Reconfigurable computing: Architectures, models and algorithms

Kiran Bondalapati* and Viktor K. Prasanna
Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-2562, USA

The performance requirements of applications have continuously superseded the computing power of architecture platforms. Increasingly larger number of transistors available on a chip have resulted in complex architectures and integration of various architecture components on the chip. But, the incremental performance gains obtained are lower as the complexity and the integration increase. Reconfigurable architectures can adapt the behaviour of the hardware resources to a specific computation that needs to be performed. Computing using reconfigurable architectures provides an alternate paradigm to utilize the available logic resources on the chip. For several classes of applications, reconfigurable computing promises several orders of magnitude speed-up compared to conventional architectures. This article provides a brief insight into the architectures, models and algorithms which facilitate reconfigurable computing.

1. Introduction

MICROPROCESSORS are at the heart of most current high performance computing platforms. They provide a flexible computing platform and are capable of executing a large class of applications. Software is developed by implementing higher level operations using the instruction set of the architecture. As a result, the same fixed hardware can be used for general purpose applications. Unfortunately, this generality is achieved at the expense of performance. The software program stored in memory has to be fetched, decoded and executed. In addition, data is fetched from and stored back into memory. These conditions force explicit sequentialization in the execution of the program. Casting all complex operations into simpler instructions to be executed on the computer results in degraded performance. Application Specific Integrated Circuits (ASICs) provide an alternate solution which addresses the performance issues of general purpose microprocessors. ASICs are designed for a specific application and hence each ASIC has fixed functionality and superior performance for a highly restricted set of applications.

A new computing paradigm using reconfigurable architectures promises an intermediate trade-off between flexibility and performance. Reconfigurable computing utilizes hardware that can be adapted at run-time to facilitate greater flexibility without compromising on performance. Reconfigurable architectures can exploit fine grain and coarse grain parallelism available in the application because of the adaptability. Exploiting this parallelism provides significant performance advantages compared to conventional microprocessors. The reconfigurability of the hardware permits adaptation of the hardware for specific computations in each application to achieve higher performance compared to software. Complex functions can be mapped onto the architecture achieving higher silicon utilization and reducing the instruction fetch and execute bottleneck.

Reconfigurable architectures have evolved from Field Programmable Gate Arrays (FPGAs). FPGAs consist of a matrix of logic blocks and interconnection network. The functionality of the logic blocks and the connections in the interconnection network can be modified by downloading bits of configuration data onto the hardware. Currently, hybrid architectures which integrate programmable logic and interconnect together with a microprocessor on the same chip are being developed. On-chip integration of reconfigurable logic reduces the memory access costs and the reconfiguration costs.

Applications are mapped onto reconfigurable architectures by analysing the computations performed. Computations that can be speeded up by using reconfigurable hardware are identified and mapped onto the reconfigurable hardware. In the presence of a microprocessor, the computations which have complex control and data structures are executed on the microprocessor. The partitioning of the computations of an application between the microprocessor and the reconfigurable hardware is performed manually or by using automatic/semi-automatic tools. The partitioned computations have to be compiled into executable code on the microprocessor and hardware configurations on the reconfigurable hardware. The reconfigurable hardware needs to be configured using the configuration information before the actual execution can be performed. This configuration can be updated at run-time to execute a different set of computations from the application.

Development of systematic scheduling and mapping techniques for computing architectures requires high level

*For correspondence. (e-mail: kiran@halcyon.usc.edu) abstractions. Computing models, which are high level abstractions of the architectures, can be utilized to develop algorithmic techniques for mapping applications onto the architectures. Reconfigurable computing is different from the Von-Neumann paradigm of computing and requires computational models different from conventional models. In this article, we briefly describe some of the theoretical and practical models of configurable computing architectures that have been developed over the past several years.

There are several application areas where reconfigurable computing has been shown to achieve significant performance. These include long multiplication[1], cryptography[1,2], genetic algorithms[3,4], image processing[5,6], genomic database search[7] and signal processing[8–10]. The nature and diversity of the reconfigurable architectures results in a wide variety of implementation issues with respect to applications. In this article, we focus on the architectural and algorithmic aspects and desist from describing the various implementation issues.

Section 2 of this tutorial presents a brief overview of reconfigurable hardware technology and architectures. Section 3 describes the methodology needed to exploit reconfigurable computing. Section 4 briefly examines some of the computational models developed for reconfigurable computing. Section 5 describes a few algorithmic techniques developed using the computational models on the reconfigurable architectures. Section 6 outlines several directions in which reconfigurable computing research is being performed. Section 7 concludes the article with a note on some of the open research issues.

2. Reconfigurable architectures

Reconfigurable architectures have evolved from FPGAs. Currently, there are a large class of FPGAs available commercially. Various computing systems have been constructed by integrating multiple FPGAs and dedicated memory. Some systems also couple a general purpose microprocessor or an ASIC such as a Digital Signal Processor (DSP) to the FPGAs. To alleviate the communication and memory access bottleneck for configuration and data, future systems are integrating configurable logic onto the same chip as a microprocessor. Such hybrid architectures can distribute the computations between different components of the system. We give a brief overview of the different classes of architectures here.

2.1 *FPGAs*

A FPGA consists of an array of combinational logic blocks overlaid with an interconnection network of wires (Figure 1). Both the logic blocks and the interconnection network are configurable. The configurability is achieved by using either anti-fuse elements or SRAM memory bits to control the configurations of transistors. Anti-fuse technology utilizes strong electric currents to create a connection between two terminals and is typically less

