EURECA: On-Chip Configuration Generation for Effective Dynamic Data Access

Xinyu Niu
Department of Computing,
Imperial College London
niu.xinyu10@imperial.ac.uk

Wayne Luk
Department of Computing,
Imperial College London
w.luk@imperial.ac.uk

Yu Wang
Department of
Electronic Engineering,
Tsinghua University
yu-
wang@tsinghua.edu.cn

ABSTRACT

This paper describes Effective Utilities for Run-timE Configuration Adaptation (EURECA), a novel memory architecture for supporting effective dynamic data access in reconfigurable devices. EURECA exploits on-chip configuration generation to reconfigure active connections in such devices cycle by cycle. When integrated into a baseline architecture based on the Virtex-6 SX475T, the EURECA memory architecture introduces small area, delay and power overhead. Three benchmark applications are developed with the proposed architecture targeting social networking (Memcached), scientific computing (sparse matrix-vector multiplication), and in-memory database (large-scale sorting). Compared with conventional static designs, up to 14.9 times reduction in area, 2.2 times reduction in critical-path delay, and 32.1 times reduction in area-delay product are achieved.

Categories and Subject Descriptors

C.0 [**Computer System Organization**]: System architectures

General Terms

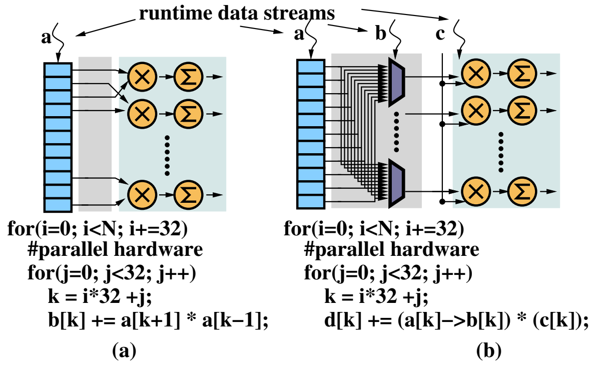
Design; performance

Keywords

On-chip configuration generation; runtime reconfiguration; dynamic data access

1. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) provide a platform to implement customised data-paths for target applications. Orders-of-magnitude improvements in performance and power efficiency have been achieved over software designs, for applications such as financial modelling [21] and signal processing [15]. The applications that can maximally exploit the potential processing capability of FPGAs tend to favour static implementations: the connections between memory data and data-paths, as well as the operations in data-paths, are predefined during compile time and stay the same during runtime (see Figure 1(a)). In these applications, the static data connections and operators are often pipelined. At each clock cycle, all the implemented hardware resources are active, generating one result per data-path.



```
for(i=0; i<N; i+=32)
  #parallel hardware
  for(j=0; j<32; j++)
    k = i*32 +j;
    b[k] += a[k+1] * a[k−1];
```
(a)

```
for(i=0; i<N; i+=32)
  #parallel hardware
  for(j=0; j<32; j++)
    k = i*32 +j;
    d[k] += (a[k]−>b[k]) * (c[k]);
```
(b)

Figure 1: Current FPGA support for applications with (a) static data access and (b) dynamic data access.

While the processing capability of FPGAs has been demonstrated, the lack of support for dynamic operations, especially dynamic data access, limits the use of FPGAs as mainstream data processors. *Dynamic data access* refers to the capability of providing efficient parallel access to dynamic data structures such as linked lists. As shown in Figure 1(b), when data access operations `a[k]->b[k]` depend on runtime variable `b` instead of fixed offsets, each data access operation requires various connections between data-paths and memory. In hardware, all such connections need to be statically implemented, while only one of the connections is active in each cycle. As the number of connections for a dynamic data access operation increases, the benefit of hardware over software implementations diminishes.

Efficiently supporting such dynamic data access is a long-standing challenge for FPGA-based designs. Applications with dynamic data access are common in social networking (Memcached), scientific computing (sparse matrix-vector multiplication), graph traversal (breadth-first search), in-memory database (sorting, selection), and embedded systems (H.264). Previously, researchers proposed solutions to work around the dynamic data access in these applications, such as compromising memory management flexibility (Memcached [5]), replicating on-chip memory buffers (sparse matrix-vector multiplication [28]), and limiting parallelism (large-scale sorting [10]). As discussed in Section 5, these solutions either limit the market acceptance of the developed applications, or affect the application performance.

The objective of this work is to address the dynamic data access challenge in hardware designs, without affecting the

FPGA'15, February 22–24, 2015, Monterey, California, USA.
Copyright © ACM 978-1-4503-3315-3/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684746.2689076.

current support for static applications. We propose a memory architecture known as EURECA that can be integrated with FPGAs, which meets the following requirements.

R1: to efficiently accommodate intensive dynamic data access (thousands of parallel wires, each with hundreds of possible runtime connections), without sacrificing application functionality or performance.

R2: to be compatible with existing synthesis tools and hardware languages, and to be transparent to applications without dynamic data access.

R3: to have overhead as small as possible when integrated with the relevant reconfigurable architectures.

Section 7 explains how these requirements are met by EURECA. The proposed memory architecture is underpinned by a novel runtime reconfiguration approach: instead of physically storing all possible configurations, the configurations are generated on-chip from user logic. At each cycle, the generated configurations update the implemented connections, ensuring implementation of only the active connections during runtime, while enabling applications with dynamic data access to be implemented with the same efficiency as static designs.

Outline. Section 2 discusses the related work. Section 3 presents an overview of the EURECA memory architecture in three aspects: (1) configuration flow, (2) architecture integration, and (3) EURECA-based designs. Section 4 shows the circuit details of a EURECA module. Section 5 studies the use of EURECA architectures in three benchmark applications: Memcached, sparse matrix-vector multiplication, and large-scale sorting. For each benchmark application, we discuss the design challenges, the improvements, and the impacts on applications with similar data access patterns. Section 6 evaluates the proposed architecture in terms of architecture efficiency, with the benchmark applications synthesised, placed and routed on a commercial FPGA enhanced with EURECA support. Measured results are based on EURECA circuits developed using Cadence Virtuoso with 65nm technology. Section 7 discusses the potential and the limitations of the proposed approach.

2. RELATED WORK

Previous work has explored communication operation support in reconfigurable system. Coarse-grain architectures such as Matrix [12], Tilera [23] and Ambric [3] implement distributed general-purpose processors and dedicated communication networks on-chip. Instruction execution of these architectures can involve dynamic data access, with the support of local memories and global communication networks. These architectures need new programming models to coordinate distributed processor execution, and fine-grain parallelism is not always captured in designs targeting these architectures. Another direction to improve communication efficiency in reconfigurable designs is to build high-level memory abstractions and optimisation tools. Memory abstractions such as CoRAM [7] decouple memory management from computation, and provide virtualised memory interfaces to users. However, CoRAM maps dynamic data access into separate local caches in processing elements, which can reduce data locality. While some high-level synthesis tools adopt polyhedral transformation [17] to improve memory bandwidth utilisation, such polyhedral approaches do not support dynamic data access. In contrast, this work enhances reconfigurable architectures with efficient support for dynamic data access, with fine-grain parallelism in reconfigurable designs preserved.

