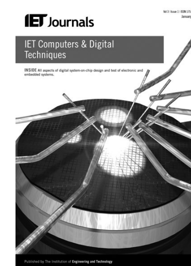
www.ietdl.org

Published in IET Computers & Digital Techniques
Received on 16th April 2014
Revised on 23rd June 2014
Accepted on 7th August 2014
doi: 10.1049/iet-cdt.2014.0078



ISSN 1751-8601

Morphable hundred-core heterogeneous architecture for energy-aware computation

Nuno Neves, Henrique Mendes, Ricardo Jorge Chaves, Pedro Tomás, Nuno Roma

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
E-mail: Nuno.Roma@inesc-id.pt

Abstract: Given the increased demand for high performance and energy-aware computational platforms, an adaptive heterogeneous computing platform composed of 100+ cores is herein proposed. The platform is based on an aggregate of multiple processing clusters, each containing multiple processing cores, whose architectures are adapted, in execution time, to the instantaneous energy and performance constraints of the software application under execution. This adaptation is ensured by a sophisticated hypervisor engine, implemented as a software layer in the host computer, which keeps a permanent record of a broad set of performance counters, gathered from the execution of each core in the field-programmable gate array (FPGA), in order to dynamically determine the optimal heterogeneous mix of processor architectures that satisfy the considered constraints. By issuing convenient reconfiguration commands to the reconfiguration engine, implemented in a static portion of the FPGA, partial dynamical reconfiguration mechanisms ensure a runtime adaptation of the cores that integrate each cluster. When compared with static instantiations of the considered many-core processor architectures, the obtained experimental results show that significant gains can be obtained with the proposed adaptive computing platform, with performance speedups up to 9.5×, while offering reductions in terms of the consumed energy as high as 10×.

1 Introduction

The increasing demand for computational processing power observed along the past decade has driven the development of heterogeneous systems composed of one or more processing devices. Such systems typically include a host general purpose processor and one or more accelerating devices, such as a graphics processing unit or a field-programmable gate array (FPGA), each integrating multiple processing elements (PEs). Although these systems allow for a significant application acceleration, it is of fundamental importance to improve processing performance while minimising the energy consumption. As a consequence, new and highly efficient processing frameworks must be developed.

To tackle the development of energy-efficient platforms, many techniques have been proposed. This includes turning-off parts of the processor [1], dynamic selective de-vectorisation [2] or using dynamic voltage and frequency scaling [3, 4] to decrease energy consumption whenever the computational requirements decrease. Recently, researchers have also turned to multi-core heterogeneous systems composed of high-performance 'big' cores and low-power 'small' cores (e.g. the ARM big. LITTLE) to decrease the power consumption of the whole system, while providing similar processing performances [4, 5]. These systems typically exploit a common instruction set architecture (ISA) among all the cores, in order to facilitate task migration from the 'big' to the 'small' cores (and vice versa), which allows for a fast and efficient switching between high performance and low-power scenarios, depending on the application requirements and constraints. Nevertheless, to adequately explore the existing resources on heterogeneous multi-core systems according to a given execution profile, an adequate scheduling needs to be performed.

To efficiently manage multiple running tasks on such systems, several scheduling algorithms have been developed. Koufaty *et al.* [6] identified key metrics that characterise the application under execution, including the core type that best suits its resource needs. Cong and Yuan [7] propose the combination of static analysis and runtime scheduling to achieve an energy-efficient scheduling on Intel's QuickIA heterogeneous platform [8]. Van Craeynest *et al.* [9] proposed the usage of a performance impact estimation as a mechanism to predict which workload-to-core mapping is likely to provide the best performance.

Nonetheless, further processing power and energy efficiency can still be exploited by adapting the computing system to the characteristics of the running multi-threaded applications. Some possible approaches are the application of network reconfiguration or adaptive core interconnection topologies [10–12], by changing the cache configuration [13] or by performing core morphing [14, 15]. Such advances have been significantly aided by the recent technological improvements in FPGA devices, which allow for a fast partial dynamic reconfiguration. This provides the

© The Institution of Engineering and Technology 2014

possibility to dynamically reconfigure a selected region, whereas the remaining logic continues to operate in an uninterrupted way [16].

In this specific domain, several high performance and adaptable many-core heterogeneous systems have been proposed to exploit these reconfigurable capabilities. ReMap [17] enables custom computation to be integrated with the reconfigurable communication scheme. In [18], reconfigurable multiprocessor systems that allow multiple configurations to coexist by using reconfigurable co-processors with multiple cores are presented. On a stream computing approach, Caspi *et al.* [19] proposes a design that incorporates a single central processing unit and multiple reconfigurable computing blocks, with data streams being transferred between the blocks over a dedicated interconnect. Different levels of parallelism can also be exploited in reconfigurable heterogeneous systems [20] by including reconfigurable ISA support in multi-core processors, providing an adaptive fine-grained parallelism to an already coarse-grained parallelised architecture. Lorenz *et al.* [21] demonstrated that dynamic reconfiguration can be efficiently used to optimise a system and save energy, and compared the energy that is required in the reconfiguration process with the potential saving that is introduced by a dynamic and adaptive change of the computing units of the processing system. However, the existing state-of-the-art lacks adequate responsiveness and adaptability to the tasks being executed.

In this paper, an adaptable and scalable architecture is proposed that not only supports more than 100 heterogeneous cores, but can also adapt its characteristics according to application requirements, by taking advantage of the partial dynamic reconfiguration capabilities of modern FPGA devices. The main feature and contribution of the proposed approach is the ability to monitor the performance of the PEs as they execute different kernels, in order to autonomously and immediately determine the most suitable architecture according to the current execution scenario. Given this, the proposed framework is able to change in real-time the architecture of each of the PEs, in order to achieve the best possible performance/energy efficiency.

This paper is organised as follows. Section 2 presents an overview of the proposed dynamic many-core heterogeneous architecture, detailing its key components. Section 3 details the implementation choices to obtain an evaluation prototype for the proposed framework. Section 4 evaluates the performance of the obtained system and the trade-off between adaptability and reconfiguration cost. Concluding remarks are presented in Section 5.

2 Dynamic many-core heterogeneous architecture

The proposed reconfigurable processing platform targets the acceleration of the parallel sections of one or more applications running on a host computer. Thus, a parallel programme (e.g. written in Open MP) could be partially or fully migrated to the PEs instantiated on the FPGA device, depending on performance, power and energy requirements. Furthermore, typical applications can be divided into phases [9], where each phase has different requirements, such as memory bandwidth, instruction-level parallelism, data-level parallelism and even functional unit requirements. Under these circumstances, it is very important to allow the accelerating platform to adapt to the application(s) requirements in real-time. The proposed platform aims at solving this issue by monitoring the performance counters of each PE, when executing the application threads, and to reconfigure each PE according to the defined system goal. By taking into account the performance counters, the system is able to autonomously identify the most suitable PE configuration in order to maximise performance, to minimise energy and/or power consumption.

Since the proposed platform targets the acceleration of highly parallel applications, a high number of PEs need to be supported. Naturally, depending on their complexity, a different number of PEs may co-exist on the reconfigurable fabric at different time intervals. For example, while the reconfigurable fabric may have space to hold over 100 ‘smaller’ PEs, a much smaller set of ‘bigger’ PEs may be supported. Under such constraints, multiple ‘small’ PEs need to be swapped out when reconfiguring the system to include a ‘bigger’ PE. However, when ‘smaller’ PEs are better suited to perform the required computation (e.g. because the application current phase requires no floating-point operations or does not allow exploring data and/or instruction-level parallelism), trading a single ‘big’ PE with a ‘small’ PE may leave reconfigurable logic unused. As a result, if one considers a fine-grained reconfiguration granularity, a likely fragmentation of the reconfigurable logic will occur. To avoid such a situation, it is herein considered that the reconfigurable fabric is equally distributed into fixed-sized clusters, where reconfiguration always occurs at the level of a full cluster.

The architecture of the proposed platform is presented in Fig. 1*a*. It comprehends a scalable multi-core accelerator, implemented on an FPGA device, tightly coupled with a host computer through a high-speed interconnection bus (in the considered case, the PCIe bus). The accelerator is composed of a dense and heterogeneous many-core processing structure, organised in several computing clusters each composed of multiple processing cores. The type and number of cores is controlled by a Hypervisor module, implemented in the host computer, which is responsible for permanently monitoring the accelerator execution and for issuing appropriate reconfiguration commands that adapt the internal architecture of each individual cluster to the instantaneous characteristics of the application being executed. To keep a permanent evaluation of the performance that is being offered by each instantiated cluster, the Hypervisor keeps a record of a comprehensive set of performance metrics relative to each core.

The following sections present a detailed description of the main components that incorporate the proposed reconfiguration framework, namely: (i) the Hypervisor module, which schedules the execution of kernels on the available processing cores and manages the framework reconfiguration by issuing commands to the reconfiguration engine; (ii) the reconfiguration engine performing the actual partial reconfiguration; (iii) the processing clusters, where the application kernels are executed; and (iv) the shared-memory and inherent synchronisation mechanisms.

