Adaptive In-Cache Streaming for Efficient Data Management

Nuno Neves, *Student Member, IEEE*, Pedro Tomás, *Member, IEEE*, and Nuno Roma, *Senior Member, IEEE*

***Abstract*— The design of adaptive architectures is frequently focused on the sole adaptation of the processing blocks, often neglecting the power/performance impact of data transfers and data indexing in the memory subsystem. In particular, conventional address-based models, supported on cache structures to mitigate the memory wall problem, often struggle when dealing with memory-bound applications or arbitrarily complex data patterns that can be hardly captured by prefetching mechanisms. Stream-based techniques have proven to efficiently tackle such limitations, although not well-suited to handle all types of applications. To mitigate the limitations of both communication paradigms, an efficient unification is herein proposed, by means of a novel in-cache stream paradigm, capable of seamlessly adapting the communication between the address-based and stream-based models. The proposed morphable infrastructure relies on a new dynamic descriptor graph specification, capable of handling regular arbitrarily complex data patterns, which is able to improve the main memory bandwidth utilization through data reutilization and reorganization techniques. When compared with state-of-the-art solutions, the proposed structure offers higher address generation efficiency and achievable memory throughputs, and a significant reduction of the amount of data transfers and main memory accesses, resulting on average in 13 times system performance speedup and in 245 times energy-delay product improvement, when compared with the previous implementations.**

***Index Terms*— Adaptive communication, energy-efficient data-management, morphable architectures, stream-based prefetching.**

I. Introduction

THE ever increasing demand for computational processing power at a significantly lower energy consumption has pushed the research for alternative heterogeneous and often specialized high-performance many-core processing architectures. In particular, to avoid the inherent disadvantages of power-hungry high-end general purpose processing systems, adaptive processing architectures have gradually been adopted to handle broader ranges of applications and workloads. Specifically, they try to extract or predict the characteristics of a running application by adapting their own processing scheme at runtime (or even their architecture, through dynamic reconfiguration), in order to make the execution as efficient as possible, both in terms of performance and energy consumption [1]–[3]. However, the design and runtime adaptation of such architectures is usually focused on the processing blocks, often neglecting the power/performance impact of the inherent data transfers and general data indexing in the memory subsystem, which usually remain nonadaptive. In fact, a common approach is to simply rely on conventional local (and often multilevel) cache structures to avoid the usually high memory access latencies. However, as the number of cores on a cache coherent system increases, the inherent contention (particularly in memory-bound applications) and the overall energy consumption tend to increase, and can even reach a point where the addition of more cores is no longer useful [4], hence limiting the system's achievable performance. Moreover, such structures often struggle when the application data set is very large and does not fit in the cache hierarchy, or in the presence of complex memory access patterns, where data locality cannot be efficiently exploited.

Nevertheless, some examples of adaptive schemes specifically targeting energy efficiency in the communication infrastructure have already been deployed, based on either dynamic reconfiguration processes [5], [6] or power-gated designs and/or hybrid memory technologies [7]. However, such strategies mostly target the efficient utilization of the memory resources, to reduce the energy consumption through architectural adaptation. As a consequence, inevitable delays in the reconfiguration and gating procedures are still incurred by these solutions. Moreover, they do not address the data communication itself, resulting in memory access structures that still rely on conventional cache-based accesses.

To tackle such problem, different techniques can be exploited in order to adapt the data communication scheme to the running application. In particular, efficient prefetching techniques have been reemerging [8]–[10], often based on the combination of several different approaches that can handle a variety of memory-intensive applications. However, there are still several issues that must be dealt with, particularly those related to prediction overheads and inaccuracy penalties that can result in added pressure and increased traffic in the memory subsystem, especially when dealing with complex memory access patterns. Conversely, *stream-based* communication schemes have been regarded as a viable alternative to conventional cache-based systems and *address-based* prefetchers. Although they have been traditionally suited for particular sets of applications, they can be easily adapted to reduce the communication contention and to manage large data sets in broader ranges of regular applications [11]. Accordingly, instead of having each system's processing element (PE) (and/or associated prefetchers) to concurrently perform main memory requests, these approaches explicitly decouple the communication and processing infrastructures, making it possible to

Manuscript received August 12, 2016; revised December 4, 2016; accepted January 29, 2017. Date of publication March 14, 2017; date of current version June 23, 2017. This work was supported by national funds through Fundação para a Ciência e a Tecnologia under Project UID/CEC/50021/2013 and Grant SFRH/BD/100697/2014.

The authors are with INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisbon, Portugal (e-mail: nuno.neves@inesc-id.pt; pedro.tomas@inesc-id.pt; nuno.roma@inesc-id.pt).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2671405

hide data transfers behind computations. Moreover, streaming approaches are capable of independently handling complex data patterns, thus exploiting the data access pattern of each processing block to smoothly and transparently buffer and stream (eventually even broadcasting) the data to the corresponding PEs [11]–[13].

However, accurate memory access pattern descriptions are still limited to applications with deterministic memory access patterns. As such, purely stream-based infrastructures can hardly deal with particular types of applications (such as those fully or partially supported by pointer-based data structures or dynamic indexing procedures), where the memory access is either nondeterministic or it is generated at runtime.

Accordingly, a new in-cache stream communication paradigm for many-core heterogeneous systems is herein proposed. The considered methodology combines a conventional cache-based memory hierarchy approach with efficient stream-based prefetching techniques. This hybrid paradigm is deployed in a single and adaptable infrastructure that is capable of in-time switching its communication paradigm to better suit a running application or its evolving requirements. As a result, it is capable of combining the different advantages that are offered by each paradigm (including exploitation of data locality, stream prefetching and manipulation, and complex memory access generation) in a single structure, while mitigating their individual drawbacks. In accordance, the main contributions of the proposed morphable infrastructure are as follows.

1) An in-cache streaming communication model supporting not only *memory-addressed* and *packed-stream* data accesses, but also an adaptive mode combining the *memory-addressed/packed-stream* schemes, especially suited for applications composed of compile-time predictable streams, while still supporting nonpredictable memory access patterns. The proposed communication paradigm makes use of a previously proposed in-cache stream controller architecture [14], specially devised to take the advantage of a conventional n-way set-associative cache memory, by making each way individually usable as a stream buffer.
2) A new data-pattern dynamic descriptor specification (extended from [13]), capable of efficiently encoding highly complex memory access patterns, through improved description flexibility and code memory utilization, when compared with the current state-of-the-art approaches [11]–[13].
3) A pattern description controller (PDC) architecture, composed of a specially devised address generation unit (AGU), capable of handling and generating complex data indexing based on the proposed descriptor specification.
4) Efficient memory access optimization techniques (i.e., bandwidth optimization, data reorganization and reutilization, and in-time stream manipulation), deployed by a memory-aware stream management controller (SMC), composed of a dedicated burst controller and stream reorder buffers.

The proposed in-cache stream communication model was prototyped on a fully functional morphable infrastructure, implemented on a state-of-the-art field-programmable gate array (FPGA) device. Its address generation efficiency was compared with state-of-the-art data-fetch and prefetching architectures, and its capabilities for prefetching and data reutilization were demonstrated through the experimental evaluation of several benchmark applications. The obtained results also show performance increases achieved by the proposed system, averaging 126.7 times when compared with a cache-based conventional setup, with system configurations of up to 64-PE. Such a gain corresponds to a 13 times performance improvement over a previous hybrid implementation [14], also revealing 91% of average total energy savings and 245 times of performance-energy efficiency improvements.

The remainder of this paper is organized as follows. In Section II, the advantages and characteristics of the proposed approach are discussed and compared with the current state-of-the-art in adaptive data-management systems, prefetching and stream-based communication. The proposed in-cache stream communication model and an overview of the devised morphable communication infrastructure are presented in Section III. Section IV describes the proposed dynamic descriptor graph specification and its potential for data-management applications. In Section V, it is detailed the proposed memory-aware SMC and its components. Section VI presents the obtained experimental results, concerning the measured data-pattern generation efficiency and achievable main memory throughput, as well as a thorough performance and energy efficiency evaluation. This paper is concluded in Section VII, by addressing the main contributions and achievements.

II. Related Work

Several different communication paradigms have been adopted to minimize the power/performance impact of the data-management subsystem [10], [12], [15]. In particular, contrasting to straightforward and aggressive *address-based* prefetch schemes, often associated with high-energy consuming memory/cache hierarchies [8], more efficient and sophisticated *stream-based* communication schemes have been deployed. In fact, by relying on streaming approaches, hence decoupling the data communication from the processing structures, it is possible to hide the data transfers overhead behind computation by using intermediate (small) buffering memories, which can be independently preloaded by the data streaming structures. This approach presents a rather advantageous outcome since, even without directly relying on aggressive *address-based* prefetching schemes and structures, it is still possible to handle complex data patterns [10] or reduce the energy consumption of the data-management infrastructure [8].

A. Advanced Prefetching Techniques

To reduce the main memory access concurrence and contention of conventional cache-based systems and shared communication infrastructures, advanced prefetching techniques [8], [13], [16] are often considered. Such techniques are usually designed to deal with the intrinsic characteristics of certain types of applications, such as reduced data locality and complex memory access patterns, memory-bound kernels or very large data sets that do fit in the local memories.

The most commonly used prefetching techniques usually rely on complex dedicated modules, which dynamically analyze the most recent memory access patterns and try to predict future accesses based on prediction heuristics. In particular, Ishii *et al.* [9] propose the access map pattern matching (AMPM) stride prefetcher to identify hot zones in memory and store a bitmap to infer strided patterns in the access stream. It showed to be able to efficiently detect regular memory accesses, independently of the order in which they are observed, providing a high prefetching coverage. Other examples include global history buffers [17], irregular access structural mapping [10], and delta-prefetching [16]. However, although such approaches allow a complete abstraction of the prefetching procedure from the application perspective, they fall short when the application is characterized by complex memory access patterns. Moreover, they impose an increased amount of resources, often related with the adopted level of prefetching aggressiveness [8].

A viable alternative that has been considered [18] relies on compiler-aided approaches, where the compiler preanalyzes the code and tries to extract/model the application memory access pattern and feeds it to on-chip prefetching modules. This results in far simpler hardware structures, since no on-time analysis is performed, in turn resulting in lower footprint and more energy efficient controllers, at the cost of an increased preprocessing effort. Such an approach also promotes the exploitation of efficient stream-based communication means, since the required information for memory data-stream generation can be created with compile-time analysis information and/or through explicit data-pattern programming [11], [13].

