# An Adaptive Memory Interface Controller for Improving Bandwidth Utilization of Hybrid and Reconfigurable Systems

Vito Giovanni Castellana*†, Antonino Tumeo†, Fabrizio Ferrandi*
*Politecnico di Milano, DEIB - Milano, Italy - {vcastellana, ferrandi}@elet.polimi.it
†Pacific Northwest National Laboratory - Richland, WA, USA - {vitogiovanni.castellana, antonino.tumeo}@pnnl.gov

***Abstract*—Data mining, bioinformatics, knowledge discovery, social network analysis, are emerging irregular applications that exploits data structures based on pointers or linked lists, such as graphs, unbalanced trees or unstructured grids. These applications are characterized by unpredictable memory accesses and generally are memory bandwidth bound, but also presents large amounts of inherent dynamic parallelism because they can potentially spawn concurrent activities for each one of the element they are exploring. Hybrid architectures, which integrate general purpose processors with reconfigurable devices, appears promising target platforms for accelerating irregular applications. These systems often connect to distributed and multi-ported memories, potentially enabling parallel memory operations. However, these memory architectures introduce several challenges, such as the necessity to manage concurrency and synchronization to avoid structural conflicts on shared memory locations and to guarantee consistency. In this paper we present an adaptive Memory Interface Controller (MIC) that addresses these issues. The MIC is a general and customizable solution that can target several different memory structures, and is suitable for High Level Synthesis frameworks. It implements a dynamic arbitration scheme, which avoids conflicts on memory resources at runtime, and supports atomic memory operations, commonly exploited for synchronization directives in parallel programming paradigms. The MIC simultaneously maps multiple accesses to different memory ports, allowing fine grained parallelism exploitation and ensuring correctness also in the presence of irregular and statically unpredictable memory access patterns. We evaluated the effectiveness of our approach on a typical irregular kernel, graph Breadth First Search (BFS), exploring different design alternatives.**

## I. Introduction

Semantic databases, social network analysis, data mining, bioinformatics, language understanding, pattern recognition and, in general, knowledge discovery are new, emerging irregular applications. They feature irregular data structures such as graph, unbalanced trees or unstructured grids, which employ pointers or linked lists. These data structures provide a large amount of inherent dynamic parallelism, because the application can potentially spawn a concurrent activity for each element to explore. However, they also present very poor spatial and temporal locality, because any element can point to any other element, leading to substantially unpredictable, fine-grained, memory accesses. In addition, these data structures usually are large, but difficult to partition without generating load unbalance. For this reasons, it is more convenient to develop irregular applications by exploiting parallel shared memory programming models. Lately, several systems targeting irregular applications that employ hybrid architectures have appeared. Hybrid architectures integrate both general purpose processors and reconfigurable logic, such as FPGAs, to accelerate some specific workloads. Solutions like the Convey HC include custom personalities (hand-designed accelerators) for some irregular algorithms, such as Breadth First Search (BFS) [2]. These platforms integrate complex, custom memory controllers with multiple distributed memories, which support many concurrent memory requests, high bandwidths and large memory sizes. These approaches demonstrated promising speed ups with respect to commodity systems, providing alternative, smaller scale, solutions than fully custom systems for irregular applications such as the Cray XMT multithreaded supercomputer [6]. However, they still are custom designs, with hand-developed accelerators or even processors, loaded on the reconfigurable logic. Modifying them means rewriting the RTL code, the software runtimes and the related interfaces towards the general purpose processors. It is difficult to adapt them to specific or new applications. On the other end of the spectrum, High Level Synthesis (HLS) tools allow automatic generation of hardware accelerators starting from high level languages such as C. They appear very promising for hybrid architectures [9]: the developers can decide to offload some of the kernels to the reconfigurable logic, and let the tool generate all the RTL code to synthesize. However current HLS paradigms do not consider many of the issues in irregular applications. They adopt very restrictive abstractions for memory, usually considering a single ported memory and serializing all the accesses. In addition, they provide poor (if any) support for synchronization directives. The common approach for supporting synchronization requires interacting with off-the-shelf soft processors or custom schedulers, which manage the execution on the hardware modules. This interaction may considerably increase the execution latencies when compared to full custom designs. In this paper we introduce an adaptive Memory Interface Controller (MIC), which enables complete management of concurrency and synchronization on multiple memory resources. The MIC dynamically maps memory operations across distributed and/or multi-ported memories, such as those available in hybrid systems. The MIC routes the memory accesses towards their memory ports at runtime, enabling the support of the unpredictable access patterns. The MIC also manages concurrency through a lightweight arbitration scheme, which avoids any structural conflict on shared resources, and which does not introduce any delay. The MIC is able to concurrently issue multiple memory operations, provided that they do not target the same memory locations, because it performs access routing and resource availability checking at runtime, thus improving the overall memory bandwidth utilization of the systems. The MIC provides synchronization management by implementing atomic memory operations, such as fetch-and-add and compare-and-swap, which are the basis for the synchronization primitives commonly adopted in shared memory parallel programming. We designed the MIC to make it adaptable to different target platforms and allowing large degrees of customization. It is possible to tune its implementation through parameters that, for example, allow changing the bitwidths, the number of concurrent memory accesses, and the number of memory banks to connect to. An embedded system developer can easily integrate the MIC in custom hand-written designs, but we specifically developed it to also enable its automatic allocation in typical HLS flows. We validate our approach by performing design space exploration for a typical irregular application kernel, the BFS, varying the number of concurrent kernels and memories.