2.1 Hypervisor

The adaptive nature of the proposed heterogeneous structure is ensured by a Hypervisor module, which is implemented in the host computer (see Fig. 1*a*). This software module provides a bridge between the reconfigurable hardware

© The Institution of Engineering and Technology 2014
IET Comput. Digit. Tech., pp. 1–14
doi: 10.1049/iet-cdt.2014.0078

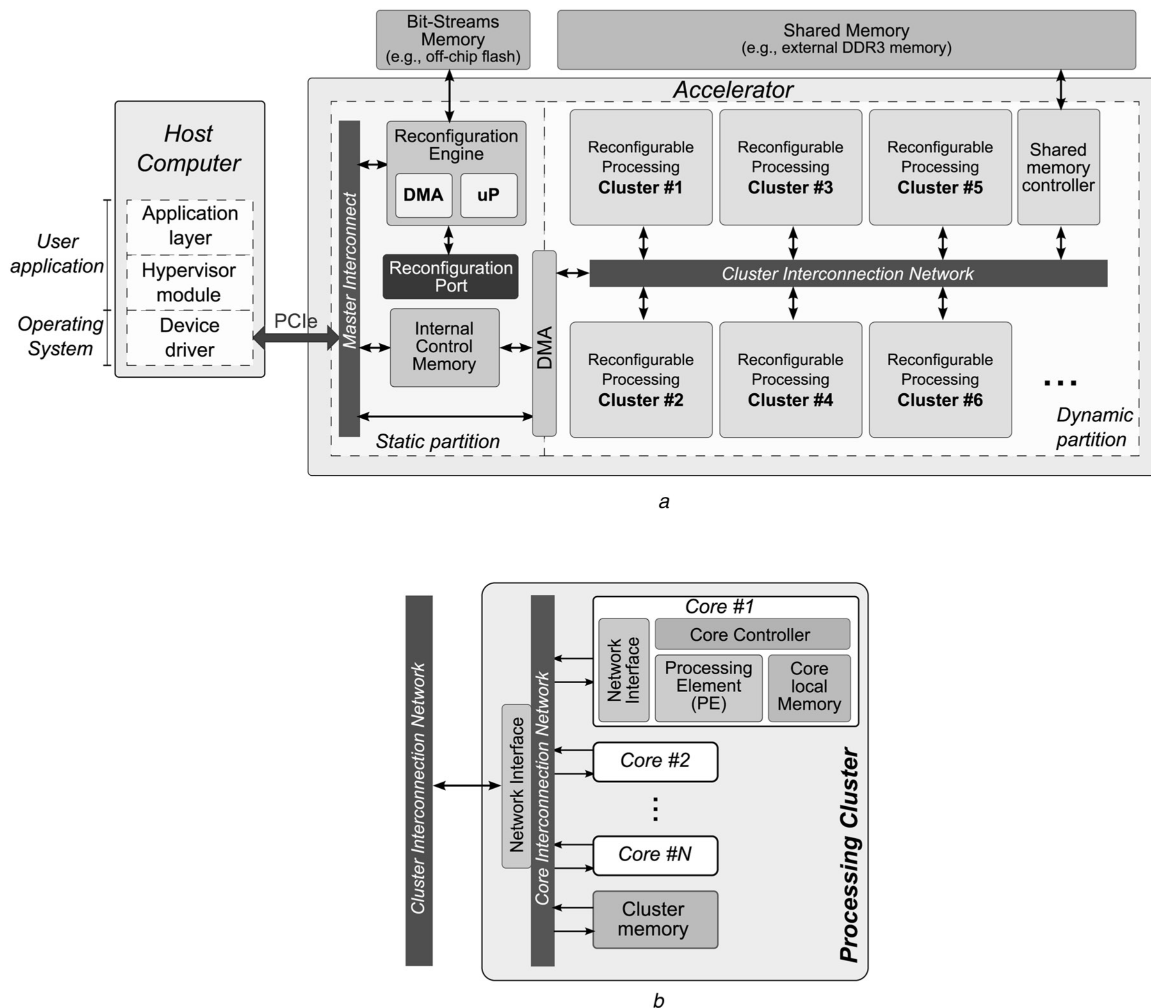


Fig. 1 *Proposed reconfigurable multi-core heterogeneous architecture*
a Block diagram of the whole processing system
b Internal block diagram of a processing cluster

resources that are offered by the accelerator architecture and the application executing in the host. Accordingly, the Hypervisor is responsible for three important tasks: (i) assigning application kernels to the several allocated cores in the instantiated clusters; (ii) keeping a permanent record of the processing performance that is being offered by each allocated core in the accelerator; and (iii) issuing of appropriate reconfiguration commands to the reconfiguration engine, located in the FPGA accelerator. To provide full compatibility with the existing application, the Hypervisor provides a complete abstraction of the underlying selection, adaptation and reconfiguration process.

To adequately exploit the processing capabilities of the proposed reconfiguration framework, the Hypervisor software module determines the processing topology that is most adequate to each application kernel that is being accelerated. For such purpose, it receives and maintains a permanent record of performance counters measured at each core in the instantiated processing clusters. With such information, it first measures the actual operational and computational intensity of each kernel, and further estimates the total execution time, power and energy consumption on the currently available processing cores. After obtaining these values, it re-estimates the same metrics by considering the reconfiguration of a processing cluster.

To decide on the best possible action, namely whether to execute the processing kernel on a currently available core or to reconfigure a cluster and then perform the computation, a set of policies are available, namely: (i) overall minimisation of the execution time; (ii) maximisation of the processing performance while establishing a ceiling for the total power consumption; and (iii) minimisation of the energy consumption, while guaranteeing a minimum quality of service (QoS), that is, performance level.

To improve the energy and power consumption, the hypervisor may decide to entirely disable a given processing cluster, by reconfiguring it as a blank (inactive) box. This will result in powering off the corresponding region on the FPGA device. Many reasons may lead to such a decision, namely: (a) when there are no further processing kernels to be executed on the accelerator (e.g. the application has insufficient parallelism), allowing to minimise the overall power consumption; (b) when the power budget will be exceeded by the active processing clusters, thus requiring parts of the reconfigurable area to be disabled; or (c) when the required performance level has been reached.

Finally, if the Hypervisor decision is the reconfiguration of a processing cluster, it then issues appropriate commands to

© The Institution of Engineering and Technology 2014

the reconfiguration engine (at the FPGA fabric) to adapt the processing structure.

2.2 Reconfiguration engine

To provide a convenient abstraction layer between the Hypervisor and the actual FPGA reconfiguration process, a reconfiguration engine was implemented and allocated on a static region of the FPGA device, that is, a region that will not be subjected to changes. This reconfiguration engine receives the reconfiguration commands from the Hypervisor, such as the cluster ID and a particular cluster configuration, and orchestrates the entire reconfiguration process, namely download of the partial configuration bitstream from the external memory and upload to the reconfiguration port. Once completed, it signals the Hypervisor about the completion of the reconfiguration.

To accomplish this task, the reconfiguration engine comprises a minimalistic microcontroller that handles the following tasks: (i) acknowledging the reconfiguration commands issued by the Hypervisor; (ii) initiating the reconfiguration interface and reading the required reconfiguration bitstreams from the bitstream storage memory; and (iii) ensuring the complete and ordered transfer of the particular bitstream to the reconfiguration port; and finally signalling the Hypervisor that the reconfiguration process has completed.

To make sure that the remaining system continues to operate normally, while still allowing the scheduling of tasks to any other processing cluster (apart from the cluster being reconfigured), dynamic partial reconfiguration is used. This reconfiguration technique allows for a particular region of the FPGA device to be reconfigured while the remaining regions remain unchanged and working.

Thanks to this permanent adaptation of the reconfigurable fabric to the application under execution, it is possible to dynamically deploy a more dedicated and better suited computational structure to the computation being performed, thus obtaining a more energy-efficient computational structures with better performances. Moreover, since each computing cluster is well defined, it can also be configured as an inactive blank box. By doing so, the logic corresponding to that particular region can be deactivated, resulting in a substantial reduction of the static energy consumption. Hence, in power critical systems, the amount of active logic regions can be finely controlled, in order to limit the peak power consumption. When more power is available, more logic regions and consequently more computational cores can be activated.

2.3 Processing clusters

As stated above, several core architectures, with different processing capabilities, can be devised and used in the proposed reconfigurable framework. The particular set of adopted core topologies can be fitted to different kinds of kernels that naturally depend on the computational complexity of the accelerated application. In particular, either programmable or fully dedicated processing cores can be considered in each instantiated cluster, as long as adequate interface structures are provided in each core input/output (I/O) interface.

With the reconfiguration process in view, the processing cores are grouped together in homogeneous clusters (see Fig. 1*b*). Since each computing cluster is well circumscribed within equal sized regions of the FPGA fabric, the number of identical cores that can be instantiated in each individual cluster depends not only on the amount of hardware resources that are available for each cluster region, but also on specific restrictions of the adopted internal interconnection bus, as described in Section 3.2. Hence, given the different requirements in terms of hardware resources of each core topology, clusters of more complex cores contain fewer processing cores than clusters of simpler cores. Furthermore, in different prototyping FPGA technologies, the area reserved for a cluster region can take different sizes depending on the distribution and amount of resources in the FPGA. Hence, this design approach provides a higher degree of granularity for the dynamic reconfiguration procedure, alleviating the resulting reconfiguration overhead. Accordingly, instead of individually reconfiguring each core, all the cores in a given cluster are reconfigured at once.

