Channel Mapping in System Level Design

Lukai Cai and Daniel Gajski

CECS Technical Report 03-03
Jan 7, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{lcai, gajski}@ics.uci.edu

Channel Mapping in System Level Design

Lukai Cai and Daniel Gajski

CECS Technical Report 03-03
Feb 3, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425,USA
(949) 824-8059

{lcai, gajski}@ics.uci.edu

Abstract

This report proposes a design flow and algorithms to implement the channel mapping, which generates interconnection topology of the system architecture, selects bus protocols, maps the communication channels among PEs to the generated buses, and configures bus parameters. Unlike previous work, the proposed design flow and algorithms are tailored for complex systems which contain hundreds of PEs with incompatible communication protocols. We have applied the proposed approach on the vocoder project and have approved the approach's correctness.

Contents

1. Introduction **1**

2. Related Work **2**

3. System Level Design Flow **3**

4. Channel Mapping Design Flow **3**

5. Bus Topology Generation **6**

6. Bus Protocol Selection **8**

7. Transducer Insertion **10**

8. Channel Mapping **10**

9. Bus Configuration **11**
9.1 Bus Library . . . 11
9.2 Delay Estimation . . . 11
9.2.1 Bus Transfer Delay . . . 11
9.2.2 PE Data Preparation Delay . . . 12
9.2.3 Transducer Delay . . . 12
9.2.4 Total Communication Delay . . . 12
9.3 Design Constraint . . . 13
9.4 Channel Rate . . . 13
9.5 Bus Selection Inequations . . . 13
9.5.1 Bus Rate Inequations . . . 13
9.5.2 Design Constraint Inequations . . . 13
9.6 Bus Type Selection Algorithm . . . 13

10Slowly Exploring Algorithm **15**

11Experimental Result **15**
11.1 10-PE Example . . . 15
11.2 Vocoder Project . . . 17

12Conclusion **19**

List of Figures

1 Y chart . . . 1
2 An example of channel mapping's influence on the design cost and design performance . . . 2
3 System modelling and design tasks of system level design . . . 4
4 Channel mapping design flow . . . 5
5 The example of topology tree and interconnection topology . . . 6
6 The algorithm of bus topology generation. . . . 7
8 The example of bus protocol selection . . . 8
7 The example of bus topology generation . . . 9
9 The algorithm of bus protocol selection. . . . 10
10 The example of transducer insertion and bus merging. . . . 11
11 The example of inter-influence relations of clusters in the topology tree. . . . 14
12 The algorithm of bus type selection(bus configuration). . . . 14
13 The slowly exploring algorithm. . . . 15
14 Behavior diagram of 10-PE example. . . . 15
15 The generated interconnection topology for 10-PE example. . . . 17
16 Block diagram of encoding part of vocoder . . . 17
17 Interconnection topology of vocoder project . . . 19

List of Tables

1 The example of channel mapping. 10
2 The communication protocols of the selected PEs in 10-PE example 16
3 The computation time of processes in 10-PE example 16
4 The bus types in the bus library for 10-PE example 16
5 The sets of time constraints of processes in 10-PE example 16
6 The bus type selection for 10-PE example 18
7 The communication time of processes after the bus type selection for 10-PE example. 18
8 The execution time of processes after the bus type selection for 10-PE example. 18
9 The behavior-PE mapping relations in vocoder project. 18
10 The attributes of selected PEs in vocoder project. 18
11 The time constraints of processes in vocoder project. 18
12 The bus types in the bus library for vocoder project 19
13 The execution time of processes in vocoder project. 19

Channel Mapping in System Level Design

Lukai Cai and Daniel Gajski
Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine

Abstract

This report proposes a design flow and algorithms to implement the channel mapping, which generates interconnection topology of the system architecture, selects bus protocols, maps the communication channels among PEs to the generated buses, and configures bus parameters. Unlike previous work, the proposed design flow and algorithms are tailored for the complex systems which contain hundreds of PEs with incompatible communication protocols. We have applied the proposed approach on the vocoder project and have approved the approach's correctness.

1. Introduction

In order to handle the ever increasing complexity and time-to-market pressures in the design of system-on-chips(SOCs) or embedded systems, the design has been raised to the system level to increase productivity. Figure 1 illustrates extended Gajski and Kuhn's Y chart representing the entire design flow, which is composed of four different levels: system level, RTL level, logic level, and transistor level. The thick arc represents the system level design. It starts from the behavior model representing the design functionality (also called system behavior or application), which is denoted by point S. The behavior model contains a set of functional blocks (also called behavior or process). It also contains a set of variables that reserve the data transferred between intra-block operations or inter-block operations. The system level design then synthesizes the behavior model to the architecture model representing the system architecture denoted by point A. An architecture model consists of a number of PEs (processing elements) and a number of global memory connected by buses. Different PEs can belong to different PE types. Each PE in the architecture model implements a number of functional blocks in the behavior model.

We divide the synthesis process of system level design to three tasks: behavior-PE mapping, variable-memory mapping, and channel mapping. **Behavior-PE mapping** selects PEs to assemble the system architecture and maps the behaviors (functional blocks) in the behavior model to PEs. **Variable-memory mapping** selects sizes of local memories of PEs and the global memories in the system architecture and maps the variables of behaviors to the memories. **Channel mapping** generates interconnection topology of the system architecture, selects bus protocols, maps the communication among PEs to the generated buses, and configures bus parameters. In this report, we focus on *channel mapping*.

