Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing

Zain-ul-Abdin *, Bertil Svensson

Centre for Research on Embedded Systems (CERES), Halmstad University, Box 823, 301 18 Halmstad, Sweden

Article history:
Available online 5 November 2008

Keywords:
Reconfigurable architectures
Coarse-grained arrays
Computation models
Globally-asynchronous locally-synchronous

Abstract

In order to meet the increased computational demands of, e.g., multimedia applications, such as video processing in HDTV, and communication applications, such as baseband processing in telecommunication systems, the architectures of reconfigurable devices have evolved to coarse-grained compositions of functional units or program controlled processors, which are operated in a coordinated manner to improve performance and energy efficiency.

In this survey we explore the field of coarse-grained reconfigurable computing on the basis of the hardware aspects of granularity, reconfigurability, and interconnection networks, and discuss the effects of these on energy related properties and scalability. We also consider the computation models that are being adopted for programming of such machines, models that expose the parallelism inherent in the application in order to achieve better performance. We classify the coarse-grained reconfigurable architectures into four categories and present some of the existing examples of these categories. Finally, we identify the emerging trends of introduction of asynchronous techniques at the architectural level and the use of nano-electronics from technological perspective in the reconfigurable computing discipline.

1. Introduction

Media applications, such as voice and video processing as in HDTV, or baseband processing in cell phone base stations, are becoming increasingly complex and consume more and more computing power. In order to fulfill the computational demands of these and other applications, new parallel architectures are emerging, which are not based on von Neumann’s paradigm of computing [1]. This has resulted in an increased inclination towards reconfigurable computing, which is based on a sequence of configurations.

Over the last decade the field of reconfigurable computing has been drawing the attention of a wider audience because of the advantages of providing better price per performance and consuming less power per computation than conventional sequential CPU based computing. Furthermore, the capability of run-time reconfiguration of the computing hardware enables it to adapt to varying application requirements, thus providing performance comparable to ASICs without compromising the flexibility. The reconfigurable computing devices have evolved over the years from the gate-level circuit design to a more coarse-grained composition of functional blocks or even program controlled processing elements. In the latter case the resulting architecture is a reconfigurable array of cooperating von Neumann-style processors.

In more conventional parallel computing, we are witnessing a trend towards architectures consisting of multiple processing cores, as an energy-efficient way to achieve performance growth by exploiting parallelism, instead of scaling the clock frequency of a single powerful processor [2]. In short, if we view the design space of parallel and reconfigurable computing in a 3D space of reconfigurability, parallelism, and granularity, as depicted in Fig. 1, we can see that early microprocessors have developed to powerful uniprocessors, which have then been combined into multi-core processors on a chip. Adding reconfiguration in the interconnection network leads to coarse-grained reconfigurable computers (CGRC). At the same time, fine-grained logic arrays (PLAs) have developed into reconfigurable devices (FPGAs) and furthermore into more coarse-grained arrays of functional units or programmable processors.

Furthermore, programs intended for reconfigurable parallel architectures contain several parts that execute concurrently, which creates dependencies between the processing elements such that one element needs data computed by another element before it can continue with the processing [3]. It is difficult to realize these communication dependencies at compile time, which limits the exploitable parallelism. In order to meet high computational demands, suitable models should be devised for developing scalable and reusable applications. The models should enhance the
understanding of the developers to organize and manage computations for the reconfigurable architectures efficiently.

It is also highlighted in Fig. 1 that the trend towards multi-core architectures from uniprocessors is for achieving performance enhancements, and the trend from pure multi-core architectures towards CGRC is influenced by the requirement to adapt the architecture to meet the increased computational demands of applications. At the same time the evolution in reconfigurable devices from FPGAs towards coarse-grained programmable cores is influenced by the programmability aspects in order to increase their usability. Similarly, in the programming domain, the programming methods for FPGAs are moving from hardware description languages (HDLs) to high-level languages and the programming languages usable in mainstream parallel computing rather than imperative-style languages where parallelism is exposed by either compiler directives or pragmas. The motivation for this survey is to account for the paradigm shift in reconfigurable computing and emphasize the concerns from both architectural and programming perspective.

A number of surveys or reviews related to reconfigurable computing have appeared in the last decade. Some (e.g. [4–6]) are concerned with fine-grained reconfigurable architectures such as FPGAs and their corresponding design methods. This survey draws its significance because of its focus mainly on coarse-grained architectures and their programming models, with emerging examples introduced in the 21st century, which differentiates it from the earlier surveys on coarse-grained architectures [1,7].

The survey starts with an account of some of the fundamental characteristics of reconfigurable computing systems from an architectural perspective and their related computation models. Thereafter, we classify the coarse-grained reconfigurable architectures into four classes, namely hybrid architectures, array of functional units, array of processors, and array of soft processors, before presenting some recently introduced and some more established examples of these classes. The chosen examples provide a clear illustration of how the different classes of coarse-grained reconfigurable architectures are evolving. The survey concludes with providing the future trends with respect to architectural perspective, programming model, and advancements in technology.

2. Architectural characteristics

2.1. Granularity

The granularity of a reconfigurable fabric is defined as the size of the smallest functional unit based on its data width and computational capability. It can be broadly divided into two categories; fine-grained and coarse-grained. Fine-grained architectures, like FPGAs, use basic logic blocks with a data width of 2- to 4-bits, whereas the coarse-grained architectures consist of functional units working on data words of 8- or 16-bits or even more [4,6]. Thus the fine-grained architectures are more suitable for applications involving bit or irregular sized data manipulations. On the other hand the coarse-grained architectures are more suited for today's growing multimedia and streaming applications which require operations on longer word-lengths.

The advancements in technology have also led to the possibility of integrating more complex logic blocks on a chip. Thus the basic functional blocks in coarse-grained reconfigurable fabrics vary from simple logic blocks to multipliers, programmable ALUs, and further to small RISC type programmable cores. The latter style closely resembles the multi-core processors, but with rather simpler processor cores. It is this coarse granularity that allows a considerable reduction in routing and placement allocation, thus more easily avoiding routing congestion and enabling higher computational density. It further leads to a decrease in configuration time and configuration memory requirements as well as to a potential reduction in the total energy consumed per computation.

The coarser granularity of the reconfigurable fabric also improves the understanding of the architecture from a software perspective, thus improving programmer’s productivity, which, along with the area benefits, results in an overall reduction in system cost. The process of partitioning and then mapping the individual components of the applications to the target architecture is facilitated by a close relationship between the application, the run-time software, and the underlying hardware [5]. Knowledge about the static and dynamic behavior of the application enables the system developer to better extract the performance benefits of reconfigurable computing. For instance, the implementation of finite state machines is normally too complex to be mapped directly to fine-grained hardware units. Thus, for such implementations, the target architecture should have more program controlled computational power per logic block, still with the capability of executing bit-level computation.

2.2. Reconfigurability

The ability to dynamically allocate the resources of a computing device during run-time has been one of the leading characteristics of reconfigurable computing, because it allows the user to implement applications which are otherwise too large to fit in a particular device. Apart from area benefits, the run-time reconfiguration, sometimes also referred to as dynamic reconfiguration, also tends to extend the product life cycle, by providing new functionality in the form of configuration updates in the field. The major bottleneck of reconfiguration is the excessive reconfiguration time. Even if the reconfiguration overhead of coarse-grained architectures is far less than the one of their fine-grained counterparts [7], if reconfiguration is to be performed quite frequently, e.g., for selecting different modes of operation, it may overshadow the
performance advantage of spatial parallelism in hardware. Thus a useful practice is to partition the configurations in such a way that the reconfiguration time is only a small fraction of the computation time between these reconfigurations [8].

