Reconfigurable Computing

David Boland[1], Chung-Kuan Cheng[2], Andrew B. Kahng[2], Philip H.W. Leong[1]

[1]School of Electrical and Information Engineering, The University of Sydney, Australia 2006

[2]Dept. of Computer Science and Engineering, University of California, La Jolla, California

Abstract:

Reconfigurable computing is the application of adaptable fabrics to solve computational problems, often taking advantage of the flexibility available to produce problem-specific architectures that achieve high performance because of customization. Reconfigurable computing has been successfully applied to fields as diverse as digital signal processing, cryptography, bioinformatics, logic emulation, CAD tool acceleration, scientific computing, and rapid prototyping.

Although Estrin-first proposed the idea of a reconfigurable system in the form of a fixed plus variable structure computer in 1960 [1] it has only been in recent years that reconfigurable fabrics, in the form of field-programmable gate arrays (FPGAs), have reached sufficient density to make them a compelling implementation platform for high performance applications and embedded systems. In this article, intended for the non-specialist, we describe some of the basic concepts, tools and architectures associated with reconfigurable computing.

Keywords:

reconfigurable computing; adaptable fabrics; application integrated circuits; field programmable gate arrays (FPGAs); system architecture; runtime

1 Introduction

Although reconfigurable fabrics can in principle be constructed from any type of technology, in practice, most contemporary designs are made using commercial field programmable gate arrays (FPGAs). An FPGA is an integrated circuit containing an array of logic gates in which the connections can be configured by downloading a bitstream to its memory. FPGAs can also be embedded in integrated circuits as intellectual property cores. More detailed surveys on reconfigurable computing are available in the literature [2-6].

Microprocessors offer an easy-to-use, powerful and flexible implementation medium for digital systems. Their utility in computing applications makes them an overwhelming first choice. Moreover, it is relatively easy to find software developers, and microprocessors are widely supported by operating systems, software engineering tools, and libraries. However, in the last decade, power constraints have limited the performance of serial computation on microprocessors. This has led to the development of multi-core processors and an increasing importance placed on the pursuit of parallel computation [7]. Unfortunately, multi-core processors are rarely the most efficient method to perform parallel computation. This inefficiency stems from the fact that each core must be general enough to support an entire instruction set. As a result, the majority of energy is used in decoding the instruction and fetching data instead of performing actual computation[8].

Hardware accelerators such as graphics processor units (GPUs) and FPGAs are parallel computational architectures that have demonstrated substantial performance and energy efficiency improvements over traditional multi-core processor designs by moving the focus back to computation [9, 10]. In terms of energy efficiency, the GPU architecture, which consists of thousands of parallel floating-point units, is best suited to so-called embarrassingly parallel computation or computationally expensive problems. However, many algorithms will not fall into this problem category. In contrast, using an FPGA or Application-Specific Integrated Circuit (ASIC), it is possible to create a fully customised datapath for a given algorithm, meaning it is possible to achieve even greater energy efficiency using these devices.

Application-specific integrated circuits (ASICs) and FPGAs achieve greater levels of parallelism than a microprocessor by arranging computations in a spatial rather than temporal fashion. This can result in performance improvements of several orders of magnitude. Also, the absence of caches and instruction decoding can result in the same amount of work being done with less chip area and lower power consumption[11]. Notable examples of application domains include cryptography, NP-hard optimization problems, pattern matching, machine learning, and molecular dynamics [6].

An example involving the implementation of a finite impulse response (FIR) filter is shown in **Fig. 1**. The reconfigurable computing solution is significantly more parallel than the microprocessor-based one. In addition, it should be apparent that the reconfigurable solution avoids the overheads associated with instruction decoding, caching, register files. Furthermore, speculative execution, unnecessary data transfers and control hardware can be omitted.

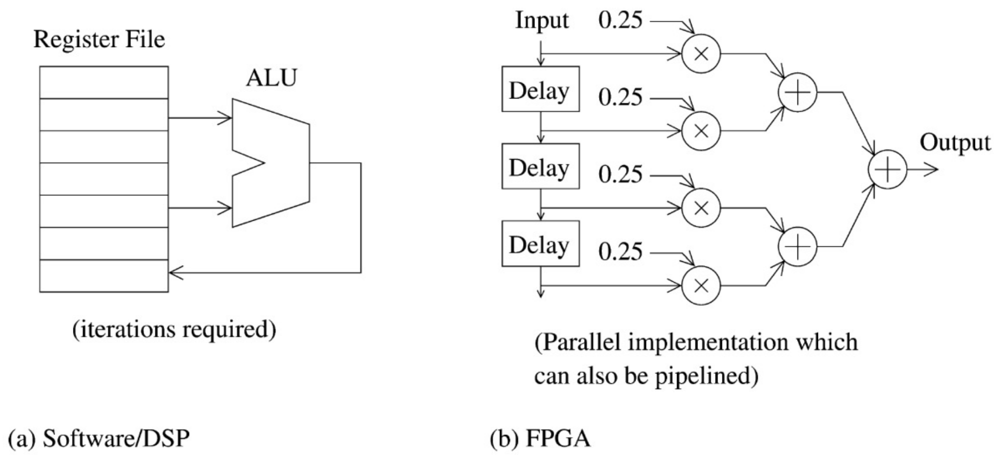


Figure 1. Illustration of a microprocessor based FIR filter vs. a reconfigurable computing solution. In the microprocessor, operations are performed in the ALU sequentially. Furthermore, instruction decoding, caching, speculative execution, control generation and so on are required. For the reconfigurable computing approach using an FPGA, spatial composition is used to increase the degree of parallelism. The FPGA implementation can be further parallelized through pipelining.

Compared with ASICs, FPGAs offer very low non-recurrent engineering (NRE) costs, which is often a more important factor than the fact that FPGAs have higher units costs. This is because many applications do not have the extremely high volumes required to make ASICs a cheaper proposition. As integrated circuit feature sizes continue to decrease, the NRE costs associated with ASICs continue to escalate, increasing the volume at which it becomes cheaper to use an ASIC (see **Fig. 2**). Being more specialized, ASICs offer area, power and speed advantages over FPGAs, this gap being reduced as more hard blocks are employed [12]. Moving forward, reconfigurable computing will be used in increasingly more applications, as ASICs become only cost effective for the highest performance or highest volume applications.

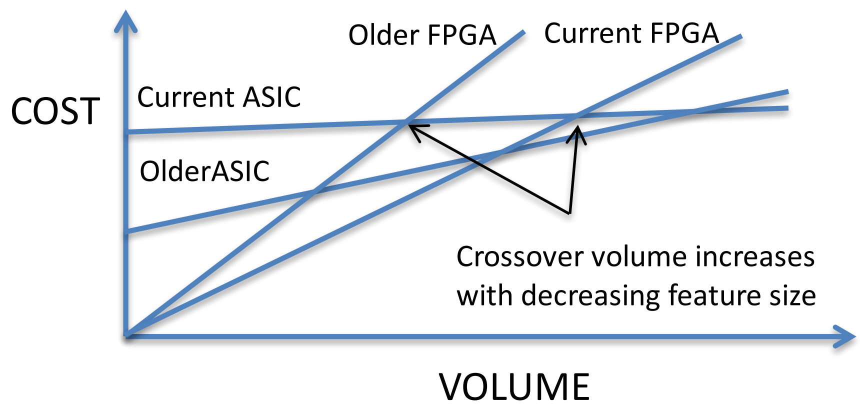


Figure 2. Cost of technology vs. volume. The crossover volume for which ASIC technology is cheaper than FPGAs increases as feature size is reduced because of increased non-recurrent engineering costs.

Additional benefits of reconfigurable computing are that its technology provides a shorter time to market than ASICs (associated FPGA fabrication time is essentially zero), making many fabrication iterations within a single day possible. This benefit allows more complex algorithms to be deployed and makes problem-specific customizations of designs possible. FPGA-based designs are inherently less risky in terms of technical feasibility and cost, as shorter design times and lower upfront costs are involved. As its name suggests, FPGAs also offer the possibility of modifications to the design in the field, which can be used to provide bug fixes, modifications to adapt to changing standards, or to add functionality, all of which can be achieved by downloading a new bitstream to an existing reconfigurable computing platform. Reconfiguration can even take place while the system is running, this being known as runtime reconfiguration (e.g., [13]).

In the next section, we introduce the basic architecture of common reconfigurable fabrics, followed by a discussion of applications of reconfigurable computing and system architectures. Runtime reconfiguration and design methods are then covered. Finally, we discuss multichip systems and end with a conclusion.

2 Reconfigurable Fabrics

A block diagram illustrating a generic fine-grained island-style FPGA is given in **Fig. 3** [14]. Products from companies such as Xilinx [15], Altera [16], and Microsemi [17] are commercial examples. The FPGA consists of a number of logic cells that can be interconnected to other logic and input/output (I/O) cells via programmable routing resources. Logic cells and routing resources are configured via bit-level programming data, which is stored in memory cells in the FPGA. A logic cell consists of user-programmable combinatorial elements, with an optional register at the output. They are often implemented as lookup tables (LUTs) with a small number of inputs, 4-input LUTs being shown in **Fig.3**. Using such an architecture, subject to FPGA-imposed limitations on the circuit's speed and density, an arbitrary circuit can be implemented. The complete design is described via the configuration bitstream which specifies the logic and I/O cell functionality, and their interconnection.

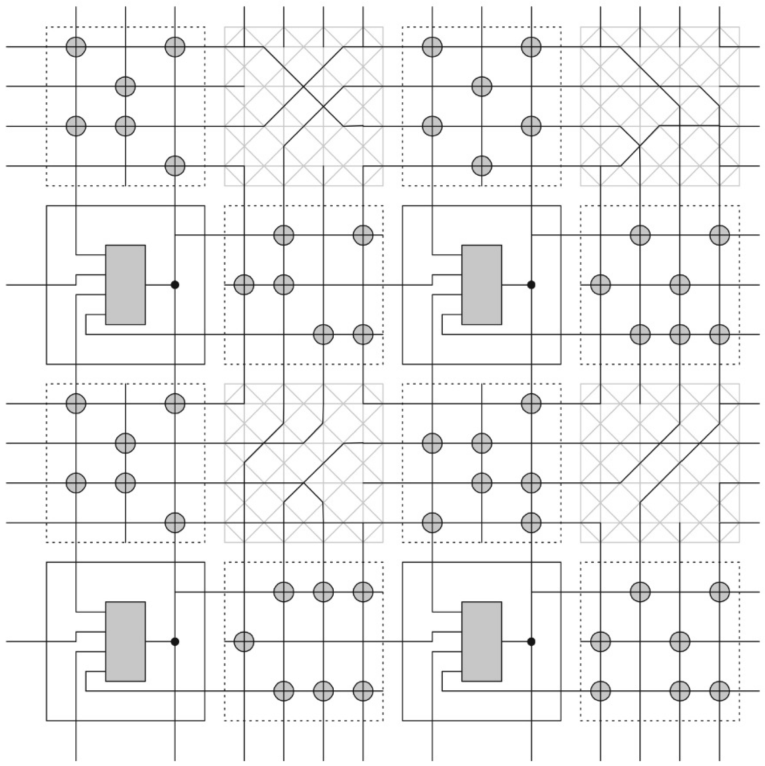


Figure 3. Architecture of a basic island-style FPGA with four-input logic cells. The logic cells, shown as gray rectangles are connected to programmable routing resources (shown as wires, dots, and diagonal switch boxes) (source: Reference [14] and [18]).