B. Stream-Based Communication Architectures

Regarding the stream-based approaches, several techniques have been proposed to improve the throughput of the data streaming and management infrastructure. Park and Diniz [15], [19] tackled the problem concerning the data fetching from an external memory to an FPGA in the context of stream-computing, acknowledging that proper data reutilization mechanisms are fundamental in FPGA-based systems. Meanwhile, Hussain *et al.* [12] proposed an advanced pattern-based memory controller (APMC) that supports up to 3-D regular data-fetching mechanisms, such as scatter-gather and strided accesses with programmable tiling. However, while the APMC represents a step forward toward the streaming of complex patterns, it was designed for moving large and regular data-chunks and falls short with irregular or complex memory indexing. In fact, irregular and pointer-based accesses are managed through runtime memory access analysis and recording.

To address the issue of complex data-pattern generation and efficient data manipulation schemes, the Hotstream framework [11] relies on PE-coupled dedicated pattern-programmable data-fetch controllers (DFCs). Nevertheless, although the considered programmable approach eases the description of complex, but still regular, data patterns, it also struggled with many complex data access patterns.

To tackle such patterns, a tree-based 3-D data-pattern descriptor specification was proposed in [13]. Such specification relies on a set of comprehensive data-pattern descriptors, organized in a treelike hierarchical fashion, which significantly eases the description of complex memory access patterns. To decode such descriptors, a descriptor tree controller (DTC) architecture was proposed that is capable of efficiently handling the memory address generation and data indexing. The proposed specification was also used to deploy different stream manipulation operations, such as stream splitting and merging, seeking an improvement of the communication efficiency by promoting data reutilization and reorganization.

However, although it proved efficient in resolving complex data patterns and stream manipulations, ultimately reducing the number of main memory accesses and communication-related energy consumption, the specification from [13] requires further improvements. In fact, it still imposes important drawbacks in what concerns the pattern description code size when the complexity of the pattern increases, which may limit the complexity of the described pattern.

C. Morphable Communication Systems

Several established processing strategies are aided by the dynamic reconfiguration capabilities of nowadays FPGA devices, where the processing infrastructures can adapt to the target application by reconfiguring its PEs at runtime [1]–[3]. However, the communication infrastructure is usually kept with nonreconfigurable and generic structures (e.g., network on chip or shared-bus structures). This is often because the reconfiguration process still results in nonnegligible time overheads and power dissipation that can critically impact the performance and energy consumption of the communication infrastructure.

Nonetheless, although still incurring in inevitable delays in the reconfiguration process, there are such cases where energy efficiency has been targeted with the adaptation of the communication subsystem (e.g., cache structures, local memories, and network/bus topologies). As an example, Sundararajan *et al.* [5] proposed a cache architecture that allows the dynamic reconfiguration of both its size and associativity organization, whose best configuration for a given application is dynamically predicted with the aid of a decision tree machine learning model. In [20], it is proposed a v-set cache design, targeting an adaptive and dynamic utilization cache blocks, for shared last-level caches in multicore environments. A reconfigurable cache design is also proposed in [7], where different memory technologies [SRAM and nonvolatile memory (NVM)] are unified at the same cache level to form a hybrid design. Moreover, power gating circuitry is also introduced to allow an adaptive powering of SRAM/NVM subarrays at each way level. To increase a multicore processor cache performance, in [6], it is proposed an architecture comprising a local scratchpad memory that can be partly

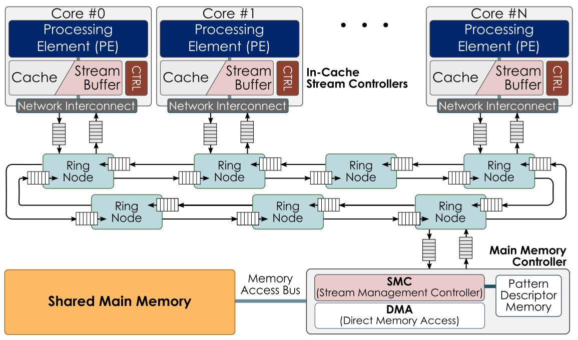
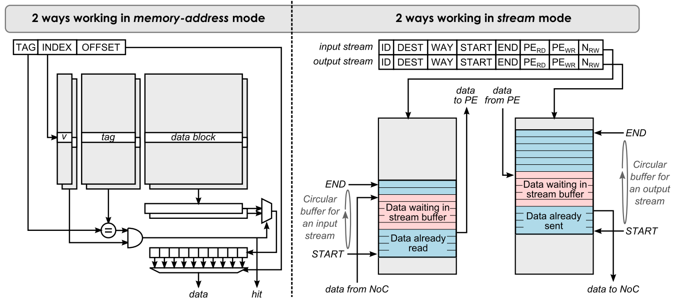


Fig. 1. Morphable communication infrastructure overview, comprising the in-cache stream controllers at the PEs interface, the main memory controller, and a ring-based interconnection.



2 ways working in memory-address mode
2 ways working in stream mode
TAG INDEX OFFSET
V tag data block
data hit
input stream
output stream
ID DEST WAY START END PE_RD PE_WR N_RW
data to PE
data from PE
END
START
Circular buffer for an input stream
Circular buffer for an output stream
Data waiting in stream buffer
Data already read
Data already sent
data from NoC
data to NoC

Fig. 2. Example configuration of a four-way set-associative memory after a control switch, becoming configured to use two ways for conventional memory-address mode and two ways for stream mode.

configured at runtime, to operate as a local second level cache, providing a unified hardware support for both implicit and explicit communication.

III. In-Cache Stream Communication Paradigm

The proposed approach differs from (and complements) other established strategies based on the sole exploitation of adaptable data-processing structures and targets the efficient runtime adaptation of an accelerator's communication scheme. In fact, even without using dynamic reconfiguration procedures, the herein proposed in-cache streaming paradigm allows a system to deploy a complete morphable communication infrastructure (from the main shared memory level up to the PEs local memory level). Hence, it is able to seamless switch and share its communication paradigm between conventional *memory-addressed* and *packed-stream* data accesses (see Fig. 1). In order to do so, the proposed approach allows each PE to view a local n-way set-associative cache memory as a set of n_1 cache ways plus $n - n_1$ stream buffers, each capable of holding multiple streams (as shown in the example of Fig. 2).

Such a convenient and dynamic memory assignment, together with the adoption of a switched control structure, allows the communication infrastructure to instantly adapt (no reconfiguration overheads are imposed) the cache memory according to the instantaneous requirements of the running application. Moreover, it also avoids a complete switching of the two paradigms, which could otherwise result in potential performance penalties in nonpure streaming applications.

The exploitation of such a morphable structure was preliminarily discussed in [14], where it was proposed a specially devised cache controller that supports both paradigms, allied with mixed scenarios composed of compile-time predictable and nonpredictable/runtime generated memory access patterns.

Such an in-cache stream communication model is herein significantly extended and even complemented with a specially devised memory-aware SMC (see Section V), that deploys a set of memory bandwidth optimization and data reutilization and reorganization techniques, through in-time stream manipulation. In fact, the proposed infrastructure is based on a data-pattern dynamic descriptor graph specification (see Section IV), capable of efficiently encoding highly complex memory access patterns and stream indexing.

A. Morphable In-Cache Stream Infrastructure Overview

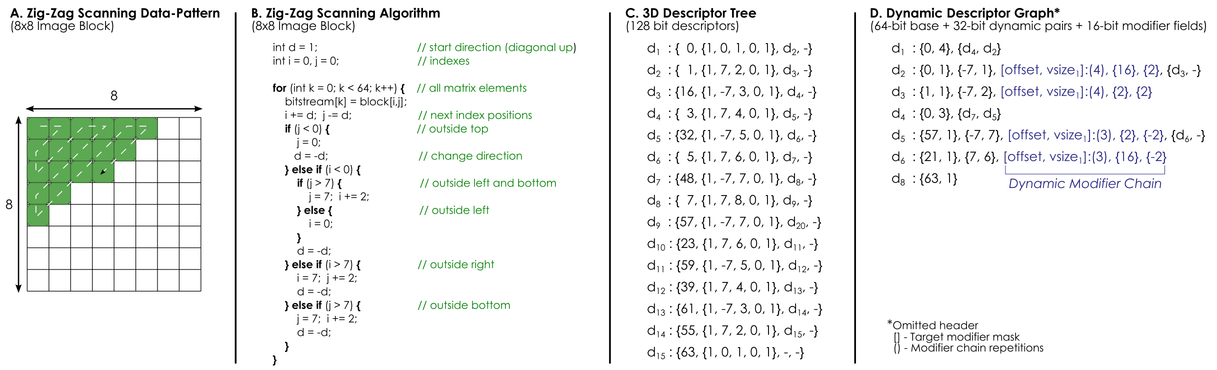
At a system level, the proposed in-cache stream communication paradigm is deployed by a complete morphable communication infrastructure (shown in Fig. 1) that is composed of: 1) PE-coupled in-cache stream controllers [14]; 2) a low-profile interconnection, supported by a message-passing protocol, with point-to-point and broadcast data transfer capabilities; and 3) a hybrid main memory controller, comprising a *memory-addressed* direct memory access (DMA) module and a *packed-stream* memory-aware SMC.

Although the communication between the main memory controller and the in-cache stream controllers can be implemented by any type of interconnection, it should not only assure point-to-point communication between all its connected components, but also support broadcast data transfer capabilities, which have been proven to be essential for efficient stream communication [13], [14]. Both the in-cache stream controllers and the hybrid main memory controller interface with the communication infrastructure by means of two input/output register-based buffers, allowing to mitigate the memory access contention through intermediate buffering.

In the particular case of the proposed infrastructure, it was considered a bidirectional ring-based topology to ensure an efficient and low-profile interconnection architecture, completely abstracted from the remaining morphable infrastructure. This, in turn, results in an insignificant impact on the performance of the intercommunication between the system components. Hence, each node is able to route the incoming messages to/from its two adjacent nodes (right and left) and to/from its connected component. To overcome the contention caused by simultaneously arriving packets, a simple round-robin priority function was devised that rotates the priority between channels upon the completion of a message transmission.

