IPSJ Transactions on System LSI Design Methodology Vol. 4 31–46 (Feb. 2011)

Invited Paper

Coarse-Grained Reconfigurable Array: Architecture and Application Mapping

Kiyoung Choi[†1]

Coarse-grained reconfigurable arrays, or CGRAs in short, have drawn increasing attention recently due to their performance and flexibility. They provide flexibility through reconfiguration, which is not attained with fixed harware such as traditional ASICs. They also provide performance through highly parallel architecture, which is hardly achieved with basically sequential software running on full-blown processors. There have been many researches on CGRAs, and many of them are commercialized or in practical use. However, they still face some challenges that are to be addressed for their widespread use. In this paper, we survey existing CGRA architectures as well as existing approaches to the mapping of applications onto the architectures.

1. Introduction

With the ever-increasing requirements for more flexibility, higher performance, and lower power consumption in embedded systems design, there have been continuous efforts on finding new architectures that fit better with the requirements. ASIC design approaches have been used for many years for their merit in performance and power consumption. Recently, processor cores are replacing ASICs mainly due to the flexibility of software implementation. It allows reusing existing architectures and debugging without involving hardware redesign, thereby helps reduce time-to-market as well as development cost. In general, however, the performance of software implementation is orders of magnitude lower than that of hardware implementation, even with much larger power consumption. It is a current trend to integrate multiple cores on a chip to alleviate such problem, but software only implementation is not yet a solution for high-end embedded systems. As an alternative solution, one can consider using coarse-grained reconfigurable arrays (CGRAs). They provide flexibility through reconfiguration, which is not achieved with fixed harware such as traditional ASICs dedicated to a single functionality. They also provide performance through highly parallel architecture, which is not possible with basically sequential software running on full-blown processors. Conversely speaking, of course, the performance of CGRAs cannot be as high as that of dedicated hardware and the flexibility cannot be better than that of software. **Figure 1** (slightly modified from the figure in Ref. 1)) is a conceptual view that shows where typical CGRAs are positioned in the performance-flexibility tradeoff space. We can say the same about power efficiency or area efficiency instead of performance. It is therefore important to design CGRAs such that the performance, power, area, and flexibility merits are best utilized.

The difference between fine-grained reconfigurable arrays (FGRAs) and CGRAs is in the granularity of reconfiguration. An FGRA such as field-programmable gate array (FPGA) typically consists of an array of gates (or bit-level logic blocks) and flip-flops, and the reconfiguration is performed at the bit-level. The reconfigurable array used in Garp [2] also belongs to this class. On the other hand, a CGRA typically consists of an array of ALUs and registers, and the reconfiguration is performed at the word-level. For an implementation of random logic operations, FGRAs can provide a more efficient solution. However, for word-level operations, CGRAs can provide better optimized computation elements. In addition, since CGRAs have much less number of program points, the size of *configurations* (sometimes called *contexts* or *instructions* for the PEs) is much smaller compared to the size of the bitstream used for programming FGRAs. Therefore, the overhead of reconfiguration of a CGRA is much lower than that of an FGRA, thereby making it easy to reconfigure a CGRA dynamically.

There have been many researches on CGRA [3)–18)], and some CGRAs have been commercialized directly or indirectly [19),21),22)]. However, they still face some challenges that are to be addressed for their widespread use. First, they need to further improve the performance and reduce the power consumption without increasing silicon area much. It is important for differenciating them from software implementation. In particular, area and power efficiency is important when

†1 School of EECS, Seoul National University

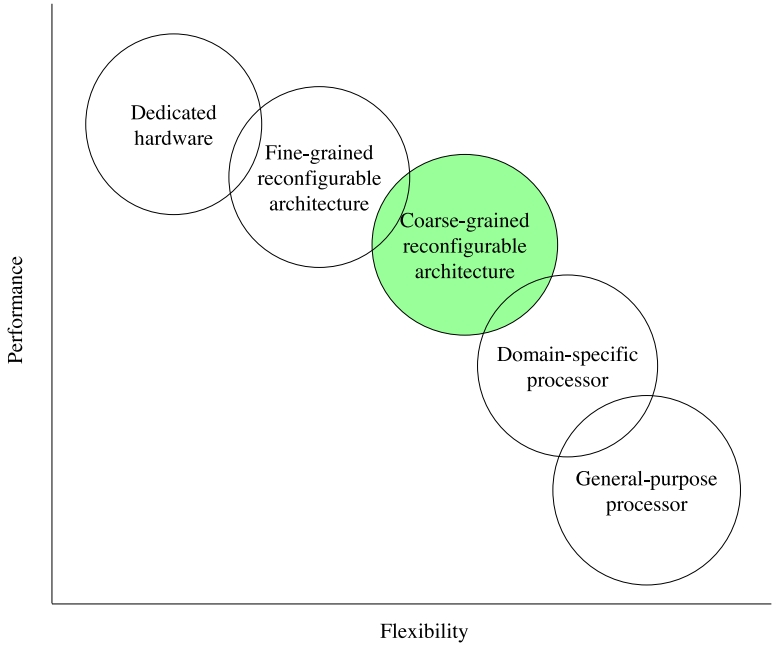

Fig. 1 Performance-flexibility tradeoffs (not in scale).

compared to software implementations on many-core architectures. More importantly, programming CGRAs must be as easy as parallel programming, if not easier.

In this paper, we survey existing CGRA architectures and see how to enhance them in terms of performance, power consumption, and area cost. In addition, we survey existing approaches to the mapping of applications onto CGRAs since it is one of the most challenging problems to be solved.

2. CGRA Architectures

There have been various CGRAs with different target domains of applications and different tradeoffs between flexibility and performance. In Ref. 23), Hartenstein summarized many CGRAs that had been suggested until 2001. Those CGRAs can be classified into two groups: linear reconfigurable array and mesh-based reconfigurable array. Most of them comprise a fixed set of specialized PEs and interconnection fabrics between them. The run-time control of the operation of each PE and the interconnection provides the reconfigurability. Mesh-based reconfigurable arrays arrange their processing elements (PEs) as a rectangular 2D array with horizontal and vertical connections, which support rich communication resources for efficient parallelism. In the case of linear reconfigurable arrays, they support pipelined execution for stream-based applications with static or dynamic reconfiguration.

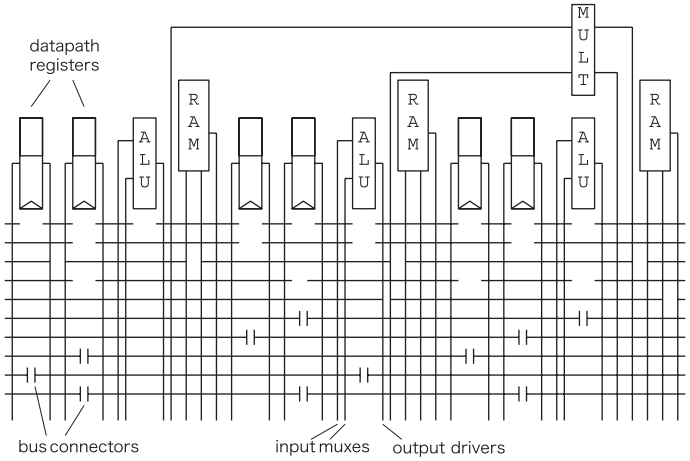

Fig. 2 A basic RaPiD cell [7].

RaPiD [7] (Reconfigurable Pipelined Datapath) and PipeRench [11] have a linear array structure. RaPiD provides different computing resources like ALUs, RAMs, multipliers and registers. **Figure 2** shows a basic RaPiD cell, which is replicated left or right to form a complete array. These resources are irregularly distributed on one dimension and are mostly reconfigured in a static way using bus segments, multiplexers, and bus connectors. However, PipeRench relies on dynamic reconfiguration, allowing the reconfiguration of a processing element (PE) in each execution cycle. It consists of stripes composed of interconnects and PEs with registers and ALUs. The reconfigurable fabric allows the configuration of a pipeline stage in every cycle, while concurrently executing all other stages.

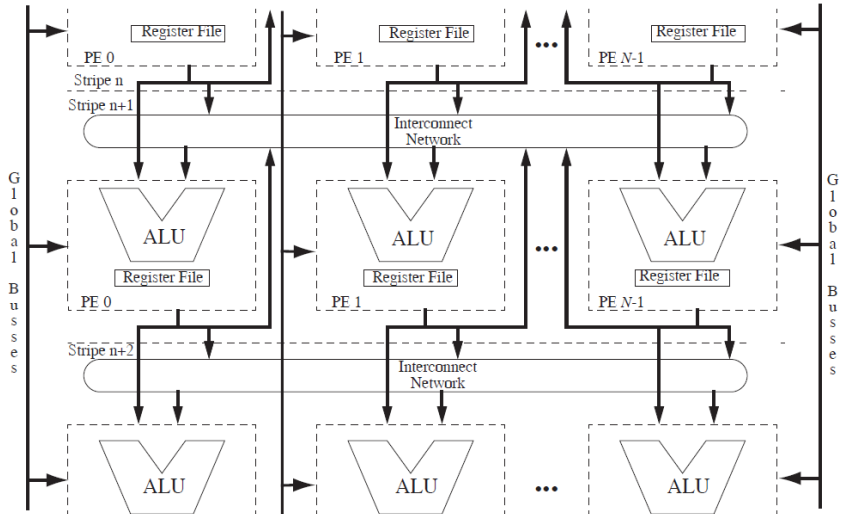

Fig. 3 PipeRench architecture [11].

