Adaptive Dynamic On-Chip Memory Management for FPGA-based Reconfigurable Architectures

Ghada Dessouky*, Michael J. Klaiber*, Donald G. Bailey†, Sven Simon*
*Institute for Parallel and Distributed Systems, University of Stuttgart, Germany
Email: michael.klaiber@ipvs.uni-stuttgart.de, ghada.dessouky@gmail.com
†School of Engineering and Advanced Technology, Massey University, Palmerston North, New Zealand

***Abstract*—In this paper, an adaptive architecture for dynamic management and allocation of on-chip FPGA Block Random Access Memory (BRAM) resources is presented. This facilitates the dynamic sharing of valuable and scarce on-chip memory among several processing elements (PEs), according to their dynamic run-time memory requirements. Different real-time applications are becoming increasingly dynamic which leads to unexpected and variable memory footprints, and static allocation of the worst-case memory requirements would result in costly overheads and inefficient memory utilization. The proposed scalable BRAM memory management architecture adaptively manages these dynamic memory requirements and balances the buffer memory over several PEs to reduce the total memory required, compared to the worst-case memory footprint for all PEs. The run-time adaptive system allocates BRAM to each PE sufficiently fast enough as required and utilized. In a case study, a significant improvement in BRAM utilization with limited overhead has been achieved due to the adaptive memory management architecture. The proposed system supports different BRAM types and configurations, and automated dynamic allocation and deallocation of BRAM resources, and is therefore well suited for the dynamic memory footprints of FPGA-based reconfigurable architectures.**

I. Introduction and Motivation

With the increasing complexity and performance requirements of real-time embedded systems and the advances in FPGA technology, came the advent of multi-processor architectures and, more recently, of reconfigurable computing. Reconfigurable computing exploits the reconfiguration capabilities of FPGA devices to reconfigure the resources on the FPGA to modify and adapt the functionality of these resources to a specific application or computation that needs to be performed [1]. More recently, dynamic partial reconfiguration (DPR) of FPGAs provided the possibility to specify and constrain certain partitions on an FPGA such that they can execute different tasks at different points in time without consuming additional area.

One main challenge of dynamic reconfigurable computing is the efficient assignment of resources to different partitions, such as the scarce and valuable block random access memory (BRAM), which is often a limiting factor in the design of complex embedded systems [2] [3]. Modules designed to occupy the same physical partition on FPGA can only utilize the on-chip BRAM resources within this partition, which are often not sufficient for memory-intense applications. One workaround to resolve the limited on-chip memory dilemma is the utilization of the more abundant off-chip memory [2] [3]. However, this imposes many physical design constraints on the FPGA-based implementation, and reduces its potential for flexibility and reconfigurability. Moreover, local on-chip memory is almost always the preferred memory choice for real-time applications, since it is the lowest latency (one clock cycle), fastest, and highest bandwidth memory solution available [3]. Hence, it becomes necessary to design the system using maximum worst-case memory footprint estimates, but such static memory allocation is inefficient and would impose excessive area and power consumption overheads [4]. Dynamic memory management is needed to enhance the gains of reconfigurable computing by meeting the dynamic context-dependent memory requirements of embedded reconfigurable applications and to avoid costly static memory allocations at design-time.

One such application is the tracking of moving objects in real-time. The run-time memory required depends on the number of objects entering the monitored region, and the number of features stored for these objects. Another example, is the connected component labeling (CCL) algorithm used to identify objects in typically binarized images [5], in which the run-time memory footprint depends on the number of objects identified and the features to be computed.

Another potential scenario is runtime reconfigurability: different PEs designed to occupy the same FPGA partition may have very different memory footprints. Static allocation of worst-case memory requirements would result in poor utilization of on-chip memory.

Hence, in this work, we propose a **D**ynamic **O**n-chip **M**emory **M**anagement **U**nit (**DOMMU**) which is customized to target the run-time dynamic management of on-chip BRAM to parallel FPGA-based PEs, according to their dynamic run-time memory footprints. DOMMU is designed with flexible user-configurability and scalability. It supports automated BRAM (de)allocation, which ensures that memory management remains transparent to the PEs. Support for sharing BRAM between PEs is also integrated, and can be extended to support additional BRAM configuration types. The functionality of DOMMU and the resulting improvement in BRAM utilization is demonstrated using an application case study, targetting the Xilinx Virtex-5 LX110T FPGA, although the design is easily ported to other device families.

The paper is organized as follows. An overview and comparison with related work is given in Section II. The architecture of DOMMU and its features are described in Section III, followed by more detailed structure and functionality in Section IV. In Section V, the testing of DOMMU and the performance and resource requirements are discussed. Finally, Section VI concludes the paper.

II. Related Work

Dynamic memory (or heap) management is extensively researched owing to its significance in software, and more recently in hardware architectures and embedded systems.

The survey in [6] examines in depth different general-purpose dynamic memory management policies and their complexities. Many general-purpose traditional memory allocators realize static memory allocation methodologies that perform well for most general cases, but fail to perform optimally in embedded applications, that are characterized by dynamic and unexpected memory footprint changes at run-time [4] [6]. Allocating memory statically to cover the worst-case due to the variations in memory footprint imposes costly overheads and poor utilization of on-chip memory, which is the motivation to investigate dynamic memory management for embedded systems [6], and to thoroughly study the timing performance of dynamic memory allocation algorithms to investigate their usability for real-time systems [7].