B. Set-Associative Cache Memory Hybridization

To adapt a n-way set-associative memory, to support the proposed communication paradigm, the cache memory is simultaneously managed by two independent modules [14]: a *hybrid cache controller* and a *stream controller*. The default *memory-addressed* communication paradigm is assured by the *cache controller*, by using any arbitrarily replacement



B. Zig-Zag Scanning Algorithm
(8x8 Image Block)

```
int d = 1;                          // start direction (diagonal up)
int i = 0, j = 0;                   // indexes

for (int k = 0; k < 64; k++) {      // all matrix elements
    bitstream[k] = block[i,j];
    i += d;  j -= d;                // next index positions
    if (j < 0) {                    // outside top
        j = 0;
        d = -d;                     // change direction
    } else if (i < 0) {
        if (j > 7) {                // outside left and bottom
            j = 7;  i += 2;
        } else {                    // outside left
            i = 0;
        }
        d = -d;
    } else if (i > 7) {             // outside right
        i = 7;  j += 2;
        d = -d;
    } else if (j > 7) {             // outside bottom
        j = 7;  i += 2;
        d = -d;
    }
}
```

C. 3D Descriptor Tree
(128 bit descriptors)

d_1 : { 0, {1, 0, 1, 0, 1}, d_2, -}
d_2 : { 1, {1, 7, 2, 0, 1}, d_3, -}
d_3 : {16, {1, -7, 3, 0, 1}, d_4, -}
d_4 : { 3, {1, 7, 4, 0, 1}, d_5, -}
d_5 : {32, {1, -7, 5, 0, 1}, d_6, -}
d_6 : { 5, {1, 7, 6, 0, 1}, d_7, -}
d_7 : {48, {1, -7, 7, 0, 1}, d_8, -}
d_8 : { 7, {1, 7, 8, 0, 1}, d_9, -}
d_9 : {57, {1, -7, 7, 0, 1}, d_{20}, -}
d_{10} : {23, {1, 7, 6, 0, 1}, d_{11}, -}
d_{11} : {59, {1, -7, 5, 0, 1}, d_{12}, -}
d_{12} : {39, {1, 7, 4, 0, 1}, d_{13}, -}
d_{13} : {61, {1, -7, 3, 0, 1}, d_{14}, -}
d_{14} : {55, {1, 7, 2, 0, 1}, d_{15}, -}
d_{15} : {63, {1, 0, 1, 0, 1}, -, -}

D. Dynamic Descriptor Graph*
(64-bit base + 32-bit dynamic pairs + 16-bit modifier fields)

d_1 : {0, 4}, {d_4, d_2}
d_2 : {0, 1}, {-7, 1}, [offset, vsize$_1$]:(4), {16}, {2}, {d_3, -}
d_3 : {1, 1}, {-7, 2}, [offset, vsize$_1$]:(4), {2}, {2}
d_4 : {0, 3}, {d_7, d_5}
d_5 : {57, 1}, {-7, 7}, [offset, vsize$_1$]:(3), {2}, {-2}, {d_6, -}
d_6 : {21, 1}, {7, 6}, $\underbrace{\text{[offset, vsize}_1\text{]:(3), \{16\}, \{-2\}}}_{\textit{Dynamic Modifier Chain}}$
d_8 : {63, 1}

*Omitted header
[] - Target modifier mask
() - Modifier chain repetitions

Fig. 3. Data pattern description of a zig-zag scanning of an 8 × 8 block, used in the entropy encoding step of an image encoder.

and write policies. On the other hand, a specific SMC is used to conveniently adapt and reuse the resources of the n-way set-associative cache memory to implement the buffering structures required by a *packed-stream* communication.

Hence, each associative way is regarded as an independent buffering structure that is accessed with a dedicated set of read and write pointers that identify the memory region where a stream is stored. This transforms the n-way set-associative memory in m independent stream buffers, each capable of storing multiple streams, while allowing the remaining $n-m$ ways to be accessed using traditional *memory-address* load/store operations (see Fig. 2). This stream-based paradigm requires a set of supporting data structures (stored in a programmable *stream table*), to accommodate the information and the state of every stream currently stored and handled by the controller.

At any instant, the memory configuration is defined by a simple register, shared by both controllers, indicating which ways are accessible (or owned) by each controller. The register can be modified either: 1) explicitly, by the cache's PE; 2) externally, by a central system manager (if available); or 3) implicitly, by the configuration of the stream table. Accordingly, the memory is configured by default with each way owned by the *cache controller*, and, upon request, the ownership of individual ways can be changed to the *steam controller*. The control switch for each way is performed transparently from each of the controllers, eliminating any unnecessary waiting times that could otherwise degrade performance.

In the *steam controller*, whenever a request is performed for a given stream identifier (see Fig. 2), the local memory is accessed according to the information depicted in the stream table, with the consequent update of its read/write pointers. Outgoing streams are automatically sent as soon as they become available and their transmission is granted by the scheduling manager of the processor aggregate. Moreover, upon their transmission, stream data can either be immediately erased from the local memory or made persistent (and later explicitly erased), allowing the data to be reused by the PE.

IV. Modeling of Complex Data Patterns

Complex memory access patterns are usually indexed by making use of several control primitives and nontrivial address computations. As a result, the required amount of control steps to generate these patterns (in a PE) presents a considerable overhead to what should, in most cases, be reduced to a straightforward set of operations. One example of such patterns is the one that is required to implement the entropy encoding algorithm used in image processing applications. To encode a quantized image block, its pixels are first divided into blocks and subsequently scanned in a zig-zag pattern [shown Fig. 3(A)]. The generation of such pattern (including all the necessary control steps) in a PE imposes a significant amount of overhead, as shown in Fig. 3(B). However, such overhead can be significantly reduced with the adoption of pattern generation controllers [13] [see Fig. 3(C)], to alleviate the PE from the need to compute the required memory addresses. Moreover, when combined with a stream-based communication scheme, the data transfer procedure can be completely detached from the PE execution, which in turn can lead to an increased system throughput.

A. Efficient Data Indexing and Manipulation

The proposed dynamic descriptor graph specification follows the general principle that, independently of their application domain, many algorithms are characterized by complex memory access patterns that can be represented by an n-dimensional (n-D) affine function [21] (or by a set of such functions). The memory address (y) is calculated by considering an initial *offset*, a base increment variable x_0, and pairs of increment variables x_k and *stride*$_k$ multiplication factors

$$y(X) = \text{offset} + x_0 + \sum_{k=1}^{n} x_k \times \text{stride}_k, \quad x_k \in \{0, \ldots, \text{size}_k\}. \tag{1}$$

Since such a representation allows indexing a significant amount of regular access patterns, this particular specification is commonly used by DMA controllers and other similar DFCs, although typically restricted to 2-D patterns ($n = 2$). However, affine functions with a higher dimensionality can also be used, which can be described by relying on specially devised instruction-set architectures [11] or by using descriptor-based approaches [12], [13]. Furthermore, even more complex memory access patterns can still be described by hierarchically combining multiple $n-D$ functions. In particular, several functions can be chained together by using affine functions in the higher levels of the hierarchy to calculate the initial *offset*, the increment variables or the multiplication factors of the functions in the lower levels. Hence, each data stream can be defined by a set of $n-D$ functions combined in a given hierarchical structure, each encapsulating the set of parameters required to generate the correct sequence of addresses.

B. Tree-Based 3-D Descriptor Limitations

The 3-D data-pattern descriptor tree representation previously proposed in [13] was specially devised to tackle the representation of complex memory access patterns, since the commonly used 2-D descriptor representation struggles when trying to describe higher levels of pattern complexity (e.g., diagonal, zig-zag, and diamond patterns). However, this requires very large descriptor lists and a very complex control, which not only increases the size of the descriptors, but also the hardware structures that support it [11]. Moreover, there is usually a certain amount of overhead associated with the switching between different descriptors, which degrades the rate and efficiency of the memory address generation procedures.

In fact, although it proved to be a valuable improvement (regarding address generation), the 3-D descriptor tree specification proposed in [13] still lacks scalability in terms of pattern encoding and descriptor code size. Not only is the 3-D base descriptor underused when the most common and simple data patterns (such as arrays and matrices) are described, but the double-pointer treelike hierarchical organization, by itself, cannot entirely exploit certain levels of regularity in the described pattern. As an example, by considering the previous zig-zag pattern, it can be observed that although it presents a certain regularity in the scanning path [see Fig. 3(A)], its descriptor is hardly encoded in an efficient manner. To describe such a pattern, the 3-D descriptor tree specification [13] requires a different descriptor for each of the pattern's diagonals, thus not exploiting the pattern's inherently regular characteristics [see Fig. 3(C), where 15 128-bit descriptors must be used to correctly encode the data pattern].

C. Dynamic Descriptor Graph Specification

The new descriptor definition that is herein proposed (specified in Fig. 4) extends the previously proposed specification [13] by significantly improving its scalability and coding efficiency. This base descriptor defines an unidimensional data access pattern by means of: 1) an `header` field, containing the descriptor configuration (discussed in the following); 2) an `offset` field, specifying the starting address of the first memory position; and 3) an `hsize` field, indicating the size of a contiguous block of memory accesses. However, in order to provide support for multidimensional memory accesses, additional fields can be added to the base descriptor. Hence, for each additional dimension, an extra tuple (hereafter referred to as a *dynamic pair*) is subsequently added to the descriptor, composed of: 1) a `stride` field, with the starting position of the next contiguous block (with relation to the previous one) and 2) a `vsize` field, indicating the number of repetitions of all the previous parameters of the descriptor.

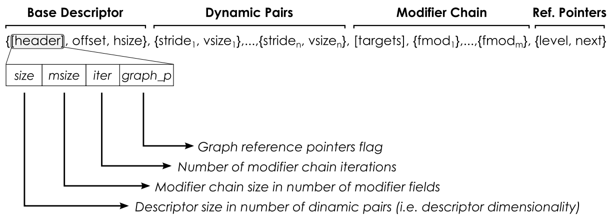

Base Descriptor
Dynamic Pairs
Modifier Chain
Ref. Pointers
{[header], offset, hsize}, {stride$_1$, vsize$_1$},...,{stride$_n$, vsize$_n$}, [targets], {fmod$_1$},...,{fmod$_m$}, {level, next}
size
msize
iter
graph_p
Graph reference pointers flag
Number of modifier chain iterations
Modifier chain size in number of modifier fields
Descriptor size in number of dinamic pairs (i.e. descriptor dimensionality)

Fig. 4. Dynamic Descriptor Graph Definition.