Runtime reconfiguration techniques provide opportunities to unfold dynamic data access in time dimension. In [27], 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. For dynamic data access in high-performance applications, the implemented connections need to be updated every iteration. In this case, the reconfiguration time dominates the overall execution time of a reconfigurable design. To reduce reconfiguration time, DPGA [20] and time-multiplexed FPGAs [22] are proposed. In these architectures, configuration memories for reconfigurable logic are replicated to store multiple configurations on-chip. The 3D programmable architecture from Tabula [19] replicates the configuration of logic blocks as well as interconnect. Configurations of implemented designs thus can be updated within a cycle. The replicated configuration memories, however, introduce large area and power overhead. The inefficiency in previous runtime reconfiguration approaches is due to the need for storing all possible configurations, either on-chip or off-chip. The EURECA approach adopts a new configuration flow that only stores the active configurations in each cycle.

3. EURECA MEMORY ARCHITECTURE

The **configuration flow** in FPGAs defines how they can be reconfigured to implement different customised designs. As shown in Figure 2(d), the EURECA configuration flow includes Static Configuration Memory (SCM), Configuration Generator (CG), and Dynamic Configuration Memory (DCM), where the SCM defines the operations of the CG, the CG generates configuration data based on runtime variables, and the output of the CG writes into the DCM. In this work, we couple the DCM with routing multiplexers, and group the runtime reconfigurable multiplexers into EURECA modules. This new configuration flow brings three benefits: (1) the necessity to store all possible configurations is eliminated, since the DCM only stores the active configuration; (2) generating and adapting configurations on-chip significantly reduces reconfiguration time, allowing runtime reconfiguration operations to finish within one cycle; (3) implemented in user logic, CGs are customised to application requirements. For static designs without dynamic data access, no CGs will be necessary.

The configuration flow for static designs is demonstrated in Figure 2(a) and (b). Static Data-Paths (SDPs) are specified with a single configuration. Idle operators are introduced when dynamic operations are required.

In previous approaches, multiple configurations are prepared in advance, as shown in Figure 2(c). This configuration flow is inefficient because only one of the configurations is used. When stored off-chip (partial reconfiguration), the large reconfiguration time prohibits fine-grained reconfiguration [27]. When stored on-chip, the increase in memory capacity introduces large area overhead [20, 22]. Moreover, the additional memory area is fixed once FPGAs are fabricated. Static designs are implemented with the same area overhead, although only one of the replicated configuration memories is required.

Architecture overview. Integrating the EURECA memory architecture into an existing reconfigurable architecture, as demonstrated in Figure 3, includes three steps: (1) divide on-chip memory blocks into memory groups, and couple each memory group with a EURECA module; (2) implement on-chip memory controllers as hard blocks, and couple each memory controller with a EURECA module; (3) arrange the EURECA modules in columns, and insert the EURECA columns into existing routing and logic fabrics. A EURECA module is the basic building block in the EURECA memory architecture. The module provides I/O ports (1) to take CG output to update the DCMs in a EURECA module, and (2) to provide reconfigurable connections between user logic and memory elements. The EURECA modules, when cou-

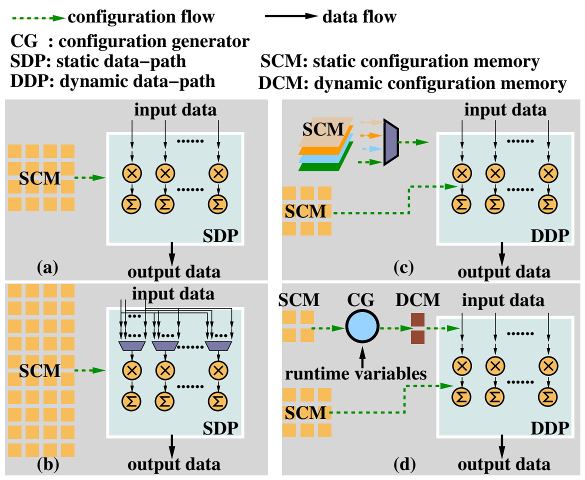


Figure 2: Data and configuration flow of (a) static design supporting static operations, (b) static design supporting dynamic operations, (c) dynamic design with prepared configurations, and (d) dynamic design reconfigured with EURECA approach.

pled with memory groups, provide access to on-chip memory, while those coupled with memory controllers provide access to off-chip memory. The I/O ports of memory groups and memory controllers are hard-wired into EURECA modules, while the connections between user logic and EURECA modules are statically reconfigurable. In the current architecture, the interconnections between EURECA modules are mapped into routing channels.

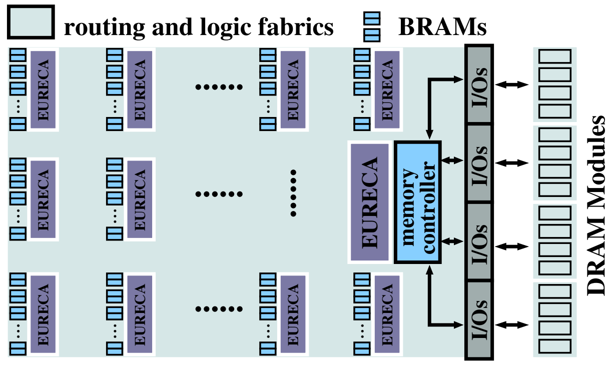
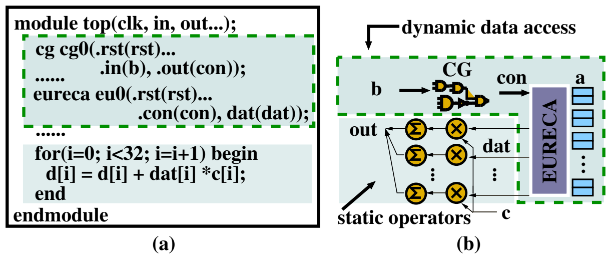


Figure 3: EURECA memory architecture overview.

EURECA-based reconfigurable designs are developed with hardware languages such as Verilog and VHDL, and synthesised with existing tool chains. A EURECA-based design contains SDPs, CGs and instantiated EURECA modules, where SDPs implement the static operations of an application, while CGs and EURECA modules support dynamic data access. Figure 4(a) and (b) respectively demonstrate the pseudocode and the hardware implementation of the algorithm in Figure 1(b). In this example, the SDPs are parallel multiply-and-accumulate modules, and the dynamic operations refer to the data access to `a`. We store `a` in a BRAM group, and instantiate the corresponding EURECA module. To reconfigure the EURECA module, we develop a CG that takes `b` as runtime input, and generates corresponding configurations `con`. As shown in Figure 4(a), the EURECA module takes `con` to update its DCM, and provides the SDPs the dynamic connections to `a`. This example design is a simplified sparse matrix-vector multiplication kernel. The application will be discussed in more detail in Section 5.2.



```
module top(clk, in, out...);
  cg cg0(.rst(rst)...
  ......      .in(b), .out(con));
  eureca eu0(.rst(rst)...
                .con(con), dat(dat));
  ......
  for(i=0; i<32; i=i+1) begin
    d[i] = d[i] + dat[i] *c[i];
  end
endmodule
```

Figure 4: A EURECA example design in Verilog, for the application in Figure 1(b).