There are a number of techniques that can be applied to reduce the effects of the reconfiguration overheads. Unlike single-context devices [7] which require reconfiguration of the complete chip at a time, the commonly used partially reconfigurable devices allows the reconfiguration of selective portions of the chip [6], as shown in Fig. 2. The arrays supporting such kind of reconfiguration have different configuration bits arranged in the form of memory areas, which can be changed on the fly. This facility allows changing the operation of some parts of the chip while the rest are performing the execution, thus hiding the latency of reconfiguration.

Another form of partial reconfiguration is referred to as Multi-context reconfiguration [6], in which the memory containing the configuration bits is arranged in the form of multiple planes or multiple memory banks. The reconfiguration is handled by performing an ultra fast context switch between these memory planes or banks. Another way of overcoming the latency of the reconfiguration process is to pipeline the different contexts of reconfiguration. Such configuration sequencing is similar to instruction sequencing in von Neumann architectures. The dynamic reloading of the configurations can be under the control of a configware operating system [9], where configware means the structural description of reconfigurable hardware.

There are several methods to increase the speed of the reconfiguration process. A configuration cache can be provided to enhance the fetching of configurations from the configuration memory. The fetching of configurations can be further improved by supporting pre-fetching mechanisms. However, the presence of conditional jumps at the start of a configuration results in inaccurate prediction of pre-fetching, by which much of its benefit can be lost. Another technique used is to compress the configuration data, resulting in a reduction in size of the configuration.

From the software point of view, some of the languages for reconfigurable architectures have inherent constructs to help the compiler perform temporal partitioning of the applications to generate multiple configurations. A major problem related to the multi-context reconfiguration is the scheduling of contexts that are interdependent, in such a way as to avoid conflicts during computations [10]. Apart from partitioning, the compiler can also provide the inclusion of run-time customization facilities such as condition monitoring, reconfiguration control, and design optimizations. The possibility of performing optimizations based on run-time values is one of the major advantages of run-time reconfigurable computing.

The reconfiguration of FPGAs involves reconfiguring the network and the logic blocks at the gate level, whereas in case of coarse-grained architectures, the reconfiguration process is at the level of functional blocks and in some cases it only involves reconfiguring the interconnection network.

2.3. Interconnection networks

With an increasing number of computational blocks available on a chip, a major bottleneck may be caused by the communication among these functional blocks. This creates the need for scalable on-chip communication mechanisms. Simple on-chip communication techniques such as buses do not scale when the number of processing elements increases. An interconnection network is a programmable system for transporting information between nodes [11]. We will restrict our discussion here to the interconnection networks available on the chip, also referred to as Network-on-Chip [12]. A brief account of the state of the art techniques applied in intra-chip communication is presented along with their respective merits and demerits.

A general structure of a Network-on-Chip (NoC) is composed of the following three basic components, as shown in Fig. 3:

- Network adapter
- Routing node
- Link

The network adapter provides the separation between the processing core and the communication network, whereas the router implements the routing mechanisms to route the information on the links, which might consist of one or more physical or logical channels. The performance of the on-chip network is measured in terms of throughput and latency. The above NoC model can employ either circuit or packet switching. It allows parallel communication between different computing nodes, thus providing high aggregated bandwidth. It also facilitates the scalability and reusability of the individual blocks of the chip by making the architecture modular. In addition to this, the property of run-time reconfiguration of reconfigurable computing systems imposes additional constraints on the communication network. For example, it should be able to cope with the dynamically varying traffic demands caused by reconfiguration, and the reconfiguration time
of the network should be shorter than that of the computational blocks [12].

The most fundamental characteristics of an interconnection network are the following:

Network topology: The structure or organization of the switching and routing nodes and the channels represent the network topology. Examples of network topologies include ring networks, mesh networks, torus networks, and a combination of these topologies.

Flow control: The traffic flow of packets is controlled by the flow control mechanism by using flow control digits often called flits. It is further classified as either buffered or bufferless depending on the capability of buffering of packets. Circuit switching and time division multiplexing (TDM) are examples of bufferless flow control, whereas Worm hole [13] is an example of buffered flow control techniques.

Routing: Routing means the mechanism of selection of path between the source and the destination node. Based on how decision making is done it is categorized as either centralized or distributed. An example of a routing algorithm is dimension-ordered routing [14].

Quality of Service (QoS): Although the performance of a network is determined in terms of latency and throughput, different packets may have different requirements for the two parameters. On the basis of these requirements, the QoS of a network can be regarded as, e.g., best effort (BE) or guaranteed service (GS) [15].

2.4. Discussion

Having presented the architectural characteristics of granularity, reconfigurability and interconnection networks, we are now going to discuss how these properties are related to energy efficiency and scalability.

2.4.1. Energy Efficiency

The growing market demand for mobile computing devices has led to the development of low-power techniques in VLSI design. However, the trend is to achieve energy efficiency not only at the circuit and architecture level, but also reduce the power consumption at the system software and protocol levels.

Reconfigurable computing reduces the control overhead for instruction decoding, sequencing and communication between the computational blocks, thereby conserving the extra energy consumption, and has a particular advantage in achieving energy efficiency provided the granularity of the architecture matches that of the computational requirements of the application [16].

The active power consumed by a reconfigurable fabric of functional blocks can be given as [17]

\[ P_a \approx N_e F_o V^2 \quad (1) \]

Here \( N_e \) represents the number of functional blocks performing the computations, \( F_o \) is the operating frequency, and \( V \) is the supply voltage of the chip. The complete system power should also include the power required for configuring/reconfiguring the chip as well as idle power, if any. From this expression it can be deduced that the overall power consumption can be reduced significantly by reducing the supply voltage and frequency and compensating the loss of processing power by providing concurrent execution of computations. In the case of coarse-grained architectures a further improvement can be obtained by completely switching off parts of the fabric that are not actively performing any computation and also by managing the operating frequency of the active parts according to the computational requirements, in a way similar to the dynamic voltage scaling (DVS) [18] technique applied to microprocessors. The use of on-chip interconnection networks also helps in improving energy efficiency by allowing wider data paths and thereby facilitating the data transfers to be completed in fewer cycles, which allows the lowering of operating speed of the interconnects. One particular approach of using specialized circuits for reducing power and delay costs of router wires is proposed by Ho et al. [19].

From the software perspective, it is observed that the development tools should be able to partition the applications in favor of low-power implementation. This can be achieved by applying optimizations based on power-awareness such as the hybrid design approach [20] and PyGen [21]. In the hybrid design methodology, performance measurements are first performed by simulation. Then, based on these estimations, optimizations such as those proposed in right-size [22] are applied at the algorithm level to achieve better energy efficiency without compromising performance. In addition to these low-level optimizations, there are methods proposed by Stitt et al. [23] to adapt the system software and protocols for low-power applications.

2.4.2. Scalability

The scalability of a reconfigurable computer can be defined as the capability of the reconfigurable fabric, including the functional blocks and the communication network, to scale without compromising performance per block and without a dramatic increase in power consumption. Thus scalability is one of the most important factors considered in the research community when devising new architectural methodologies, keeping in mind the limiting effects of wire delays and power constraints on the scalability [24]. A possible way to limit the effects of wire delays is to use producer–consumer locality in the software implementation to avoid excessive global data movement. In order to achieve this, software development needs to be based on a suitable model of computation.