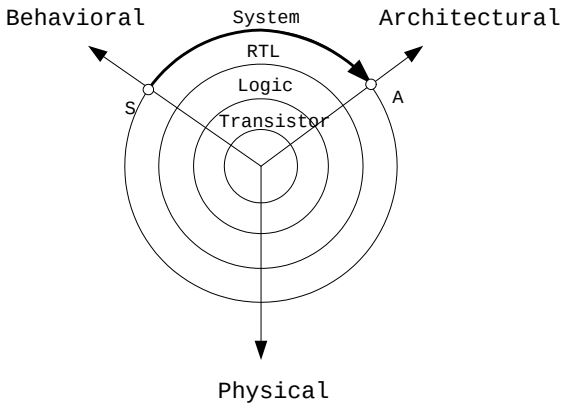
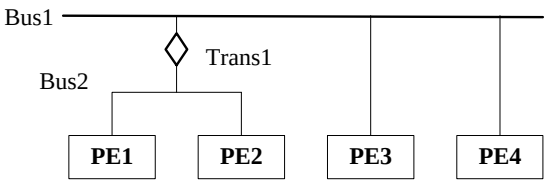
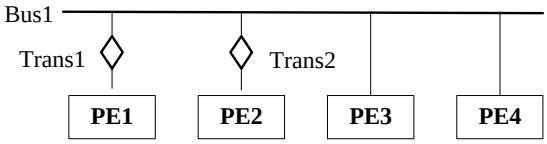


Figure 1. Y chart

Channel mapping becomes essential because it greatly influences the design cost and performance. Figure 2 shows two alternatives of channel mapping results. The system architecture contains four PEs. *PE1* and *PE2* use protocol A as their communication protocol, while *PE3* and *PE4* use protocol B. Protocols A and B are incompatible. Triangle represents transducer which makes the incompatible buses communicate. As shown in Figure 2, *Solution 1* contains one transducer while *solution 2* contains two transducers. As a result, their design cost are different. Furthermore, the data transferred between *PE1* and *PE2* must go through



(a) Channel mapping solution 1



(b) Channel mapping solution 2

Figure 2. An example of channel mapping's influence on the design cost and design performance

two transducers in *solution 2*. On the other hand, it goes through no transducers in *solution 1*. Obviously, different alternatives have different communication time required to transfer data between *PE1* and *PE2*.

Channel mapping explores four design spaces which influence the design cost and performance. First, it determines the interconnection topology which connects PEs by a number of generated system buses. Second, it selects the bus protocols for the system buses. Third, it maps communication channels in system behavior to system buses in the interconnection topology. Finally, it configures the bus parameters, such as bus width and bus speed. The goal of *channel mapping* is to produce a design which meets the design time constraint and has the lowest design cost.

The *channel mapping* becomes more and more complex with the increase of the complexity of system behavior and targeted system architecture. One of the main challenges is that the future system architecture will contain hundreds of PEs with incompatible communication protocols. This feature enforces us to correctly group PEs based not only on generated communication traffic, but also on the protocol compatibility. Furthermore, we must insert a transducer between any two incompatible PEs/buses, which increases the design cost and communication time.

To satisfy the requirement of the complex system design, in this report, we propose a methodology of *channel mapping*. This report is organized as followed. Section 2 describes the related work and our contribution. Section 3 describes the entire system level design flow and input/output of *channel mapping*. The design flow of channel mapping is given in section 4. Section 5 introduces the algorithm of interconnection topology generation. Section 6 introduces the algorithm of bus protocol selection. Section 7 describes the algorithm of transducer insertion. Section 8 describes the process of channel mapping. The algorithm of bus configuration is given in Section 9. Section 10 extends the approach mentioned above to enlarge the explored design space. Experimental result is described in section 11. Finally, section 12 gives the conclusion.

2. Related Work

Much research has been done for communication synthesis. In [3], Daveau deals with both bus protocol selection and interface generation and is based on binding/allocation of communication units. In [11], Yen crate a new processing element and a bus when it is not possible to assign a process to an already existing processing elements or a communication on a bus without violating real time constraints. In [7], Knudsen presents a communication estimation model and shows the importance of integrating communication protocol selection with hardware/software partitioning by the use of this model. In [8], Lahiri automatically maps the various communications between system components onto a target communication architecture template and configure the communication protocols of each channel in the architecture in order to optimize system performance by taking the bus conflict into account. In [10], Ortega maps high-level specifications to arbitrary architectures by analyzing the global view of communication.

Compared to above classical communication synthesis approaches, our approach has the following three advantages:

1. It outlines the entire flow of channel mapping, which contains steps of topology generation, bus protocol selection, channel mapping, and bus configuration. This flow is suitable for the complex system architecture which contains many PEs with incompatible communication protocols.

2. It takes the transducer into account during the design. It not only describes the step of transducer insertion, bus also also computes the transducer delay as part of the communication delay.

3. It evaluates the validation of channel-mapping decision not only by checking the design constraint, bus also by checking the bus load. As a result, it divides the process of bus implementation selection into three steps: bus protocol selection, bus configuration based on bus load, and bus configuration based on communication delay. Each step reduces the bus exploration space thus shorten the design time.

The limitations of our approach are as follows:

1. Our approach doesn't take the bus confliction into account during communication delay estimation.

2. Our approach doesn't take the synchronization into account during communication delay estimation.

3. System Level Design Flow

Before introducing *channel-mapping* methodology, in this section, we first describe how *channel-mapping* works under the entire system level design flow, and the input/output of *channel-mapping*.

We model the design using system level design langauges(SLDL) such as SpecC [5] and SystemC [1]. The SLDL allows modelling the design at different levels of abstraction, from the behavior model reflecting design functionality, to the architecture model reflecting the detailed system architecture. Modelling design using SLDL enables designers to refine the design specification by gradually adding implementation details.

