NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip

Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, *Student Member*, *IEEE*, Rutuparna Tamhankar, *Student Member*, *IEEE*, Stergios Stergiou, *Student Member*, *IEEE*, Luca Benini, *Member*, *IEEE*, and Giovanni De Micheli, *Fellow*, *IEEE*

Abstract—The growing complexity of customizable single-chip multiprocessors is requiring communication resources that can only be provided by a highly-scalable communication infrastructure. This trend is exemplified by the growing number of Network-on-Chip (*NoC*) architectures that have been proposed recently for System-on-Chip (*SoC*) integration. Developing NoC-based systems tailored to a particular application domain is crucial for achieving high-performance, energy-efficient customized solutions. The effectiveness of this approach largely depends on the availability of an ad hoc design methodology that, starting from a high-level application specification, derives an optimized NoC configuration with respect to different design objectives and instantiates the selected application specific on-chip micronetwork. Automatic execution of these design steps is highly desirable to increase SoC design productivity. This paper illustrates a complete synthesis flow, called `NetChip`, for customized NoC architectures, that partitions the development work into major steps (topology mapping, selection, and generation) and provides proper tools for their automatic execution (`SUNMAP`, `×pipesCompiler`). The entire flow leverages the flexibility of a fully reusable and scalable network components library called `×pipes`, consisting of highly-parameterizable network building blocks (network interface, switches, switch-to-switch links) that are design-time tunable and composable to achieve arbitrary topologies and customized domain-specific NoC architectures. Several experimental case studies are presented in the paper, showing the powerful design space exploration capabilities of the proposed methodology and tools.

Index Terms—Systems-on-chip, networks on chip, synthesis, mapping, architecture.

1 Introduction

In contrast to past projections, today the introduction of new technology solutions is increasingly application driven. As an example, let us consider ambient intelligence, which is regarded as the new paradigm for consumer electronics. Systems designed for ambient intelligence will be based on high-speed digital signal processing, with computational loads ranging from 10 MOPS for lightweight audio processing, 3 GOPS for video processing, 20 GOPS for multilingual conversation interfaces, and up to 1 TOPS for synthetic video generation [4]. This computational challenge will have to be addressed at manageable power levels and affordable costs, and a single processor will not suffice, thus driving the development of increasingly more complex Multi-Processor Systems-on-Chip (*MPSoCs*).

SoCs represent high-complexity, high-value semiconductor products that incorporate building blocks from multiple sources (either in-house made or externally supplied), such as general-purpose fully programmable processors, coprocessors, DSPs, dedicated hardware accelerators, memory blocks, I/O blocks, etc. Even though commercial products currently exhibit only a few integrated cores (e.g., NEC's new TCP/IP offload engine is powered by 10 Tensilica Xtensa Processor Cores [42]), in the next few years technology will support the integration of thousands of cores, making a large computational power available.

Full exploitation of the increased level of SoC integration requires new paradigms and significant improvements of design productivity, as current system architectures and design styles do not scale up to such dimensions and complexities. A relevant example regards the system architecture, whose paradigm is progressively shifting from computation-centric to communication-centric. In fact, MPSoC performance will be increasingly determined by the ability of the communication infrastructure to efficiently accommodate the communication needs of the integrated computation resources. Traditional state-of-the-art shared busses cannot meet the scalability requirements of complex MPSoCs due to the serialization of bus access requests, and turn out to be also energy-inefficient due to the broadcast communication paradigm.

A scalable communication architecture that supports the trend of SoC integration consists of an on-chip packet-switched micronetwork of interconnects, generally known as Network-on-Chip (*NoC*) [2], [21], [34]. The scalable and modular nature of NoCs and their support for efficient on-chip communication potentially leads to NoC-based multiprocessor systems characterized by high structural complexity and functional diversity. It is observed in [14] that NoC-based systems are economically feasible if they can be used in several product variants, and if the design can be reused in different application areas. On the other hand, successful products must provide good performance characteristics,

- *D. Bertozzi and L. Benini are with DEIS, University of Bologna, Viale Risorgimento 2, 40136, Bologna, Italy. E-mail: {dbertozzi, lbenini}@deis.unibo.it.*
- *A. Jalabert is with CEA-LETI, France. E-mail: antoine.jalabert@cea.fr.*
- *S. Murali, R. Tamhankar, S. Stergiou, and G. De Micheli are with the Department of Electrical Engineering, Gates Computer Science Building, Room 330, 353 Serra Mall, Stanford University, Stanford, CA 94305. E-mail: {smurali, rutu, utopcell, nanni}@stanford.edu.*

Manuscript received 30 Jan. 2004; revised 29 June 2004; accepted 21 July 2004; published online 20 Dec. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-0035-0104.

thus requiring dedicated solutions that are tailored to specific needs. As a consequence, the challenge lies in the capability to design hardware-optimized, customizable computation platforms for each application domain [9].

Hardware optimization can be achieved by facilitating the integration of domain-specific computation resources in a plug-and-play design style. Standard interface sockets such as *Virtual Component Interface* (VCI) [43] and *Open Core Protocol* (OCP) [44] have been developed for this purpose and support the use of a common NoC as the basis for system integration. A relevant task of these interfaces is to make the NoC adaptive to the different features of the integrated cores (e.g., data and address bus width).

NoC architectures are pushing the evolution of traditional IC design methodologies in order to more effectively deal with functional diversity and complexity. At the application level, the key design challenge is to expose task-level parallelism and to formally capture concurrent communication in models of computation [14]. Then, high-level concurrent tasks have to be mapped to the underlying communication and computation resources. At this level, an abstract model of the hardware architecture is usually exposed to the mapping tool, so that area and power estimates can be given in the early design stage, and different objective functions (e.g., minimization of communication energy) can be considered to evaluate the feasibility of alternative mappings. For NoC-based MPSoCs, a critical step in communication mapping is the network topology selection for its significant impact on overall system performance, which is increasingly communication-dominated.

Although a lot of research efforts are being devoted to improving individual design activities, there are very few complete NoC design methodologies and CAD tools. Setting up a fully automated synthesis framework for NoCs is a nontrivial task, particularly for the case of application specific MPSoCs, where a set of heterogeneous computing and storage resources have to be interconnected to each other by means of a custom-tailored communication network. This translates into the need to provide design time instantiation of different network schemes and topologies, tailored to the specific application domain.

A library-based approach to NoC design could be an effective solution [12], [24], wherein predesigned soft macros are composed at instantiation time to build arbitrary topologies. However, the full exploitation of a customizable network topology requires an ad hoc design methodology spanning different levels of abstractions (from application specification to physical implementation) and deriving the most efficient NoC configuration for a given application domain.

The design methodology has to partition the design problem into manageable tasks and to define the tools and practices for those tasks. In this paper, we propose a NoC synthesis flow, called `NetChip`, for designing domain-specific NoCs and automating most of the complex and time-intensive design steps. Significantly, `NetChip` provides design support also for regular network topologies and, therefore, lends itself to the implementation of both homogeneous and heterogeneous system interconnects.

`NetChip` assumes that the application has already been mapped onto cores by using preexisting tools (such as [15]) and the resulting cores together with their communication requirements represent the inputs to our NoC synthesis flow. The tool-assisted design and generation of a customized NoC-based communication architecture is the ultimate goal of `NetChip`, and is achieved by means of three major design activities: *topology mapping*, *topology selection*, and *topology generation*. `NetChip` leverages two tools: `SUNMAP`, which performs the network *topology mapping* and *selection* functions, and `×pipesCompiler`, which performs the *topology generation* function.

`SUNMAP` produces a mapping of cores onto various NoC topologies that are defined in a topology library. The mappings are optimized for the chosen design objective (such as minimizing area, power or hop delay) and satisfy the design constraints (such as area or bandwidth constraints). `SUNMAP` uses floorplanning information early in the mapping process to determine the area-power estimates of a mapping and to produce feasible mappings (satisfying the design constraints). The tool supports various routing functions (dimension ordered, minimum-path, traffic splitting across minimum-paths, traffic splitting across all paths) and chooses the mapping onto the best topology from the library of available ones.

A design file describing the chosen topology is input to the `×pipesCompiler`, which automatically generates the SystemC description of the network components (switches, links, and network interfaces) and their interconnection with the cores. A custom hand-mapped topology specification can also be accepted by the NoC synthesizer, and the network components with the selected configuration can be generated accordingly. The resulting SystemC code for the whole design can be simulated at the cycle-accurate and signal accurate level. The `×pipesCompiler` uses the `×pipes` library, which consists of highly parameterizable network building blocks that can be tuned and composed at design time to generate the chosen topology. Thus, `NetChip` automates NoC mapping, selection, and generation functions of a design, thereby bridging an important design gap in building NoCs.

The rest of the paper is organized as follows: In the next section, we present the previous works in this area. In Section 3, we present the design methodology of `NetChip`. In Sections 4 and 5, we present the `SUNMAP` tool and the area-power models used in the tool. In Section 6, we present the architecture of networks components defined in the `×pipes` library. We present the `×pipesCompiler` in Section 7. The `NetChip` design flow is used to model several video and network applications. The communication pattern in these applications differ, thereby requiring various topologies for different applications. These are presented in Sections 8.1 and 8.2. The rich design space exploration capabilities of `NetChip` is shown in Section 8.3. The design flow can also be used to model custom hand-mapped topologies and is explained in Section 8.4. In Section 8.5, we model a DSP Filter application and generate the SystemC files of the chosen topology. The resulting design is simulated at the cycle-accurate level and the simulations are checked for functional and timing correctness, validating the output of our tools.

2 Previous Work

The most advanced state-of-the-art SoC communication architectures represent evolutionary solutions with respect to shared busses. Sonics MicroNetwork [36] is a TDMA-based bus which can easily adapt to the data-word width, burst attributes, interrupt schemes, and other critical parameters of the integrated cores, while providing very high bandwidth utilization. STBUS interconnect is a high performance

communication infrastructure that allows to instantiate shared busses as well as more advanced topologies such as partial or full crossbars. Although evolutionary from a topology viewpoint, these solutions can rely on advanced and highly automated design methodologies for the implementation of generic communication subsystems, allowing designers to rapidly assemble, synthesize, and verify their SoCs using the MicroNetwork or the STBUS interconnect as integration platforms.

However, the early works in [2], [34] pointed out the need for more scalable architectures for on-chip communication and, therefore, to progressively replace shared busses with on-chip networks. Many NoC architectures have therefore been proposed in the open literature so far, but in most cases, the design methodologies and tools are still in the early stage.

One of the earliest contributions in this area is the Maia heterogeneous signal processing architecture, proposed by Zhang et al., based on a hierarchical mesh network [10]. Unfortunately, Maia's interconnect is fully instance-specific. Furthermore, routing is static at configuration time and communication is based on circuit switching, as opposed to packet switching. In this direction, Dally and Lacy sketch the architecture of a VLSI multicomputer using 2009 technology [35]. A chip with 64 processor-memory tiles is envisioned. Communication is based on packet switching. This seminal work draws upon past experiences in designing parallel computers and reconfigurable architectures (FPGAs and their evolutions) [30], [31], [32].

Most proposed NoC platforms are packet switched and exhibit regular structure. An example is a mesh interconnection, which can rely on a simple layout and the switch independence on the network size. The NOSTRUM network described in [5] takes this approach: The platform includes both a mesh topology and the relative design methodology, wherein a concrete architecture is derived from a general NoC template, then application mapping follows.