978-3-9815370-2-4/DATE14/©2014 EDAA

## II. Related Work

In the last few years, several approaches to accelerate irregular kernels, such as graph traversal, with hybrid architectures and reconfigurable devices have appeared. The most important examples are the BFS personalities for the Convey HC systems [2], and the new Convey MX system, which couples a multithreaded custom processor on the reconfigurable logic with an OpenMP programming environment (CHOMP - Convey Hybrid OpenMP) [3], providing significant speed ups for graph exploration kernels. Betkaoui *et al.* [4] present a reconfigurable hardware methodology for efficient parallel processing of large-scale graph exploration problems. The authors design the architecture by hand, and increase parallelism by replicating the basic BFS kernel. The application execution takes place in a three stages process, through the interaction between the kernel instances (Graph Processing Elements, GPEs) and a Runtime Management Unit (RMU). The RMU partitions the vertices, assigning each partition to a GPE. Then, the GPEs execute concurrently, until each GPE has processed its assigned partition, notified through a termination signal to the RMU. In the last step, the RMU manages synchronization of the GPEs, evaluating whether the execution process should restart. The authors suggest the adoption of HLS tools to only generate the kernel implementation. We can, completely integrate our approach in a HLS framework. In fact, it does not require the definition of custom RMUs, because it directly manages synchronization through atomic memory operations. In [8] the authors propose a parameterized speculative multi-ported memory subsystem: it enables speculative execution of memory operations, aiming at increasing the available parallelism. The introduction of multi-ported caches allows concurrent execution of multiple memory operations. A lightweight protocol implements coherency by partitioning the data on different Coherency Clusters, therefore reducing its management costs. Access towards the external DDR memories is serial. The experimental evaluation highlights that the adoption of speculation has diminishing returns when targeting irregular applications: for the considered irregular kernels, it shows a performance gain less than 10%. In our work we do not consider either caching or speculation, because unpredictable accesses increase the costs of coherency management, and speculation has negligible effects on performance. In [5], Cong *et al* present an implementation of the fluid registration algorithm on a Convey HC-1 multi-FPGA platform. The authors implement the algorithm exploiting a HLS tool, but they suggest that further effort is required to fully support the features of a hybrid architecture in an automatic framework. In [10] the authors show how ROCCC 2.0, a HLS tool, can support the Convey HC-1 platform. The paper shows how Dynamic Time Warping and Viola-Jones algorithms are converted from C specification to a HDL specification, targeting the Convey system. However, the approach still requires performing optimizations on the code to fully support the platform. To the best of our knowledge, current hardware synthesis methodologies do not address the issues of irregular applications in their entirety. These include: abundant task level parallelism, unpredictable and parallel memory accesses, fine grain synchronization through atomic memory operations. There are, however, some approaches that look at supporting some of these features. [7] discusses how to extend ROCCC to support irregular applications. The authors introduce multithreading to tolerate long memory access latencies, and describe how they customized the ROCC compiler to generate concurrent hardware threads and to support customized state information for each dynamically generated thread. However, they do not address atomic memory operations.

```
1 void application_template(){
2     //code block
3     for(id=init; id<NUM_it; id=id+1 ){
4         kernel(id, data);
5     }
6     //code block
7 }
```