4. EURECA MODULE

The EURECA modules play a key role in our runtime reconfigurable design: to take runtime configurations and to update data connections correspondingly. As shown in Figure 5, a EURECA module consists of runtime reconfigurable multiplexers, configuration control units, and a configuration distribution network. A EURECA module supports both dynamic read and dynamic write. We use the dynamic read to illustrate the functionality of a EURECA module. The development of a EURECA module follows three principles: (1) grouping dynamic connections to reduce routing complexity; (2) sharing runtime configurations to minimise CG resource usage; (3) supporting both static data access and dynamic data access with various data widths

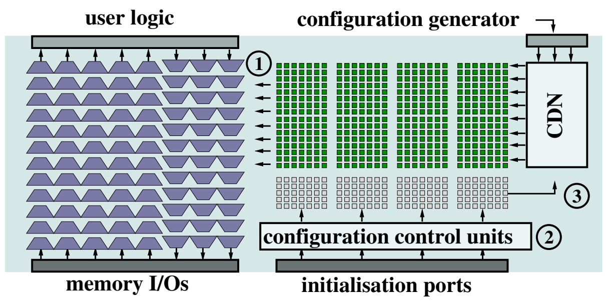


Figure 5: A EURECA module with (1) runtime reconfigurable multiplexers, (2) configuration control units, and (3) Configuration Distribution Network (CDN).

Runtime reconfigurable multiplexers refer to the dynamic connections between memory I/Os and user logic. A reconfigurable multiplexer contains a routing multiplexer and runtime writable SRAM cells (named as multiplexing SRAM cells) that define the implemented connection. We set the minimum data width supported in a EURECA module to be 1 byte, and divide the reconfigurable multiplexers into connection groups, with each group containing 8 multiplexers (bit-level dynamic connections are implemented with user logic due to the relatively small area usage). Figure 6 shows an example connection group with 256 input wires from memory, labelled as i_0 to i_{255}. The 256 input wires correspond to 32 input bytes. Correspondingly, routing multiplexers $m_0 \sim m_7$ in the connection group have 32 input wires. External designs use the output of the routing multiplexers $o_0 \sim o_7$ as an 8-bit dynamic connection.

We align the connections between memory elements and routing multiplexers, such that the multiplexers in a connection group can share the same multiplexing SRAM cells. For the example in Figure 6, the wires in the first input byte ($i_0 \sim i_7$) are correspondingly connected to the first input wires of $m_0 \sim m_7$. Similarly, the second input byte ($i_8 \sim i_{15}$) are connected to the second input wires of $m_0 \sim m_7$, in the same order. During runtime, to connect to the second input byte, $m_0 \sim m_7$ share the same configuration value (00001). To dynamically reconfigure the connection, CGs only need to generate one configuration for a connection group, which reduces the resource usage of CGs by 8 times. The connections between EURECA modules and memory I/Os are fixed, while the connections to user logic are configurable. This reduces the routing complexity of EURECA-based designs, and preserves full configurability between user logic and dynamically accessed data.

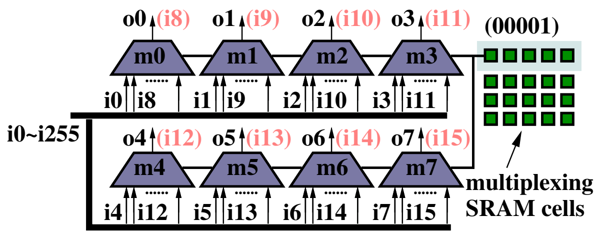


Figure 6: 8 reconfigurable routing multiplexers in a EURECA module, sharing the same SRAM cells.

Configuration control units operate EURECA modules in 4 different modes, to support static data access as well as dynamic data access with different data widths. In a EURECA module, we use SRAM cells to store configuration information, and organise the SRAM cells in rows. In the example case with 256 input wires, each row contains the multiplexing SRAM cells for 6 connection groups (30 SRAM bits). Figure 7 shows the SRAM organisation, and the circuit details of a multiplexing SRAM cell. In an SRAM cell, the WL port controls whether the cell value can be updated. A Write-Enable (WE) row is inserted below the multiplexing SRAM rows, and a column of shift registers is added. Each bit in the WE row controls the WL ports of an SRAM column, and each shift register controls the WL ports of an SRAM row. The input of an SRAM cell is multiplexed between the initialisation ports and the configuration ports. A EURECA module also contains a state bit, to indicate whether the current module operates in static or dynamic mode.

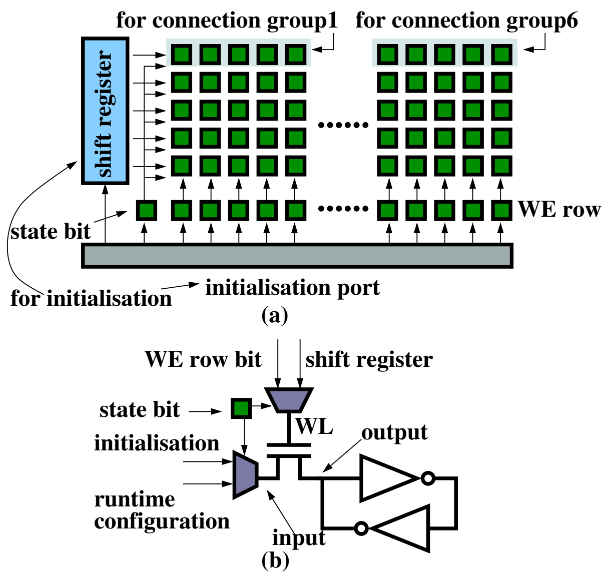


Figure 7: (a) The control units for the multiplexing SRAM cells in a EURECA module. (b) The circuit details of a multiplexing SRAM cell.

In the `initialisation mode`, the first 32-bit configuration data from the initialisation ports write the state bit to '1', and push a '1' bit into the shift registers. In a multiplexing SRAM cell, the initialised state bit selects initialisation port data as input, and use shift register output as the WL signal. As the '1' bit shifts through the registers, the configuration data are written into SRAM cells row by row, which define the initial connections within a EURECA module. The state bit and the WE row are initialised last, with the values determined by the following operation mode.

To operate a EURECA module in the `static mode` during runtime, we set the state bit to '1', and push '0' into the shift registers, so that the connections between user logic and accessed data are fixed during runtime. This mode is used for applications with only static data access. No CG is implemented, and the EURECA modules become transparent to the implemented data-paths.

In the `dynamic mode`, all multiplexing SRAM cells are updated by CG output in parallel. The state bit is set to be '0' to select runtime reconfigurations as input, and use WE row bits to control the WL ports. All bits in the WE row are initialised to be '1'. In this mode, the dynamic connections in a EURECA module get full reconfigurability, i.e., the connections can be connected to all input bytes. The dynamic mode is used when the data involved in dynamic data access are 8-bit wide. As an example, in Memcached, due to the flexible key width, dynamic pointers implemented in hardware can point at any byte in the fetched off-chip memory data. The dynamic mode is therefore used.

In the `partially dynamic mode`, part of the multiplexing SRAM cells are updated during runtime, while the others remain static. In this mode, certain bits in the WE row are configured to be '0', turning off the WL ports of the corresponding SRAM columns. The partially dynamic mode is used when application data width is larger than 8 bits, and therefore not all possible connections for 8-bit data are required. More details for this mode will be given next.

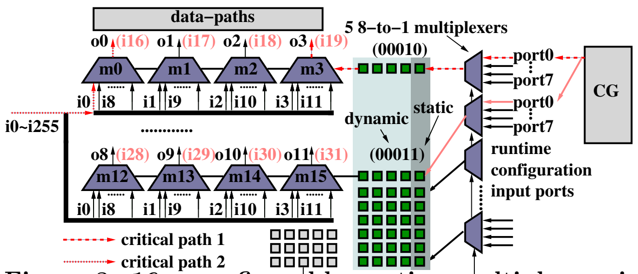


Figure 8: 16 reconfigurable routing multiplexers in partially dynamic mode, providing a 16-bit dynamic connection.

