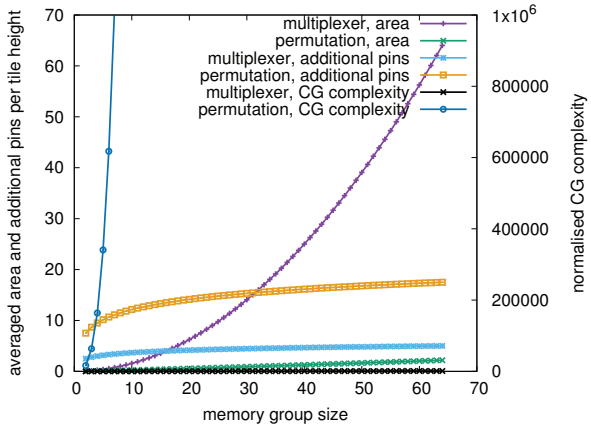

Fig. 4: CRM properties as group size increases.

Memory group size determines the scale of a CRM and the number of CRMs on chip, given a number of available BRAMs. As the memory group size M increases, the CRM can support higher data access parallelism, at the expense of increasing CRM area and CG complexity. We define architecture efficiency R_{arc} as the product of application area reduction and chip area overhead.

$$R_{arc} = \frac{Static}{CRM + CG} \cdot \frac{Area_{org}}{Area_{rec}} \tag{3}$$

where area reduction $\frac{Static}{CRM+CG}$ refers to the ratio between area usage of statically implementing dynamic accesses in user logic and using CRMs, and $\frac{Area_{org}}{Area_{rec}}$ accounts for the chip area overhead for integrating CRMs. We measure $Static$ with Verilog designs describing all possible runtime connections, and calculate CRM as the number of CLBs that consume the same layout area. The CRM area is calculated based on measured layout area (discussed in Section V) and area models in Table I. Similarly, CG is calculated with measured CG resource usage and area models in Table I.

The architecture efficiency, as shown in Figure 5, reaches a maximum for memory group size 32. While area reduction for dynamic accesses improves as group size increases, the area overhead starts to have a large impact on static designs that do not use CRMs. CRMs based on permutation network cannot efficiently support dynamic accesses due to the large CG complexity. Given sufficient on-chip resources, a cycle-reconfigurable architecture contains multiple CRMs to support (a) dynamic accesses to different arrays, and (b) dynamic accesses that require more than 64 memory ports. These CRMs can be connected to construct larger cycle-reconfigurable memory architectures.

TABLE I: Modelled CRM properties with memory group size M. CG stands for Configuration Generator.

Topology	area[1]			configuration pins[2]	CG complexity
	runtime connection	configuration memory	controller		
Multiplexer	$2(8M \cdot 8 \cdot (8M-1))$	$7(log_2 8M \cdot 2M)$	$28(log_2 8M \cdot 2M)$	$log_2 8M \cdot 2M + 2M$	$2M \cdot log_2 4M$
Permutation	$4((2log_2 8M-1) \cdot 4M)$	$7((2log_2 8M-1) \cdot 4M)$	$22((2log_2 8M-1) \cdot 4M)$	$(2log_2 8M-1) \cdot 4M + 2M$	$(8M)^3 \cdot (log_2 8M - 1/2)$

[1] We represent area with transistor count; a 2-to-1 multiplexer, an SRAM cell, and a register respectively consume 2, 7 and 22 transistors. A multiplexer-based CRM needs additional 2-to-1 multiplexers to share configurations.
[2] Configuration pins indicate the number of additional pins required to update connections cycle by cycle.

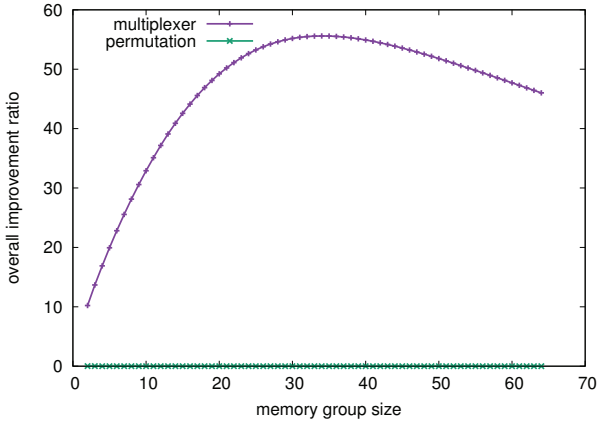

Fig. 5: Architecture efficiency R_{arc} as group size increases.