The *Scalable Programmable Integrated Network (SPIN)* described in [3] is another regular, fat-tree-based network architecture. It adopts cut-through switching to minimize message latency and storage requirements in the design of network switches. The *Linkoeping SoCBUS* [39] is a two-dimensional mesh network which uses a packet connected circuit (PCC) to set up routes through the network: A packet is switched through the network locking the circuit as it goes. This notion of virtual circuit leads to deterministic communication behavior but restricts routing flexibility for the rest of the communication traffic.

In [8], the use of *octagon* communication topology for network processors is presented. Instead, the implementation of a *star-connected* on-chip network supporting *plesiochronous communication* among system components is described in [13].

The *Aethereal* NoC design framework presented in [7] aims at providing a complete infrastructure for developing heterogeneous NoC with end-to-end quality of service guarantees. The network supports *guaranteed throughput (GT)* for real-time applications and *best effort (BE)* traffic for timing unconstrained applications. Support for heterogeneous architectures requires highly configurable network building blocks, customizable at instantiation time for a specific application domain. For instance, the *Proteo NoC* [12] consists of a small library of predefined, parameterized components that allow the implementation of a large range of different topologies, protocols and configurations. `×pipes` interconnect [24] and its synthesizer `×pipesCompiler` [25] push this approach to the limit, by instantiating an application specific NoC from a library of composable soft macros (network interface, link, and switch). The components are highly parameterizable and provide reliable and latency insensitive operation. They represent the core of the NoC synthesis flow illustrated in this paper.

In [11], a hierarchical approach for designing on-chip networks was presented to help designers compare different design options. Design methodologies for building irregular networks have been proposed in [18], [20]. Pinto et al. [18] presents a heuristic for the constraint-driven communication synthesis of on-chip communication networks, while [20] describes a design methodology for finding minimal topologies that support low contention or contention-free communication for known communication patterns. In [19], memory optimization in single chip network fabrics is explored.

The problem of mapping cores onto NoC architectures is addressed in [22], [23], [26], [27]. In [22], a branch-and-bound algorithm is used to map cores onto a mesh-based architecture with the objective of minimizing energy and satisfying the bandwidth constraints of the NoC. A simple dimension-ordered routing is assumed in the work. In [23], the authors extend the above work for other deadlock free minimal path routing algorithms. In [26], fast algorithms for mesh NoC architectures under different routing functions (minimum path, split-traffic) and delay/bandwidth constraints are presented.

The design methodology and tools presented in this paper aim at providing MPSoC designers with a framework for the rapid selection and synthesis of application-specific NoC architectures. While still allowing the comparison and generation of regular network topologies, our NoC design framework supports the synthesis of customized irregular topologies, and bridges a gap in a largely unexplored research area.

3 Design Flow of NetChip

The design flow of `NetChip` is presented in Fig. 1a. The application is mapped onto cores during the hardware/software codesign phase using existing tools such as [15]. By means of static analysis or simulation, it is possible to determine the average rate of data transfer between the cores. The resulting cores and communication demands between them is represented by a graph, called *core graph*, and is the input to our tool. `NetChip` has three phases of operation: *topology mapping phase*, *topology selection phase*, and *topology generation phase*. `NetChip` in-turn has two tools built into it: `SUNMAP` which performs the topology mapping and selection phases and the `×pipesCompiler` which generates the selected topology.

In the *topology mapping phase*, `NetChip` takes as inputs:

- the core graph with communication among cores annotated as edge weights,
- the design objective function that needs to be optimized, and
- the design constraints that are to be satisfied by the mapping.

Netchip has a *Graphical User Interface (GUI)* designed in `TCL/TK` for entering the inputs. A snapshot of the GUI is

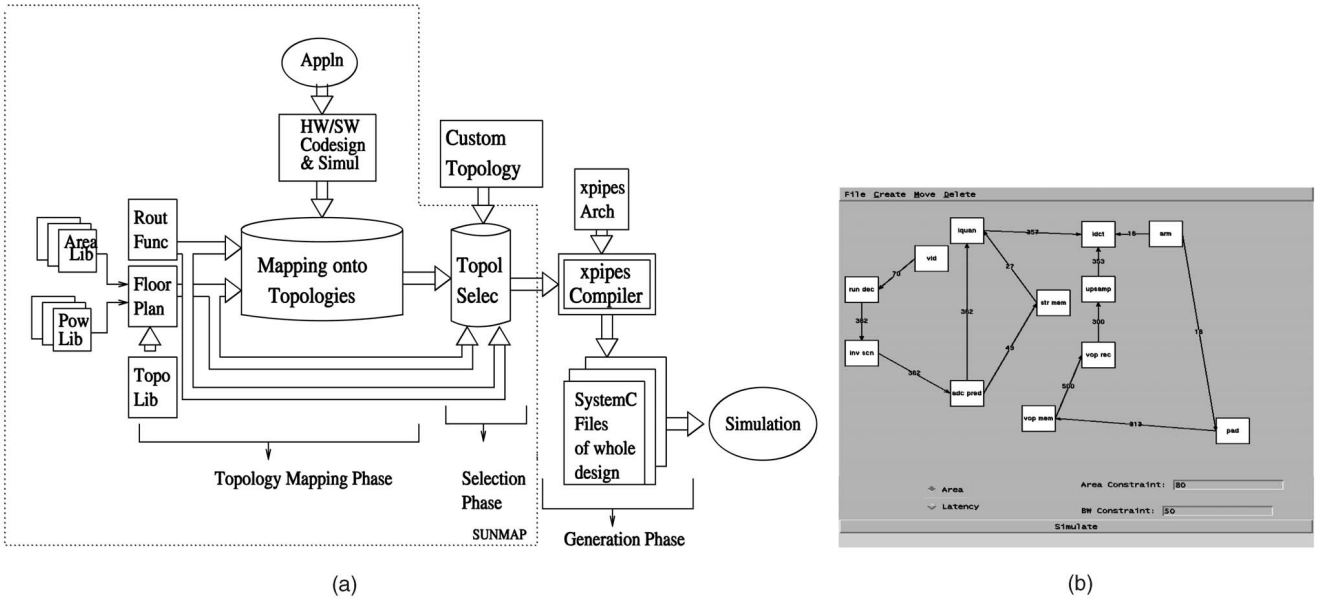

Fig. 1. Design flow of NetChip and the input GUI. (a) Design flow of NetChip and (b) snapshot of the input GUI.

presented in Fig. 1b. The input core graph is then mapped onto various standard topologies (*mesh*, *torus*, *hypercube*, *Clos*, and *butterfly*) defined in the topology library. The approach presented here is general and other topologies (such as the *star network* or the *octagon network* [13], [8]) can be easily added to the library. Netchip explores various design objectives such as minimizing average hop delay, area, and power dissipation. The tool also supports different routing functions: *dimension-ordered*, *minimum path*, *traffic splitting across minimum paths*, and *traffic splitting across all paths*. For each mapping, the bandwidth and area constraints are evaluated, so that only feasible mappings are chosen. The area-power models and floorplanner are built into NetChip, so that area-power estimates can be incorporated early in the mapping process. For a chosen design objective and routing function, the best feasible mappings onto various topologies are obtained.

In the *topology selection phase*, the various topologies (with mappings produced from the mapping phase) are evaluated for several design objectives and the best topology for the application is chosen. The design file describing the selected topology and routing files describing the routes (or paths) to be taken (which depends on the chosen routing function) are automatically generated. The SUNMAP tool which incorporates these two phases is explained in Section 4.

In the *topology generation* phase, NetChip reads the design and routing files and generates SystemC description of network components for the selected topology using ×pipesCompiler. The ×pipesCompiler instantiates a network of building blocks from the ×pipes library, which consists of composable soft macros (switches, network interfaces, and links) described in SystemC at the cycle-accurate level. The network components generated are optimized for that particular network and support reliable, latency-insensitive operation. The architecture of the ×pipes network components is presented in Section 6. In Section 7, the ×pipesCompiler is presented.

NetChip can also accept a custom hand-mapped topology and generate the network components for the topology. In such a case, the first two phases are skipped, as shown in Fig. 1a. The resulting network generated by the ×pipesCompiler is highly optimized for that particular topology. The area, power, and latency savings of the custom mappings can also be compared with mappings onto standard topologies using the NetChip tool.

4 Topology Mapping and Selection

We formulate the mapping problem mathematically as follows. The communication between the cores of the SoC is represented by the *core graph*:

Definition 1. *The core graph is a directed graph $G(V, E)$, where $V = \{v_i,\ i = 1, 2, \ldots, N1\}$, $N1 = |V|$, with each v_i representing a core and the directed edge (v_i, v_j), denoted as $e_{i,j} \in E$, representing the communication between the cores v_i and v_j. The weight of the edge $e_{i,j}$, denoted by $comm_{i,j}$, represents the bandwidth of the communication from v_i to v_j.*

The connectivity and link bandwidth of the NoC is represented by the *NoC topology graph*:

Definition 2. *The NoC topology graph is a directed graph $P(U, F)$, where $U = \{u_i,\ i = 1, 2, \ldots, N2\}$, $N2 = |U|$, with each vertex $u_i \in U$ representing a node in the topology and the directed edge (u_i, u_j), denoted as $f_{i,j} \in F$ representing a direct communication between the vertices u_i and u_j. The weight of the edge $f_{i,j}$, denoted by $bw_{i,j}$, represents the bandwidth available across the edge $f_{i,j}$.*

The mapping of the core graph $G(V, E)$ onto the processor graph $P(U, F)$ is defined by the one-to-one mapping function map:

$$map : V \rightarrow U, \text{s.t. } map(v_i) = u_j, \forall v_i \in V, \exists u_j \in U. \quad (1)$$

The mapping is defined when $|V| \leq |U|$.

As an example, the core graph of Video Object Plane Decoder (Fig. 2a) is shown in Fig. 2b. Example topology graphs of mesh, torus, hypercube, Clos, and butterfly are

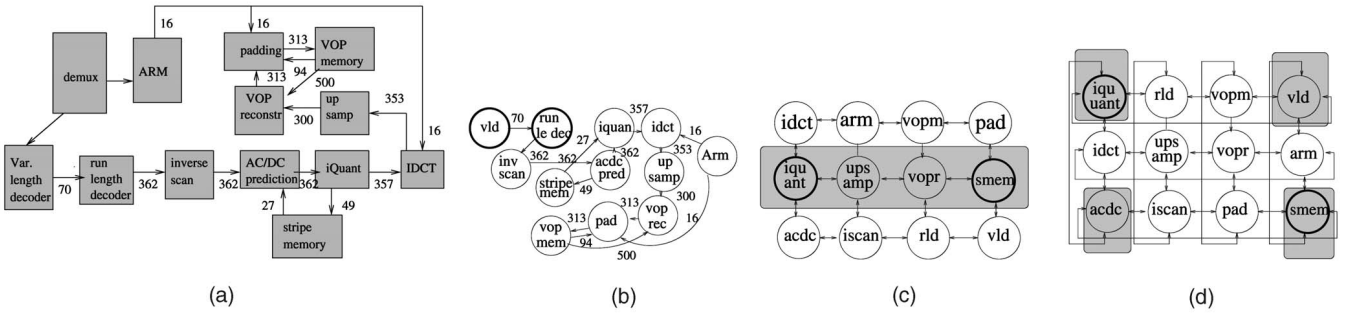

Fig. 2. *VOPD* block diagram and core graph, with communication BW annotated (in MB/s) and its mapping onto mesh and torus topologies. (a) *VOPD* block diagram, (b) *VOPD* graph, (c) mesh mapping, and (d) torus mapping.

shown in Fig. 4. An example mapping of the *VOPD* core graph onto mesh and torus topology graphs is shown in Figs. 2c and 2d.