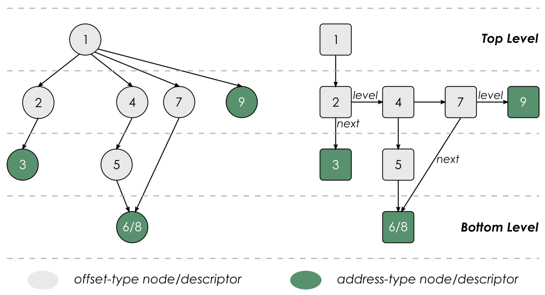
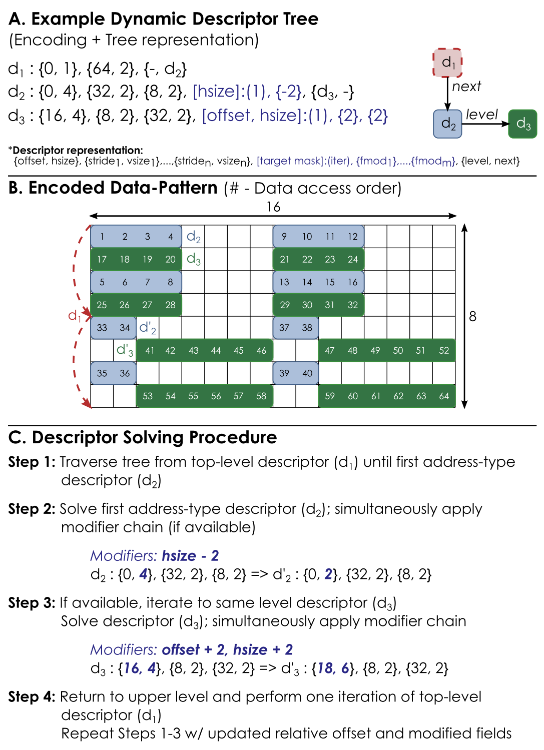


Fig. 5. Adopted graph-based hierarchical descriptor organization, showing both the graph representation (left) and the corresponding referencing between the descriptors (right). The number associated with each node represents the order in which they are iterated. Note that the descriptor at the bottom level is referenced twice; hence, it is solved with two different offset patterns.

Furthermore, to exploit the inherent regularity of the most complex data patterns (without requiring very large descriptor graphs), each field can be made modifiable upon its completion. As such, besides the introduced *dynamic pairs*, each descriptor also contains an optional *modifier chain*, composed of: 1) a `target mask` field, indicating the fields of the base and dynamic pairs changeable by the *modifier chain* and 2) a variable number of `fmod` fields, indicating the value used to modify the corresponding target field.

Moreover, the proposed descriptor organization relies on a graph-based hierarchical scheme, as shown in Fig. 5. Accordingly, multiple parent–child relations can be established between descriptors, representing dependencies between different descriptors. For such purpose, descriptors are distinguished between *offset*-type descriptors (used to calculate relative offsets) and *address*-type descriptors, which generate memory addresses based on such relative offsets. Moreover, *address*-type descriptors are characterized by not having any depending-child descriptors in the hierarchy.

To deploy a graphlike hierarchy, a double-referencing representation (typically employed to represent graphs in programming languages) was used. Hence, each descriptor has optional references to a child descriptor (`next`) and to a descriptor that shares the same parent descriptor (`level`). This way, a descriptor with multiple child descriptors only references one of them, which in turn establishes a reference chain that includes all its siblings (see Fig. 5). By adopting such a representation, the addressed positions are resolved by traversing this graph structure in child-priority order (see Fig. 5). Hence, for each iteration of a given descriptor,


A. Example Dynamic Descriptor Tree
(Encoding + Tree representation)

d_1 : {0, 1}, {64, 2}, {-, d_2}
d_2 : {0, 4}, {32, 2}, {8, 2}, [hsize]:(1), {-2}, {d_3, -}
d_3 : {16, 4}, {8, 2}, {32, 2}, [offset, hsize]:(1), {2}, {2}

*Descriptor representation:
{offset, hsize}, {stride_1, vsize_1},...,{stride_n, vsize_n}, [target mask]:(iter), {fmod_1},...,{fmod_m}, {level, next}

B. Encoded Data-Pattern (# - Data access order)

C. Descriptor Solving Procedure

Step 1: Traverse tree from top-level descriptor (d_1) until first address-type descriptor (d_2)

Step 2: Solve first address-type descriptor (d_2); simultaneously apply modifier chain (if available)

Modifiers: ***hsize - 2***
d_2 : {0, **4**}, {32, 2}, {8, 2} => d'_2 : {0, **2**}, {32, 2}, {8, 2}

Step 3: If available, iterate to same level descriptor (d_3)
Solve descriptor (d_3); simultaneously apply modifier chain

Modifiers: ***offset + 2, hsize + 2***
d_3 : {***16, 4***}, {8, 2}, {32, 2} => d'_3 : {***18, 6***}, {8, 2}, {32, 2}

Step 4: Return to upper level and perform one iteration of top-level descriptor (d_1)
Repeat Steps 1-3 w/ updated relative offset and modified fields

Fig. 6. Example of a data-pattern dynamic descriptor graph (A). Note the order in which the blocks are accessed in (B). d_1 calculates a relative offset for both d_2 and d_3, which, accordingly, generate two distinct patterns. Note that the patterns generated by d_2 and d_3 are modified after a complete iteration, which are performed according to their *modifier chain* (C).

its child descriptor (referenced by the `next` field) should be completely solved once. Subsequently, when the first child is itself completely solved, the procedure will solve the following sibling descriptor that shares the same parent descriptor (referenced by the `level` field).

To configure each descriptor, in what concerns its parameterization and dimensionality, the base descriptor tuple contains a convenient `header` field that encodes the following variables: 1) a *size* variable, used to configure the dimensionality of the descriptor (i.e., the number of $\{\texttt{stride}_k, \texttt{vsize}_k\}$ pairs of the descriptor); 2) a *msize* variable, indicating the size of the descriptor's modifier chain (i.e., the number of $\{\texttt{fmod}_m\}$ fields); 3) an *iter* value, indicating the number of repetitions of the modifier chain over the target fields of the base and dynamic pairs; and 4) a $graph_p$ flag, indicating the presence of the {`level`, `next`} reference pointer pair.

In accordance, the proposed dynamic descriptor graph specification is defined as shown in Fig. 4. Fig. 6 shows its resolution procedure applied to an example descriptor graph.

Hence, with this approach, it is possible to achieve the envisaged pattern regularity while still targeting the aimed flexibility and code size reduction, as can be observed in its description of the adopted zig-zag pattern example [see Fig. 3(D)] when compared with the equivalent description code for the previously proposed 3-D descriptor tree specification [13] [see Fig. 3(C)]. Such advantages and the added capabilities are thoroughly discussed in Section VI and compared with other state-of-the-art pattern description solutions.

Although it is out of the scope of this paper, it is worth mentioning that several solutions are currently being considered in what concerns the automatic extraction of a given application memory access pattern and its encoding with the proposed descriptor specification. In fact, straightforward solutions can either rely on the explicit encoding of patterns with high-level APIs [11], or on using extensions to parallel programming languages in the form of code annotations and dedicated compiler directives. On the other hand, considering the deterministic nature of the proposed specification's mathematical representation, more ambitious approaches can be considered, such as the extraction of memory access patterns during compilation. In particular, prominent compiler methods, such as polyhedral analysis [22], [23] or memory access profiling [24], can be conveniently explored and adapted to analyze a given application/kernel data indexing pattern and automatically generate the corresponding descriptor encoding.

V. Memory-Aware Data-Stream Generation

In a stream-based communication environment, each component must be capable of generating/processing its own data streams. Although each PE can manage the flow of its data streams with the aid of the in-cache stream controllers, streams fetched from the main memory require a dedicated structure to handle the translation between a *memory-addressed* access and a *stream*-based communication. This is accomplished through a dedicated main memory controller, composed of: 1) a low-profile address-based DMA controller, to perform address-based memory operations upon memory access requests performed by the PEs and 2) a memory-aware SMC, which automatically generates and saves the streams, respectively, by fetching/storing data from/to the main memory.

The SMC itself relies on a Pattern Description Controller (PDC) architecture (shown in Fig. 7 and detailed in Section V-A), which is able to generate memory access patterns according to the proposed dynamic descriptor graph specification (see Section IV). A dedicated burst controller was also devised to optimize the main memory access by taking advantage of its burst capabilities, as presented in Section V-B. The SMC operation in managed by a low-profile programmable controller, which is responsible for sequencing the required stream generation operations for each application under execution. The stream storing procedure, on the other hand, is automatically performed upon the reception of a stream from the communication infrastructure.

A. Pattern Description Controller Architecture

To efficiently index the memory access patterns described by the previously presented dynamic descriptor graph specification, the proposed PDC architecture was designed to use the least number of clock cycles (per memory address) as possible. For such purpose, the PDC architecture is conveniently divided into three parallel submodules, namely: 1) a graph solver unit (GSU), responsible for iterating over the descriptor graph; 2) an AGU, responsible for generating the memory addresses according to the described pattern and starting at the offset address defined by its parent descriptor; and 3) a dynamic chain unit (DCU), responsible for modifying a descriptor according to its modifier chain (if available).

The GSU communicates with the other units through a dedicated register bank, and all these units operate completely in

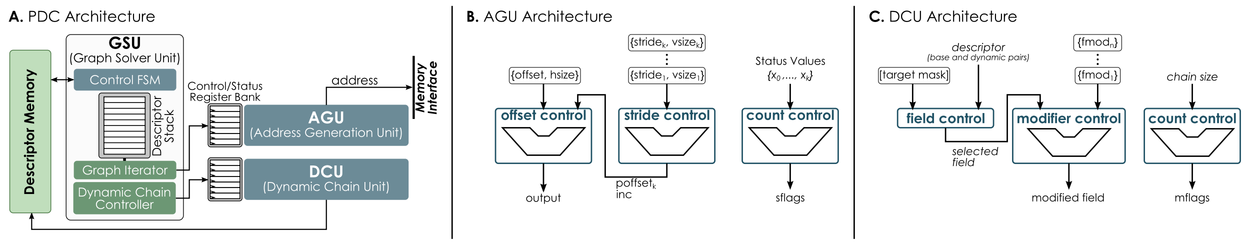


Fig. 7. Proposed pattern descriptor controller (PDC) and its functional units architecture.

parallel. Hence, while the AGU is executing a given *address*-type descriptor, the DCU can solve that descriptor's modifier chain and replace the modified fields in the original descriptor, while the GSU iterates over the graph to calculate the relative offset for a subsequent descriptor. A minimal local scratchpad memory is used to store the descriptors being processed.

