A Distributed Processing Network Architecture for Reconfigurable Computing

Fernando Martinez Vallina, Erdal Oruklu, and Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, Illinois 60616

Abstract

This paper introduces a set of rules and guidelines for the implementation of a Distributed Processing Network (DPN) as the basis for a dynamic reconfigurable architecture targeted at improving the performance of microprocessor (μP) based systems in computationally intensive application domains. In order to provide the computation gains needed to improve upon the performance of the μP, the DPN architecture offers: 1) A low reconfiguration overhead, 2) A simple control interface, 3) Dynamic resource allocation, 4) Concurrent execution with dynamic reconfiguration, 5) Lower power dissipation than a μP executing the same computation kernel and, 6) Scalability to tackle tasks of varying resource requirements. DPN is currently targeted at real-time computationally intensive application domains such as compression, and signal transformations.

1. Introduction

The use of FPGAs has become popular as a way of integrating programmable cores into System on a Chip (SoC) designs [1]. The main advantage of this consists in using configurable logical blocks such as the FPGA, which allows for spatial mapping of algorithms instead of the temporal mapping used in most microprocessor (μP) based systems. The application of spatial mapping allows algorithms with a high degree of parallelism and/or frequent instances of simultaneous memory access to demonstrate improvements in throughput and power consumption in relation to the performance of the same algorithm on a conventional μP [2].

The improvements seen as a result of spatial mapping are a direct result of the approach taken to implement a specific algorithm. Another way of analyzing spatial vs. temporal mapping is to think of algorithmic execution in terms of hardware vs. software [3]. The differentiating factors between these two design domains are that a hardware platform offers:

- A customized solution without the need of extra circuitry for command interpretation.
- Fast execution due to spatial mapping.

On the other hand, solving a problem in software takes advantage of the flexibility found in μP systems that can tackle many different problems with relative ease. The main characteristics of this approach are:

- Flexibility since switching task only implies switching the instruction stream to the μP.
- A relatively slow solution since there is little or no hardware acceleration and there are temporal boundaries associated with instruction fetching and decoding.
- An inefficient solution from a resource allocation viewpoint. The computational task is not tightly matched to the capabilities of the hardware.

FPGAs have become a popular way for bridging these two solution domains. This platform allows designers some degree of programmability while at the same time allowing for the spatially mapped implementation of an algorithm. Although a spatial solution can be implemented with an FPGA, there are practical limits to the application of this type of solution due to the scalability of the FPGA and the resource requirements of different algorithms [4]. In the case of an algorithm spatially overflowing the FPGA, the only available solutions are to chain multiple FPGA chips together or to time sequence the execution of an algorithm. The problem with chaining several FPGA chips together comes from severe performance degradation due to the lack of external bandwidth. The other alternative, which is to realize the algorithm sequentially, requires the dynamic switching in and out of configuration contexts for reprogramming the FPGA [5]. The switching of configuration contexts adds temporal overhead to the overall execution and decreases the performance gains of spatial mapping.

The reconfigurable DPN architecture presented in this paper has both the performance characteristics of an ASIC as well as the simple configuration characteristics of an FPGA. The DPN architecture can be both implemented on an FPGA as well as in an ASIC design retaining all of its characteristics in both design domains.

The only characteristic of the DPN which is limited by the design domain is the overall latency of the processing network. This characteristic is directly dependant on the physical bandwidth constraints found within each design domain.

The hierarchy and underlying architecture of the DPN is described in the following sections. These sections will also introduce the concept of an application domain as it is a key component in determining the overall function and performance of the DPN.

2. DPN Overview

DPN is a hierarchical distributed processing network architecture, which approaches the problem of reconfigurable computing in a manner similar to both the GARP[6] and CHIMAERA[7] reconfigurable processors. Both of these processors are similar in that the reconfigurable component is in both cases coupled to a MIPS style processor. In both architectures, the reconfigurable component is used for μP acceleration, but can not be used as a standalone unit. The main difference between these two processors is the level of operand granularity. GARP employs coarse grain optimizations [6]. In contrast, the CHIMAERA is targeted at fine grain optimizations [7].