The communication and interface mechanisms between the cores and the host computer are managed through a dedicated controller, associated to each core. This controller was specially designed to: (i) start and monitor the execution of the core, (ii) receive the kernel parameterisation and (iii) transmit performance metrics to the Hypervisor module (at the host), for subsequent analysis.

Besides the previously described reconfiguration engine, the static portion of the FPGA fabric also includes a memory controller that provides a convenient shared-memory interface to the dynamically instantiated processing cores. Such shared-memory can either be implemented by using the internal memory resources within the FPGA, or even by using an external memory connected to the FPGA device.

In addition to this shared-memory space, a restricted set of dedicated memory positions, managed by an atomic access scheme, are also provided to the implemented cores, in order to support the implementation of convenient synchronisation mechanisms in the parallel processing structure.

3 Architecture design and implementation

Following the above general description of the reconfigurable heterogeneous computational platform herein proposed, this section details several key aspects in terms of its design and implementation on an FPGA device. This description initiates with the presentation of the control infrastructure that is tightly connected with each processing core. The communication infrastructure of the framework is also explained, including several aspects related with the intra-cluster communication bus, shared by all cores in each cluster, but also with the main system bus, connecting all clusters with the host computer. A detailed description of the reconfiguration engine is then presented, covering several aspects concerned with its internal architecture and implementation. This section concludes with a description of the considered replacement and reconfiguration policies that are implemented by the Hypervisor software layer, in order to optimise the achievable performance in terms of computational throughput and energy consumption.

3.1 Processing cores

To maximise the flexibility and versatility of the proposed reconfigurable structure, the heterogeneous accelerator architecture was designed by following a fully modular

© The Institution of Engineering and Technology 2014

doi: 10.1049/iet-cdt.2014.0078

approach. In particular, each processing core was designed in order to be as independent as possible of the underlying PE architecture. In fact, the only imposed restriction to the architecture of the PEs is concerned with the provision of straightforward monitoring and communication interfaces to the attached core controller.

Fig. 1*b* depicts the main components that comprise this core infrastructure. Each core controller is implemented by a finite-state machine, ensuring the coordinated execution of the following four sequential states in the core: (i) wait for a configuration word from the host computer, while maintaining the PE in a reset (idle) state; (ii) release of the PE from idle state (on reception of a host message) and wait until the PE has finished reading the configuration word; (iii) monitoring of the PE execution, while keeping record of a set of performance counters; and (iv) transmitting the measured performance counters to the host and reverting of the PE to a reset state, on assertion by the PE that the kernel execution has completed.

To accomplish this coordination, two separate communication interfaces are featured in each core controller: a host communication interface and a PE communication interface. The 'host interface' is composed of two unidirectional channels, compatible with the AXI-Stream protocol [22]. This interface is used by the core to receive configuration words from the host and to send packets to the host, containing the measured performance counters (as described above). The 'PE interface' is dependent on the adopted PE architecture and is used by the controller to gather relevant information about the execution and performance of the PE. It is usually implemented by a simple interface, which can be either memory mapped to the PE or connected through a custom I/O interface.

By default, the controller monitors the number of clock cycles that is required to execute a specific kernel. However, depending on the considered application, it is also possible to monitor other parameters, which are specifically customised based on the underlying architecture of the PE. Examples of additional counters currently implemented include the number of specific operations (e.g. integer divisions or floating-point additions or multiplications), counting branch mispredictions, identification of specific function calls (e.g. to detect operations that are performed through software libraries, instead of hardwired instructions). Hence, by providing this monitoring flexibility, it is possible to offer the Hypervisor with the necessary means to optimise any application in terms of the performance-energy balance, by easily tuning the reconfiguration policies to the specific characteristics of the available PEs and to the requirements of the application kernels to be accelerated.

Finally, depending on the accelerated application and on the adopted PE architecture, optional 'programme' and 'data' local memories can also be accommodated inside each core. Such memory devices may either comprise an attached cache controller or may be characterised by a non-coherent access mechanisms, comprised by a straightforward scratch-pad memory. Independent of the considered approach, such memories will represent the first level of the accelerator memory hierarchy.

3.2 Communication and interfacing networks

A fully compliant bus based on the AXI-Stream protocol [22] was adopted for both the 'core interconnection' and the 'cluster interconnection' networks (see Fig. 1). In fact, despite the several available Xilinx IP cores that implement an AXI-Stream interconnect fabric, it was decided to implement a lighter, but still fully compliant and custom interconnection, since not all the protocol signals are required and it is only necessary to create communication channels between the host and the cores. Moreover, this decision was taken with hardware resource overhead reduction in mind, since the complexity of the custom module is much lower than the original intellectual property (IP) core.

The implemented interconnection provides single-cycle communication mechanism between up to 16 peripherals which, in accordance to the protocol, corresponds to 16 AXI-Stream Master and 16 AXI-Stream Slave ports. Consequently, this interconnection features two independent unidirectional channels: a one-to-many channel and a many-to-one channel. The first channel routes data signals to the corresponding port (core) by using a decoder driven by the 4 bit destination TDEST signal. The second channel is managed by a round-robin arbiter, with a priority function based on the equations presented in [23], being the TVALID and TLAST signals used as the request and the end-of-burst acknowledge signals, respectively.

As such, and depending on the area restrictions described in Section 2.3, each cluster can accommodate up to 15 processing cores, being one of the interconnection ports reserved for outer-cluster communication. Hence, by daisy-chaining together a number of instantiations of the interconnection module through a specially designed bridge, it is possible to create a communication network with any number of levels.

To support the interconnection between the clusters and the host computer, a specific connection to the AXI4 bus was implemented using the Xilinx IP AXI direct memory access (DMA) [24] core. This IP core provides a bridge between an AXI-Stream interface and an AXI4 interface, allowing the translation of a stream-based communication to a memory-mapped communication. The AXI4 bus used in the prototyping device connects to the PCIe external interface and, from there, to the host computer.

Despite the adopted simplifications in the communication protocol, it still ensures the required set of functionalities, as well as the flexibility to implement additional features. To initiate each core execution, the host sends one 32 bit word to the target core controller, containing the necessary information for the core execution. At the end of the execution, the core controller sends a packet to the host with a 32 bit word reserved for a return message, followed by a configurable amount of 32 bit words containing the measured performance counter values. The most-significant 8 bits of each packet word are reserved for the tuple ('cluster_id, core_id'), containing the cluster and core identifications.

Finally, to allow a flexible access of each core to the shared local memory in the cluster and to the external memory, a different interconnection module was derived from the previous one and provided as a second layer of the interconnection network. Hence, while maintaining the same base structure, it is possible to obtain a single-cycle, arbitrated and shared-bus interconnection. This is achieved by exchanging the TDEST signal with an address signal, named TADDR, and by including memory and core interfaces, that see the unidirectional channels as a single bidirectional channel. Moreover, by including extra signals and providing the appropriate controllers, coherent cache-based memory hierarchies can also be implemented.

© The Institution of Engineering and Technology 2014

3.3 Reconfiguration

To provide the reconfiguration capabilities required to implement the proposed architecture, the Xilinx Virtex-7 FPGA was selected as target technology. It provides three different configuration ports to perform the reconfiguration, namely: joint test action group (JTAG), SelectMap and internal configuration access port (ICAP). The main differentiating factors between them are the accessibility and the entity that is responsible for the reconfiguration process. Each configuration port also provides different data width and working frequency interfaces, resulting in different reconfiguration throughputs.

The JTAG interface is a configuration port external to the FPGA, commonly used to load the initial configuration into the device, by directly interfacing it with the external flash memory, which stores the initial configuration file. This port provides a 16 bit configuration data-port, with a maximum frequency of 40 MHz. The SelectMap interface is also external, but with a higher reconfiguration throughout. Finally, the ICAP, which is essentially an internal version of the SelectMap, has a 32 bit data-port supporting a writing bandwidth of up to 100 MHz. Being internal to the FPGA, it allows for the reconfiguration process to be controlled by an entity instantiated within the device itself. Since in the proposed framework the core reconfiguration procedure is to be controlled by the reconfiguration engine, inside the FPGA, with the host computer serving as the Hypervisor, the ICAP port presents itself as the most suitable interface.

Besides these reconfiguration means, the targeted FPGA technology also supports Multiboot reconfiguration. This reconfiguration capability offers the possibility to load different full-configuration images in few cycles, allowing for different configuration layers to switch dynamically. However, this reconfiguration process only allows for 'full-device' reconfiguration, not providing the needed reconfiguration granularity. In contrast, partial dynamic reconfiguration allows the reconfiguration of specific processing groups (clusters) to be performed one by one, adjusting the system to the desired configuration.

To allow for a modular approach to the reconfiguration procedure that defines the several instantiated clusters, it is necessary to constraint each module to a specific and well-defined region of the reconfigurable fabric of the FPGA. Accordingly, in order to ensure that the reconfiguration of each assigned cluster only changes a predefined region in the device, appropriate region delimitation has to be applied to each processing cluster. Hence, by evaluating the area resources required by each resulting computational cluster, it is possible to define the required reconfiguration region of each cluster, as well as the maximum amount of supported clusters in the device. Another limiting factor in mapping the clusters to the configurable logic is the location of both the PCIe module and the ICAP interfaces. In fact, these locations are somewhat more conditioned, since they cannot be included in a region with an assigned reconfigurable module. These two modules are implemented by hard-cores, being allocated in specific regions of the device. Each of these regions is delimited using the floor-planner tool in order to reserve areas for each specific architecture module. This ensures that the reconfiguration engine can load the partial bitstream with no risk of the process affecting the on-going activity of the remaining cores or their communication with the Hypervisor.