3. Computation Models for Reconfigurable Architectures

A computation model is an abstract representation of computational semantics that defines a high-level composition of an application, whereas a programming model provides an abstract view of the concrete syntax of the programming language [25] in terms of language constructs. In other words, a programming language is an instantiation of the programming model, and the programming model is itself derived from the computation model.

The procedural models of imperative languages such as C and Pascal consist of arrays, sequential control flow, procedures and recursion, which are difficult to adapt for parallel computing arrays [26]. Because these languages were originally designed for sequential computers, applying them for parallel computing arrays results in inefficient use of available hardware that leads to increased power consumption, increased wire delays, and limited extraction of instruction-level parallelism [27].

The computation models that are presented below have the inherent ability to express communication explicitly, separated from the computations, which makes them suitable for exposing parallelism. The separation of communication from computations allows the scheduling of communication in advance of the execution, which opens the possibilities of hiding the memory latency. The facility of exposing communication patterns to the developer helps in optimizing the global memory bandwidth requirement. In this section we present an overview of some of the computation models adopted for reconfigurable architectures, and also give examples of programming languages. The models that we consider are:

- Stream processing model
- Communicating sequential processes (CSP) based model
- Kahn process networks
- Spatial computation model.
3.1. Stream processing model

A stream processing system consists of three parts; a source that loads data into the system, transformers (also called filters or kernels) that carry out computation on the data, and sinks that output the data from the system. An illustration of this can be seen in Fig. 4. Stephens defines a stream as: “A stream is essentially an infinite list of elements a0, a1, a2, ... taken from some data set of interest” [28]. A stream transformer can have one or more input streams and one or more output streams [29]. The stream works like a FIFO queue where the first element put on the queue by the previous stream transformer is the first one received by the next transformer. The elements of the stream can be of any data type.

The stream processing model can be further classified based on the following characteristics [28]:

- Synchronous or asynchronous
- Deterministic or non-deterministic
- Unidirectional or bidirectional streams

The above mentioned features represent the behavior of filters. Based on the above features, typical signal processing systems are characterized as synchronous, deterministic, and having unidirectional streams where the dataflow is directed from the source to the sink. In case of these signal processing systems various filters communicate using signals which are not only used for synchronization but also carry the data.

The stream processing model separates the communication from the computational parts and leads to an intuitive programming model that can be directly mapped to communication-exposed architectures. The streaming model also helps the developer in identifying potential communication related bottlenecks by exposing the overall structure of the implementation.

3.1.1. Streaming languages

A programming language that is based on the stream processing model provides primitives to build stream transformers, mechanisms to combine stream transformers into a network and the possibility to define stream transformers and networks. In a stream programming language, the programmer has to define the stream transformers and show how they are connected to each other. The stream transformers do not have any access to global memory, but only to local memory and the incoming stream(s). This is good for reconfigurable computer architectures where the access to global memory is expected to be minimized, something that also speeds up computation because memory access is costly.

The stream transformer receives an arbitrary number of elements from the input stream(s) in each execution. These elements are used together with any local variables in the stream transformer for the computation of the output stream(s). As each stream transformer only uses the local memory and the incoming stream(s), the transformers can execute in parallel.

The most widely known stream programming languages for reconfigurable computing include StreamIt [29] and StreamC/KernelC [30] (based on statically schedulable dataflow), as well as task description format (TDF) [31] (based on dynamic rate model).

The StreamIt language is an implementation of the stream processing model that allows the compiler to construct a stream graph, containing stream transformers with a single input and a single output, and enables the development of efficient stream programs. The ability of the compiler to reconstruct the stream graphs enables it to combine adjacent stream transformers or split computationally intensive transformers into multiple parts to have a more parallel computation. The compiler optimizes the stream graph to produce efficient code so that not all the responsibility has to be put on the programmer. Furthermore, the language is especially built to facilitate implementation of streaming computations and simplify for the compiler to map the application onto a parallel architecture. Thus, the language allows programmers to write high performance applications for parallel systems without detailed knowledge of the target architecture.

The Stanford University developed a combination of two languages, StreamC and KernelC, to organize an application into streams and kernels. The notion of streams and kernels appears to be a natural way of exposing inherent locality and concurrency in expressing the media processing applications. The stream-oriented functions are expressed in StreamC and computations are described in KernelC. The compiler uses communication scheduling to coordinate the data transfer in streams and the execution of the KernelC program. The Stream programming model restricts the KernelC programs to accessing only local memory, which guarantees the producer-consumer locality [30].

The task description format is based on stream computation organized for reconfigurable execution (SCORE), which is derived from a stream computation model supporting dynamic flow rates on streams [31]. The TDF language can be seen as a hardware description language with built-in streams. The operators are defined as filters, which communicate with each other via designated stream connections, and the operators are always deterministic, which means that their behaviors can be determined entirely by their inputs. It is this property that allows the compiler to perform pipelining, restructuring of operators, and optimizations for better throughput without changing the system response. The streams are defined as buffered first-in-first-out queues with single input and single output. There is no timing relationship between stream producer and stream consumer, and both are asynchronous in nature, which allows greater flexibility in porting implementations across different hardware platforms. In order to support dynamic consumption/production rates the operators are implemented as finite state machines (FSM). Each state specifies the inputs whose presence is required for firing. These FSMs allow the developer to describe highly sequential control-oriented applications. The TDF language also allows instantiation of operators connected via streams. Thus a TDF application can be regarded as a hierarchical stream graph containing composition of operators.

The above streaming languages contain slightly different definitions of streams, stream access methods, and filter structures along with differences in exposure for compiler analysis. In addition, these languages provide semantics to capture the data-flow communication. In this respect both StreamIt and StreamC/KernelC belong to the category having statically scheduled dataflow, whereas TDF supports dynamic data rates.
3.2. CSP computation model

Communicating sequential processes [32] developed by C.A.R. Hoare, is a non-deterministic computation model for describing concurrency in applications. A CSP program is a composition of processes with communication between the processes being managed by unbuffered message passing channels. In the original version of CSP, the processes are sequential but can run in parallel based on the outer syntax level. There is no support for recursive processes, which means that a process cannot communicate with itself. Since synchronization in communication is managed by handshaking, an attempt for self-communication results in a deadlock. One way to overcome this shortcoming is to create a stack or array of processes and allocate a new process from the stack for each level of recursion [32]. Although CSP provides a lot of facilities, it still lacks the capabilities of timing, as well as arithmetic and data structures.

Over the years there have been efforts to extend the ideas of the original CSP model for reconfigurable hardware–software co-design. This has led to the development of CSPP (CSP including priority) and HCSP (Hardware CSP). Following is a brief description of the salient features of the two variants of basic CSP.

CSPP extends CSP with extra operators for expressing priority and this priority provides mechanism to cater for the degree of fairness [33]. The events of a process are sampled at different points and if they appear simultaneously, the process assigns priority to one event over the other. However, CSPP models concurrency in a similar way as classical CSP by retaining interleaving semantics.

HCSP, also regarded as Hardware CSP, is an extension of CSP that is focused towards a high-level abstract representation of reconfigurable hardware along with the support to describe fine hardware details [34]. HCSP inherits most of its semantics from CSPP in a way to avoid the complexity of a complete language by providing an abstract model for capturing hardware semantics such as logic blocks in FPGAs. All the assignment operations are regarded as events in HCSP, which involves updating of state and it is the event that determines the change of state. In synchronous circuits the clock edges are defined as origination of events between different processes. HCSP includes true concurrency by allowing simultaneous events, which was not available in either classical CSP or CSPP. Furthermore, the properties of the complete system modeled in HCSP can be entirely determined from the definition of its individual components, which facilitates the design of a correct system and makes the system modeling relatively easier. The system model described by HCSP is non-deterministic, which relieves the system designer from satisfying complex system specifications [34].