V. Implementation and Prototype

A CRM contains three major components: connection network, configuration storage, and operation control. Given the optimal architecture suggested by architecture exploration, we present implementation details to support new strategies, describe the prototype chip, and discuss the experience in implementing the chip layout.

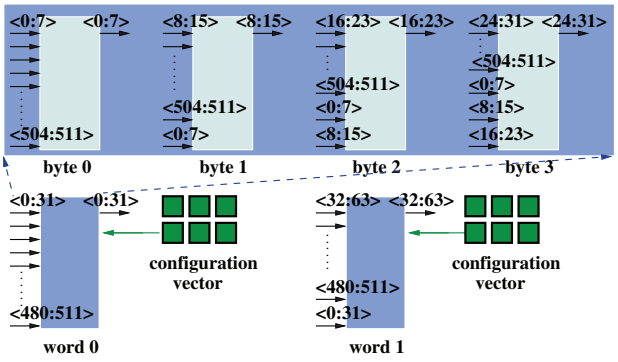

Fig. 6: Byte-level and word-level connection network organisation.

Connection network integrates pre-aligned byte and word connections to enable configurations to be better shared. As shown in Figure 6, a word connection contains 4 byte connections, and each byte connection contains 8 N-to-1 multiplexers, where N is the number of bytes in the incoming wires. For the example CRM with 512 input wires, $N = 64$. To handle dynamic accesses with stride value 1, every two consecutive byte connections have a 1-byte offset, and thus two consecutive word connections have a 1-word offset. In Figure 6, the input wires for $byte1$ are aligned as $< 8:511, 0:7 >$. Therefore, dynamic accesses with fixed stride value can be supported with the same runtime configuration. For the examples in Figure 3(1) and (2), memory ports (a, b, c, d) share the same configuration value 2 to support dynamic data accesses starting from the second memory port.

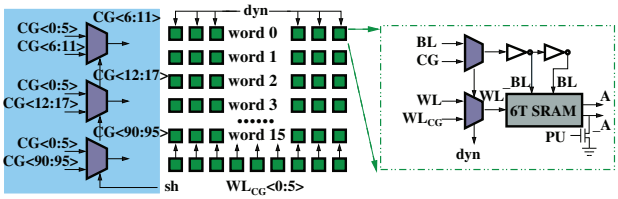

Fig. 7: Array of SRAM cells. sh determines whether a single configuration is shared in CRM, dyn select SRAM inputs from CGs or initialisation ports, and WL_{CG} control the write enable signals of SRAM cells.

Configuration storage and operation mode control enable CRMs to support different reconfiguration strategies. Figure 7 briefly shows the internal organisations of the SRAM cells and associated control units. Compared with EURECA [6], our enhanced design has three features: module-level configuration sharing, data-address synchronisation, and improved connection parallelism. (1) With the pre-aligned connection, for *dynamic FIFOs* and *dynamic caches*, the whole CRM can share a single reconfiguration to re-align data read with non-deterministic starting addresses. The share selection signal sh determines whether the configuration inputs for different word connections are from the same ports or updated in parallel (to support *dynamic shared memories*). (2) When grouped memory blocks are not configured as FIFOs, it takes an extra cycle for output data to appear at data ports. If configured in the same cycle as address connections, data connections are updated before accessed data appear at output ports. Inside a CRM, configuration input for data connections can be selected from direct input or registered input. The registered input is used to cooperate with the 1-cycle delay. We discuss in more detail in the execution timing part. (3) Given a memory group size M with $2M$ 32-bit data ports, a CRM provides up to $8M$ different input connections, with each 32-bit port considered as four 1-byte ports. This provides high data parallelism for applications involving byte-level data accesses, and enables offset address to start at any of the $8M$ ports. Previously, such parallelism can only be achieved with off-chip data, and a memory group provides up to $2M$ connections, with each BRAM port configured to be 1-byte in width. Furthermore, with the pre-aligned connections, the improved parallelism

does not require additional configuration inputs.