In this particular implementation, the developed reconfiguration engine (illustrated in Fig. 2) is composed by a Xilinx ICAP controller (AXI HWICAP), an external memory controller connected to the on-board linear flash, a MicroBlaze microprocessor, an AXI memory controller connected to a 4 kB FIFO and an AXI PCIe bridge, all interconnected by an AXI4 bus.

Since the reconfiguration of the clusters is triggered by the Hypervisor, in the host, a set of flags is used to communicate with the reconfiguration engine. These flags are implemented on a shared block RAM (BRAM), accessible to both the host computer and the microcontroller in the internal reconfiguration control logic, via the PCIe bridge. This shared memory is used to trigger the reconfiguration command, to inform the host computer of the reconfiguration conclusion, and as an indicator of which configuration is to be loaded. With this approach, the host computer does not need to actively wait for the conclusion of the reconfiguration process, being allowed to continue processing the information obtained from other clusters that might be still running.

In this particular implementation, the configuration bitstreams are stored on the on-board linear flash. This flash is used both for loading the initial full-configuration when the system boots (since it is a non-volatile memory) and to store the partial bitstream configurations. The option for using this flash memory to accommodate the repository of the partial bitstreams (instead of using the external RAM memory) is to avoid any impact in the computation throughput when both the reconfiguration engine and the processing cores would be simultaneously accessing the external RAM memory. With this solution, it is possible to have a reconfiguration being performed while the computational cores access the main memory to access the data.

On receiving the command with the identification of the required configuration, the microcontroller of the reconfiguration engine issues read commands to the linear flash controller, in order to obtain the bitstream header. This header contains the information regarding the configuration bitstream size, thus obtaining the amount of bytes that need to be sent through the ICAP. After this initial phase, two approaches can be taken to carry out the actual reconfiguration. In the first option, the microcontroller controls each word that is transferred to the ICAP. This is performed by reading the 16 bit words from flash memory and by packing them into 32 bit words, before sending them to the ICAP port. This option has the disadvantage of reading the flash word-by-word, and of requiring the intervention of the microcontroller for each word transfer. In the second option, the data are directly transferred from

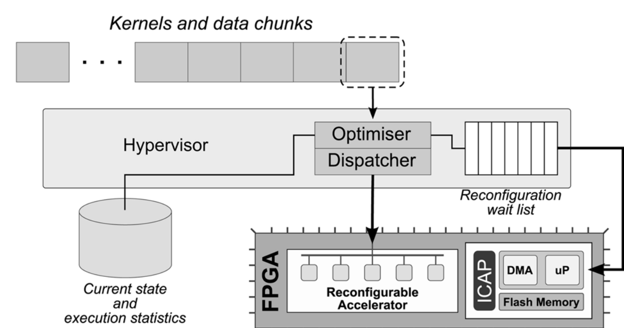


Fig. 2 *Reconfiguration procedure: the Hypervisor is responsible for dispatching the workload and for issuing reconfiguration commands to the on-chip reconfiguration engine*

© The Institution of Engineering and Technology 2014
doi: 10.1049/iet-cdt.2014.0078

the flash memory to the ICAP port. The microcontroller only has to set up a DMA descriptor and order the DMA transfer to start. Although this last option requires the presence of the DMA engine, it allows the usage of the flash burst mode, as well as a faster merge of the 16 bit words into 32 bit words, required by the ICAP. This results in much higher transfer rates and consequently faster reconfigurations. Once the transfer is completed, the reconfiguration engine controller signals the host computer, informing it that the requested reconfiguration is completed. With this approach, only one reconfiguration command can be issued at a time, since the previous reconfiguration must be concluded before a new reconfiguration command can be issued. Besides simplifying the reconfiguration procedure (avoiding the presence of reconfiguration command queues), it is worth noting that it does not significantly affect the resulting performance, since the contention to access the flash memory would prevent greater reconfiguration throughputs.

3.4 Hypervisor replacement policies

As mentioned in Section 2.1, the Hypervisor software layer at the host computer implements a set of optimisation policies targeting different application requirements and constraints. In the considered implementation, such policies provide three cumulative levels of optimisations: (i) execution-time/energy optimisation; (ii) power-ceiling dynamic constraints; and (iii) power saving with a minimum predefined and assured performance level.

The algorithm presented in Fig. 3*a* implements a runtime performance prediction routine. To decide when it is advantageous to reconfigure a given cluster, before executing the required kernel, the algorithm performs two distinct steps. Initially, it searches for a configuration allowing for higher gains in terms of performance energy or power consumption. This decision also takes into account the reconfiguration overhead. If such configuration is found and if the required time to complete all other scheduled reconfigurations is lower than executing the kernel with the current configuration, the targeted cluster is put in a waiting list for reconfiguration at the host side. It is worth noting that this algorithm can also be used for energy and power optimisations, by changing all time-base variables to energy variables.

The algorithm presented in Fig. 3*b* adds an extra level of optimisation to the previous algorithm, by introducing a

```
Input: kernel, chunk, current_arch
Global Variables: reconf_time, inprogress_reconfs
Output: new_arch or 0

  cups ← load_kernel_stats(kernel, current_arch)
  curr_time ← cups × chunk

  new_arch ← 0
  time ← curr_time
  for each arch do
    cups ← load_kernel_stats(kernel, arch)
    test ← cups × chunk

    if test × 2 + reconf_time < time × 2 then
      time ← test
      new_arch ← arch
    end if
  end for

  if (new_arch > 0) and (curr_time < time + inprogress_reconfs × reconf_time) then
    new_arch ← 0
  end if
  return new_arch
```

a

```
Input: kernel, chunk, idle_list, max_power, rec_list
Global Variables: reconf_power, static_power

  curr_power ← static_power
  for each cluster do
    if cluster.arch ≠ EMPTY then
      curr_power ← curr_power + get_power(cluster, cluster.arch)
    end if
  end for

  for each cluster ∈ idle_list do
    if curr_power > max_power then
      curr_power ← curr_power - get_power(cluster, cluster.arch)
      rec_list ← (cluster, EMPTY)
    end if
  end for

  cluster ← find_idle_or_off_cluster()
  new_arch ← time_optimisation(kernel, chunk, cluster.arch)
  if (new_arch > 0) and (curr_power + reconf_power ≤ max_power) then
    rec_list ← (cluster, new_arch)
  end if
```

b

```
Input: kernel,chunk,idle_list,min_cups,rec_list
Global Variables: reconf_time

  for each cluster do
    if cluster.arch ≠ EMPTY then
      curr_cups ← curr_cups + load_kernel_stats(cluster.kernel, cluster.arch)
    end if
  end for

  cluster ← find_idle_or_off_cluster()
  if cluster == NULL then
    new_arch ← time_optimisation(kernel, chunk, EMPTY)
  else
    new_arch ← time_optimisation(kernel, chunk, cluster.arch)
  end if

  if cluster ≠ NULL and new_arch > 0 then
    test ← curr_cups - load_kernel_stats(kernel, cluster.arch)
    cups ← load_kernel_stats(kernel, new_arch)
    test ← test + (cups × chunk + reconf_time)/chunk
    if test ≥ min_cups then
      rec_list ← (cluster, new_arch)
      curr_cups ← curr_cups - test
    end if
  end if

  for each cluster ∈ idle_list do
    cups ← load_kernel_stats(kernel, cluster.arch)
    if curr_cups - cups < min_cups then
      continue
    end if
    curr_cups ← curr_cups - cups
    rec_list ← (cluster, EMPTY)
  end for
```

c

Fig. 3 *Considered Hypervisor replacement policies*

a Execution-time optimisation
b Power-ceiling algorithm
c Minimum assured performance algorithm

© The Institution of Engineering and Technology 2014

dynamic power-ceiling constraint. This power constraint can change at runtime, depending on the dynamic requisites of the application. On the basis of the total power budget of the system at a given time, the algorithm tries to turn off clusters that are inactive until the power constraint is met. Each cluster is turned off by reconfiguring it to a blank (inactive) box, which turns off the FPGA logic in the considered cluster area. As soon as the power budget increases, an idle or turned-off cluster is searched, to be analysed with the algorithm presented in Fig. 3*a* for the current kernel chunk. Under this assumption, each cluster is only reconfigured if the power overhead for the reconfiguration procedure and for the new configured architecture does not violate the power-ceiling constraint.

The algorithm presented in Fig. 3*c* further adds a new level of optimisation, with a minimum assured performance policy. This algorithm tries to minimise the power consumption while maintaining a given minimum performance level. Initially, the algorithm presented in Fig. 3*a* is executed to check for the performance requirements of the new kernel chunk. If a reconfiguration is required for an idle cluster, the reconfiguration overhead and the future performance of the new cluster architecture are checked and it only reconfigures such cluster provided that the minimum assured performance is met. Finally, the algorithm tries to turn off idle clusters to lower the total power consumption, as long as the minimum performance is met for the current kernel.