The communication between each pair of cores (i.e., each edge $e_{i,j} \in E$) is treated as a flow of single commodity, represented as $\{d_k, k = 1, 2, \cdots, |E|\}$. The value of d_k represents the bandwidth of communication across the edge and is denoted by $vl(d_k)$. The set of all commodities is represented by D and is defined as:

$$D = \left\{ \begin{array}{l} d_k : vl(d_k) = comm_{i,j}, k = 1, 2, \cdots, |E|, \forall i, j : e_{i,j} \in E, \\ \text{with } source(d_k) = map(v_i), dest(d_k) = map(v_j). \end{array} \right\} \quad (2)$$

As an example, in Fig. 2b, the communication between `vld` and `rld` is represented by a single commodity, d_1, with value $vl(d_1)$ equal to 70, $source(d_1)$ representing `vld` and $dest(d_1)$ representing `rld`.

4.1 General Minimum-Path Mapping Algorithm

In this section, we present the general mapping algorithm, and in the next sections, we show how the algorithm is adapted for each topology. `NetChip` supports different routing functions: *dimension-ordered*, *minimum-path*, *traffic splitting across minimum-paths*, and *traffic splitting across all paths*. In dimension-ordered and minimum-path routing, the communication between every pair of cores takes place through a single path. Splitting the traffic across multiple paths reduces the bandwidth requirements of network links. In traffic splitting across minimum paths, the communication between cores is spread only across the minimum paths between them. Clearly, this is a special case of the all-path traffic splitting. The advantage of this scheme is that it has bandwidth requirements which are intermediate to that of minimum-path routing and all-path traffic splitting. Also, traffic streams across different paths have the same hop delay, thereby reducing the jitter associated with traffic splitting.

As the graph mapping problem is a special case of the quadratic assignment problem, which is intractable [22], [29], we use a heuristic approach with three phases:

1. An initial mapping is obtained using a greedy algorithm.
2. For minimum-path routing, the minimum-paths and mapping costs are computed. When the routing function is traffic splitting, the paths are obtained by solving a system of *Multi-Commodity Flow (MCF)* equations [26].
3. The solution is iteratively improved by invoking the second phase for every mapping produced by pairwise swapping of vertices.

The minimum-path mapping algorithm is presented in Figs. 3a and 3b. In this paper, we present only mapping algorithms for minimum-path routing and we refer the interested reader to [26] for description of the mapping algorithms for other routing functions. In the initial mapping procedure, first the core that has maximum communication demand is placed onto one of the mesh nodes with maximum number of neighbors. Then, the core that communicates the most with placed cores is chosen. This core is placed onto the NoC node that minimizes the cost function and this procedure is repeated until all the cores are placed.

Once an initial mapping is obtained, in the second phase (steps 2 to 8 in Fig. 3), the commodities are sorted in decreasing order of their values. Then, for each commodity in order, a *quadrant graph* between the source and destination of the commodity is formed, as the shortest path between the source and destination lies within the quadrant between them. The shaded regions in Figs. 2c and 2d are examples of quadrant graphs for the communication between the cores `smem` and `iquant`. The procedure for forming quadrant graphs is presented in Section 4.3. Then, Dijkstra's shortest path algorithm is applied (step 5) to the quadrant graph and the minimum path is obtained. The edge weights are incremented suitably and the procedure is repeated for each commodity in order. After routing all commodities, if the bandwidth and area constraints are satisfied, the cost of communication is calculated. Bandwidth constraints are satisfied, if in the resulting mapping, the traffic across any link is smaller than or equal to the capacity of the link.[1] The area constraints are satisfied when the mapped design area is lower than the maximum allowed area and aspect ratios of the design and soft core blocks (blocks that have flexible sizes) are within permissible ranges. For the area-power estimates, area-power models of the switches and floorplanner are incorporated into `NetChip` as explained in Section 5. The mapping algorithms can have many different objectives such as minimizing average hop delay, area, or power dissipation and is an input parameter to `NetChip`. Depending on the objective function, the cost function calculation (done as part of step 8) varies.

1. Capacity of a link in an NoC is technology and implementation dependent and is assumed as an input to `NetChip`.

```
Initialmapping(G(V,E), P(U,F))
{
  1 assign vertex with max comm in G(V,E)
    to maxs;
  2 assign vertex with max neighbors in P(U,F)
    to maxt;
  3 map(maxs) = maxt;
  4 mark maxt as unavailable and maxs as mapped;
  5 while (not all vertices in G mapped)
    {
  6   choose the unmapped vertex in G(V,E), maxs
      with max comm with already mapped vertices;
  7   find the best available vertex, maxt in P(U,F) with
      min comm cost when maxs is mapped onto it;
  8   map(maxs) = maxt;
  9   mark maxt as unavailable and maxs as mapped;
    }
 10 return the initial mapping;
}
```

(a)

```
Mapping (G(V,E), P(U,F))
{
  1 Initialmapping(G(V,E),P(U,F));
  2 sort commodities in D with decrasing comm
    costs;
  3 for each d_k in D in order of decreasing cost
    {
  4       make quadrant graph Q(d_k) with source(d_k)
          and dest(d_k) as end vertices;
  5       Path(source(d_k), dest(d_k)) = minpath(Q(d_k));
  6       Increase edge weigths in Path by vl(d_k);
    }
  7 floorplan_area_power_estimates(P);
  8 if bandwidth and area constraints are satisfied,
    find the mapping cost;
  9 repeat steps 2 to 8 for each pair–wise swap of
    vertices in P;
 10 return the mapping with lowest cost of all evaluated
    mappings;
}
```

(b)

Fig. 3. Minimum-path mapping algorithms. (a) Initial mapping procedure and (b) minimum-path mapping procedure.

In the last phase of the algorithm (steps 9-10), for each pair-wise swapping of vertices, phase-2 is repeated. Finally, the best mapping from all evaluated mappings is returned by the procedure. The best mappings of VOPD onto mesh and torus topologies for the design objective of minimizing power is presented in Figs. 2c and 2d.

As the minimum-path computations are performed on the quadrant graph instead of the entire NoC graph, large computational time savings is achieved, as the number of nodes in a quadrant graph is much smaller than the total NoC nodes. The runtime of the algorithm is just a few minutes even for large applications (around 100 nodes) on a 1 GHz SUN workstation. The detailed comparison of this algorithm (applied to mesh topology mapping) with existing algorithms is presented in [26].

In the mapping algorithm presented above, all but two of the steps are common to all topologies: The first step is the formation of NoC topology graph, which is obviously specific to a particular topology, and the second step is the procedure used to form quadrant graphs, which varies with the topology used. These two steps are explained in detail in the following sections.

4.2 NoC Topology Graph Definition

The formal definition of the NoC topology graph was presented in the beginning of this section. The edges in the NoC graph represent connections between adjacent NoC nodes. Thus, for defining a topology graph, we simply need to define the nodes that are adjacent to a particular node in that topology.

For a mesh, each node, except the nodes on the edges, have four neighbors, nodes in the four corners have two neighbors, and other nodes in the edges have three neighbors. A torus has similar structure as the mesh, but has additional wrap-around channels between the edge nodes.

Example 1. In the mesh topology shown in Fig. 4a, corner nodes such as node 0 has two neighbors, edge nodes such as node 1 has three neighbors, and other nodes such as node 4 has four neighbors. The wrap-around channels of a torus are shown in Fig. 4b (such as the channels

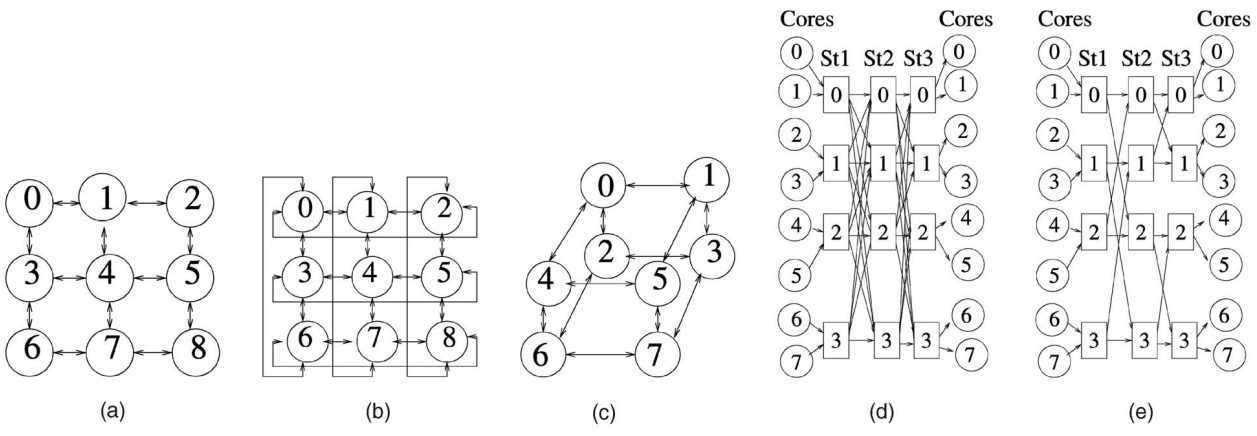

Fig. 4. NoC topologies. (a) Mesh, (b) torus, (c) hypercube, (d) 3-stage Clos, and (e) butterfly.

connecting node 0 with nodes 2 and 6).

For a hypercube with N nodes (also called as 2-ary n-cube), each node has n (which is equal to $\log_2 N$) neighbors. A node u_i in such a network is represented by the n-tuple: $(h_1, h_2, \ldots, h_n)$, which is the binary representation of the decimal i. Intuitively, each h_j represents a single dimension and, thus, an n-tuple uniquely identifies every node of a 2-ary n-cube.

Example 2. Node 2 in Fig. 4c is represented by $(0, 1, 0)$. All nodes with n-tuples distance 1 apart from the n-tuple for u_i are neighbors of u_i (node 6 whose 3-tuple is $(1, 1, 0)$ is adjacent to 2).

For Clos networks (for the sake of simplicity, we restrict to three-stage Clos networks), each switch in a stage is connected to every switch in the next stage (e.g., switch 0 of stage 1 in Fig. 4d is connected to switches $0, 1, 2, 3$ of stage 2). Thus, adjacency calculations are simple for this topology. Butterfly networks (with N core nodes) are also known as k-ary n-fly networks, where k is the radix of switches in the network, and n is the number of stages in the network ($n = \log_k N$). A 2-ary 3-fly network is presented in Fig. 4e. As seen, the switches in each stage are connected to two switches in the next stage. The maximum distance between adjacent switches halves with each stage (e.g., switch 0 of stage 1 is connected to switches 0 and 2 of stage 2, resulting in a maximum distance of 2. Switch 0 of second stage is connected to switches 0 and 1 of third stage, thus resulting in a maximum distance of 1).

4.3 Quadrant Graph Formation

The procedure for forming quadrant graphs is specific to a topology as the nodes that lie in the shortest path of a commodity is topology specific. Example quadrant graphs for mesh and torus networks were presented in Figs. 2c and 2d (shaded areas in the figure). For a mesh network, the nodes that are within the bounding box formed by the row and column boundaries of the source and destination nodes of a commodity form the elements of the quadrant graph of that commodity (Fig. 2c). For a torus network, the wrap-around channels need to be considered for computing the smallest bounding box between the source and destination nodes (Fig. 2d).

For hypercubes, all nodes that have matching h_j values (of the n-tuple) as that of the source and destination nodes of a commodity are included in the quadrant graph. As an example, for source node 0 (represented by (0,0,0)) and destination 3 (represented by (0,1,1)), all nodes with n-tuples of the form (0,*,*) (where * represents do not care values), form the quadrant graph (i.e., nodes 0,1,2,3 are the elements of the quadrant graph).