Execution timing of a cycle-reconfigurable design includes configuration generation, circuit reconfiguration, and data processing. Figure 8 shows the timing diagram of a cycle-reconfigurable design. At the beginning of a clock cycle, CGs generate circuit configurations based on runtime variables, and reconfigure the connections in CRMs. Once connections are updated, data appearing at the memory ports of *dynamic FIFOs* are read are read into data-paths. For *dynamic caches* and *dynamic shared memories*, data-paths first input addresses, and wait for data appear at memory ports. The data configuration outputs are registered to align with the additional one cycle delay. BRAMs connected to a CRMs can be configured to load address at the rising edge or the falling edge of a clock. In cases that one cycle delay is required (e.g. the CG inputs depend on loaded data), we configure BRAMs to load addresses at falling edges, as shown in Figure 8.

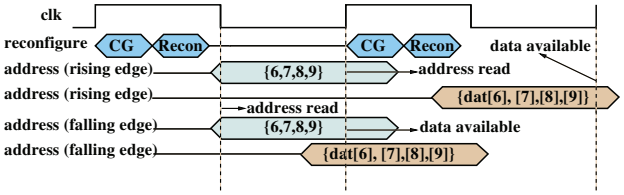

Fig. 8: Timing diagram of the runtime execution of a cycle-reconfigurable design.

Prototype architecture. To evaluate the proposed features, we develop a prototype FPGA integrated with a CRM, in the SMIC 130-nm technology. Bounded by the tap-out budget, the prototype chip array size is 17 x 32, with 17 and 32 respectively indicating 17 columns of on-chip resources, with each column containing 32 CLBs Figure 9 shows the chip layout, and Table II summarises the chip properties. Given the 32 column height and the BRAM height of 4, the group size is bounded to be $M = 8$ with 16 32-bit output ports. With the supported features and basic circuits integrated, the area usage of a CRM mainly comes from the horizontal long wires that connect output ports to connection network. As M increases, the CRM area is proportional to M^2 since both the number and the length of horizontal wires double as M doubles. This aligns with the area model in Table I. For $M = 8$, the module width is 300 μm. In contrast, the width of a CLB is 250 μm, and the width of a BRAM is 680 μm. In other words, the CRM takes the same area as 38.4 CLBs (4.7% of the prototype chip area). The measured delay for connection reconfiguration is 1.15 ns, which is small compared with the 20 ns to 50 ns chip clock period. The prototype chip is under the tape-out process, and the circuit delay properties used in the next section to calculate critical-path delay are based on simulation results from Cadence.

VI. Case Studies

Experiment Methodology. We study the benefits of the proposed architecture with three applications. We prepare three designs and three corresponding architecture files for each application. We present the application performance in Table III, when mapped into the prototype chip with $M = 8$. Static designs (`static` in Table III), with dynamic connections statically implemented in user logic, maps to baseline FPGA architectures. EURECA designs (`eureca` Table III) and dynamic designs (`dynamic` Table III) refer to cycle-reconfigurable designs based on the EURECA architecture [6] and the proposed architecture. The synthesis tool uses Design Compiler (DC) for circuit synthesis, ABC [17] for mapping, a graph matching algorithm for packing, simulated annealing algorithm for placement, and path-finder [18] for routing. Architecture files are modified to recognise a CRM as a hard block connected to a BRAM column. We map the applications into the prototype chip. For Memcached, due to its relatively large application scale, the architecture array size is doubled to accommodate the design blocks.