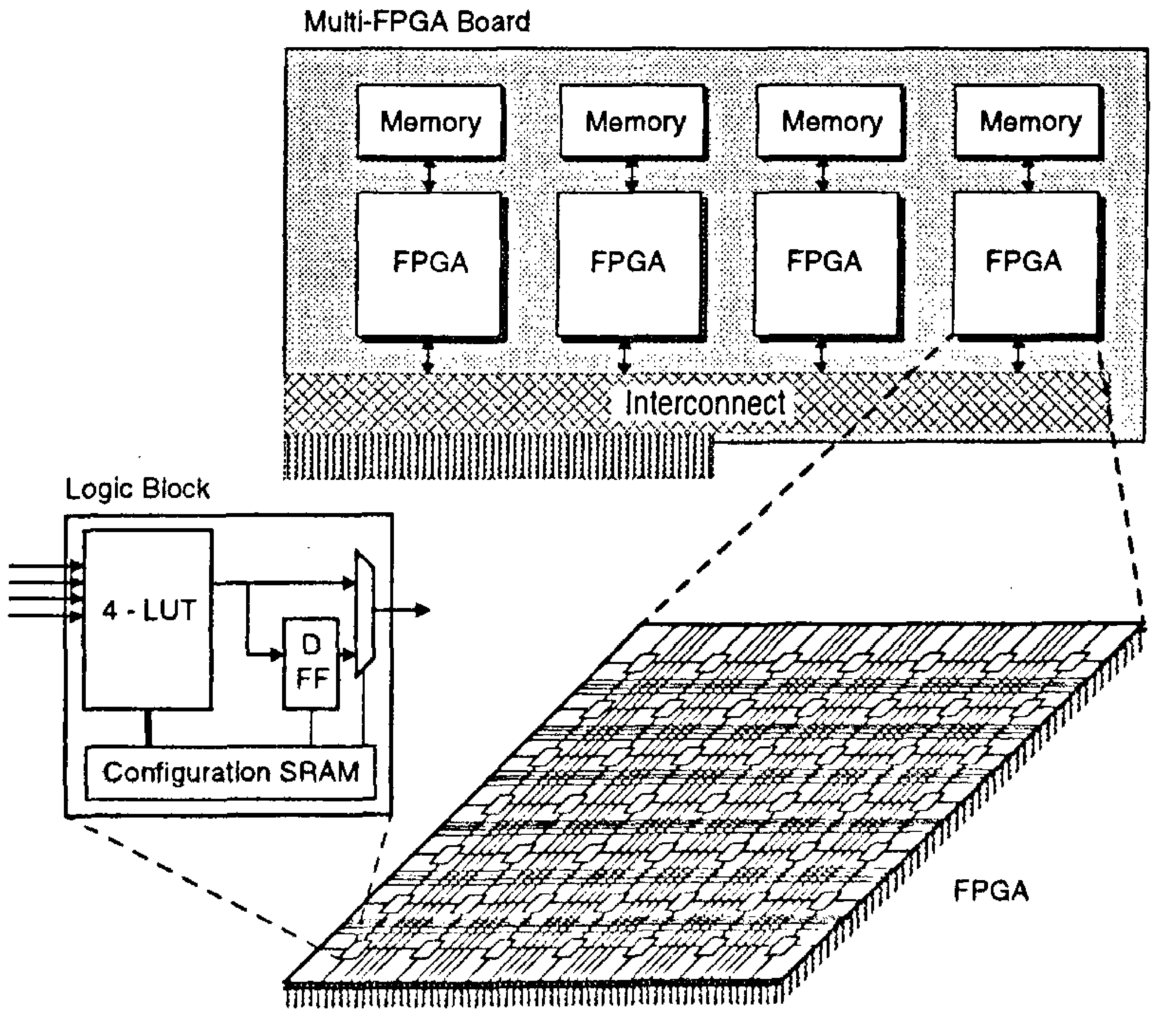

Figure 1. Typical FPGA board, device and logic block architecture.

reprogrammable. SRAM-based configuration can be reprogrammed on the fly by downloading different configuration bits into the SRAM memory cells.

Typical logic block architectures contain a look-up table, a flip-flop, additional combinational logic and SRAM memory cells to control the configuration of the logic block (see Figure 1). The logic blocks at the periphery of the device also perform the I/O operations. The interconnection network can be reconfigured by changing the connections between the logic blocks and the wires and by configuring the switch boxes which connect different wires. The switch boxes for the interconnection network are also controlled by SRAM memory cells. The functions computed in the logic block, the interconnection network and the I/O blocks can be configured using external data. FPGAs typically permit unlimited reconfiguration. These versatile devices have been used to build processors and coprocessors whose internal architecture as well as interconnections can be configured to match the needs of a given application. For a detailed architectural survey of FPGAs and related systems, see refs 11–13.

Current and future generation reconfigurable devices ameliorate the reconfiguration cost by providing *partial* and *dynamic* reconfigurability[4,14–17]. In *partial* reconfiguration, it is possible to modify the configuration of a part of the device while the configuration of the remaining part is retained. In *dynamic* reconfiguration devices permit this partial reconfiguration even while other logic blocks are performing computations. Devices in which multiple contexts of the configuration of a logic block can be stored in the logic block and the context switched dynamically have also been proposed[14,16].

Typically, the application requirements increase at a rate faster than the increase in the size of logic resources on most FPGA devices. FPGA architectures also have limits on the I/O capability due to the limitation on the number of I/O pins on the device. To map large applications onto configurable logic, various systems have been designed which have several FPGAs on a board. These architectures also provide local memory and dedicated or programmable interconnect between the FPGAs. These board-level architectures are usually designed to function under an external controller or use one of the onboard FPGAs as a controller. Examples of such systems include the experimental DECPeRLe board[1], SPLASH-2 (ref. 18) and Teramac[19] and the commercial WILD series from Annapolis Microsystems[20]. Some software tools exist which can automatically partition the design between multiple FPGAs on a board using higher level abstractions[21]. For a detailed overview of FPGA devices and multi-FPGA architectures, see Hauck[12].

2.2 *Hybrid architectures*

Configurable platforms which have shown impressive results typically have configurable logic attached to a host system through some interface such as the system bus or the I/O channels. These systems have shown significant speedups for specific applications. The limiting factor in this case in achieving higher performance on all applications is the delay in communicating with the configurable logic. This delay results in higher data transfer and reconfiguration overheads. Currently, systems which try to alleviate this problem by moving configurable logic to the processor die are being designed.

The Berkeley Garp architecture combines configurable logic with a standard MIPS processor on the same chip[15]. The configurable array is composed of a matrix of logic blocks which are organized into 32 rows of 24 blocks. One block in each row is a control block and the remaining are logic blocks which can implement a 2-bit operation. Four memory buses run vertically through the rows for moving information into and out of the array. They can be used for data transfer and memory accesses. A separate wire network provides interconnection between the logic blocks. The loading and execution of the configurations is under the control of the main processor. A transparent integrated configuration cache holds the equivalent of 128 total rows of configurations (as 4 cached configurations for each row). Reconfiguration from this cache takes 4 cycles irrespective of the number of rows. The operation of the reconfigurable array is carried out by using some extended instructions to the MIPS instruction set. The reconfigurable array, however, can perform data cache or memory accesses independent of the MIPS core.

The hybrid architecture approach is currently being commercialized by several companies. Triscend Corporation and Chameleon Systems are two of the commercial architectures available in the market. The Triscend configurable system-on-a-chip E5 architecture integrates a 8032 8-bit microcontroller, onchip programmable logic, RAM and I/Os on a chip[22]. Future generations are expected to utilize the ARM 32-bit RISC core in the reconfigurable architecture. The E5 is targeted towards embedded systems and promises fast development and high level of customization. Chameleon Systems uses the ARC 32-bit RISC-based microprocessor engine to power its reconfigurable network-processor line[23]. Next to the embedded processor, the architecture features several banks of programmable logic, allowing customers to choose system functions to suit their needs.

3. Algorithmic reconfigurable computing

To achieve high performance using configurable architectures, effective configuration design techniques have to be developed. Existing design methodologies are based on ASIC design tools and fail to realize the full potential of configurable logic. These logic synthesis tools are geared towards compiling a hardware oblivious algorithm. The behavioural description of the algorithm is mapped to logic using synthesis tools in several phases. In this process, the structure of the algorithm is not utilized resulting in sub-optimal designs with respect to area and delay performance. Also, this design methodology does not incorporate the input data knowledge into the configuration. Algorithm specific and instance-aware configurations are the key to achieving large speedups on configurable architectures. Current design compilation times are too long and preclude any run-time, dynamic modification of the configurations. Existing designs also lack modularity and scalability and have low performance (e.g. clock rates) unless optimized by hand.

One major problem in using FPGAs to speedup a computation is the design process. The 'standard CAD approach' used for digital design is typically employed (see Figure 2). The required functionality is specified at a high level of abstraction via an HDL or a schematic. FPGA libraries specific to a given device (e.g. Xilinx, Altera, etc.) and time consuming placement and routing steps are required to perform the logic mapping. This approach of *logic synthesis* as opposed to *algorithm synthesis* allows the user to specify the design using a behavioural model. But this abstraction is achieved at the expense of performance. The semantics and nature of the algorithm are lost in the mapping phases.

The model-based mapping environment takes into account the capabilities and limitations of current as well as projected hardware technologies. In Section 4 we illustrate some of the models and describe algorithmic approaches in Section 5. Parameterized models for algorithm design and analysis will possess the following characteristics:

- Cost models for analysis of reconfigurable architectures.
- Techniques for partitioning and placement of designs exploiting algorithm and input structure.
- Cost analysis incorporating the cost of reconfiguration and *partial* and *dynamic* reconfigurability.
- Impact of off-chip communication in designing reconfigurable computing solutions.
- Tradeoffs between reconfigurability and redundancy of hardware.