As Clos networks have full interconnection pattern between switches of adjacent stages and butterfly networks have no path diversity (a single path from any source to any destination), the quadrant graph formation for these networks is simple.

5 Area-Power Models and Floorplanning

We developed analytical models to estimate the area of switches. We assume that the switches are based on the ×pipes architecture. The area calculations include the crossbar area, buffer area, logic (including control) area, and include fine granularity of details (like accounting for pipeline registers, cross points, etc). We used ORION [37], a power modeling tool, for developing bit energy models for the switches. The area-power models are developed for various switch configurations for different technology parameters (in the rest of this paper, we assume 0.1μ technology). The area and energy consumption for some example switch configurations are presented in Table 1. We use wiring parameters from [38] to estimate link power dissipation. We assume that the area-power values of the cores are an input to our tool.

TABLE 1
Switch Area and Energy

Size (in×out)	Area (mm^2)	Energy (pJ/bit)
1×1	0.018	7.08
2×2	0.037	21.94
3×3	0.08	45.96
4×4	0.10	79.08
8×8	0.74	313.04

In order to estimate the area and link power dissipation for a mapping, we need to know the exact size and position of the cores and length of links. For this purpose, we have implemented a floorplanner in `NetChip`. We assume there are two types of blocks (the mapped cores are referred to as blocks in the floorplanning formulation): *hard blocks* whose dimensions are known prior to floorplanning and *soft blocks* that have fixed area but unknown dimensions. In this work, we assume the blocks to be rectangular in shape. For soft blocks, the maximum and minimum permissible aspect ratios (the ratio of width to length of a block) are inputs to the floorplanner. The general solution to the floorplanning problem has two basic steps: first is finding the relative position of modules and the second is finding the exact position, area, and size of the modules [28]. For a particular mapping that needs to be evaluated for area-power-delay, the relative positions of the cores are known. Thus, the floorplanning problem is reduced to one of finding the exact position and size of the cores and switches. We use a simple *Linear Program* (LP)-based floorplanner existing in literature [33] for this purpose. For the floorplanning problem, we assume a greedy 2D mapping of higher dimensional topologies (such as the 2-ary n-cube). Note that a more sophisticated floorplanner such as the one presented in [28] can be used in place of the simple floorplanner and the floorplanner can consider specific features of NoCs.

The area and aspect ratio constraints (for feasibility of mapping) are evaluated and link lengths in the NoC are obtained from the floorplanner. Using the built-in power models, power dissipation for the switches and links are calculated based on the average traffic (shown as edge annotations in Fig. 2b) through them. The computed area, power values are returned (step 7 in Fig. 3b) to the mapping algorithm.

6 ×Pipes

Once the best communication architecture has been selected following the aforementioned steps, the customized NoC configuration is generated by the `×pipesCompiler`. `×pipesCompiler` uses the ×pipes library, which consists of highly parameterized network components that can be tailored to the communication needs of the selected

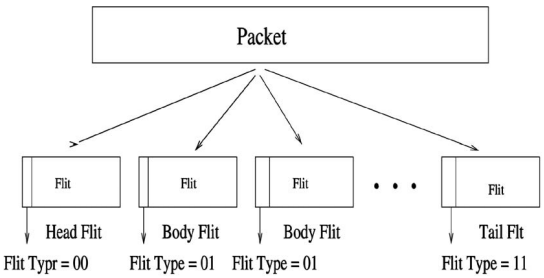

Fig. 5. Packet partitioning into flits.

architecture. This section provides the details of the basic building blocks of the ×pipes library. The custom-tailored generation of network component instances (operated by ×pipesCompiler) will be addressed in the next section.

The ×pipes components target heterogeneous packet-switched NoCs, thanks to the aggressive design of the network components for high performance and to their instantiation time flexibility. The high degree of parameterization of ×pipes components is achieved by using both global network-specific parameters and local block-specific parameters. The former ones include flit size, degree of redundancy of error control logic, address space of the cores, maximum number of hops between any two nodes, maximum number of bits allocated with in a packet for end-to-end flow control, etc. On the other hand, specific parameters of the network interface are: type of interface (master, slave, or both), flit buffer size at the output port, content of routing tables for source-based routing, other interface parameters to the cores such as number of address/data lines, maximum burst length, etc. Parameterization of the switches mainly regards the number of their I/O ports, the number of virtual channels for each physical output link, and the link buffer size. Finally, the length of each individual link can be specified in terms of number of repeater stages, as will be discussed next.

The ×pipes network interface uses OCP as point-to-point communication protocol with the cores, and takes care of protocol conversion to adapt to the network protocol. Data packetization results in the packet partitioning procedure illustrated in Fig. 5. A flit type field allows to identify the head and the tail flit, and to distinguish between header and payload flits.

The NoC backbone relies on wormhole switching and static routing. Routes are obtained by the network interface by accessing a look-up table based on the destination address. Each route is represented by a set of direction bits. Each switch directs the flits belonging to a certain packet to the particular output port, based on the direction bits. This routing algorithm allows a lightweight switch implementation as no dynamic decisions have to be taken at the switching nodes.

One relevant ×pipes feature is the support for communication reliability. It is achieved by means of distributed error detection with link-level retransmission as error recovery technique. Although a distributed approach to error detection causes a higher area overhead at the network nodes compared with an end-to-end solution, the effects of error propagation are reduced, for instance, preventing packets with corrupted header from being directed to the wrong path. In order to counterbalance this overhead, error detection with retransmission of incorrectly received data was preferred to error correction, since the latter technique requires complex energy-inefficient decoders. If the bit error rate is not high, average performance penalty caused by retransmissions as perceived from the application can be neglected and error detection schemes have been showed to minimize average energy-per-bit [6]; as a consequence, they were the preferred choice to provide communication reliability in ×pipes. The retransmission policy implemented in ×pipes is GO-BACK-N. Flits are transmitted continuously and the transmitter does not wait for an ACK after sending a flit. Such an ACK is received after a round-trip delay. When a NACK is received, the transmitter backs up to the flit that is negatively acknowledged and resends it in addition to the N succeeding flits that were transmitted during the round-trip delay. At the receiver, the N-1 received flits following the corrupted one are discarded regardless of whether they were received error-free or not. GO-BACK-N trades off inefficiency in bus usage (retransmission of many error-free flits) with a moderate implementation cost, and was preferred to a selective-repeat scheme, wherein only those flits that are negatively acknowledged are retransmitted but more buffering resources are required at the destination switch.

In ×pipes switches, different error detecting decoders can be instantiated at design time, depending on their latency and redundancy (additional link parity lines) overhead and on their error detecting capability, that has to be compared with the estimated (technology-dependent) bit error rate and the required mean time to failure (MTTF) for a specific implementation. The considered error detecting codes consist of several versions of the Hamming code and of the Cyclic Redundancy Check (CRC) code, each one characterized by a different minimum distance and, hence, error detection capability. The ultimate objective is to select at design time, for a specified supply voltage and flit size, the coding schemes that provide a MTTF of at least one year and, among them, the scheme that minimizes decoding latency and/or redundancy, based on the most critical design constraint.

Finally, it is worth noting that the support for high performance communication was provided in ×pipes by means of proper design techniques for the basic network components (such as link pipelining, deeply pipelined switches, and latency insensitive component design). The architecture of the network components (links, switches, and network interfaces) is presented in the following sections.

6.1 Network Link

A solution to overcome the interconnect-delay problem consists of pipelining interconnects [40], [41]: The link data introduction rate can be decoupled from link delay by changing this latter into latency. This requires the system to be made of modules whose behavior does not depend on the latency of the communication channels (latency insensitive operation) [40].

×pipes interconnect takes this approach. Switch-to-switch links are subdivided into basic segments whose length guarantees that the desired clock frequency (up to the maximum speed achievable by a certain technology) can be used and that the system operating frequency is not bound by the delay of the longest link. Depending on the specific link length, a certain number of clock cycles is needed by a flit to cross that interconnect. Fig. 6 illustrates the link model, which is equivalent to a pipelined shift register. Pipelining has been used both for data and control lines.

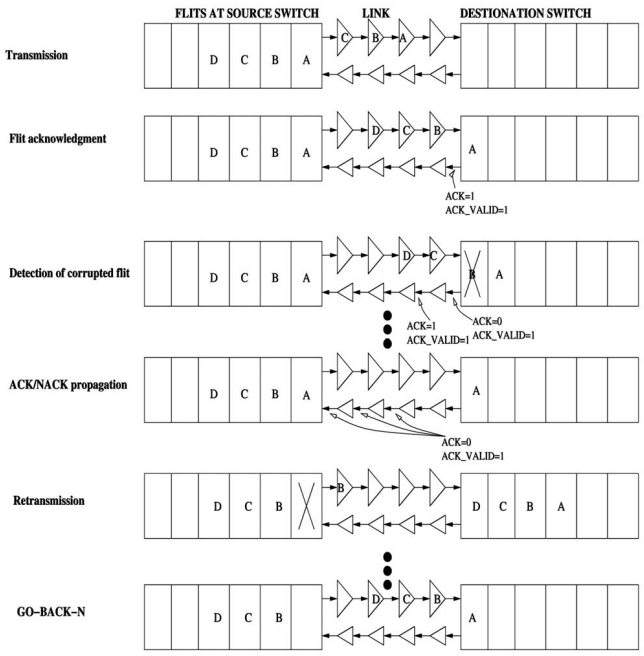

Fig. 6. Pipelined link model and latency-insensitive link-level error control.

The figure also illustrates that pipelined links affect operation of link-level error control, making it latency-insensitive. The retransmission of a corrupted flit between two successive switches is represented. Multiple outstanding flits propagate across the link during the same clock cycle. When flits are correctly received at the destination switch, an ACK is propagated back to the source, and after N clock cycles (where N is the length of the link expressed as number of repeater stages) the flit will be discarded from the buffer of the source switch. On the contrary, a corrupted flit is NACKed and will be retransmitted in due time.

By means of a proper buffering policy, network switches have been designed in such a way that their functional correctness depends on the flit arriving order and not on their exact timing, so that input links of the switches can be different and of any length. These design choices are at the basis of latency insensitive operation of the overall NoC and allow the construction of an arbitrary network topology and, hence, support for heterogeneous architectures.

6.2 Switch

A schematic representation of a ×pipes switch is illustrated in Fig. 7a. The example configuration exhibits four inputs, four outputs, and two virtual channels multiplexed across the same physical output link. Switch operation is latency insensitive, in that correct operation is guaranteed for arbitrary link pipeline depth.

For latency insensitive operation, the switch has virtual channel registers to store $2N + M$ flits, where N is the link length (expressed as number of basic repeater stages) and M is a switch architecture related contribution (12 cycles in this design). The reason is that each transmitted flit has to be acknowledged before being discarded from the buffer. Before an ACK is received, the flit has to travel across the link N cycles, an ACK/NACK decision has to be taken at the destination switch (a portion of M cycles), the ACK/NACK signal has to be propagated back (N cycles) and recognized by the source switch (remaining portion of M cycles). During this time, other $2N + M$ flits have been transmitted but not yet ACKed.