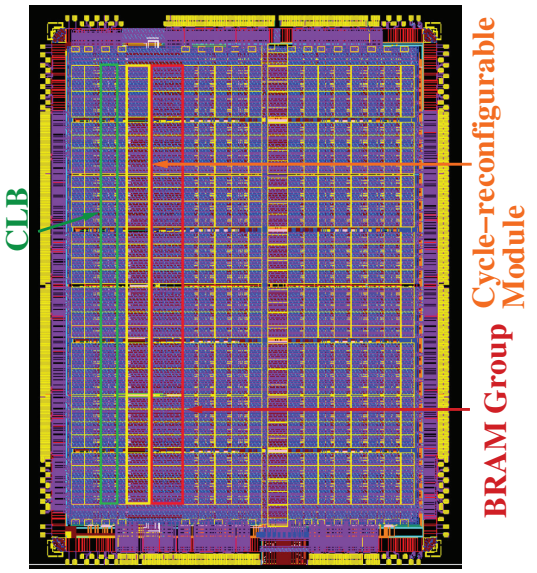

Fig. 9: Cycle-reconfigurable chip layout, with a column of CLB, CRM, and BRAM group labelled.

TABLE II: Cycle-reconfigurable architecture properties.

baseline FPGA	CLB: 384 DSP: 24	BRAM: 8 array size: 32x17
architecture area	chip width: 6.44 mm CLB width: 250 μm	chip height: 7.8 mm BRAM width: 680 μm
CRM properties	width: 300 μm group size: 8	delay: 1.15 ns additional pins: 82

Large-Scale Sorting. Sorting large-scale data sets [4] often uses sorting networks [19], [20] to sort small chunks of data, and use mergers to combine the sorted small chunks. Figure 10(a) shows a parallel merger that merges N data per iteration from sorted arrays `A` and `B`. We assume ascending order in the sorting algorithm. At each iteration, the algorithm reads the N smallest data from both arrays, and commits the N smallest data (in this example, $1, 3$ from `A` and $2, 3$ from `B`). Therefore, the starting address of read accesses in the next iteration depends on the number of committed data in the

TABLE III: Benchmark application performance.

	Large-scale Sorting			Memcached			SpMV		
	static	eureca [6]	dynamic	static	eureca [6]	dynamic	static	eureca [6]	dynamic
slices (total)	8676	1174	1054	11763	3082	2684	3549	900	876
slices (CG)	0	126	6	0	31	8	0	43	19
DSP	0	0	0	0	0	0	16	16	16
BRAM	16	16	16	8	8	8	8	8	8
CRM	0	1	1	0	1	1	0	1	1
critical-path delay (ns)	25.72	23.87	18.9	60.4	60.0	52.1	15.1	14.9	13.9
area[1]	**8.23x**	1.1x	1x	4.38x	1.15x	1x	4.05x	1.03x	1x
CG area[1]	n/a	21x	1x	n/a	3.9x	1x	n/a	2.26x	1x
area-delay product	**11.2x**	1.39x	1x	5.08x	1.32x	1x	4.4x	1.1x	1x
throughput (per cycle)	16 sorted data			64 bytes			16 partial results		

[1] We compare the resource usage with the number of used CLBs. A CLB contains 4 slices, and a CRM is considered as 38.4 CLBs based on layout area.

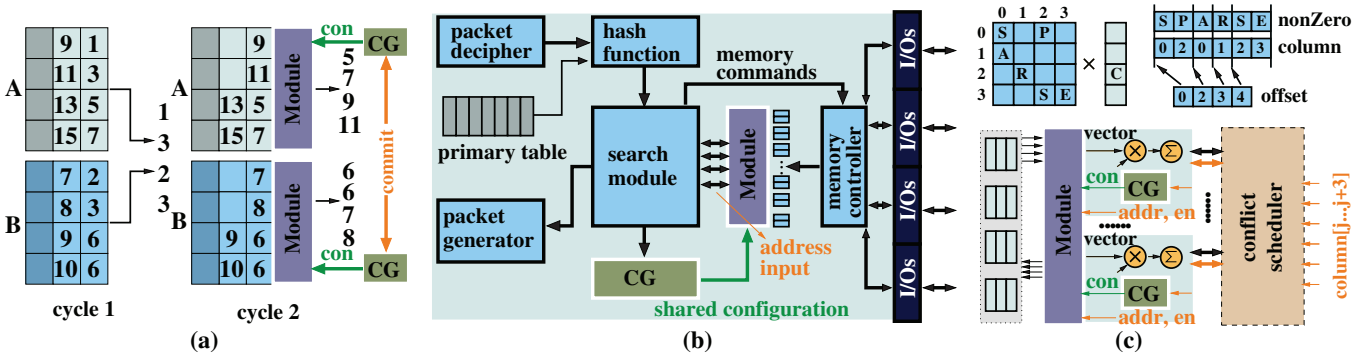

Fig. 10: Cycle-reconfigurable designs for (a) large-scale sorting (*dynamic FIFO*), (b) Memcached (*dynamic cache*), and (c) SpMV (*dynamic shared memory*).

current iteration, and changes from time to time. In previous designs [4], N is limited to 1 or 2 due to non-deterministic starting addresses.

