EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits

Xinyu Niu*, Nicholas Ng*, Tomofumi Yuki†, Shaojun Wang‡, Nobuko Yoshida* and Wayne Luk*
*Dept. of Computing, School of Engineering, Imperial College London, UK
†INRIA / LIP / ENS Lyon, France
‡Harbin Institute of Technology, China
Email: {nx210, nickng, yoshida, w.luk}@doc.ic.ac.uk, tomofumi.yuki@inria.fr, wangsj@hit.edu.cn

***Abstract*—EURECA architectures have been proposed as an enhancement to the existing FPGAs, to enable cycle-by-cycle reconfiguration. Applications with irregular data accesses, which previously cannot be efficiently supported in hardware, can be efficiently mapped into EURECA architectures. One major challenge to apply the EURECA architectures to practical applications is the intensive design efforts required to analyse and optimise cycle-reconfigurable operations, in order to obtain accurate and high-performance results while underlying circuits reconfigure cycle by cycle. In this work, we propose compiler support for EURECA-based designs. The compiler support adopts techniques based on session types to automatically derive a runtime reconfiguration scheduler that guarantees design correctness; and a streaming circuit model to ensure high-performance circuits. Three benchmark applications —large-scale sorting, Memcached, and SpMV— developed with the proposed compiler support show up to 11.2 times (21.8 times when architecture scales) reduction in area-delay product when compared with conventional architectures, and achieve up to 39% improvements compared with manually optimised EURECA designs.**

I. Introduction

While recent progresses in FPGAs and development tools show good promise for mainstream hardware accelerators, one major limitation of FPGAs comes from the inefficient support for dynamic operations, i.e., operations with execution status only known at runtime. To handle such program operations (such as dynamic pointers), EURECA architecture [1] was proposed to support cycle-reconfigurable circuits: within a clock cycle, the architecture can modify the underlying circuits based on runtime variables. Fig. 1 shows an example EURECA-based design, which streams the matrix data `nonZero` from off-chip memory, and stores the vector data `vector` on chip. We vectorise the example design in Fig. 1: (1) 32 data-paths process 32 vector multiplication per clock cycle, and (2) the vector memory contains 32 memory ports to fetch up to 32 vector data in parallel. However, the accesses to vector data rely on runtime variable `column`, requiring all-to-all connections between the memory ports and duplicated data-paths.

```
for(i = 1 : N)
  res[i] = 0;
  for(j = row[i−1] : row[i])
    res[i] += nonZero[j] * vector[column[j]];
}
```

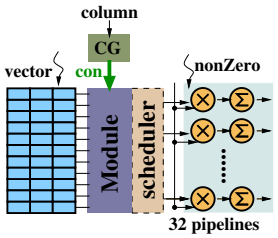

Fig. 1: A Sparse Matrix Vector Multiplication (SpMV) kernel, mapped into EURECA architecture.

A EURECA-based design, as shown in Fig. 1, applies runtime reconfiguration to resolve this challenge. Within a single clock cycle, a EURECA design takes runtime variables (e.g. `column`), calculates runtime configurations, and applies the reconfiguration to update the connections between memory ports and data-paths. This enables the example design to be supported with linear instead of quadratic (all-to-all) area complexity, and thus supports applications that are not efficiently implementable in hardware. Preliminary results [1] show enhancing conventional FPGA architecture with the cycle reconfigurability brings 1% area overhead.

In practice, given circuits reconfigure cycle by cycle, there are three major challenges in developing efficient and correct EURECA designs:

- To guarantee correct functionality while reconfiguring cycle by cycle.
- To ensure high efficiency and performance of the EURECA designs.

The objective of this work is to develop automated compiler support to address the above challenges for EURECA designs. We develop compiler support for source-to-source transformation of high-level EURECA descriptions to improve design performance and area efficiency, while preserving correctness. Our approach includes three novel aspects addressing the above three challenges:

- Conflict-free runtime reconfiguration by runtime scheduling based on session-based communication protocols [2], [3]. See Section IV.
- Streaming circuit models are used in optimising the scheduling function, to approximate the theoretical peak performance of circuits configured at each cycle. See Section V.