Output buffering was chosen for ×pipes switches, and the resulting architecture consists of multiple replications of the same output module reported in Fig. 7b, one for each switch output port. All switch inputs are connected to the inputs of each output module. Flow-control signals generated by each module (such as ACK and NACK for incoming flits) are collected by a centralized switch unit, that directs them back to the proper source switch. As can be observed in Fig. 7b, each output module is deeply pipelined (seven

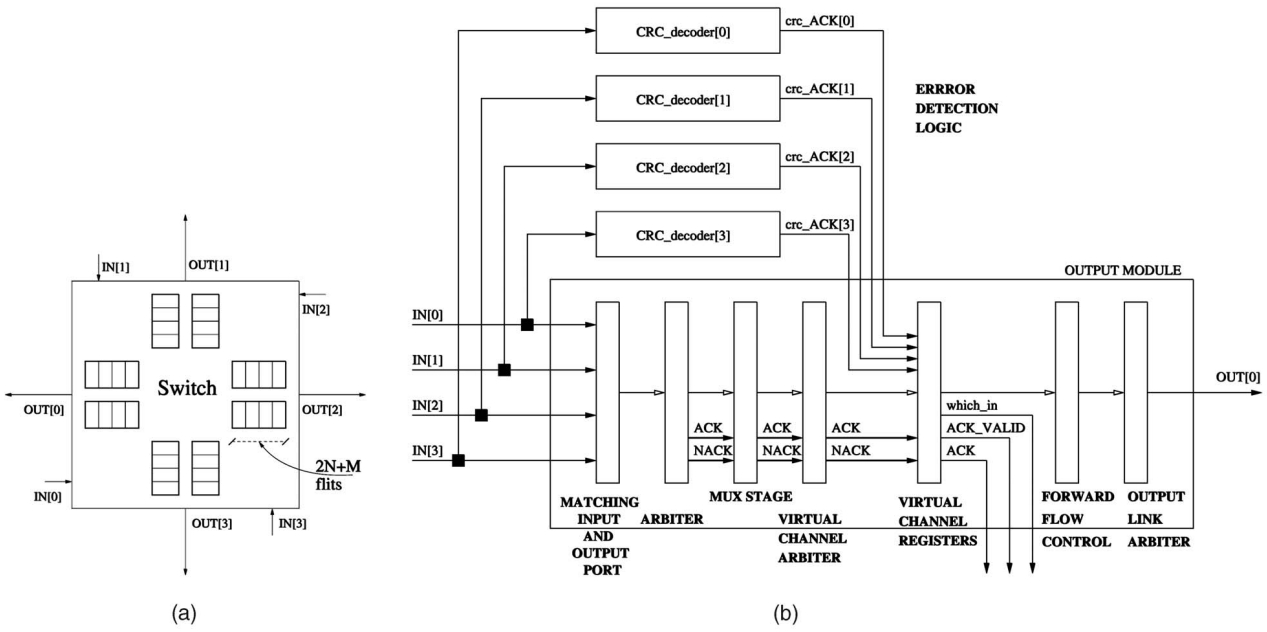

Fig. 7. ×pipes switch architecture. (a) Example of 4×4 switch configuration with two virtual channels and (b) architecture of the output module for each ×pipes switch output port.

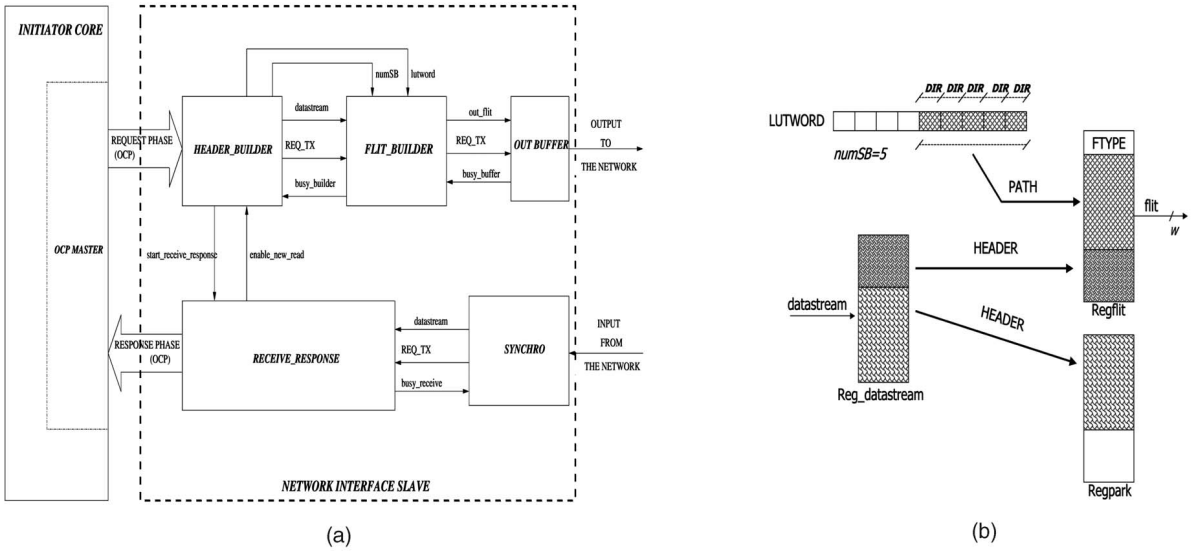

Fig. 8. `×pipes` network interface and the mechanism for building flits. (a) Architecture of `×pipes` network interface slave and (b) mechanism for building header flits.

pipeline stages) so to maximize the operating clock frequency of the switch. The CRC decoders for error detection work in parallel with the switch operation, thereby hiding their latency.

The first pipeline stage checks the header of incoming packets on the different input ports to determine whether those packets have to be routed through the output port under consideration. Only matching packets are forwarded to the second stage, which resolves contention based on a round robin policy. Arbitration is carried out when the tail flit of the preceding packet is received, so that all other flits of a packet can be propagated without contention related delay at this stage. A NACK for flits of nonselected packets is generated. The third stage is just a multiplexer, which selects the prioritized input port. The following arbitration stage keeps the status of virtual channel registers and determines whether flits can be stored into the registers or not. A header flit is sent to the register with more free locations, followed by successive flits of the same packet. The fifth stage is the actual buffering stage, and the ACK/NACK response at this stage indicates whether a flit has been successfully stored or not. The following stage takes care of forward flow control: A flit is transmitted to the next switch only when adequate free register locations are available at the output port of interest of the destination switch. Finally, a last arbitration stage multiplexes flits from the virtual channels on the physical output link on a round-robin basis, thereby improving network throughput.

6.3 Network Interface

The `×pipes` network interface (*NI*) provides a standardized OCP-based interface to networked cores. The NI for cores that initiate communication (initiators) needs to turn OCP-compliant transactions into packets to be transmitted across the network. It represents the slave side of an OCP end-to-end connection (the master side being the initiator core), and it is therefore referred to as network interface slave (NIS). Its architecture is shown Fig. 8a. The NIS has to assemble the packet header, which has to be spread over a variable number of flits depending on the length of the path to the destination node. The look-up table containing static routing information is accessed by the *HEADER_ BUILDER* block and the destination address is used as table entry. In this way, two routing fields are extracted: *numSB* (number of hops to destination) and *lutword* (actual direction bits read from the look-up table). Together with the core transaction related information (*datastream*), they are forwarded to the *FLIT_BUILDER* block, provided the enable signal *busy_builder* is not asserted. Based on the input data, module *FLIT_BUILDER* has the task of building the flits to be transmitted via the output buffer *OUT_BUFFER*, according to the mechanism illustrated in Fig. 8b. Let us assume that a packet requires $numSB = 5$ hops to get to destination, and that the direction to be taken at each switch is expressed by *DIR*. Module *FLIT_BUILDER* builds the first flit by concatenating the flit type field with path information. If there is some space left in the flit, it is filled with header information derived from the input datastream. The unused part of the datastream is stored in a regpark register, so that a new datastream can be read from the *HEADER_BUILDER* block. The following header and/or payload flits will be formed by combining data stored in *regpark* and *reg_datastream*. No partially filled flits are transmitted to make transmission more efficient. Finally, module *OUT_BUFFER* stores flits to be sent across the network, and allows the NIS to keep preparing successive flits also when the network is congested. The size of this buffer is a design parameter. The response phase is carried out by means of two modules. *SYNCHRO* receives incoming flits and reads out only useful information (e.g., it discards routing fields). At the same time, it contains buffering resources to synchronize the network's requests to transmit remaining flits of a packet with the core consuming rate. The *RECEIVE_RESPONSE* module translates useful header and payload information into OCP-compliant response fields. When a read transaction is initiated by the initiator core, the *HEADER_BUILDER* block asserts a *start_receive_response* signal that triggers the waiting phase of the *RECEIVE_RESPONSE* module for the requested data. As a consequence, the NIS supports only one outstanding read operation to keep interface complexity low. Although no read after read transactions can be

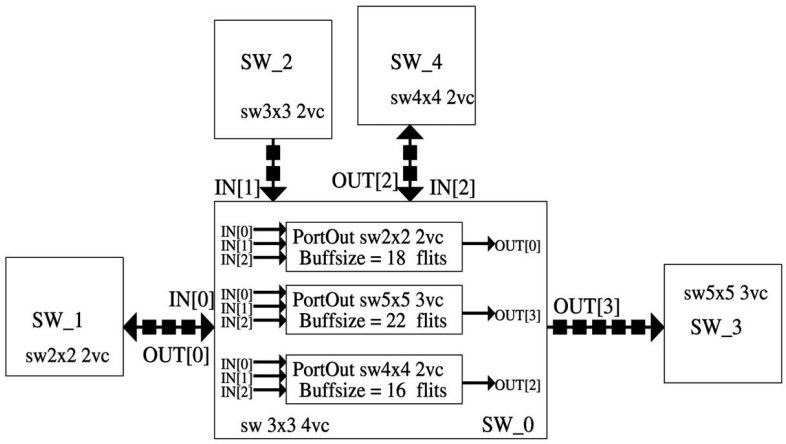

Fig. 9. Irregular n/w optimizations.

initiated unless the previous one has completed, an indefinite number of write transactions can be carried out after an outstanding read or write has been initiated. The latency incurred by the interface (sum of the source and destination interface delay) is 10 cycles. The architecture of a network interface master is similar to the one just described.

7 The ×pipesCompiler

Conceptually, the ×pipesCompiler generates a SystemC description of the NoC after the network topology has been chosen. The input to the compiler is a high-level description of the NoC topology, provided either automatically by SUNMAP or manually by the designer. The high-level description consists of the definition of the cores, network interfaces, switches, links, and their interconnections. The number of pipeline stages of each link is also dictated, based on link length estimations by the floorplanner and target clock speed. The ×pipes library of SystemC soft macros described in the previous section is used by ×pipesCompiler to generate the network components for the chosen topology.

The output is a SystemC hierarchical description of all the switches, links, network nodes and interfaces that specifies their topological connectivity. The structure of the SystemC output can be optimized either for simulation or synthesis. In the latter case, the programming constructs utilized are constrained to a specific synthesizable SystemC subset. The final description can be compiled and simulated at the cycle-accurate and signal-accurate level, and can be synthesized by back-end RTL synthesis tools for silicon implementation.

7.1 Input Specification and Parsing

The ×pipesCompiler input comprises the design description file together with the routing table information for each core on the NoC. The design file describes the cores, switches, links, and the interconnections between them. From the designer's viewpoint, the implementation of the NI that connects a core to the NoC is transparent. The ×pipesCompiler will instantiate the needed NIs according to the type of the core (Master, Slave, Master/Slave).

Each core is identified by its name, SystemC file, its type (master, slave or master/slave), the switch and port to which it is connected, and its routing table information file. Each switch is identified by its name, number of I/O ports and the number of the virtual channels of each output. Each link is identified by its name, the names and ports of the source and destination switches that it connects, and the number of repeaters used in the link. For every core, the routing table utilized by it is described in a separate file. More precisely, all paths through which a core communicates with the rest of the cores in the selected topology are explicitly given.