Coarse grain and fine grain architectures are both complementary approaches, which differ in the level of possible optimization. A coarse grain optimization can also be considered a loop-level optimization [7]. This means that a processor based on this approach is highly optimized (e.g. pipelined) for the execution of entire loops of code and depends on the ability of a compiler to determine the best way to parallelize a loop.

The complimentary approach of fine grain optimization works at the instruction level. The basis for this approach is to reduce the overall instruction count with the use of computational units which can implement the functions of several instructions. This allows the combination of data dependent instructions into a single instruction, which takes advantage of available hardware resources to accelerate program execution.

The DPN architecture provides both fine and coarse grain optimization at different stages of the architecture implementation. The fine grain optimization seen in CHIMAERA is present in the DPN during the selection of the computational units to be employed. Also, DPN possesses coarse grain optimization similar to GARP. This level of optimization is used to map algorithms within a target application domain to a specific implementation of the DPN. At this stage the compiler used for algorithm mapping can take advantage of the hierarchical nature of the DPN to carry out loop unrolling and pipelining.

The level of optimization possible in both the fine and coarse grain domains depends on the target application domain, which determines the primary function and minimum performance characteristics of a discrete DPN implementation. The application domain is defined as a set of algorithms, which are used in the same computational field. Examples of computational fields include compression, and signal transformation. Also, the definition of an application domain can be expanded to include any set of algorithms which have similar computational resource requirements. This means that even within a computational field, the application domain might only encompass a few of the algorithms if there is a large degree of variation in the computational requirements across the field. Figure 1 shows a sample of different application domains that can be accelerated by the use of the DPN architecture.

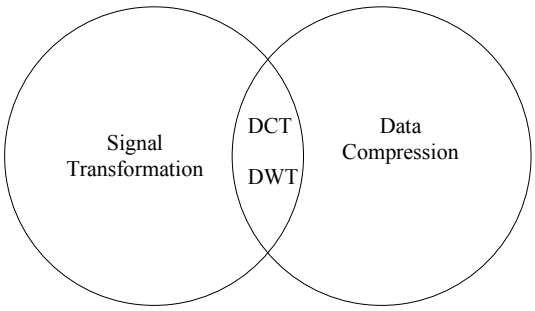

Figure 1. Sample application domains

As shown in Figure 1, different application domains have areas of overlap, which are a result of the sharing of algorithms across computational fields. For example, this figure shows two shared algorithms between the data compression and signal transformation domains. Although more than one algorithm can be shared across domains, this example focuses on showing the application domain overlap through the use of the Discrete Cosine Transform (DCT)[8] and the Discrete Wavelet Transform (DWT)[9]. DCT and DWT are orthogonal transforms which are utilized in data compression algorithms such as JPEG and JPEG2000. Furthermore, application domain overlap can occur as a result of similar computational resource requirements between algorithms of different application domains.

For example, domain overlaps like the one shown in Figure 1 allow a DPN targeted at data compression domain to execute algorithms from the signal transformation domain. Hence, execution of signal transformation algorithms provides a suboptimal solution in terms of resource allocation and computational latency. Although the optimal solution will only be generated by a DPN from the target application domain, the relationships resulting from overlaps extend the applicability of a DPN beyond its target application domain. A DPN executing an application outside of its intended domain will require

more reconfiguration than a DPN in the target domain of the algorithm.

The hierarchical nature of the DPN is what allows the same architecture to be employed across different application domains without modifications to the core of the processing network. A top level block diagram of the DPN architecture is shown in Figure 2.