General-purpose dynamic allocation mechanisms such as those presented in [6] cannot be directly implemented for embedded systems since they impose costly overheads for the limited resources of embedded systems. Hence, different specific-purpose dynamic memory management solutions have been proposed in literature, such as a real-time operating system for embedded systems proposed in [8] which offers customized dynamic memory management depending on the running applications. Further customization is presented in [9] which proposes a methodology to define custom dynamic memory management solutions with the desired performance for embedded systems, by exploring the dynamic memory management design space in order to reduce the amount of memory access involved.

Dynamic memory management for embedded systems can be realized in hardware [10] [11] [12] [13] [14], as well as in software [8] [9] [4]. Software-only dynamic memory management has been extensively researched [6] [9] [15] and established as the more flexible and adaptive solution. However, it has lower performance and higher latency than hardware realizations. In [10], a hardware System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) is presented, which manages the (de)allocation of global on-chip memory within a two-level memory management hierarchy in a fast and deterministic manner. SoCDMMU allocates global pages of fixed number of equally sized blocks to the processors and the (de)allocation of memory assigned to each processor is managed locally. SoCDMMU is restricted to a memory granularity of equal fixed-sized memory blocks. Moreover, no methodology is provided for automatic or predictive BRAM (de)allocation. In [14], a general-purpose hardware memory management unit is proposed for NoC architectures which is restricted to managing shared memory only and with (de)allocation occurring at granularity of complete pages. Microcoded dynamic memory management services for Multi-Processor System-on-Chip (MPSoC) platforms which are customized and application-specific are proposed in [13], but targeting distributed on-chip shared memory. Similar to our design, is the adaptive memory core presented in [12], which is a multi-client memory core, designed for deployment in a MPSoC to facilitate access to external off-chip memory via dynamic mapping of the address space for the different processors. The core is scalable to support up to 16 processors and is restricted to support (de)allocation based on fixed number and fixed size memory blocks. Moreover, it targets the dynamic memory management of abundant off-chip memory resources.

As concluded from a review of the literature and as reported in [4], dedicated hardware solutions provide the higher performance, but with poor flexibility and poor support for reuse and run-time adaptivity. Our work attempts to provide both high performance and run-time adaptivity, by extending the aforementioned established solutions of dynamic memory management to offer a customized high-performance dynamic memory management architecture, which (de)allocates on-chip FPGA BRAM to several PEs.

III. Design Goals and Functionality of DOMMU

For DOMMU to dynamically manage on-chip memory allocation of PEs in reconfigurable computing, it has to meet the following requirements:

- **Dynamic Memory (De)Allocation**

Static memory allocation architectures often force PEs to reserve enough BRAM to cover worst-case requirements and to resort to off-chip memory for more. In typical cases, significantly less than worst-case memory is required, and the worst-case buffer can be provided for other PEs while unused. This dynamic sharing and allocation of memory can reduce the total memory required at run-time and improve BRAM utilization. However, dynamic allocation should be guaranteed to occur faster than the first access of the PE to this BRAM to ensure that memory requirements are served with quality.

- **Transparency**

An important design goal is to decouple the internal functionality of DOMMU from the PEs using it. Therefore, DOMMU's interface as well as its behaviour and timing performance has to be identical to that of traditional BRAM access. This is achieved by BRAM virtual address mapping which is transparent to the PEs, and maintaining a single clock cycle latency for BRAM access. Moreover, it is necessary to provide all the PEs with access to their allocated BRAM simultaneously via independent dedicated channels without any bandwidth sharing. To provide a PE transparently with memory when it is needed, automated dynamic BRAM (de)allocation is realized which should be enabled or disabled for different PEs independently at run-time, according to the application requirements.

- **Scalability**

DOMMU has to be designed with user-configured parameters to make it reusable and scalable in terms of the number of memory ports, number of BRAMs managed, their types and configurations. Moreover, the required hardware resources have to scale well with increasing numbers of memory ports and managed BRAMs. Additionally, the design has to provide integrated support for shared BRAM for communication between PEs through dual-port BRAM access, and should be extensible to integrate application-specific BRAM type templates.

- **Conservation of an optimal point in design space**

Since DOMMU replaces static allocation of BRAMs, the design space exploration [16] for the architecture using DOMMU has to consider bandwidth, latency and hardware resources. Independent dedicated channels between PEs and their associated BRAMs assure a latency of one clock cycle for memory

accesses. In order not to outweigh the gains of DOMMU, the hardware resources have to be kept minimal. This preserves the point in the design space of the original architecture, while enabling efficient utilization of BRAM resources by dynamic management.

A. Proposed Design

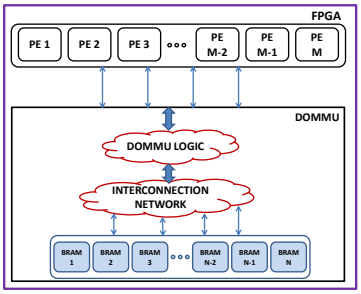


Fig. 1: Illustration of the general system overview of DOMMU.