Configuration distribution network (CDN) shares runtime configurations among connection groups, when application data width is wider than 8 bits. As data width increases, a dynamic connection contains multiple connection groups. These connection groups are connected to the same input data during runtime, with static offsets. Therefore, we operate EURECA modules in the `partially dynamic mode`, to fix the static configuration bits, and share the remaining dynamic bits in the connection groups. Figure 8 shows the data connections for a 16-bit dynamic data access, which contains two connection groups $m_0 \sim m_7$ and $m_8 \sim m_{15}$. In the dynamic data access, $m_0 \sim m_7$ provide the lower 8 bits, and $m_8 \sim m_{15}$ provide the higher 8 bits. In other words, there is a 1 byte offset between the two connection groups. We thus fix the lowest configuration bit for the first and the second connection group to be '0' and '1', respectively. As an example, to dynamically connect to the second input data ($i_{16} \sim i_{31}$), $m_0 \sim m_7$ are configured with "00010", and $m_8 \sim m_{15}$ are configured with "00011". Since the lowest bit is fixed, the same runtime configuration "0001"

Algorithm 1 Key searching algorithm in Memcached.

```
input: char *key, nkey
output: item *it
1: hv = hash(key, nkey);
2: it = primary_table[hv];
3: while it do
4:     if nkey==it→nkey && strcmp(key, it→key) then
5:         return it;
6:     end if
7:     it = it→next;
8: end while
9: return NULL;
```

can be shard via the CDN, from `port0`. In this work, we use 8-to-1 multiplexers in a CDN, to support applications with 8-bit to 64-bit data width. The configurations for the CDN are static, and are initialised in the `initialisation mode`.

Critical path. For a EURECA-based design, the execution process within a clock cycle is as follows. First, CGs output configuration information based on runtime variables. Second, the CDN distributes the configuration information to the multiplexing SRAMs in a EURECA module, which reconfigures the implemented connections. Third, when the connections are reconfigured and memory data appear at the EURECA I/O ports, SDPs start data processing. Therefore, as shown in Figure 8, there are two potential critical paths in a EURECA module: (1) between CG, multiplexers in CDN, multiplexing SRAMs, and routing multiplexers; and (2) between memory data and routing multiplexers.

5. CASE STUDIES

5.1 Memcached

Our first benchmark application, 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, and uses slab allocation to allocate data chunks. Each hash bucket contains one to multiple hash entries, and each hash entry stores the address of its slab data. In hardware, as shown in Figure 9(a), we store a primary hash table on-chip to keep track of hash bucket addresses, and keep the hash data and the slab data in off-chip memory. Algorithm 1 presents the kernel operations to search for a hash entry. Once a request packet arrives, the packet decipher takes the Memcached command (e.g. get news21), and the hash function generates a hash value hv based on the key value (news21) and key length (6) (line 1). The algorithm then fetches the hash bucket address it based on hv (line 2), and traverses all hash entries in the bucket (line 3∼8). Once a hash entry with matched key value and key length is found, the corresponding slab data are fetched, and Memcached returns a response packet.

Design challenge. In Memcached, the key length varies from 1 byte to 256 bytes. When pointed to by dynamic pointers, the hash data have unaligned starting addresses, which lead to design challenges in hashed item search. As an example, if we assume a 6-byte memory channel, and data address it aligns with the channel width (i.e. $it\%6 = 0$), the data can be directly fetched and compared (see Figure 9(b)). Due to the variable key length, the accessed data are unaligned during runtime. As shown in Figure 9(c), when $it\%6 = 2$, the loaded data are in wrong order `21news`. The search module continues searching as `21news` != `news21`, although this hash entry stores the target hash data.

Previous solutions. In an FPGA implementation [4], the key size $nkey$ is fixed at 64 bytes. Fixing the key size

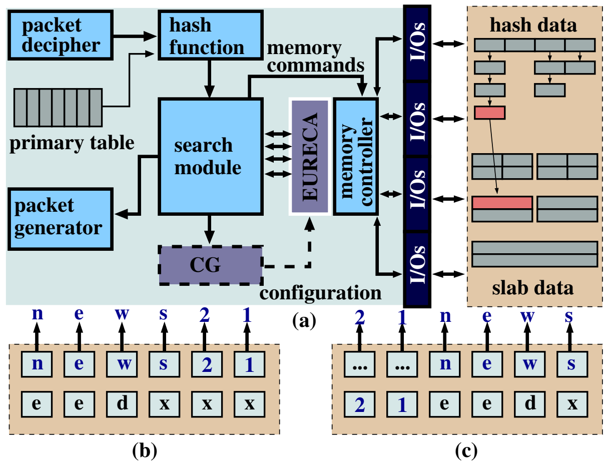


Figure 9: (a) A EURECA-based Memcached design. Off-chip data access operations with (b) aligned and (c) unaligned starting addresses.

aligns the runtime data. However, this design compromises functionality as keys smaller than 64 bytes need to be padded, and keys larger than 64 bytes cannot be supported. It is argued in [11] that this restrictive memory management will limit the market acceptance, based on recent industry trends. An architecture with soft processors and reconfigurable fabric is proposed in [11]. Parallel processing is implemented in hardware, and memory data are managed in soft processors. This approach consumes large on-chip area to integrate the processors, and introduces intensive communication between hardware and software.

EURECA solution. In a EURECA architecture, we instantiate a EURECA module to align the off-chip data access $it \rightarrow (nkey, key)$. The EURECA module is coupled with a memory controller. Since Memcached accesses data at byte level, we operate the EURECA module in the `dynamic mode`. As shown in Figure 9(a), the search module takes the initial it from Memcached commands, and updates it with the fetched data when traversing the hash buckets, until the matched hash entry is found. The memory controller fetches off-chip data pointed by it. The CG generates configurations for byte-level dynamic connections based on it as well as the position of a byte in off-chip memory channels. The CG operations can be expressed as $con_i = (it + i)\%N$, where i indicates the i-th byte in data bus, and N is the memory channel width (N=6 in the example in Figure 9(b)). The generated configurations therefore are updated cycle by cycle in the search process.

Table 1: Comparison of Memcached solutions.

solution	complexity	functionality	throughput
static	N^2	full	N
[4]	N	compromised	$< N$
[11]	$N + C_{cpu}$	full	n/a
EURECA	$N + C_{eureca}$	full	N

N: on-chip data bus width (byte).
C_{cpu}, C_{eureca}: constant overhead.

Discussion. Unaligned data access is common in applications with complex data structures. The unaligned data addresses come from variable data length, such as the hash key in Memcached, the chromosomes in genetic algorithm, and the FM-index in DNA sequencing. Table 1 compares the various solutions. Statically implementing all possible connections introduces N^2 area usage, where N is the data stream width in bytes. For reconfigurable designs with high memory bandwidth, this is infeasible to implement. Sacrificing application functionality affects the market acceptance of the product, and integrating general processors introduces large area and communication overhead. The EURECA architecture enables applications with unaligned data access to be efficiently implemented, without sacrificing functionality or requiring general-purpose instruction processors

5.2 Sparse Matrix-Vector Multiplication

Our second benchmark application, Sparse Matrix-Vector multiplication (SpMV), is widely used in scientific computing and industrial development. SpMV multiplies a sparse matrix `M` with a dense vector `vec`, as shown in Figure 10(a). A sparse matrix can be stored in Compressed Sparse Row (CSR) format. Figure 10(b) illustrates the CSR format for the sparse matrix in Figure 10(a). The CSR data contain three vectors: `val`, `col` and `offset`. `val` stores all the non-zero elements of a sparse matrix, `col` indicates which column each `val` is in, and `offset` points to the starting data of each row. In this example sparse matrix, item A is stored in the first column of the second row. `col[2]=0` indicates A is in the first column, and `offset[1]=2` indicates the second row starts from the third non-zero element.