The design starts from behavior model reflecting design's functionality. For example, Figure 3(a) contains four processes named *A*,*B*,*C*, and *D*. Process *B* is executed after *A*, while processes *C*, *D*, and *AB*(hierarchical process containing *A* and *B*) are executed concurrently. Processes communicate through global variables. In Figure 3(a), process *A* sends data to *B* by variable *v1*, processes *A* and *C* exchange data through variable *v2*, and processes *B*, *C*, and *D* exchange data through variable *v3*.

Task *behavior-PE mapping* then selects PEs to assemble the system architecture and map processes to the PEs. The generated system model is illustrated by Figure 3(b). It contains three computation components, *PE1*, *PE2*, and *PE3*. Processes *A*, *B*, *C* and *D* are mapped to them respectively. The generated models explicitly specifies the computation components. The communication among computation components are still represented by global variable accessing.

After *behavior-PE mapping*, task *variable-memory mapping* selects storage components of system architecture, chooses variable-memory mapping schemes, and refines the global variables to system channels. In Figure 3(c), each PE has a local copy of the global variable that it accesses. The communication is implemented by message-passing mechanism. In Figure 3(d), a global memory(GM) is added to the system architecture, and the global variable *v3* is stored in the global memory. In both Figure 3(c) and (d), computation/storage components communicates through channels, which replace the global variable accesses in the previous model in Figure 3(b). For example, channel *c2* in Figure 3(c) and (d) replaces variable *v2* access in Figure 3(b). Channels are modelled at the abstract level, which are independent of bus implementation. Each channel can be only used by two PEs to perform point-to-point communication.

The task *channel mapping* is implemented after *variable-memory mapping*. It reads the system model such as Figure 3(c) and (d) as inputs, and determines the bus topology, selects bus protocols, maps channels to buses, and chooses bus configuration. The system models reflecting the result of *channel mapping* are displayed in Figure 3(e) and (f). The generated topology can be flat(e), or hierarchical(f). The transducers are inserted between incompatible PEs, which are denoted by triangles. It should be noted that *channel mapping* only makes design decision without refining the system model. The system model should be refined manually by designers or refined automatically by communication refinement tools.

4. Channel Mapping Design Flow

The design flow of *channel mapping* contains five steps, which is illustrated in Figure 4. Figure 4 uses *Model 2* in Figure 3 as the example.

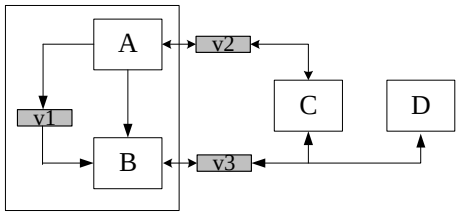
The first step is *bus topology generation*. It generates the interconnection topology and connects the PEs to buses. The *bus topology generation* doesn't determine the bus details. It is accomplished by analyzing the compatibility of PE's communication protocols and analyzing the traffic among PEs. Models (a) and (b) in Figure 4 illustrate the designs before and after *bus topology generation*. *Bus topology generation* is introduced in section 5.

The second step is *bus protocol selection*. It selects the bus protocol for every bus in the interconnection topology. It is accomplished by analyzing the traffic among PEs. Model (c) in Figure 4 illustrates the design after *bus protocol selection*. *Bus protocol selection* is introduced in section 6.

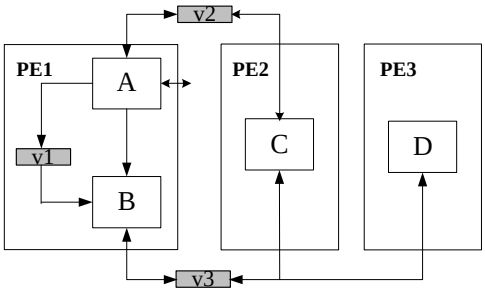
The third step is *transducer insertion*. It inserts a transducer between PE and its connecting bus or between buses if they have incompatible protocols. Model (d) in Figure 4 illustrates the design after *transducer insertion*. *Transducer insertion* is introduced in section 7.

The fourth step is *channel mapping*. It statically maps different channels to the buses . Some channels are mapped to one bus, which others are mapped to multiple buses, according to the interconnection topology. Model (e) in Figure 4 illustrates the design after *channel mapping*. *Channel mapping* is introduced in section 8.

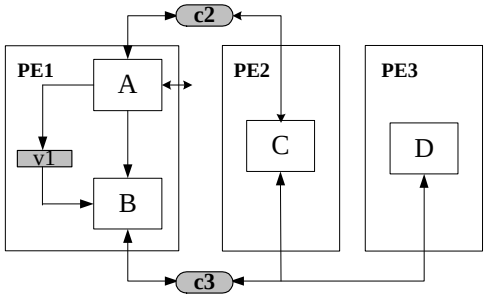
The final step is *bus configuration*, which determines the bus speed. The bus speed is determined by the bus width and the bus delay. We select bus speed to ensure that the design time constraint can be met and the maximum bus rates are not exceed. Model (f) in Figure 4 illustrates the design after *bus configuration*. *Bus configuration* is introduced in section 9.