4 Evaluation

To evaluate the performance of the proposed many-core reconfiguration framework, the complete system was prototyped in a Xilinx Virtex-7 FPGA (XC7VX485T), connected through an 8× PCIe Gen 2 link to a personal computer equipped with an Intel Core i7 3770 K, running at 3.5 GHz. The synthesis and place and route procedures were performed using Xilinx ISE 14.5. On the Intel Core i7, cycle accurate measurements were obtained by using the performance application programming interface (PAPI) library.

4.1 Evaluation benchmark

To demonstrate the performance and energy-aware capabilities of the proposed adaptive framework, a 100+ computational cores architecture was prototyped and evaluated. The considered benchmark is an application composed of four phases, each one corresponding to a linear algebra data-parallel computation kernel with distinctive processing requirements, namely: (i) Kernel 1 performs the sum of two integer vectors (1); (ii) Kernel 2 computes the inner product of two integer vectors (2); (iii) Kernel 3 performs the sum of two single-precision floating-point (FP) vectors (3); and (iv) Kernel 4 computes the inner product of two single-precision FP vectors (4)

$$\textbf{Kernel1}: \boldsymbol{v}_i^{\mathrm{I}} = \boldsymbol{a}_i^{\mathrm{I}} + \boldsymbol{b}_i^{\mathrm{I}}|_{i=1,\ldots,N} \tag{1}$$

$$\textbf{Kernel2}: \alpha^{\mathrm{I}} = \sum\nolimits_{i=[1,N]} \boldsymbol{a}_i^{\mathrm{I}} \times \boldsymbol{b}_i^{\mathrm{I}} \tag{2}$$

$$\textbf{Kernel3}: \boldsymbol{v}_i^{\mathrm{F}} = \boldsymbol{a}_i^{\mathrm{F}} + \boldsymbol{b}_i^{\mathrm{F}}|_{i=1,\ldots,N} \tag{3}$$

$$\textbf{Kernel4}: \alpha^{\mathrm{F}} = \sum\nolimits_{i=[1,N]} \boldsymbol{a}_i^{\mathrm{F}} \times \boldsymbol{b}_i^{\mathrm{F}} \tag{4}$$

Each input dataset ($\boldsymbol{a}_i^{\mathrm{I}}$, $\boldsymbol{b}_i^{\mathrm{I}}$, $\boldsymbol{a}_i^{\mathrm{F}}$ and $\boldsymbol{b}_i^{\mathrm{F}}$) is an independent and randomly generated array with about 300 million integer/FP cells. These datasets are then partitioned and dynamically assigned to the processing clusters by the Hypervisor engine during runtime. Since the main focus of the proposed work is to evaluate the system reconfiguration capabilities (and not the memory access contention), this experimental evaluation assumes that the data are locally stored in the processing cluster. As such, all cores are implemented with an integrated scratch-pad memory, to alleviate the contention in the memory hierarchy.

Since each kernel requires operations with different levels of complexity, three different PE architectures were considered providing different trade-offs in terms of hardware complexity, energy consumption, and area occupancy. As the base architecture, the 32 bit reduced instruction set computer (RISC) MB-LITE [25] soft-core was used for each of these PE, because of its portable processing structure, with an implementation compliant with the well-known MicroBlaze ISA [26]. Furthermore, it allows taking advantage of the GNU compiler collection [27] support. Moreover, the MB-LITE design requires few hardware resources and is highly configurable, being relatively easy to include custom modules.

The simplest deployed PE architecture (‘Type A’) corresponds to the basic configuration of the MB-LITE core, that is, with both the barrel shifter and multiplier deactivated, as the presence of these structures would impose significant hardware overheads and a lower operating frequency [25]. The second architecture (‘Type B’) includes the referred structures, corresponding to the full MB-LITE architecture. For the third considered architecture (‘Type C’), a full MB-LITE architecture supporting a single-precision FP unit (FPU) was considered. In this particular case, the considered four-stage pipelined FPU is composed of three Xilinx IP FP operator [28] cores. Since the MB-LITE core is compatible with the MicroBlaze ISA, the corresponding FP instruction opcodes [26] for the addition, subtraction, multiplication and compare were adopted.

To provide the Hypervisor with relevant profiling information about each architecture, five performance metrics were monitored during execution, corresponding to the counts of clock cycles (CLK), integer multiplications (MUL), FP operations (FP) and calls to software-emulated integer multiplications (SW_MUL) and FP operations (SW_FP).

4.2 Hardware resources

Having defined the PE architectures, a hardware resource analysis was performed in order to define the number of processing cores to be instantiated in each cluster topology. Table 1 presents the required hardware resources for each type of processing core. As expected, the resource overhead increases with the complexity of each architecture. Hence, the number of processing cores in each cluster topology was dimensioned in order to ensure that the total occupied area by each cluster is approximately the same. This way, each ‘Type A’ cluster contains 15 cores, each ‘Type B’ cluster contains 12 cores and each ‘Type C’ cluster contains eight cores (see Table 2).

By analysing the power consumption of each cluster topology, by considering an operating frequency of 100 MHz, it was observed a rather similar value for the three topologies (bottom of Table 2). This is explained by the fact that the three cluster topologies were properly defined in order to occupy the same amount of hardware resources.

© The Institution of Engineering and Technology 2014
doi: 10.1049/iet-cdt.2014.0078

Table 1 Experimental evaluation of each type of core, in terms of hardware resources, maximum operating frequency and power consumption

	Available resources	Processing cores		
		Type A	Type B	Type C
registers	607 200	530 (<1%)	536 (<1%)	841 (<1%)
LUTs	303 600	1132 (<1%)	1406 (<1%)	1932 (<1%)
RAMB36E1	1030	4 (<1%)	4 (<1%)	4 (<1%)
RAMB18E1	2060	3 (<1%)	3 (<1%)	3 (<1%)
DSP48E1	2800	0 (0%)	3 (<1%)	7 (<1%)
max frequency, MHz	–	211.1	114.7	113.4

The static part of the platform was also analysed in terms of the required hardware resources and power consumption. Hence, all the components responsible for the reconfiguration process, such as the MicroBlaze and the ICAP controller, as well as the communication part with the host, such as the AXI DMA and PCIe bridge, were taken into account for the total power consumption of the system. Table 3 presents the hardware resources required for the static part. It can be observed that despite the complexity of the static components, a low occupancy of about 15% was achieved in this FPGA device. Furthermore, the static design only consumes a total of 367.9 mW.

According to the obtained results, it was possible to implement a seven-cluster accelerator in the considered FPGA. In the whole, this represents a number of processing cores ranging from 56 to 105, in the implemented system.

4.3 Reconfiguration overhead

In what concerns the evaluation of the real-time adaptation of the system, it was observed the expected dependency of the reconfiguration time with the size of the partial bitstream that is loaded into the ICAP. Since the bitstreams for the three considered cluster topologies are ∼2 MBytes, a reconfiguration time of ∼10 ms is observed. A special case is worth noting in what concerns the bitstream corresponding to the blank-box configuration. Since this file is only 460 kBytes long, a smaller reconfiguration time of ∼2 ms was observed for each reconfiguration.

To obtain the dynamic power that is spent in the reconfiguration procedure of each cluster, the power that is consumed by the reconfiguration engine, when it is in its idle state, was subtracted to the power consumed by the same engine while performing a reconfiguration procedure. The obtained difference between these two measures results in an estimated reconfiguration power of about 44 mW. Despite consuming significantly less power than what it is required by the actual processing clusters, this result is also considered by the Hypervisor, when making decisions regarding reconfiguration commands and energy savings.

4.4 Performance evaluation

To further evaluate the proposed system, the following sections present its characterisation in terms of the offered adaptability. This is performed by first considering two situations without any previous knowledge of the application being executed, resulting in the definition of an optimised execution model. This model is then used to demonstrate the optimisation policies proposed in Section 3.4. The presented results are shown in terms of the attained performance and energy savings.

4.4.1 Runtime architecture adaptation and model definition: To demonstrate the adaptive capabilities offered by the implemented Hypervisor and to provide the best fitted architecture for a given kernel, two different execution scenarios were considered. Both these scenarios assume a blind execution model, where no a priori knowledge of the

Table 2 Experimental evaluation of each cluster type, in terms of hardware resources, maximum operating frequency and power consumption

	Available resources per cluster region	Processing clusters		
		Type A	Type B	Type C
number of cores	–	15	12	8
registers	48 864	7655 (16%)	6146 (13%)	6569 (13%)
LUTs	24 432	16 706 (68%)	16 614 (68%)	15 334 (63%)
RAMB36E1	60	60 (100%)	48 (80%)	32 (53%)
RAMB18E1	45	45 (100%)	36 (80%)	24 (53%)
DSP48E1	174	0 (0%)	36 (21%)	56 (32%)
max frequency, MHz	–	206.2	114.7	113.2
static power, mW	–	210.9	210.7	209.9
total power at 100 MHz, mW	–	615.2	636.6	600.7

Table 3 Experimental evaluation of the static fraction of the reconfigurable platform, in terms of hardware resources, maximum operating frequency and power consumption

	Registers	LUTs	RAMB36E1	RAMB18E1	DSP48E1
MicroBlaze	2346 (<1%)	2126 (<1%)	22 (2%)	0 (0%)	3 (<1%)
AXI4 bus	21 041 (3%)	17325 (6%)	4 (<1%)	1 (<1%)	0 (0%)
AXI4-Lite Bus	248 (<1%)	483 (<1%)	1 (<1%)	0 (0%)	0 (0%)
PCIe bridge	12 595 (2%)	17 954 (6%)	0 (0%)	0 (0%)	0 (0%)
ICAP controller	742 (<1%)	546 (<1%)	1 (<1%)	1 (<1%)	0 (0%)
DMA	2891 (<1%)	2930 (<1%)	1 (<1%)	0 (0%)	0 (0%)
BRAMS	864 (<1%)	1242 (<1%)	3 (<1%)	0 (0%)	0 (0%)
total power consumption, mW	367.9				

© The Institution of Engineering and Technology 2014

kernels is assumed to determine the first configuration of the architecture. In particular, these two scenarios only differ in the order the computing kernels are executed.