Utilizing FPGAs for speeding-up applications has been mostly limited to developing configurations which optimize the computation time for a given task. The optimized configuration is then used to execute the task. This process is illustrated in Figure 3. A given computational task is analysed and an optimized configuration is developed for that computational task. The configurable logic device is then configured, usually under the control of the host, with this optimized configuration. Finally the configuration is executed by initiating the computation and communicating the data to the device. The programmability of the device is not exploited and the logic resources are not reused during a computation.

The conventional approach is *static*, because the hardware is configured just once followed by execution. The concept of *dynamic* configurable computing is illustrated in Figure 4. The configurable resources are reused by reconfiguring the hardware after a computation is completed. The configuration of the logic and the interconnection network are adapted on the fly during the execution. The run-time reconfiguration can be based on intermediate results generated by the computations. This

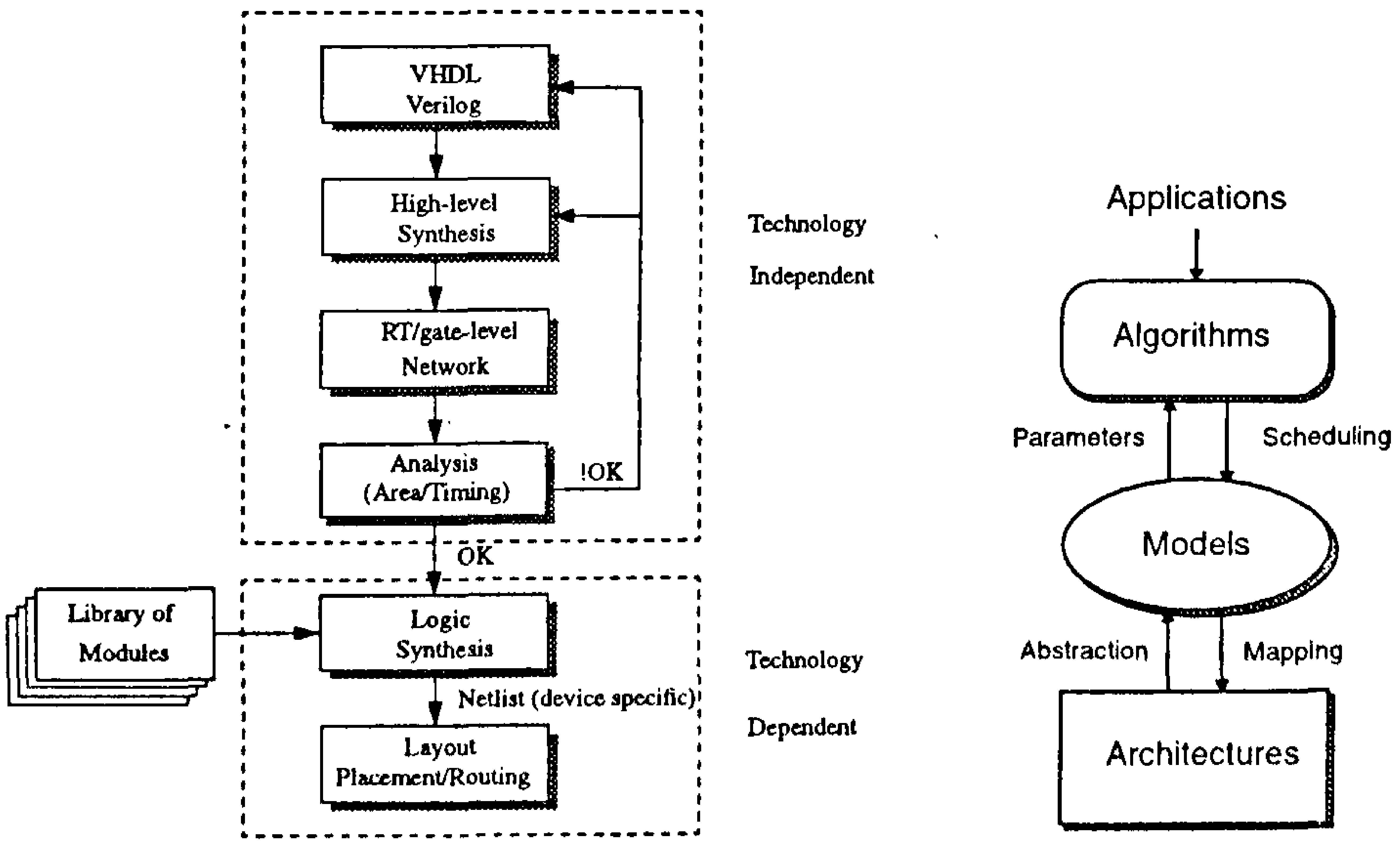

Figure 2. Traditional design synthesis approach and the model-based approach.

approach has enormous opportunities to achieve higher performance than the conventional approach by closely adapting the hardware to the nature of the computation.

4. Reconfigurable computing models

Bridging the semantic gap between the algorithm and the hardware by using such a model allows the user to develop reconfigurable computing solutions in a natural manner. We discuss here briefly a theoretical and a practical model which has been developed to map computations onto reconfigurable architectures.

4.1 *Reconfigurable mesh model*

A reconfigurable mesh is a theoretical model of a VLSI array of processors overlaid with a reconfigurable bus architecture[24,25]. An overview of various models and architectures designed based on the model is given in Bondalapati and Prasanna[26]. A reconfigurable bus architecture consists of a multi-dimensional array of processing elements (PEs) connected to a bus through a fixed number of I/O ports. This bus architecture is capable, on a per instruction basis, of configuring a topology that contributes to solving the problem at hand. Bus reconfiguration is achieved by locally configuring the switches within each PE. Different shapes of buses such as rows, columns, diagonals, zig-zag and staircase can be formed by configuring the switches/ports.

A two-dimensional processor array with a reconfigurable-bus system of size MN consisting of identical processors connected as a $M \times N$ rectangular mesh system is called a reconfigurable mesh. An example of a 4×4 reconfigurable mesh is shown in Figure 5. A set of four I/O ports labelled N, E, W and S connect each PE to its four neighbours to the north, east, west and south, respectively. Each PE has locally controllable switches which configure the connection patterns between the four I/O ports. The switches allow the broadcast bus to be divided into *sub-buses*, providing smaller reconfigurable meshes. The bus and all I/O ports are assumed to be m-bit wide. The connection patterns are represented as $\{g_1, g_2, \ldots\}$, where each of g_i represents a group of switches connected together. For example {NS, E, W} represents the connection pattern with N and S connected and E and W unconnected.

The basic computational unit of the reconfigurable mesh is the PE which consists of a switch, local storage and an ALU (Figure 5). In an unit time, a PE can perform:

1. Setting up of a connection pattern.
2. Read from or write onto a bus or local storage.
3. Logical or arithmetic operations on local data.

Various models of reconfigurable meshes have been proposed in the literature. Most of these models are synchronous in nature and permit unconditional global switch setting in addition to local switch control. Unconditional global switch setting is performed by the broadcast of a global instruction from a central controller. Reconfigurable mesh models can be characterized by several parameters such as data width of the PE, signal propagation delay, shared/exclusive access to the bus, switch connection patterns, among others.

4.2 *Hybrid system architecture model*

Hybrid system architecture model (HySAM) is a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. The HySAM model cleanly partitions the *capabilities* of the hardware from the *implementations* and presents a very clean interface to the user. Figure 6 shows the architecture of the HySAM model and an example of an architecture. The architecture consists of a traditional microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network. More details of the model can be found in refs 27 and 28.