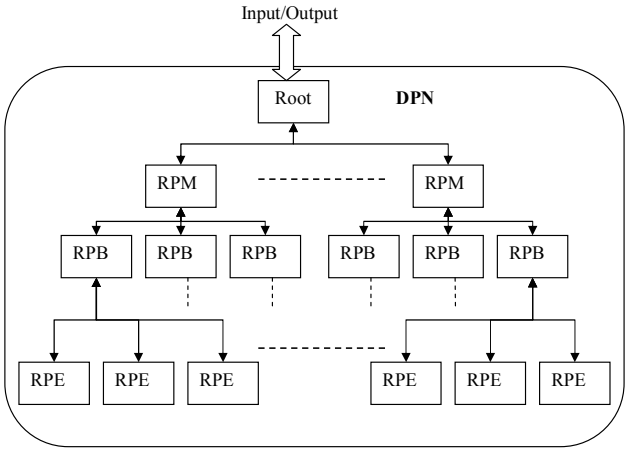

Figure 2. DPN hierarchical overview

As shown in Figure 2, DPN is a hierarchical network in which all computational work is done at the edge of the network in the Reconfigurable Processing Element (RPE). The hierarchy of the DPN consists of 4 main levels called the Root, the Reconfigurable Processing Module (RPM), the Reconfigurable Processing Block (RPB), and the Reconfigurable Processing Element (RPE). The main responsibility of the Root and the RPMs is to direct the flow of data to and from the RPBs and to provide facilities such as local memory to speed up computation. There is only one Root in the network, which is responsible for communication to devices external to the DPN and for the configuration of all the nodes in the network. Also, the Root is one of the elements which separates the DPN from the GARP and CHIMAERA processors. In both of these processors, the reconfigurable components are subject to the control of a MIPS style processor and can not be used independently. In contrast, the Root of the DPN allows the DPN to act as either a coprocessor or as an independent signal processing engine. This characteristic makes the DPN adaptable towards different SoC systems.

The targeted application domain and the design constraints of the intended implementation platform will determine the size of the DPN. In most cases, the processing network will be composed of several RPMs. RPMs can be thought of as processing Macro Cells, which organize the computational power of the network. Each RPM is then composed of three RPBs which are directly connected to the RPEs. The total computational power of an instance of a DPN is dependent on the number of RPMs in the network.

The hierarchical network approach of the DPN handles the problem of reconfigurable computing in a different way from an array based approach [10][11]. An example of this approach can be seen in Figure 3.

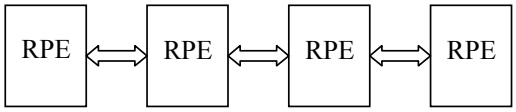

Figure 3. Array reconfigurable architecture

Figure 3 shows how the array approach tackles the problem of reconfigurable computing with a regular pattern for computation and simple interconnection between processing units. These characteristics make the array architecture simple to implement in both the ASIC and FPGA design domains. The main drawback of this type of architecture has to do with communication between processing elements. The rigid bus structure of the array makes it difficult to move data from one processing element to any other element in the architecture without having to resort to complex bus arbitration schemes. The difficulties in moving data within the array as a result of the bus structure leads to two cases. The first case is the underutilization of the array, because it can not be split for concurrent execution of different tasks. The second case, which is related to the first case, is the need for frequent reconfiguration to carry out portions of an algorithm. The combination of frequent reconfiguration and resource underutilization decreases overall performance and creates a loss of concurrency when attempting simultaneous execution.

In contrast the DPN provides a completely flexible interconnect framework to move data from one processing element to any other element in the network. This interconnect framework allows the DPN to maximize resource utilization and to minimize the instances of runtime reconfiguration. The network topology for interconnection provides the DPN with a high degree of scaling and flexibility. This allows the DPN to have better performance and higher resource utilization rates than those of reconfigurable implementations similar to Figure 3.

Although optimal performance of a DPN based system is only possible within a target application domain, the DPN architecture does not predetermine an application domain or set constraints on the size of the processing network. These characteristics combined with the flexibility of a network topology allow the DPN architecture to easily adapt to different application domains and provide an optimal solution to the computational needs of a desired algorithm.

3. DPN Architecture

This section provides a detailed explanation of the major architectural elements of the DPN. Since the most important component of the DPN is the RPE, the discussion of the different elements will start at the RPE level and build up to the Root of the DPN.