Current trends are to incorporate additional embedded blocks so that designers can integrate entire systems on a single FPGA device. Apart from density, cost, and board area benefits, this process also improves performance because more specialized logic and routing can be used and all components are on the same chip. A contemporary FPGA commonly has features such as carry chains to enable fast addition; wide decoders; tristate buffers; blocks of on-chip memory and multipliers; embedded microprocessors; programmable I/O standards in the input/output cells; delay locked loops; phase locked loops for clock de-skewing, phase shifting and multiplication; multi-gigabit transceivers (MGTs); and embedded microprocessors. Embedded microprocessors can be implemented either as soft cores using the internal FPGA resources or as hardwired cores.

In addition to the architectural features described, intellectual property (IP) cores, implemented using the logic cell resources of the FPGA, are available from vendors and can be incorporated into a design. These cores include bus interfaces, networking components, memory interfaces, signal processing functions, microprocessors and so on and can significantly reduce development time and effort.

The bit-level organization of the logic and routing resources in island-style FPGAs is extremely flexible but has high implementation overhead as a result. Tradeoffs exist in the granularity of the logic cells and routing resources. Fine-grained devices have the best flexibility; however, coarse-grained elements can trade some flexibility for higher performance and density [19].

With modern technologies, the speed of the routing resource is a limiting factor. Trends have been to increase the functionality of the logic cells e.g., use logic cells with larger numbers of inputs which can also be configured as smaller LUTs [20] and to add pipeline registers to the routing fabric [21]. For datapath oriented applications such as in digital signal processing, coarse-grained architectures [22] such as Pipewrench [23] and RaPID [24] employ bus-based routing and word-based functional units to utilize silicon resources more efficiently.

3 Applications

Reconfigurable computing has found widespread application in the form of "custom computing-machines" to accelerate computation over algorithms implemented on a CPU. Application domains include high-energy physics [25], genome analysis [26], signal processing [27, 28], computer vision [29], cryptography [30, 31], financial engineering [10, 32], scientific computing [33], machine learning [34] and security [35].

In many of these problem domains, a general purpose GPU has also demonstrated considerable acceleration over a CPU and will often outperform an FPGA as well in terms of raw performance. This is because it is a parallel architecture with many hardened floating-point units and substantially greater memory bandwidth, meaning it is an ideal architecture provided algorithms that can be broken down into a large number of parallel threads. However, outright performance is no longer the only benchmark; energy consumption is also important. In terms of high performance computing, supercomputing clusters and datacenters now consume vast amounts of energy, not only on computation, but also on cooling in order to maintain performance and reliability. It follows that reducing energy consumption provides both environmental and economic benefits. Energy minimization is also important for embedded applications; for example, reducing power consumption on smartphones or other battery-powered devices is desirable from an end user perspective. As a result, FPGA and GPU vendors are focusing their engineering efforts towards making future architectures more energy efficient. This is reflected in the most recent FPGA and GPU architectures: Nvidia's P100 claims a peak performance of 10.6 TFLOPs (single precision) with a TDP of only 300W [36], while Altera claims performance of up to 9.3 TFLOPs (single precision) at 80 GFLOPs/watt is achievable on their upcoming Stratix 10 device [16].

Many experimental studies have been performed comparing the energy efficiency of FPGAs, GPUs and CPUs; a recent survey is provided [37]. Both FPGAs and GPUs typically outperform CPUs according to this metric. GPUs have been shown to be more energy efficient than FPGAs for certain applications such as matrix multiplication. To some extent, this is a result of the decision to optimize the GPU architecture for this problem [38]. However, the flexibility of an FPGA has seen it outperform a GPU, in terms of energy-efficiency or performance-per-watt, across a broader spectrum of applications. Examples include: 2-D FIR (finite-impulse response) filters, Viola-Jones face detection, K-means clustering, Monte-Carlo options pricing, random number generation, Smith-Waterman, 3-D ultrasound computer tomography [37]. Energy efficiency gains using FPGAs have also been claimed on commercial systems. For example, Microsoft reported a 3x energy efficiency gain, and a reduced latency, when using FPGAs instead of GPUs on their Catapult machine [39], which is discussed later in Section 4.

While most of these performance comparisons have been performed using IEEE standard single or double precision arithmetic, this is not necessarily the most energy-efficient design possible on an FPGA. This is because FPGAs have the freedom to implement any precision, so it may be possible to create a working design using a custom (reduced precision) fixed or floating-point number format that is sufficient to satisfy a design specification. This will avoid unnecessary computation and can improve the energy-efficiency and performance of an FPGA implementation dramatically [40, 41].

To a degree, the flexibility of an FPGA is even beyond that possible in an ASIC. For example, in an FPGA-based implementation of RSA cryptography [30], a different hardware modular multiplier for each prime modulus was employed (i.e., the modulus was hardwired in the logic equations of the design). Such an approach would not be practical in an ASIC as the design effort and cost is too high to develop a different chip for different moduli. This led to greatly reduced hardware and improved performance, the implementation being an order of magnitude faster than any reported implementation in any technology at the time.

Another important application is logic emulation [42, 43] where reconfigurable computing is not only used for simulation acceleration, but also for prototyping of ASICs and in-circuit emulation. In-circuit emulation allows the possibility of testing prototypes at full or near-full speed, allowing more thorough testing of time-dependent applications such as networks. It also removes many of the dependencies between ASIC and firmware development, allowing them to proceed in parallel and hence shortening development time. As an example, it was used in [44] for the development of a two-million-gate ASIC containing an IEEE 802.11 medium access controller and IEEE 802.1 la/b/g physical layer processor. Using a reconfigurable prototype of the ASIC on a commodity FPGA board, the ASIC went through one complete pass of real-time beta testing before tape-out.

Digital logic, of course, maps extremely well to fine-grained FPGA devices. The main design issues for such systems lie in partitioning of a design among multiple FPGAs and dealing with the interconnect bottleneck between chips. The Cadence Protium Rapid Prototyping Platform [45] is a commercial example of a logic emulation system and has 100 million-gate logic capacity and fast compilation and partitioning algorithms. Further discussion of interconnect time-multiplexing and system decomposition is given later in this article. Some examples of applications accelerated using early multiple FPGA systems are discussed below.

Hoang [26] implemented algorithms to find minimum edit distances for protein and DNA sequences on the Splash 2 architecture. Splash 2 can be modeled in terms of both bidirectional and unidirectional systolic arrays. In the bidirectional algorithm, the source character stream is fed to the leftmost processing element (PE), whereas the target stream is fed to the rightmost PE. Comparing two sequences of length m and n requires at least $2 \times \max(m+1, n+1)$ processors, and the number of steps required to compute the edit distance is proportional to the size of the array. The unidirectional algorithm is suited for comparing a single source sequence against multiple target sequences. The source sequence is first loaded as in the bidirectional case, and the target sequences are fed in one after the other and processed as they pass through the PEs (which results in virtually 100% utilization of processors, so that the unidirectional model is better suited for large database searches).

A common application domain for reconfigurable computing is in real-time data acquisition and signal processing. The BEE2 system [27], described in the next section, was applied to the radio astronomy signal processing domain, which included development of a billion-channel spectrometer, a 1024-channel polyphase filter bank, and a two-input, 1024-channel correlator. The FPGA-based system used a 130-nm technology FPGA and performance was compared with 130-and 90-nm DSP chips as well as a 90-nm microprocessor. Performance in terms of computational throughput per chip was found to be a factor of 10 to 34 over the DSP chip in 130-nm technology and 4 to 13 times better than the microprocessor. In terms of power efficiency, the FPGA was one order of magnitude better than the DSP and two orders of magnitude better than the microprocessor. Compute throughput per unit chip cost was 20–307% better than the 90-nm DSP and 50–500% better than the microprocessor.

One final emerging application domain is machine learning. Reconfigurable implementations show great promise for addressing their heavy computational demands, and reconfigurable computing is particularly strong in embedded and low-precision scenarios. Tridgell et. al. demonstrated regression, classification and novelty detection using online kernel methods. Their fully pipelined implementation could process continuous data at rates higher than 1 Gbps and perform simultaneous learning and prediction with a latency of 100 ns [46]. Zhang et. al. applied a roofline model to balance resource utilization and memory bandwidth in the acceleration of a deep convolutional neural network (CNN). They achieved 62 GFLOPS on a single Xilinx Virtex VC707 board, this being a 4.8X speedup over a 16 thread implementation on an Intel Xeon E5-2430 processor [47].

4 System Architectures

Reconfigurable computing machines are constructed by utilizing one or more FPGAs. Most systems include other elements, such as microprocessors and storage, and can be treated as processing elements and memory that are interconnected. Obviously, the arrangement of these elements affects the system performance and routability, and some examples are given in this section.

The Avnet Zedboard is a development board which integrates a single Xilinx Zynq XC7Z020 FPGA (which contains FPGA logic and a dual-core ARM Cortex-A9 processor), DDR memory, SD card, Ethernet, USB and video interfaces. This single board computer can run the Linux operating system, and it provides a low-cost entry point for teaching and research in reconfigurable computing [48].

The simplest topology for connecting multiple FPGAs involves a ring, mesh, or other fixed pattern. FPGAs serve as both logic and interconnect, providing direct communication between adjacent devices. Such an architecture is predicated on locality in the circuit design and further assumes that the circuit design maps well to the planar mesh. This architecture fits well for applications with regular local communications [49]. However, in general, high performance is hard to obtain for arbitrary communication patterns because the architecture only provides direct communications between neighboring FPGAs and two distant FPGAs may need many other devices as “hops” to communicate, resulting in long and widely variable delays. Furthermore, FPGAs, when used as interconnects, often result in poor timing characteristics.

Figure 4 depicts the SPLASH 2 architecture [50] published in 1990. Each board contains 16 FPGAs, X1 through X16. The blocks M1 through M16 are local memories of the FPGAs. A simplified 36-bit bus crossbar, with no permutation of the bit-lines within each bus, interconnects the 16 FPGAs. Another 36-bit bus connects the FPGAs in a linear systolic fashion. The local memories are dual ported with one port connecting to the FPGAs and the other port connecting to the external bus. It is interesting to note that the crossbar was added to the SPLASH 2 machine, the original SPLASH 1 machine only having the linear connections. SPLASH 2 has been successfully used for custom computing applications such as search in genetic databases and string matching [26].