In Figs. 4 and 5, it is possible to see the Hypervisor allowing each cluster to execute its assigned chunk of a kernel with its currently assigned configuration. Then, on completion of such kernel chunk, the Hypervisor sends a reconfiguration command to that same cluster, in order to adapt its architecture to the currently executing kernel, according to the set of received values of the performance counters. The longer execution time observed in Fig. 5 is due to the fact that the first chunks of kernels 3 and 4 are initially executed in clusters of 'Type A' and 'Type B', and only then does the Hypervisor know that FP operations are needed. This means that the FP operations present in those kernels are initially executed with software libraries, resulting in an increased latency.

4.4.2 Adaptive model-based policies: After the first execution of the application in the previously described 'untrained' mode, the obtained model of the application can be used to demonstrate the other developed optimisation policies. In these scenarios, since there is a previously obtained execution model, when a kernel is to be executed, the Hypervisor can immediately trigger the reconfiguration process to adapt the assigned cluster to the best fitted architecture for that kernel.

The execution time policy described in Fig. 3*a* allows the system to dynamically select the set of clusters that provide the best performance for each kernel under execution. The experimental results for this policy are presented in Fig. 6 which conclude that the system was able to adapt to the best possible configuration, while also achieving a well-balanced data chunk distribution to the several processing clusters.

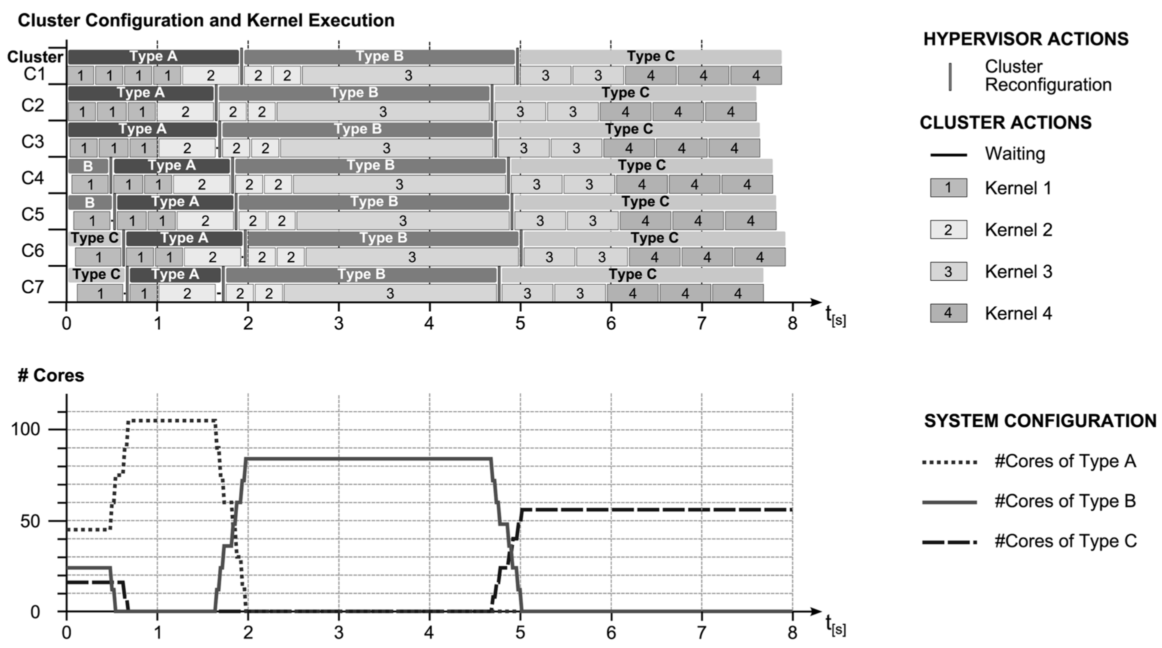


Fig. 4 *Real-time adaptation of the processing architecture, without any a priori knowledge of the computing kernels (Kernel order: 1, 2, 3, 4)*

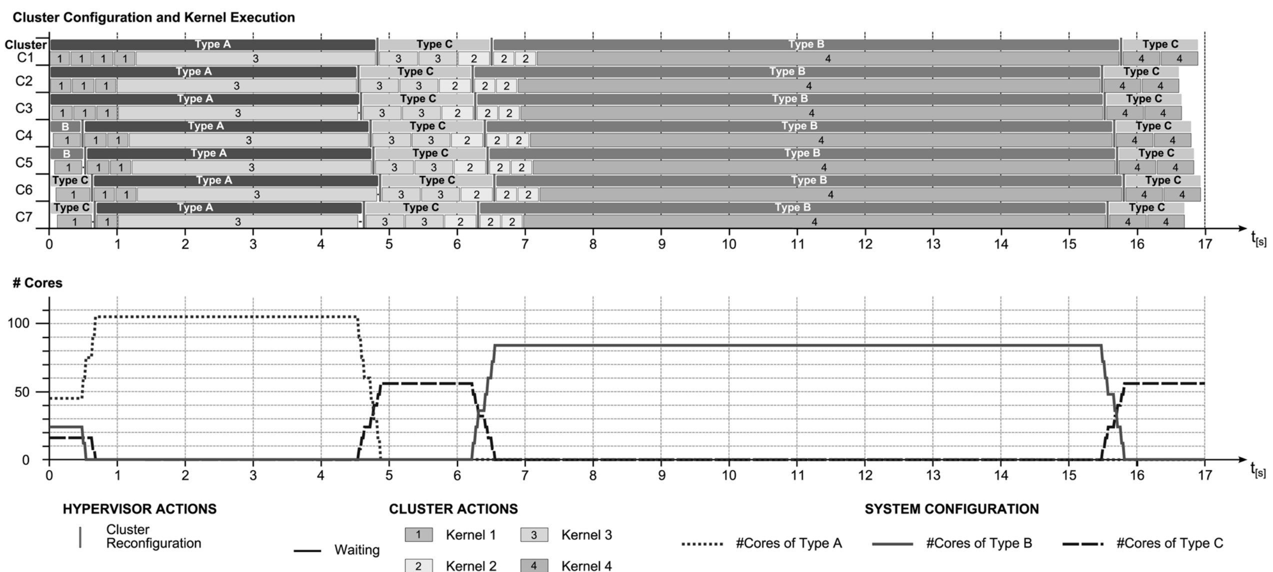


Fig. 5 *Real-time adaptation of the processing architecture, without any a priori knowledge of the computing kernels (Kernel order: 1, 3, 2, 4)*

© The Institution of Engineering and Technology 2014
doi: 10.1049/iet-cdt.2014.0078

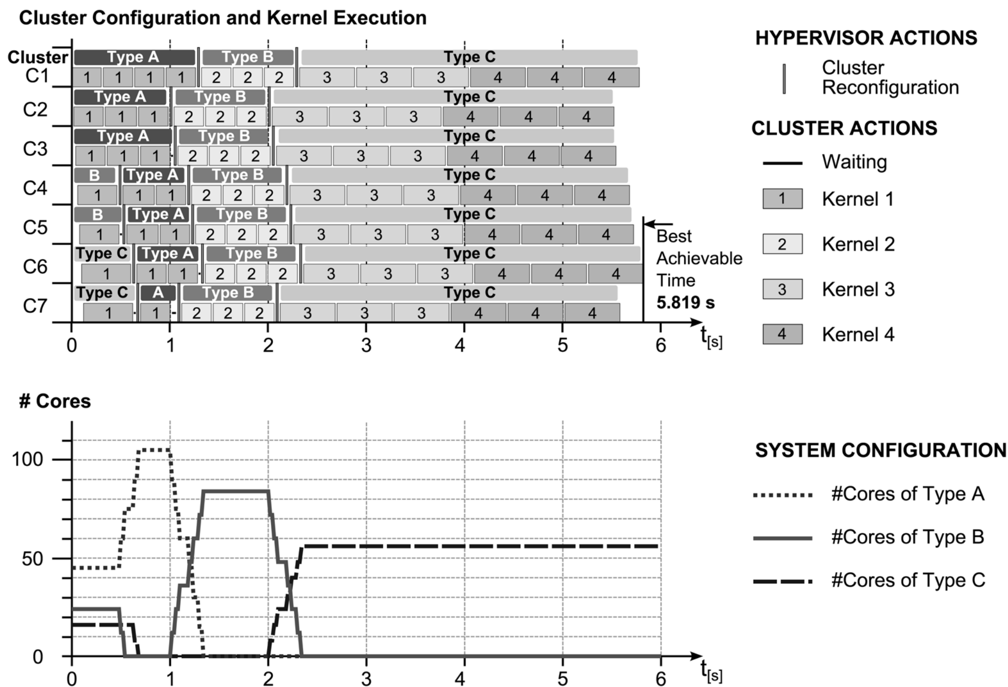


Fig. 6 *System real-time adaptation, according to the minimum execution-time optimisation policy*