We support dynamic accesses in large-scale sorting with two CRMs implemented as *dynamic FIFOs*. As shown in Figure 10(a), each CRM shares a single configuration for all enable inputs and data outputs. In each cycle, up to 16 data can be committed in parallel. In each clock cycle, the CG takes the number of committed data and the starting address in the current cycle to calculate the starting address for the next cycle, and generates runtime configurations based on the calculated starting address. The proposed reconfiguration strategy significantly reduces the CG complexity. As shown in Table III, the CG area is reduced by 21 times. Furthermore, since fewer configurations need to be routed to a CRM, the critical path delay is reduced. The dynamic sorting design achieves 1.39 times improvement compared with the EURECA design, and 11.2 times improvement compared with the static design, in terms of area-delay product. In practice, the *dynamic FIFOs* can efficiently support streaming database applications, especially in-memory database applications.

Memcached is a distributed memory caching system widely used in the servers of web service companies (Facebook, Twitter, YouTube, Wikipedia, etc.). A Memcached server stores frequently accessed data in memory to provide quick responses to web requests. Memcached uses hash tables to index stored data. Once receiving a new packet, Memcached hashes the key in the packet, and use the generated hash value to search for a match in stored data. As shown in Figure 10(b), the search process iterates through all linked hash entries until a match is found. In each search, the starting address depends on the hash value or the fetched next entry address, and the matching operations check the fetched key value and key length.

We support dynamic accesses in Memcached with a CRM implemented as a *dynamic cache*. As shown in Figure 10(b), a memory group buffers loaded off-chip data. As the search module loads a hash entry, the CG takes the address of the linked next hash entry to generate runtime configurations. Operating as a *dynamic cache*, the runtime data and address connections in a CRM all share the same runtime reconfiguration. For the prototype chip with $M = 8$, a Memcached design can fetch up to 64 bytes of data per clock cycle, and iterates through the search operations with dynamic pointers with the efficiency of hardware and the flexibility of memory management. The dynamic design reduces the application area-delay product by 1.32 and 5.08 times, compared with EURECA and static designs. Besides Memcached, applications that use hash tables can benefit from the proposed architecture. For example, a Gzip design [21] loads hash table data in parallel to compress more than one datum per clock cycle, with dynamic connections between hash tables and parallel data-paths statically implemented. Applying the *dynamic cache*

will significantly reduce design area.

Sparse Matrix-Vector Multiplication. Sparse Matrix-Vector multiplication (SpMV) is widely used in scientific computing and industrial development. SpMV multiplies a sparse matrix with a dense vector. In this work, we store the non-zeros of the sparse matrix in Compressed Sparse Row (CSR) format. As shown in Figure 10(c), the CSR data contain three vectors: non-zeros `nonZero`, position of non-zeros `column`, and vector data `vector`. In the multiplication process, SpMV multiplies `nonZero[j]` with corresponding `vector[column[j]]`. To avoid the M^2 area complexity discussed in the motivating example, conventional SpMV architectures [3] replicate vector memories. This limits the size of vector data can be stored on-chip, and leads to idles cycles for matrix with low sparsity. In [3], the idle cycles reduce the average efficiency to 42%.