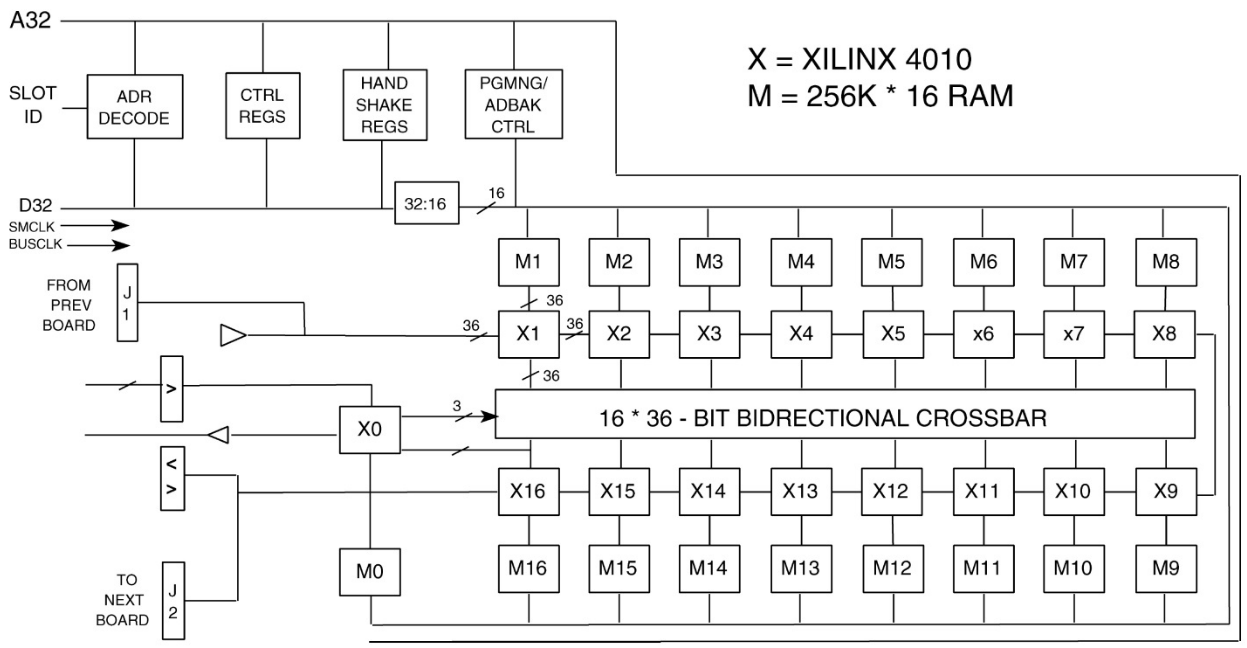


Figure 4. SPLASH2 architecture. Each board contains 16 FPGAs, XI through XI6. The blocks Ml through Ml6 are local memories of the FPGAs. A simplified 36-bit bus crossbar, with no permutation of the bit-lines within each bus, interconnects the 16 FPGAs. Another 36-bit bus connects the FPGAs in daisy-chain fashion. The local memories are dual ported with one port connecting to the FPGAs and the other port connecting to the external bus.

Other designs have used a hierarchy of interconnect schemes, differing in performance. The use of multi-gigabit transceivers (MGT) available on contemporary FPGAs allows high bandwidth interconnection using commodity components. An example is the Berkeley Emulation Engine 2 (BEE2) [27], designed for reconfigurable computing and illustrated in **Fig. 5**. Each compute module consists of five FPGAs (Xilinx XC2VP70) connected to four double data rate 2 (DDR2) dual inline memory modules (DIMMs) with a maximum capacity of 4GB per FPGA. Four FPGAs are used for computation and one for control. Each PPGA has two PowerPC 405 processor cores. A local mesh connects the computation FPGAs in a 2-D grid using low-voltage CMOS (LVCMOS) parallel signaling. Off-module communications are of via 18 (two from the control FPGA and four from each of the compute FPGAs) Infiniband 4*X* channel-bonded 2.5-Gbps connectors that operate full-duplex, which corresponds to a 180-Gbps off-module full-duplex communication bandwidth. Modules can be interconnected in different topologies including tree, 3-D mesh, or crossbar. The use of standard interfaces allows standard network switches such as Infiniband and 10-Gigabit Ethernet to be used. Finally, a 100 base-T Ethernet connection to the control FPGA is present for out-of-band communications, monitoring, and control.

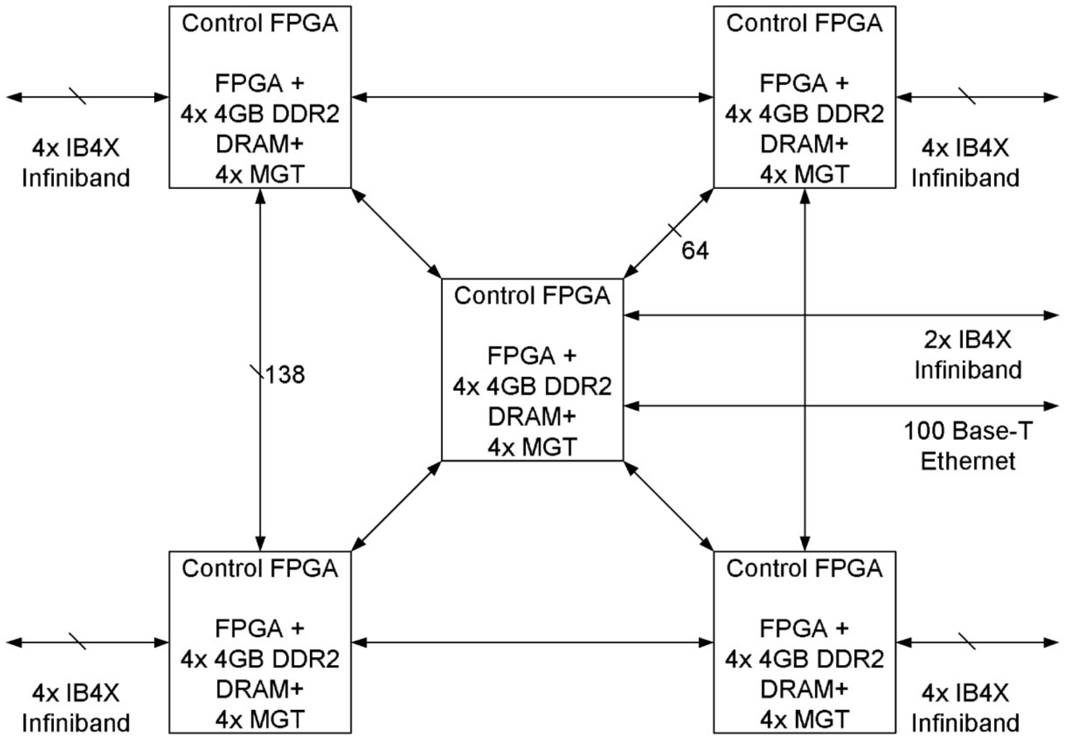


Figure 5. BEE2 Compute Module block diagram. Compute modules can be interconnected via the Infiniband IB4X connectors, either directly or via a 10-Gigabit Ethernet switch. The 100-Base T Ethernet can be used for control, monitoring, or data archiving.

Commercial machines, such as the Maxeler MPC-X2000 system [51], have a similar interconnect structure to the BEE2 in that they are parallel machines employing high performance microprocessors tightly coupled to a relatively small number of FPGA devices per node. The MPC-X2000 is a 1U server with eight large FPGAs, called dataflow engines (DFEs), interconnected in a ring arrangement. A total of 384 GB of dynamic RAM is supported and multiple host processors can communicate with each DFE via a high-speed Infiniband switched interconnect network. Such machines can have orders of magnitude performance improvement over conventional architectures and switching topologies can be altered via configuration of the switching fabric.

Microsoft took a different approach in their Catapult machine, choosing a single daughter card per server over multi-FPGA boards for the reasons of scalability, capacity, power, space and reliability [52]. Each FPGA card operates under 20 W, is hosted by a server via PCI Express and contains 8 GB of dynamic RAM. The FPGA boards are organized in a 24U arrangement of 48U half-width 1U servers, directly connected together with SAS cables. A test system containing 1,632 servers was shown to reduce the tail latency of the Microsoft Bing search engine by 29% and improve ranking throughput of each server by 95%.

The Intel-Altera Heterogeneous Architecture Research Platform (HARP) utilizes Intel Quickpath Interconnect (QPI) in a dual socket motherboard with the processor and FPGA residing each occupying a socket [53]. This offers higher bandwidth and lower latency over conventional daughter cards. A coherent shared memory between the processor and FPGA gives the promise of a greatly simplified programming model and tighter processor-FPGA coupling which will benefit irregular data access patterns.

5 Runtime Reconfiguration

A reconfigurable computing system can have its functionality updated during execution, resulting in reduced resource requirements. A runtime reconfigurable system partitions a design temporally so that the entire design does not need to be resident in the FPGA at any given moment [54, 55]. Instead, the FPGA fabric is time-shared between specialized hardware accelerators at runtime. Using this technique, designs larger than the available hardware resources can be realized, or alternatively, an existing design may be implemented on a smaller or cheaper device. Furthermore, energy efficiency can be increased because the entire fabric can be used more effectively.

Single context, multiple context architectures and partially reconfigurable FPGAs been developed. In a single context system, any changes to the functionality of the FPGA involve reloading the entire bitstream; early FPGAs were of this type. This scheme has the disadvantage of long reconfiguration time. Multiple context or time-sharing architectures, lie at the other extreme. These allow a number of complete configurations to be stored in the fabric simultaneously and thus reconfiguration can be achieved in a small number of cycles. These architectures were also proposed for early FPGAs. As an example, an architecture named Dharma, was proposed that contains a functional block and an interconnect network [56]. By breaking a large design into levels in a folded pipeline, the logic modules and interconnect can be time-shared by dynamically reconfiguring each level. This topology simplifies the architecture and provides predictable interconnect delay (**Fig. 6**). Multiple context architectures, such as NEC's Dynamically Reconfigurable Processor (DRP) [57], were later developed. Such architectures have the shortest context switch time, however, a larger area overhead is associated with implementation of this scheme.

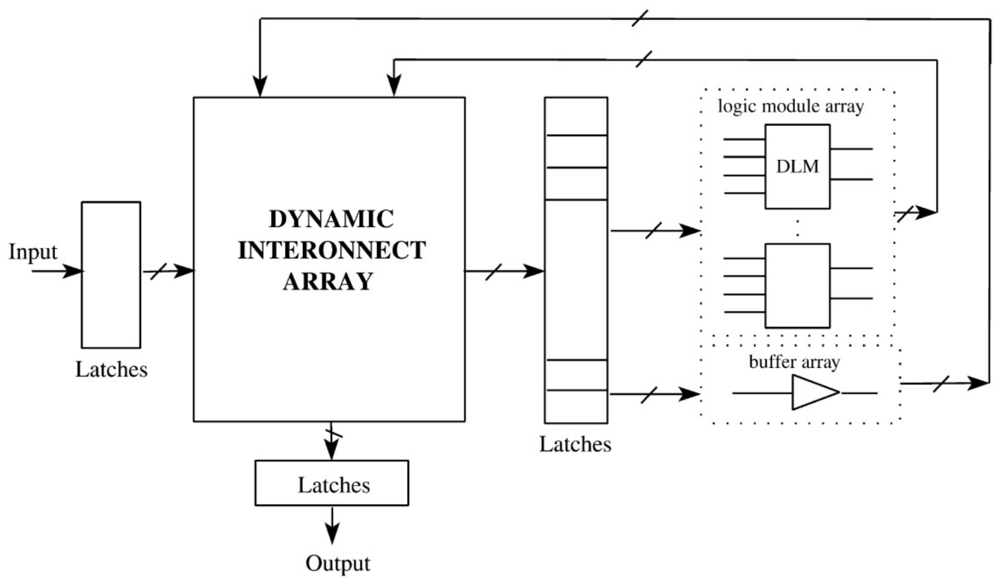


Figure 6. Dynamic Architecture for FPGA-based systems. The architecture contains a functional block and an interconnect network. The interconnect and the logic can be time shared. The emulated design topology is levelized in a folded pipeline manner. The levelized topology simplifies the architecture with predictable interconnect delay.