There have been a number of initiatives in both the industry and the academia to address the requirement of high-level languages for reconfigurable silicon devices. A few of them based on the CSP model are described here. Occam, which was not intended for direct hardware compilation, will be mentioned here because other languages such as Handel-C and Mobius have language semantics derived from Occam.

Occam is based on the CSP concurrent model developed by Inmos for their microprocessor chip transputer [35]. The language supports distributed computing such as application processes, which can be executed on either one transputer or a number of such devices with very little modification required. The communication between the processes is handled via channels using message passing, which helps in avoiding interference problems. Occam does not support shared variables, pointers, and dynamic memory allocation, and the communication is completely synchronized.

Handel-C is a high-level language with ANSI-C like syntax used to program gate-level reconfigurable hardware [36]. It supports behavioral descriptions with parallel processing statements (par) and constructs to offer communication between parallel elements. Handel-C is being used for compilation to synchronous hardware and inherits sequential behaviors. The communication between the different parallel blocks is handled by channels, which also manage the synchronization. The language constructs also help the developer to determine the timing behavior of the implementation. The developer can also implement pipelining to obtain better throughput by trading of latency.

Streams-C, a project initiated by Los Alamos National Laboratory, is based on the CSP model for communication between processes and used for stream-oriented FPGA applications. The Streams-C implementation consists of annotations and library function calls for stream modules. The annotations define the process, stream, and signal. The stream module uses standard interface to communicate between the process and the I/O port. The abstract syntax tree (AST), consisting of sequences of the basic and pipelined data path blocks, is generated by the compiler, and the compiler analyses the AST for partitioning of control and data flow blocks [37]. Streams-C, which is a subset of ANSI-C, lacks the support for two-dimensional arrays.

Mobius is a tiny, domain specific, concurrent programming language with CSP-based interprocess communication and synchronization methodologies using handshaking [38]. It has a Pascal like syntax with bit specific control and Occam like extensions, suitable for both fine-grained and coarse-grained architectures. The hierarchical modules in Mobius are composed of procedures and functions. The processes execute concurrently and communicate with each other through message passing unidirectional channels. A channel consists of req, ack and data signals and provides unbuffered, point-to-point communication. The data width can be user defined. An active port can initiate either a read or a write. There are also constructs to manage the flow control such as seq (sequential block), par (parallel block), alt (alternative communication), while, if(else), for and repeat. The non-deterministic alt construct can be used to choose between multiple channel events.

Although all of the discussed languages are based on the CSP computation model, they differ from each other in the way they expose parallelism. For instance, while Handel-C and Streams-C both have C like syntax, Streams-C relies on the compiler to expose parallelism but Handel-C has extensions of statement level parallel constructs to identify collection of instructions to be executed in parallel, which is similar to the approach taken in the recently emerging language Mobius. In CSP the channels between the processes are bufferless, which distinguishes it from our next computation model, which supports buffered communication channels.

3.3. Kahn process networks

Kahn process networks (KPN) [39] provides a model of computation for distributed systems, where processes communicate over channels to form a network. The channels are unidirectional and comprise first-in first-out sequences of tokens of unbounded capacity. Writing of the tokens is non-blocking, whereas reading is performed in a blocking manner. Thus a process is halted when attempting to read a token from an empty channel, and it further implies that a process is not allowed to check the availability of data items on the channel. Furthermore, the KPN model is monotonic and deterministic [40]. The monotonicity allows the processes to produce partial information of the output stream based on a partial input stream data, and this holds even for process networks with feedback loops [40]. The determinism offers the possibility to determine the output tokens, entirely based on the processes, the network and the initial tokens, without taking into account the execution order of the processes. The system model can be classified as an untimed model because there is no timing...
relationship involved among the process nodes. A direct conse-
quence of this property is that processes can be modeled as either
sequential or parallel programs as long as determinism is pre-
served. However, the FIFO buffer sizes do depend on the order of
read and write operations. Therefore, a scheduling policy has been
developed [41], which reduces the possible sets of order of execu-
tion of reads and writes to those where the buffer sizes are kept
within limits.

The major research work related to the process networks is
going on as part of the Ptolemy Project [42] at University of Califor-
tia, Berkeley. The Ptolemy Project is focused on studying models,
simulation techniques and design of embedded systems. Apart
from Ptolemy, the Artemis Project [43] explores the use of KPN
for modeling applications for reconfigurable computing architect-
tures. The Artemis Project proposes an approach in which a
sequential application written in a subset of MATLAB language is
translated to KPN by using the Compaan tool [44] and is targeted
to a hybrid reconfigurable architecture. Compaan translates the
MATLAB code into single assignment code before applying suitable
source transformations that convert the code to polyhedral re-
duced dependency graph (PRDG). Finally, the PRDG graph is asso-
ciated with the KPN network.

3.4. Spatial computation model

The Spatial computation model organizes the computations in
space rather than in time and is specialized for direct compilation
to hardware [45]. The resultant circuit has no global control and
the communication is restricted to be local.

In the spatial computation model, each implementation is tai-
lored for a specific application and there is no instruction overhead
involved for fetching, decoding, and execution. The absence of
instructions enables the computations to be mapped directly to
the underlying hardware and thus allows extraction of virtually
unlimited instruction-level parallelism. A simplistic communica-
tion mechanism based on handshaking protocol is used for re-
source arbitration instead of having complex control circuitry.
The spatial computation model is employed for asynchronous sys-
tems, because these are tolerant for latency and offer speed advan-
tages. The modularity that is inherent in the spatial computation
model facilitates the formal verification of the generated asynchro-
nous circuit.

Pegasus [46] is an intermediate representation derived from the
spatial computation model. Pegasus represents a control flow
graph, and compiler for application-specific hardware (CASH) is a
compiler that uses the representation of Pegasus.

4. Coarse-grained reconfigurable architectures

Having presented the basic distinguishing characteristics of
reconfigurable architectures namely granularity, reconfigurability,
and interconnection network, we are now going to outline some
illustrative examples of these architectures. We have divided the
overall list into four broad categories:

- Hybrid architectures
- Array of functional units
- Array of processors
- Array of soft processors

Each of these categories, illustrated in Fig. 5, represents a particu-
lar trend in the evolution of reconfigurable architectures. The hy-
brid architectures are characterized by having a host processor
which is tightly coupled with the coarse-grained reconfigurable ar-
ray. On the other hand, the array of functional units (FUs) is com-
posed of functional units of varying granularity where the
reconfiguration is under control of a configuration controller which
itself cannot perform any computations. The array of processors
consists of numerous processors (denoted as PR in Fig. 5) with pro-
gram counters and their own instruction memories, which differ-
entiates them from the previous two categories where there is
no mechanism for instruction fetch. The array of soft processors
depicts a flexible variation when compared with the previous cat-
egory; here the computing nodes (CN) are implemented as soft
processor cores.

Hybrid architectures: The granularity of the reconfigurable fabric
varies from fine-grained to coarse-grained, and the reconfiguration
of the fabric is controlled by the host CPU, which can itself perform
computations. In terms of interconnection network, the hybrid
architectures provide nearest neighbor connectivity in combina-
tion with global busses.

Array of functional units: The arrays of functional units also have
a granularity ranging from fine-grained to coarse-grained, but in
contrast to the hybrid architectures, the arrays of functional units
typically have dedicated configuration controllers as part of the
reconfigurable array and these controllers can only manage the
sequencing of different configurations, without having the capabil-
ity of performing any computational task. In terms of intercon-
nection the arrays of functional units provide the same kind of
facilities as the ones discussed for hybrid architectures.