7.2 Network Instantiation

A tree data structure is generated upon parsing the input file description. Each node in the tree denotes a specific object of the design. This structure is then recursively optimized in order to remove redundancies and later on utilized to generate the SystemC output.

Initially, the different types of switches, links, and network interfaces that are required in the design are identified. In simulation mode, a single template class is output for each type. Multiple objects are instantiated as needed.

During network instantiation, the ×pipesCompiler performs several optimizations to remove redundant logic from the generic library components. If, for example, a switch has only one input link connected to a port, the logic and buffers of the corresponding nonexisting output port will not be generated. In the case of a custom-made irregular design, this valuable optimization drastically reduces hardware complexity. The optimized network component classes are stored into arrays of structures.

During output generation, the tree structure is recursively parsed starting from the leaves and the class corresponding to each node is generated. The root of the structure represents the main object file (main.cc) that instantiates all objects of the design at runtime. The signals needed to connect the objects according to the design description file are also defined there.

If possible, the ×pipesCompiler will share the signals that are common to all objects. In order to automate tracing of signals, a debugging mode has been implemented, that enables monitoring of any signal in the design.

Example 3. Fig. 9 clarifies how a network is optimized by the removal of redundant logic during network instantiation in a simple topology. Since port[1] of SW_0 has only a single input connection the ×pipesCompiler will not instantiate a PortOUT Block that would be connected to

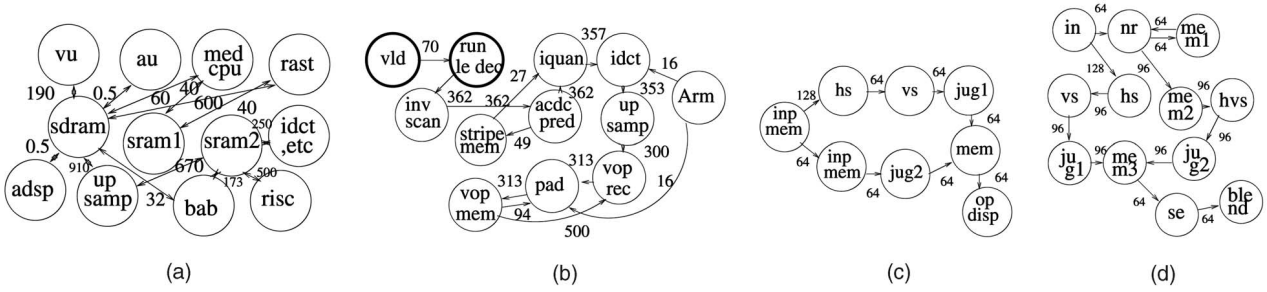

Fig. 10. Core graphs of video processing applications. (a) *MPEG4* core graph, (b) *VOPD* core graph, (c) *PIP* core graph, and (d) *MWD* core graph.

the corresponding output. Similar optimization is performed for an output-only port like `port[3]`. In each `PortOUT Block` generated, the signals related to this nonexisting input are removed from the generic template files. The `×pipesCompiler` will also optimize the size of each output buffer according to the number of repeaters on each output link.

Execution times of the `×pipesCompiler` depend on the design characteristics. A regular topology, for example, such as a 16×16 mesh, can be generated faster than an irregular, application-specific topology with only few cores and switches. In absolute terms, execution time is not a major consideration. For a small custom design with four switches and two cores, the execution time is about 2 seconds on a 1.8GHz Pentium 4. In all of the experiments performed, network optimization and generation was completed within a few minutes.

8 Experimental Validation and Case Studies

In this section, we present experimental case studies of `NetChip`, exploring various functionalities of the tool. In Sections 8.1 and 8.2, we present the mappings produced by `NetChip` for two types of applications: video applications and network processor applications. As the communication pattern of these applications are different, the topologies produced have different characteristics. We then present the design-exploration capabilities of `NetChip` in Section 8.3. `NetChip` can also be used to generate custom topologies and this is presented in Section 8.4. In the last section, a DSP application designed in SystemC is used to validate `NetChip`.

8.1 Video Applications

We applied `NetChip` onto four different video processing applications: *Video Object Plane Decoder* (*VOPD*-mapped onto 12 cores), *MPEG4 decoder* (14 cores), *Picture-In-Picture* application (*PIP*-mapped onto eight cores), and *Multi-Window Display* application (*MWD*-mapped onto 14 cores). These are high-end video-processing applications and the hardware-software partitioning of the applications is presented in [16], [17]. The core graphs of these applications are presented in Fig. 10. The maximum link bandwidth for the NoCs is conservatively assumed to be 500 MB/s.

The results of mapping *VOPD* onto various topologies are presented in Fig. 11. As seen from Fig. 11a, the butterfly topology (`4-ary 2-fly`) has the least average hop delay out of all topologies. The lower hop delay is due to the fact that a `4-ary 2-fly` has two stages of switches, which means an average delay of two hops for all communication. Mesh, torus, and hypercube networks have a higher average hop delay as the least possible hop delay (that of adjacent nodes) itself is two and it was not possible to place all communicating nodes adjacent to each other. As the Clos network has three stages, the average hop delay is three. The area, power estimates for the topologies are presented in Figs. 11c and 11d. As seen from Fig. 11b, the butterfly topology has the least number of switches, but has more links when compared to mesh, torus, or hypercube. The large power savings achieved by the butterfly network (Fig. 11d) is attributed to the fact that there are fewer switches and smaller number of hops for communication. Moreover, all the switches are 4×4, while the direct topologies have 5×5 switches. The average link length in the butterfly network (obtained from floorplanner) was observed to be longer than the link lengths (around $1.5 \times$) of direct networks. However, as the link power dissipation is much lower than the switch power dissipation, we get large power savings for the butterfly network. The smaller number of switches and smaller switch sizes also account for the large area savings achieved by the butterfly network. Thus, butterfly is the best topology for *VOPD*. The performance gains for the butterfly over other topologies may be surprising, but after careful inspection we see the reason.

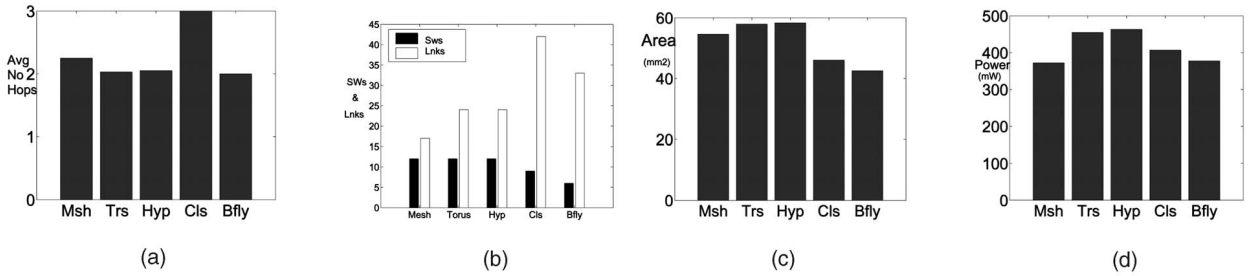

Fig. 11. Mapping characteristics of *VOPD*. (a) Avg hop delay, (b) resource util, (c) design area, and (d) design power.

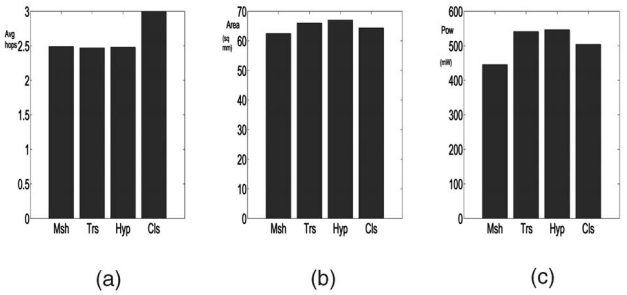

Fig. 12. Mapping characteristics of *MPEG4*. (a) Avg hop delay, (b) design area, and (c) design power.

Butterfly network trades off path diversity for network switches and average hop delay. As the *VOPD* example has lower bandwidth demands compared to *MPEG4*, we are able to satisfy the bandwidth demands of the application using a butterfly network.

The results of mapping *MPEG4* are presented in Fig. 12. As seen from the core graph of *MPEG4* decoder (Fig. 10a), the amount of communication between the cores (such as to/from the shared SDRAM) is much higher than that can be supported by minimum-path routing. When the link capacities are assumed be 500 MB/s, all topologies violate the bandwidth constraints for minimum-path routing. So, we apply multipath routing, splitting the traffic across many paths. As the butterfly topology has no path diversity, traffic cannot be split across multiple paths and, thus, it does not produce any feasible mapping for *MPEG4*. Feasible mappings onto other topologies are obtained and the mapping results are presented in Figs. 12a, 12b, and 12c. The torus topology has slightly lower communication hop delay than the mesh (Fig. 12a) as it utilizes more network resources. However, the mesh topology has large savings in area and power which overshadow the slightly higher hop delay cost. The significant area savings is attributed to the fact that in the torus topology all the switches are of size 5×5 (for communicating with four neighbors and to the core), whereas in the 12-node mesh topology, there are four 3×3 switches, six 4×4 switches, and two 5×5 switches (an example 12-node mesh is shown in Fig. 2c). Although the mesh network has a slightly higher hop delay, which in turn means a larger amount of traffic flow across the switches, the power consumption is significantly lower than the torus topology. This is attributed to the fact that energy dissipation in a switch increases nonlinearly with increase in switch size (refer Table 1). Thus, a mesh topology is more suitable for the *MPEG4* than other topologies.

The results of mapping *PIP* and *MWD* are presented in Tables 2 and 3. For the *PIP* application, mesh topology has the least area and power consumption. The *PIP* has eight cores and we use a 3×3 mesh, with one of the nodes (of the total nine nodes) removed. A 3×3 mesh is more evenly spread, length and breadth-wise, when compared to the 4-ary 2-fly butterfly network (refer Fig. 4). Thus, mesh topology provides more flexibility on the sizes of soft-core blocks without violating the aspect ratio constraints of the design. This results in much efficient floorplan for the mesh topology resulting in significant area savings. In the mesh topology, much of the traffic (30 percent) flows through the 3×3 switches. In the 4-ary 2-fly butterfly topology, all the traffic travels through the 4×4 switches. This results in small power savings for the mesh topology. Thus for the *PIP* application, mesh is the most suitable topology. For the *MWD* application, like the *VOPD*, butterfly topology results in the least area, while the mesh topology provides small power savings. Depending on the design objective, either of the topologies can be chosen for *MWD* mapping.

TABLE 2
PIP Mapping

Top	hop	area mm2	power mW
Msh	2.1	21.06	201.47
Trs	2	30.72	204.62
Hcb	2	30.72	208.02
Cls	3	29.18	208.16
Bfy	2	24.46	204.76

8.2 Network Processor Application

We consider a network processor with 16-nodes, each node having the architecture shown in Fig. 13a, obtained from [8]. The objective of the communication architecture is to provide low contention for the data transfer between the nodes. As Clos networks have maximum path diversity, they have the least congestion for large data flows and are more suitable for the network applications. We validated the need for Clos networks by producing mappings onto various topologies by relaxing the bandwidth constraints and simulating the resulting SystemC design. We use traffic generators to generate adversarial traffic pattern for each topology. As seen from Fig. 13b, where the average packet latency is plotted with increasing traffic injection (which in turn means that the network processor is processing larger amounts of data), Clos topology clearly outperforms other topologies. Moreover, the area and average power dissipation in a Clos network (Figs. 13c and 13d) is only slightly higher than the butterfly topology, justifying its use for network processing applications.