Fig. 1: *Application Template*

## III. Accelerating Irregular Applications

Irregular applications typically expose coarse grain parallelism, usually located in loops. The general template provided in Figure 1 can map most irregular applications, such as graph problems [4]. There are several challenges when mapping these applications to hardware with automated synthesis flows. First, hardware acceleration generally relies on fine grained parallelism exploitation: ILP inside each kernel is usually limited. Furthermore, the kernels are mostly memory bound, because the largest part of parallel operations are memory accesses. As a result, hardware design methodologies focused on ILP provide limited speed ups. Multi-ported or distributed memories can mitigate this issue by allowing multiple concurrent memory accesses. However, the memory access patterns are irregular. Thus, statically binding a memory operation to a hardware resource is not possible. In turn, this makes concurrent execution of memory operations non-trivial. Synchronization among different kernels is an additional source of complexity. In fact, different kernels may share the same memory resources, so a way to preserve consistency is required. The graph BFS algorithm presents all the previously mentioned aspects. We mapped a queue-based implementation of the BFS algorithm on the general template of Figure 1, identifying kernels which may execute concurrently. Each one of them iterates over the out edges of a given vertex: when it traverses a new neighbor, the neighbor is marked as visited and added to the next iteration queue. If multiple kernels run concurrently, they perform write accesses to a shared data structure. Consequently, there must be a way to synchronize the accesses. In software parallel programming, atomic operations, such as compare-and-swap and fetch-and-add, provide synchronization. In this work we propose a hardware accelerator design for irregular applications, based on the definition of an adaptive Memory Interface Controller. To overcome the highlighted issues and to achieve significant speed ups, the proposed design: 1) allows concurrent execution of memory operations on multiple memory banks, also with an irregular access pattern; 2) guarantees memory consistency when multiple kernels concurrently access shared data; this is achieved by implementing atomic operations through dedicated hardware; 3) improves coarse grained parallelism exploitation, allowing kernels to run concurrently without the need of inter-process communication. The methodology exploits coarse grained parallelism through "spatial" multithreading, i.e. by replicating multiple times the same kernel. The introduction of the MIC enables support of concurrent memory accesses to different memory banks. The MIC dynamically steers memory access requests to the proper memory ports, while managing concurrency among them. The MIC also provides atomic operations, which are considered as a special type of memory operations. Figure 2 shows the targeted accelerator structure, which maps on hardware the application template proposed in Figure 1, after applying partial unrolling with unrolling factor of $N$. This better exposes task level parallelism, and facilitates the implementation process. If a kernel function performs memory operations on shared memories, it forwards execution requests to the caller. In