We support dynamic accesses in SpMV with a CRM implemented as a *dynamic shared memory*. As shown in Figure 10(c), we implement a conflict scheduler to resolve data accesses that point to the same memory port at the same clock cycle. The scheduler buffers conflicted data accesses, and enables the access with pre-defined priority order. Each data-path has a separate CG to generate configurations for the address input and data output of a memory port, based on vector data address `column[j]`. The access conflict rate, simulated with 10 sparse matrices from [22], reduces to 15% when $M = 32$. As N increases, the memory conflict ratio will further decrease. Inside a CRM, the configurations for data connections are buffered to align the 1-cycle delay between address input and data output. For *dynamic shared memory*, since configurations cannot be shared, the resource saving compared with EURECA designs comes from shared configurations between data and address connections, and simplified enable signal distribution. As shown in Table III, the dynamic design reduces area-delay product by 1.1 to 4.4 times, compared with EURECA and static designs. The *dynamic shared memory* can benefit applications with indirect data accesses, such as graph problems.

VII. Discussion and Conclusion

This work presents the first prototype chip for cycle-reconfigurable architectures that generate configurations on-chip with user logic. We propose new runtime reconfiguration strategies to minimize the logic and the routing complexity to generate and apply runtime configurations on-chip, and explore the design space of a cycle-reconfigurable architecture to derive the optimal architecture organisation. Experimental results show integrating cycle-reconfigurable module brings small overhead in chip area (the area of 1.2 columns of CLBs). For applications with dynamic data accesses, the prototype chip targeting the benchmark applications achieves up to 1.4 and 11.2 times reduction in application area-delay product, compared with applications mapped to EURECA architectures and baseline FPGA architectures respectively. Current and future work includes developing tools to automatically detect categorised data access scenarios and generate optimized designs, and exploring cycle-reconfigurable on-chip network to connect CRMs.

VIII. Acknowledgement

The support of EPSRC grant EP/I012036/1, EP/N031768/1, EP/K011715/1, EP/L00058X/1, EP/K034413/1, the European Union Horizon 2020 Programme under Grant Agreement Number 671653, and the National Natural Science Foundation of China 61131001 is gratefully acknowledged.

References

[1] X. Niu *et al.*, "Exploiting run-time reconfiguration in stencil computation," in *FPL*, 2012, pp. 173–180.

[2] S. R. Chalamalasetti *et al.*, "An FPGA memcached appliance," in *FPGA*, 2013, pp. 245–254.

[3] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *FPGA*, 2005, pp. 63–74.

[4] D. Koch and J. Torresen, "FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *FPGA*, 2011, pp. 45–54.

[5] G. C. Chow *et al.*, "An efficient sparse conjugate gradient solver using a benes permutation network," in *FPL*, 2014, pp. 151–160.

[6] X. Niu, W. Luk, and Y. Wang, "EURECA: on-chip configuration generation for effective dynamic data access," in *FPGA*, 2015, pp. 74–83.

[7] E. Mirsky and A. DeHon, "Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *FCCM*, 1996, pp. 157–166.

[8] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the tile processor," *IEEE Micro 27(5):15-31*.

[9] M. Butts *et al.*, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, 2007, pp. 55–64.

[10] E. S. Chung *et al.*, "CoRAM: an in-fabric memory architecture for FPGA-based computing," in *FPGA*, 2011, pp. 97–106.

[11] L.-N. Pouchet *et al.*, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA*, 2013, pp. 29–38.

[12] S. Young *et al.*, "A high I/O reconfigurable crossbar switch," in *FCCM*, 2003, pp. 3–10.

[13] E. Tau *et al.*, "A first generation DPGA implementation," in *FPD*, 1995, pp. 138–143.

[14] S. Trimberger *et al.*, "A time-multiplexed FPGA," in *FCCM*, 1997, pp. 22–29.

[15] Tabula, "Tabula corporate backgrounder," http://www.tabula.com/about/Tabula_CorpBackgrounder4_13.pdf.

[16] A. Waksman, "A permutation network," *J. ACM*, vol. 15, no. 1, pp. 159–163, 1968.

[17] S. Cho *et al.*, "Efficient FPGA mapping using priority cuts," in *FPGA*, 2007.

[18] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *FPGA*, 1995, pp. 111–117.

[19] K. E. Batcher, "Sorting networks and their applications," in *AFIPS*, 1968, pp. 307–314.

[20] I. Parberry, "The pairwise sorting network," *Parallel Processing Letters*, vol. 2, pp. 205–211, 1992.

[21] J. Fowers *et al.*, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *FCCM*, 2015, pp. 52–59.

[22] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw. 38(1):1*.