Section VI illustrates how the proposed compilation flow can be used in developing three applications: Memcached, large-scale sorting, and SpMV, showing up to 21.8 times improvements in area-delay product. The proposed compilation flow shows potential for automating the implementation of computational kernels with parallelisable computation and complex data accesses; such kernels cannot be efficiently supported using current high-level synthesis frameworks.

II. Related Work

Previous design automation and optimisation approaches mainly focus on conventional FPGA architectures. Chung et. al. [4] proposed CoRAM as a memory abstraction to simplify application development and to improve application portability.

Data reuse techniques exploit the locality of on-chip and off-chip data access to improve data access efficiency. The data transfer and storage exploration approach [5] analyses and groups various data access requirements. Polyhedral models have been used to analyse data dependencies between data accesses that can be reused [6], to generate addresses for off-chip data accesses [7], and to transform on-chip data accesses to better exploit data locality [8]. These approaches, while effective, rely on the conventional FPGA architectures. The concept of runtime reconfiguration and its potential benefits are missing from the proposed approaches. As an example, the polyhedral-based compiler support in [8] targets at Static Control Parts (SCoP) with static data accesses: the computational kernels known to be efficient on existing FPGAs. In this work, we introduce cycle-reconfigurable architectures into the compiler support, and transform and optimise the target applications to exploit cycle reconfigurability.

III. Approach Overview

Architecture Overview. A EURECA architecture enhances conventional FPGA architectures with cycle-reconfigurable EURECA modules. A EURECA architecture contains multiple EURECA regions, with each region containing user logic and a group of parallel memory blocks coupled with a EURECA module. In a EURECA design, the EURECA modules in EURECA regions are instantiated to provide cycle-reconfigurable connections between memory blocks and duplicated data-paths, while data-paths, Configuration Generators (CGs), and schedulers are implemented with user logic on-chip. Within a clock cycle, a EURECA design goes through four steps. (1) CGs take runtime variables and generate runtime reconfigurations. (2) A scheduler ensures the configurations to be applied will not cause data access conflicts; (3) The configurations that pass the scheduler are applied to the EURECA module; (4) Memory data are read though the reconfigured connections, and processed with connected data-paths in the same way as non-cycle reconfigurable designs.

Compilation flow. The proposed approach starts from C applications, and goes through kernel identification, reconfiguration scheduling, and circuit mapping to provide optimised EURECA designs. The design flow targets at loop kernels with irregular data accesses. (1) While irregular data accesses without access conflicts can be efficiently optimised and mapped to EURECA architectures, the data accesses with access conflicts will lead to incorrect runtime circuit configurations. We develop a novel approach based on multi-party session types to derive reconfiguration scheduler from rigorous communication protocols, which guarantees conflict-free runtime reconfiguration. (2) The identified computation kernels, the on-chip configuration generation modules, and the configuration scheduling modules are connected in streaming fashion to ensure each data-path, while being reconfigured cycle by cycle, can generate one correct result per clock cycle. Section VI presents how applications can be developed with the proposed approach.

IV. Conflict-free Scheduling

When implemented in hardware, loop with irregular data accesses is unrolled to access and process data in parallel. Fig. 2 shows two example designs for irregular data accesses with (a) fixed stride values and (b) unknown stride values. In these two designs, a EURECA module is coupled with a memory group with 4 memory blocks, and 4 data-paths access data in the memory group in parallel. We define access conflicts as the runtime states where two or more data-paths access the same memory block at the same clock cycle. For a EURECA design, this leads to more than one connections are configured to connect to the same memory port, generating incorrect results. For data accesses with fixed stride values, conflict-free EURECA designs can be developed. For the example in Fig. 2(a), the stride value is 1 (each inner loop iteration increments the accessed location by 1). There will not be conflicts as long as the number of accessed data is less than the number of parallel memory blocks (4). For the example in Fig. 2(b), the first and the last data-paths try to access memory port at the same cycle, due to unknown stride values. The approach presented in this section automatically derives a scheduler to determine which runtime connections can be established (i.e. which runtime configurations need to be applied).