The PDC's AGU and DCU functional units (shown in Fig. 7) iterate over a descriptor according to the addressing function defined in (1) and modify its fields each time it is solved, according to its respective *modifier chain*. To keep the architecture footprint as low as possible, it is solely based on adder elements and control and status registers. Both units are connected to register-based structures that store a single descriptor's parameters and its current iteration status.

Since the proposed dynamic descriptor graph specification relies on descriptors that can have a variable dimensionality, their resolving architecture must be able to efficiently iterate over each dimension without incurring in overheads, both in terms of required hardware structures and latency between generated output values. Hence, the proposed resolving architecture operates by iterating over a single dynamic pair, in parallel with the base descriptor {OFFSET, HSIZE} pair. A dedicated register bank (programmable by the GSU) is used to store each dynamic pair and its current iteration status. Moreover, a set of control registers is used to store the base descriptor configuration (indicated by the HEADER and OFFSET fields) and a set of control and status flags, used to select the correct sequence of dynamic pairs to be considered by the resolving architecture.

Accordingly, the AGU functional unit [see Fig. 7(B)] comprises three parallel functional blocks, each composed of an adder and specific operand selection and control logic. Depending on the currently selected dimension (k), the *stride control* block is responsible for incrementing an `inc` variable (representing the current contiguous data block) and for generating the required multiplication factors, by successively adding each dynamic pair $\texttt{stride}_k$ field to the current descriptor `offset` field. The resulting intermediate values are stored in a $\texttt{poffset}_k$ register. The *offset control* block calculates the output of the functional unit (the actual memory address) based on the $\texttt{poffset}_k$ value and either the `inc` or $\texttt{stride}_k$ variables, depending on the current descriptor state. The *count control* block is used to calculate the current descriptor iteration state, by incrementing the $\texttt{x}_0$ and $\texttt{x}_k$ values [refer to (1)], limited by the `hsize` and $\texttt{vsize}_k$ descriptor fields, respectively. In accordance, a set of control flags (`sflags`) is generated at the end of the logic path of the *count control* block to represent the iteration state of the descriptor. These flags are used to control all three functional blocks in terms of operand selection and reset logic. This way, each functional block performs one iteration per clock cycle, involving the computation of the current memory address, of the multiplication factors for the next iteration and of the next descriptor state, together with its corresponding control flags.

Similarly, the DCU functional unit [see Fig. 7(C)] is composed of two parallel functional blocks and a field selection block. This *field control* block is responsible for selecting the correct fields to be modified by iterating over each bit of the descriptor's modifier chain `target mask`. Each selected field (n), is sent to the *modifier control* block, which performs the correct modification operation by adding it with its corresponding $\texttt{fmod}_n$ value. The mask iteration procedure and the operation of the DCU are managed by the *count control* block, which receives the number of targets (size) in the descriptor's modifier chain and generates a set of control flags (`mflags`) by counting the number of iterations of the *field control* block.

The GSU control unit, represented in Fig. 7, is composed of: 1) a finite-state machine that deploys the descriptor graph solving procedure (see Section IV); 2) an iteration structure, composed by the same three-adder topology used in the AGU and DCU functional units, used to perform single iterations over the graph's *offset*-type descriptors; and 3) a descriptor stack, to store the descriptors' state. Specifically, the GSU is initiated upon the reception of an offset and a descriptor reference, iterating over the descriptor graph in the child-priority order described in Section IV. This way, *offset*-type descriptors are iterated once by the GSU and pushed into the stack as the GSU is going down through the graph's hierarchy. This approach significantly eases the operation of the GSU, since the order in which the descriptor states are pushed into the stack is the reverse order in which they are needed when the GSU is going up through the hierarchy.

B. Memory Burst Controller

Although highly efficient when dealing with fast-access local memories (e.g., BRAMs), AGUs often struggle to perform requests to long-latency external memories (e.g., DDR3 memories). This is mainly because these memories usually impose costly latency overheads (up to tens of clock cycles per request). To mitigate such delays and increase the throughput, most memories offer the ability to burst

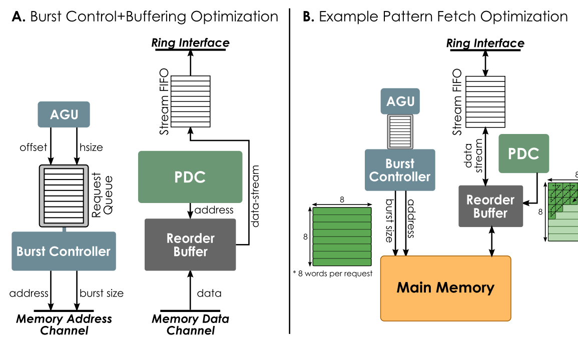


Fig. 8. Burst controller and reorder buffer optimization structures (A). The example in (B) shows the memory requests performed by the PDC for the zig-zag pattern descriptor depicted in Fig. 3 with proposed optimizations.

data transfers of contiguous memory regions. Accordingly, the herein proposed PDC's AGU is duplicated and coupled with a specially devised burst controller [see Fig. 8(A)] that generates and manages burst requests to the main memory, based on the descriptor being resolved.

However, instead of sending each address (indexed by a given descriptor), the AGU sends the descriptor's current `offset` and `hsize` fields to the burst controller, indicating a contiguous memory region to be accessed [see Fig 8(B)]. The burst controller is then responsible for splitting the request in one or more burst requests (depending on the maximum burst length supported by the memory and its communication protocol), which is performed by a very simple register-based increment/subtract architecture.

C. Reorder Buffer Optimization and Stream Manipulation

Although the proposed burst controller can increase the main memory throughput for most regular access patterns, dealing with complex patterns characterized by poor data locality is still a difficult challenge. One example is the previously described zig-zag pattern (shown in Fig. 3). Due to its diagonal scanning pattern, the direct application of the referred burst controller would not improve the resulting memory throughput. However, most of these complex patterns are usually contained in regular memory regions that can be prefetched from the main memory and temporarily buffered in fast-access memory structures. In particular, the adopted zig-zag pattern example is scanned in 8×8 data blocks and each block can be individually fetched and stored in a specially devised reorder buffer, which can then be accessed by the PDC with a zig-zag descriptor, as shown in Fig 8(B).

To offer such a functionality, each stream can be regarded as a block of contiguous data, where the data block is stored in memory address (offset) 0×0 [13]. By adopting such an analogy, it is possible to extract a specific data block from a previously prefetched data stream by simply adding a stream start pointer to an offset generated by the PDC. This allows applying the proposed descriptors to straightforwardly extract subpatterns from prefetched data streams. Furthermore, it is possible to easily apply runtime stream manipulation operations (e.g., stream splitting and merging) based on the proposed descriptor specification, as well as rerouting operations over the flowing streams between the PEs and memory.

To be properly accessed by the PDC, the specially devised reorder buffer is composed of a set of stream buffer banks, each able to store multiple streams. In fact, its control architecture is similar to that of the *stream controller* from the in-cache stream controller, where a *stream table* is used to maintain the information of the stored streams. Hence, the PDC is used to access the new reorder buffer (obtaining new data blocks as soon as they become available in the buffer), while the duplicated AGU is used to prefetch the data blocks from memory, aided by the burst controller [see Fig.8(A)].

VI. Experimental Evaluation

To evaluate the proposed morphable infrastructure, a prototype was implemented in a Xilinx VC707 development board, equipped with an XC7VX485T Virtex-7 FPGA and a 1-GB DDR3-1600 SODIMM memory module (MT8JTF12864HZ-1G6G1). The DDR3 module is accessed through a Xilinx IP MIG controller, comprising an AXI4-Full interface with a 20 clock cycle overhead per transfer request (independently of its length). The AXI4 protocol [25] allows burst transfers of up to 256 words and an attainable bus throughput of 400 MB/s per channel direction (32-bit words at 100 MHz), resulting in a maximum theoretical throughput of 800 MB/s. The Synthesis and Place&Route procedures were performed using Xilinx ISE 14.5. The power consumption of each of the system's components was estimated with the Xilinx Power Estimation toolchain and the DDR3 memory power consumption was calculated according to the vendor's guidelines and estimation tool [26]. Accurate clock cycle simulations were performed using Xilinx iSim simulator. The obtained results in what concerns the address generation and efficiency were compared with the most relevant related work, namely, with a Xilinx AXI DMA engine [27], whose functionalities are equivalent to those of the APMC [12], the Hotstream framework [11], and the previously proposed DTC [13]. All the proposed dynamic descriptor's fields are configured as 16-bit wide values (including the `header`), except for the `offset` and reference fields, which are 32- and 8-bit wide, respectively. Such parameter configuration results in 64-bit base descriptors, 32-bit $\{\mathtt{stride}_k, \mathtt{vsize}_k\}$ dynamic pairs, and 16-bit $\{\mathtt{fmod}_m\}$ modifier chain fields, which allows to fully address the 1-GB DDR3 memory module.

To evaluate the in-cache stream infrastructure with the proposed memory-aware SMC (ICS-SMC setup), a comparison was performed with the implementations of the previously proposed in-cache stream infrastructure [14] (ICS) and the state-of-the-art AMPM [9] stride prefetcher.[1] To guarantee a fair and realistic comparison, a conventional cache-based system (BASE) was used as reference setup. A computing infrastructure (common to all setups) was used, composed of a variable amount of PEs (ranging between 1 and 64), each comprising: an adapted MB-LITE [28] processor (as characterized in Table I) modified to support vector instructions and floating point operations; a private scratchpad for program data; and a memory-mapped interface to a L1 cache (BASE and AMPM) or an in-cache stream controller (ICS and ICS-SMC).

[1] An idealized behavioral model was implemented and simulated according to the architecture description provided in [9].