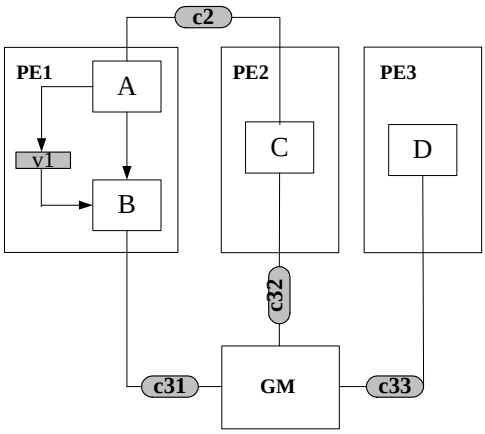
(a) Initial design model



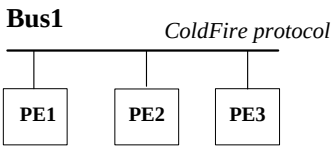
(b) Model after behavior-PE mapping



(c) Model after variable mapping
(without global memory GM)

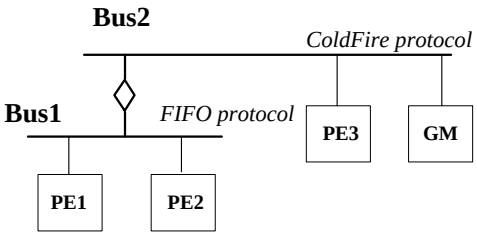


(d) Model after variable mapping
(with global memory GM)



Channel	Mapped buses
c2	bus1
c3	bus1

(e) Model after channel mapping
(from (c))



Channel	Mapped buses
c2	bus1
c31	bus1, bus2
c32	bus1, bus2
c33	bus2

(f) Model after channel mapping
(from (d))

Figure 3. System modelling and design tasks of system level design

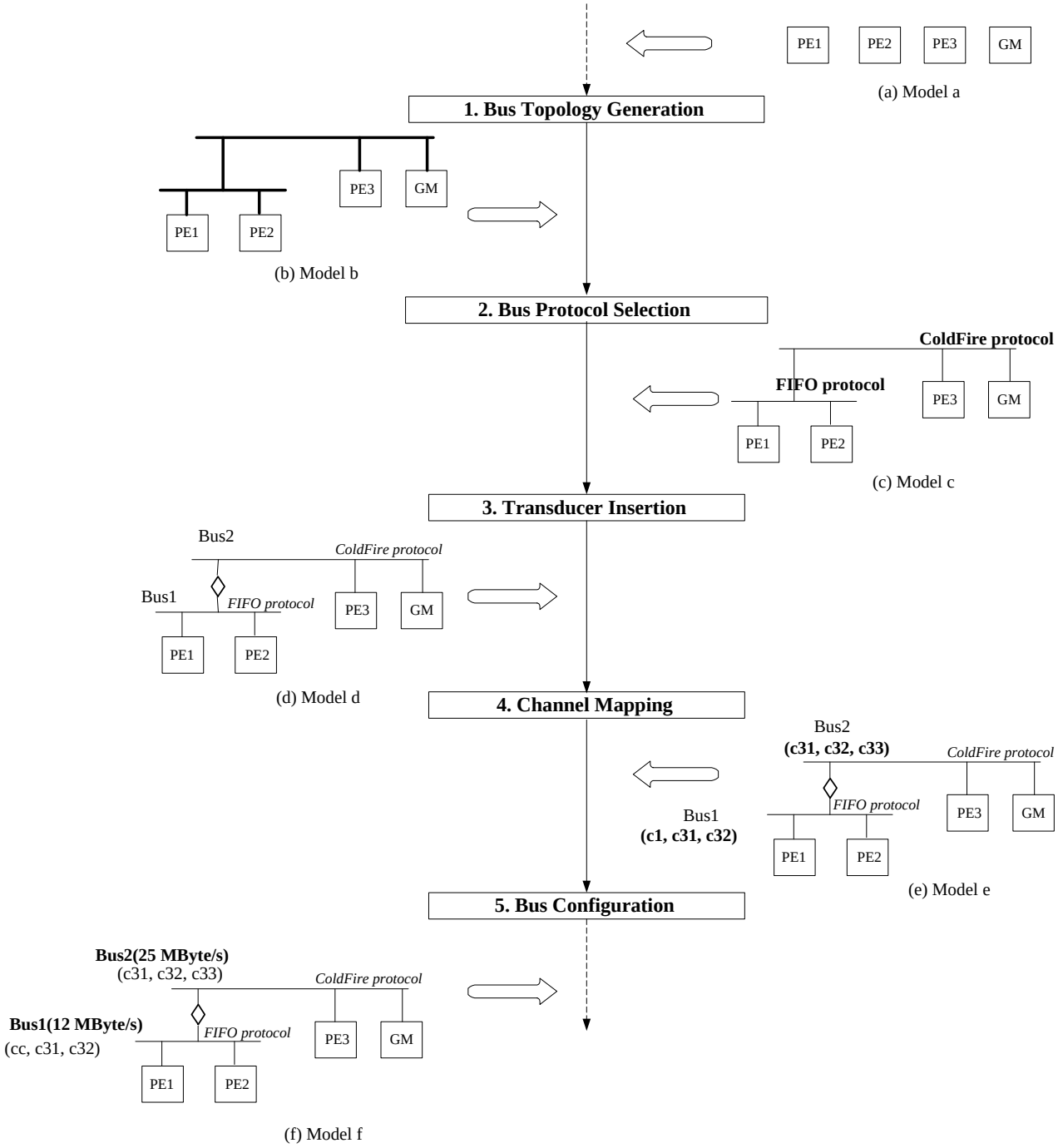


Figure 4. Channel mapping design flow

5. Bus Topology Generation

We generate the interconnection topology by taking two issues into account: the protocol compatibility among PEs, and the traffic among PEs.