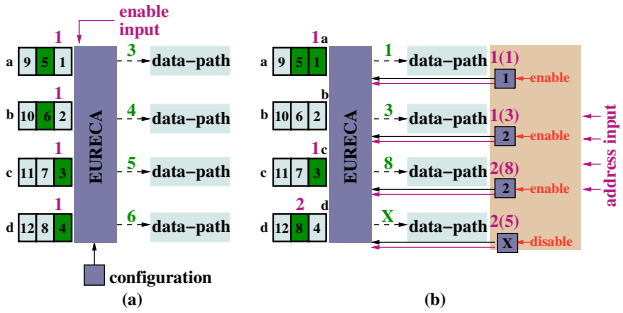

Fig. 2: EURECA designs for irregular data accesses with (a) fixed stride values and (b) unknown stride values.

Scheduling function. Our scheduler computes the connections between input data-paths and output ports during execution from the current input data. A schedule is the set of edges in a bipartite graph between data-paths and ports. Given the input data, our scheduler outputs an adjacency matrix connecting the data-paths with the ports. A sample scheduling function for a 2-data-path, 2-port configuration is given below.

```
void sched2(int idx0, int idx1, int* enabled)
{
  enabled[0] = (!(((idx0&1)^0)))
             || (!(((idx1&1)^0)) && ((idx0&1)^0));
  enabled[1] = (!(((idx0&1)^1)) && ((idx1&1)^1))
             || (!(((idx1&1)^1)));
}
```

The scheduling function takes as input `idx0` and `idx1`, the next addresses accessed by data-path 0 and 1 respectively, and outputs a boolean array `enabled`. The `enabled` array is indexed by the port number, and each element is a disjunction of enable signal from each data-path. They represent the schedule at each port, expressed as a static boolean expression computed from the dynamic values of the input data-paths `idx0` and `idx1`, and the resulting boolean value indicates whether the port should be enabled for the current set of inputs. The expression `!((idx`n`&1)^`i) is true if port i is accessed by for input data-path `idx`n. Suppose the inputs from `idx0` and `idx1` access port 0 and port 1 respectively. As both inputs are accessing different ports, there is no data access conflict, hence

both port 0 and port 1 are enabled. The resulting `enabled` array should contain `{1, 1}` as shown below.

However, if both `idx0` and `idx1` are accessing port 0, and since each port can only be used by a single data-path (and vice versa) at any instant, this indicates a data access conflict. To resolve the conflicts, we assign priorities to data-path and port pairs. In our implementation, we assign the highest priority to data-path/port pair (i, i), and lower priority for the next data-path accessing the same port $((i+1) \mod N, i)$. Port 0 is accessed by both `idx0` and `idx1`, and $(\texttt{idx0}, 0)$ has a higher priority than $(\texttt{idx1}, 0)$, hence port 0 is enabled but only for `idx0`. The resulting schedule is given below, and since only port 0 is used, the `enabled` array contains `{1, 0}`.

	port 0	(priority)	port 1	(priority)
`idx0` = 0	enabled	*1*	-	*2*
`idx1` = 0	(conflict)	*2*	-	*1*
`enabled`	1		0	

(A smaller number indicates higher priority)

Scheduler generation. The scheduling function above is automatically generated, and also produces a corresponding protocol, specified in the protocol description language Scribble [3], developed from the theory of multiparty session types [2]. We use the protocol to verify the absence of data conflicts in our scheduling function. In the protocol, we express a schedule as a message-passing protocol for control messages sent between data-paths and ports, modelled as a coherent global view of all interactions (we call this global protocol). There are two types of controlling messages, *enable* and *disable*. A data-path and a port are connected if an *enable* message is sent between them. If multiple *enable* messages are received by a port then there is a scheduling error (conflict). The priorities of data-path-port pairs are represented in the protocol, where pairs with lower priorities are nested deeper in the nested branching decisions.

Through modelling the protocol in Scribble, we guarantee that (1) no control message is mismatched (e.g. data-path to a port which is not to be connected); (2) the lack of data conflicts in the scheduling algorithm, by leveraging the well-formedness property of Scribble protocols, where messages sent between two participants in all well-formed Scribble protocols must be unambiguous; and (3) the resulting valid protocol corresponds to the scheduler code through transforming the global protocol to endpoint protocols which are localised views of the protocol, and each endpoint protocol describes the interactions at its endpoint. The endpoint protocols can be used to examine the control messages receivable by the endpoints which correspond to the expressions in the `enabled` array in the `sched2` function in the previous subsection.