TABLE I
RESOURCE USAGE FOR EACH COMPONENT OF THE MORPHABLE COMMUNICATION INFRASTRUCTURE

	Available Resources	Baseline Cache Ctrl.[2]	In-Cache Stream Ctrl.[2]	DTC Architecture	Proposed PDC	Memory-Aware SMC (w/ PDC)	Burst Controller	Ring Node	MB-LITE-based PE
Slices	75,900	1896 (2.5%)	2370 (3.1%)	498 (.7%)	605 (.8%)	852 (1.1%)	274 (.4%)	155 (.2%)	1753 (2.3%)
LUTs	303,600	3602 (1.4%)	4367 (1.4%)	952 (.3%)	1466 (.5%)	1666 (.5%)	508 (.2%)	297 (.1%)	4566 (1.5%)
Registers	607,200	365 (.1%)	1176 (.2%)	692 (.1%)	19 (.1%)	991 (.2%)	415 (.1%)	164 (.1%)	1013 (.2%)
BRAM	3,090	16 (.5%)	16 (.5%)	7 (.2%)	7 (.2%)	9 (.3%)	0 (0%)	2 (.1%)	6 (.2%)
Static Power[1]	-					210.42			
Dynamic Power*[1]	-	89.3	91.2	22.8	34.1	43.2	44.2	10.3	138

* @100 MHz [1] Power consumption values displayed in mW [2] Considering an 8KB 4-way set-associative memory w/ 64B cache lines

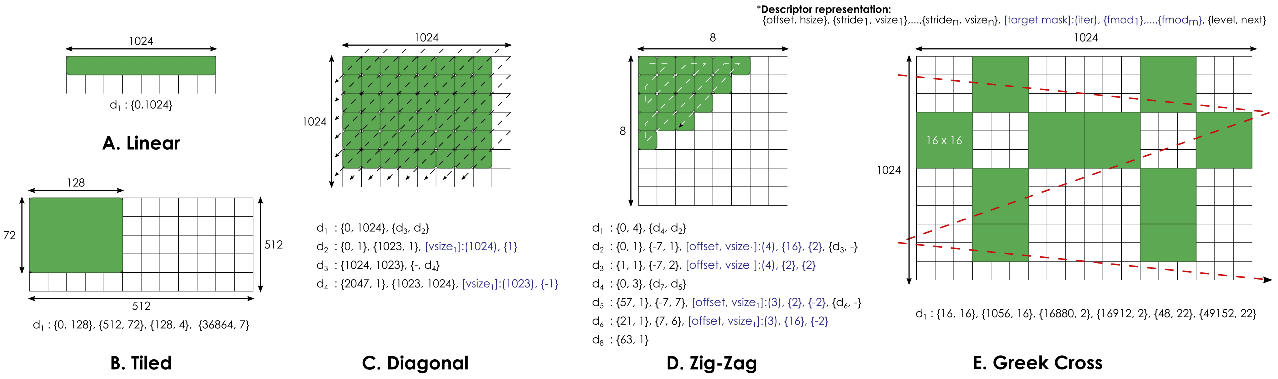

*Descriptor representation:
{offset, hsize}, {stride1, vsize1},...,{striden, vsizen}, [target mask]:(iter), {fmod1},...,{fmodm}, {level, next}
1024
d1 : {0,1024}
A. Linear
128
72
512
512
d1 : {0, 128}, {512, 72}, {128, 4}, {36864, 7}
B. Tiled
1024
1024
d1 : {0, 1024}, {d3, d2}
d2 : {0, 1}, {1023, 1}, [vsize1]:(1024), {1}
d3 : {1024, 1023}, {-, d4}
d4 : {2047, 1}, {1023, 1024}, [vsize1]:(1023), {-1}
C. Diagonal
8
8
d1 : {0, 4}, {d4, d2}
d2 : {0, 1}, {-7, 1}, [offset, vsize1]:(4), {16}, {2}, {d3, -}
d3 : {1, 1}, {-7, 2}, [offset, vsize1]:(4), {2}, {2}
d4 : {0, 3}, {d7, d5}
d5 : {57, 1}, {-7, 7}, [offset, vsize1]:(3), {2}, {-2}, {d6, -}
d6 : {21, 1}, {7, 6}, [offset, vsize1]:(3), {16}, {-2}
d8 : {63, 1}
D. Zig-Zag
1024
1024
16 x 16
d1 : {16, 16}, {1056, 16}, {16880, 2}, {16912, 2}, {48, 22}, {49152, 22}
E. Greek Cross

Fig. 9. Considered data patterns (shown using 2-D memory regions representations) and their corresponding dynamic descriptor graph encoding. The HEADER field is omitted in each descriptor for the convenience of the illustration.

The cache memory configurations were made identical to a typical ARM Cortex A7 configuration. Hence, each PE is associated with an 8 kB four-way set-associative data cache memory with 64-Byte cache lines, deploying a write-through-invalidate, write no-allocate snooping protocol, managed by a binary-tree-based pseudoleast recently used replacement policy. The proposed SMC was configured with a 256 × 64-bit descriptor memory (scratchpad), while the reorder buffer comprises a 4 kB, 32-bitline, direct-mapped memory structure. The AMPM prefetcher was configured with a prefetch degree of 4 (four prefetch requests per memory access), hence achieving top performance, while avoiding overprefetching, as presented in [9]. The memory access map was tuned for a 4-kB storage (large enough for the considered data sets).

Finally, the considered setups are connected to the DDR3 controller through an AXI4-Full bus interconnection (memory access bus in Fig 1). Accordingly, the main memory controller for the BASE, AMPM, and ICS setups was configured to perform cache-line-sized burst transfers. On the other hand, in the proposed SMC, the burst controller was set to use a maximum burst length of 256 words (the same as the AXI4 protocol), meaning that it splits the requested data transfers in individual 256-word burst requests.

A. Hardware Resources Overhead

The results of the implementation of the proposed morphable infrastructure (shown in Fig. 1) in the considered Virtex-7 FPGA device are presented in Table I. Despite the added versatility of the offered streaming capabilities, the results obtained for the devised in-cache stream controller represent a lightweight increase of the required hardware resources (about 0.6% of the total FPGA slices), when compared with the baseline cache controller. In fact, each of the devised components requires less than 3% of the FPGA resources. Moreover, since the adopted ring-based interconnection is inherently scalable in what concerns its hardware footprint and operating frequency, it can be efficiently used to support a very large number of PEs, being the only limiting factor the increased communication latency between nodes. Besides the 8-kB four-way set-associative cache memories present in both the baseline cache and the proposed in-cache streaming architectures, the presented BRAM utilization also comprises the buffering structures that are found in each component, except for the in-cache stream controller, which implements these buffers with registers.

B. Data-Pattern Generation Efficiency

To evaluate the proposed PDC data-pattern generation efficiency and compare it with the state-of-the-art APMC [12], Hotstream [11] and the previously proposed DTC [13], a representative set of benchmark kernels were executed, using the data patterns considered in [13], namely: *Linear*; *Tiled*; *Diagonal*; *Zig-Zag*; and *Greek Cross*. Fig. 9 shows the considered patterns (in a 2-D representation) and their corresponding descriptor specifications. Table II presents the efficiency of the PDC in resolving the data patterns encoded with the proposed dynamic descriptor graph specification, compared with the considered state-of-the-art approaches (with their corresponding description encoding).

TABLE II
ADDRESS GENERATION RATE AND DESCRIPTOR SIZE (IN BYTES) FOR THE PROPOSED PDC AND COMPARISON WITH THE CONSIDERED RELATED WORK APPROACHES

Pattern Type	Applications/ Data Structures	Pattern Length (# words)	Proposed PDC			DTC [13]		DFC [11]		AXI DMA [27]	
			Size[1]	Addr/cycle		Size[1]	Addr/cycle	Size[1]	Addr/cycle	Size[1]	Addr/cycle
				No Burst	Burst[3]						
Linear	Array, Table	1024	8	1	256	16	1	24	1	32	0.96
Tiled	Arithmetic, Matrix	128×72^2	20	1	128	32	1	40	0.99	32	1
Diagonal	Bioinformatics [29]	1024×1024	52	1	1	128	1	44	1	65k	1
Zig-Zag	Entropy Encoding [30]	8×8	104	1	1	208	1	48 (132*)	0.36 (0.71*)	480	0.63
Greek Cross	Diamond Search [31]	1024×1024	28	1	16	48	1	132	0.89	228k	1

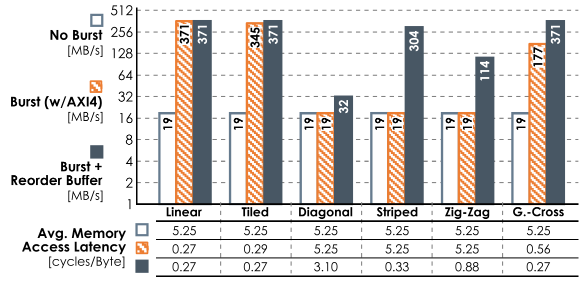
* Values obtained after loop unrolling
[2] Within a memory block of 512×512
[1] Description code size in bytes
[3] Assuming a maximum burst length of 256 words (ref. AXI4 Protocol [25])

By analyzing the values presented in Table II, it is clear that the proposed PDC architecture, by itself (not accounting for the SMC burst control), provides a steady one-address per cycle generation rate, which is the same as the DTC architecture [13] (hence, showing no performance loss from the added complexity), whereas the remaining considered state-of-the-art pattern generation structures show a significant performance degradation for high complex patterns. This is rather important, since the address generation rate typically constraints the system throughput, especially in memory-bound applications. Another advantage of the proposed system, when compared with the state-of-art approaches, regards the memory requirements for storing the actual data access pattern. As it can be concluded from Table II, the proposed graph-based pattern descriptor leads to an overall reduction of the required amount of memory, achieving up to 3.69 times and 8142 times memory savings for the *Greek Cross* pattern, when compared with the HotStream [11] and with the APMC [12] (since it is equivalent to an AXI DMA [27]). This corresponds to a maximum 41% improvement in memory resource requirements when compared with the previously proposed DTC [13].

The evaluation presented in Table II also shows that for some of the patterns, the Hotstream DFC requires slightly fewer memory resources for its pattern description code than the other approaches. However, while the Hotstream framework deploys one DFC controller per stream and per PE [11] (corresponding to at least two DFCs for a minimal scenario with one input and one output stream), the proposed morphable infrastructure requires only one SMC, independently of the number of PEs and streams. This way, with a slightly higher memory size that is required in some cases to store the proposed descriptor graph specification, it is possible to achieve a higher pattern generation rate (for any described pattern), with significantly fewer data-fetching hardware structures. As such, not only is most of the FPGA fabric left free for computing structures, but it also allows reducing the total energy consumption of the data-management infrastructure.

C. Main Memory Throughput

To complement such results, the benefits of the proposed burst control and reorder buffering optimizations to the main memory access were also evaluated. For such purpose, it was measured the throughput and average memory access latency (measured in cycles per byte) of the considered DDR memory with the adopted evaluation patterns for both optimization setups and compared with a direct connection between the PDC and the main memory, where only a single word is transferred per burst.