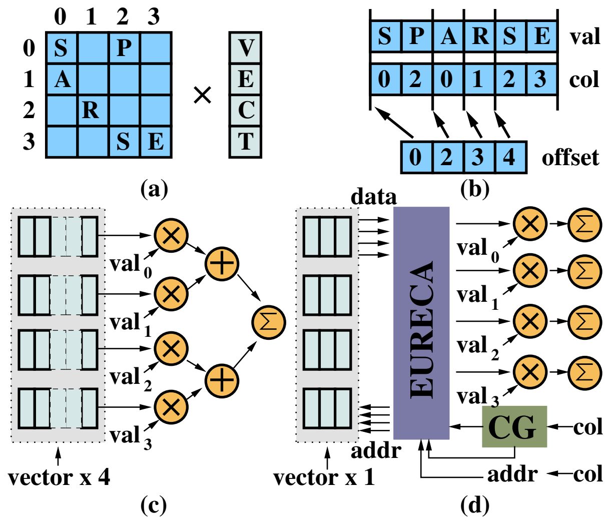


Figure 10: (a) An SpmV problem, stored in (b) CSR format. SpMV architectures with (c) replicated vector memory and (d) shared EURECA-based vector memory.

Challenge. The design challenge comes from the random data access to the dense vector. As shown in Algorithm 2, in each row, SpMV multiplies the `val[j]` with corresponding vector value `vec[col[j]]` (line 4). Since `col` are runtime data, the vector data access becomes random. To avoid the long latency of off-chip random data access, designers normally buffer `vec` on-chip. In FPGAs, the distributed BRAMs can be grouped as a unified memory architecture, which provides replicated SpMV data-paths the parallel data access to `vec`. Due to the randomness of the vector data access, each data-path needs to be able to access all BRAMs in the memory architecture. As an example, if a shared vector memory provides 4 output ports to 4 parallel data-paths, each data-path needs to connect to all the memory blocks in this vector memory, i.e., all of the 4 output ports. For an SpMV design with N data-paths, this leads to N^2 possible runtime connections.

Previous solution. To address this issue, the SpMV architecture in [28] assigns each data-path a separate copy of the vector data, as shown in Figure 10(c). Given an FPGA with *mem* on-chip memory capacity, the architecture supports vector size up to mem/N, and needs to block the sparse matrix data access when the vector size exceeds

Algorithm 2 Sparse matrix-vector multiplication.

```
1: for i ∈ 0 → num_rows do
2:     res[i]=0;
3:     for j ∈ offset[i] → offset[i+1] do
4:         res[i] += val[j] * vec[col[j]];
5:     end for
6: end for
```

this limit. This architecture is sensitive to matrix sparsity. When the blocked matrix rows contain fewer than N non-zeros in the data-paths. In Table 2, we calculate the efficiency as the ratio between measured performance and the theoretical peak performance. In [28], the idle cycles reduce the average efficiency to 42%. As N increases, the efficiency will further reduce.

EURECA solution. Our EURECA-based SpMV design instantiates a EURECA module coupled with a memory group with N memory blocks. The EURECA module operates at the `partially dynamic mode`, as the design uses 32-bit data. We save the vector data into a shared on-chip memory architecture, which is connected with N data-paths. The data-paths stream CSR data from off-chip memory. A CG generates the runtime configurations for memory address and vector data, based on the value of `col`, as shown in Figure 10(d). For a vector data request, the CG first reconfigures the address connection to direct the address input into the right memory block, and reconfigures the data connection when the requested data appear at the memory output ports. The address input and the data output are pipelined. In this work we use 10 sparse matrices from [8] to simulate the computation. The average efficiency for the EURECA solution reaches 85%, when $N = 64$. The idle cycles in this solution mainly come from memory conflicts, where multiple data-paths are accessing data in the same memory block. As N increases, the memory conflict ratio decreases, and the computational efficiency increases.

Table 2: Comparison of SpMV solutions.

solution	complexity	vector size	efficiency
static	N^2	*mem*	85% ($\propto N$)
[28]	N	mem/N	42% ($\propto 1/N$)
EURECA	$N + C_{eureca}$	*mem*	85% ($\propto N$)

Discussion. Random access to parallel data is common in complex computational kernels such as sparse matrix processing and graph traversal. In these applications, the frequently accessed data (dense vector and bit-mask matrix) are normally stored on-chip to reduce data access latency. In modern reconfigurable computing platforms, hundreds of data-paths are implemented to process data in parallel (N=128 in [6]). Statically implementing a shared memory architecture for these applications is not feasible, due to the N^2 area complexity. Replicating on-chip data significantly reduces the effective cache size. The EURECA architecture enables a shared on-chip memory architecture to be implemented in reconfigurable designs, to support parallel random data access. The cache miss rate and the data-path idle cycles can thus be reduced.

5.3 Large-Scale Sorting

Our third benchmark application, sorting, is one of the most extensively researched subjects because of the need to quickly organise millions to billions data items in a database. As shown in Figure 11(a), sorting networks [1, 16] sort small data set in parallel. When targeting large-scale data, the sorting operations are divided into a sorting phase and a merging phase. In the merging phase, as shown in Figure 11(b), select-and-pop units merge small sorted data chunks step by step. In each step, the data chunks are

Algorithm 3 Parallel merging of sorted data chunks.

```
input: sorted data chunk A, B, with size n
output: merged data chunk C, with size 2n
 1: a=&A[0]; b=&B[0]; c=&C[0];
 2: while c ≤ 2n do
 3:   for i ∈ 0 → N do
 4:     if a[i] < b[N-1-i] && a[i+1] > b[N-2-i] then
 5:       commit = i+1; break;
 6:     end if
 7:   end for
 8:   assign(c, a, commit);
 9:   assign(c+commit, b, N-commit);
10:   sort(c, N);
11:   a+= commit; b+= N-commit; c+=N;
12: end while
```

buffered in on-chip FIFOs followed by the select-and-pop units.

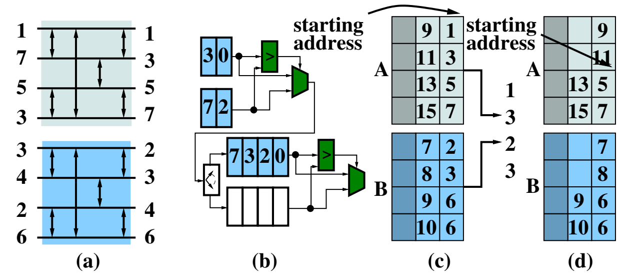


Figure 11: (a) A sorting network. (b) Two basic select-and-pop units to merge sorted data. FIFOs starting addresses in (c) the first cycle and (d) the second cycle.

Challenge. The main challenge is to commit multiple sorted data in each cycle when parallelising the merging operations. Algorithm 3 presents a parallel merging algorithm, where N data are merged every iteration. The algorithm first compares the first N data from both A and B, and labels the index for the smallest N data as *commit* (line 3∼7). *commit* indicates the first *commit* data from A and the first $N - commit$ from B should be committed into C (line 8∼9). Finally, we sort the committed N data, and increase array indices correspondingly (line 10∼11).