In Figure 1, each PE is assigned one or more memory ports, by the user at design-time. Thesse memory ports interface with DOMMU for BRAM (de)allocation and access. **M** memory ports share access to **N** BRAM elements via an interconnection network as shown in Figure 1.

To manage this dynamic sharing while keeping the BRAM management transparent to the PEs, it must keep track of the BRAM configurations (width and depth) available "in stock", the BRAM assigned to each PE, the configuration details of this BRAM, how often the BRAM is accessed, and how much more or less BRAM is required by each PE at any point in time. To keep the BRAM management transparent to the PEs, an address mapping scheme ensures correct PE-BRAM association.

B. BRAM Organization and Address Translation Scheme

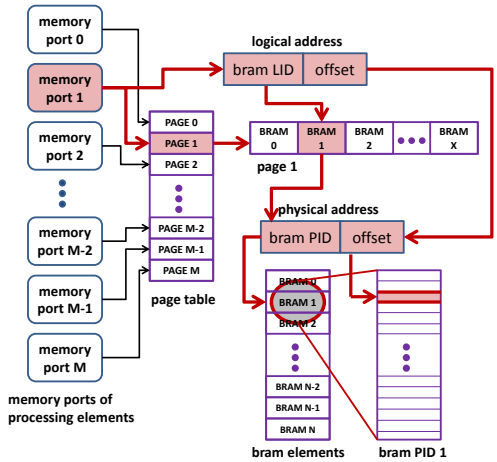


Fig. 2: Logical to physical address translation scheme of DOMMU.

The set of BRAM elements shown in Figure 2 and their physical configurations is the BRAM physical address space, which is realized by initializing a subset of the available BRAM resources on the device in different configurations (width X depth) depending on the design requirements.

To provide transparency to the PEs, the BRAM elements are also arranged in a logical address space, in the form of logical pages. Concepts of logical addressing and paging are borrowed from software memory management of operating systems [17], and employed similarly in the design of DOMMU. Each memory port is assigned a logical page which can be assigned up to **X** BRAM elements as shown in Figure 2. The BRAM elements are assigned a Logical Identification (LID) according to their order of assignment within the logical page. These LIDs are assigned at run-time independently of the Physical ID (PIDs) of the BRAM elements managed by DOMMU. Each memory port should "know" its logical page, its word width and depth. Each PE accesses its allocated BRAM by communicating the logical addresses via its memory port(s) to the DOMMU. The logical address is mapped to the physical address (BRAM PID and offset within the BRAM element) to access the correct data word.

DOMMU interfaces with the PEs via the memory ports shown in Figure 2, which introduces a degree of freedom to assign more than one memory port for each PE at design-time.

IV. Architecture of DOMMU

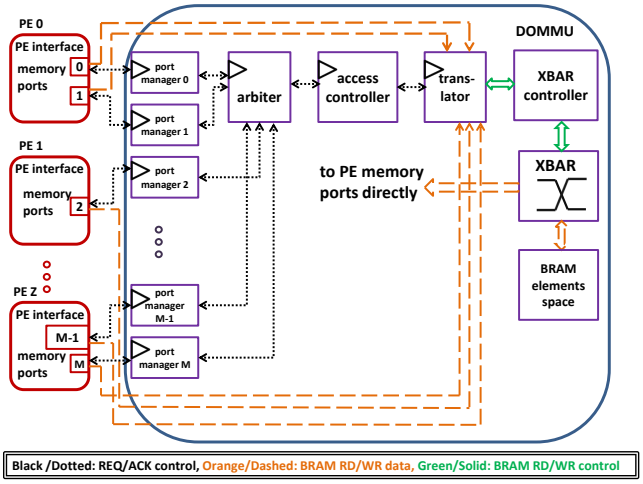


Fig. 3: Block diagram of DOMMU architecture.

A detailed block diagram of DOMMU and its components is illustrated in Figure 3.

Crossbar (XBAR) switch - The PE↔BRAM interconnection network required in DOMMU must allow all PEs to be physically able to access all the configured BRAM elements. Bi-directional communication is required to support both read and write access, as well as non-blocking switching to ensure that multiple simultaneous PE ↔ BRAM interconnections can always be established. The crossbar switch satisfies these requirements.

The original idea was to dynamically reconfigure the FPGA routing resources to implement the crossbar switch, or implementing the crossbar multiplexers using LUTs and reconfiguring their configuration contents by bitstream manipulation via internal dynamic partial reconfiguration of the corresponding FPGA configuration frames [18], in order to control the multiplexed output, as suggested by Hoo *et al.* in [19]. However, for ease of initial implementation and proof-of-concept, the crossbar is implemented in this work using regular multiplexers. It is realized using two crossbars: a uni-directional (PE → BRAM) crossbar for writing to BRAM, and a bi-directional (PE ↔ BRAM) crossbar for reading

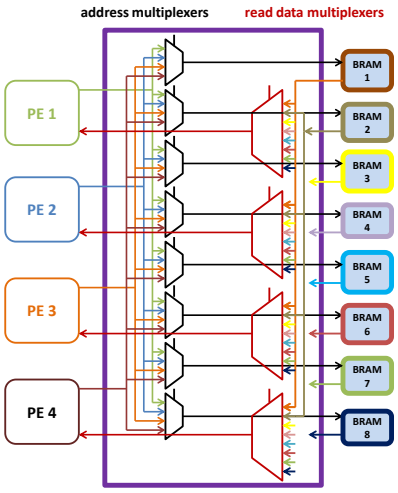


Fig. 4: Implementation of bi-directional 4 x 8 read crossbar using multiplexers.