V. Circuit Model

Cycle-reconfigurable connections. EURECA designs make use of streaming models to approximate the theoretical peak performance at which each duplicated data-path generates one result per clock cycle. For EURECA designs without access conflicts, we focus on runtime reconfigurable connections between data-paths and memory ports. We use large-scale sorting as an example application, where two sorted arrays are merged in parallel into a larger sorted array. The algorithm detail can be found in [1]. As illustrated in Fig. 3(a), a

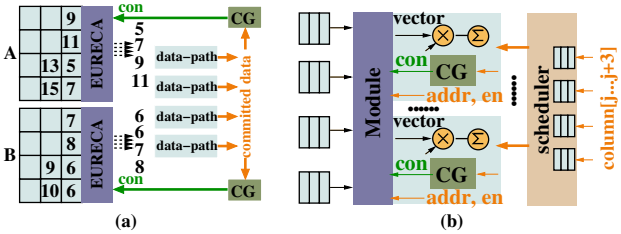

Fig. 3: EURECA designs for (a) large-scale sorting (irregular accesses with fixed stride values) and (b) SpMV (fully irregular). CG stands for Configuration Generator.

EURECA design stores the two sorted arrays in memory blocks configured as parallel FIFOs, which couple with EURECA modules. Similar to the example in Fig. 2(a), the irregular data accesses change the starting addresses every cycle, and require runtime reconfiguration to enable correct processing in connected data-paths.

In order to integrate runtime reconfiguration into the streaming architectures, we use memory blocks coupled with EURECA modules to implement the FIFOs in this example, and implement CGs with user logic. As presented in Listing **??**, the runtime variables that determine the runtime connections (i.e. starting address) in the next cycle are `commit` and the starting address in the current cycle. The CG takes these runtime variables and generates the configuration: it first computes the starting address in the next cycle, and then generates the configuration to implement that connection. As discussed in [1], the EURECA module is designed to simplify the configuration generation logic. For data accesses with fixed stride, each data access shares the same access offset, and thus can share the same runtime configuration. The update of runtime configuration can happen at either the falling edge of the current clock cycle or the rising edge of the next clock cycle, to ensure reconfiguration, data access, and data processing can be finished within the same cycle.

Configuration scheduling. The scheduler described in Section IV takes access indices as input, and output enable signals for runtime reconfiguration operations. In hardware, the expressions to compute the enable signals are mapped to a scheduler. For streaming architectures integrated with runtime reconfigurable connections, the scheduler connects to enable signals for each duplicated data-path and CG; for irregular accesses with unknown stride values, each runtime connection has a CG since the configuration cannot be shared. When access conflicts occur, the data-paths and CGs with lower priority are disabled in the current cycle, to prevent incorrect functionality.

In this work, we use SpMV as an example for the application with access conflicts. The scheduler module takes indices of the irregular data access operations (`column`) as input, and outputs enable signals to replicated data-paths. As shown in Fig. 3(b), `column` are buffered in the scheduler module. At each cycle, the scheduler module reads in accessed location for each data-path. Based on the assigned priority, access operations without conflicts and with the highest priority for each memory port are passed to the data-path and configuration generator. These passed access operations (i.e. `column` values) enable proper runtime reconfigurations to be generated and applied. Moreover, these signals disable data-paths that do not have access operations at the current cycle.

TABLE I: Benchmark application performance.

	Large-scale Sorting			Memcached			SpMV		
	static	initial [1]	dynamic	static	initial [1]	dynamic	static	initial [1]	dynamic
slices (total)	8676	1174	1054	11763	3082	2684	3549	900	876
DSP	0	0	0	0	0	0	16	16	16
BRAM	16	16	16	8	8	8	8	8	8
EURECA module	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
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 slices (both logic slices and EURECA modules are included). A CLB contains 4 slices, and a EURECA module consumes the same area as 154 slices.

VI. Results

This section presents the performance of EURECA designs developed with the proposed compiler support. To evaluate the application performance, we develop reference designs that implement the benchmark applications without EURECA support. In the reference designs, the all-to-all connections between duplicated data-paths and memory ports are statically implemented with `if-else` expressions in Verilog HDL, and mapped into user logic.