When implemented in hardware, the data connections between sorted data chunks (A, B) and comparators depend on runtime computation results. In the example in Figure 11(c), each sorted data chunk is stored in a FIFO with 4 output ports, and *commit*=2 in the first cycle. After reading the first two data from A and B, the starting addresses change from 0 to 2. As shown in Figure 11(d), the output data from FIFO A become (9,11,5,7) in the second cycle, although the right order is (5,7,9,11). If we statically configure the connections between FIFO output and comparator input, the comparison operations (line 3∼7) will return wrong results. During runtime, for a FIFO with N output ports, the FIFO output data have N different starting addresses.

Previous solutions. In a software implementation, the committed data are read sequentially, which eliminates the impact of variable starting addresses. In order to read multiple data per cycle with correct functionality, a hardware implementation needs to cover all possible runtime scenarios, which leads to $O(N^2)$ complexity for FIFOs with N output ports. [10] proposes a select-and-pop unit with 2 data committed each cycle. However, the approach is not scalable for higher parallelism due to the quadratic design complexity.

EURECA solution. In our solution, we implement 2 EURECA modules, each coupled with a BRAM group. A BRAM group is implemented as a FIFO with N inputs and N outputs, where N is the number of committed data in each cycle. During runtime, we load the sorted data from the FIFOs, and reorganise the loaded data in the EURECA modules. A comparison module compares the reorganised data, commits the smallest N data, updates the *commit* variable, and outputs FIFO read enable signals to shift the committed data out of the FIFOs. CGs for the EURECA modules calculate the configurations based on the current starting address, *commit*, and the position of the reconfigured connections in a FIFO. The CGs and the comparators in the comparison module form a feedback loop. Operations in this feedback loop are not pipelined.

Table 3: Comparison of sorting solutions.

solution	complexity	data size	throughput
sorting network	$C \cdot N \log_2(N)$	small	N
merger	N^2	large	N
EURECA	$N + C_{eureca}$	large	N

Discussion. The sorting problem is an example of applications in which data access operations in the current cycle depend on the computation results from previous cycles. When developed in parallel programming models for hardware, such a data access operation unfolds into multiple possible connections. Table 3 compares different solutions for the sorting application. Sorting networks [1, 16] are only applicable to small-scale data sets, as the design complexity is proportional to data size. Combining sorting networks and data mergers solves the data size limitation. However, a merger with N parallel units suffers $O(N^2)$ complexity due to the N possible starting addresses for read operations. Our EURECA solution solves both the data size limitation and the quadratic area complexity, by generating configurations for the next cycle based on the computation results in the current cycle.

6. EVALUATION

This section evaluates the EURECA architecture. First, we enhance a baseline architecture with EURECA support, and set up an experiment environment to synthesise designs into the enhanced architecture. Second, we evaluate the general benefits and the overhead of the enhanced architecture, and compare the EURECA approach with previous runtime reconfiguration techniques, for supporting dynamic data access. Third, for the benchmark applications, we develop both EURECA-based dynamic designs and static designs based on the baseline architecture, which are synthesised, placed and routed in the corresponding architectures, to measure the impacts on application performance.

6.1 Experiment Methodology

Baseline system. The experiment setup simulates a commercial reconfigurable system in terms of FPGA specification and off-chip memory systems, to capture the design realisation in practice. We assume the baseline system to be the Max3424A from Maxeler Technologies, which contains a Xilinx Virtex-6 SX475T FPGA and provides up to 38.4 GB/s memory bandwidth. As listed in Table 4, the off-chip memory system contains 4 128-bit DDR3 data channels, which operate at 303 MHz. We adapt the detailed architecture file developed in [9] to describe the baseline architecture. A CLB in this architecture contains two slices, each of which contains four Basic Logic Elements (BLEs), and a BLE contains a fracturable 6-input LUT and two FFs. We modify the architecture file to align with the architectural details of Virtex-6 SX475T shown in Xilinx FPGA Editor. The measured average channel width in FPGA Editor is 150. However, VTR [18], the synthesis tool used in our experiments, assumes a simplified routing model. Routing features in commercial FPGAs, such as non-unified channel width and diagonal wires, are not supported by VTR. In [13], the channel width is set to be 300 for Stratix IV. In this work, we inflate the channel width to 256 to approximate the actual routing capacity.

EURECA implementation. We develop the EURECA module full-custom at the transistor level in the Cadence Design Platform Virtuoso, with a 65-nm CMOS technology from UMC. Inside a EURECA module, multiplexers are implemented with pass transistors, 5T SRAM cells are used to dynamically configure routing multiplexers, and 6T SRAM cells are used to statically configure the 8-to-1 multiplexers in the CDN. In terms of area, the routing multiplexers and the 5T SRAM cells occupy most of a EURECA module. In terms of delay, as discussed in Section 4, the 8-to-1 multiplexers, the 5T SRAM cells, and the routing multiplexers are in the critical paths. To balance the module area and delay, we size the routing multiplexers to minimum width, optimise the 8-to-1 multiplexers for speed, and insert 4-time drivers between the 8-to-1 multiplexers and the 5T SRAM cells. The sizing approach for 5T SRAM in [14] is adopted to ensure robust read and write operations during runtime. Circuits outside the critical paths, such as the shift registers and the 6T SRAM cells, are optimised for area.

Table 4: EURECA architecture parameters.

baseline FPGA	CLB: 37,200 BRAM36: 1064 DSP48x2: 1008 tile: 136x360 channel width: 256
DDR3 memory	width: 128*4 bits @ 303 MHz
a EURECA module	data I/Os: 1024 con I/Os: 896 address I/Os: 480 con I/Os: 160 area: 593,210 (325,754) delay: 0.17 ns power: 96.56 mw @ 150 MHz
on-chip memory	width: 1024*2 @ 150 MHz memory group: 32 BRAM36
EURECA layout	BRAMs: 7 columns * 4 modules memory controller: 2 modules

Architecture integration. EURECA modules are integrated with BRAMs and memory controllers. Given the parallelism in recent reconfigurable designs, we set each BRAM group to contain 32 BRAMs. The baseline architecture contains 1064 36Kb BRAMs, which are organised in 15 columns. Each of the BRAM groups is coupled with a EURECA module, which provides dynamic access to up to 32 memory blocks, or 128 different input bytes. The enhanced architecture therefore contains 7 EURECA columns. A EURECA column contains 4 EURECA modules, and sits in the middle of two BRAM columns. The left BRAMs are not connected to EURECA modules due to BRAM group granularity: a BRAM column contains 72 BRAMs, and the upper 8 BRAMs cannot construct a complete BRAM group.

The on-chip data-paths are set to operate at 150 MHz. The 38.4 GB/s memory bandwidth therefore corresponds to 2 1024-bit on-chip streams. We implement 2 memory controllers to connect the off-chip and on-chip data streams. Each memory controller is coupled with a EURECA module. As listed in Table 4, the off-chip memory system contains 4 128-bit (64 byte) DDR3 data channels. Along with the 303 MHz bandwidth and double data rate of DDR3, the 64-byte off-chip memory channel provides 38.4 GB/s bandwidth. Therefore, the maximum dynamic offset in off-chip data access is 64 bytes, i.e., a dynamic connection to off-chip data has up to 64 different input bytes. The routing multiplexers and 5T SRAMs in the 2 EURECA modules are customised for the reduced dynamic degree.