Array of processors: From the granularity point of view, these
architectures are more coarse-grained than the architectures in
the previous two categories. In terms of reconfigurability, the basic
tile architecture remains fixed, but the interconnection network
between the basic building blocks is reconfigurable. In addition,
the arrays of processors have routing mechanisms for supporting
communication over the interconnection network.

Array of soft processors: The approach taken in the array of soft
processors, finally, is the most flexible among all of the categories
mentioned. Here the basic building block is implemented in soft-
ware, which can be changed along with the interconnection net-
work if desired.

For each of the above mentioned categories, we take one exam-
ple that is almost a decade old, while the rest of the examples are
among the more recently appearing examples belonging to that
category. For each of the examples, we describe the granularity,
reconfiguration process, interconnection network, and the pro-
gramming method used.

4.1. Hybrid architectures

A hybrid architecture, as the name depicts, consists of a combi-
nation of an ordinary processor with a reconfigurable fabric as a
computational engine to accelerate the computations. Since we
are focusing on coarse-grained architectures, we are going to dis-
cuss only the examples that have coarse-grained reconfigurable
units.

MorphoSys: A MorphoSys [47] chip, developed at University of
California, Irvine, is composed of an array of reconfigurable cells
(RC) with configuration memory (also called context memory), a
TinyRISC processor as a host processor, data buffer and a DMA
controller [48] as shown in Fig. 6. The RC array is arranged in the
form of an 8 × 8 2D-mesh with each cell consisting of an ALU to-
gether with a multiplier and a register file. The RC array is special-
ized for 8–16-bit word-length operations and operates in SIMD
fashion.

The 32-bit context words used to configure the individual cells
are stored in the context memory, which can store up to 32 planes
different configurations. The RC array allows run-time partial
reconfiguration by changing the non-active parts of the context
memory.
The interconnection network, apart from the nearest neighbor connectivity provided by the 2D-mesh, has additional row/column connectivity at intra-quadrant level of $4 \times 4$ cells as well as express lane connectivity at inter-quadrant level.

The host processor is based on a RISC architecture with some extensions in instruction set to handle the control of the RC array. It also initiates the data transfers to and from the frame buffer, whereas the fast data transfers between main memory (on a sepa-
rate chip) and the frame buffer is under control of the on-chip DMA controller.

The programming environment of MorphoSys consists of the Single Assignment C (SA-C) language and its compiler. The compiler partitions the application code into kernels for RC array execution and loops for TinyRISC execution [48]. The compiler generates configurations for the RC array and assembly code for the TinyRISC processor separately. Due to its SIMD nature of operation, the chip is recommended for data-intensive multimedia applications.

Zippy: The Zippy architecture [49] from Swiss Federal Institute of Technology (ETH) Zurich, integrates a scalable reconfigurable array unit (RU) and a processor core, which are connected via a coprocessor port, as shown in Fig. 7. The basic cell of the RU has an ALU based operator block with parameterized data-width and input and output registers.

The array contains configuration memory along with a context sequencer that controls the sequencing of configurations. The configuration memory can store multiple configurations simultaneously on-chip. There are also FIFOs for transferring data between the main processor and the reconfigurable fabric.

The RU consists of computing cells (denoted as CU in Fig. 7) and memory blocks organized in a mesh and interconnected by horizontal and vertical buses. In addition, each cell can read the output data from its 8 immediate neighbors through local interconnect.

The software toolset of Zippy contains a C compiler with library support for accessing the reconfigurable hardware. The toolset converts the application described in C to Zippy netlist format (ZNF) and then generates the binaries for the CPU and configuration memory. The Zippy architecture [49] from Swiss Federal Institute of Technology (ETH) Zurich, integrates a scalable reconfigurable array unit (RU) and a processor core, which are connected via a coprocessor port, as shown in Fig. 7. The basic cell of the RU has an ALU based operator block with parameterized data-width and input and output registers.

The array contains configuration memory along with a context sequencer that controls the sequencing of configurations. The configuration memory can store multiple configurations simultaneously on-chip. There are also FIFOs for transferring data between the main processor and the reconfigurable fabric.

The RU consists of computing cells (denoted as CU in Fig. 7) and memory blocks organized in a mesh and interconnected by horizontal and vertical buses. In addition, each cell can read the output data from its 8 immediate neighbors through local interconnect.

The software toolset of Zippy contains a C compiler with library support for accessing the reconfigurable hardware. The toolset converts the application described in C to Zippy netlist format (ZNF) and then generates the binaries for the CPU and configuration memory. The Zippy architecture [49] from Swiss Federal Institute of Technology (ETH) Zurich, integrates a scalable reconfigurable array unit (RU) and a processor core, which are connected via a coprocessor port, as shown in Fig. 7. The basic cell of the RU has an ALU based operator block with parameterized data-width and input and output registers.

Tartan: Tartan, developed at Carnegie Mellon University, is a hybrid architecture composed of a coarse-grained, clockless reconfigurable fabric (RF) [45] and a RISC CPU core, as shown in Fig. 8 (simplified view).

The RF is a hierarchical composition of RF pages to form clusters, and these clusters are then merged together at the chip level. The RF page is built from 16 stripes, each composed of 16 PEs. The individual PE is ALU based having 8-bit data width. The grouping of 4 \times 4 pages constitutes a cluster.

Tartan adopts a virtualization technique where program configuration is moved on and off the tartan RF at procedure granularity and each procedure uses a rectangular array of clusters. Whenever a call to a procedure is made, an off-fabric configuration engine fetches and configures the procedure at a location which is best suited according to its communication dependencies and the fabric resources.

The intra-cluster communication is performed on channels communicating through switch-boxes. The clusters are integrated with a dynamically routed, packet-switched NoC. The on-chip network is based on an asynchronous NoC called message-passing asynchronous Network-on-chip providing guaranteed services over OCP interface (MANGO) [50]. The physical interface between the CPU and the RF is a 96-bit wide bus, and mixed-timing FIFOs are used to communicate between the synchronous CPU and the asynchronous RF.

The Tartan architecture is intended for the spatial computation model. The application developed in the C language is partitioned, and then compiled by separate tools. The translation of C functions to RF logic netlist is performed in two steps, first by compiling C to Pegasus [46] intermediate representation (IR) and then synthesizing the IR to logic configurations. The configuration of cluster interconnect is determined statically by the compiler.

DAPDNA: The DAPDNA [51] from IPFlex Inc. is a hybrid architecture consisting of a 32-bit RISC core digital application processor (DAP) and a dynamically reconfigurable matrix, named distributed network architecture (DNA). The DAP controls the dynamic configuration of the DNA, which comprises 376 processing elements (PES) arranged in the form of a 2D-mesh, as shown in Fig. 9.

Each PE, as well as the interconnection network, can be reconfigured after every clock cycle by switching to another stored configuration. The DNA contains four memory banks for storing configurations, out of which one is always active while the remaining can be reconfigured in the background. The communication between DAP and DNA is via a high-speed bus interface, whereas the transfer of data to and from the DNA is performed over six channels of DNA Direct I/O.

The application development for the DAPDNA architecture can be performed in C language with data-flow extensions called data flow C (DFC) [52], and the compiler generates hardware binaries directly. The DFC compiler can also be used in collaboration with the MATLAB/Simulink design environment.