Figure 3 shows the architecture of PipeRench. Such linear array structures are good for applications that can be executed in the form of linear pipeline. However, mapping of an application that performs 2D data processing or forks into multiple branches onto such an architecture will be inefficient and require many global or non-local interconnects.

Most of the CGRAs have 2D array structure [3),5),6),8)–10),12),14)–16)]. REMARC [12] (Reconfigurable Multimedia Array Coprocessor) and MorphoSys [14] are representative examples of 2D array architectures. REMARC consists of a global control unit and an 8×8 array of 16-bit nano processors. A nano processor consists of an ALU, a 16-entry data RAM, an 8-entry register file, data input registers, and data output registers. The configuration for each nano processor is stored in the 32-entry instruction RAM to support MIMD execution model as well as SIMD model. However, each nano processor does not directly control the instructions it executes. Every cycle, the nano processor receives a PC value called nano PC from the global control unit. All nano processors use the same nano PC and execute the instructions indexed by the nano PC in their nano instruction RAM.

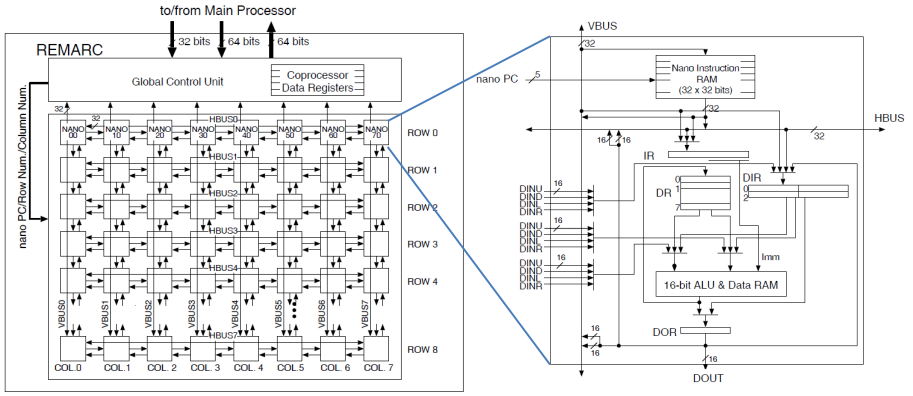

Fig. 4 REMARC architecture [12].

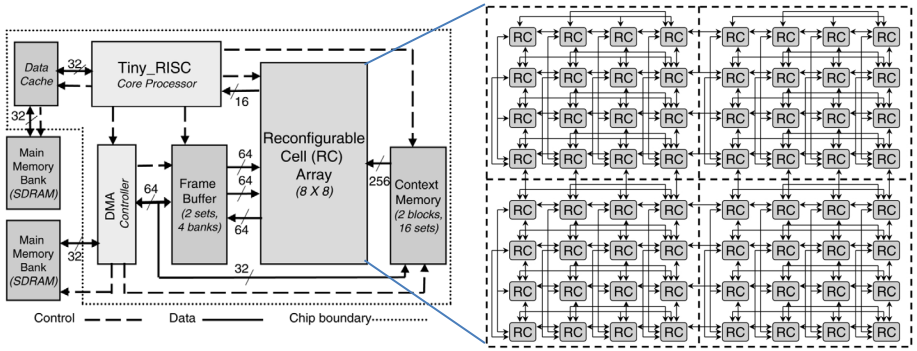

Fig. 5 Morphosys architecture [14].

Figure 4 shows the architecture of REMARC.

As shown in **Fig. 5**, MorphoSys consists of Tiny_RISC processor, RC (Reconfigurable Cell) array, frame buffer, context memory and DMA controller. RC array is an 8×8 array of ALUs that performs 16-bit operations based on SIMD programming model. The ALUs are connected by a multilevel interconnection network having abundant interconnect resources as compared to a simple mesh network. The frame buffer consists of two sets of memory thereby supports double buffering, which helps hide communication overhead. The DMA controller

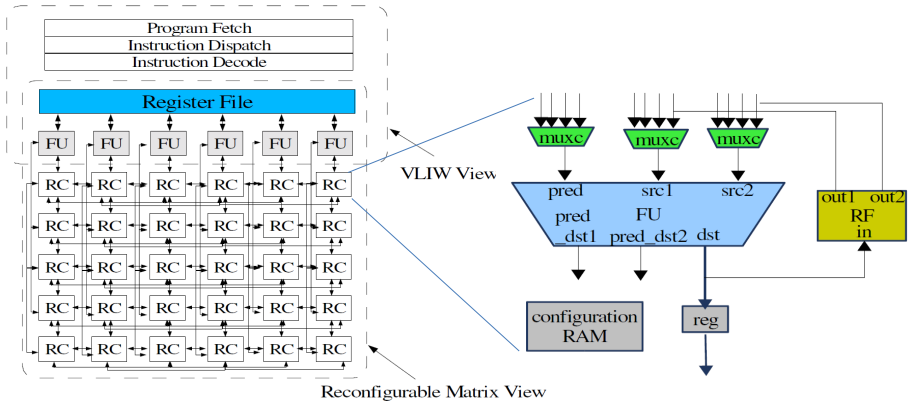

Fig. 6 ADRES architecture [17].

is used to perform more efficiently the data transfer between the frame buffer and the main memory. Depending on applications, however, the communication between the frame buffer and the main memory through a 32-bit bus can still be a bottleneck of the overall system performance.

The MorphoSys M1 chip [24], implemented with 0.35 μm CMOS technology, has area of 14 mm×12 mm and shows peak performance of 6.4 GOPS on 16 bit data at 100 MHz clock frequency with 3.3 V power supply. It is reported in Ref. 14) that MorphoSys shows performance about 23 times higher than Pentium MMX [25] for motion estimation of MPEG2 encoder. Pentium MMX has 4.5 milion transistors consuming 7.9 W typical power, whereas MorphoSys has about 1.56 milion transistors consuming less than 5 W power for the motion estimation. In general, compared to general purpose processors, 2D array CGRAs like Mophosys can achieve enormous speedup with much less power consumption using abundant PEs for applications with high degree of data-parallelism.

Many more new CGRAs have been continuously proposed and evolved [17)–19),21),22]. ADRES [17] (Architecture for Dynamically Reconfigurable Embedded System) is a VLIW architecture tightly coupled with a reconfigurable array of PEs. **Figure 6** shows the architecture of ADRES having two views: VLIW view and reconfigurable matrix view. The reconfigurable matrix part works as a co-processor of the VLIW part and so their executions never overlap with each other, which allows them to share some physical resources. The tight coupling also allows the VLIW processor and the reconfigurable matrix to communicate efficiently. However, since they cannot execute concurrently, the code running on the VLIW processor and the rest of the code accelerated on the reconfigurable matrix may not be pipelined efficiently. For the VLIW processor, several functional units (FUs) are allocated and connected together through one multi-port register file (RF). These FUs are more powerful in terms of functionality and speed compared with those in the reconfigurable matrix. For example, they can execute operations such as branch operations which are not supported in the reconfigurable matrix. Some of these FUs are connected to the memory hierarchy, depending on available ports. Thus the data access to the memory is done through the load/store operation available on those FUs.

For the reconfigurable matrix part, it shares the FUs and RF of the VLIW processor. The access to the memory is performed through the VLIW processor's FUs. Since the reconfigurable matrix will usually process a lot of data concurrently, it may need a big memory to store the data. It may use the multi-port RF for this purpose, but the cost of the RF will grow much faster than its capacity. Alternatively, it may access the external memory through the VLIW processor's FUs. In this case, however, the long latency will degrade the overall system performance. Apart from the FUs and RF shared with the VLIW processor, there are a number of reconfigurable cells (RC) which also contain their own FUs and RFs as shown in Fig. 6. The FUs can be heterogeneous (different RCs in the matrix can have different types of FUs) supporting different operation sets. To remove the control flow inside loops, the FUs support predicated operations.

Other components in the RCs are common to most CGRAs. For example, the multiplexors are used to direct data from different sources. The configuration RAM stores a few configurations locally, which can be loaded on a cycle-by-cycle basis to control the behavior of the basic components. The configurations can also be loaded from the external memory at the cost of extra delay, if the local configuration RAM is not big enough.