Partially reconfigurable FPGAs, as supported by the major FPGA vendors in Xilinx Virtex [15] and Altera Stratix [16] architectures, have begun to dominate the market. These architectures allow portions of the FPGA to be changed via a memory mapped scheme whilst the other portions of the FPGA continue functioning. In comparison to a single context scheme, there is some area overhead associated in providing this feature; ideally this is compensated for by more efficient use of the fabric.

Many tools have been developed to help support runtime reconfiguration. Commercial tools, provided by the main FPGA vendors Xilinx and Altera aim to abstract the low level implementation details from the engineer. However, other open source tools have been developed to enable more flexible systems. For example, ReCoBus-Builder introduced a simple interface for communication between the static part of a system and the dynamic modules, as well as the ability to place and route partial modules separately, before linking these compiled bitstreams at run-time [58]. This makes modules interchangeable and speeds up the compilation process. The GoAhead tool takes this further, allowing the FPGA fabric to be separated into different regions, with individual modules compiled to fit into one or more of these regions [59]. It also provides support for modules to communicate between or across regions. This improves flexibility in placement of modules and promotes sharing across regions. Tools to help determine the optimum number of regions have also been generated [60].

The aforementioned tools are also able to support hierarchical designs, where a partial region does not need to be fully reconfigured; instead a smaller region within this area can be reconfigured. This has multiple advantages. Firstly, storing a few different modules at each hierarchical level provides a huge amount of flexibility, saving significant configuration memory in comparison to storing all different modules at the highest level. Furthermore, since only a small region needs to be reconfigured, the reconfiguration time is reduced [61].

Tools also exist to help overlap re-configuration and computation to maximize the performance of the device. For example ZyCAP, which is based on the Xilinx Zynq architecture with an embedded ARM CPU, provides software drivers to help reconfiguration be overlapped with computation by controlling all the reconfiguration processes [62]. It can alert the software that configuration is complete, and also manages how partial bitstreams are stored in memory. This is important to help maximize performance, for example, this tool offers the ability to cache partial bitstreams in DRAM to speed up the reconfiguration process. Finally, there are also efforts to verify the partial bitstreams perform the desired functionality [63].

There are many examples of run-time reconfiguration, with the logical unit of reconfiguration ranging from application-level down to a sub-instruction. These are discussed below:

At the application level, examples include adapting the bitstream according to changes in environmental conditions. For example, Claus et al. discussed how hardware accelerators may be needed for real-time video processing, but in the context driver assistance, adapting them according to changing light conditions could improve performance. They demonstrated that this can prove worthwhile since modules can be quickly reconfigured between frames [64].

Task level reconfiguration is common for software defined radio, for example when switching between encoding schemes. The trade-offs between full or partial reconfiguration in this problem domain are discussed by Delahaye et al. [65]. Similarly, Feillen et al. discussed how different stages of digital video decoding do not needs to operate concurrently, meaning the same hardware could be re-used in this example [66]. Task level reconfiguration for an operating system has also been proposed [67]. Under control of software running on a microprocessor, task circuits can be scheduled online and placed in a suitable free space in a hardware task area. Communications between tasks and I/O are done through a task communication bus, and termination of a task frees the reconfigurable resources used. It was shown that hardware in the hardware task area can be shared by tasks and the overheads associated with its implementation on a partially configurable platform were acceptably low. This helps improve scheduling of real-time tasks.

Instruction level reconfiguration has been demonstrated for hardware accelerated database queries. Different hardware modules for SQL queries could be dynamically configured to improve performance [68] and energy efficiency [69]. A CPU system with custom instructions is another common candidate for instruction level reconfiguration. An early example includes the Dynamic Instruction Set Computer (DISC) [70], which supported demand-driven modification of the instruction set through partial reconfiguration. The commercial Stretch processor [71] combines reconfigurable fabric with a processor to support the execution of custom instructions implemented on a reconfigurable fabric. Furthermore, the fabric can be reconfigured at runtime and the design environment is software-centric, with programming of the processor being in Stretch C.

Finally, partial reconfiguration has also been shown for sub-instructions. For example, a pipeline stage could be a convenient unit of reconfiguration, as demonstrated by incremental pipeline reconfiguration [72]. Assume an FPGA that has enough silicon area for N physical pipeline stages, but the design contains M pipeline stages (where M>>N). Through adding one pipeline stage and removing the trailing pipeline stage in each stage of the computation, execution and computation can be overlapped. Such a circuit will implement a pipeline of depth N and fully utilize the FPGA at any given point in time. Runtime reconfiguration can be done at even lower levels. Examples include those supporting hierarchical designs for a CPU with greater numbers of custom instructions [61] and a crossbar switch which employs runtime reconfiguration of the FPGA's routing resources [73]. By partially reconfiguring routing multiplexers, this scheme was able to achieve density, switch update latency and performance higher than possible using conventional means.

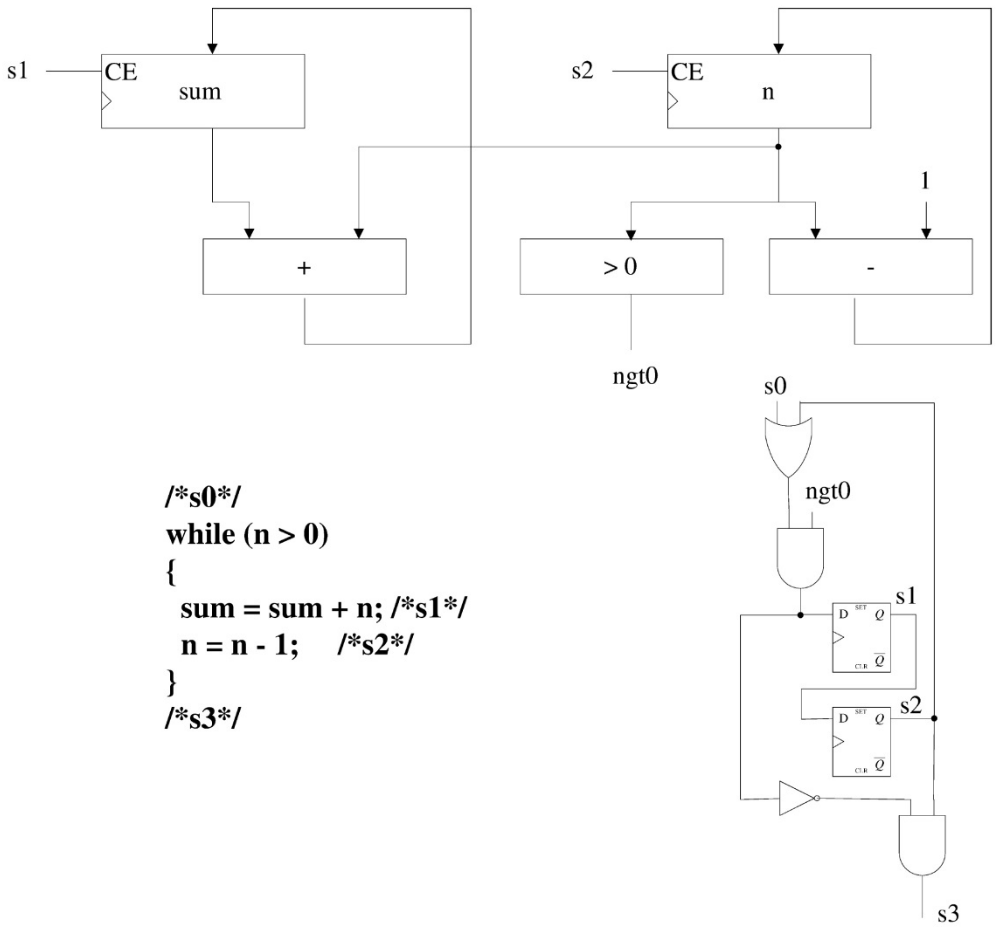
6 Design methods

Hardware description languages (HDLs) such as the Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog are commonly used to specify the logic of a reconfigurable system. Descriptions in these languages have the advantage of being vendor neutral, so the same description can be synthesized for different targets such as different FPGA devices, different FPGA vendors, and ASICs. For this reason, these languages are often the target language for higher level tools that offer higher levels of abstraction.

Module generators and libraries are commonly deployed to promote reuse. For example, vendors such as Altera and Xilinx have parameterized libraries of components that can be used in a design. These libraries are generated so that a circuit optimized for the particular application can be produced. As an example, a parameterized floating point library might allow the wordlength of the exponent and significand to be specified as well as whether denormalized numbers are supported. The module generator then generates a netlist or VHDL-based floating point adder that can be included in a design. Open source alternatives, such as the FloPoCo library also provide vendor neutral alternatives to generate many key components [74].

In an effort to help make FPGAs more mainstream, efforts have been placed into high-level synthesis, which is the process of compiling a traditional high level language down to a netlist or HDL. The use of traditional programming languages improves productivity as low level details are handled by the compiler. This is analogous to C versus assembly language for software development. Another difference with potentially large implications is that, using these tools, software developers can also design reconfigurable computing applications

As an early example, Luk and Page described a simple compilation process [75, 76] from a high level language with explicit parallel extensions to a register transfer language (RTL) description. Parallel execution of statements is implemented via parallel processes, and these can communicate via channels through which a single-word message can be passed. Variables in the user program are mapped to registers, all expressions are implemented as combinational logic, and multiplexers are used in the case a register has multiple sources. A datapath that matches the dataflow graph of the input source description is generated using this strategy. The clocking scheme employed is a global, synchronous one, and a convention that each assignment takes exactly one clock cycle is followed. A start signal is used to feed the clock and to enable each register that corresponds to a variable, and a finish signal is generated for the assignment in the following clock cycle. To execute statements sequentially, the start and finish signals of adjacent statements are simply connected together, creating a one-hot distributed control scheme. Conditional statements and loops are formed by asserting one of several possible start signals that correspond to alternative basic blocks in a program. Completion of conditional or loop constructs and synchronization of parallel blocks are implemented by combining relevant finish signals using the appropriate combinatorial logic. An example showing the translation of a simple code fragment to control and datapath is shown in **Fig. 7**.



```
/*s0*/
while (n > 0)
{
  sum = sum + n; /*s1*/
  n = n - 1;     /*s2*/
}
/*s3*/
```

Figure 7. Hardware compilation example. The C program is translated into a datapath (top) and control (bottom). Execution of statements in the while loop are controlled by s1 and s2; s0 and s3 correspond to the start signals of the statements before and after the while loop.