4.1.1. Discussion

The hybrid class of architectures comprises a combination of an ordinary CPU and a reconfigurable fabric that is tightly coupled to the CPU. The main concept is that the reconfigurable fabric
performs the bulk of the computations and the CPU is used to control the sequencing of the computations.

Granularity: From the granularity perspective the basic computing element of MorphoSys, Zippy and Tartan is an 8-bit wide ALU, whereas DAPDNA consists of 32-bit wide PEs.

Reconfiguration: MorphoSys, Zippy, and DAPDNA have multi-context configuration memories and thus they support runtime partial reconfiguration by context switching between the different configuration planes. Tartan allows the reconfiguration of individual clusters of the reconfigurable fabric at the procedure level.

Interconnection network: These building blocks are arranged in a 2D-mesh and are combined with varying interconnection techniques. MorphoSys and Zippy employ nearest neighbor connectivity together with additional connections for distant communication, whereas Tartan has a packet-switched NoC supporting dynamic routing for inter-cluster communication. In all of these architectures some sort of buffering has been implemented to synchronize the communication between the synchronous CPU and the reconfigurable unit.

Programming model: Another characteristic that is visible in these examples is that they adopt the programming model of imperative C language, which is then translated by the compiler to generate binaries for the CPU and the configurations for the reconfigurable fabric. Thus the reconfigurable array acts as a co-processor performing specific parallel tasks that are assigned to it by the imperative-style programmed host.

4.2. Array of functional units

The next category comprises architectures that are formed by a composition of a large number of reconfigurable functional units to perform computations. When compared to the hybrid architectures, this category lacks the on-chip control processor, and due to the absence of a central processing unit, complex algorithms are partitioned into a sequential flow of configurations to be dynamically configured into the functional units. The dynamic reconfiguration is controlled by the configuration manager, which itself does not perform any computations. Following are some examples of such devices.
**MATRIX**: Multiple ALU architecture with Reconfigurable Interconnect (MATRIX) [53], developed at MIT is chosen as our example of an early representative of this class. It consists of an array of identical 8-bit basic functional units (BFU). Each BFU contains 256 byte local memory, an ALU supporting 8-bit operations (such as NAND, NOR, XOR, shift and add), multiply unit and control logic as shown in Fig. 10. The MATRIX operations are pipelined at the BFU level. The BFU can perform three types of functions: instruction storage, operand storage, and ALU functions.

MATRIX has a configurable interconnection network, which provides the nearest neighbor connectivity as well as supporting bypass connections of length four and global lines on every node that span the entire row or column. MATRIX has a very flexible control for arranging functions, control and dataflow, which enables it to be customized according to the application requirements. It is because of this flexibility that it can be configured in either SIMD or MIMD fashion, or a combination of SIMD and MIMD computations. The application development for MATRIX is performed at the microcode level.

**PACT XPP**: The eXtreme Processing Platform (XPP) [54] from PACT XPP Technologies has a packet-oriented communication mechanism for both data and events, which makes it very suitable for compute intensive streaming applications. The PACT XPP-64A-1 processor core consists of an 8 × 8 array of ALU-PAEs (Processing Array Elements) in the middle, along with two rows of RAM-PAEs at the edges, as shown in Fig. 11. The ALU-PAEs consist of several objects that perform 24-bit arithmetic, shift and logical operations.

The interconnection network consists of horizontal and vertical buses, and data and event packets are communicated on every clock cycle over these buses. A configuration manager with integrated cache memory configures the array and controls the configuration sequencing. It also has a facility for partial run-time reconfiguration.

The XPP array can be programmed in either Native Mapping Language (NML) or XPP-VC language. NML is a low-level language based on the streaming model of computation which allows full access to all the resources of the array, whereas XPP-VC is a subset of C language, which is compiled by the XPP vectorizing compiler to generate NML code. The vectorization capability of XPP-VC allows loop iterations to be executed in a pipelined fashion.

**MathStar**: The Arrix family of Field-Programmable Object Arrays (FPOA) from MathStar [55] consists of 256 ALUs, 80 register files (RF), and 64 multiply-accumulator (MAC) units as array objects, as shown in Fig. 12. An FPOA also includes 12 banks of 2K × 76 bits internal memory (IRAM). The interconnection network allows each object to communicate with eight nearest neighbors and ten party line connections. The devices are capable of operating to a maximum frequency of 1 GHz.

The development of application for FPOA is done in object hardware description language (OHDL) which is similar to Verilog and supports both structural and behavioral description of modules. The floor planning and placement is performed in the Connection and Assignment Tool (COAST) [56].

---

**Fig. 10.** MATRIX network and BFU architecture.

**Fig. 11.** Block diagram of XPP64 A-1.
Silicon Hive: The Dutch company Silicon Hive provides their instance of reconfigurable architecture in the form of intellectual property cores. The basic building block of the architecture is the processing and storage element (PSE) [57]. Each PSE consists of a number of interconnection networks (IN), one or more operation-issue slots (IS) with associated function units (FU), distributed register files (RF), and local memory (MEM) as an optional component, as shown in Fig. 13. The PSE has a data path similar to that of VLIW architectures.

A combination of multiple PSEs along with the configuration memory and a VLIW controller constitute a cell. The PSEs within a cell communicate via communication lines. Each cell can perform an independent computational task in an application, so the number of PSEs used in a cell depends on the computational requirements for the task allocated to the cell. A number of cells can be arranged with a data driven communication mechanism to form a streaming array.

The toolset available from Silicon Hive allows the user to adapt the arrangement and configuration of PSEs and cells to different application domains. The toolset consists of a partitioning compiler for partitioning ANSI-C code between the host and the array, a spatial compiler to compile the code for the array, and the array mapping tool Silicon Hive array programming environment (SHAPE).

Dynamically reconfigurable processor (DRP): The DRP core from NEC [58] is a coarse-grained reconfigurable system consisting of processing elements (PEs) along with memory units, arranged in the form of two-dimensional arrays to form a computing block. Each PE of the DRP has an 8-bit data path and is composed of an ALU, a register file and a data multiply unit (DMU), as shown in Fig. 14. In addition, the DRP core also contains a state transition controller (STC) to manage the dynamic reconfiguration of the processing elements, which can be performed in a single cycle.

A single DRP tile is composed of 8 x 8 PEs, and NEC’s first silicon device DRP-1 consists of eight such tiles giving it a total of 512 PEs. The chip is specialized for streaming data applications. Application development for DRP is performed in the C language, and the compiler partitions the source code to generate configurations for the PEs and the control code for the STC.

4.2.1. Discussion

Architecture: A fundamental difference between the example architectures in this category lies in the internal memory arrangement, where MATRIX and Silicon Hive have local memory available within each individual basic functional unit, whereas XPP, FPOA and DRP have memory banks arranged external to the functional units and accessible by all the functional units across the array. Another significant difference is the size of the data path, where MATRIX and DRP have 8-bit data width, and MathStar and XPP have 16-bit and 24-bit data widths, respectively. There is also a gradual increase in complexity of the building blocks, starting from homogeneous reconfigurable ALUs in MATRIX and XPP to complex PEs with heterogeneous structure in Silicon Hive and DRP.

Reconfiguration: DRP differs from the others in the sense that it supports partial run-time reconfiguration by having multi-context configuration memories and switching between different contexts, whereas the others implement some sort of configuration data delivery mechanism.

Interconnection network: MATRIX and FPOA use nearest neighbor connectivity, while XPP and DRP provide communication over buses.

Programming techniques: The programming techniques for these architectures still rely on low-level languages which are relatively close to describing the corresponding hardware. Although some of them, like XPP and DRP, are programmable by using C language, the compilers for this high level language cannot generate as good performance as their low-level language counterparts, and thus need significant improvement.