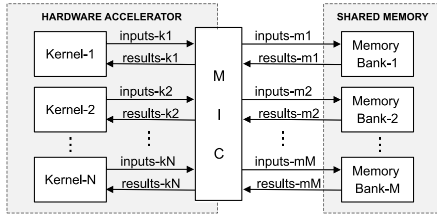


Fig. 2: *Accelerator design template schematic representation.*

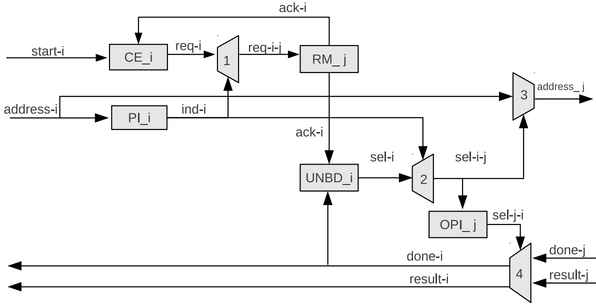


Fig. 3: *Memory Interface Controller schematic representation.*

case of nested calls, forwarding is recursive, until the top level is reached. At the top level, the MIC manages the memory accesses. The introduction of the MIC adds an abstraction layer between the accelerator and the memory structure, which decouples the problems of designing the two components. Varying for example the number of available memory banks, or the scrambling function used to distribute data, have no impact on the accelerator implementation. In fact, the accelerator can be implemented as an independent module: the designer or the synthesis tool may ignore mutual interferences between different kernels, or in general, memory accesses, because the MIC manages synchronization and concurrency. This abstraction also facilitates the design process: for example, operation scheduling may be addressed independently for each kernel, without performing difficult inter-functional analysis. In addition, it allows trivial kernel replication. All these aspects improve modules reusability and the efficiency of Design Space Exploration tasks, while reducing their complexity. This is an important characteristic, because there are several design choices that may affect system performance, such as the number of allocated kernel instances, and the number of concurrent memory operations which the MIC should manage.

## IV. Memory Interface Controller

We designed the Memory Interface Controller (MIC) with the objectives of dynamically addressing irregular memory accesses to the corresponding memory port, while managing their concurrency. Basically, the MIC takes in input memory access requests from $N$ ports, which have an address, a data and an operation type (load/store) line. It routes requests towards one of the $M$ output ports by evaluating their addresses. It serves a request as soon as the corresponding port is available. In a similar way, it routes back $M$ done signals (which notify termination of an operation) and the results (in case of loads) to the requesting operation. The memory is composed of $M$ different and independent banks, and each output port accesses one bank (Figure 2). Each memory bank has non-overlapping addresses. This is equivalent to having $M$ different distributed memories. Figure 3 provides a schematic view of the controller structure. The MIC associates each input operation $i$ to a Control Element (CE) and a module (PI) that analyzes the input address to establish the destination port. This is obtained by embedding in the PI design the hardware implementation of the scrambling function, which distributes data on the multiple memory partitions. The design allocates a Resource Manager (RM), which has the role of managing concurrency, for each output $j$. Each CE intercepts execution requests and forwards them to the RMs, until they are accepted. A Port Index signal produced by the PI, working as selector of the steering logic (connection 1), allows performing the routing of the requests. Once a RM accepts a request, it sends back an ack signal to the corresponding CE, disabling it, and to the UNBD module, which is responsible of setting the selections while the operation is running. These signals, according to the output of PIs (connection 2), drive the steering logic that feeds the memory ports (connection 3). Figure 3 only shows the connections and logic for the address line of an input port. All the other input lines follow a similar approach, duplicating the steering logic and interconnections. Similarly, the MIC routes done signals and results coming from memory (read accesses), according to the requesting input port. In this case, OPeration Index (OPI) modules provide the selectors for the interconnections. OPIs identify the input port requesting the memory access. The design of the MIC, thanks to its modularity and regularity, is not constrained by any particular characteristic of the ports (number or bitsize of inputs/outputs).

*Atomic operations:* Atomic operations indivisibly perform a sequence of operations (read and write) on a given memory location, ensuring that its content is not modified by other operations during their execution. In our design, we obtain an atomic behavior by delegating management of atomic operations to the MIC. When the MIC accepts the execution request of an atomic operation, it exclusively binds the associated memory port to the operation, until its completion. The MIC manages atomic execution through dedicated hardware. For example, fetch-and-add operations read the value at the specified address, add the provided operand to the previously read value, and then store the result in the same memory location. They return the old value read. The MIC implements this operation as follows. First, it performs the load operation. When the MIC receives the done signal coming from the memory, it intercepts the signal, buffers the loaded value, and calculates the sum. Then, it stores the result of the sum into the memory. The subsequent done signal is associated to the whole atomic operation, which then returns the buffered value. The MIC implements other atomic operations following the same approach. The MIC includes dedicated hardware to manage atomic operations for each memory port, thus allowing concurrent execution of one atomic operation per memory bank.