High-level synthesis tools have since moved beyond simply translating a high-level language to a hardware design; instead they focus on creating an optimized hardware design. Straightforward examples may include extracting parallelism through loop unrolling or creating deeply pipelined designs to maximize clock frequency. However, finer-grained optimisations are also possible. For example, since moving data onto the FPGA can be expensive, storing data locally on the chip and re-using the data can have substantial performance implications [77]. While the idea is similar to that of caching on a microprocessor, on an FPGA the local buffers can be sized according to the needs of a particular algorithm, saving resources. Alternatively, the memory can be arranged in a fashion that provides a large memory bandwidth, which may be necessary to feed a parallel datapath. Moreoever, the interface with the DRAM can also be controlled to ensure that memory reads occur faster, for example by 'activating' rows that are soon to be read [78]. Another optimization is to fine tune the precision used throughout computations, i.e. to use as little precision as is necessary to meet your design specification. Unlike a CPU implementation, an FPGA design has the freedom to implement any precision. Since arithmetic operators with less precision use less silicon area, using the minimum precision necessary frees resources, allowing for greater parallel performance. Tools to support this design methodology, both in fixed and floating point arithmetic are being supported [79, 80].

Various other issues with the mapping of algorithms to hardware are more generally discussed by Isshiki and Dai [81], who focus on the differences between implementing bit-serial versus bit-parallel modules (e.g., adders and multipliers) on FPGA architectures. Although latency is larger for bit-serial modules, the reduction in area frequently makes area-time products significantly lower for such implementations. More specifically, such advantages as the following can be obtained: 1) For bit-parallel modules, the I/O pin limitation is a major problem, and the large size of the module cluster can result in unused space and underutilized logic resources; 2) bit-serial modules are easier to partition as cell-to-cell connections are sparse and do not cause I/O problems; and 3) high fanout nets can impair routability of bit-parallel modules. Leong and Leong [82] generalized further with a design methodology that can translate a dataflow description with signals of different wordlengths to a digit serial design.

Commercial tools that can compile standard programming languages such as Java, C, or C++ (e.g., [76]) are available. Examples include Xilinx's Vivado HLS [15], Maxeler's MaxCompiler [83] and Catapult C from Mentor Graphics [84]. Domain-specific languages such as MATLAB/Simulink offer even greater improvements in productivity because they are interactive, include a large library of primitive routines and toolkits, and have good graphing capabilities. Indeed, many designs for communications and signal processing are first prototyped in MATLAB and then converted to other languages for implementation. Tools such as the MATCH compiler [85] and Xilinx System Generator [15], Altera DSP builder [16] and Mathwork's HDL coder [86] can translate a subset of MATLAB/Simulink directly to an FPGA design. There is also interest in supporting more parallel C-to-gates flows. Support for more recent parallel programming languages is gaining traction, for example Altera SDK for OpenCL [16] and efforts to support NVidia’s CUDA using FPGAs [87].

Due to the difficulty in creating a full-custom design, there is also support for creating hardware/software co-designs. The availability of embedded operating systems such as Linux for microprocessors on an FPGA provide a familiar software development environment for programmers, greatly facilitating program development through the availability of a large range of open-source libraries as well as high quality development tools. Such tools can greatly speed up the development time and improve the quality of embedded systems. For example, Altera's Nios II C-to-Hardware acceleration compiler enable time-critical functions in a C program to be converted to a hardware accelerator that is tightly coupled to a microprocessor within the FPGA [88]. These tools will support soft processors, such as the Altera NIOS or Xilinx Microblaze, and embedded processors, such as those on the Xilinx Zynq or Altera SoCs. With the latter, for optimal performance, the parts of an algorithm that are easily parallelizable should make use of the parallel FPGA fabric, whereas serial parts of the algorithm should be run on a processor [89].

A final design approach to allow for fast FPGA prototyping is the use of overlay architectures. These are coarse-grained architectures with software-like programmability, with the aim of sacrificing some performance in exchange for ease of implementation. For example, VectorBlox extends the hardware-software paradigm by using the FPGA fabric to provide parallel vector instructions that can be easily executed [90]. A typical design flow using this technology would be to create an initial software design, add vector instructions within a software style development to obtain some acceleration and finally create custom hardware instructions for the most time consuming parts of an algorithm. This may provide a faster time to market. Many overlays architectures have been created, including some for specific applications, such as for efficient network on chip (NOC) interconnections of processors [91] or data flow graphs [92], and some designed to make use of specific hardened components on FPGAs such as DSPs [93].

7 Multichip Systems

Special care must be taken in the design of large and multichip reconfigurable systems. In this section, we describe some theoretic results relevant to the major architectural and issues associated with such designs.

7.1 Interconnect Organization

A classic Clos network [94] contains three stages: inputs, intermediate switches, and outputs, as shown in **Fig. 8**. It can be used to interconnect pins in a reconfigurable computing system, and its input and output stages are symmetric. Suppose the first stage has r $n\times m$ crossbar switches, the second stage has m $r \times r$ switches, and the third stage has r $m \times n$ switches, let us denote the network as $c(n, m, r)$. For any two-pin net interconnect requirement, the network $c(n, m, r)$ can achieve complete routability if m is not less than n. The routing method can be described by recursive operations [95]. In the first iteration, we reduce the network to $c(n\text{-}1, m\text{-}1, r)$. In the i^{th} iteration, we reduce the network to $c(n\text{-}i, m\text{-}i, r)$. When $n\text{-}i$=1, we have $r1\times(m\text{-}n+1)$ switches in the first stage, $m\text{-}n+1 r\times r$ switches in the second stage, and $r(m\text{-}n+1)\times 1$ switches in the third stage. In other words, only one input exists in each first-stage switch and one output in each third-stage switch. In this case, one second-stage $r \times r$ switch is enough to route the r inputs of r first-stage switches to the r outputs of r third-stage switches, thus completing the interconnect.

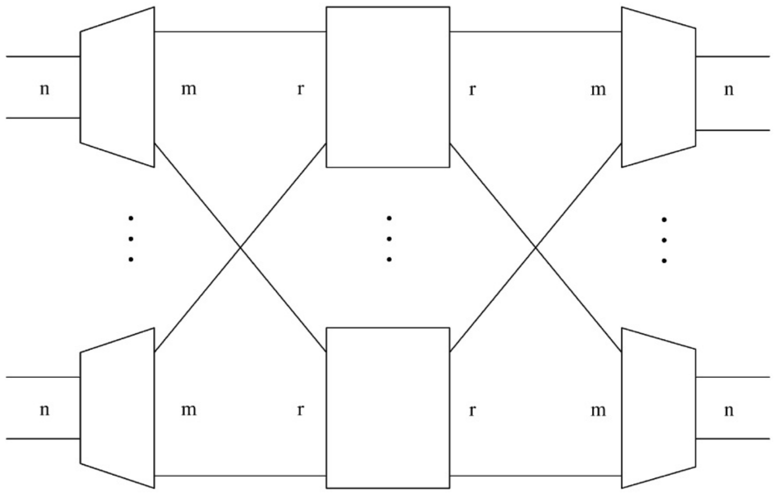


Figure 8. Clos network. A Clos network contains three stages: inputs, intermediate switches, and outputs. The input and output stages are symmetric. In the figure, the first-stage has r $n \times m$ switches, the second-stage has m $r \times r$ switches, and the third-stage has r $m \times n$ switches.

The reduction from $c(n\text{-i}, m\text{-i}, r)$ to $c(n\text{-i-}1, m\text{-i-}1, r)$ can be derived by a maximum matching algorithm. The matching algorithm selects disjoint signals from different input switches to different output switches. One second-stage switch is then used to route the selected signals. From Hall's theorem, the maximum matching and routing can always reduce the network to $c(n\text{-}i\text{-}1, m\text{-}i\text{-}1, r)$.

Conceptually, the routing problem can also be formulated as edge coloring on a bipartite graph $G(V_1, V_2, E)$ [96]. The node sets V_1 andV_2 represent the switches in the input and output stages, respectively. An edge in E represents a two-pin net interconnect requirement between the corresponding input and output switches. In Reference [96], Chan and Schlag assigned colors to the edges of the bipartite graph. Edges of the same color are bundled into one group and the corresponding set of nets are routed by one switch in the second stage. The work of Reference [97] was then used to find a minimum edge coloring solution in $O(|E| \log n)$.

The three-stage Clos network can be folded into a two-stage network (**Fig. 9**) so that the inputs and outputs are mixed in the first stage. Thus, the corresponding bipartite graph $G(V_1, V_2, E)$ constructed above for edge coloring is also folded with V_1 and V_2 merged into one set.

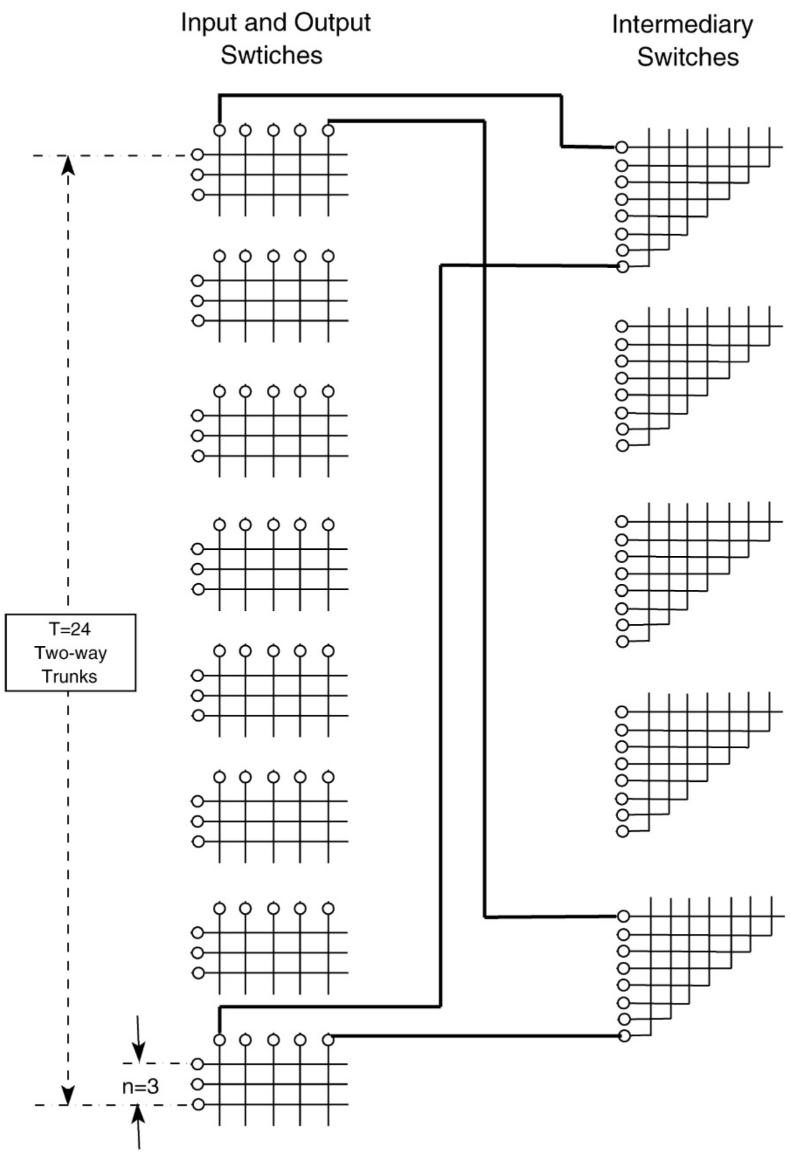


Figure 9. Folded Clos network. The three-stage Clos network is folded into a two-stage network so that the inputs and outputs are mixed in the first stage.