Experiment methodology. The experiment is based on a prototype EURECA architecture layout developed with the SMIC 130-nm technology, and a synthesis flow adapted from VTR [9]. The prototype architecture is a 10.9 mm x 7.8 mm chip with one EURECA region, which contains 704 CLBs, 32 BRAMs in 4 columns, 16 DSPs in 2 columns, and 1 EURECA module. The EURECA module is coupled with 8 dual-port BRAMs. Besides the large-scale sorting and SpMV discussed in Section V, we develop Memcached with the proposed approach. Details for Memcached can be found in [1].

The inputs to the proposed compilation flow include a C program for the target application, as well as a EURECA architecture file. The architecture file, captured in XML, contains hardware details of EURECA modules in the target architecture. EURECA design parallelism (the number of duplicated data-paths in a EURECA design) is determined by the number of parallel memory ports a EURECA module can support. The design parallelism is fed into (1) scheduler models to resolve access conflicts, (2) and a back-end compiler (such as LLVM) to unroll the inner loop identified by polyhedral analysis. The transformed C program contains identified inner loops with irregular data accesses, and scheduler function to handle access conflicts. EURECA-based Verilog modules are developed based on the transformed C program.

Application performance. We present the application performance in Table I. For each application, we compare the performance of three designs. In the table, `static` indicates static designs without cycle-reconfigurable modules, `initial` indicates the manually developed cycle-reconfigurable designs in [1], and `proposed` indicates cycle-reconfigurable designs developed with the proposed compiler support, with correctness guaranteed. Compared with static designs, the dynamic designs reduce overall design area by up to 8.2 times and reduce area-delay product by up to 11.2 times. Large-scale sorting achieves the largest resource saving mainly due to it has relatively simple data-paths, and thus the reduction in connection resource usage leads to larger area saving.

The performance of the EURECA designs developed with the proposed approach, as shown in Table I, is slightly better than that of the manually developed EURECA designs. In the `initial` designs, runtime configurations are generated for each memory port, while with categorised data accesses, the irregular data accesses with fixed stride can share the same runtime reconfiguration. Besides assuring correctness, this method reduces the area of CGs implemented in user logic, and leads to up to 39% reduction in area-delay product.

VII. Conclusion and Discussion

This work proposes compiler support for EURECA-based designs, to automate design process to identify, categorise, and schedule runtime reconfiguration operations. Experimental results show large improvement in applications can be achieved with correct functionality guaranteed. This work demonstrates the potential for a high-level synthesis tool that efficiently transforms complex high-level programs into low-level parallel streaming circuits. A complete compiler integrated with the proposed approach would (1) compile a wide range of applications with and without irregular data accesses; (2) generate optimised cycle-reconfigurable designs when targeting EURECA architectures; (3) generate static designs with constraints when targeting general FPGA architectures. Future work includes polyhedral analysis to support automatic detection of irregular data accesses, global scheduling for design with multiple EURECA regions, and development of a streaming-based compiler tool chain.

Acknowledgement The support of EPSRC grant EP/I012036/1, EP/K011715/1, and the European Union Horizon 2020 Programme under Grant Agreement Number 671653 is gratefully acknowledged.

References

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

[2] K. Honda, N. Yoshida, and M. Carbone, "Multiparty Asynchronous Session Types," *JACM*, vol. 63, pp. 1–67, 2016.

[3] N. Yoshida, R. Hu, R. Neykova, and N. Ng, "The Scribble Protocol Language," in *TGC 2013*, ser. LNCS, vol. 8358. Springer, 2013, pp. 22–41.

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

[5] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. J. Omnés, *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer, 2002.

[6] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Trans. Design Autom. Electr. Syst.*, vol. 12, no. 2, 2007.

[7] S. Bayliss and G. A. Constantinides, "Optimizing SDRAM bandwidth for custom FPGA loop accelerators," in *Proceedings of the ACM/SIGDA 20th International Symposium on Field Programmable Gate Arrays, FPGA 2012, Monterey, California, USA, February 22-24, 2012*, 2012, pp. 195–204.

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

[9] J. Rose *et al.*, "The VTR project: architecture and CAD for FPGAs from verilog to routing," in *FPGA*, 2012, pp. 77–86.