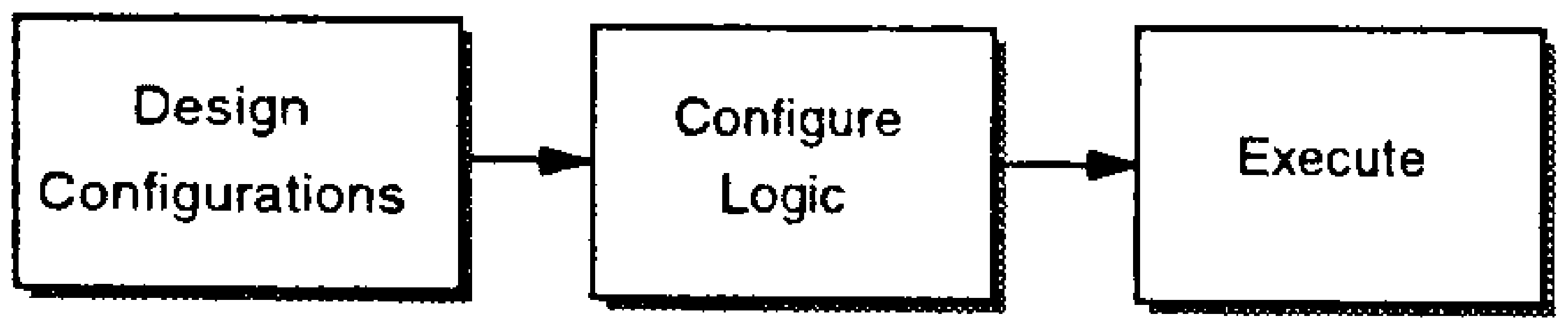

Figure 3. Static configurable computing.

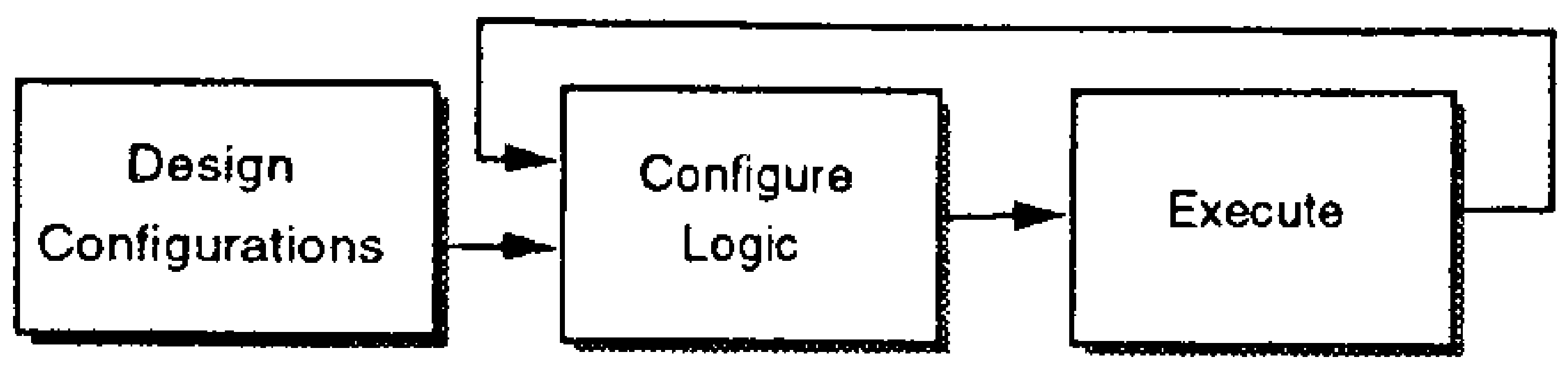

Figure 4. Dynamic configurable computing.

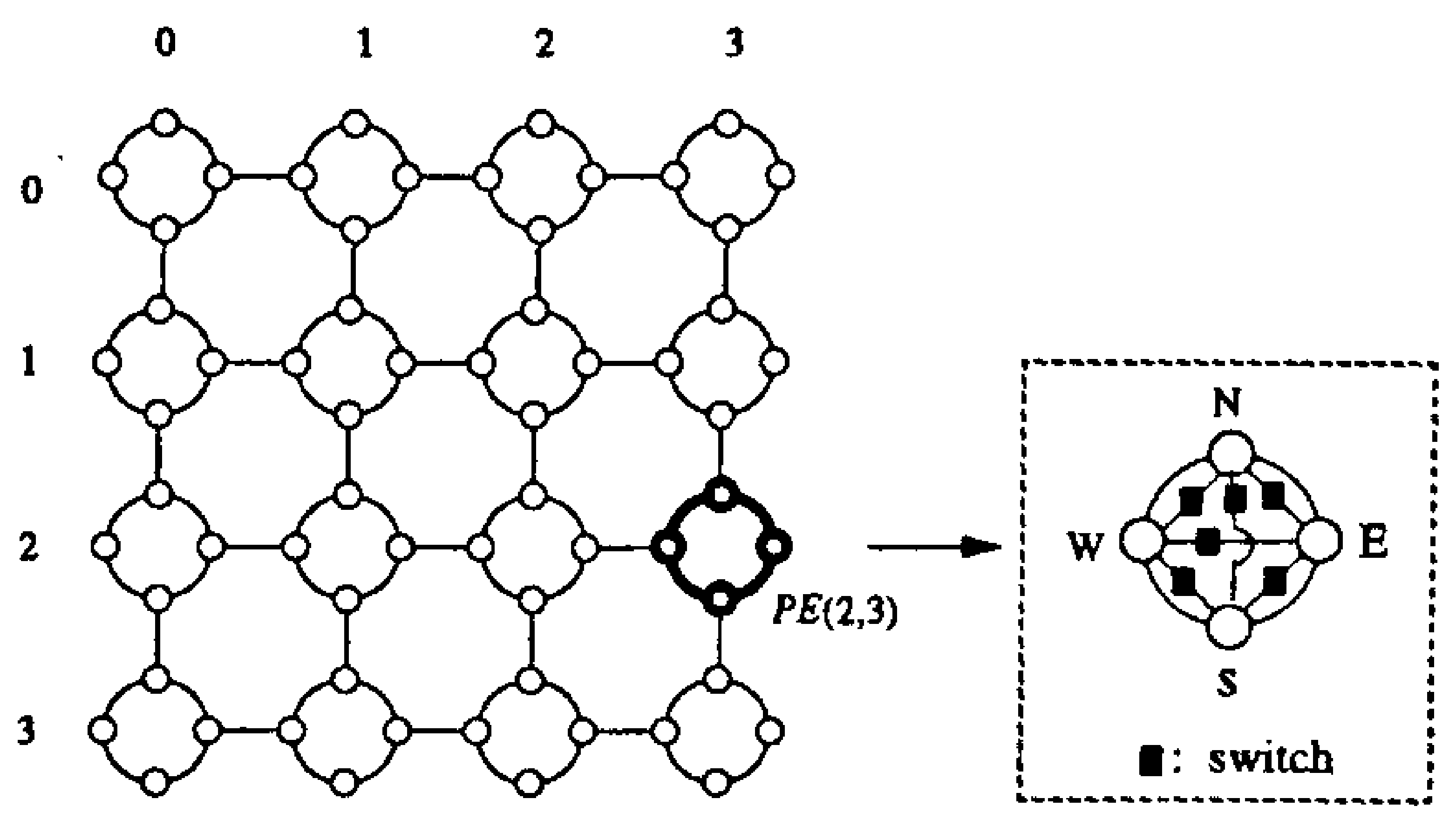

Figure 5. Reconfigurable mesh.

ome of the main parameters of the HySAM are out-d below.

$\mathcal{F}$: Set of functions $\mathcal{F}_1 \ldots \mathcal{F}_n$ which can be performed on configurable logic (*capabilities*).

C: Set of possible configurations $C_1 \ldots C_m$ of the configurable logic unit (*implementations*).

$\mathcal{A}_{ij}$: Set of attributes for implementation of function $\mathcal{F}_i$ using configuration C_j.

$\mathcal{R}_{ij}$: Reconfiguration cost in changing configuration from C_i to C_j.