To find the routing assignment, the folded edge coloring graph can be unfolded back to a bipartite graph using an Euler path search. The Euler path traverses every edge exactly once and defines the edge direction according to the direction of the traversal. We then recover the original bipartite graph by splitting the node set back into two sets V_1 and V_2 and unfold the edges such that all edges are directed from $V_1$to V_2. We can find the minimum edge coloring solution of the unfolded bipartite graph and apply the solution back to the folded routing problem.

In practice, the first-level crossbar of the Clos network is replaced with FPGAs to save board space (**Fig. 10**). Routability is worse than an ideal Clos network. Even with a true Clos network, complete routability of multipin nets is not guaranteed, which is an important practical consideration because in microelectronic design, many multipin nets typically exist.

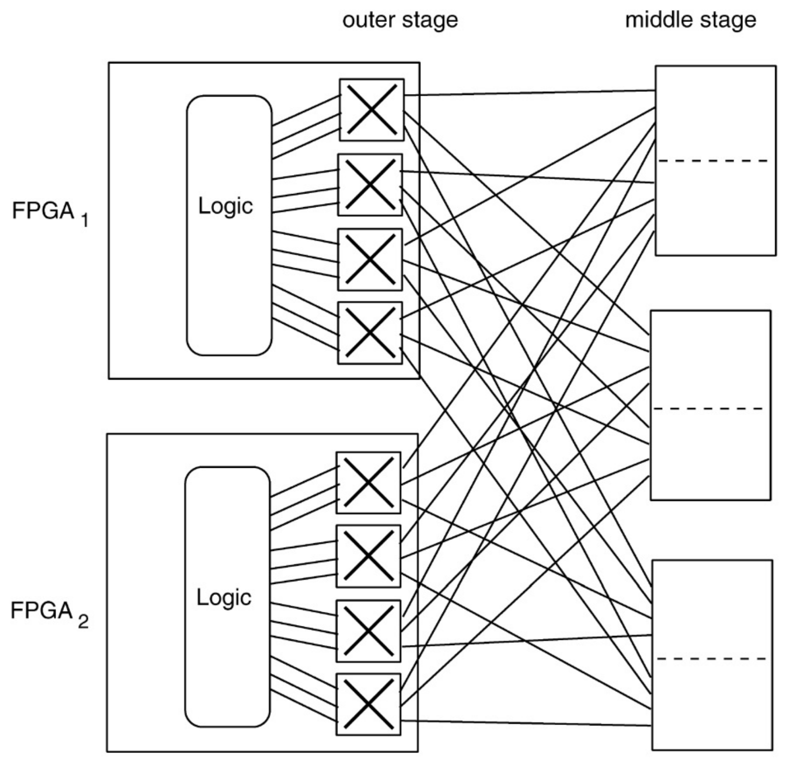


Figure 10. Variations of the Clos network. The first level crossbar of the Clos network is replaced with FPGAs to save board space. Routability is worse than an ideal Clos network.

In an attempt to solve the multipin net and routability problem, we can introduce extra connections among FPIDs as shown in **Fig. 11**. However, extra FPID interconnections also incur extra delay. We can also expand the fanout width of FPGAs so that each FPGA I/O pin is connected to more than one FPIC [98, 99]. The fanout width expansion improves routability without significant additional delay. The multiple appearances of I/O pins increase the probability that a signal connection can be made in a single stage, which is especially critical for multipin nets. However, the additional fanouts increase the needed pin count of FPICs. Thus, we need to find a balanced fanout distribution that reduces the interconnect delay with a minimal pin requirement.

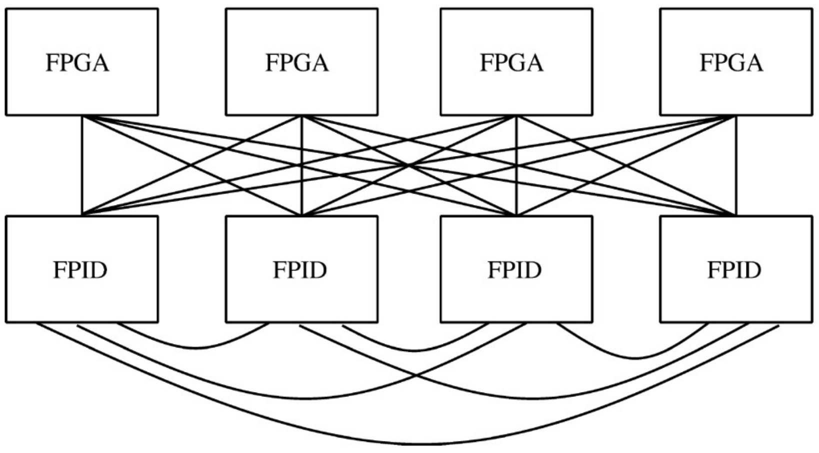


Figure 11. Variations of Clos network. The fanout width of FPGAs is expanded so that each FPGA I/O pin is connected to more than one FPIC. The fanout width expansion improves routability without significant additional delay.

A tree-structured network can simplify the mapping process for certain applications. In Reference [100], an example of a tree-structured network is illustrated for a Very Large Scale Simulator (VLSS). The VLSS tree structure has all logic components located at the leaves and interconnect switches at the internal nodes. The machine covers a capacity of eight million gates. Each branch is an 8-bit bus. The higher up the level of the tree, the less parallelism the signal distribution can achieve. Therefore, a partitioning process is designed to minimize the high level interconnect and maximize the parallel operation.

7.2 Interconnect Multiplexing

Time multiplexing is an effective method for tackling the scalability problem in interconnecting large designs. The time-sharing method can be extended from traditional bus organization [42, 100] to network sharing [101] and further to function block sharing [56].

Interconnect can be time shared as a bus [42, 100]. If *n* communication lines exist between two FPGAs, they can be reduced to a single line by storing logical outputs in shift registers and time-multiplexing the communication in phases. Such a scheme was employed in the virtual wires logic emulation system [42], which is efficient because interconnects are normally capable of being clocked at much higher rates than the critical path of the rest of the system, and all logical wires are not simultaneously active. This scheme can greatly reduce the complexity of the interconnecting network or printed circuit board in a multi-FPGA system.

Li and Cheng [101] proposed that a dynamic network be viewed as overlapping *L* conventional FPICs together but sharing the same I/O pins. A dynamic routing architecture can increase the routability and shorten interconnect length. Each switching network is a full crossbar, which can be reconfigured to provide any connections among I/O pins. The select lines are used to activate only one switching network at a time; thus the I/O pins are dynamically connected according to the configuration of this active switching network. By dynamically reconfiguring the FPICs, *L* logic signals can time-share the same interconnect resources.

7.3 Memory Allocation

Interconnect schemes should also consider how memory is connected to the FPGAs. Although combining memory with logic in the same FPGA is the most desirable method for reducing routing congestion and signal delay, separate components can supply much larger capacity at higher density and lower price. **Figure 12** demonstrates three different ways of allocating the memories in a Clos network [96, 102]. The memory may be attached directly to a local FPGA (**Fig. 12a**), attached to the second-stage switches of the Clos network via a host interface (**Fig. 12b**), or attached to the first-stage switches of the Clos network (**Fig. 12c**). The first method provides good performance for local memory access. However, for the case of nonlocal memory access, the routability and delay are concerns. The second method is slower than the first method for local memory accesses but provides better routability. The third is the most flexible as the memory is attached to the network and the routability is high. However, every logic-to-memory communication must go through the second interconnect stage.

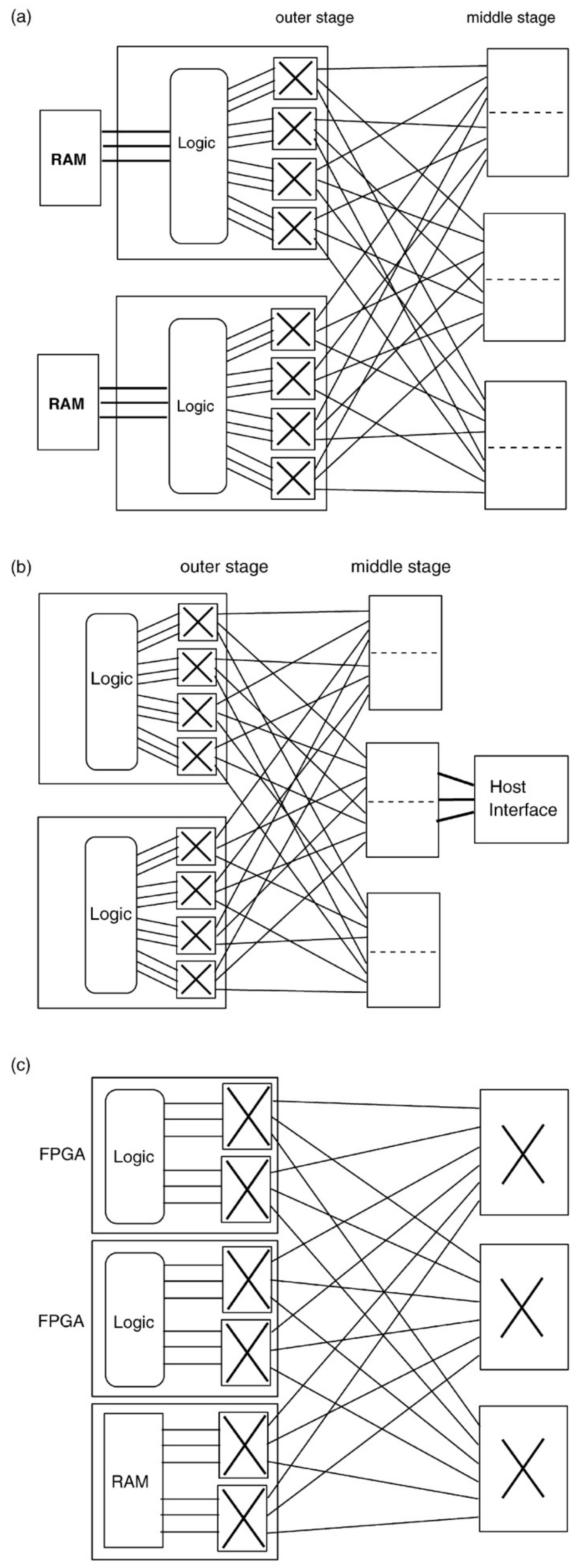

(a)
outer stage
middle stage
RAM
Logic
RAM
Logic
(b)
outer stage
middle stage
Logic
Logic
Host Interface
(c)
FPGA
Logic
FPGA
Logic
RAM

Figure 12. Memory organization, (a) Memory is attached directly to a local FPGA. (b) Memory is attached to the second-stage switches of the Clos network via a host interface, (c) Memory is attached to the first-stage switches of the Clos network.