FloRA [18] (Floating-point Reconfigurable Array) has a 2D homogeneous array of PEs and a RISC processor connected by a bus as shown in **Fig. 7**. One salient feature of this CGRA is that the PEs in the array can be paired to compute

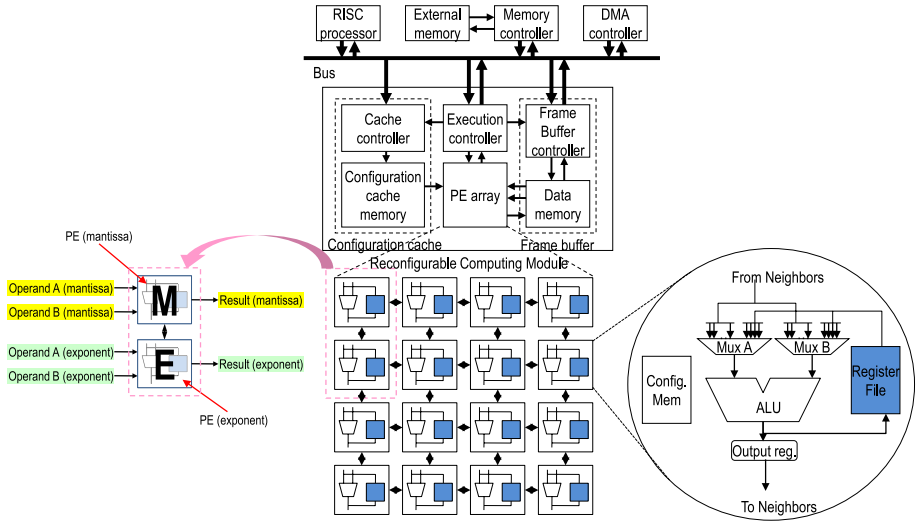

Fig. 7 FloRA architecture [18].

floating-point operations. As floating-point applications such as 3D graphics become more prevalent, acceleration of floating-point operations become more important. More recent version [26] of FloRA has memory-centric communication between the RISC processor and the PE array instead of bus-centric communication as shown in **Fig. 8**. This helps avoid the bottleneck problem caused by the communication through a bus. The cenral memory is divided into several smaller memory blocks, with each part connected to its own master (processor, PE array, network/bus interface are masters). Communication can be done by switching the connections between the masters and the memory blocks. Each memory block can be implemented as a single port memory and therefore costly multi-port memory implementation is not necessary. However, there is some overhead in implementing the communication switch between the masters and memory blocks with a crossbar.

DRP [19] (Dynamically Reconfigurable Processor) from NEC is one of the commertially used CGRAs. It consists of an array of Tiles, each of which is an array of PEs. **Figure 9** shows the architecture of DRP-1 containing 4×2 Tiles, with

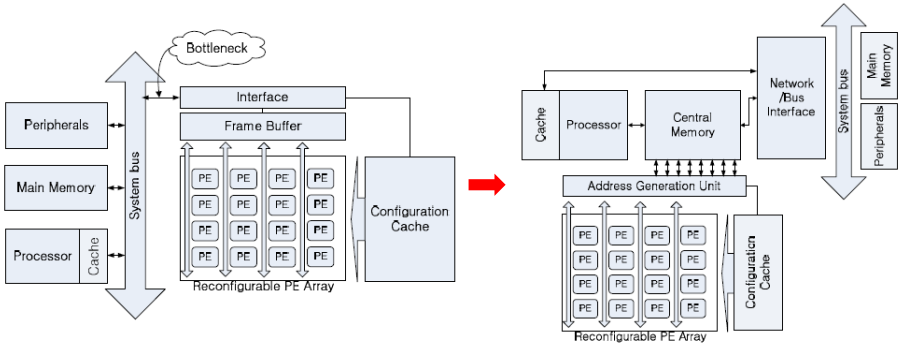

Fig. 8 Memory-centric communication [26].

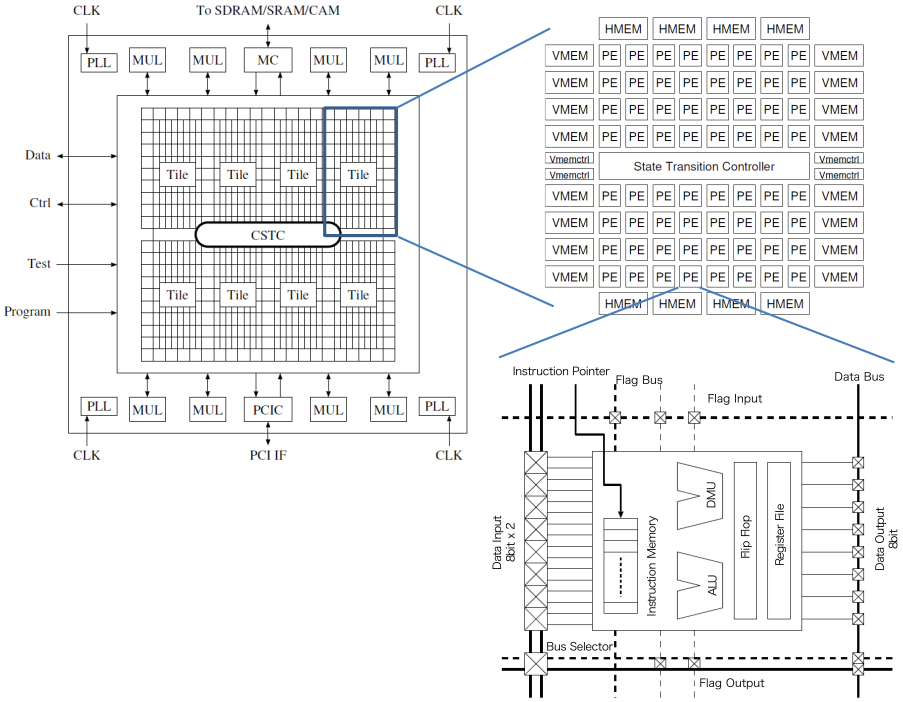

Fig. 9 DRP-1 arcchitecture [19].

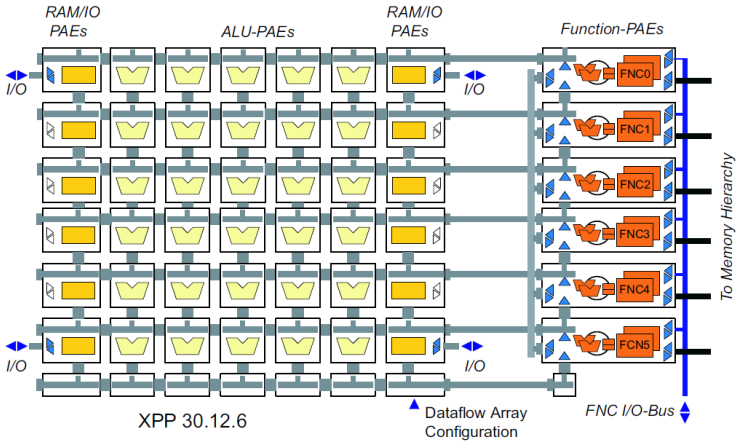

Fig. 10 XPP-III Core architecture [21].

each Tile consisting of an 8×8 array of PEs. Each Tile also has a State Transition Controller (STC), Vertical MEMories (VMEMs), and Horizontal MEMories (HMEMs). Each PE has an 8-bit ALU, an 8-bit DMU (Data Manipulation Unit: for shift/mask operations), an 8-bit×16-word register file, and an 8-bit flip-flop. It also has a 16-depth instruction memory, whose instruction pointer is set by the STC in the Tile. According to the evaluation with stream applications presented in[20], DRP with eight Tiles running at 33 MHz shows five times higher performance than Pentium IV running at 2.5 GHz for Viterbi decoder and six times higher performance than MIPS64 running at 500 MHz for Block Cipher RC6.

XPP [21] (eXtreme Processing Platform) from PACT is another example of commercial CGRA. **Figure 10** shows the architecture of an XPP-III Core. It contains a rectangular array of three types of PAEs (Processing Array Elements). Those in the center of the array are ALU-PAEs. To the left and right side of the ALU-PAEs are RAM-PAEs with I/O. On the right side of the array, there is a column of FNC-PAEs (Function-PAEs). The ALU-PAEs and RAM-PAEs comprise a dataflow array, which can process dataflow graphs efficiently. An FNC-PAE contains a complete VLIW-like sequential processor kernel and is suitable for processing control-oriented part of the application. The horizontal routing

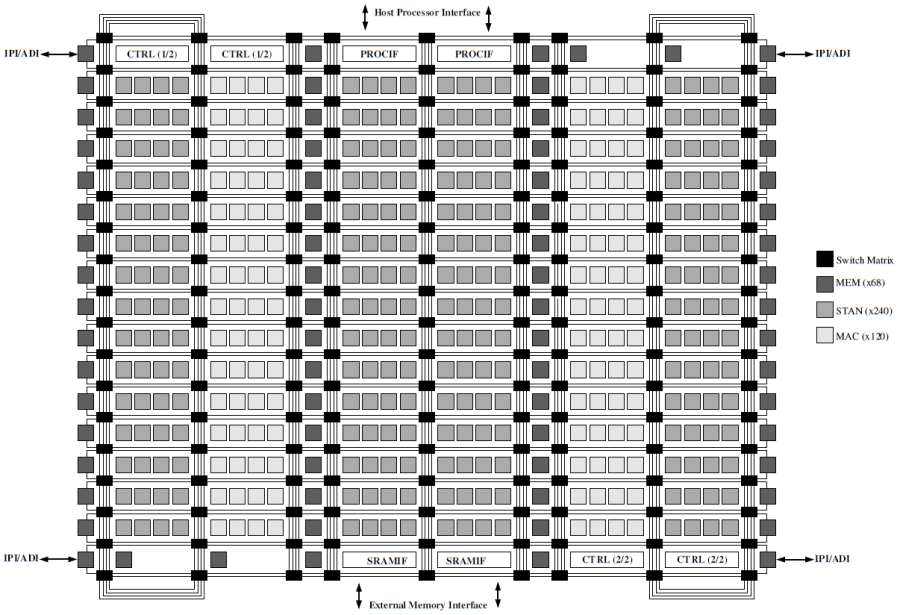

Fig. 11 PC101 picoArray architecture [22].

buses between XPP objects (ALUs, RAMs, I/O objects, etc.) can be segmented by configurable switch objects. The buses include n-bit data buses and 1-bit events buses. The XPP Core data bitwidth (for ALUs, RAMs, and data buses) can be chosen from 16, 24 or 32 bit. Vertical routing connections are provided within the ALU- and RAM-PAEs. Between the FNC-PAEs, there is an additional dedicated vertical routing connection. When the dataflow array is used, a FNC-PAE instructs a DMA controller to configure the ALU- and RAM-PAEs as well as the communication network from external memory. Each configuration word contains the address of the PAE and XPP object to be configured and the configuration value (ALU operator, bus connection etc.) and enters the dataflow array through its configuration interface. The I/O objects allow to cascade XPP Cores and to access external streaming data sources or destinations or external RAM.