from BRAM. Figure 4 presents a multiplexer-based 4x8 bi-directional crossbar switch. A crossbar controller takes the translated physical addresses required for all the memory ports and assigns the crossbar select and enable lines, to provide transparent BRAM access with single clock cycle latency. The crossbar switch is realized such that each PE can simultaneously read from and/or write to single-port or dual-port BRAM. Inter-PE communication is configured by mapping separate ports of a dual-port BRAM page to different PEs.

Address translator (BRAT) - The PEs communicate with the BRAMs by logical addresses. Hence, each memory port is assigned a BRAM Address Translator (BRAT), which performs the functionality described in Figure 2.

When a read or write access request is received through a memory port (orange/dashed path in Figure 3), BRAT maps this memory port to the associated logical page by quering an array which maps each port to its corresponding logical page and the allowed access credentials (RD, WR, or RD/WR) of this memory port to this page. If the address is out-of-bounds or involves illegal access, the incoming address is rejected, and the PE is flagged for requesting an illegal access. This feature enforces implicit memory access rights to ensure that each PE can only access its assigned memory.

BRAT also receives incoming control requests from a controller to update its stored arrays for new (de)allocations. ACK/NACK message reporting the status and details of each request is returned to the controller. Errors such as a full logical page that cannot be allocated more BRAM or an empty page that cannot be deallocated from are handled by returning the corresponding NACK message back to the controller. In general, all incoming control requests are acknowledged with ACK/NACK response messages communicated to the controller which indicate the details of status of the request. BRAT also keeps track of the logical page associated with each memory port, and the details of each logical page, such as its access credentials, word width, allowed maximum and actual depth. All details about the BRAM elements assigned to each page are also stored to ensure correct PE-BRAM association, correct logical-to-physical address mapping, and detection of illegal accesses.

Access controller (BRAC) - The BRAM Access Controller, or BRAC, is the controller component of DOMMU, which receives and handles BRAM (de)allocation requests (black/dotted path in Figure 3). It is realized as a five-state finite state machine controller. A PE can issue request to (de)allocate more than one BRAM element at a time to reduce control traffic, but only one element can be processed at a time by BRAC. Hence, BRAC receives the request, interprets it, and keeps track of the number of elements to process. It processes the (de)allocation of each element, one after the other, until the request has been fully processed, and a final ACK/NACK response message is returned to the PE to indicate the status and relevant details of the request (such as the number of BRAM elements allocated if it was not possible to serve the full request).

In order for BRAC to perform its functionality correctly, it keeps track of the complete BRAM elements space, and stores arrays of the details and counters of the unallocated, allocated, and deallocated (were once allocated) BRAM elements of each implemented BRAM configuration type. When a PE requests a certain BRAM configuration, BRAC queries the available BRAM elements of that specific configuration, and assigns BRAM elements to the corresponding logical page. When a BRAM element is (de)allocated from/to a logical page, BRAC updates its arrays and communicates these changes to the corresponding BRAT by a (N)ACK message. Illegal (de)allocation requests are handled by being denied and sending back the corresponding NACK message to the requesting PE.

As shown in Figure 3, a single shared channel and resources are dedicated for processing BRAM control requests. Requests can be issued by multiple PEs simultaneously, which requires an arbiter to schedule oustanding requests. Note that BRAM access occurs with single clock cycle latency since all PEs have dedicated BRAM read/write channels, and no sharing or arbitration is involved. However, control requests from multiple PEs cannot be processed simultaneously since this would result in conflicts in BRAM (de)allocations and arrays. Therefore, a single BRAM control channel is shared by all.

Arbiter - Arbitration using adaptive, dynamic and user-configurable priorities was implemented, since this was most suited for dynamic reconfigurable systems. The scheduling priorities associated with these PEs are dynamic and can change at run-time. Every memory port is assigned a priority, which is one of the three levels: low, medium or high, and this priority level is assigned as static or dynamic either at design-time or run-time. A static priority maintains its default value throughout operation, unless it gets re-assigned explicitly by the PE. At every clock cycle, all incoming requests from all memory ports are read and arbitration selects the request to serve. If the waiting time of a request exceeds a user-configured threshold and if the priority of that port is dynamic, then the corresponding priority is upgraded to the next level. The dynamic priority of a memory port also gets downgraded if its pending request gets served, and its request waiting time is smaller than a user-configured threshold. Configurable and dynamic priority arbitration is suitable for real-time embedded systems in which some running applications are more time-critical than others, and scheduling priorities can be adjusted accordingly.