There are two types of PEs: fixed-protocol PE and unfixed-protocol PE. Fixed-protocol PEs, such as microprocessors, DSPs, and global memories, have predefined communication protocols. Unfixed-protocol PEs, such as custom hardware, doesn't define any communication protocols. After designers select the protocol for custom hardware, they will generate corresponding custom hardware interface during interface synthesis. Therefore, the protocol of the custom hardware can be treated as compatible to the protocols of any other PEs.

Two PEs having incompatible communication protocols cannot directly communicate. A transducer is required to insert between the incompatible PEs to convert one communication protocol to another. Transducer insertion increases the design cost and communication delay. As a result, during the design, we attempt to keep the smallest number of inserted transducers.

Beside the protocol compatibility, we also take the traffic among PEs into account. In order to localize the traffic, we put PEs having heavy traffic physically close to each other. We evaluate the physical closeness of PEs by computing the smallest number of buses on communication paths between them. Localizing the traffic not only reduces the communication delay between PEs having heavy traffic, but also reduces the overall system bus load.

We use a topology tree to represent an interconnection topology. The leaf nodes in the tree represent PEs. The hierarchical nodes represents buses. Figure 5 displays two examples of the hierarchical trees and their representing interconnection topology.

The simplest topology is illustrated by Figure 5(b). In this topology, all the PEs are connected to a single system bus. To keep the smallest number of inserted transducers and localize the traffic, we adopt hierarchically clustering algorithm [4] to improve the simplest interconnection topology. The hierarchical-clustering algorithm considers a set of objects and groups them according to some measure of closeness. The two closest objects are clustered first and considered to be a single object for future clustering. The clustering process continues by grouping two individual objects, or an object or cluster with another cluster on each iteration. The algorithm stops when a single cluster is generated and a hierarchical cluster tree has been formed. In our algorithm, the topology tree is such a hierarchical cluster tree. Each PE is a leaf object/cluster. Each bus is a hierarchical object/cluster.

We have two measures of closeness: protocol compatibility of clusters and traffic between clusters. To define the protocol compatibility, we first define the type of clusters. If all the PEs in a cluster are compatible fixed-protocol PEs, we define the cluster as a *SW* cluster. If all the PEs in a cluster are unfixed-protocol PEs, we define the cluster as a *HW* cluster. If a cluster contains both fixed-protocol PEs and unfixed-protocol PEs and all the fixed-protocol PEs are compatible, then we define it as a *SWHW* cluster. Otherwise, if a cluster contains incompatible fixed-protocol PEs, then we define it as an *incompatible* cluster. The protocol of a *SW* or *SWHW* cluster is defined as the protocol of any fixed-protocol PEs in the cluster.

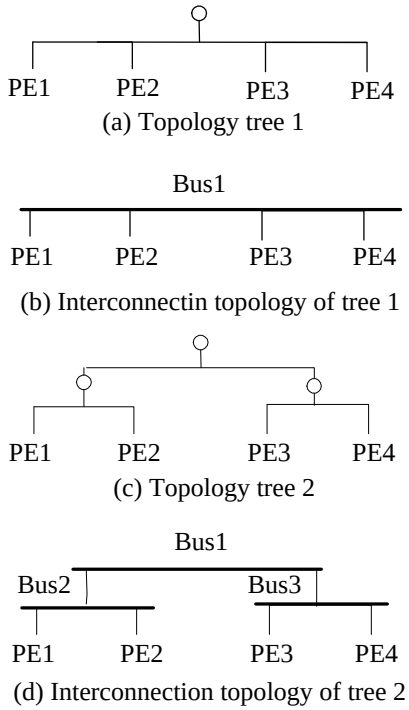


Figure 5. The example of topology tree and interconnection topology

We define three types of protocol compatibility: *strictly compatible*, *generally compatible*, and *incompatible*. If the protocols of two *SW* clusters are compatible, they are *strictly compatible*. If the protocols of a *SW*/*SWHW*/*HW* cluster and a *SWHW*/*HW* cluster are compatible, they are *generally compatible*. Otherwise, the clusters are *incompatible*. We use protocol compatibility as the first priority key to measure the closeness: the distance of any *strictly compatible* clusters are closer than the distance of any *generally compatible* cluster; the distance of any *generally compatible* clusters are closer than the distance of any *incompatible* clusters.

The second measure of closeness is traffic between clusters. If two pairs of clusters have the same type of protocol

compatibility, then the distance of the one with larger inter-cluster traffic is closer than the distance of another pair.

At each iteration of grouping, we localize the traffic between a pair of clusters and generates an additional cluster representing a bus. Therefore, the generated topology contains a large number of buses. In order to reduce the number of inserted clusters/buses, at each iteration of grouping, we flatten the grouped clusters if the grouped cluster and the new generated cluster either have compatible protocols, or are all *incompatible* clusters. We call this step *cluster flattening*. Cluster flattening reduces the number of buses and the communication delay without introducing new transducers. In our algorithm, we perform cluster flattening after each grouping step. On the other hand, cluster flattening will increase the traffic on the generated bus of merging, which may cause bus-overload, or may increase PE's bus competition. For these cases, we take the tradeoff with cluster flattening or without it into consideration in Section 10.

After each iteration of grouping, a new cluster tree is generated, which represents a new interconnection topology. If we perform further channel mapping tasks explained in Figure 4 for each intermediately generated interconnection topology, then the design time will be very long. Therefore, we only perform further channel mapping tasks for the finally generated interconnection topology of clustering algorithm. We will take each intermediately generated interconnection topology into account in Section 10.