Fig. 12. MathStar’s FPOA block diagram.

Fig. 13. Silicon Hive’s architecture.
4.3. Array of processors

In the class of array of processors we are considering scalable architectures consisting of a number of simple processors having local memories and specialized interconnect to support interprocessor communication. These processor arrays allow greater flexibility when compared to the previous category of arrays of functional units because of their instruction fetching at run-time. Although the basic architecture of the individual processing unit remains fixed, the interconnection network can be configured dynamically.

RAW: The Reconfigurable Architecture Workstation (RAW) [59] from MIT is a scalable tiled processor with point-to-point pipelined network. In its first implemented version, the processor consists of 16 tiles arranged in a 2D-mesh arrangement. Each tile consists of a MIPS style compute processor comprising a single-issue 8-stage pipe along with a 32-bit floating point unit, 32KB of instruction and data cache memory and two routers, for static and dynamic dimension-ordered, worm-hole routing, respectively, as shown in Fig. 15. There are also four on-chip interconnection networks, two static networks used for operand fetching, one memory dynamic network for cache misses and DMA traffic and one general dynamic network for message passing.

The application programming for RAW can be done in StreamIt, which is a high-level portable streaming language for communication-exposed architectures. The StreamIt compiler backend for the RAW processor implements a three-step technique: partitioning of the stream graph, mapping of the stream graph to the network topology, and scheduling of the communication. In addition, the tile to tile communication can be statically controlled at compile time by configuring the static switches available in all tiles [60].

Quite recently, it has been reported that Tilera has adopted the RAW architecture to build a TILE64 processor [61] consisting of 64 homogenous tiles coupled with five 2D-mesh networks.

PicoArray: The picoArray from Picochip is a high performance communication processor optimized for wireless signal processing applications. PC102, one of the implementations of picoArray, consists of a heterogeneous array of 322 16-bit RISC Processor Array Elements (PAE) and 14 Functional Accelerator Units (FAU) [62]. Each processor has separate local instruction and data memories organized as in a Harvard architecture. The array elements in PC102 are arranged in a 2D grid and communicate over a network consisting of 32-bit buses and programmable bus switches, as shown in Fig. 16.

Application development for the picoArray involves specifying interactions between processes in the form of signal flows. This is done in structural form using VHDL. Processes for the individual processors are programmed in either C language or assembly language [63].

Ambric: The basic unit of the architecture from Ambric Inc. is called a bric. Each bric is composed of two pairs of units, each pair consisting of a Compute Unit (CU) and a RAM Unit (RU) [64]. The CU consists of two 32-bit Streaming RISC (SR) processors with 64 words of local memory, two 32-bit Streaming RISC processors with DSP extensions (SRD), and a 32-bit channel interconnect for interprocessor and inter-CU communications, as shown in Fig. 17.
RU consists of four banks of 1KB of RAM each, along with a dynamic channel interconnect to facilitate communication with these memories. The Am2045 core from Ambric consists of 45 brics arranged in a 5×9 array. Since each bric has eight processors, the Am2045 encapsulates a total of 360 processors with 585KB of on-chip memory.

The hierarchical composition of brics allows the processors within a bric to be locally synchronized with each other but they are globally-asynchronous with respect to other bric processors. The operating speed of a group of processors (one or more brics) can be varied from less than 1.0MHz to 333MHz depending on the software workload assigned to the particular group. Thus the Ambric fabric is based on the globally-asynchronous locally synchronous (GALS) architecture. It uses the KPN model of computation with bounded buffering. In terms of programming model, Ambric adopts a structured object programming model, where the individual objects are programmed in a sequential manner in a subset of the Java language or assembly language [65]. These objects communicate with each other using hardware channels without using any shared memory. These individual objects are then linked together using either a visual interface or by using the ast-struct language.

4.3.1. Discussion

Granularity: In all the examples in the array of processors category, the basic building block consists of one or a few simple processors. In the case of RAW the individual tile consists of a single MIPS style processor; the picoArray comprises a 16-bit RISC processor, whereas Ambric’s individual bric is composed of two 32-bit SR and two 32-bit SRD processors. Thus from granularity point of view Ambric provides more computational capacity.

Interconnection network: The dynamic communication network of RAW supports packet-oriented routing, while Ambric has circuit switched interconnect of channels and picoArray has programmable switched buses. The static communication network of the tiled processors gives support to communication patterns that can be predetermined at compile time, which makes it suitable for applications where the compiler can extract and statically schedule fine-grain parallelism.

The execution of multiple processors simultaneously, irrespective of whether they are performing any computations or not, results in higher power consumption than needed. In the case of picoArray, the communication is determined at compile time, but the individual processors can be turned to sleep mode when waiting for communication events or turned off when not used by the particular application, thus making low-power design possible. Ambric’s architecture is based on GALS technology and has a more flexible approach to adapt the processing speed according to the computing requirements of the application. Here the clocks of the individual cluster of processors can be adjusted dynamically, thus conserving energy.
**Programming model:** The structured object programming model of Ambric is tailored to its specific hardware architecture, allowing the programmer to form a clear relation between software objects and the underlying hardware bricks. On the other hand, RAW has a streaming programming model which facilitates the extraction of parallelism, but the optimization of the stream distribution is mainly dealt with by the compiler.

### 4.4. Array of soft processors

An array of soft processors, sometimes also called soft instruction processors, consists of programmable instruction processors implemented in reconfigurable logic such as FPGAs. Due to increased cost and time to market for ASICs, soft processors have found increased deployment in FPGA based embedded systems. Although the soft processors cannot meet the area, power and speed characteristics of their corresponding hardware solutions, they can take advantage of the reconfigurability available in FPGAs to match the complexity and implementation requirements of the application. Major FPGA vendors have provided their own soft processor cores to be used on their FPGA platforms, such as MicroBlaze from Xilinx and Nios from Altera. Soft cores can also be used as elements in larger processor arrays, which gives solutions similar to the class of array of processors presented earlier, but in a more flexible way. Below we provide a brief overview of the Mitrion platform which is portable to a number of FPGA computer platforms. It differs from the conventional soft processor cores in the way it offers the parallelism.

**Mitrion Virtual Processor:** The Mitrion virtual processor from Mitrionics consists of a cluster of soft computing nodes placed on the same FPGA [66]. The company offers a processor configurator to tailor the virtual processor configuration according to the application requirements. The interconnection network between the compute nodes is adapted according to the application to provide an adhoc network with simple point-to-point links having a guaranteed latency of a single clock cycle. Since each node is able to execute a single instruction per clock cycle, the resulting behavior can be viewed as transforming the instruction sequencing in a traditional von Neumann architecture to dynamic packet switching by the network in a Mitrion architecture. The programming language Mitrion-C allows the programmer to extract parallelism based on data dependencies rather than on the order of execution.

Thus, the methodology adopted by Mitrionics provides the flexibility of developing applications at a high level, whereas the optimizations in mapping to hardware are achieved by providing virtualized processor architecture in the form of configurable soft IP cores.

### 5. Discussion

We now discuss the applicability of the computation models to the different classes of coarse-grained reconfigurable architectures. An illustration of this relationship, based on the example architectures and the ways they have been programmed, is shown in Fig. 18. Obviously, selecting an inappropriate model of computation might result in an inefficient implementation. It is also desired for efficiency reasons that the programming models expose fine-grained bit-level control to the programmer. Furthermore, researchers at Berkeley [42] have studied the use of different computation models for a wide variety of application domains, and their assessment was that no single computation model can cover the whole range of applications. For embedded systems incorporating reconfigurable architectures, the most important properties of the computation models are their handling of concurrency, communication mechanism, and timing as summarized in Table 1.