Hierarchical arbitration is implemented in which higher priority is always reserved for all allocation requests followed by a lower priority for all deallocation requests because allocation

requests are more critical to the PE. Within every level of hierarchy, the assigned priorities are examined to schedule the highest priority request to be first served. Latency overhead due to arbitration is unavoidable yet critical, since it is a significant factor in the latency incurred in serving memory allocation requests, which is crucial for scheduling memory requests associated with real-time applications. Arbitration latency has a determinstic maximum which is a function of the number of PE memory ports configured and the maximum number of BRAM elements that can be requested at one time by any memory port. This latency should be considered when scheduling BRAM allocation requests, and is guaranteed, when dynamic automated BRAM allocation is enabled, to remain below the first access of the PE to the requested BRAM. This overhead can be reduced by minimizing the arbiter logic and dynamism. Moreover, more aggressive pipelining can be attempted in order to serve multiple BRAM requests at one time, although in the current architecture design this would result in inconsistencies in shared data arrays.

Memory port manager - Each PE memory port that interfaces with DOMMU consists of a dedicated BRAM access port and a control port for (de)allocation control requests. The BRAM access port constitutes of two independent ports. Data can be read from or written to one or both of them simultaneously which enables access of single-port BRAM as well as dual-port BRAM for double the bandwidth, and supports inter-PE communication, which is often required in real-time image processing applications. If a PE requires more BRAM bandwidth, additional memory ports can be configured for it. The control port is assigned a memory port manager which matches the requested BRAM type, word width, and number of words to the closest BRAM configuration (width X depth) available. This ensures that the internal BRAM management and configuration details remain transparent to the PE. Since this mapping is embedded and time-critical, and has to occur with minimal impact on timing performance and area overhead, this limits the maximum complexity of the methodology and logic implemented. There is no optimal resolution to this mapping problem due to the different optimization factors that can be considered such as speed, power or area utilization [20]. The methodology implemented in this work selects the match that minimizes in the number of BRAM elements assigned.

Automated dynamic (de)allocation of BRAM is one of the distinguishing features of DOMMU. This allows additional BRAM to be requested for allocation automatically when the assigned BRAM for the memory port is close to running out. This is indicated when the number of BRAM addresses that get written to, increase beyond a user-configured threshold. If the assigned BRAM remains idle and unaccessed for a number of clock cycles greater than a user-configured threshold, the BRAM elements (but one element, which requires explicit deallocation) get deallocated automatically. Dynamic (de)allocation can be enabled or disabled by each port manager at run-time according to the application requirements. This feature is based upon several simplifying assumptions that every incoming read/write access is a valid one, that every incoming write access is associated with a new BRAM address, and that when the number of idle cycles exceeds a certain threshold, that this BRAM is not required by the memory port anymore, and should be deallocated.

The currently supported control requests a PE can issue via a memory port to its memory port manager are: allocating a new single-port or shared BRAM logical page, deallocating a BRAM logical page, or a requested number of words from a logical page, or assigning a new priority to the concerned memory port. The parameters required for each request depend on the request code issued, and each request is acknowledged by a response message which indicates the details of the granted/denied request.

Component	Register Utilization [%]	LUT Utilization[%]
Port Manager	0%	1%
Arbiter	0%	<1%
BRAC	1%	2%
BRAT	<1%	2%
XBAR Controller	<1%	<1%
XBAR Switch	0%	**9%**
Total	3%	17%

TABLE I: Resource requirements of individual components of DOMMU for **ports_number** = 8, **max_bram_per_page** = 4, **bram_number** = 40 on Virtex-5 FPGA.

V. Results and Discussion

The main parameters of DOMMU that are configured by the user at design-time are the number of PEs, the number of memory ports which will interface with DOMMU, the number of memory ports assigned for each PE, the number of BRAMs, their configurations and types, and the maximum number of BRAM elements that can be assigned to one logical page. Results for different DOMMU configurations are presented in this section.

A. Resource Requirements

The resource requirements for DOMMU depend on the configuration of its parameters. Table I presents the fraction of resources required by each component of DOMMU within the total resources utilization for a configuration of 8 memory ports and 40 BRAM elements. LUT utilization is 17% with the highest 9% dedicated for the crossbar switch, owing to the number of multiplexers required to implement the switch. All percentages in this subsection are in reference to the FPGA Xilinx Virtex-5 LX110T.

Figures 5 and 6 illustrate the scalability of DOMMU and the impact of increasing the number of memory ports or BRAM elements on the resource utilization. In Figure 5, LUT utilization increases to 35% for DOMMU configuration with 16 PE memory ports and 40 BRAM elements. A linear growth in required resources can be observed from the plot, which indicates that the number of PE memory ports configured should be carefully selected with consideration for cost in resource consumption. Figure 6 also shows how LUT utilization increases from 5% to 15% as the number of BRAM elements configured increases from 5 to 100 with a fixed number of 4 PE memory ports. Increases in resource utilization are due to the increasing number of crosspoints (ports_number $\times$ bram_number) in the crossbar switch, as well as increasing volume of logic and arrays required to handle more memory ports or BRAM elements. However,

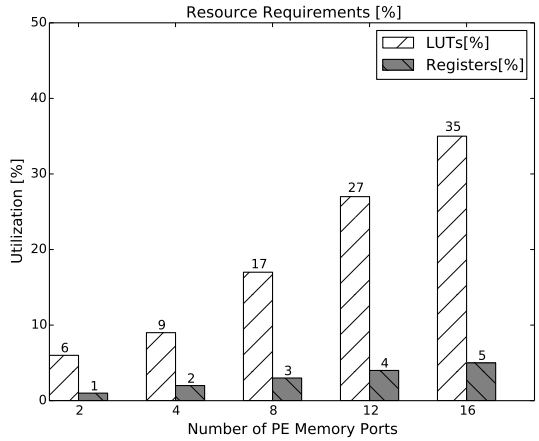


Fig. 5: Resource utilization for DOMMU for different number of memory ports and bram_number = 40 on Virtex-5 FPGA.