8.3 Design Space Exploration of a Topology

In this section, we explore the *MPEG4* mappings onto a mesh topology. There are two ways in which a chosen topology can be explored: the first is to evaluate the effects of various routing functions and the second is to obtain a set of area-power-performance trade off points for the mappings from which the optimum design point can be chosen.

The minimum bandwidth for different routing functions (DO—Dimension Ordered, MP—Minimum-path, SM—Split-traffic across Minimum-paths, SA—Split-traffic across All paths) is shown in Fig. 13e. When the maximum available link bandwidth is 500 MB/s, only split-traffic routing can be used for mapping *MPEG4*.

TABLE 3
MWD Mapping

Top	hop	area mm2	power mW
Msh	2.1	70.32	321.99
Trs	2	76.8	323.56
Hcb	2.35	80.24	341.62
Cls	3	78.96	333.29
Bfy	2	68.64	329.70

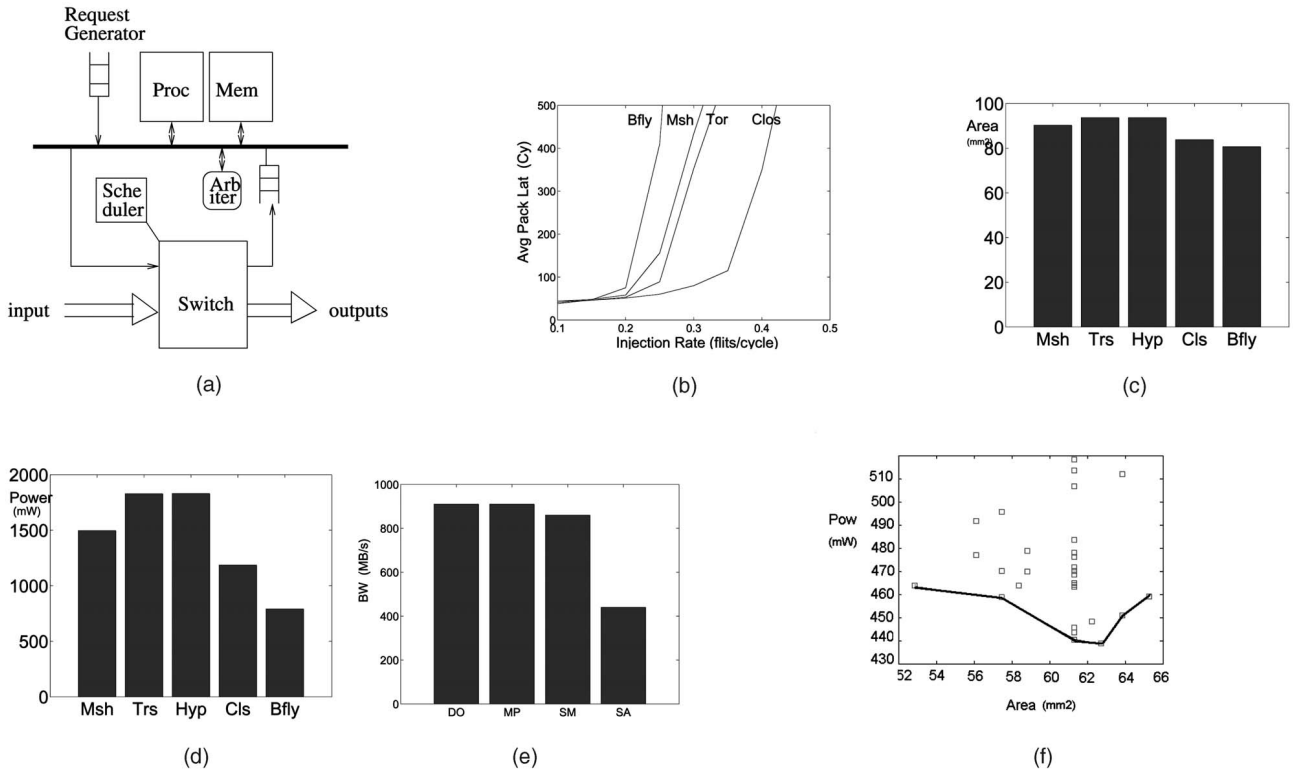

Fig. 13. Network application characteristics ((a)-(d)) and design space exploration of an MPEG4 mapping ((e)-(f)). (a) Node arch, (b) latency, (c) area, (d) power, (e) routing Fn, and (e) area-power plot.

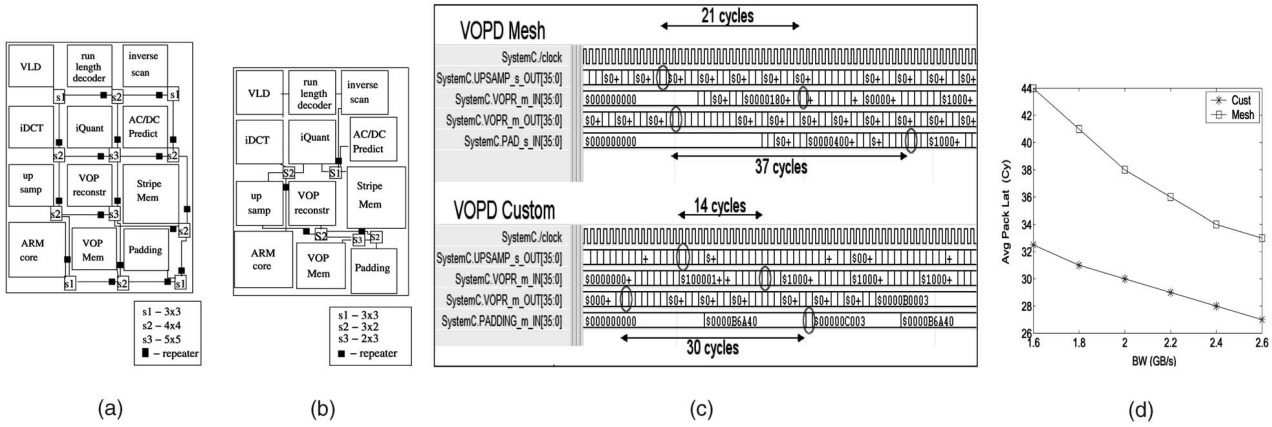

Fig. 14. Mesh and custom mappings of video object plane decoder. (a) Mesh NoC, (b) Appln specific NoC, (c) SystemC output, and (d) VOPD avg lat.

Fig. 13f shows the area-power trade off points in the design space of the mapping from which the optimum design point can be chosen.

8.4 Generating Custom Topologies

An important application of `NetChip` is to construct application-specific custom topologies. With an application-specific network, the designer is faced with the additional task of designing network components (e.g., switches) with different configurations (e.g., different I/Os, virtual channels, buffers) and interconnecting them with links of uneven length. These steps require significant design time and the need to verify network components and their communications for every design. `NetChip` bridges the design gap for building such heterogeneous application-specific NoCs.

TABLE 4
Custom and Mesh Mappings of VOPD

Param	Mesh	Custom	Ratio
Area (mm^2)	1.26	0.22	5.73
Power (mW)	108.74	40.08	2.71

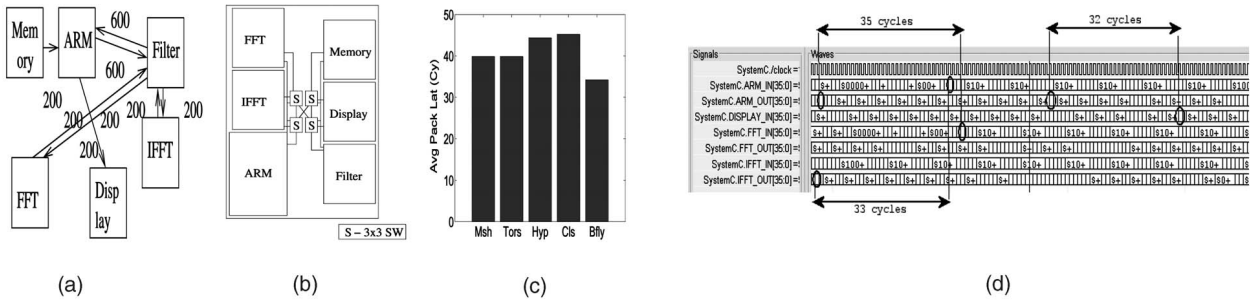

Fig. 15. Mapping of DSP application. (a) Filter appln, (b) bfly floorplan, (c) SystemC plots, and (d) SystemC snapshot.

The designer can invoke ×pipesCompiler to assemble a custom NoC, which is described using the *GUI*. A custom NoC for the VOPD is presented in Fig. 14b. In the VOPD, about half the cores communicate to more than a single core. This motivates the configuration of this custom NoC, having less than half the number of switches than the mesh NoC. The results of mesh and custom mappings of VOPD is presented in Table 4. By using the area-power models built into NetChip, the area and power consumption of the network components of the custom NoC were automatically obtained. Significant area and power improvements are obtained with the custom NoC, as fewer number of switches are used and the switches have smaller size than the mesh switches.

We performed cycle-accurate simulation of the SystemC models of the NoCs generated by the NetChip for the *VOPD*. We use two-state Markov Models as stochastic traffic generators to model the bursty nature of the application traffic, with average communication bandwidth matching the applications' average communication bandwidth. Snapshots of SystemC simulations of mesh and custom NoCs for some of the cores of VOPD are shown in Fig. 14c. The time between transmission of a flit and its reception, which includes the switch delay, link delay, and contention delay, is marked in the figure. The variation of average packet latency (for 64 byte packets, 32 bit flits, and 7 cycle switch delay) with link bandwidth is shown in Fig. 14d. Application-specific NoCs have lower packet latency as the average number of switch and link traversals is lower. Moreover, the latency increases more rapidly for the mesh NoCs with decrease in bandwidth. With the custom NoC, we achieve an average of 25 percent savings in latency (see Fig. 14d).

8.5 DSP Application and SystemC Simulations

We applied NetChip to a DSP Filter design with six cores (refer Fig. 15a). The cores are modeled in SystemC and the design is simulated at the transaction level. The resulting core graph is used by the NetChip which produces mappings onto the butterfly topology (Fig. 15b). Then, the network components for the butterfly topology are automatically generated and the resulting NoC design of the DSP is simulated at cycle accurate and signal accurate level in SystemC. A snap-shot of the SystemC simulation is shown in Fig. 15d. We also generated the best mappings of other topologies for comparison purposes. The SystemC simulation of all topologies is carried out, and the observed average packet latency for the topologies plotted (shown in Fig. 15c). As seen from figure, the butterfly topology indeed has the minimum latency.

For all these applications, NoC selection and generation was obtained in few minutes on a 1GHz SUN workstation. The SystemC simulations were also checked for functional and timing correctness validating the output of NetChip.

9 Conclusion

While the communication needs of a high-speed general purpose on-chip multiprocessor can be accommodated on a homogeneous fabric, heterogeneous structures are required for application-specific computing systems, wherein a set of heterogeneous computing and storage resources have to be interconnected by means of custom-tailored NoCs. In this case, an ad hoc NoC synthesis flow must include selection of the most suitable topology for a certain application domain, mapping of cores onto that topology and generation of the resulting network instance. This paper presents a NoC synthesis framework, called NetChip, wherein these issues are addressed and automated by means of proper tools, thus bridging an important design gap in building NoCs. A library of highly parameterized, design time composable network building blocks (×pipes) is at the core of the proposed design methodology. The entire design flow has been tested with several experimental case studies, showing the rich design space exploration capabilities of this framework.

Acknowledgments

This research is supported by MARCO Gigascale Systems Research Center (GSRC) and the US National Science Foundation (under contract CCR-0305718).