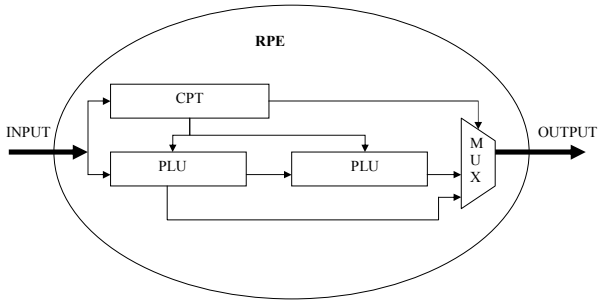

Figure 4. RPE architecture

Figure 4 shows the architectural overview of the RPE elements. The RPE is functionally similar to the Reconfigurable Functional Unit (RFU) found in the CHIMAERA processor [8]. Like the RFU, the RPE is the main computational element of the reconfigurable architecture. Also, it is the RPE which allows designers to implement fine grain optimizations on the DPN. The basic RPE consists of a Configuration Profile Table (CPT) and 2 Programmable Logic Units (PLUs). The purpose of having a local CPT inside each RPE is to reduce the number of instructions needed to configure a single RPE. This reduction in configuration instruction count is magnified throughout the network, and gives the DPN instruction level optimization.

The actual computational work of the RPE is carried out in the 2 PLUs. The use of 2 PLUs per RPE instead of a single PLU helps reduce the network overhead of passing data between RPEs. The number of PLUs per RPE is limited to only 2, due to the existing trade off between RPE computational power and RPE control abstraction. Each PLU increases both the computational power and the control abstraction of the RPE. The increase in abstraction at the RPE level can have both positive and negative effects for the overall performance of the DPN. The positive effect of increasing abstraction is that the RPE becomes computationally more powerful. At the same time increasing abstraction can lead to under utilization of available resources, unnecessary reconfiguration, and deteriorate overall performance of the DPN. The number of PLUs in the RPE is fixed at 2, because it increases RPE computational power but at the same time minimizes the negative effect of increasing abstraction.

The logical and arithmetic functions that can be implemented by an RPE determine the application domain for which it is best suited for. The RPEs in the DPN are the determining factor when differentiating between DPN implementations by application domain. The rest of the DPN architecture is not affected by the application domain. Therefore, switching application domains implies changing the type of RPE used in the network.

Although the computational behavior of the RPE is determined by the application domain, there is only one type of interface between the RPE and the RPB. This common interface allows for the remaining architectural elements of the DPN to be isolated from the application domain, which simplifies the task of customizing the DPN for a target application domain.

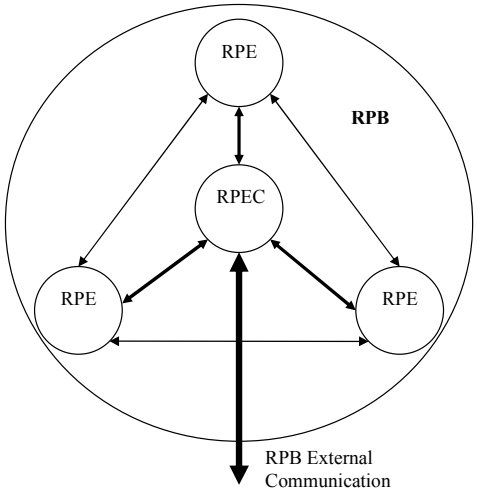

Figure 5. Reconfigurable processing block (RPB)

Figure 5 shows the DPN architecture at the RPB level. At this level, the RPE units are arranged in groups of 3 to make up a single RPB block. This arrangement is chosen to reduce the total number of buses needed for communication within the RPB, but at the same time make the RPB computationally powerful enough for one full step of an algorithm in a target application domain. The number of buses, NB, needed for data transfer in the DPN at all levels starting at the RPB level given by the following equation.

$$NB = 1 + \sum_{j=1}^{n-1} j$$

In this equation, n refers to the number of elements at the current level. For example, at the RPB level (see Fig. 5), there are 4 elements for a total of 7 communication buses. At each level of the DPN hierarchy, the number of elements refers to the largest distinct architectural elements found at that level. In the case of the RPB level, the four elements consist of the 3 RPEs and the Reconfigurable Processing Element Controller (RPEC).

At the RPB level, the interconnection network requires a direct bidirectional connection between RPEs and a direct

bidirectional connection between each RPE and the RPEC. It is the job of the RPEC to distribute configuration data to each RPE and to arbitrate the flow of data in and out of the RPB.