	Linear	Tiled	Diagonal	Striped	Zig-Zag	G.-Cross
□	5.25	5.25	5.25	5.25	5.25	5.25
▨	0.27	0.29	5.25	5.25	5.25	0.56
■	0.27	0.27	3.10	0.33	0.88	0.27

Fig. 10. Main memory throughput and associated average memory access latencies with the proposed burst control and reorder buffer optimizations for the considered data patterns.

The obtained results for an SMC implementation operating at 100 MHz are shown in Fig. 10. From these results, it is clear that the most regular access patterns (with larger contiguous data regions) take the most advantage of the proposed burst controller. In particular, for the *Linear* and *Tiled* patterns, it is possible to reduce the average latency by as much as 20 times, resulting in a throughput of 371 MB/s, close to the AXI4 bus theoretical maximum (400 MB/s for a single channel direction), when compared with a direct PDC connection. On the other hand, for the *Greek-Cross* pattern, the achieved throughput was limited at 128 MB/s (only nine times average latency reduction), due to its smaller 16-word burst lengths. The remaining patterns have no contiguous data regions and as such cannot take advantage of the proposed burst controller by itself.

However, some substantial improvements are observed when adding the proposed reorder buffer optimization. In particular, it is possible to increase the throughput of the *Greek-Cross* pattern to 371 MB/s by prefetching the 48×48 blocks before applying the crossed pattern. Similarly, some significant benefits can be observed for the *Diagonal* and *Zig-Zag* patterns. Similarly, for the zig-zag pattern, it is possible to prefetch each 8×8 block while applying the zig-zag scan (reducing the average latency by six times), hence increasing the effective throughput to 90 MB/s. Conversely, for the *Diagonal* pattern, only a slight throughput increase of 13 MB/s is observed, due to the limited size (4 kB) of the adopted reorder buffer, which provides limited data reuse opportunities (even when using a tiling procedure) when the

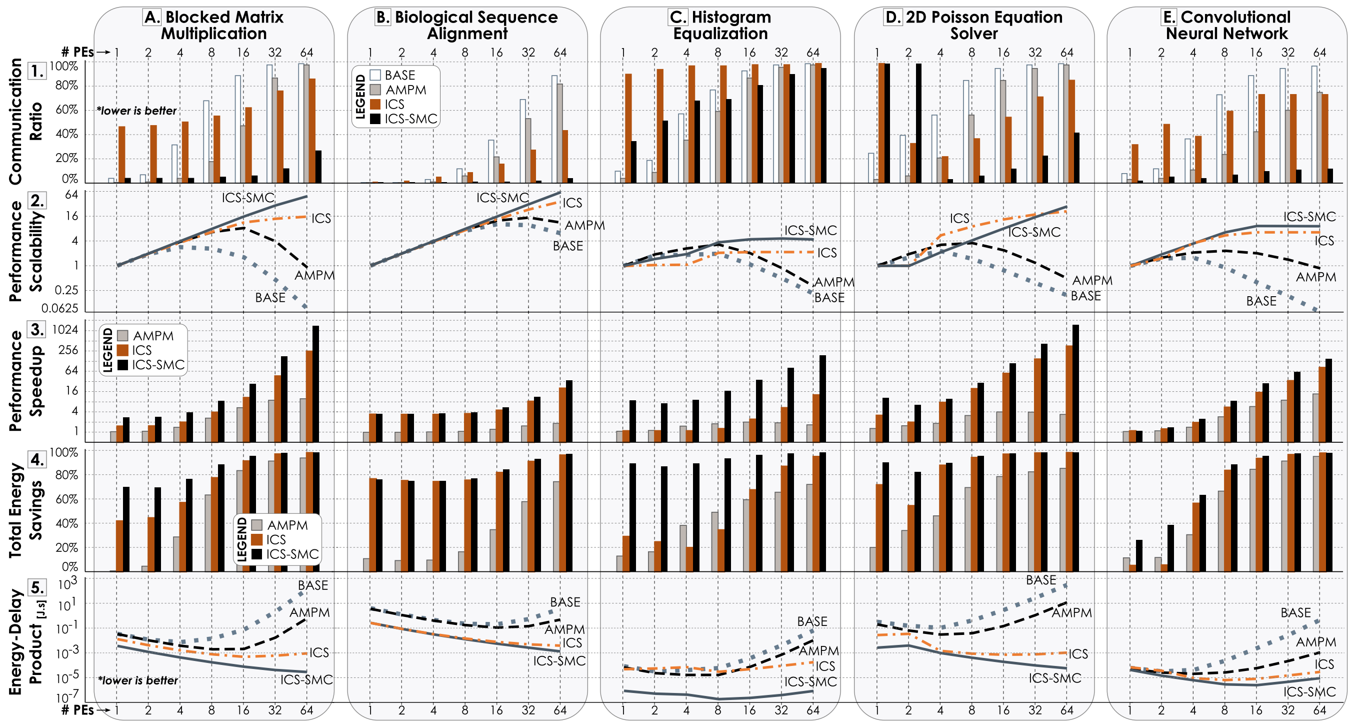


Fig. 11. Comparison of the proposed ICS-SMC with the BASE, AMPM, and ICS setups in: communication and execution time ratio improvement (top row), performance speedup scalability (second row) for each setup against its own single-PE configuration, execution time speedup (third row) and the total energy savings (fourth row) against the BASE reference, and overall system performance-energy efficiency (bottom row).

diagonal size is too large. In fact, at the larger diagonal, the throughput becomes effectively the same as with a direct connection to the memory. Naturally, when allowed by the processing algorithm, a possible optimization for this pattern is the adoption of a stripped (or banded) processing approach, where $n \times 1024$ stripes are prefetched and processed in smaller diagonals. With such an optimization, it becomes possible to take full advantage of the proposed burst controller and reorder buffer, resulting in an overall throughput increase. For example, with a configuration of 16 times 1024 stripes, the average memory access latency is reduced by 16 times and a maximum throughput of 304 MB/s is achieved (see Fig. 10, column *Striped*).

TABLE III
ADOPTED BENCHMARK APPLICATIONS AND THEIR CORRESPONDING DATA SETS, DATA ACCESS PATTERNS AND COMMUNICATION CHARACTERISTICS

BENCHMARK	INPUT SIZE	DATA PATTERNS	COMM./DATA MANAGEMENT[1]
Blocked Matrix Multiplication	128×128 8×8 blocks	*Tiled*	Hybrid, Broadcast Data Reuse
Biological Sequence Alignment [32]	128× 1024 queries 4096 reference	*Diagonal Striped*	Stream, PE-PE, Broadcast Data Reorganization
Histogram Equalization	256×256 image	*Tiled Linear*	Hybrid, PE-PE, Broadcast, Data Reuse
2D Poisson Equation Solver[2] [33]	128×128 grid 100 iterations	*Tiled Linear*	Stream, PE-PE Data Pipelining
Convolutional Neural Network[3] [34]	32×32 image 142× 5×5 kernels	*Tiled*	Stream, PE-PE, Broadcast Data Reuse+Pipelining

[1] As used in the In-Cache Stream setups
[2] Jacobi parallel iteration method (striped decomposition)
[3] For image classification

D. Performance Evaluation

In order to evaluate the gains provided by the data transfer and communication capabilities of the proposed morphable infrastructure (ICS-SMC), several different benchmarks were considered (see Table III), namely: 1) block-based matrix multiplication; 2) biological sequence alignment [32]; 3) histogram equalization; 4) 2-D Poisson equation parallel solver [33]; and 5) convolutional neural network for image recognition [34]. The considered setups were evaluated in terms of: 1) data communication and memory access time reduction; 2) performance speedup scalability for each setup against its own single-PE configuration; 3) performance speedup and total energy savings against the reference BASE setup: and 4) performance-energy efficiency [in the form of an energy-delay product (EDP) study].

1) Performance Evaluation: The obtained results for the considered evaluation benchmarks are shown in the graphs of Fig. 11, by considering a variable number of PEs. In particular, the bar plots on the top row represent the ratio of the communication time over the total execution time for each setup. Although the AMPM shows significant data-communication overhead reductions, when compared with the BASE reference, they are mostly present in configurations with fewer PEs (only up to 8 in most cases). Hence, despite their efficient capabilities, this type of prefetching mechanisms is not particularly suited for many-core systems, since they cannot cope with the increase in memory access traffic and bus contention in larger systems.

Conversely, as it can be observed for most of the considered benchmarks, the previous ICS setup already provides mechanisms that target the mitigation of such drawbacks, allowing a significant data-communication overhead reduction when compared with the AMPM setup. In particular, its offered stream prefetching, broadcasting, and data reutilization capabilities, allow a significant reduction of the total number of main memory accesses, therefore decreasing the observed contention in the interconnections. This is particularly evident with higher numbers of PEs, where increased levels of data reutilization and inter-PE communication are possible, due to the added amount of local memory resources available in the infrastructure. In fact, as it can be observed in the performance scalability graphs in the second row of Fig. 11, the AMPM setup with more than 16 PEs (8 in the Poisson equation solver benchmark) incurs in a high communication contention, resulting in a noticeable performance degradation (even resulting in performance slowdowns). Contrarily, the proposed ICS-SMC is capable of fully taking advantage of its capabilities to mitigate contention and increase overall performance, even with an amount of PEs as high as 64. For the histogram equalization [Fig. 11(C)] and Poisson equation solver [Fig. 11(D)], it is possible to observe a sudden increase in performance when the entire data set fits in the local memories and starts being directly communicated between PEs, thus avoiding unnecessary communication with the main memory.

Despite providing increased levels of communication efficiency with the addition of the proposed memory-aware SMC, an even higher data communication efficiency is observed for the ICS-SMC setup, as it can be observed from the top two rows of Fig. 11. In fact, a significant communication-execution time ratio reduction is observed, when compared with the AMPM and ICS setups, resulting from the proposed memory bandwidth optimization techniques of the SMC (burst controller and reorder buffer). This allows a greater mitigation of the communication overhead resulting from costly main memory accesses, which is particularly clear when observing the communication-execution time ratio improvement for the memory-bound matrix multiplication application [Fig. 11(A)].

Such a significant memory access optimization and the observed performance scalability are directly reflected on the total execution time speedups (against the BASE reference) presented in the bar plots from the third row of Fig. 11. As it could be expected, the AMPM is only capable of partly mitigating contention and memory traffic with fewer cores, achieving an average four times speedup for the considered benchmarks. Contrarily, the preliminary optimizations introduced by the ICS setup already allow significant performance scalability improvements, not showing the slowdowns observed in the BASE and AMPM setups, with configurations with more than 8 PEs. This, in turn, results in average 32 times execution time speedups for the ICS setup, when compared with the BASE setup, corresponding to a 10 times improvement over the AMPM.