Synthesis environment. We use the VTR tool chain to synthesise designs into the EURECA architecture, with updated area and delay models. Carry chain is not supported in the current work. To model the design delay, we extract the delay information of CLBs, DSPs and BRAMs from the Xilinx TRCE results for the baseline architecture in speed grade -3, and use the routing delay information in [13], which is based on the same 40-nm technology node. The delay information for EURECA modules is measured from Cadence designs (room temperature and nominal voltage). The area information is expressed in the unit of minimum-width transistor area. We use the area model in [2] to estimate silicon area based on the drive strength of implemented transistors. The drive strength of the transistors in EURECA modules is collected from the implemented circuits. Table 4 lists the EURECA module areas (the area number in brackets is for the EURECA modules coupled with memory controllers). We estimate the CLB area based on the circuit specifications in [26], and adapt the area models for logic blocks based on those in [2]. To reduce routing complexity, for a EURECA module, we fix the configuration pins at the middle of the left and right sides, and spread the data I/O pins. The placement algorithm is modified such that once a EURECA module is placed, the coupled BRAMs are labelled as occupied.

6.2 Dynamic Connection Efficiency

This section evaluates the efficiency of the EURECA architecture, when supporting dynamic data access. We define the dynamic connection efficiency E as the ratio between the number of different runtime connections R supported by the same routing resources, and the overhead O for enabling such connections.

$$E = \frac{R}{O} = \frac{R}{o_a \cdot o_t} \tag{1}$$

For an architecture with enhanced runtime reconfigurability ($R > 1$), the overhead O includes (1) additional silicon area O_a consumed by reconfiguration infrastructure, (2) additional execution time O_t due to reconfiguration time, and (3) impacts on static designs. The third overhead is application-dependent, and therefore is difficult to generally quantify. Figure 12 compares the maximum channel width and the critical path of various applications targeting the baseline and the EURECA architectures. For static designs, using the EURECA architecture increases the channel width and the delay by less than 2% in average.

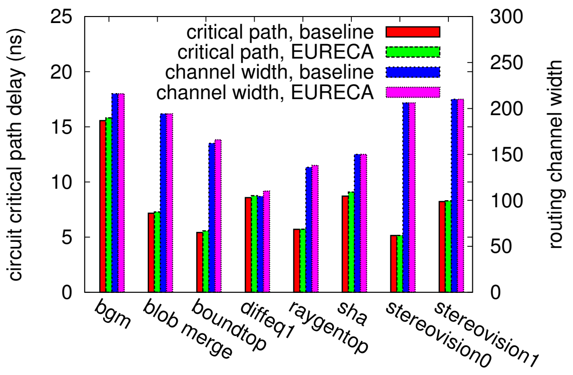


Figure 12: Impacts of the EURECA memory architecture on static designs, in terms of critical path delay and channel width.

An ideal architecture supports unlimited reconfigurability without area overhead ($o_a = 1$) or time overhead ($o_t = 1$). The efficiency E increases linearly with the number of possible connections for a dynamic data access operation, as shown in Figure 13. Based on the area information in Table 4, integrating the EURECA architecture increases the overall area of the baseline architecture by 1.17% (o_a=1.017). For the reconfiguration time, since in the same cycle, the connections in EURECA modules are reconfigured, and data are loaded through the reconfigured connections, o_t=1. The efficiency of EURECA architecture approximates the optimal level.

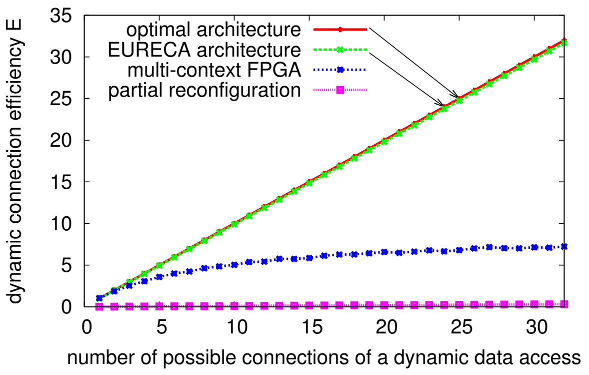


Figure 13: Dynamic connection efficiency E when the number of possible connections for a dynamic data access operation increases.

Partial reconfiguration provides unlimited R, as designers can store all partial configurations off-chip. The efficiency is limited by reconfiguration time. Given the smallest addressable configuration size (3232 bits) and the maximum reconfiguration throughput (400 MB/s) in the latest devices [24, 25], the minimum reconfiguration time is 1.01 μs (151 clock cycles for the 150 MHz operating frequency). When the implemented configurations need to be updated every cycle, 1 result is generated every 152 cycles (o_t=152). The efficiency for partially reconfiguring the routing fabric, as shown in Figure 13, is far from optimal.

Multi-context FPGAs enable new configurations to be applied within a cycle, since possible configurations are stored on-chip. In this experiment, we replicate the configuration memory of routing multiplexers in the connection blocks of the baseline architecture. The number of replicated configuration memory increases with R. As R increases, the area overhead quickly outweighs the increase in runtime reconfigurability. As shown in Figure 13, the efficiency of multi-context FPGAs reduces to 7.2 when $R = 32$, which indicates an area overhead of 4.42. To achieve the same reconfigurability of a EURECA module, the multi-context FPGAs need to replicate 128 configuration memory sets.

6.3 Application Evaluation

To further evaluate the EURECA approach, we develop a static design and a dynamic design for each benchmark application. The static design and the dynamic design share the same SDPs. In the static design, all the possible connections for dynamic data access are statically implemented into the baseline architecture, with `if-else` expressions. The dynamic design adopts the EURECA solutions proposed in Section 5, and targets the EURECA architecture. Except the feedback loop in the sorting design, operations in the benchmark applications are pipelined to reduce critical path delay. Table 5 presents the measured design properties. Given the 64-byte off-chip memory channel, the Memcached application has 64 possible connection sets, each 128 bytes wide. The parallelism for SpMV and large-scale sorting is set to be 32.

The resource usage and the critical-path delay of SDPs indicate the initial design properties, when dynamic data access operations are not required. The design properties of the dynamic designs are at the same level as the initial design properties. The critical-path delay of dynamic designs is slightly reduced as the EURECA module aggregates the distributed memory I/O connections into a single module, which eliminates some long connections in the SDPs. When statically implementing the possible connections, the resource usage is increased by up to 14.9 times, and the delay is doubled. In the sorting design, two memory groups are used to build the two input FIFOs, the intensive connections between the memory groups and comparators introduce a comparatively large initial delay. Overall, the dynamic designs reduce the area-delay product by up to 32.1 times.

Statically implementing all possible connections significantly increases the routing complexity of the benchmark designs, since thousands of N-to-1 connections need to be routed, where where $N = 64$ for Memcached, and N=32 for SpMV and for large-scale sorting. As shown in Table 5, all the three static designs cannot be routed under the current routing infrastructure. The average channel width for the static designs is 356, which indicates the routing difficulty of the static designs, even on large-scale commercial FPGAs. In other words, with the EURECA architecture, applications and design approaches previously considered not suitable for hardware acceleration can be efficiently implemented. As discussed in Section 5 and Table 1∼3, the enabled features include flexible memory management, shared memory architecture supporting random and parallel data access, and parallel merging.

7. DISCUSSION AND CONCLUSION

This work is an initial investigation into the EURECA approach. We have only scratched the surface in this investigation, and more questions need to be answered. We discuss below the potential of the EURECA approach that has not been addressed in this work, and the limitations of the current research.