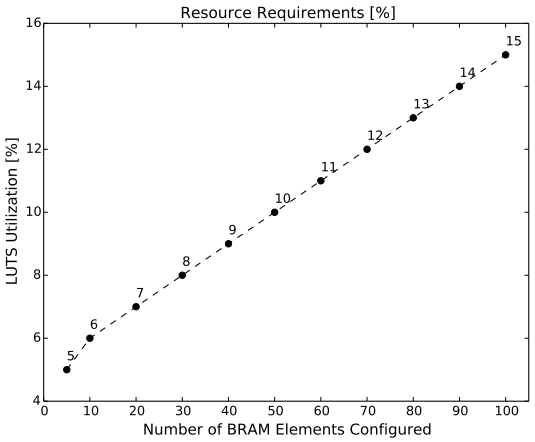


Fig. 6: Resource utilization for DOMMU for different number of BRAM elements and ports_number = 4 on Virtex-5 FPGA.

results indicate that altering the number of BRAM elements has a smaller impact than altering the number of PE memory ports configured on the LUT utilization of DOMMU. Such is expected since each memory port configured requires a corresponding memory port manager.

DOMMU can be operated at up to 140 MHz on the Virtex-5 device, which varies according to the configuration. Table II shows how the timing performance of DOMMU varies according to the number of memory ports and BRAM elements managed. As these increase, the maximum operating frequency decreases. Again, the number of PE memory ports has a greater

Number of PE Memory Ports bram_number = 40	Max Freq [MHz]	Number of BRAM Elements ports_number = 4	Max Freq [MHz]
2	140	5	135
4	111	20	123
8	89	40	111
12	75	70	98
16	59	100	95

TABLE II: Maximum operating frequency of DOMMU for different configurations measured post-synthesis on Virtex-5 FPGA.

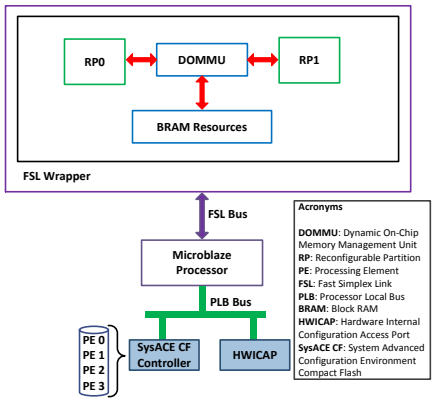


Fig. 7: Overview of the implemented test system for DOMMU.

impact than the number of BRAM elements. This imposes another cost constraint when configuring DOMMU: a trade-off between the number of PE memory ports and the desired operating frequency. The maximum operating frequency is constrained by the critical path involved in BRAM access which is the address mapping and interconnections establishment via the crossbar switch, since it is constrained to occur in single clock cycle to provide equivalence to traditional BRAM access. The measured frequencies in table II are approximately one quarter of the maximum frequency for traditional BRAM access that can be achieved for -1 speed grade Virtex-5 FPGA, which is 450 MHz. However, such clock frequencies are not very often achieved for practical systems. This is due to logic and wire delay, which often impose a critical path of a lower operating frequency, not very different from those in table II.

Furthermore, the scalability of DOMMU is critically determined by how the operating clock frequency varies with its configuration. The distribution of BRAM resources across the entire FPGA area and the need to enable access of multiple PEs to all of the configured BRAM resources strongly limits the maximum operating clock frequency that can be achieved for DOMMU, which can hinder its scalability and degrade performance. This can be tackled by imposing FPGA placement constraints and fine-grained floorplanning to define specific partition locations for the PEs, or grouping and categorizing BRAM resources according to their type and physical proximity from the PE partitions, and assigning higher allocation priority to closer BRAM elements. Another suggestion would be limiting access and sharing of PE partitions to only subsets of BRAM elements which are within a certain physical proximity of each partition.

B. Test Platform

To demonstrate the functionality of DOMMU and its impact on BRAM utilization, the system shown in Figure 7 is implemented, which consists of two reconfigurable partitions (RPs), each of which can host one or more PEs. An instance of DOMMU is configured with the BRAM resources. Dynamic partial reconfiguration of the reconfigurable partitions is controlled by a Microblaze processor which communicates with the reconfigurable partitions over a Fast Simplex Link (FSL) [21] bus. A compact flash (CF) holds the partial bitstreams required for reconfiguring each of the RPs, and the System ACE CF controller [22] manages the transfer of data to the FPGA. FPGA push buttons and DIP switches enable the user to trigger reconfiguration of one of the RPs with one of the

possible PEs at any time. The Microblaze reads and processes the user request, and accordingly reads the corresponding partial BIT file from the CF. The BIT file is then written to the Xilinx XPS Hardware Internal Configuration Access Port IP (HWICAP IP) [23] to reconfigure the FPGA partitions. The system was implemented on a Xilinx Virtex-5 LX110T FPGA; but can be easily implemented on other FPGA devices, since the HDL design code for DOMMU (BRAM elements are HDL-inferred and not explicitly instantiated to support portability) is vendor- and device- independent.