B: Bandwidth of the interconnection network (bytes/cycle).

K_d: Cost of accessing data from the on-chip and external memory, respectively (cycles/byte).

'he parameterized HySAM which is outlined above can lel a wide range of systems from board-level archiures to systems on a chip. Such systems include ,ASH[18], DEC PeRLE[1], Oxford HARP[29], Berkeley p[15], Triscend E5 (ref. 22) among others. The values each of the parameters establish the architecture and ı dictate the class of applications which can be effecly mapped onto the architecture.

'he applications tasks to be executed are decomposed a sequence of Configurable Logic Unit (CLU) funcs ($\mathcal{F}$). Execution of a function on the CPU is repre:ed as execution in a special configuration ($C_{\rm cpu}$). cution of a function on the CLU involves loading the figuration onto the CLU and communicating the ıired data to the CLU. $\mathcal{F}_i$ can be executed by using any configuration from a subset of the configurations. The different configurations might have been generated by different tools, libraries or algorithms. These configurations have different area, time, reconfiguration, precision, power, etc. characteristics. For example, it is possible to design multipliers of various area/time characteristics by choosing various degrees of pipelining and carry look-ahead techniques. The multiplier can have different values for the area, pipeline stages, cycle time and number of cycles for finishing the computation. The attributes $\mathcal{A}_{ij}$ describe the characteristics of executing a function $\mathcal{F}_j$ in a configuration C_j.

After executing in a configuration C_i, to execute in a different configuration C_j, the CLU has to be reconfigured which takes a time $\mathcal{R}_{ij}$. This cost is a measure of the amount of logic reconfigured and the time spent in reconfiguring. It is possible to reduce the reconfiguration overhead by exploiting partial and dynamic reconfiguration cost. More architectural features such as the configuration caching and configuration prefetching combined with dynamic reconfiguration can be exploited to reduce this cost. The total execution time is the time spent in executing in each configuration and the time spent in reconfiguration between the executions.

5. Algorithms

The design flow for mapping application and subsequent execution on the hardware is illustrated in Figure 7. The application and the specific problem instance are utilized by the model to generate the architecture required for

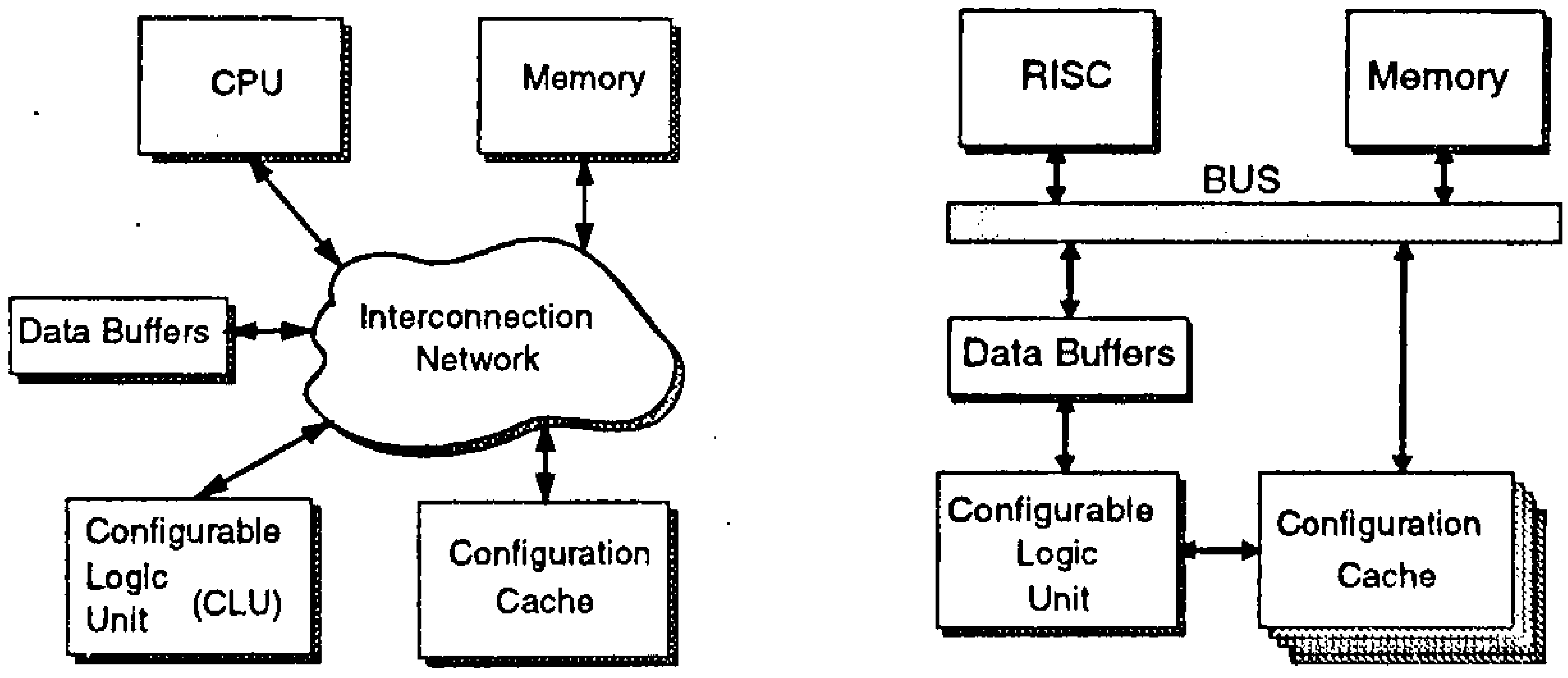

Figure 6. Hybrid system architecture model and an example architecture.

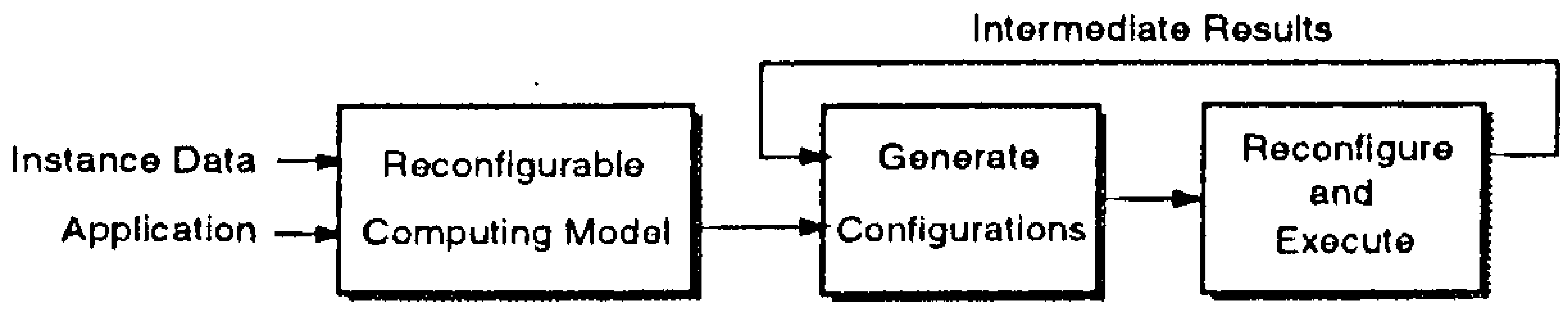

Figure 7. Computation and reconfiguration in the model-based approach.

executing the application. The configurations of the hardware required for executing are then generated by using the architecture features and the analysis performed in the model-based analysis. The generated configurations are used to reconfigure the hardware and execute the application. It is possible to update the required configurations and reconfigure the hardware by using intermediate results from the computation. The following sections illustrate the mapping of some computations on the two different models that are outlined in Section 4.