The pseudo algorithm of bus topology generation is shown in Figure 6. At the beginning, function *InitTree* generates an initial cluster tree representing the simplest bus topology. All the leaf nodes in the tree represent PEs, which are directly connected to a root node representing a bus. Function *InitCloseMeasure* then computes the closeness among leaf nodes. Then at each iteration, the algorithm first selects a pair of nodes to be clustered. Function *StrictPair* finds the pair of strict-compatible nodes among which have the heaviest traffic. If no pair of strict-compatible nodes exists in the tree, then it returns *NULL*. Similarly, Function *GeneralPair* finds the pair of general-compatible nodes among which have the heaviest traffic and *InCompatiablePair* finds the pair of incompatible nodes among which have the heaviest traffic. After node-pair selection, algorithm then groups the two nodes in the pair represented by *node1* and *node2* by function *GroupTwoNodes*. Function *ClusterFlatten* implements *cluster-flattening* for the generated node *new_node*. The algorithm then restructures the cluster tree by removing *node1* and *node2* and adding *new_node* in. Finally, *ReComputeCloseMeasure* recomputes the closeness of nodes in the cluster tree. Algorithm continues until root node *root* has only one child node left.

An example of bus topology generation is illustrated in Figure 7. The left hand graphs in the Figure 7 represents

```
Bus_Topo_Generation(){
  cluster_tree = InitTree();
  InitCloseMeasure();
  while(#(root->child) > 1) do
    pair = StrictPair();
    if(pair = NULL) do
      pair = GeneralPair();
    endif
    if (pair = NULL) do
      pair = InCompatiablePair();
    endif
    if (pair != NULL) do
      node1 = pair->mNode1;
      node2 = pair->mNode2;
      new_node = GroupTwoNodes(
                 node1,node2);
      ClusterFlatten(new_node,
                     node1, node2);
      cluster_tree->Remove(node1);
      cluster_tree->Remove(node2);
      cluster_tree->Add(new_node);
      ReComputeCloseMeasure();
    endif
  endwhile
}
```

Figure 6. The algorithm of bus topology generation.

the PE/cluster communication graph: the node denotes the PE/cluster and the edge denotes the traffic among PEs/clusters. The number attached to the edge denotes the amount of traffic. The right hand graphs in the Figure 7 represents the cluster tree.

6. Bus Protocol Selection

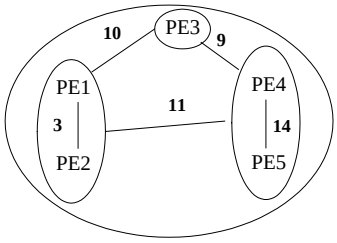
After bus topology generation, we select bus protocols for buses in the interconnection topology. During bus protocol selection, we use different approaches to select protocols for buses represented by different types of clusters. The algorithm of protocol selection is illustrated in Figure 9.

In *SW*/*SWHW* clusters, all the fixed-protocol PEs have the same bus protocol, which is defined as the protocol of the cluster. Therefore, Function *PEProtocol* selects the protocol of a SW/SWHW cluster as the protocol of the bus represented by the cluster.

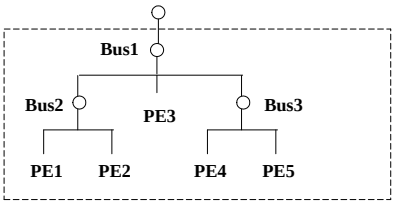
On the other hand, in any *incompatible* clusters, there are more than one protocols that are used by PEs in the clusters. We first compute the traffic of *incompatible* clusters' child clusters. We classify the cluster traffic into two types: inter-cluster traffic and intra-cluster traffic. Inter-cluster traffic of cluster ω equals the sum of traffic between any PE in cluster ω and any PE beyond cluster ω. Intra-cluster traffic of cluster ω equals the sum of traffic between any of two ω's child clusters. The example in Figure 8 illustrates cluster traffic computation, which follows the design steps of the example in Figure 7. The second and third columns of table in Figure 8(d) represents the computed inter-traffic and intra-traffic.

During protocol selection for *incompatible* cluster , we select the bus protocols of a child cluster before the bus protocol of its parent cluster in the topology tree. For example, in Figure 8, we select bus protocols for *Bus2* and *Bus3* before the protocol selection of *Bus1*. When we select protocol for *incompatible* cluster ω, we first compute the ω's traffic of each protocol ρ, which is represented *by traffic[p]* in the algorithm. We define the ω's traffic of protocol ρ as the sum of the inter-traffic of ω's child clusters whose selected protocols are ρ. We choose the protocol which has the largest ω's traffic as ω's protocol. For example, in Figure 8, because the traffic of protocol *A* for *Bus1* is the largest(=21), which is larger than the traffic of protocol *B*(=19) and the traffic of protocol *C*(=20), we select protocol *A* as the protocol of *Bus1*. Selecting protocols in this way avoid inserts transducer between the bus represented by ω and the buses/PEs represented by ω's child clusters which have the largest amount of traffic over ω.