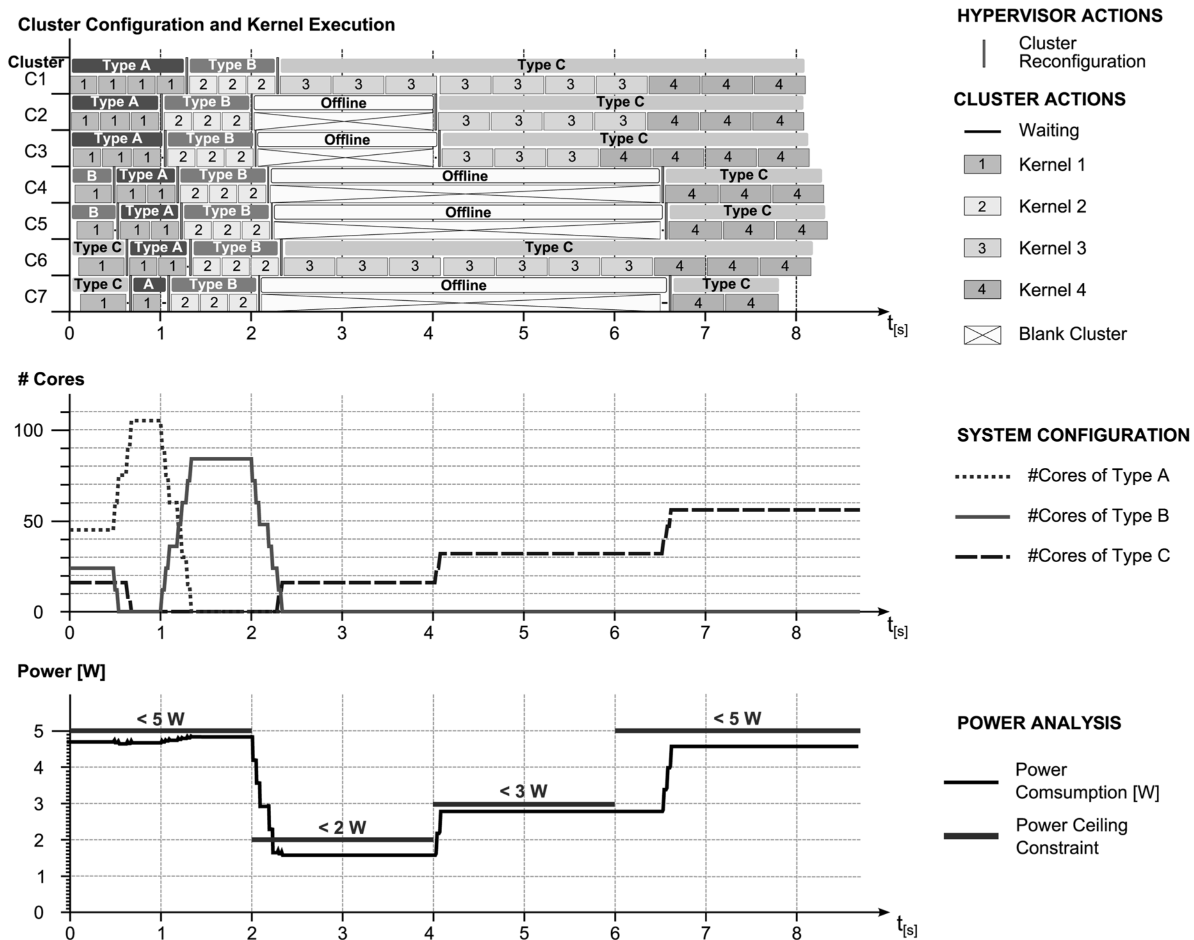


Fig. 7 *System real-time adaptation, according to the established power-ceiling constraint policy*

The second considered optimisation policy considers the maximisation of the system performance, while establishing a given power-ceiling (see Fig. 3*b*). To ensure a more realistic test, this power-ceiling was also varied in runtime. In Fig. 7, it is possible to observe idle clusters being replaced by empty blank-boxes when the power-ceiling

© The Institution of Engineering and Technology 2014

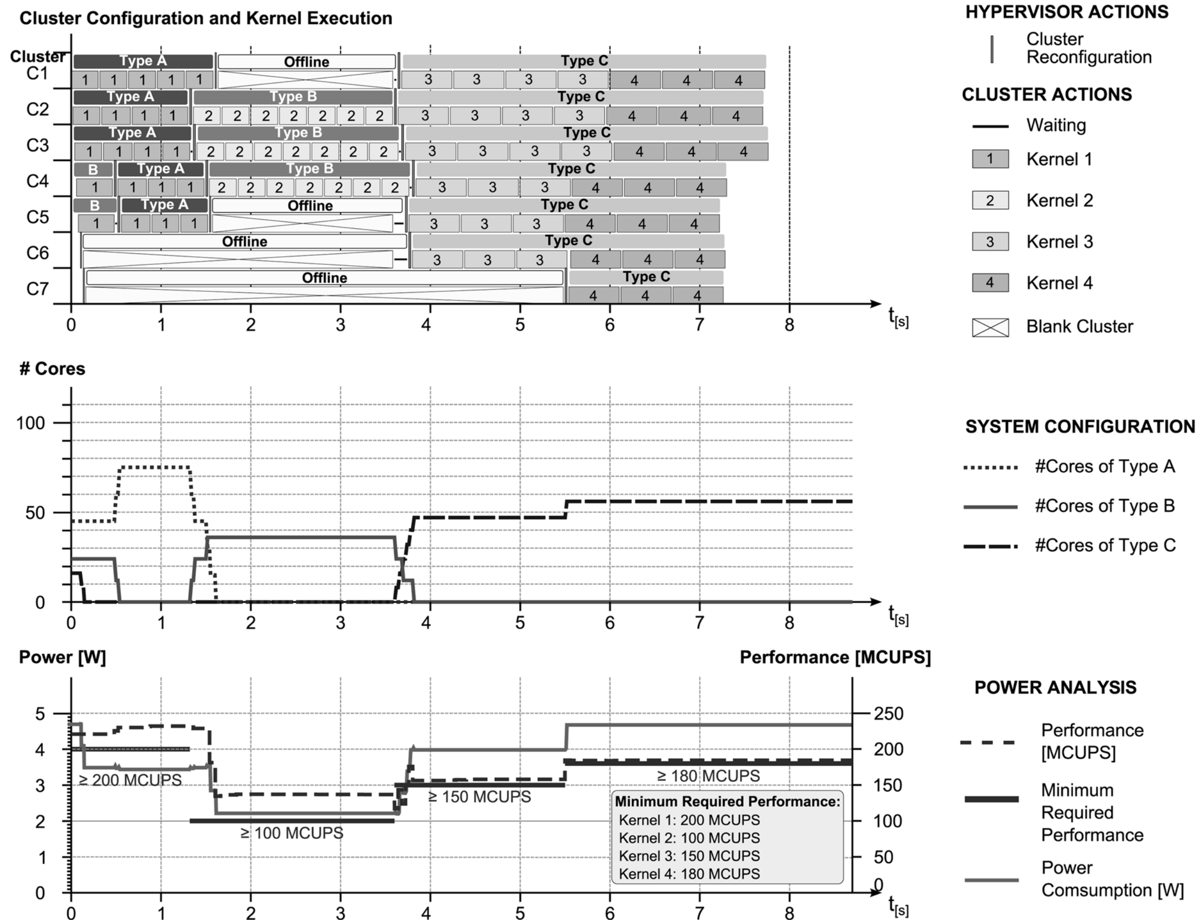


Fig. 8 *System real-time adaptation, according to the minimum assured performance policy*

decreases, in order to meet this constraint. On the other hand, as soon as the allowed power consumption level increases, the system reactivates these turned-off clusters, in order to maximise the accelerator throughput.

The last proposed optimisation policy, previously described in Fig. 3*c*, considers the minimisation of the power consumption while assuring a minimum performance level. To show the adaptivity of the proposed system, it is further assumed that the application under execution establishes a different minimum throughput for each kernel, as shown in Fig. 8. As it can be observed, the system is able to adapt the clusters in real-time, not only to ensure the required performance level, but also to minimise the power consumption, by disabling inactive clusters.

4.4.3 Speedup and energy reduction: To evaluate the performance gains and energy savings resulting from the proposed adaptive system, the dynamic execution policy presented in Fig. 6 was compared with four different static configurations (i.e. without reconfiguration), each one composed by seven independent clusters of PE: (i) a system with seven 'Type A' clusters; (ii) a system with seven 'Type B' clusters; (iii) a system with seven 'Type C' clusters; and (iv) a heterogeneous mix composed of two 'Type A' clusters, two 'Type B' clusters and three 'Type C' clusters.