5.1 *Reconfigurable mesh computations*

A significant amount of research has been performed in exploiting the power of reconfigurable meshes. Algorithms for basic computations such as Or, And, Exor, Addition, Multiplication, etc. have been designed and shown to be optimal on several variants of the reconfigurable mesh models. Using these basic data operations and additional non-trivial techniques of exploiting reconfiguration, algorithms for problems in image processing, computational geometry, graphs, etc. have been designed. In this section some algorithms are described to illustrate the power of these architectures.

5.1.1 *EXOR computation*: The EXOR of N bits of data can be computed on a reconfigurable mesh of size $2n \times 3$ in $\theta(1)$ time assuming unit-time delay for the broadcast operation in the mesh[25]. An example EXOR computation of 3 input bits with 18 PEs is shown in Figure 8. The highlighted path shows the flow of the 1-signal from the left to the right. The result of the EXOR computation appears at the output after a constant delay. Based on a single input bit a 3×2 array of PEs set their local switch configurations to one of the two patterns as shown in Figure 8. If the input bit is 1, the top two rows cross-over and the 1-signal toggles to the other row; if the input bit is 0, the 1-signal passes through the PEs in the same row. When a 1-signal is applied to the top row input of the first processor of the system the EXOR of all the inputs appears at the last processor in the mesh. A 1-signal out of the top row indicates a result of 0 and a 1-signal out of the middle row indicates a result of 1.

5.1.2 *Sorting*: There are several sorting algorithms on reconfigurable mesh models. We describe here the algorithm presented in Jang and Prasanna[30]. Sorting of a sequence can be decomposed into sort of its subsequences and data movement between the sorted subsequences. The reconfigurable mesh algorithm uses a variation of Leighton's eight-stage column sort. The stages are a combination of stages of $n^{1/4}$ sorters, each capable of sorting $n^{3/4}$ numbers, and $n^{1/4}$-shuffle network stages. The input sequence of n numbers is assumed to be initially stored in the top row of the reconfigurable mesh. The sequence is partitioned into subsequences of $n^{3/4}$ numbers each. Sorting of a subsequence is done by computing the ranks of all the numbers and then storing each number according to its rank by using shuffle networks.

Sorting of $n^{3/4}$ numbers in constant time is carried out using a $n \times n^{3/4}$ reconfigurable mesh. In the first step, each of the $n^{3/4}$ PEs broadcast their numbers along each column of n PEs. Then the mesh is divided into $n^{3/4}$ submeshes each of size $n^{1/4} \times n^{3/4}$. The rank of number x_i is computed by submesh i using row broadcasts. The results of the comparisons made after this row broadcasts are added to give the rank of each number. The addition can be done in constant time using an algorithm similar to the EXOR computation. The $n^{1/4}$-shuffle stage can also be implemented in constant time using a sequence of broadcast operations.

5.2 *Summary of results*

We present a brief summary of algorithms on the reconfigurable mesh models. A comprehensive bibliography of results can be found in Nakano[31]. All results are with respect to the unit-time delay reconfigurable mesh model.

5.3 *HySAM dynamic precision management*

Reconfigurable hardware possesses more flexibility than ASIC hardware and can be utilized for a more diverse set

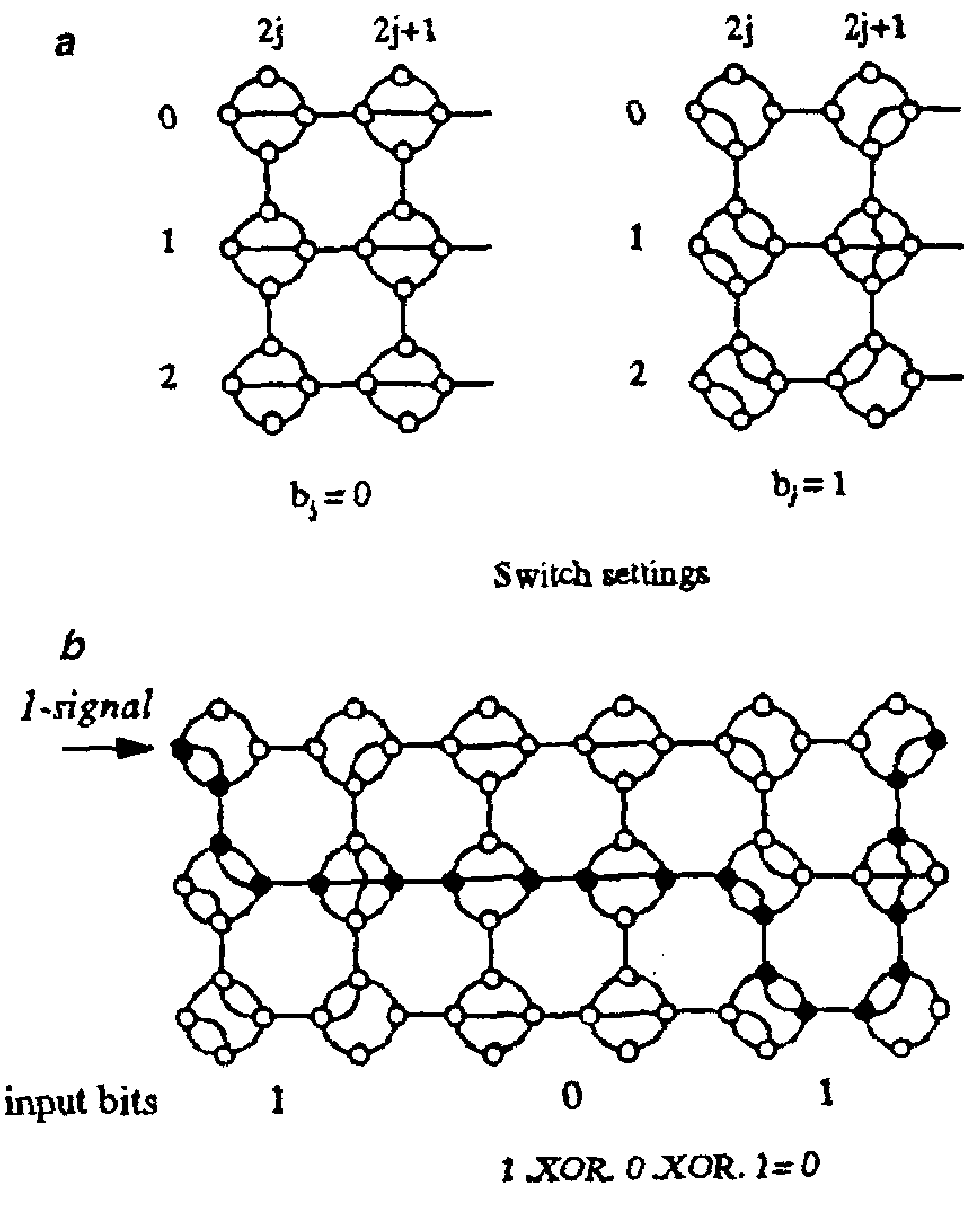

Figure 8. EXOR computation.

of computations. There are several methods of generating custom hardware configurations suited to the computations to be performed. The ability to perform variable precision arithmetic is one of the significant advantages of reconfigurable hardware[1]. Reconfigurable hardware contains finegrained configurable resources which can be utilized to build computing modules of various sizes. For example, it is possible to build a standard 16-bit × 16-bit multiplier or a 8-bit × 12-bit multiplier using reconfigurable hardware. The 8-bit × 12-bit multiplier would consume less area and execute faster than the standard 16-bit × 16-bit multiplier. Reconfigurable architectures also support *dynamic precision*, which is the ability of the hardware to change its precision at run-time in response to variant precision demands of the algorithm.