After protocol selection for buses represented by *incompatible* clusters and *SW*/*SWHW* clusters, we select protocols for buses represented by *HW* clusters. During *HW* bus protocol selection, we select the protocol of the parent clus-



(a) PE traffic and connection



(b)Bus topology

PE	Protocol
PE1	A
PE2	A
PE3	B
PE4	Any
PE5	C

(c)PE protocols

Cluster	Inter-traffic	Intra-traffic	Total traffic	Selected protocol
Bus1	0	30	30	A
Bus2	21	3	24	A
Bus3	20	14	34	C
PE4	29	0	29	C
PE3	19	0	19	-

(d)Bus traffic and selected protocols

Figure 8. The example of bus protocol selection

PE	Protocol
PE1	A
PE2	A
PE3	B
PE4	Any
PE5	C

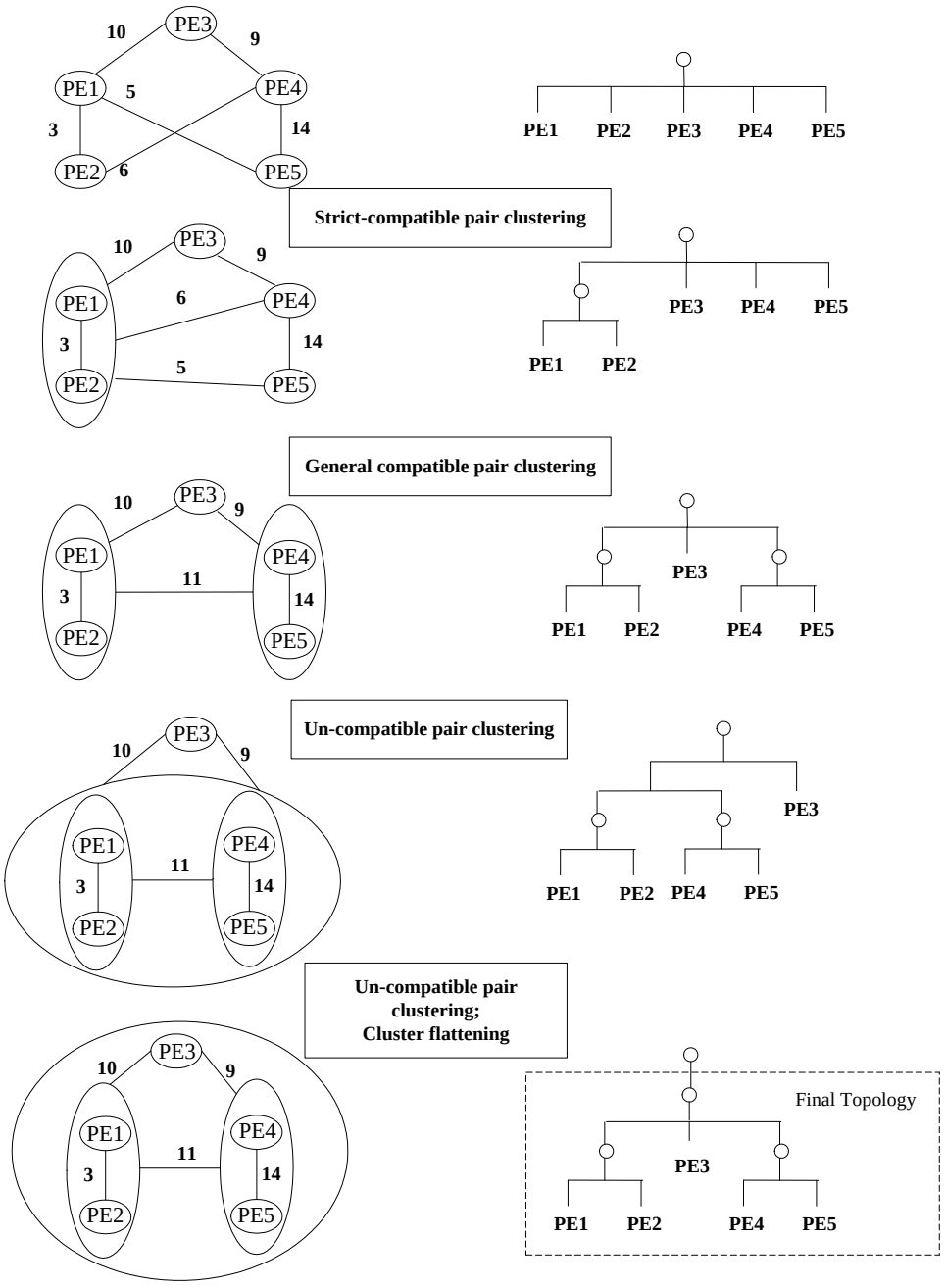


Figure 7. The example of bus topology generation

Channel between PEs	Mapped buses
PE1-PE2	Bus1
PE1-PE3	Bus1, Bus4
PE1-PE5	Bus1, Bus3
PE2-PE4	Bus1, Bus3
PE3-PE4	Bus4, Bus1, Bus3
PE4-PE5	Bus3

Table 1. The example of channel mapping.

ter before the protocol of all its child clusters. If the root cluster is a *HW* cluster, then we select the fastest protocol for it. Otherwise, the protocol of *HW* cluster equals the protocol of its parent cluster, which is shown in function *BusProtocolSelectionAll*.

The example of protocol selection is displayed in Figure 8. Figure 8(c) displays the protocols of PEs. *Any* refers that the protocol of PE is un-fixed. Figure 8 (d) displays the selected bus protocol for *Bus1*, *Bus2*, *Bus3* and *PE4*.

7. Transducer Insertion

After bus protocol selection, the protocols of buses are defined. The protocols of a child cluster and its parent cluster are either incompatible or compatible.

If the protocols of child and parent clusters are incompatible, both the buses represented by the clusters cannot transfer data directly. In this case, we must insert a transducer between them. In our approach, we attach a transducer to the child cluster to represent the inserted transducer. After transducer insertion, the topology tree and represented system architecture of the example in Figure 8 are displayed in Figure 10 (a) and (b). The tab *(B-A)* besides *PE3* in Figure 10(a) represents the transducer connecting buses with protocols *A* and *B*.