The picoArray [22] from picoChip is yet another example of commercial CGRA.

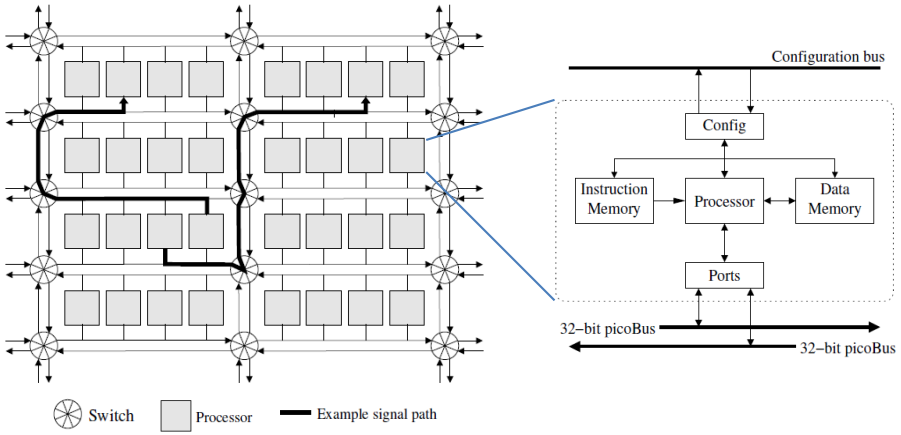

Fig. 12 Interconnect and processor architecture of picoArray [22].

Figure 11 shows the architecture of PC101, the initial version of the picoArray, which contains 430 heterogeneous processors. The processors are connected together using 32-bit unidirectional buses called picoBus and bus switches as shown in **Fig. 12**. The inter-processor communication protocol implemented by the picoBus is based on a time division multiplexing (TDM) scheme. There is no run-time bus arbitration, so communication bandwidth is guaranteed as scheduled by software and controlled using the bus switches. Signals may be point-to-point, or point-to-multi-points. Faster signals can be allocated time-slots more frequently than slower signals. All of the processors in the picoArray are 16-bit, 3-way VLIW processors. The basic structure of the processors is also shown in Fig. 12. Each processor has its own small memory (between 1 KB and 32 KB), which is organised as separate data and instruction banks (i.e., a Harvard architecture). The processor contains a number of communication ports, which allow access to the interconnect buses through which it can communicate with other processors. Each processor is programmed and initialised using a special configuration bus.

3. Architecture Optimization

With the development of various architectures, it has been an important issue to optimize the architectures for given applications or application domains. There have been quite a few attempts to solve the problem through an exploration of architectural parameters such as number of PEs, number of interconnects, number of registers, etc. [27)–30)]

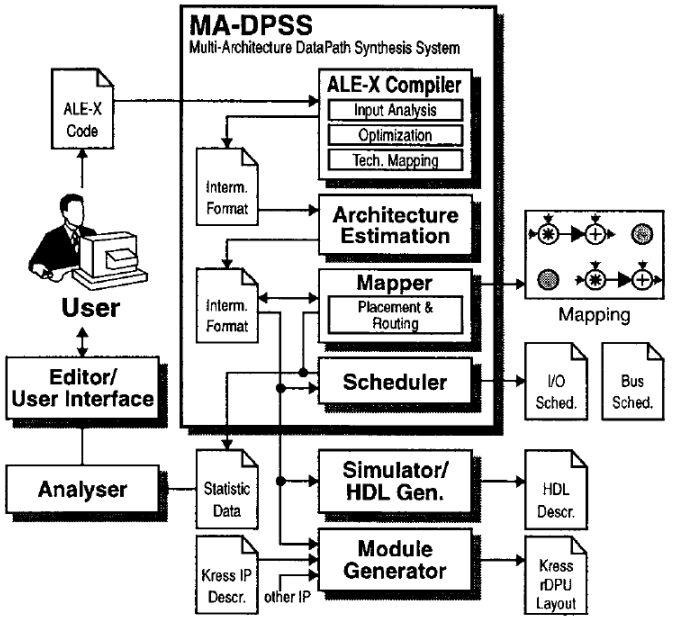

Fig. 13 Design flow of KressArray Xplorer [27].

KressArray Xplorer [27] is an interactive design space exploration (DSE) environment that assists the user in optimizing the architecture of KressArray for a given application domain, considering KressArray as an architecture template rather than a single architecture instance. **Figure 13** shows the overall design flow of KressArray Xplorer. It provides a language called ALE-X to descrive the application to be mapped onto a KressArray. Given the ALE-X description, it generates an expression tree and measures the complexity of the tree to estimate the minimal requirements for the architecture and then determines the initial architecture for the exploration process. It also provides a mapper and scheduler, where the mapper performs simulated anealing based placement and routing of

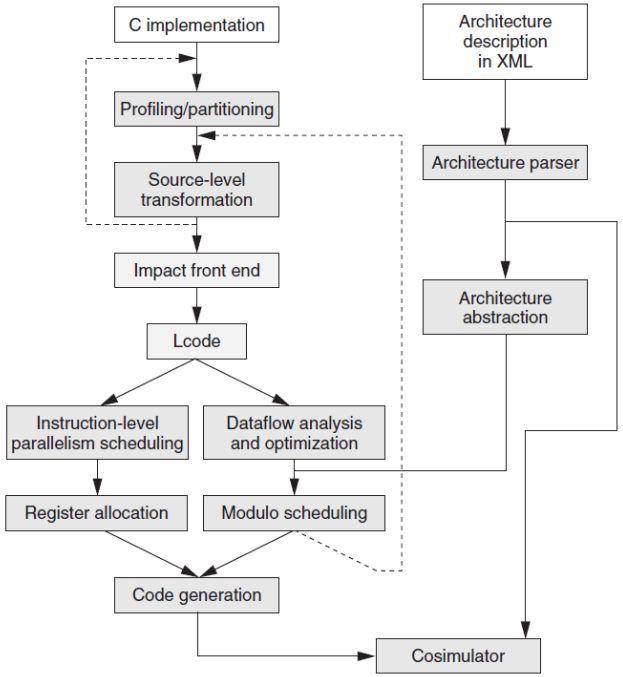

Fig. 14 DRESC framework [28].

the operations in the application onto the KressArray, and the scheduler produces schedules for both global bus and row/column buses. Then the results are analyzed so that the user can redefine architectural parameters, modify the mapping results, and/or tune the optimization parameters (e.g., parameters for the simulated annealing). The atchitectural parameters include array size, functionality of certain PEs, number and maximum legnth of nearest neighbor connections, number of buses, etc.

ADRES is also regarded as an architecture template rather than a fixed architecture instance. Even the internal organization of each PE is not fixed, but FUs and RFs can be put together in several ways. For example, two FUs can share one RF. **Figure 14** shows the DRESC (Dynamically Reconfigurable Embedded System Compiler) framework that has been suggested in Refs. 28) and 31) to explore the architecture design space and generate a good instance of ADRES. The design flow starts from a C-language description of the application. The profiling/partitioning step identifies the candidate loops for mapping on the reconfigurable matrix based on the execution time and possible speedup. The flow uses IMPACT [32], a compiler framework mainly for VLIW, to parse the C code, do some analysis and optimization, and emit an intermediate representation called Lcode, which is used as the input for scheduling. On the other side in the flow, the target architecture is described in an XML-based language to specify the overall topology, supported operation set, resource allocation, and timing. The specified architecture is translated to an internal representation to facilitate modulo scheduling of the operations in the application kernels on the reconfigurable matrix. For the non-kernel code, ILP scheduling is used to generate code for the VLIW processor.

The DRESC tool chain can be used for architecture exploration. However, deriving an optimal instance from the architecture template requires the determination of many architectural parameters such as the numbers of FUs and RFs, the interconnection topology, the operation set each FU supports, and the sizes of the distributed RFs. Moreover, the trade-offs involved in choosing the optimal architectural parameters are not obvious. In Ref. 28), they just try to explore the effects of varying important architectural aspects including interconnection patterns, heterogeneous FUs, memory ports, and distributed RFs to obtain a pointer to a better architecture design and a better insight into the effect of different parameter choices and their interaction.