The interconnection among RPEs is dynamic and depends on the configuration profile loaded into each RPE (see Fig. 4). Besides configuring the behavior of the programmable logic blocks within the RPE, the RPEC also determines the cooperative behavior of the RPEs. This means the each RPE in the RPB can carry out joint or independent calculations with the other RPEs in the block depending on the algorithm being executed.

Although the interconnection between RPEs at the RPB level is completely flexible, the behavior of these buses becomes fixed after the configuration profiles are loaded. This reduces the communication overhead within the RPB, because there are no addressing issues involved with passing data from one RPE to the other. Data packet addressing within the DPN is only necessary under two circumstances:

- Data is moved between hierarchical levels.
- Data is moved between DPN elements at the same hierarchical level. This only applies to DPN elements at the RPB level and higher

By eliminating the need for addressing at the computational layer of the DPN, the latency associated with addressing and routing of data packets can be avoided at this level of the hierarchy. At the same time, the flexibility associated with a data packet network is not lost in the levels above the computational layer. This allows for the concurrent execution of different algorithms within the DPN and for a low reconfiguration overhead.

The hierarchical level on top of the RPB is the RPM level (see Fig. 2). At this level, the RPB blocks are arranged using the same interconnection framework as lower levels with the addition of a local shared memory (MEM). Figure 6 shows the structural overview of the DPN at the RPM level.

Figure 6 also shows how the same interconnect structure found at the RPB level is repeated with only a few modifications. The reason for keeping the same interconnection setup is to introduce a degree of regularity into the network, which aids in the implementation of the DPN on either the ASIC or FPGA platforms. The main addition at this level is the presence of a shared memory (MEM), which is accessible by all the RPB blocks within the RPM. The purpose of this memory is to serve as a temporary storage and scratch pad for all computations carried out in the RPM. Also, this memory acts as a cache for the RPM, which decreases the time penalty associated with fetching data from main memory, by decreasing the number of main memory calls per algorithm. The MEM has the same function and behavior as an integrated data and instruction cache. The presence of these caches distributed throughout the DPN allows this architecture to provide dynamic resource allocation and concurrent execution of multiple tasks.

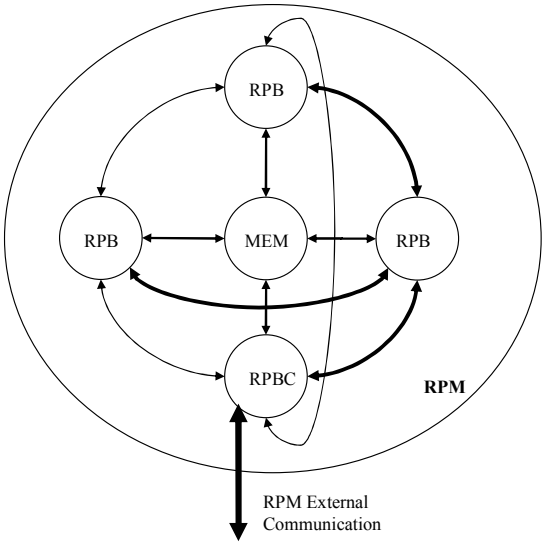

Figure 6. Reconfigurable processing module (RPM)

As in the RPB level (see Fig. 5), the RPM also has a central controller to direct the flow of data within the RPM and to interface to elements external to the RPM. The central controller of the RPM is the Reconfigurable Processing Block Controller (RPBC). Communication with external elements involves sending and receiving data from another RPM or from the Root of the DPN.

The RPBC builds upon the functions of the RPEC and also has control over the MEM component of the RPM. Although all the RPBs within an RPM can have simultaneous access to the MEM, the RPBC can block this access during reconfiguration or when a computational task is completed. The RPBC is also responsible for the beginning and end of algorithm execution based on commands from the DPN Root.