Potential. *(1) Supporting a wide range of applications.* Our initial investigation divides the applications with dynamic data access into 4 categories: (a) unaligned data access, such as the hash key in Memcached, the chromosomes in genetic algorithms and the FM-index in genetic sequence alignment; (b) random parallel data access, such as the dense vector in SpMV and the breadth-first traversal; (c) data-dependent data access, such as in-memory database operations (selection, merge join, etc); (d) pre-defined access patterns, such as the intra-prediction modes in H.264 and the orientations in histogram of oriented gradients. *(2) Enhancing FPGA programmability.* In existing memory abstractions and high-level synthesis tools such as CoRAM, LegUp and Vivado HLS, memory access of the replicated data-paths is restricted to avoid inefficient hardware. With the EURECA architecture, more programmer-friendly languages can be developed. *(3) Applying the EURECA approach to other reconfigurable architectures.* Given the area complexity of CGs, the EURECA approach is more suitable to coarse-grain computational units than fine-grain computational units, to rapidly switch between different runtime computational operations.

Limitations. *(1) Large-scale EURECA-based designs.* When a large number of EURECA modules are used in a reconfigurable design, the communication operations between the EURECA modules could potentially increase the critical-path delay. This problem can be solved by integrating a Network-on-Chip (NoC) with the memory architecture, where the communication operations between EURECA modules are mapped into the NoC. *(2) Experiment*

Table 5: Benchmark application performance. Each application contains SDPs and dynamic data access operations. baseline: a static design implemented in the baseline architecture, with dynamic data access operations expressed as if-else expressions. EURECA: a EURECA-based dynamic design.

	Memcached			SpMV			Large-scale Sorting		
	SDPs	baseline	EURECA	SDPs	baseline	EURECA	SDPs	baseline	EURECA
CLB	227	4399	234	459	3521	465	550	4875	561
DSP48x2	0	0	0	64	64	64	0	0	0
RAM36Kb	64	64	64	1024	1024	1024	64	64	64
EURECA	0	0	1	0	0	1	0	0	2
area[1] (10^6)	1.185	22.97	1.54	2.396	18.39	3.02	2.872	25.46	3.52
critical-path delay (ns)	6.7	13.94	6.46	6.54	12.74	6.17	9.51	11.56	9.51
area-delay product		**32.14x**	1x		12.57x	1x		8.792x	1x
routable[2]	✓	✗	✓	✓	✗	✓	✓	✗	✓
channel width	202	382[3]	211	215	317[3]	221	211	368[3]	204
throughput (per cycle)	128 bytes			32 partial results			32 sorted data		
enabled feature	flexible memory management			shared memory architecture			parallel merging		

[1] The design area is estimated by the minimum-width transistor area of consumed CLB and EURECA blocks. The areas of DSPs and RAM blocks are not included due to the lack of area information.
[2] Routable indicates whether the designs can be routed under the current routing infrastructure (channel width 256).
[3] Unroutble designs are re-synthesised with variable channel width.

setup. Architecture evaluation involves estimations which can affect the outcome of our experiments: (a) the VTR routing model inflates the channel width to match the routing capability of the actual FPGAs; (b) the minimum-width transistor area model [18] does not include wire area; (c) the baseline architecture area does not include the area of DSP and RAM blocks, due to lack of their area information; (d) the EURECA module is developed in 65-nm technology, while the baseline architecture is in 40-nm technology. The second limitation leads to underestimated module area, while the last two limitations overestimate the architecture overhead. Given the large improvements in design properties, we expect the impacts of these issues to be minor.

Conclusion. This paper proposes a novel memory architecture that supports dynamic data access. Instead of physically storing configurations during runtime, EURECA generates the active configuration at each time using reconfigurable logic, which eliminates inefficiency in previous runtime configuration approaches. In addressing the requirement R1 in Section 1, significant reductions in area and critical-path delay are achieved for three benchmark applications with intensive dynamic data access. For R2, static and dynamic designs are developed in Verilog and synthesised with VTR, into the EURECA architecture. For R3, we show that a Virtex-6 SX475T device enhanced by EURECA has 1% area overhead. Current and future work includes laying out the EURECA architecture, enhancing the EURECA architecture with NoC and coarse-grain computational units, developing more applications, and building high-level tools to enable automatic development of applications targeting devices enhanced by EURECA.

Acknowledgement. This work was supported in part by the UK EPSRC, the European Union Seventh Framework Programme under Grant agreement number 257906, 287804 and 318521, the 973 project 2013CB329000, the National Science Foundation of China (No. 61373026), the HiPEAC NoE, the Maxeler University Program, and Xilinx.

8. REFERENCES

[1] K. E. Batcher. Sorting networks and their applications. In *AFIPS*, pages 307–314, 1968.
[2] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for deep-submicron FPGAs.* Kluwer Academic Publishers, 2002.
[3] M. Butts et al. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *FCCM*, pages 55–64, 2007.
[4] S. R. Chalamalasetti et al. An FPGA memcached appliance. In *FPGA*, pages 245–254, 2013.
[5] S. R. Chalamalasetti, K. T. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *FPGA*, pages 245–254, 2013.
[6] G. C. Chow et al. An efficient sparse conjugate gradient solver using a benes permutation network. In *FPL*, pages 151–160, 2014.
[7] E. S. Chung et al. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *FPGA*, pages 97–106, 2011.
[8] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1, 2011.
[9] E. Hung et al. Escaping the academic sandbox: Realizing VPR circuits on xilinx devices. In *FCCM*, pages 45–52, 2013.
[10] D. Koch and J. Torresen. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *FPGA*, pages 45–54, 2011.
[11] K. T. Lim et al. Thin servers with smart pipes: designing SoC accelerators for memcached. In *ISCA*, pages 36–47, 2013.
[12] E. Mirsky and A. DeHon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, pages 157–166, 1996.
[13] K. E. Murray et al. Titan: Enabling large and complex benchmarks in academic CAD. In *FPL*, pages 1–8, 2013.
[14] S. Nalam and B. H. Calhoun. Asymmetric sizing in a 45nm 5t SRAM to improve read stability over 6t. In *CICC*, pages 709–712, 2009.
[15] X. Niu et al. Exploiting run-time reconfiguration in stencil computation. In *FPL*, pages 173–180, 2012.
[16] I. Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992.
[17] L.-N. Pouchet et al. Polyhedral-based data reuse optimization for configurable computing. In *FPGA*, pages 29–38, 2013.
[18] J. Rose et al. The VTR project: architecture and CAD for FPGAs from verilog to routing. In *FPGA*, pages 77–86, 2012.
[19] Tabula. Tabula corporate backgrounder. `http://www.tabula.com/about/Tabula_CorpBackgrounder4_13.pdf`.
[20] E. Tau et al. A first generation DPGA implementation. In *FPD*, pages 138–143, 1995.
[21] D. B. Thomas and W. Luk. Credit risk modelling using hardware accelerated monte-carlo simulation. In *FCCM*, pages 229–238, 2008.
[22] S. Trimberger et al. A time-multiplexed FPGA. In *FCCM*, pages 22–29, 1997.
[23] D. Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.
[24] Xilinx. 7 series FPGAs configuration user guide. `http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf`.
[25] Xilinx. LogiCORE IP AXI HWICAP (v2.01.a). `http://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v2_01_a/ds817_axi_hwicap.pdf`.
[26] Xilinx. Virtex-6 FPGA configurable logic block. `http://www.xilinx.com/support/documentation/user_guides/ug364.pdf`.
[27] S. Young et al. A high I/O reconfigurable crossbar switch. In *FCCM*, pages 3–10, 2003.
[28] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *FPGA*, pages 63–74, 2005.