7.4 Bus Buffer Insertion

In FPGAs, signal propagation is inherently slow because of its programmable interconnect feature. However, the delay of long routing wires can be drastically reduced by buffer insertion. The principle at work is that by inserting buffers we can decouple capacitive effects of components and interconnect driven by the buffers and thereby improve RC delay.

Given a routing topology for a net and timing requirements for its sinks, an efficient optimal buffer insertion algorithm was proposed in [103]. Experimental results show dramatic improvement versus the unbuffered solution. Thus, it is advantageous to have abundant buffers in FPGAs. However, each possible buffer and its programmable switch adds capacitance to the wires, which in turn will contribute to delay. Thus, a balance point needs to be identified to tradeoff between the additional delay and capacitance of the buffers versus the improvement they can provide.

For a multisourced bus, the problem of buffer insertion becomes more complicated, because the optimization for one source may sacrifice the delay of others. Furthermore, the direction of the buffer needs to be arbitrated by a controller. Instead of using such a controller, a novel approach is to use a patented open collector bus repeater [104]. When idle, the two ends of the repeater are set to high. When the repeater senses the pull-down action on one side, it presents the signal on the other side until the pull-down action is released from the originated signal. The bus repeater eliminates the need for a direction control signal, resulting in a simpler design and better use of resources.

7.5 System Decomposition

To decompose a system into multiple devices, Yeh et al. [105]proposed an algorithm based on the relationship between uniform multi-commodity flow and min-cut partitioning. Yeh et al. construct a flow network wherein each net initially corresponded an edge with flow cost one. Two random modules in the network were chosen and the shortest path (i.e., path with lowest cost) between them was computed. A constant $\Delta < 1$ was added to the flow for each net in the shortest path, and the cost for every net in the path was incremented. Adjusting the cost penalizes paths through congested areas and forces alternative shortest paths. This random shortest path computation is repeated until every path between the chosen pair of modules passes through at least one “saturated” net. The set of saturated nets induces a multi-way partitioning in which two modules belong to the same cluster if and only if there is a path of unsaturated nets between them.

For each of these clusters, the *flux* (defined as the cutsize between the cluster and its complement, divided by the size of the cluster) is computed and the clusters are sorted based on their flux value. Yeh et al. began with a single cluster equal to the entire netlist, and then peeled off the clusters with lowest flux. This approach was attractive because the saturated nets are good candidates to be cut in a partitioning solution. As peeled clusters can be very small, a second phase may be used to make the multi-way partitioning more balanced. This approach, with its subsequent speedup by Yeh [106], is well-suited for large-scale multi-way partitioning instances.

The system prototyping phase may also explore netlist transformations such as logic replication and retiming to minimize cut size (I/O usage) or system cycle time. Such transformations are needed as inter-device delays can be relatively large and because devices are often I/O-limited. In Reference [107], Liu et al. proposed a partitioning algorithm that permits logic replication to minimize both cut size and clock cycle of sequential circuits. Given a netlist $G = (V, E)$, their approach chooses two modules as seeds s and t, then constructs a "replication graph" that is twice the size of the original circuit. This graph has the special property that a type of directed minimum cut yields the replication cut (i.e., a decomposition of V into S, T, and $R = V\text{-}S\text{-}T$ where $s \in S$, $t \in T$ and R is the replicated logic) that is optimal. A directed version of the Fiduccia-Mattheyses algorithm is used to find a heuristic directed minimum cut in the replication graph. Cong et al. [108] present an efficient algorithm for the performance-driven multi-way circuit partitioning problem that considers the different local and global interconnect delay introduced by the partitioning.

Alpert and Kahng [109] survey the FPGA partitioning literature in the context of major graph partitioning paradigms. The current partitioning problems are (*i*) low usage rate of FPGA gate capacity because I/O pin limit, (*ii*) low clock rate because of interconnect delay between multiple FPGAs and (*iii*) long CPU time for the mapping process.

7.6 System Planning and Design Changes

For a given system decomposition to be implemented on a multi-FPGA prototyping architecture, all connections within each device and between devices must be routable. Chan et al. [110] invoke much literature on routability prediction in gate arrays, as well as theoretical concepts, such as the Rent parameter, to obtain a fast routability estimate for arbitrary netlists and FPGA architectures. Their method ascribes one of three levels of routable (easily routable, marginally routable, or unroutable) to a netlist based on various parameters. Specifically, combining a wirelength estimator due to Feuer, the average number of pins-per-cell, and the estimated Rent parameter yields a relatively accurate routability predictor. The utility of these parameters is contrasted with that of other criteria such as El Gamal's channel width requirement [111] or the average pins-per-net ratio.

In addition to routability, connections must also meet system timing constraints. Selvidge et al. [112] extend the original virtual wires [42] concept in their TIERS (Topology-IndEpendent Routing and Scheduling) approach. The problem formulation assumes that an assignment from a multiple-FPGA partitioning (i.e., a design graph) to a target topology graph has already been made. The objective is to assign “links” (i.e., signal nets) to channels between devices; as with the Virtual Wires concept, specific timeslices for a channel can be assigned to multiple links as long as no two links need to transmit signals at the same time. The TIERS algorithm uses a greedy method to order the links and then routes each link in the scheduled order while reserving channel resources; factors of up to 2.5 improvement in system cycle time are achieved.

Chang, et al. [113] address the combined issues of routability and system timing by applying layout-driven logic resynthesis techniques. For a given wire that cannot be routed, “alternative wires” and alternative functions are identified, such that the given unroutable wire can be removed from the circuit and replaced with a new wire (or wires) or new logic without affecting functionality. Cheng et al. estimate that between 30% and 50% of wires have so-called “triple-wire alternatives” (i.e., replacements consisting of three or fewer wires). Their method first routes the wires that do not have any alternatives then replaces any unroutable wire with available alternatives. System timing can be improved by replacing long wires with shorter alternatives.

8 Conclusions

Reconfigurable computing offers a middle ground between software-based systems and ASIC implementations, and is often able to combine important benefits of both. Implementations are able to avoid overheads such as unnecessary data transfers, decoding and control mandatory in microprocessors, and designs can be optimized on a basis specific to an application, a problem instance or even an execution. Using this technology, it is possible to achieve size, performance, cost, or power improvements over more conventional computing technologies.

9 Acknowledgments

The authors would like to thank Y M. Lam for his help in preparing this manuscript and Prof. Wayne Luk (Imperial College) for his proofreading of this article.

Bibliography

[1] G. Estrin, "Reconfigurable Computer Origins: The UCLA Fixed-plus-variable (F+V) Structure computer," *IEEE Ann. Hist. Comput,* vol. 24, pp. 3--9, 2002.

[2] S. Hauck, "The roles of FPGAs in reprogrammable systems," *Proc. IEEE,* vol. 86, pp. 615-639, 1998.

[3] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surveys (CSUR),* vol. 34, pp. 171-210, 2002.

[4] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," *Proc. IEEE,* vol. 90, pp. 1201-1217, 2002.

[5] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proc. Computers and Digital Techniques,* vol. 152, pp. 193-205, 2005.

[6] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proc. IEEE,* vol. 103, pp. 332-354, 2015.

[7] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue,* vol. 3, pp. 54--62, 2005.

[8] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-Rich Architectures," in *Proc. Design Automation Conference*, pp. 1--6, 2014.

[9] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays* pp. 47–56, 2012.

[10] D. B. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," *Proc. Int. Symp. on Field programmable gate arrays,* pp. 63-72, 2009.

[11] A. DeHon, "The density advantage of configurable computing," *IEEE Computer,* vol. 33, pp. 41-49, 2000.

[12] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, pp. 203-215, 2007.

[13] J. Liang, R. Tessier, and D. Goeckel, "A Dynamically-Reconfigurable, Power-Efficient Turbo Decoder," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, pp. 91--100, 2004.

[14] V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAS," ed. Dordrecht, the Netherlands: Kluwer Academic Publisher, 1999.

[15] Xilinx, "http://www.xilinx.com," (accessed 2016).

[16] Altera, "http://www.altera.com," (accessed 2016).

[17] Microsemi, "http://www.microsemi.com," (accessed 2016).

[18] M. P. Leong, "FPGA Design Methodologies for High Performance Applications," The Chinese University of Hong Kong 2001.

[19] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," in *Proc. ACM/SIGDA Int. Symp. on Field programmable gate arrays*, pp. 3-12, 2000.

[20] D. Lewis, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia*, et al.*, "The Stratix II logic and routing architecture," in *Proc. Int. Symp. on Field-programmable gate arrays*, pp. 14-20, 2005.

[21] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah*, et al.*, "The Stratix™ 10 Highly Pipelined FPGA Architecture," in *Proc. Int. Symp. on Field-Programmable Gate Arrays*, pp. 159-168, 2016.

[22] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proc. conf. on Asia South Pacific design automation*, 2001.

[23] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer,* vol. 33, pp. 70-77, 2000.

[24] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD — Reconfigurable pipelined datapath," in *Proc. Int. Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126-135, 1996.

[25] L. Moll, J. Vuillemin, and P. Boucard, "High-energy physics on DECPeRLe-1 programmable active memory," in *Proc. ACM Int. Symp. on Field-programmable gate arrays*, pp. 47-52, 1995.

[26] D. T. Hoang, "Searching genetic databases on Splash 2," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines* pp. 185-191, 1993.

[27] C. Chen, J. Wawrzynek, and R. W. Brodersen, "BEE2 A High-End Reconfigurable Computing System," *IEEE Des. Test. Comput.,* vol. 22, pp. 114-125, 2005.

[28] L.-K. Ting, R. Woods, and C. F. N. Cowan, "Virtex FPGA implementation of a pipelined adaptive LMS predictor for electronic support measures receivers," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 13, pp. 86-95, 2005.

[29] M. Pohl, M. Schaeferling, and G. Kiefer, "An efficient FPGA-based hardware framework for natural feature extraction and related Computer Vision tasks," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 1-8, 2014.

[30] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," in *Proc. IEEE Symp. on Computer Arithmetic*, pp. 252-259, 1993.

[31] K. H. Tsoi, K. H. Lee, and P. H. W. Leong, "A massively parallel RC4 key search engine," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, pp. 13-21, 2002.

[32] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. C. Cheung, D. Lee*, et al.*, "Reconfigurable acceleration for Monte Carlo based financial simulation," in *Proc. Int. Conf. on Field-Programmable Technology, 2005.*, pp. 215-222, 2005.

[33] D. Boland, "Reducing Memory Requirements for High-Performance and Numerically Stable Gaussian Elimination," *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays,* pp. 244-253, 2016.

[34] N. J. Fraser, D. J. M. Moss, L. JunKyu, S. Tridgell, C. T. Jin, and P. H. W. Leong, "A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 1--6, 2015.

[35] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, "Programmable active memories: reconfigurable systems come of age," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 4, pp. 56-69, 1996.