However, when combining the in-cache stream infrastructure with the proposed memory-aware techniques, the ICS-SMC setup consistently provided performance speedup levels with no noticeable performance loss for most of the considered benchmarks. In fact, the only exceptions are the histogram equalization [Fig. 11(C)] and the convolutional neural network [Fig. 11(E)] benchmarks, where no added gain is observed with a 64-PE configuration (when compared with a 32-PE configuration), due to computation- and synchronization-related overheads. Such an improved communication efficiency allows average performance speedups of 126.7 times against the BASE setup, corresponding to average 54 times and 13 times improvements over the AMPM and ICS setups, respectively. Moreover, maximum performance speedups of up to 1500 times are observed for the ICS-SMC setup in the matrix multiplication [Fig. 11(A)] and Poisson equation solver [Fig. 11(D)] benchmarks.

2) Energy Savings Evaluation: The observed performance increases and data transfer overhead reductions directly impact the total energy consumption of the whole system, as shown in the energy savings attained by the proposed ICS-SMC (when compared with the other setups) (bar plots from the fourth row of Fig. 11). From the graphs, it is clear that besides the discussed performance improvements and the efficient data management of the proposed streaming and broadcasting techniques, the considered main memory access optimizations also allow a significant reduction of the system's total energy consumption. Such savings (averaging 91% with 64-PE setups) are directly related to the obtained execution time speedups, the significant decrease in bus communication (resulting from to the offered broadcast capabilities) and the reduced number of accesses to the main memory (accounted in the ratios of the top row bar plots for data transfers reduction).

3) Performance-Energy Efficiency Evaluation: To gather all the observed results in a single metric, an additional performance-energy efficiency study was performed. In this case, it was used an EDP metric, calculated by multiplying the total energy consumption of a given setup by the corresponding execution time of each application. The line plots in the bottom row of Fig. 11 represent the measured EDP for each setup. By keeping in mind that lower values represent a higher efficiency, the measured results reflect the attained lower energy consumption and the improved scalability of the proposed morphable infrastructure, when compared with the hybrid setup. As it could be expected, the poorer performance scalability and higher energy consumptions observed in the BASE and AMPM setups also result in a poorly scalable EDP with the increase of the number of PEs, due to the increased contention in its shared interconnections. On the contrary, the measured EDP values for the proposed ICS-SMC show that it is capable of mitigating the contention to the shared structures and consequently reduce data transfer overheads, allowing higher communication throughputs with lower energy consumptions. This is a direct result of the proposed data prefetching, reutilization, efficient communication capabilities and memory access optimization mechanisms. In fact, thanks to the introduction of the proposed SMC, it is possible to observe energy efficiency improvements as high as 245 times when compared with the previously proposed ICS setup.

VII. Conclusion

A novel in-cache stream communication model for many-core systems was proposed. The devised infrastructure is able to seamlessly switching between conventional memory-address and stream-based communication paradigms, while offering a set of streaming capabilities, including prefetching, complex memory access generation, and stream manipulation. The generation of data streams is performed by a new memory-aware SMC structure, which relies on a novel dynamic descriptor graph specification, capable of easily describing any arbitrarily complex access pattern. To decode such descriptors, a PDC architecture was specially devised that is capable of offering a steady-state one-address per clock cycle pattern generation efficiency. Furthermore, the conceived SMC is paired with a burst controller to optimize the main memory bandwidth, along with a reorder buffer to further exploit data reorganization and reutilization techniques through in-time stream manipulation.

The obtained results show performance increases, averaging 126.7 times, 54 times, and 13 times, when compared with a reference setup, a state-of-the-art stride prefetcher [9] and a previous implementation [14], respectively. The proposed infrastructure also allows significant total energy savings (averaging 91%), which result in overall processing energy efficiency improvements as high as 245 times, when compared with the previous implementation. To conclude, the obtained results highlight the proposed morphable infrastructure capabilities to significantly reduce the data transfer overheads with its efficient data prefetching, reutilization and communication management techniques, mitigating contention in the shared interconnections, which in turn increases performance and overall energy efficiency.

References

[1] T. Chau, X. Niu, A. Eele, W. Luk, P. Cheung, and J. Maciejowski, “Heterogeneous reconfigurable system for adaptive particle filters in real-time applications,” in *Proceedings of the Reconfigurable Computing: Architectures, Tools and Applications: 9th International Symposium, ARC 2013* (Lecture Notes in Computer Science), vol. 7806. Berlin, Germany: Springer, 2013, pp. 1–12.

[2] M. Modarressi, A. Tavakkol, and H. Sarbazi-Azad, “Application-aware topology reconfiguration for on-chip networks,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 11, pp. 2010–2022, Nov. 2011.

[3] R. Pal, K. Paul, and S. Prasad, “Rekonf: A reconfigurable adaptive manycore architecture,” in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. (ISPA)*, Apr. 2012, pp. 182–191.

[4] R. Kumar, T. G. Mattson, G. Pokam, and R. Van Der Wijngaart, “The case for message passing on many-core chips,” in *Multiprocessor System-on-Chip*. New York, NY, USA: Springer, 2011, pp. 115–123.

[5] K. T. Sundararajan, T. M. Jones, and N. P. Topham, “The smart cache: An energy-efficient cache architecture through dynamic adaptation,” *Int. J. Parallel Program.*, vol. 41, no. 2, pp. 305–330, 2013.

[6] G. Kalokerinos *et al.*, “FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability,” in *Proc. Int. Symp. Syst., Archit., Modeling Simulation (SAMOS)*, 2009, pp. 149–156.

[7] Y.-T. Chen *et al.*, “Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design,” in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Apr. 2012, pp. 45–50.

[8] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, “Energy-efficient hardware data prefetching,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 2, pp. 250–263, Feb. 2011.

[9] Y. Ishii, M. Inaba, and K. Hiraki, “Access map pattern matching for high performance data cache prefetch,” *J. Instruct.-Level Parallelism*, vol. 13, pp. 1–24, Sep. 2011.

[10] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2013, pp. 247–259.

[11] S. Paiagua, F. Pratas, P. Tomas, N. Roma, and R. Chaves, “Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels,” in *Proc. 25th Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Sep. 2013, pp. 17–24.

[12] T. Hussain, O. Palomar, O. Unsal, A. Cristal, and E. Ayguadé, and M. Valero, “Advanced pattern based memory controller for FPGA based HPC applications,” in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Sep. 2014, pp. 287–294.

[13] N. Neves, P. Tomás, and N. Roma, “Efficient data-stream management for shared-memory many-core systems,” in *Proc. 25th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2015, pp. 508–515.

[14] N. Neves, A. Mussio, F. Gonçalves, P. Tomás, and N. Roma, “In-cache streaming: Morphable infrastructure for many-core processing systems,” in *Proc. Eur. 9th Workshop Conventional High Perform. Comput. (UCHPC)*, 2016.

[15] J. Park and P. C. Diniz, “Data reorganization and prefetching of pointer-based data structures,” *IEEE Design Test Comput.*, vol. 28, no. 4, pp. 38–47, Apr. 2011.

[16] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently prefetching complex address patterns,” in *Proc. 48th Int. Symp. Microarchit.*, Sep. 2015, pp. 141–152.

[17] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, “Practical off-chip meta-data for temporal memory streaming,” in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit.*, Sep. 2009, pp. 79–90.

[18] E. Ebrahimi, O. Mutlu, and Y. N. Patt, “Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems,” in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit.*, Apr. 2009, pp. 7–17.

[19] J. Park and P. Diniz, “Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines,” in *Proc. 14th Int. Symp. Syst. Synth.*, 2001, pp. 221–226.

[20] A. A. El-Moursy and F. N. Sibai, “V-set cache: An efficient adaptive shared cache for multi-core processors,” *J. Circuits, Syst., Comput.*, vol. 23, no. 07, p. 1450095, 2014.

[21] S. Ghosh *et al.*, “Cache miss equations: An analytical representation of cache misses,” in *Proc. ACM Int. Conf. Supercomput.*, 1997, pp. 317–324.

[22] T. Grosser, H. Zheng, R. Aloor, and A. Simburger, A. Grosslinger, and L.-N. Pouchet, “Polly-polyhedral optimization in LLVM,” in *Proc. 1st Int. Workshop Polyhedral Compilation Techn. (IMPACT)*, 2011.

[23] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, “GRAPHITE: Polyhedral analyses and optimizations for GCC,” in *Proc. GCC Develop. Summit*, 2006.

[24] Z. Majo and T. R. Gross, “Matching memory access patterns and data placement for NUMA systems,” in *Proc. 10th Int. Symp. Code Generat. Optim.*, Sep. 2012, pp. 230–241.

[25] ARM Ltd. (Feb. 2013). *AMBAAXI and ACE Protocol Specification, Issue E*. [Online]. Available: http://infocenter.arm.com

[26] “TN-41–01: Calculating memory system power for DDR3,” Micron Technol., Inc., Boise, Idaho, USA, Tech. Rep. TN-41-01, 2007.

[27] “LogiCORE IP AXI DMA v6.03a,” Xilinx, San Jose, CA, USA, Tech. Rep. PG021, 2012.

[28] T. Kranenburg and R. van Leuken, “MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture,” in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2010, pp. 997–1000.

[29] N. Neves, N. Sebastiao, D. Matos, and P. Tomás, P. Flores, and N. Roma, “Multicore SIMD ASIP for next-generation sequencing and alignment biochip platforms,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 7, pp. 1287–1300, Apr. 2015.

[30] G. Wallace, “The JPEG still picture compression standard,” *IEEE Trans. Consum. Electron.*, vol. 38, no. 1, pp. xviii–xxxiv, Jan. 1992.

[31] S. Zhu and K.-K. Ma, “A new diamond search algorithm for fast block-matching motion estimation,” *IEEE Trans. Image Process.*, vol. 9, no. 2, pp. 287–290, Feb. 2000.

[32] M. Farrar, “Striped Smith–Waterman speeds database searches six times over other SIMD implementations,” *Bioinformatics*, vol. 23, no. 2, p. 156, 2007.

[33] V. Bandishti, I. Pananilath, and U. Bondhugula, “Tiling stencil computations to maximize parallelism,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Sep. 2012, pp. 1–40.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.