C. Case Study

A connected component labeling (CCL) algorithm [24] [25] is used as the case study to evaluate the functionality of DOMMU and to observe and compare performance, in terms of BRAM utilization for specific memory access patterns, between using DOMMU for managing the on-chip BRAM resources and traditional static on-chip memory allocation.

CCL is an essential step in many image processing applications that labels all image pixels into independent components depending on pixel connectivity, and computes features of the image components such as the bounding box or the area [5]. In the used architecture, the number of simultaneously processed image components can vary from 0 up to the width of the image [24] [25]. A proportional amount of run-time memory is required to store these component labels and their computed features. The two RPs in the test system shown in Figure 7 are occupied by two parallel CCL PEs, and can be reconfigured at run-time to be occupied with other PEs as well. An input image is divided spatially into two slices, each of which is processed by one of the PEs in parallel. Static memory allocation would need to accomodate the worst-case scenario for each of the CCL PEs is provided, which is costly and inefficient. Therefore, this makes CCL a good case study for adaptive memory management by DOMMU.

Measuring the improvement in BRAM allocation and utilization definitively is not possible, since the improvement achieved is context-dependent, i.e. for CCL it depends on the input image, and in general, it depends on the memory access pattern involved. Hence, improvement can only be measured quantitatively for individual memory access patterns.

Figure 8 shows the memory requirements of the CCL hardware architecture from [25] for two sample images: a binarized photograph in 8a and random noise in 8b. The maximum number of labels required is 4096 labels. This corresponds to 200 kBit for the extraction of the bounding box of each image component. Figure 8c shows that for processing the photograph with DOMMU, only 3% of the worst-case memory is actually required which corresponds to an improvement in BRAM utilization by a factor of 33, and for the less-probable-to-occur random noise image in 8b, only 25% is required as shown in 8d, which corresponds to an improvement in BRAM utilization by a factor of 4. For these scenarios, the difference between the actually required memory and the worst-case memory is made available by DOMMU for the other PEs on the FPGA, while still retaining BRAM resources to cover the worst-case scenarios. Moreover, it was also shown using the test platform that DOMMU deallocates the BRAM of the PE which gets reconfigured dynamically by another PE (which is communicated by the Microblaze processor via the FSL bus), and adapts the BRAM (de)allocations to the BRAM requirements of the new PE. This demonstrates how DOMMU adapts to the varying on-chip memory requirements of dynamically reconfigurable FPGA architectures.

VI. Conclusion

In this paper, a Dynamic On-chip Memory Management Unit (DOMMU) is proposed to support dynamic BRAM sharing among several processing elements in FPGA-based dynamic reconfigurable architectures, such that the BRAM allocation and utilization adapt to the variable run-time memory footprints of the PEs. A dynamic fine-grain control of BRAM (de)allocation, as opposed to previous static traditional approaches is introduced, as well as a virtual BRAM addressing scheme, and an automated dynamic memory (de)allocation algorithm, thus making DOMMU superior to previous architectures in terms of scalability, flexibility and its usability for reconfigurable computing in particular. DOMMU was realized on a Virtex-5 FPGA device, and its functionality was evaluated using connected component labelling (CCL) as a case study application. An improvement in BRAM utilization by a factor of 4 was demonstrated when using DOMMU for managing the on-chip memory requirements of CCL hardware architecture when processing a random noise image, and a factor of 33 for a more realistic image.

The work in this paper can be extended in different directions. Exploring the alternative implementation approaches for the crossbar switch presented originally in Section IV is recommended, in order to reduce the logic resources consumed. DOMMU needs to be more extensively tested with more potential applications to assess its gains and performance in a wider range of real life uncontrolled scenarios. The timing performance needs to be more thoroughly analyzed, and different measures and approaches to reduce the degradation of the maximum operating clock frequency should be investigated, as discussed in Section V.

Acknowledgments

The authors would like to thank the German Research Foundation (DFG) for the financial support for this work carried out within Si 586 7/1 belonging to DFG-SPP 1423 and SI 587/11-1 belonging to DFG-SPP 1740.

References

[1] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The Erlangen slot machine: A dynamically reconfigurable FPGA-based computer," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, pp. 15–31, 2007.

[2] M. Platzner, J. Teich, and N. Wehn, *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*. Springer, 2010.

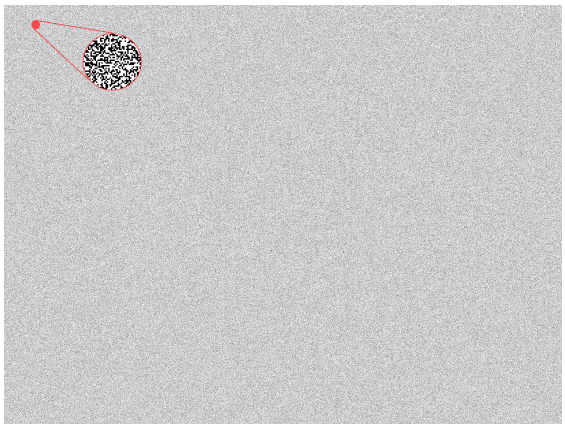
[3] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2010.