The DSE flow presented in Ref. 29) considers resource sharing and pipelining. It assumes that an area critical resource is not directly contained in each PE of the architecture but is shared among a set of PEs. **Figure 15** (a) shows an example where two multipliers are shared among four PEs in each row of the PE array. The area critical resources can be pipelined not to slow down the clock and at the same time to enhance their utilization. Since there are various ways of allocating and placing the shared resources (see Fig. 15 (b)), DSE is needed to obtain an optimal or near optimal architecture. As shown in **Fig. 16**, the DSE flow starts from a set of applications in the target domain. The applications are profiled to extract critical loops and the base architecture is determined through

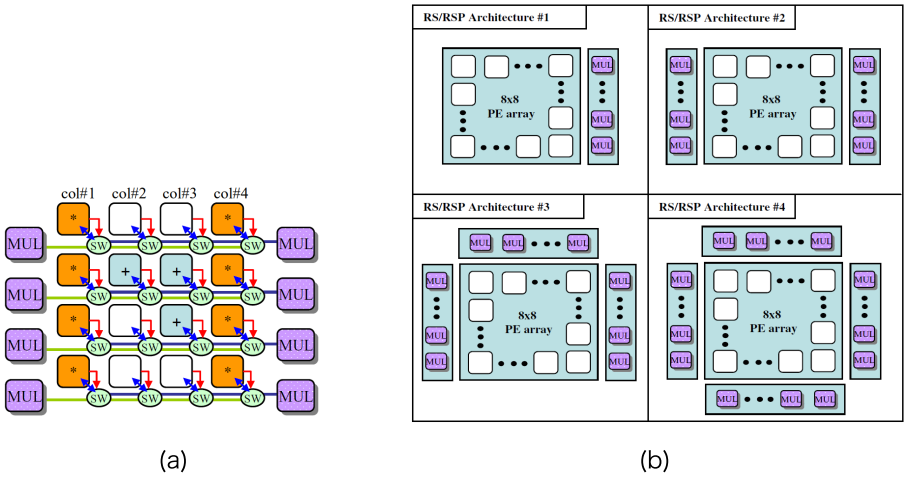

Fig. 15 Critical resource sharing (a) resources shared among PEs in the same row (b) four different ways of allocating and placing shared resources [29].

the first phase of DSE. The parameters of the base architecture determined in this phase include the numbers of rows and columns and the functionalities of each PE. Also determined are the interconnect resources, typically by adding more vertical and/or horizontal interconnects to a basic mesh structure. Then the loops are mapped on the base architecture and the RSP (Resource Sharing and Pipelining) parameters are determined through the second phase of DSE. The RSP parameters include the types of shared functional resources, the types of pipelined resources, the number of pipeline stages of the pipelined resources, the number of rows and columns of the shared resources. During the second phase of DSE the performance is estimated roughly by inserting stall cycles due to lack of resources or multi-cycle operations.

There have been researches on interconnect optimization in terms of performance or energy consumption. The approach in Ref. 33) explores the network topology to study the effects of three aspects of network topology exploration on the performance: (a) changing the interconnection between PEs, (b) changing the way the network topology is traversed while mapping operations to the PEs, and (c) changing the communication delays on the interconnects between

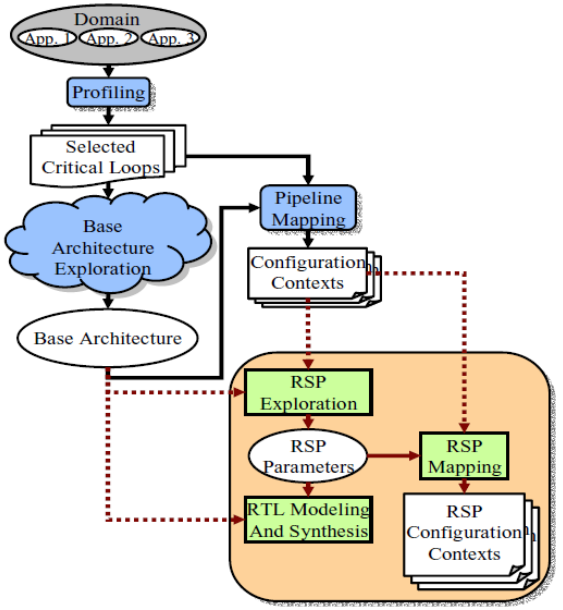

Fig. 16 Architecture exploration considering resource sharing and pipelining [29].

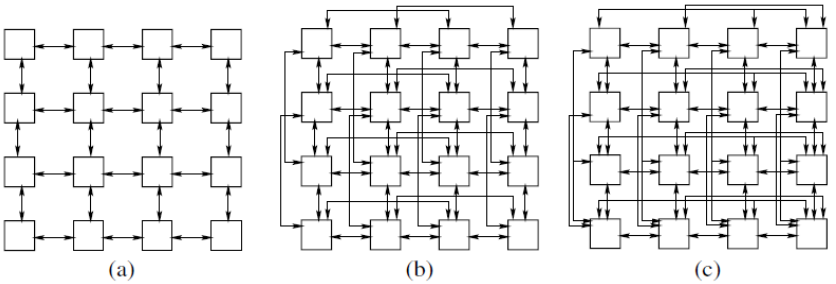

Fig. 17 Different connection topologies [33].

PEs. **Figure 17** shows an example of different connection topologies that can be explored by this approach. In Ref. 30), interconnect architecture explorations have been suggested for low energy. Because CGRA has complex interconnection for performance and flexibility, energy consumption due to the interconnection

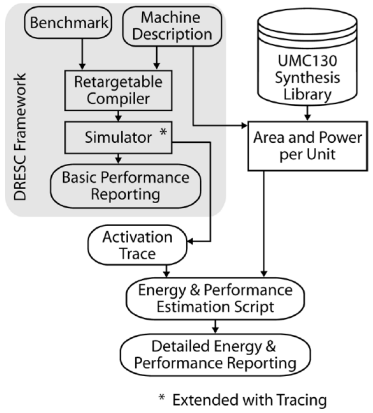

Fig. 18 Energy-aware architecture exploration framework [30].

can be significant. In Ref. 30), the authors present an energy and performance-aware exploration for the interconnect of the ADRES architecture. They use the framework shown in **Fig. 18** for the exploration, where the energy consumption is estimated based on the activation trace generated by the simulator.

Considering that the memory for storing configurations is another source of significant power and area consumption, Ref. 34) proposes a power conscious configuration cache structure. One of the key ideas of this paper is to pipeline the fetched configurations to the next columns instead of fetching each column's configurations directly from the configuration cache. To implement this, the authors add pipeline registers to the PE array such that the configurations fetched to the first column of the array are pipelined to the next columns. This concept works well with loop pipelining, which can be easily implemented by mapping each iteration of the loop on a different column of the PE array through temporal mapping. Temporal mapping is used since all the operations in the loop body are to be executed on a single column. **Figure 19** shows the PE array with pipeline registers and the hybrid structure of spatial cache and temporal cache. When spatial mapping is done, the spatial cache is used. In this case, all PEs can perform different operations like MIMD (Multiple Instruction, Multiple Data). When temporal mapping is done, the temporal cache is used. In this case, all

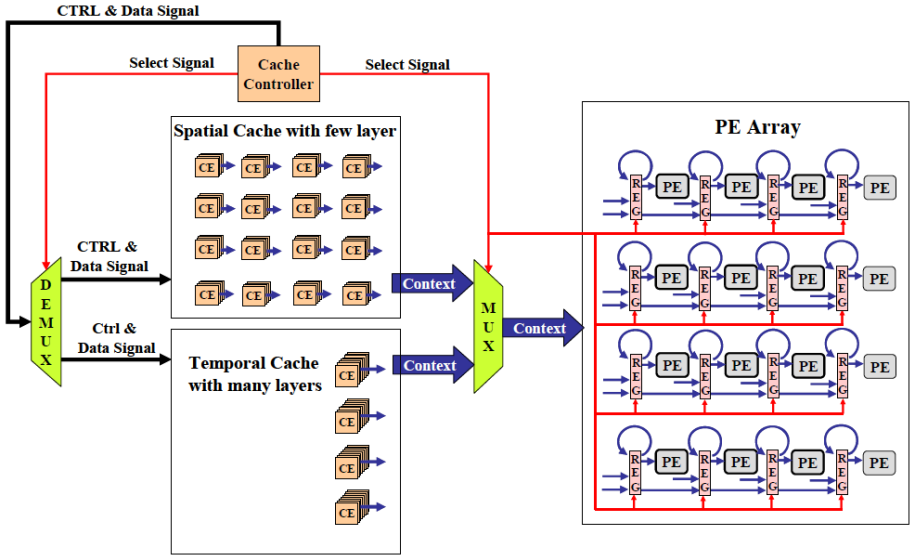

Fig. 19 Hybrid configuration cache structure [34].

PEs in a column can perform different operations but the PEs in a row perform the same operation like SIMD (Single Instruction, Multiple Data), but off by one cycle per iteration. There are also many other approaches to DSE and optimization of CGRAs including the one for register file architecture optimization [35] and the one for DSE of a CGRA for software-defined radio [36].