The streaming model has the characteristics of being synchronous, deterministic, and bounded, which makes it suitable for data-intensive systems with regular communication patterns. The model exposes concurrency explicitly and allows the static scheduling of firing of filters at compile time, and these filters can then be mapped to a variety of coarse-grained computing architectures, as witnessed in Fig. 18.Unlike the streaming model, the CSP-based model of computation is asynchronous, non-deterministic, and uses unbuffered channels with message passing for describing communication between processing nodes. The non-determinism of CSP enables it to model unpredictable sequences of events, and this makes it suitable for being transformed to

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Stream processing</th>
<th>CSP</th>
<th>KPN</th>
<th>Spatial computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronism</td>
<td>Synchronous</td>
<td>Asynchronous</td>
<td>Asynchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Determinism</td>
<td>Deterministic</td>
<td>Non-deterministic</td>
<td>Deterministic</td>
<td>Deterministic</td>
</tr>
<tr>
<td>Buffering</td>
<td>Bounded</td>
<td>None</td>
<td>Unbounded</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 1: Summary of salient features of computation models.

![Fig. 18. Example based relation diagram between computation models and computing architectures.](image)
configurations of asynchronous fine-grained hardware, e.g. on FPGAs.

KPN models concurrency in the form of networks of processes while retaining the data-flow characteristics. The channels between processes have unbounded buffering capability and the communication over the channels is performed by asynchronous message passing. The process networks allow dynamic data rates between operators, which makes them more powerful than statically schedulable streaming models. The resultant model is deterministic, but a drawback of this technique is that it is difficult to determine bounds on buffering requirements. The KPN model can be applied to applications that have multi-rate behaviors, and it has been applied for fine-grained and array of processor architectures, as evident in Fig. 18. The spatial computation model is implemented as asynchronous, deterministic, and without having any buffering, and is specialized for hardware with distributed program implementation and lack of centralized control.

Based on the example architectures presented in Section 4, we can identify some evolutionary trends over the years. During a ten years period we observe that the individual building blocks are designed to perform increasingly complex computational tasks and that the widths of the data paths in the interconnection network is also increasing. This trend is visible in all the coarse-grained architecture categories, as shown in Fig. 19.

In the figure we have indicated the relative complexity of the PEs as a number reflecting the number of gates in the PE with a variation from the simplest to the most complex PE structure. There is also a trend within each of the categories to introduce heterogeneity either within the structure of the individual PE, as witnessed in Tartan, Silicon Hive, and DRP, or to have heterogeneous building blocks, as seen in picoArray and MathStar. Another trend visible is that more industrial initiatives (shown as underlined in Fig. 19) are coming forward in recent years in all categories.

A trend across the categories is to keep reconfigurability at the interconnection network level and have heterogeneous programmable cores implemented with instruction streaming. The interconnection network shows a trend towards hierarchical composition of nearest neighborhood connectivity channels along with routing mechanisms for distant communication. We also observe that, while in the past the techniques used to program these architectures involve either low-level micro-coding or generating data-flow graphs from C language source code, the emerging trend is to base the application development on concurrent computation models such as CSP, KPN and Stream processing.

6. Future trends

Now let us look at some of the emerging future trends from the architectural and technological perspective.

6.1. Architectural aspects

We have seen from the previous section that there is a growing trend towards increasing the number of functional blocks or processors on a chip to achieve performance enhancements. As the number of basic blocks increases, the complexity of distribution of clock increases too. Following the ideas of the VLSI community, the negative impacts of the global clock distribution in the multi-core architectures can be minimized by adopting the GALS principle at a coarse-grained level. One of the emerging examples of such an implementation is Ambric's massively-parallel MIMD style array of RISC processors communicating over a fabric of asynchronous message passing channels [65]. The appropriate computation model for such architecture builds on processes which communicate using messages. Examples of such computation models include CSP and Kahn Process Networks.

As an intermediate step for going from a totally synchronous circuit to a totally asynchronous circuit, researchers have introduced the concept of GALS systems first proposed by Chapiro et al. [68] in 1984. GALS systems are modular and they allow different modules to operate at their corresponding optimum frequencies, which can be applied to conserve energy. The idea of enclosing synchronous modules within asynchronous wrappers to produce the effect of timing independence was proposed by Muttersbach et al. [68]. It is an established fact now that using asynchronous techniques in the design of very large scale integration (VLSI) systems has some interesting advantages when compared to synchronous circuits [69]. Asynchronous circuits have better speed because the next computation step is performed immediately after the completion of the current step, without waiting for the clock transition, and as a result they consume less energy. On the other hand, they require extra logic for implementing control and are expensive to build because of limited CAD tools support and inadequately trained manpower.

6.2. Technology trends

The phenomenal growth of the use of computing devices in increasingly challenging applications has resulted in paradigm shifts not only at the architectural level but also from the technology perspective. At the architecture level we have seen the
emergence of multi-core processors as witnessed by the examples in preceding sections. From the technological point of view, researchers have introduced new nano-electronics based process technologies as an alternative to complementary metal-oxide semiconductor (CMOS) devices. These molecular scale technologies have the advantages of high speed, high density, and low power, but bear the disadvantage of having higher fault rates and more frequent defects [70]. Fault tolerance of nano-devices can be introduced by making use of redundancy. The impact of defects can be reduced by introducing the feature of reconfigurability in the fabricated devices. Thus the reconfigurable computing architectures consisting of functional units that are well suited for the realization of nano-technology due to their inherent property of reconfigurability. As an immediate step towards building a system entirely based on nano-technology, researchers have proposed the hybrid approach of combining nano-scale devices with CMOS devices to build larger circuits. A combination of CMOS with chemically assembled electronic nano-technology (CAEN) is proposed by Goldstein and Budiu [71], using a molecular-based reconfigurable switch as the main computing element of the fabric. Another example of such a hybrid technique is CMOL [72] which stands for CMOS/nano-wire/Molecular hybrids and can be used to build uniform FPGA like structures.

Therefore, the success of the nano-scale architectures relies on effective use of their enormous computing resources by employing techniques such as reconfigurability and introduction of asynchrony, so that applications based on such devices can tolerate their fault-prone nature. There is also a need for exploring new computation models as well as design tools for modeling of the nano-devices.

7. Conclusions

We have presented an overview of the fundamental characteristics of granularity, reconfigurability, and interconnection networks and provided some of the emerging examples of coarse-grained reconfigurable architectures. We have also outlined the computation models used for programming these architectures and discussed their relationship with the different classes of hardware architectures.

It has been observed in all of the discussed computation models that the main aim of devising these models is to abstract away the computational complexities of the hardware from the programmer, while retaining the functional effectiveness. These models allow optimizations and correct design quite easily. One of the major concerns in the reconfigurable computing community is the existence of long interconnect delays which has a negative effect on the efficiency of the system when adopted for high performance computing. For dealing with such (interconnect delay) problems the computation models that handle asynchrony are useful, and a growing trend in the design of hardware is to introduce asynchrony at the inter-module level and apply synchronous implementation within the module. This technique is also referred to as GALS (Globally-Asynchronous Locally-Synchronous).

Acknowledgements

We would like to thank our colleagues (Dr. Veronica Gaspes, Jerker Bengtsson, Andreas Persson, and Prof. Dan Hammerstrom) for their valuable feedback during the internal review of the paper.

References


177