[4] I. Koutras, A. Bartzas, and D. Soudris, "Adaptive dynamic memory allocators by estimating application workloads," in *2012 International Conference on Embedded Computer Systems (SAMOS)*. IEEE, 2012, pp. 252–259.

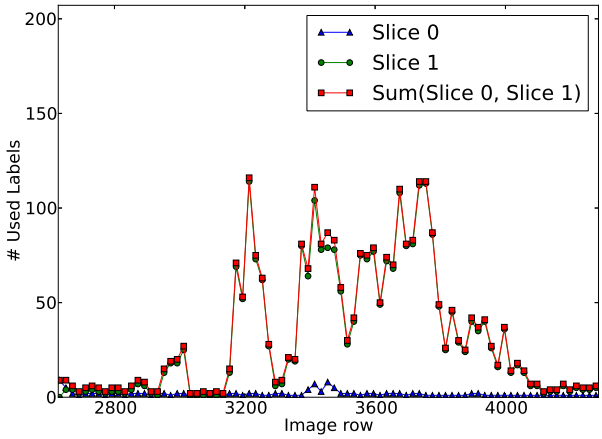
[5] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM (JACM)*, vol. 13, no. 4, pp. 471–494, 1966.

[6] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*. Springer, 1995, pp. 1–116.

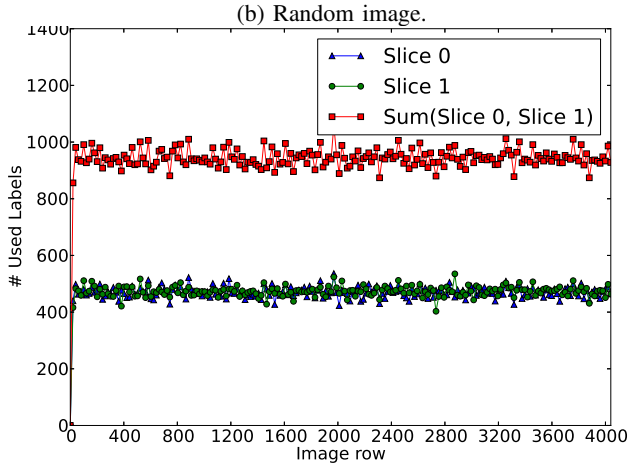
(a) Cathedral Cove, New Zealand.



(b) Random image.



(c) CCL memory requirements for Cathedral Cove image (a).



(d) CCL memory requirements for random image (b).

Fig. 8: CCL memory requirements for 12 MegaPixel images (4096x3072).

[7] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *14th Euromicro Conference on Real-Time Systems, 2002*. IEEE, 2002, pp. 41–49.

[8] Rtems real time operating system (RTOS). [Online]. Available: http://www.rtems.org/

[9] D. Atienza, S. Mamagkakis, F. Catthoor, J. Mendias, and D. Soudris, "Reducing memory accesses with a system-level design methodology in customized dynamic memory management," in *2nd Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)*, Sept 2004, pp. 93–98.

[10] M. Shalan and V. J. Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2000*, vol. 17, no. 19, 2000, pp. 180–186.

[11] J. M. Chang and E. F. Gehringer, "A high performance memory allocator for object-oriented systems," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 357–366, 1996.

[12] D. Goehringer, L. Meder, M. Hubner, and J. Becker, "Adaptive multi-client network-on-chip memory," in *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp. 7–12.

[13] I. Anagnostopoulos, S. Xydis, A. Bartzas, Z. Lu, D. Soudris, and A. Jantsch, "Custom microcoded dynamic memory management for distributed on-chip memory organizations," *IEEE Embedded Systems Letters*, vol. 3, no. 2, pp. 66–69, 2011.

[14] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of distributed shared memory architectures for noc-based multiprocessors," *Journal of Systems Architecture*, vol. 53, no. 10, pp. 719–732, 2007.

[15] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[16] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integrated VLSI Journal*, vol. 38, no. 2, pp. 131–183, Dec. 2004.

[17] W. Stallings, *Operating Systems: Internals and Design Principles*. Pearson/Prentice Hall, 2008.

[18] "Virtex-5 FPGA Configuration User Guide," Xilinx, Inc., Oct 2012.

[19] C. H. Hoo and A. Kumar, "An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay," in *22nd 2012 International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 400–406.

[20] "LogiCORE IP Block Memory Generator v6.1," Xilinx, Inc., Mar. 2011.

[21] "LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)," Xilinx, Inc., April 2010.

[22] "System ACE CompactFlash Solution," Xilinx, Inc., April 2002.

[23] "LogiCORE IP XPS HWICAP(v5.00a)," Xilinx, Inc., July 2010.

[24] M. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud, and S. Simon, "A memory-efficient parallel single pass architecture for connected component labeling of streamed images," in *2012 International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 159–165.

[25] M. J. Klaiber, D. G. Bailey, S. Ahmed, Y. Baroud, and S. Simon, "A high-throughput FPGA architecture for parallel connected components analysis based on label reuse," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 302–305.