4. Application Mapping onto CGRAs

For CGRAs to be adopted widely, they should be easy to use. In particular, the application should be easily programmed, compiled, and mapped onto the taget system containing a CGRA. If the programmer should specify explicitly a part of the program that is to be mapped onto the CGRA, then it will be good to have a programming model which is easy to understand and follow so that the programmer's productivity is not degraded much. It will be even better if the programming can be done in a conventional way and the compilation and mapping processes automatically identify compute-intensive kernel parts of the

program and map them onto the CGRA to obtain the best result. With the tools available today, profiling and partitioning are not fully automated but typically done manually with the help of the tools.

Once the kernel parts of the program (typically loops) are identified, they are mapped onto the CGRA. However, the mapping is not an easy process because of the high complexity of the problem that requires compiling the kernel parts for the target architecture of dynamically reconfigurable array of PEs, with additional complexity of dealing with limited routing resources. It is more difficult than simply compiling the kernels for a processor target, since it requires maximally exploiting the parallelism provided by the abundant PEs. Even if we compare it with compiling for a VLIW procesor with multiple functional units, it is still more difficult since there are many more PEs (a PE is more complex than a simple funtional unit) in a CGRA, and the interconnect resources should also be considered at the same time. The mapping of an application kernel onto a CGRA is similar to the synthesis from an HDL (hardware description language) description for an FPGA target. It requires placement of the operations onto a PE array (instead of a gate array) and routing between them. A difference is in the granularity of the operations; CGRA deals with word-level operations and FPGA deals with bit-level operations. A more important difference, however, is that a PE in a CGRA is dynamically shared among many operations whereas a logic block in an FPGA is typically dedicated to its own operation. These all together create new challenges for the mapping of applications onto a CGRA. There have been many researches on developing mapping tools that exploit parallelism found in the applications as well as abundant computation resources of the target CGRAs.

The approach in Ref. 14) for MorphoSys uses GUI-based design tools called *mView* (for manual mapping), *mLoad* (for generating contexts from the mapping result), and *mcc* (for generating TinyRISC instructions to control the PE array execution) to manually generate a mapping (see **Fig. 20**). Such an approach may have difficulty in handling big designs. The approach in Ref. 38) focuses only on instruction-level parallelism, failing to fully utilize the resources in CGRAs, which is possible by exploiting loop-level parallelism. More recent approaches [31),39)–43)] better exploit the parallelism provided by the abundant resources in the target

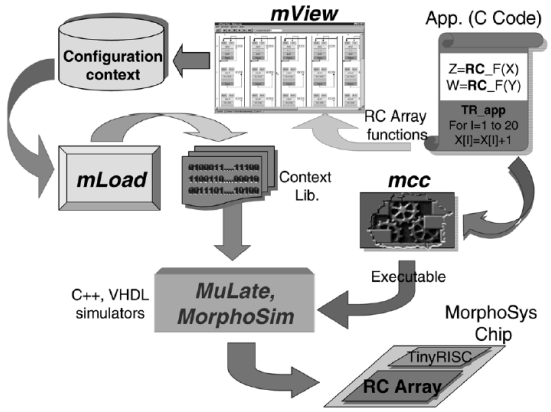

Fig. 20 Mapping and simulation environment for MorphoSys [14)].

CGRA.

The approach in Ref. 31) adopts modulo scheduling to maximally exploit loop-level parallelism. Modulo scheduling is a framework within which algorithms for software pipelining of innermost loops may be defined [44)]. It generates a schedule for one iteration of the given loop such that the same schedule is repeated at regular intervals while satisfying intra- and inter-iteration dependency and resource constraints. The interval, termed initiation interval (II), reflects the performance (throughput) of the scheduled loop. For the scheduling of operations on a CGRA, placement and routing (P&R) problem should also be solved. So the problem is more complex than a simple modulo scheduling problem. **Figure 21** shows an example of mapping a dataflow graph onto a 2×2 array of PEs using modulo scheduling [45)].

The scheduling problem can be formulated as a mapping of a dataflow graph G1=(V1, E1) onto a MRRG (modulo routing resource graph) G2=(V2, E2, II). The dataflow graph models the given loop and the MRRG models the routing resources of the target architecture replicated each cycle along the time axis. Thus each node v in the set of nodes V2 is a tuple (r,t), where r refers to a port of a resource and t refers to the time stamp. The edges in E2 corresponds to possible connections between nodes in V2. The algorithm starts with an II equal

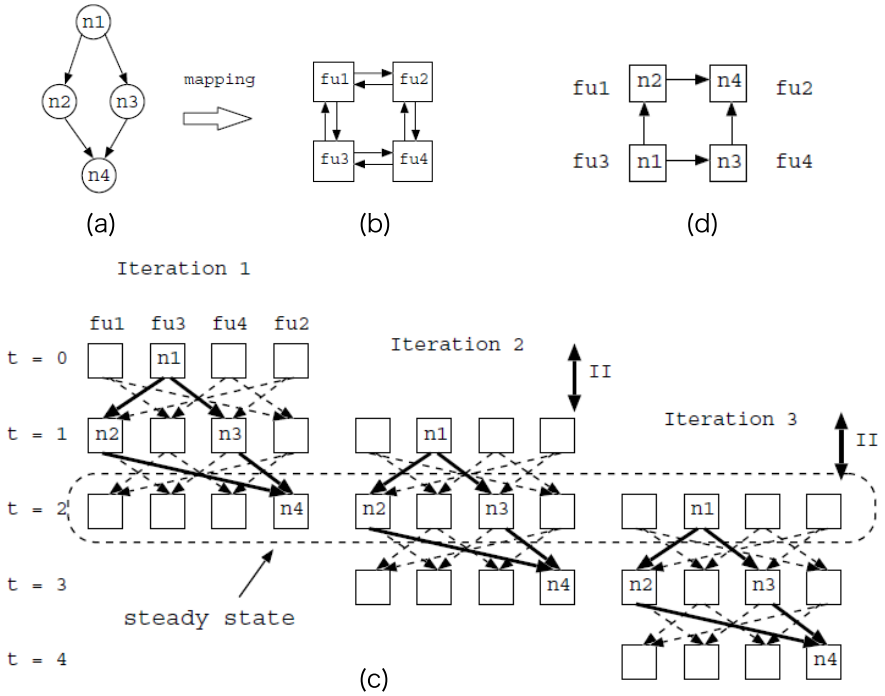

Fig. 21 Modulo scheduling example (a) simple dataflow graph (b) 2×2 array of PEs (c) modulo scheduling and P&R (d) scheduling and P&R result [45].

to MII (minimum initiation interval), and tries successively larger II, until the loop is scheduled. For each II, it first generates an initial schedule satisfying the dependency constraints, but may overuse resources. For example, more than one operation may be scheduled on one FU at the same cycle. In the inner loop, the algorithm iteratively reduces resource overuse and tries to come up with a legal schedule, using simulated annealing. Thus, at every iteration, an operation is placed randomly in a new location. The connected nets are rerouted accordingly. Then a cost function is computed to evaluate the new P&R. If the new cost is lower than the old one, the new P&R of this operation is accepted. Even if the new cost is higher, there is still a chance to accept the move, depending on the *temprature*, which helps to escape from a local minimum. The temperature is

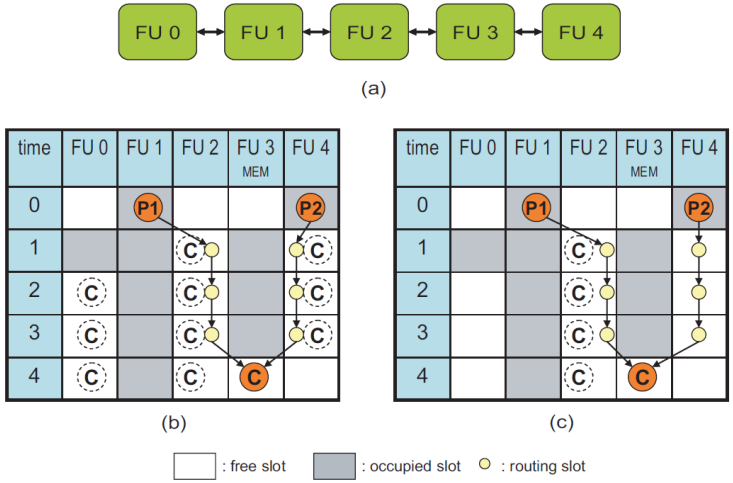

Fig. 22 Comparison of NMS and EMS (a) 1×5 CGRA (b) search by NMS (c) search by EMS [40].

gradually decreased from a high value. So the operation becomes increasingly difficult to move. The cost function is constructed by taking overused resources into account. The penalty associated with them is increased every iteration. If the algorithm runs out of time budget without finding a valid schedule, it starts with the next II. This algorithm is time-consuming.