Table 4 presents the obtained results in terms of execution-time and energy consumption for the considered setups. Despite containing 105 cores, it can be observed that the system with only 'Type A' clusters represents the worst case, both in terms of performance and energy. This is explained by the fact that 'Type A' PE must perform the multiplication operations of Kernel 2 through a combination of logic shifts and additions, and the floating-point operations of Kernels 3 and 4 through calls to software libraries. Naturally, this represents a large energy overhead, which results in a total consumption of 240 J. The best homogeneous static configuration is obtained by using only 'Type C' clusters. Even though only 56 cores can be implemented in this case, it performs about 4× faster than the worst-case configuration. The best static configuration was achieved by using the considered heterogeneous

Table 4 Execution-time and energy results

	Execution time, s	Energy consumption, J	Dynamic system speedup	Dynamic system energy gain
dynamic system	5.819	23.28	–	–
static 7 × Type A clusters	55.715	239.93	9.575	10.31
static 7 × Type B clusters	29.784	132.72	5.118	5.70
static 7 × Type C clusters	11.997	50.44	2.062	2.17
static heterogeneous mix	18.378	79.13	3.158	3.40

© The Institution of Engineering and Technology 2014

doi: 10.1049/iet-cdt.2014.0078

configuration (2× ‘Type A’ + 2× ‘Type B’ + 3× ‘Type C’), which provides a trade-off between complexity and execution-time/energy consumption.

Finally, it can be observed that the offered adaptive capabilities allow the dynamic system to combine all the advantages of the above described configurations. By adapting, at runtime, to the requirements of the different kernels, it is assured that the system always provides the best optimised configuration for each application phase, by trading core complexity with the total number of cores. Thus, it is possible to achieve execution speedups ranging from 2.1× , when compared with the best static case, to 9.5× , when compared with the worst static case, while consuming from 2.2× to 10.3× less energy.

5 Conclusions

A new morphable heterogeneous computing platform, integrating hundreds of processing cores and offering runtime adaptation capabilities to meet the performance and energy constraints imposed by the application under execution is proposed in this paper. The adaptive nature of this platform is achieved by monitoring, in real-time, the performance of the computational cores while they execute the application kernels, and by determining the processing architectures and topologies that maximise the performance and/or energy efficiency. To perform this decision, a Hypervisor software module is also proposed, which is not only responsible for scheduling the computing kernels to the available processing cores, but it is also able to trigger the reconfiguration of the currently instantiated cores, by issuing appropriate commands to an on-chip reconfiguration engine. This latter module performs the actual adaptation, by exploiting the existing partial dynamical reconfiguration mechanisms of modern FPGA devices.

To perform the adaptation of this hundred-core heterogeneous platform, different algorithms (policies) were considered, corresponding to typical optimisation goals, namely: minimisation of the execution time; maximisation of the processing performance for a given power-ceiling; and minimisation of the power consumption, while guaranteeing a minimum QoS (performance level). To evaluate the proposed system and the corresponding policies, a set of computation kernels from the algebraic domain were used. The obtained experimental results allow concluding that the proposed policies provide a significant reduction of both the execution-time and energy consumption when compared with static homogeneous or non-homogeneous implementations with a fixed number of cores. The proposed reconfigurable system achieves performance gains between 2× and 9.5× , whereas the energy consumption was reduced between 2× and 10× .

Accordingly, the proposed morphable heterogeneous structure has shown to be a highly viable and efficient approach to provide an adaptable architecture, being able to morph the computing cores according to the instantaneous system restrictions, resulting not only in improved performances but, more importantly, in an energy-aware computing platform.

6 Acknowledgments

This work was partially supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects HELIX (ref. PTDC/EEA-ELC/113999/2009), Threads (ref. PTDC/EEA-ELC/117329/2010), P2HCS (ref. PTDC/EEI-ELC/3152/2012) and project PEstOE/EEI/LA0021/2013.

References

1 Srivastava, M., Chandrakasan, A., Brodersen, R.: ‘Predictive system shutdown and other architectural techniques for energy efficient programmable computation’, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1996, **4**, (1), pp. 42–55

2 Kumar, R., Martinez, A., Gonzalez, A.: ‘Dynamic selective devectorization for efficient power gating of SIMD units in a HW/SW co-designed environment’. 25th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD), October 2013, pp. 81–88

3 Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D., Dwarkadas, S., Scott, M.: ‘Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling’. Eighth Int. Symp. on High-Performance Computer Architecture (ISCA), February 2002, pp. 29–40

4 Muthukaruppan, T.S., Pricopi, M., Venkataramani, V., Mitra, T., Vishin, S.: ‘Hierarchical power management for asymmetric multi-core in dark silicon era’. Proc. of the 50th Annual Design Automation Conf., 2013, pp. 174:1–174:9

5 Zhu, Y., Reddi, V.: ‘High-performance and energy-efficient mobile web browsing on big/little systems’. 2013 IEEE 19th Int. Symp. on High Performance Computer Architecture (HPCA), February 2013, pp. 13–24

6 Koufaty, D., Reddy, D., Hahn, S.: ‘Bias scheduling in heterogeneous multi-core architectures’. Proc. of the Fifth European Conf. on Computer Systems, ser. EuroSys ‘10, 2010, pp. 125–138

7 Cong, J., Yuan, B.: ‘Energy-efficient scheduling on heterogeneous multi-core architectures’. Proc. of the 2012 ACM/IEEE Int. Symp. on Low Power Electronics and Design, 2012, pp. 345–350

8 Chitlur, N., Srinivasa, G., Hahn, S., *et al*.: ‘Quickia: exploring heterogeneous architectures on real prototypes’. IEEE 18th Int. Symp. on High Performance Computer Architecture (HPCA), February 2012, pp. 1–8

9 Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., Emer, J.: ‘Scheduling heterogeneous multi-cores through performance impact estimation (PIE)’. Proc. of the 39th Annual Int. Symp. on Computer Architecture (ISCA’2012), 2012, pp. 213–224

10 Akram, S., Papakonstantinou, A., Kumar, R., Chen, D.: ‘A workload-adaptive and reconfigurable bus architecture for multicore processors’, *Int. J. Reconfigurable Comput.*, 2010, **2010**, p. 2

11 Modarressi, M., Tavakkol, A., Sarbazi-Azad, H.: ‘Application-aware topology reconfiguration for on-chip networks’, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2011, **19**, (11), pp. 2010–2022

12 Pal, R., Paul, K., Prasad, S.: ‘Rekonf: a reconfigurable adaptive manycore architecture’. 2012 IEEE Tenth Int. Symp. on Parallel and Distributed Processing with Applications (ISPA), 2012, pp. 182–191

13 Wang, W., Mishra, P., Ranka, S.: ‘Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems’. 48th ACM/EDAC/IEEE Design Automation Conf. (DAC), June 2011, pp. 948–953

14 Rodrigues, R., Annamalai, A., Koren, I., Kundu, S., Khan, O.: ‘Performance per watt benefits of dynamic core morphing in asymmetric multicores’. 2011 Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), October 2011, pp. 121–130

15 Rodrigues, R., Annamalai, A., Koren, I., Kundu, S.: ‘Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing’, *ACM Trans. Des. Autom. Electron. Syst.*, 2013, **18**, (1), pp. 5:1–5:23

16 Chaves, R., Kuzmanov, G., Sousa, L.: ‘On-the-fly attestation of reconfigurable hardware’. Int. Conf. on Field Programmable Logic and Applications (FPL). IEEE, 2008, pp. 71–76

17 Watkins, M.A., Albonesi, D.H.: ‘Remap: a reconfigurable heterogeneous multicore architecture’. 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO). IEEE, 2010, pp. 497–508

18 Garcia, P., Compton, K.: ‘Kernel sharing on reconfigurable multiprocessor systems’. Int. Conf. on ICECE Technology (FPT). IEEE, 2008, pp. 225–232

19 Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., DeHon, A.: ‘Stream computations organized for reconfigurable execution (SCORE)’. Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing, 2000, pp. 605–614

20 Chen, Z., Pittman, R.N., Forin, A.: ‘Combining multicore and reconfigurable instruction set extensions’. Proc. of the 18th Annual ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA), 2010, pp. 33–36

© The Institution of Engineering and Technology 2014

21 Lorenz, M.G., Mengibar, L., Valderas, M.G., Entrena, L.: ‘Power consumption reduction through dynamic reconfiguration’. Field Programmable Logic and Application, 2004, pp. 751–760
22 AMBA® 4 AXI4-Stream Protocol, v1.0, ARM, Ltd., March 2010. Available at http://www.infocenter.arm.com
23 Shin, E.S., Mooney, V.J.III, Riley, G.F.: ‘Round-robin arbiter design and generation’. Proc. of the 15th Int. Symp. on System Synthesis, 2002, pp. 243–248
24 LogiCORE IP AXI DMA v6.03a, Xilinx Inc., December 2012. Available at http://www.xilinx.com/support/documentation/ip_documentation/
25 Kranenburg, T., van Leuken, R.: ‘MB-LITE: a robust, light-weight soft-core implementation of the MicroBlaze architecture’. Design, Automation and Test in Europe Conf. and Exhibition (DATE), March 2010, pp. 997–1000
26 MicroBlaze Processor Reference Guide, v14.3, Xilinx Inc., October 2012
27 GCC, the GNU Compiler Collection, GNU Project, October 2013. Available at http://www.gcc.gnu.org/
28 LogiCORE IP Floating-Point Operator v5.0, Xilinx Inc., March 2011. Available at http://www.xilinx.com/support/documentation/ip_documentation/

© The Institution of Engineering and Technology 2014

doi: 10.1049/iet-cdt.2014.0078