Applications typically perform operations on standard 32-bit variables. The precision of the operands and the operations is sufficient to guarantee the correctness of the operations in the worst case. But, in most applications, the actual precision required for computations is usually much lower than the precision implemented. In addition to the performance benefits obtained by mapping of computations in a loop onto configurable hardware, loops can also take advantage of variable precision. We briefly outline how the HySAM model is utilized to develop an optimal solution to the dynamic precision management problem. The complete details of the approach and the proofs can be found in Bondalapati and Prasanna[32].

Given the characteristics of the dynamic precision variation in a loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration, the precision of the corresponding configuration which executes the iteration should be equal to or greater than the required precision for that iteration. The greedy strategy of reconfiguring the hardware whenever the required precision changes can result in significant reconfiguration overheads. For architectures in which the reconfiguration times are much higher than the execution times, the reconfiguration overhead might be prohibitive. Also, the set of configurations which are available for executing an operation might not encompass all the possible precision values that are required. Some of the operations will have to be executed with more precision than is necessary in the absence of configurations with the exact precision.

Thus, it is necessary to identify an optimal set of configurations which minimizes the overall execution cost, including the reconfiguration cost. Efficient techniques using dynamic programming have been developed to map application tasks onto available configurations. These algorithmic techniques consider the reconfiguration overheads in minimizing the total execution time for a given operation in a loop[32].

An illustrativé example: The approach is illustrated by a mapping of the multiplication operation from the example code segment given below. The total execution time for the *MAXQ*SCALE(I)* computation on Xilinx XC6200 (ref. 33) is measured using five different approaches. The first two approaches do not exploit the dynamic precision variation.

```
     DO 20 I=1,N
      DO 10 J=1,N
         RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
10       IF (MAXQ.LT.RSQ(J)) THEN
            MAXQ = RSQ(J)
```

Box 1. Summary of results on the reconfigurable mesh

Problem	Mesh size	Time
EXOR of n bits	$2n \times 3$*	Constant
Prefix-And of n 1-bit numbers	$1 \times n$*	Constant
Maximum (minimum) of n log n-bit numbers	$n \times n$	Constant
Addition of n k-bit numbers, $1 \le k \le n$	$n \times nk$*	Constant
Multiplication of two n-bit numbers	$n \times n$*	Constant
Division of two n-bit numbers	$n \times n$*	Constant
Histogram of an $n \times n$ image (h grey levels)	$n \times n$	$O(\min(\sqrt{h} + \log(n/h), n))$
Sort of n $O(\log n)$ bit numbers	$n \times n$	Constant
Convex Hull of n points	$n \times n$	Constant
Smallest enclosing rectangle of n points	$n \times n$	Constant
Triangulation of n planar points	$n^2 \times n$	Constant
All-pairs nearest neighbours of n points	$n \times n$	Constant
Two-set dominance counting of n points	$n \times n$	Constant
Connected components of an $n \times n$ image	$n \times n$	$O(\log n)$

*The bit model of reconfigurable mesh is used.

```
   POVERR = POVERR / MAXQ
20 VIRTXY = VIRTXY + MAXQ * SCALE(I)
```

The execution times including the reconfiguration times are summarized in Table 1. The approaches using *dynamic precision* achieve significantly lower execution times compared to the fixed precision approaches. The dynamic programming-based algorithms (DPMA and DPMA-run) result in optimal schedules that have up to 30% lower execution cost compared with other approaches.

5. Further reading

The following topics highlight the different aspects of reconfigurable computing that research has been addressing in the past several years. The list cites resources for exploring the different research directions and is not intended to be an exhaustive or definitive list.

- *Architectures*[15,22,23,34–38]: Device and system architectures are being developed which propose various ways of organizing and interfacing configurable logic. Some architectures are also based on coarse grain functional units that are configured on the fly to execute an operation from a given set of operations. Commercial architectures are exploring integration of reconfigurable logic and microprocessors on the same chip.
- *Theoretical models*[2,5,27,39]: Various theoretical models have been proposed, including the *reconfigurable mesh* and HySAM mentioned in this article.
- *Applications*[5,8,10,40–43]: Specialized configurable architectures which are utilized for speeding up specific applications are replacing some ASICs. Some applications also exploit optimization based on a specific input instance of the computation.
- *Algorithmic synthesis*[28,29,44–54]: Dynamically reconfigurable architectures give rise to new classes of problems in mapping computations onto the architectures. New algorithmic techniques are needed to schedule the computations. Existing algorithmic mapping techniques focus primarily on loops in general-purpose programs. Loop structures provide repetitive computations, scope for pipelining and parallelization and are candidates for mapping to reconfigurable hardware.
- *Software tools*[37,55–61]: Current software tools still rely on CAD-based mapping techniques. But there are several tools being developed to address run-time reconfiguration, compilation from high level languages such as C, simulation of dynamically reconfigurable logic in software and complete operating system for dynamically reconfigurable platforms.

6. Future directions

Reconfigurable computing is an active research area with several new directions being explored to develop better architectures, algorithms and software tools. Self-reconfiguration of the devices is being explored to provide on-chip reconfiguration control avoiding external controller[62,63]. Self-reconfiguration promises higher gains and greater flexibility by removing the reconfiguration data access bottleneck and external scheduling mechanisms. Reconfigurable mesh is one model which is potentially realizable by using self-reconfigurations.

A significant bottleneck in mapping applications to reconfigurable architectures is the lack of software tools. In spite of a large number of tools developed for reconfigurable computing, most implementations are hand tuned. Most of the tools do not support or exploit dynamic reconfiguration and result in sub-optimal designs. They have been adapted from existing ASIC CAD tools as mentioned in Section 3. The designs developed using high level abstractions have to be hand tuned to achieve high performance on reconfigurable architectures. There is an acute need for software tools that permit simulation of dynamic reconfiguration and mapping of applications onto dynamically reconfigurable architectures.

To alleviate the software tools and reconfiguration overhead problems, domain-specific mapping tools are also being explored. Domain-specific mapping tools reduce the problem space by developing mapping techniques and tools for a specific application domain[48,64]. The algorithmic techniques learnt in this process are expected to be general enough to be applied to designing tools for several different domains. Domain-specific tools also permit the study of using run-time specialization of hardware based on the problem instance. Instance-dependent mapping is an approach that can provide significant performance gains compared to ASICs and microprocessors.

Table 1. Execution times using different approaches

Algorithm	Execution time (ns)	Reconfiguration time (ns)	Total (ns)
Standard	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

1. Vuillemin, J., Bertin, P., Roncin, D., Shand, M., Touati, H. and Boucard, P., *IEEE Trans. VLSI Syst.*, 1996, 4, 56–69.
2. Dandalis, A., Prasanna, V. K. and Rolim, J. D. P., IEEE Symp. on FPGAs for Custom Computing Machines, April 2000 (submitted).
3. Graham, P. and Nelson, B., in IEEE Symp. on FPGAs for Custom Computing Machines, April 1996.
4. Sidhu, R. P., Mei, A. and Prasanna, V. K., in Int. Workshop on Field Programmable Logic and Applications, September 1999.
5. Athanas, P. and Abbott, A., *IEEE Comput.*, 1995, 16–24.
6. Chung, Y. and Prasanna, V. K., in Int. Workshop on Computer Architectures for Machine Perception, October 1997.
7. Lemoine, E. and Merceron, D., in IEEE Symp. on FPGAs for Custom Computing Machines, 1995.
8. Dandalis, A. and Prasanna, V. K., in 7th Int. Workshop on Field-programmable Logic and Applications, September 1997.