The above approach for modulo scheduling relies on simulated annealing which can result in a long convergence time for loops with modest numbers of operations. The approach in Ref. 40) speeds up the mapping process through *edge-centric modulo scheduling* (EMS) rather than conventional *node-centric modulo scheduling* (NMS). Thus, instead of assigning operations to PEs before the routing of data communications, this approach defers the placement decision until routing information is discovered. The example in **Fig. 22** shows how it achieves speedup compared to the node-centric approach. Given the hypothetical 1×5 CGRA in Fig. 22 (a), two producer operations P1 and P2 have already been placed and the consumer operation C is going to be placed. NMS may place C by visiting all empty slots as shown in Fig. 22 (b). The slots with dotted circles are failed attempts where the scheduler could not route data from P1 or

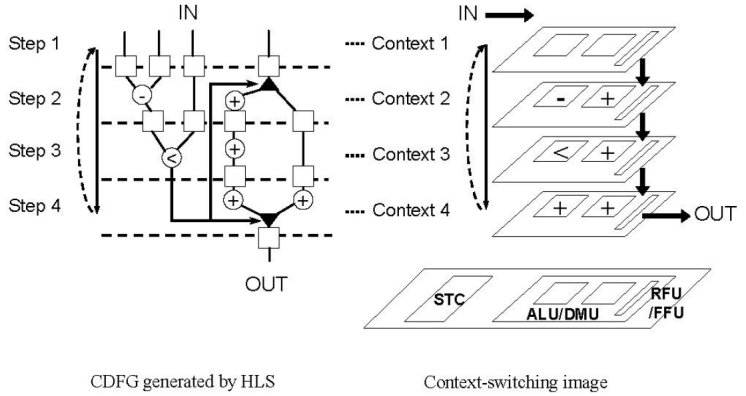

Fig. 23 Basic relationship between control steps and contexts [42].

P2 due to resource conflicts. After visiting those slots, the scheduler successfully places C at slot (3,4), i.e., on PE 3 (FU 3 in the figure) at time 4. On the other hand, EMS begins with the edge from P1 to C, instead of scheduling operation C directly. While routing the edge, if an empty slot is encountered, the scheduler temporarily places the target operation C and checks to see if there are other edges connected to the consumer; if so, it recursively routes those edges. In the example, when the router visits slot (2,1) in Fig. 22 (c), it temporarily places C there and recursively calls the router function to route the edge from P2 to C. When it fails to route the edge, it resumes the routing from slot (2,1). This process is continued until it finds a solution at slot (3,4). So, slots (2,1), (2,2), (2,3), (2,4), and (3,4) are all visited in one routing call, thus making the approach much faster. This approach has a greedy nature and the solution can easily fall into local minima, which is addressed by employing several other techniqes.

The approach in Ref. 39) takes compilation approach to map applications onto parameterizable generic CGRAs. For a better mapping result, it considers the memory bottleneck problem, which typically limits the performance of many applications executed on a CGRA. The mapping flow starts with *microoperation trees* that represent a given loop body. Then it partitions the tree such that the set of microoperations in each partition can be mapped to a PE and executed with a single configuration. Now the problem becomes placing the partitions onto

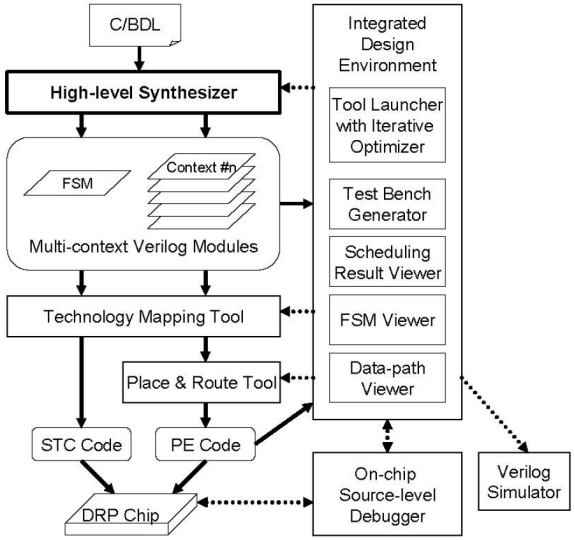

Fig. 24 Overall mapping flow and environment for DRP [42].

the 2D array of PEs. It is solved by two-levels of mapping: *line-level mapping* and *plane-level mapping*. The *line-level mapping* groups together the partitions such that the total number of memory operations in each group does not exceed the capacity of the memory interface bus. Then the *plane-level mapping* stitches together the line placements on the 2D plane of PEs.

The approach in Ref. 42) uses a high-level synthesis (HLS) tool for the mapping of applications in the C language or BDL (behavioral design language: extension of C with some constructs removed) onto the DRP [19] architecture shown in Fig. 9. The HLS tool generates a *multi-context* Verilog code including a finite state machine (FSM) that is mapped onto the STC. As shown in **Fig. 23**, each context basically corresponds to a control step of the HLS result. However, multiple steps can be combined to be mapped into a single context to reduce the number of contexts and achieve better area efficiency. Also PE can be chained and in this case wire delays should be considered. Since accurate wire delays cannot be obtained before P&R, a typical wire delay between PEs are added to each operational delay. **Figure 24** shows the overall mapping flow and environment.

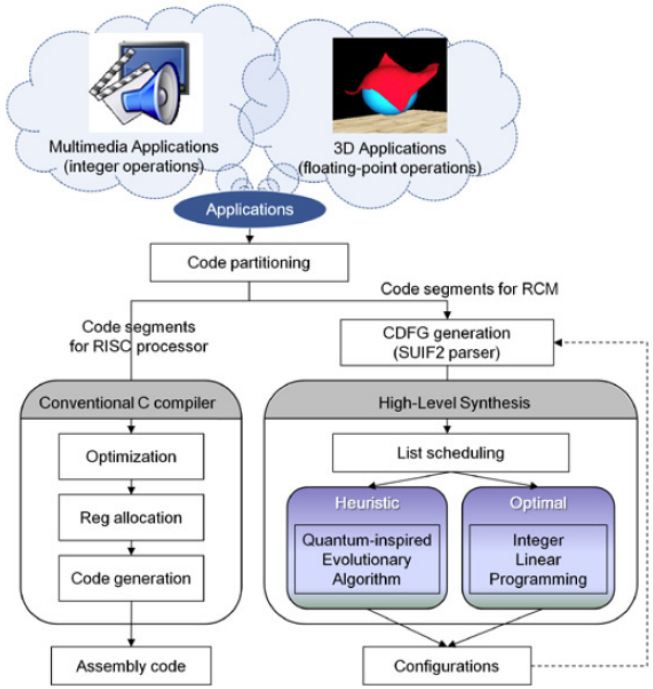

Fig. 25 Overall mapping flow for FloRA [43].

The approach in Ref. 43) also adopts HLS techniques to map applications onto FloRA architecture. It handles loop-level parallelism by applying loop unrolling and pipelining techniques. For the mapping, it first applies the list scheduling algorithm to find an initial solution and then refines the solution using the quantum-inspired evolutionary algorithm (QEA) [46], QEA is an evolutionary algorithm that is known to be very efficient compared to other kinds of evolutionary algorithm. It has been first used for optimizing electronic designs in Ref. 47). The QEA, like genetic algorithms, generates tens or hundreds of possible cases (through rotation instead of crossover operation) and evaluates each case to choose the best solution. The fitness function that we use for the QEA is the performance, which is the inverse of the total latency. The solution is then used to produce the next generation. This iterative improvement continues until there is no improvement during a given time interval or the total latency hits the pre-calculated lower bound of optimal latency. Since the QEA starts with a relatively good initial solution obtained by the list scheduling, it tends to reach a better solution sooner than starting with a random seed. When the schedule and binding of each vertex are determined, it tries to find the routing path among the vertices with unused remaining PEs to see if these schedule and binding results violate the interconnect constraint. It considers a Steiner tree (instead of a spanning tree) for multiple writes from a single source. **Figure 25** shows the overall mapping flow for FloRA. Aside from the QEA-based mapping, it also provides optimal solutions based on integer linear programming (ILP). However, since the ILP-based mapping takes long time, it can be applied only to small examples.

5. Concluding Remarks

For a fixed application, CGRAs cannot beat hardware implementations in terms of performance, power, and area. In terms of flexibility, they can hardly beat genenral purpose processors. However, CGRAs may have the best architecture that can provide hardware-like performance and software-like flexibility at the same time, while maintaining reasonably low power consumption and area. Therefore, for a CGRA, it is important to find its own sweetspot in the target system's design space and optimize it to maximally exploit its merits.

There have been many researches on devising/improving the architectures of CGRAs and their development environments. In particular, the interconnection architecture within the PE array has been an important issue of those researches. ¿From the entire system's perspective, however, devising an efficient interconnection architecture between the PE array and external memory/processor will be a more important reserch direction.

Another future direction is integrating multiple (and maybe smaller) CGRAs in a system, where each of the CGRAs can be optimized such that multiple concurrent tasks can be accelerated at minimal costs in terms of area, power, performance, and flexibility. This again requires researches on efficient interconnection networks among the CGRAs, other processing elements, and memories.

Yet more important issue is developing good programming models and compilers for seamless mapping of application software onto the processor-CGRA

combined systems, so that the architecture does not put too much burden on the application developer. This requires researches on automatically generating code for the interface between the code blocks executed on processors and CGRAs, in addition to the researches on traditional issues of parallel programming. Since it heavily depends on the interface architecture, the researches may have to consider hardware-software codesign.

Acknowledgments This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (no.2010-0019522).