## V. Experimental Evaluation

We implemented the MIC in Verilog, and designed two different versions of it. The first only considers loads and stores, without support for atomic operations. It takes in input memory addresses, input data for store operations, selectors for identifying the operation type, and a start signal. It produces in output done signals and, in case of load operations, a result. The second version of the module extends the previous one by providing support to fetch-and-add and compare-and-swap atomic operations. There are additional selectors in input, and dedicated output lines for providing the results. The same lines used for input data of store operations provide the operands. Both the MIC implementations are customizable, and take the number of input operations and the number of memory banks as parameters. We evaluated the area of the designed modules, varying both the number of inputs N (associated with the maximum number of parallel memory operations) and the number of outputs M (associated with the number of memory banks). We synthesized

TABLE I: *Performance evaluation: speed-ups with respect to serial executions; input graph: 5000 nodes; average out degree: 10 (22767 edges), 20 (47597 edges) and 30 (72887 edges).*

| | **M = 4** | | | | | | | | | **M = 8** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **avg out degree: 10** | | | **avg out degree: 20** | | | **avg out degree: 30** | | | **avg out degree: 10** | | | **avg out degree: 20** | | | **avg out degree: 30** | | |
| **ker** | **2cc** | **5cc** | **10cc** | **2cc** | **5cc** | **10cc** | **2cc** | **5cc** | **10cc** | **2cc** | **5cc** | **10cc** | **2cc** | **5cc** | **10cc** | **2cc** | **5cc** | **10cc** |
| 4 | 2.7 | 2.36 | 2.2 | 2.85 | 2.48 | 2.27 | 2.9 | 2.52 | 2.29 | 2.84 | 2.61 | 2.48 | 2.97 | 2.71 | 2.53 | 3.01 | 2.74 | 2.57 |
| 5 | 3.09 | 2.62 | 2.39 | 3.33 | 2.76 | 2.49 | 3.36 | 2.8 | 2.51 | 3.34 | 2.98 | 2.79 | 3.53 | 3.14 | 2.9 | 3.57 | 3.14 | 2.91 |
| 6 | 3.45 | 2.85 | 2.57 | 3.7 | 3 | 2.65 | 3.78 | 3.04 | 2.68 | 3.81 | 3.34 | 3.09 | 4.03 | 3.53 | 3.23 | 4.1 | 3.58 | 3.27 |
| 7 | 3.74 | 3.01 | 2.68 | 4.02 | 3.15 | 2.79 | 4.13 | 3.22 | 2.82 | 4.2 | 3.66 | 3.33 | 4.5 | 3.88 | 3.53 | 4.58 | 3.93 | 3.58 |
| 8 | 3.96 | 3.16 | 2.8 | 4.32 | 3.29 | 2.89 | 4.43 | 3.35 | 2.92 | 4.63 | 3.95 | 3.61 | 4.98 | 4.22 | 3.82 | 5.08 | 4.3 | 3.89 |

TABLE II: *Area evaluation of the generated designs.*

| | **1ker/M=1** | **4ker** | **5ker** | **6ker** | **7ker** | **8ker** |
|---|---|---|---|---|---|---|
| FF,M=4 | 942 | 2466 | 3066 | 3415 | 4124 | 4563 |
| LUT,M=4 | 1141 | 4981 | 6200 | 7218 | 8378 | 8911 |
| FF,M=8 | 942 | 2611 | 3210 | 3559 | 4268 | 4706 |
| LUT,M=8 | 1141 | 6367 | 7912 | 9305 | 10935 | 11586 |