[36] Nvidia, "(accessed 2016)." http://www.nvidia.com.

[37] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," *ACM Comput. Surv.,* vol. 47, pp. 1-23, 2014.

[38] Nvidia. ((accessed 2016)). *NVIDIA Tesla® K20-K20X GPU Accelerators Benchmarks Application Performance Technical Brief* http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf

[39] K. Ovtcharov, O. Ruwase, J.-Y. Kim, K. Strauss, and E. Chung, *Accelerating Deep Cconvolutional Neural Networks Using Specialized Hardware*: Microsoft Research, 2015.

[40] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *Int. Conf. on Machine Learning*, pp. 1337–1345, 2013.

[41] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "Fixed Point Lanczos: Sustaining TFLOP-equivalent Performance in FPGAs for Scientific Computing," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, pp. 53-60, 2012.

[42] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal, "Logic emulation with virtual wires," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 16, pp. 609-626, 1997.

[43] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 1, pp. 171-174, 1993.

[44] L. de Souza, P. Ryan, J. Crawford, K. Wong, G. Zyner, and T. McDermott, "Prototyping for the Concurrent Development of an IEEE 802.11 Wireless LAN Chipset," in *Proc. Int. Conf. on Field Programmable Logic and Application*, ed, 2003, pp. 51-60.

[45] Cadence, "Protium Rapid Prototyping Platform https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/fpga-basedprototyping/protium-rapid-prototyping-platform.html," (accessed 2016).

[46] D. M. Stephen Tridgell, Nicholas J. Fraser, and Philip H.W. Leong, "Braiding: a scheme for resolving hazards in NORMA," in *Proc. Int. Conf. on Field Programmable Technology*, pp. 136–143, 2015.

[47] P. L. Chen Zhang, Guangyu Sun, Yijin Guan, Bingjun Xiao and Jason Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, pp. 161-170, 2015.

[48] R. A. E. Louise H. Crockett, Martin A. Enderwitz, and Robert W. Stewart. , *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 all Programmable* UK, 2014.

[49] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to Programmable Active Memories," ed: DEC Memo 3, 1989, pp. 1-9.

[50] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *Proc. ACM symp. on Parallel algorithms and architectures*, 1992.

[51] Maxeler, "https://www.maxeler.com/products/mpc-xseries/," (accessed 2016).

[52] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme*, et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Int. Symp. on Computer Architecture (ISCA)*, 2014.

[53] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. Design Automation Conference*, 2016.

[54] J. Villasenor and W. H. Mangione-Smith, "Configurable Computing," *Scientif. Amer.,* vol. 276, pp. 66-71, 1997.

[55] J. Becker and M. Hübner, "Run-time reconfigurabilility and other future trends," in *Proc. symp. on Integrated circuits and systems design*, pp. 9-11, 2006.

[56] N. B. C. Bhat, K.; Kuh, E. S, "Performance-oriented Fully Routable Dynamic Architecture for a Field-programmable Logic Device," *Memorandum No. UCB/ERL M93/42, Electronics Research Lab., College of Engineering, UC Berkeley,* pp. 1-21, 1993.

[57] M.Motomura, "A Dynamically Reconfigurable Processor Architecture,," *Microprocessor Forum,* 2002.

[58] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 119-124, 2008.

[59] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A Partial Reconfiguration Framework," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, pp. 37-44, 2012.

[60] K. Vipin and S. A. Fahmy, "Efficient region allocation for adaptive partial reconfiguration," in *Proc. Int. Conf. on Field-Programmable Technology*, pp. 1-6, 2011.

[61] D. Koch and C. Beckhoff, "Hierarchical reconfiguration of FPGAs," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 1-8, 2014.

[62] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters,* vol. 6, pp. 41-44, 2014.

[63] L. Gong and O. Diessel, *Functional Verification of Dynamically Reconfigurable FPGA-based Systems*, 1 ed.: Springer International Publishing, 2015.

[64] C. Claus, R. Ahmed, F. Altenried, and W. Stechele, "Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems," in *Proc. Int. Symp on Reconfigurable Computing: Architectures, Tools and Applications*, ed, 2010, pp. 55-67.

[65] G. G. Jean-Philippe Delahaye, Christian Roland, Pierre Bomel, "Software radio and dynamic reconfiguration on a dsp/fpga platform," *Frequenz, journal of telecommunications,* pp. 152-159, 2004.

[66] M. Feilen, M. Ihmig, C. Schwarzbauer, and W. Stechele, "Efficient DVB-T2 decoding accelerator design by time-multiplexing FPGA resources," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 75-82, 2012.

[67] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Trans. on Computers,* vol. 53, pp. 1393-1407, 2004.

[68] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 45-52, 2012.

[69] A. Becher, F. Bauer, D. Ziener, and J. Teich, "Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 1-8, 2014.

[70] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 99–107, 1995.

[71] Stretch, "http://www.stretchinc.com/," (accessed 2016).

[72] H. Schmit, "Incremental reconfiguration for pipelined applications," in *Proc. 5th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 47-55, 1997.

[73] S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, and D. Levi, "A high I/O reconfigurable crossbar switch," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, pp. 3-10, 2003.

[74] F. d. D. a. B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers,* vol. 28, pp. 18--27, 2011.

[75] W. Luk and I. Page, "Compiling Occam into FPGAs," ed: EE & CS books, 1991, pp. 271-283.

[76] I. Page, "Constructing hardware-software systems from a single description," *VLSI Signal Processing,* vol. 12, pp. 87-107, 1996.

[77] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic On-chip Memory Minimization for Data Reuse," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines* pp. 251-260, 2007.

[78] S. Bayliss and G. A. Constantinides, "Optimizing SDRAM bandwidth for custom FPGA loop accelerators," *Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays,* pp. 195--204, 2012.

[79] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-Guaranteed Bit-Width Optimization," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, pp. 1990-2000, 2006.

[80] D. Boland and G. A. Constantinides, "Bounding Variable Values and Round-Off Effects Using Handelman Representations," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 30, pp. 1691-1704, 2011.

[81] T. Isshiki and W. W. Dai, "High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems," in *Proc. Int. Workshop on FPGAs*, pp. 161-174, 1995.

[82] M. P. Leong and P. H. W. Leong, "A variable-radix digit-serial design methodology and its application to the discrete cosine transform," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 11, pp. 90-104, 2003.

[83] M. Technologies, "MaxCompiler " *(white paper),* 2011.

[84] M. Graphics, "Catapult High-Level Synthesis https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls," (Accessed 2016).

[85] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A system for synthesizing optimized FPGA hardware from Matlab," in *IEEE/ACM Int. Conf. on Computer Aided Design*, pp. 314–319, 2001.

[86] Mathworks, "http://www.mathworks.com/products/hdl-coder/," (accessed 2016).

[87] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Symp. on Application Specific Processors*, pp. 35-42, 2009.

[88] D. Lau, O. Pritchard, and P. Molson, "Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines* pp. 45-56, 2006.

[89] A. G. Weisz, A. J. Melber, A. Y. Wang, A. K. Fleming, A. E. Nurvitadhi, and A. J. C. Hoe, "A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems," *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays,* pp. 264-273, 2016

[90] Vectorblox, "http://vectorblox.com/," (accessed 2016).

[91] N. Kapre and J. Gray, "Hoplite: Building austere overlay NoCs for FPGAs," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 1-8, 2015.

[92] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Int. Conf. on Field programmable Logic and Applications*, pp. 1-8, 2013.

[93] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy, "DeCO: A DSP Block Based FPGA Accelerator Overlay With Low Overhead Interconnect," *Proc. Int. Symp. on Field-Programmable Custom Computing Machines,* pp. 1--8, 2016.

[94] C. Clos, "A Study of Non-Blocking Switching Networks," *Bell System Technical Journal,* vol. 32, pp. 406-424, 1953.

[95] V. E. Beneš, "Mathematical Theory of Connecting Networks and Telephone Traffic," ed. New York: Academic Press, 1965.

[96] P. K. Chan and M. D. F. Schlag, "Architectural tradeoffs in field-programmable-device-based computing systems," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 152-161, 1993.

[97] R. Cole and J. Hopcroft, "On Edge Coloring Bipartite Graphs," *SIAM J. Comput.,* vol. 11, pp. 540-546, 1982.

[98] G. Richards and F. Hwang, "A Two-Stage Rearrangeable Broadcast Switching Network," *IEEE Trans. Communications,* vol. 33, pp. 1025-1035, 1985.

[99] I-Cube, "Using FPID Devies in FPGA-based Prototyping," ed: Application Note, 1994, pp. 1–11.

[100] Y. C. Wei, C. K. Cheng, and Z. Wurman, "Multiple-level partitioning: an application to the very large-scale hardware simulator," *IEEE J. Solid-State Circuits,* vol. 26, pp. 706-716, 1991.

[101] J. Li and C. K. Cheng, "Routability improvement using dynamic interconnect architecture," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 2-7, 1995.

[102] P. K. S. Chan, M. D. F.; Martin, M., "BORG: A Reconfigurable Prototyping Board Using Field-programmable Gate Arrays," in *Int. Workshop on FPGA*, pp. 47–51, 1992.

[103] J. Lillis, C. K. Cheng, and T. T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," in *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, pp. 138-143, 1995.

[104] W. J. Hsieh, Y. C. Jenq, C. S. Horng, and K. Lofstrom, "Input/output I/O Bidirectional Buffer for Interfacing I/O Parts of a Field Programmable Interconnection Device with Array Ports of a Cross-point Switch. ," US Patent 5,428,800, 1992.

[105] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin, "A probabilistic multicommodity-flow solution to circuit clustering problems," in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 428–431, 1992.

[106] Y. Ching-Wei, "On the acceleration of flow-oriented circuit clustering," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 14, pp. 1305-1308, 1995.

[107] L.-T. Liu, M.-t. Kuo, C.-K. Cheng, and T. C. Hu, "Performance-Driven Partitioning Using a Replication Graph Approach," in *Proc. Design Automation Conference* pp. 206-210, 1995.

[108] J. Cong, S. K. Lim, and C. Wu, "Performance driven multi-level and multiway partitioning with retiming," in *Proc. Design Automation Conf.*, pp. 274-279, 2000.

[109] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration, the VLSI Journal,* vol. 19, pp. 1-81, 1995.

[110] P. K. Chan, M. D. F. Schlag, and J. Y. Zien, "On routability prediction for Field-Programmable Gate Arrays," in *Proc. Design Automation Conference* pp. 326-330, 1993.

[111] A. E. Gamal, "Two-dimensional stochastic model for interconnections in master slice integrated circuits," *IEEE Trans. Circuits Syst.,* vol. 28, pp. 127-138, 1981.

[112] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb, "TIERS: Topology Independent Pipelined Routing and Scheduling," in *Proc. ACM Int. Symp. on Field-programmable gate arrays* pp. 25-31, 1995.

[113] S.-C. Chang, K.-T. Cheng, N.-S. Woo, and M. Marek-Sadowska, "Layout driven logic synthesis for FPGAs," in *Proc. Design Automation Conference* pp. 308-313, 1994.