References

1) Dutt, N. and Choi, K.: Configurable Processors for Embedded Computing, *IEEE Computer*, Vol.36, No.1, pp.120–123 (2003).
2) Hauser, J. and Wawrzynek J.: Garp: A MIPS Processor with a Reconfigurable Coprocessor, *FCCM*, pp.12–21 (1997).
3) Yeung, A.K.W. and Rabaey, J.M.: A Reconfigurable Data-driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms, *HICSS-26*, pp.169–178 (1993).
4) Hartenstein R.W. and Kress, R.: A Datapath Synthesis System for the Reconfigurable Datapath Architecture, *ASP-DAC*, pp.479–484 (1995).
5) Hartenstein, R., Herz, M., Hoffmann, T. and Nageldinger, U.: On Reconfigurable Co-Processing Units, *Reconfigurable Architectures Workshop* (1998).
6) Bittner, R.A., Athanas, P.M. and Musgrove, M.D.: Colt: An Experiment in Wormhole Run-time Reconfiguration, *SPIE Photonics East* (1996).
7) Ebeling, C., Cronquist, D. and Franklin, P.: Configurable Computing: The Catalyst for High-Performance Architectures, *IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors*, pp.364–372 (1997).
8) Mirsky, E. and DeHon, A.: MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources, *FCCM*, pp.157–166 (1996).
9) Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. and Agarwal, A.: Baring It All to Software: RAW Machines, *IEEE Computer*, pp.86–93 (1997).
10) Rabaey, J.: Reconfigurable Processing: The Solution to Low Power Programmable DSP, *ICASSP* (1997).
11) Goldstein, S.C., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R.R. and Laufer, R.: PipeRench: A Coprocessor for Streaming Multimedia Acceleration, *ISCA* (1999).
12) Miyamori, T. and Olukotun, K.: REMARC: Reconfigurable Multimedia Array Coprocessor, *FPGA* (1998).
13) Singh, H., Lee, M.-H., Lu, G., Kurdahi, F.J., Bagherzadeh, N. and Lang, T.: MorphoSys: An Integrated Re-configurable Architecture, *NATO RTO Symp. on System Concepts and Integration* (1998).
14) Singh, H., Lee, M.-H., Lu, G., Kurdahi, F.J., Bagherzadeh, N. and Filho, E.M.C.: Morphosys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications, *IEEE Trans. Comput.*, Vol.49, No.5, pp.465–481 (2000).
15) Marshall, A., Stansfield, T., Kostarnov, I., Vuillemin, J. and Hutchings, B.: A Reconfigurable Arithmetic Array for Multimedia Applications, *FPGA* (1999).
16) Alsolaim, A., Becker, J., Glesner, M. and Starzyk, J.: Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems, *FCCM* (2000).
17) Mei, B., Vernalde, S., Verkest, D., Man, H.D. and Lauwereins, R.: ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix, *FPL* (2003).
18) Lee, D., Jo, M., Han, K. and Choi, K.: FloRA: Coarse-Grained Reconfigurable Architecture with Floating-Point Operation Capability, *International Conference on Field-Programmable Technology*, pp.376–379 (2009).
19) Motomura, M.: A Dynamically Reconfigurable Processor Architecture, *Microprocessor Forum* (2002).
20) Suzuki, N., Kurotaki, S., Suzuki, M., Kaneko, N., Yamada, Y., Deguchi, K., Hasegawa, Y. and Amano, H.: Implementing and Evaluating Stream Applications on the Dynamically Reconfigurable Processor, *FCCM* (2004).
21) PACT XPP Technologies: *XPP-III Processor Overview.* http://www.pactxpp.com/
22) Duller, A., Panesar, G. and Towner, D.: Parallel Processing the picoChip way!, *Communicating Processing Architectures*, Broenink, J.F. and Hilderink, G.H. (eds.), IOS Press, pp.299–312 (2003).
23) Hartenstein, R.: A Decade of Reconfigurable Computing: A Visionary Retrospective, in *Design Automation and Test in Europe Conf.*, pp.642–649 (2001).
24) Lee, M.-H., Singh, H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., Filho, E.M.C. and Alves, V.C.: Design and Implementation of the MorphoSys Reconfigurable Computing Processor, *Journal of VLSI Signal Processing*, Vol.24, pp.147–164 (2000).
25) Intel: *Embedded Pentium Processor with MMX Technology Datasheet.* http://www.intel.com/design/archives/Processors/mmx/
26) Chang, K. and Choi, K.: Memory-Centric Communication Architecture for Reconfigurable Computing, *International Symposium on Reconfigurable Computing: Architectures, Tools and Applications*, pp.400–405 (2010).
27) Hartenstein, R., Herz, M., Hoffmann, T. and Nageldinger, U.: KressArray Xplorer:

A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures, *Asia and South Pacific Design Automation Conf.*, pp.163–168 (2000).
28) Mei, B., Vernalde, S., Verkest, D. and Lauwereins, R.: Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study, *Design Automation and Test in Europe Conf.*, pp.1224–1229 (2004).
29) Kim, Y., Kiemb, M., Park, C., Jung, J. and Choi, K.: Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization, *Design Automation and Test in Europe Conf.*, pp.12–17 (2005).
30) Lambrechts, A., Raghavan, P., Jayapala, M., Catthoor, F. and Verkest, D.: Interconnect Exploration for Energy Versus Performance Tradeoffs for Coarse Grained Reconfigurable Architectures, *IEEE Trans. Very Large Scale Integration* (*VLSI*) *Systems*, Vol.17, No.1, pp.151–155 (2009).
31) Mei, B., Vernalde, S., Verkest, D., De Man, H. and Lauwereins, R.: DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures, *International Conference on Field Programmable Technology*, pp.166–173 (2002).
32) The IMPACT Research Group. http://impact.crhc.illinois.edu/
33) Bansal, N., Gupta, S., Dutt, N., Nicolau, A. and Gupta, R.: Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures, *Design Automation and Test in Europe Conf.*, pp.474–479 (2004).
34) Kim, Y., Park, I., Choi, K. and Paek, Y.: Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architec-ture, *International Symposium on Low Power Electronics and Design*, pp.310–315 (2006).
35) Kwok, Z. and Wilton, S.: Register File Architecture Optimization in a Coarse-Grained Reconfigurable Architecture, *Symposium on Field-Programmable Custom Computing Machines* (2005).
36) Novo, D., Bougard, B., Raghavan, P., Schuster, T., Kim, H.-S., Yang, Ho. and Van der Perre, L.: Energy-Performance Exploration of a CGA-Based SDR Processor, *SDR 06 Technical Conference and Product Exposition* (2006).
37) Callahan, T.J. and Wawrzynek, J.: Instruction-Level Parallelism for Reconfigurable Computing, *IWFPL*, pp.248–257 (1998).
38) Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V. and Amarasinghe, S.P.: Space-Time Scheduling of Instruction Level Parallelism on a RAW Machine, *ASPLOSV* (1998).
39) Lee, J.-e., Choi, K. and Dutt, N.: Compilation Approach for Coarse-Grained Reconfigurable Architectures, *IEEE Design & Test of Computers*, Vol.20, No.1, pp.26–33 (2003).
40) Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H. and Kim, H.: Edge-Centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures, *PACT* (2008).
41) Ahn, M., Yoon, J.W., Paek, Y., Kim, Y., Kiemb, M. and Choi, K.: A Spatial Mapping Algorithm for Heterogeneous Coarse-Grained Reconfigurable Architectures, *Design Automation and Test in Europe Conf.*, pp.363–368 (2006).
42) Toi, T., Nakamura, N., Kato, Y., Awashima, T., Wakabayashi, K. and Jing, L.: High-Level Synthesis Challenges and Solutions for a Dynamically Reconfigurable Processor, *ICCAD*, pp.702–708 (2006).
43) Lee, G., Lee, S., Choi, K. and Dutt, N.: Routing-Aware Application Mapping Considering Steiner Point for Coarse-Grained Reconfigurable Architecture, *ARC*, pp.231–243 (2010).
44) Rau, B.R.: Iterative Modulo Scheduling, Technical Report HPL-94-115, Hewlett-Packard Lab (1995).
45) Mei, B., Vernalde, S., Verkest, D., De Man, H. and Lauwereins, R.: Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling, *Design Automation and Test in Europe Conf.* (2003).
46) Han, K.-H. and Kim, J.-H.: Quantum-Inspired Evolutionary Algorithm for a Class of Combinatorial Optimizaion, *IEEE Trans. Evolutionary Computation*, Vol.6, No.6, pp.580–593 (2002).
47) Ahn, Y., Han, K., Lee, G., Song, H., Yoo, J. and Choi, K.: SoCDAL: System-on-Chip Design AcceLerator, *ACM Trans. Design Automation of Electronic Systems*, Vol.13, No.1 (2008).

(Received October 6, 2010)
(Released February 8, 2011)

(Invited by Editor-in-Chief: *Hidetoshi Onodera*)

Kiyoung Choi was born in 1955. He received his B.S. degree in electronics engineering from Seoul National University in 1978, M.S. degree in electrical and electronics engineering from Korea Advanced Institute of Science and Technology in 1980, and Ph.D. degree in electrical engineering from Stanford University in 1989. From 1989 to 1991, he was with Cadence Design Systems, Inc. In 1991, he joined the faculty of the Department of Electrical Engineering and Computer Science, Seoul National University. His primary interests include various aspects of computer-aided electronic systems design including embedded systems design, high-level synthesis, and low-power systems design. He is also interested in computer architecture and especially in configurable and reconfigurable computer architecture design.