the circuits with Xilinx ISE ver 14.4, targeting a Virtex 6 xc6vlx75t device. We evaluated the proposed approach by exploring area and performance when synthesizing the BFS algorithms with different parameters. The choice is motivated by the fact that the BFS is considered one of the most typical irregular application kernel. We compared different implementations, varying the number of allocated kernels and the number of available memory ports. With respect to similar methodologies (e.g. [4], [8], [7]), our approach facilitates the adoption of HLS for the synthesis of the accelerators. In fact, we synthesized all the designs through *Bambu*, a state of the art HLS tool [1]. We introduced the MIC in the automatically generated designs, with negligible effort. Each kernel performs six memory accesses, and two atomic operations, i.e., one fetch-and-add and one compare-and-swap. Since the kernels require access to the whole memory, the MIC manages the memory accesses at the top level. According to the program dependences, each kernel can issue up to two concurrent memory operations. For this reason, we reserve two input ports of the MIC for each kernel. We evaluated the performance, in terms of execution latency, while also varying the size of the input graph and the latency model of the memory operations (2, 5 and 10 clock cycles per operation). We targeted an operating frequency of 100 MHz. Table I reports the obtained Speed-Ups against single kernel executions, for for the different data sets. To preserve the irregularity, we randomly generated the graphs. All the results refer to the execution of the complete BFS algorithm. We first synthesized the specification allocating only one kernel function, and interfacing it to a single bank memory. In these settings, parallelism exploitation is strictly bound to ILP. We compared the obtained latencies, progressively increasing the number of allocated kernels (from 4 to 8), targeting a 4-bank and a 8-bank memory architecture. For all the experiments, we verified speed-ups when increasing the number of kernels. To demonstrate the effectiveness of the proposed approach in parallelizing irregular memory accesses, we must also consider the relative speed-ups when varying the memory latency. Regardless of the number of kernels, we reported a significant speedup when increasing the number of memory banks. The gains are higher with high latency memories, because in such a case the execution latency is dominated by the memory accesses. This is a valuable result, especially when considering that the speed up increases when increasing the number of kernels, thus providing higher concurrency on the memory resources. Table II summarizes the area requirements (number of FF and LUT slices) of the accelerators. We remark that the single kernel implementation interfaces with a single banked memory. Thus, it is not affected by the area overhead of the MIC. Two aspects mainly determine the area utilization. The first one is associated with the number of allocated kernels: each kernel requires 433 FF and 488 LUT slices. The second component is the cost of the MIC: increasing the number of kernels slightly increases the cost of the controller, because two 2 additional ports per kernel are added to the MIC.

## VI. Conclusions and Future Work

In this paper we presented the implementation of an adaptive Memory Interface Controller (MIC) targeted at irregular applications. We specifically designed the MIC for the inclusion in automated HLS flows. The MIC supports distributed and multi-ported memories, which provide very high bandwidths for fine-grained memory operations. The MIC allows simultaneous execution of multiple memory accesses and introduces an abstraction layer that facilitates the design of custom accelerators, because it manages both synchronization and concurrency among the memory resources. The MIC also provides support for atomic memory operations. We described a case study for our approach, focusing on the Breadth First Search algorithm. We explored several trade-offs in terms of number of kernels and number of memories, showing how the MIC allows exploiting the parallelism available in the algorithm, maximizing concurrency of memory operations, and thus improving the system bandwidth utilization.

## References


[1] Bambu: A Free Framework for the High-Level Synthesis of Complex Applications. http://panda.dei.polimi.it.

[2] Convey computer doubles graph500 performance, develops new graph personality. available at http://www.conveycomputer.com/files/2413/5095/9078/sc11_graph500_release.final.pdf.

[3] Convey MX Series. Architectural Overview. available at http://www.conveycomputer.com.

[4] B. Betkaoui, Y. Wang, D. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15, 2012.

[5] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-fpga platforms. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 50–57, 2011.

[6] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM Press.

[7] R. J. Halstead, J. Villarreal, and W. Najjar. Exploring irregular memory accesses on fpgas. In *Proceedings of the first workshop on Irregular applications: architectures and algorithms*, IAAA '11, 2011.

[8] H. Lange, T. Wink, and A. Koch. MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*.

[9] B. Meyer, J. Schumacher, C. Plessl, and J. Forstner. Convey vector personalities - fpga acceleration with an openmp-like programming effort? In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 189–196, 2012.

[10] J. Villarreal, A. Park, R. Atadero, W. A. Najjar, and G. Edwards. Programming the convey HC-1 with ROCCC 2.0. In *CARL 2010: The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.