The highest level of the DPN hierarchy is the Root, which is directly above the RPM level (see Fig. 2). The main purpose of the Root is to load and control the execution of algorithms within the DPN. This process is done through the loading of DPN compiled code from a main memory and the translation of these codes into the DPN automatic mapping language as the resources become available to begin execution. The DPN compile code and the automatic mapping language are the same in that they both describe an algorithm for its exact mapping into the computational fabric of the DPN. The main difference between these two languages is in their generation time and the level of detail provided by each set of codes. The compile DPN code provides an overall mapping of an algorithm into hardware by stating

resource requirements and interconnection between these resources. This code does not include any conditions on execution time nor on where the algorithm will be physically mapped into the DPN. Automatic mapping language codes generated by the Root are only created when the hardware resources needed by all or part of an algorithm become available. The automatic mapping language provides the actual mapping of an algorithm based on resource requirements and currently available resources. Also, it specifically states the interconnection of all resources as well as their behavior. The inner structure of the Root can be modeled by a microcontroller and it is capable of controlling n RPMs.

As previously mentioned, the size of the DPN is determined by its target application domain. The DPN architecture does not place a bound on the maximum size of a DPN implementation; nevertheless, there is a lower bound for the size of the DPN. The minimum size for a DPN implementation is defined as a single RPM. On the other hand, the maximum size is limited by the physical constraints of the intended implementation platform. This variation in size of the DPN, gives this architecture the ability to tightly match the computational requirements of an application domain with the available computational resources of the DPN.

4. Conclusion

This paper presents the DPN, which is an efficient architecture for solving the problem of reconfigurable computing. The low power dissipation, scalability, and dynamic resource allocation allow this architecture to handle a wide range of signal processing problems. By combining these characteristics with a low reconfiguration overhead and concurrent execution, the DPN has the potential of becoming a real-time customizable signal processing engine. Furthermore, the DPN architecture is independent of implementation platform. The decision of implementing the DPN on an FPGA or an ASIC is left to the designer. This decision depends on the desired cost and performance characteristics for a given application. The entire topology of the DPN is self-contained and can be seamlessly switched between the FPGA and ASIC domains.

5. References

[1] W.B. Andrew, G. Carl, R.K. Charath, J.F. Hoff, R. Modo, H. Nguyen, W. Smith, D. Rhein, J. Schulingkamp, C.W. Spivak, J.P. Steward, and A. Subramanian, "A Field Programmable System Chip with Combines FPGA and ASIC Circuitry," Proceedings of the IEEE 99 CICC, pp. 183-186, May. 1999.

[2] A. Dehon, "Trends Towards Spatial Computing Architectures," ISSCC Digest of Technical Papers, 21.2, Feb. 1999.

[3] A. DeHon and J. Wawryznek, "Reconfigurable Computing: What, Why, and Implications for Design Automation," ACM/IEEE DAC 99, June 1999.

[4] K. Furuta, T. Fujii, M. Motomura, K. Wakabayashi, and M. Yamashina, "Spatial-Temporal Mapping of Real Applications on a Dynamically Reconfigurable Logic Engine (DRLE) LSI," Proceedings of the IEEE 00 CICC, pp. 151-154, May. 2000.

[5] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," Symposium on FCCM, pp. 22-28, Apr. 1997.

[6] J. Hauser and J. Wawryznek, "GARP: A MIPS processor with a reconfigurable processor," Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Apr. 1997.

[7] Z.A. Ye, A. Moshovos, S. Hauck, and P. Banerjee,"CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," Proceedings of the 27th International Symposium on Computer Architecture, June 2000.

[8] K.R.Rao and P.Yip, "Discrete Cosine Transform," Academic Press, 1990.

[9] S. Mallat, "A Wavelet Tour of Signal Processing 2nd edition," Academic Press, San Diego, 2001.

[10] E. Oruklu, F. Martinez Vallina, and J. Saniie, "A Reconfigurable Architecture for Target Detection in High Density Clutter Environments using Subband Decoding Algorithms," GSPx 04 Technical Conference, Sept. 2004.

[11] T. Fujii, K.I. Furuta, M. Motomura, M. Nomura, M. Mizuno, K.I. Anjo, K. Wakabayashi, Y. Hirota, Y.E. Nakazawa, H. Ito, and M. Yamashina, "A Dynamically Reconfigurable Logic Engine with a Multi-Context/Multi-Mode Unified-Cell Architecture," Proceedings of the IEEE ISSCC 99, pp. 364-365, Feb. 1999.