9. Xilinx DSP Application Notes, The Fastest FFT in the West, http://www.xilinx.com/apps/displit.htm.
10. Shirazi, N., Athanas, P. M. and Abbott, A. L., in Int. Workshop on Field-Programmable Logic and Applications, September 1995.
11. Stephen Brown and Jonathan Rose, IEEE Design Test of Computers, Summer 1996.
12. Scott Hauck, Proc. of the IEEE–86, April 1998.
13. Jonathan Rose, Abbas El Gamal and Alberto Sangiovanni-Vincentelli, *Proc. IEEE*, 1993.
14. Andre DeHon, Ph D thesis, MIT AI Lab, September 1996.
15. Hauser, J. and Wawrzynek, J., in IEEE Symp. on FPGAs for Custom Computing Machines, April 1997, pp. 12–21.
16. Scalera, S. M. and Vázquez, J. R., IEEE Symp. on Field-Programmable Custom Computing Machines, April 1998.
17. Xilinx Inc., (www.xilinx.com), *Virtex Series FPGAS.*
18. Buell, D. A., Arnold, J. M. and Kleinfelder, W. J., *Splash 2: FPGAS in a Custom Computing Machine*, IEEE Computer Society Press, 1996.
19. Amerson, R., Carter, R. J., Culbertson, W. B., Kuekes, P. and Snider, G., in IEEE Symposium on FPGAs for Custom Computing Machines, April 1995, pp. 32–38.
20. Annapolis Microsystems, http://www.annapmicro.com/.
21. Tessier, R., Babb, J. and Agarwal, A., IEEE Workshop on FPGAs for Custom Computing Machines, April 1993.
22. Triscend Corporation, http://www.triscend.com/.
23. Chameleon Systems, http://www.chameleonsystems.com/.
24. Miller, R., Prasanna Kumar, V. K., Reisis, D. I. and Stout, Q. F., Proc. 5th MIT Conference on Advanced Research in VLSI, March 1988, pp. 163–178.
25. Miller, R., Prasanna Kumar, V. K., Reisis, D. I. and Stout, Q. F., *IEEE Trans. Comput.*, 1993.
26. Bondalapati, K. and Prasanna, V. K., in Reconfigurable Architectures Workshop, RAW '97, April 1997.
27. Bondalapati, K., Ph D thesis, University of Southern California (under preparation).
28. Bondalapati, K. and Prasanna V. K., in 8th Int. Workshop on Field-Programmable Logic and Applications, September 1998.
29. Lawrence, A., Kay, A., Luk, W., Nomura, T. and Page, I., in 5th Int. Workshop on Field-Programmable Logic and Applications, 1995.
30. Jang, J. and Prasanna, V. K., *J. Parallel Distributed Comput.*, 1995, 25, 31–41.
31. Nakano, K., A Bibliography of Published Papers on Dynamically Reconfigurable Architectures. Parallel Processing Letters, Special Issue on Dynamically Reconfigurable Architectures, 1995.
32. Bondalapati, K. and Prasanna, V. K., in IEEE Symp. on FPGAs for Custom Computing Machines, April 1999.
33. Xilinx Inc., *XC6200*, Advance Product Specification.
34. Bittner, R. and Athanas, P., in ACM Int. Symp. on Field-Programmable Gate Arrays, February 1997, pp. 79–85.
35. Cadambi, S., Weener, J., Goldstein, S. C., Schmit, H. and Thomas, D. E., in Proc. ACM/SIGDA Sixth Int. Symp. on Field-Programmable Gate Array's, February 1998.
36. Ebeling, C., Cronquist, D. C. and Franklin, P., in 6th Int. Workshop on Field-Programmable Logic and Applications, 1996.
37. Kress, R., Hartenstein, R. W. and Nageldinger, U., in 7th Int. Workshop on Field-Programmable Logic and Applications, September 1997, pp. 304–313.
38. Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe and Anant Agarwal, *IEEE Comp.*, 1997, 86–93.
39. Brebner, G., in Int. Workshop on Field-Programmable Logic and Applications, September 1997.
40. Choi, S. and Prasanna, V. K., in Int. Conf. on Parallel and Distributed Systems, December 1997.
41. Petersen, R. J. and Hutchings, B., in 5th Int. Workshop on Field-Programmable Logic and Applications, 1995.
42. Rashid, A., Leonard, J. and Mangione-Smith, W. H., IEEE Symp. on FPGAs for Custom Computing Machines, April 1998.
43. Zhong, P., Martonosi, M., Pranav Ashar and Sharad Malik, Int. Workshop on Field Programmable Logic, September 1998.
44. Jonathan Babb, Matthew Frank and Anant Agarwal, SPIE Photonics East: Reconfigurable Technology for Rapid Product Development and Computing, November 1996.
45. Bondalapati, K. and Prasanna, V. K., in Reconfigurable Architectures Workshop (RAW '2000) to be held in May 2000.
46. Callahan, T. J. and Wawrzynek, J., Int. Workshop on Field-Programmable Logic, September 1998.
47. Chang, D. and Marek-Sadowska, M., in *IEEE Trans. Comput.*, June 1999.
48. Dandalis, A., Mei, A. and Prasanna, V. K., in Reconfigurable Architectures Workshop, April 1999.
49. Luk, W., Shirazi, N., Guo, S. R. and Cheung, P. Y. K., in 7th Int. Workshop on Field-Programmable Logic and Applications, September 1997.
50. Payne, R., in 7th Int. Workshop on Field-Programmable Logic and Applications, September 1997, pp. 161–172.
51. Purna, K. M. G. and Bhatia, D., in *IEEE Trans. Comput.*, 1999.
52. Subramanian, R., Ramasubramanian, N. and Pande, S., in *Languages and Compilers for Parallel Computing*, August 1998.
53. Weinhardt, M., in Reconfigurable Architectures Workshop (RAW '97), IT Press Verlag, April 1997.
54. Wirthlin, M. J. and Hutchings, B. L., in ACM Int. Symp. on Field Programmable Gate Arrays, February 1997, pp. 86–92.
55. Bellows, P. and Hutchings, B., in IEEE Symp. on Field-Programmable Custom Computing Machines, April 1998.
56. Bondalapati, K., Diniz, P., Duncan, P., Granacki, J., Hall, M., Jain, R. and Ziegler, H., in Reconfigurable Architectures Workshop, RAW '99, April 1999.
57. Bondalapati, K. and Prasanna, V. K., in Int. Workshop on Field-Programmable Logic and Applications, September 1999.
58. Mike Donlin, *Comput. Design*, 1996.
59. Gokhale, M. B. and Marks, A., in Proc. 1995 Int. Workshop on Field Programmable Logic and Applications, Oxford, England, September 1995.
60. Levi, D. and Guccione, S., in ACM Int. Symp. on Field Programmable Gate Arrays, February 1999.
61. Lysaght, P. and Stockwood, J., *IEEE Trans. VLSI Syst.*, 1996.
62. Sidhu, R. P. S., Wadhwa, S., Mei, A. and Prasanna, V. K., IEEE Symp. on FPGAs for Custom Computing Machines, April 2000 (submitted).
63. Zebulum, R., Stoica, A. and Keymeulen, D., Third Int. Conference on Evolvable Systems: From Biology to Hardware, April 2000.
64. Hutchings, B. L., Int. Workshop on Field-Programmable Logic and Applications, September 1997.
65. MAARC Homepage, http://maarc.usc.edu.
66. MAARCII Homepage, http://maarcII.usc.edu.

ACKNOWLEDGEMENTS. Over the years, several students at USC have conducted research in reconfigurable architectures and algorithms. We would like to acknowledge the contributions of Seonil Choi, Andreas Dandalis, Ju-wook Jang, Heonchul Park, Dionisis Reisis and Reetinder Sidhu. This paper summarizes some of their contributions in reconfigurable computing.

The USC MAARC[65] and MAARCII[66] (Models, Algorithms and Architectures for Reconfigurable Computing) projects have addressed several issues described in this article. Research at USC in Reconfigurable Computing has been supported in part by the DARPA Adaptive Computing Systems program under contract numbers DABT63-96-C-0049 and DABT63-99-1-0004.