References

[1] M. Sgroi et al., "Addressing the System-on-a-Chip Interconnect Woes through Communication-Based Design," *Proc. Design Automation Conf.*, pp. 667-672, 2001.

[2] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, pp. 70-78, Jan. 2002.

[3] P. Guerrier and A. Greiner, "A Generic Architecture for On-Chip Packet Switched Interconnections," *Proc. DATE 2000*, pp. 250-256, Mar. 2000.

[4] F. Boekhorst, "Ambient Intelligence, the Next Paradigm for Consumer Electronics: How Will it Affect Silicon," *Proc. Int'l Solid State Circuits Conf. 2002*, pp. 28-31, Feb. 2002.

[5] S. Kumar et al., "A Network on Chip Architecture and Design Methodology," *Proc. Int'l Symp. VLSI 2002*, pp. 105-112, Apr. 2002.

[6] D. Bertozzi, L. Benini, and G. De Micheli, "Low Power Error Resilient Encoding for On-Chip Data Buses," *Proc. Conf. Design Automation and Testing in Europe DATE 2002*, pp. 102-109, Mar. 2002.

[7] E. Rijpkema et al., "Trade-Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip," *Proc. Conf. DATE 2003*, pp. 350-355, Mar. 2003.

[8] F. Karim et al., "On-Chip Communication Architecture for OC-768 Network Processors," *Proc. Design Automation Conf.*, pp. 678-678, June 2001.
[9] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. Computer-Aided Design of Circuits and Systems*, vol. 19, no. 12, pp. 1523-1543, Dec. 2000.
[10] H. Zhang et al., "A 1V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing," *IEEE J. Solid State Circuits*, vol. 35, no. 11, pp. 1697-1704, Nov. 2000.
[11] X. Zhu and S. Malik, "A Hierarchical Modeling Framework for On-Chip Communication Architectures," *Proc. Int'l Conf. Computer Design 2002*, pp. 663-671, Nov. 2002.
[12] I. Saastamoinen, D. Siguenza-Tortosa, and J. Nurmi, "Interconnect IP Node for Future System-on-Chip Designs," *Proc. First IEEE Int'l Workshop Electronic Design, Test and Applications*, pp. 116-120, Jan. 2002.
[13] S.J. Lee et al., "An 800MHz Star-Connected On-Chip Network for Application to Systems on a Chip," *Digest of Technical Papers, ISSCC 2003*, pp. 468-469, Feb. 2003.
[14] A. Jantsch and H. Tenhunen, *Networks on Chip*. Kluwer Academic Publishers, 2003.
[15] S.J. Krolikoski et al., "Methodology and Technology for Virtual Component Driven Hardware/Software Co-Design on the System-Level," *Proc. IEEE Int'l Symp. Circuits and Systems '99*, pp. 456-459, June 1999.
[16] E.B. Van der Tol and E.G.T. Jaspers, "Mapping of MPEG-4 Decoding on a Flexible Architecture Platform," *Proc. SPIE 2002*, pp. 1-13, Jan. 2002.
[17] E.G.T. Jaspers et al., "Chip-set for Video Display of Multimedia Information," *IEEE Trans. Consumer Electronics*, vol 45, no. 3, pp. 707-716, Aug. 1999.
[18] A. Pinto et al., "Efficient Synthesis of Networks on Chip," *Proc. Int'l Conf. Computer Design 2003*, pp. 146-150, Oct. 2003.
[19] D. Whelihan and H. Schmit, "Memory Optimization in Single Chip Network Fabrics," *Proc. Design Automation Conf. 2002*, pp. 530-535, June 2002.
[20] W.H. Ho and T.M. Pinkston, "A Methodology for Designing Efficient On-Chip Interconnects on Well-Behaved Communication Patterns," *Proc. Symp. High Performance Computer Architecture 2003*, pp. 377-388, Feb. 2003.
[21] P. Wielage and K. Goossens, "Networks on Silicon: Blessing or Nightmare?" *Proc. Euromicro Symp. Digital System Design DSD 2002*, pp. 196-200, Sept. 2002.
[22] J. Hu and R. Marculescu, "Energy-Aware Mapping for Tile-Based NOC Architectures under Performance Constraints," *Proc. Asia and South Pacific Design Automation Conf. 2003*, pp. 233-239, Jan. 2003.
[23] J. Hu and R. Marculescu, "Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures," *Proc. DATE Conf. 2003*, Mar. 2003.
[24] M. Dallosso et al., "`×pipes`: A Latency Insensitive Parameterized Network-on-chip Architecture for Multi-Processor SoCs," pp. 536-539, *Proc. Int'l Conf. Computer Design*, 2003.
[25] A. Jalabert et al., "`×pipesCompiler`: A Tool For Instantiating Application Specific Networks on Chips," *Proc. Conf. DATE*, 2004.
[26] S. Murali and G. De Micheli, "Bandwidth Constrained Mapping of Cores onto NoC Architectures," *Proc. Conf. DATE*, 2004.
[27] S. Murali and G. De Micheli, "SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs," *Proc. Design Automation Conf.*, 2004.
[28] J.G. Kim and Y.D. Kim, "A Linear Programming-Based Algorithm for Floorplanning in VLSI Design," *IEEE Trans. CAD*, pp. 584-592, vol. 22, no. 5 , May 2003.
[29] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
[30] D. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture, a Hardware/Software Approach*. Morgan Kaufmann, 1999.
[31] K. Compton and S. Hauck, "Reconfigurable Computing: a Survey of System and Software," *ACM Computing Surveys*, pp. 171-210, vol. 34, no. 2, June 2002.
[32] R. Tessier and W. Burleson, "Reconfigurable Computing and Digital Signal Processing: A Survey," *J. VLSI Signal Processing*, pp. 7-27, vol. 28, no. 3, May 2001.
[33] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, pp. 187-189. Kluwer Academic Publishers, 1995.
[34] W.J. Dally and B. Towles, "Route Packets, not Wires: On-Chip Interconnection Networks," *Proc. Design and Automation Conf. DAC 2001*, pp. 684-689, June 2001.
[35] W.J. Dally and S. Lacy, "VLSI Architecture: Past, Present and Future," *Proc. Conf. Advanced Research in VLSI*, pp. 232-241, 1999.
[36] D. Wingard, "MicroNetwork-Based Integration for SoCs," *Proc. Design Automation Conf. DAC 2001*, pp. 673-677, June 2001.
[37] H.S Wang et al., "Orion: A Power-Performance Simulator for Interconnection Networks," *IEEE MICRO*, Nov. 2002.
[38] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires," *Proc. IEEE*, pp. 490-504, Apr. 2001.
[39] D. Wiklund and D. Liu, "SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems," *Proc. Int'l Parallel and Distributed Processing Symp. 2003*, pp. 78-85, 2003.
[40] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni Vincentelli, "Theory of Latency-Insensitive Design," *IEEE Trans. CAD of ICs and Systems*, vol. 20, no. 9, pp. 1059-1076, Sept. 2001.
[41] L. Scheffer, "Methodologies and Tools for Pipelined On-Chip Interconnects," *Proc. Int'l Conf. Computer Design*, pp. 152-157, 2002.
[42] Tensilica Offload Engine, http://www.tensilica.com/html/pr_2003_05_12.html, 2004.
[43] VSI Alliance, http://www.vsi.org/, 2004.
[44] Open Core Protocol, http://www.ocpip.org/, 2004.

Davide Bertozzi received the Electrical Engineering degree from the University of Bologna (Italy) in 1999 and the PhD degree in 2003 from the same university. He currently holds a postdoctorate position in the Department of Electrical Engineering and provides consulting activity for several semiconductor industries. His main research interests concern system level design and communication architectures for multiprocessor systems-on-chip.

Antoine Jalabert received the Engineer Diploma and the MSc degree in electrical engineering in 2003 from the Ecole Supérieure d'Electronique de l'Ouest (ESEO), Angers, France, after joining one year (2003) at the Institut Supérieur d'Electronique de Paris (ISEP), Paris, France, where he specialized in microelectronics. He is currently pursuing the PhD degree with the Laboratoire d'Electronique, de Technologie et d'Instrumentation (LETI), CEA-Grenoble, France, and ISEP. His work is now dedicated to computing architectures using molecular electronics (Post-CMOS Project).

Srinivasan Murali received the bachelor's degree in computer science and engineering from the University of Madras, India in 2002. He is currently pursuing the PhD degree at Stanford University. His research interests include mapping of applications onto networks on chip architectures, multiprocessor systems on chips, and reliability issues of SoCs. He is a student member of the IEEE.

Rutuparna Tamhankar received the BE degree in electronics engineering from the University of Pune, India, in 1999 and the MS degree in electrical engineering from West Virginia University in 2001. He is currently pursuing an Engineer's degree at Stanford University and is also working at Sun Microsystems Inc., Sunnyvale, California. At Sun Microsystems Inc., he is working on timing methodology for high performance microprocessor designs. His current research includes low latency and efficient link design for network on chips. He is a student member of the IEEE.

Stergios Stergiou received the BS degree from the University of Athens, Greece, and the MS degree in computer science from the University of Patras, Greece. He is currently pursuing the PhD degree in electrical engineering at Stanford University, Stanford, California. His research interests comprise all aspects of computer-aided design of digital circuits, with particular emphasis on networks on chip. He is a student member of the IEEE.

Luca Benini is an associate professor in the Department of Electrical Engineering and Computer Science (DEIS) at the Universty of Bologna. He received the PhD degree in electrical engineering from Stanford University in 1997. He also holds visiting researcher positions at Stanford University and the Hewlett-Packard Laboratories, Palo Alto, California. Dr. Benini's research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications, and in the design of portable systems. On these topics, he has published more than 200 papers in international journals and conferences and three books. He has been program chair and vice-chair of Design Automation and Test in Europe Conference. He is a member of the technical program committee and organizing committee of several technical conferences, including the Design Automation Conference, International Symposium on Low Power Design, and the Symposiom on Hardware-Software Codesign. He is a member of the IEEE.

Giovanni De Micheli is a professor of electrical engineering and, by courtesy, of computer science at Stanford University. Previously, he held positions at the IBM T.J. Watson Research Center, Yorktown Heights, New York, at the Department of Electronics of the Politecnico di Milano, Italy, and at Harris Semiconductor, Melbourne, Florida. He received the Nuclear Engineer degree (Politecnico di Milano, 1979), and the MS and PhD degrees in electrical engineering and computer science (University of California at Berkeley, 1980 and 1983). His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software codesign, and low-power design. He is the author of *Synthesis and Optimization of Digital Circuits* (McGraw-Hill, 1994), coauthor and/or coeditor of five other books, and more than 270 technical articles. He is, or has been, member of the technical advisory board of several companies, including Magma Design Automation, Coware, Aplus Design Technologies, Ambit Design Systems, and STMicroelectronics. Dr. De Micheli is the recipient of the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems. He is a fellow of the ACM and IEEE. He received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He received the 1987 D. Pederson Award for the best paper on the *IEEE Transactions on CAD/ICAS* and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He is past president of the IEEE CAS Society. He was Editor in Chief of the *IEEE Transactions on CAD/ICAS* from 1987-2001. Dr. De Micheli was the program chair and general chair of the Design Automation Conference (DAC) from 1996-1997 and 2000, respectively. He was the program and general chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was also codirector of the NATO Advanced Study Institutes on Hardware/Software Co-Design, held in Tremezzo, Italy, 1995 and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, 1986. He is a founding member of the ALaRI Institute at Universita' della Svizzera Italiana (USI), in Lugano, Switzerland, where he is currently scientific counselor.