If the protocols of child and parent clusters are compatible, in order to reduce the amount of buses in the interconnection topology, we merge the two clusters. We call this process *bus merging*. *Bus merging* is similar to the step *cluster flattening* in *bus topology generation*. After bus merging, the topology tree and represented system architecture of the example are displayed in Figure 10(c) and (d).

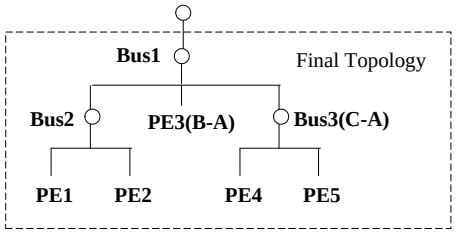
8. Channel Mapping

After transducer insertion, we map the channels among the PEs to the system buses.

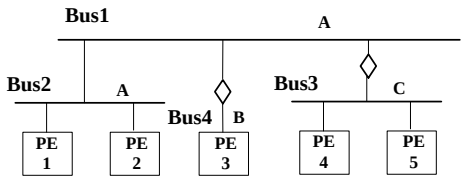
In general, if there are more than one communication paths between PEs, then we should map channels to the buses with the shortest delay or with the lowest bus load. In our approach, the communication path between any two PEs is unique. As a result, we directly map the channels

```
BusProtocolSelectionAll(){
  if(root_cluster.type == HW_Cluster) do
    root_cluster.protocol =
    FastestProtocol();
  endif
    BusProtocolSelection(root_cluster);
};

BusProtocolSelection(cluster A){
  for(each c = child of A )do
    BusProtocolSelection(c);
  endfor

  switch(A.type) do
    case SW_Cluster:
    case SWHW_Cluster:
      A.protocol = PEProtocol(A);
      break;
    case UNMATCH_Cluster:
      for(each possible protocol p) do
        traffic[p] = 0;
      endfor
      for( each c = child of A) do
        if(c.type == SW/SWHW/UNCOMPATIBLE) do
          traffic[c.protocol]+=
              c.inter_traffic;
        endif
      endfor

      temp_traffic = 0;
      for(each possible protocol p) do
        if(traffic[p] > temp_traffic) do
          selected_protocol = p;
          temp_traffic = traffic[p];
        endif
      endfor

      A.protocol = selected_protocol;
      break;
    default:
      break;
  endswitch;

  for(each c = child of A )do
    if (c.type == HW_Cluster) do
      c.protocol = A.protocol;
    endif
  endfor
}
```

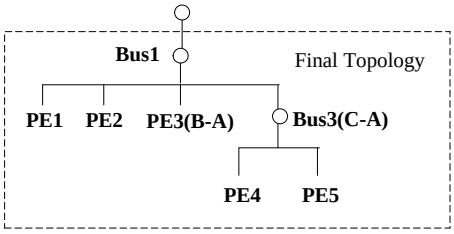
Figure 9. The algorithm of bus protocol selection.



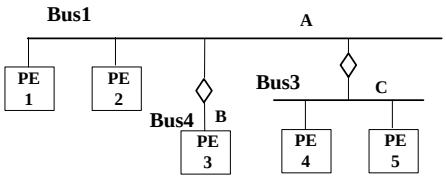
(a) Topology tree after transducer insertion



(b) System architecture after transdducer insertion



(c) Topology tree after bus merging



(d) System architecture after bus merging

Figure 10. The example of transducer insertion and bus merging.

between PEs to the buses on the unique communication path. The table of channel mapping for the example in Figure 10(d) is displayed in Table 1.

9. Bus Configuration

Although the protocols of buses have been selected, we needs to configure the bus parameters which determine the bus speed. The bus configuration must ensure that design constraints can be met.

9.1 Bus Library

All the bus types are stored in a bus library. For any bus *b* with a certain protocol, two parameters can be configured: *Width(b)* and *Delay(b)*. *Width(b)* refers to the number of data lines in bus *b*. *Delay(b)* refers to the total delay associated with the protocol used by a process transfer data over the bus. In our approach, we first pre-configure buses by assigning possible values to the bus parameters. We call each bus configuration as a bus type. In this way, we convert the problem of bus configuration to the problem of the bus type selection for the buses in the interconnection topology of design.

For each bus type *b*, two attributes are pre-computed according to the bus parameters. The first attribute is *BusRate(b)*, which is defined as the maximum rate at which data can be transferred across the bus. The bus rate is computed by the formulation

$$BusRate(b) = Width(b)/Delay(b)$$

Another attribute is the bus delay associated with the protocol used by a process transfer data of type *t* over the bus, which is denoted by $Delay(b,t)$. $Delay(b,t)$ is computed by the formulation

$$Delay(b,t) = \lceil Bits(t) \div Width(b) \rceil \times Delay(b)$$

We compute $Delay(b,t)$ for all the possible data types in the behavior model, such as integer, and float types. All of the $Delay(b,t)$ are stored in a weight table associated with bus type *b* in the bus library.

In the bus library, there are multiple bus types of the same bus protocol. The bus types with the same bus protocol are stored in the same link, in the decreasing order of the bus cost. In general, bus cost is determined by bus width, bus delay, and bus protocol.

9.2 Delay Estimation

9.2.1 Bus Transfer Delay

There are two types of protocols: blocked protocol and unblocked protocol. Sanjiv [9] introduces the communication

delay estimation for unblocked protocol. According to this method, when channel *c* is mapped to bus *b*, we compute the communication time of channel *c* over bus *b* as

$$BusTime(c,b) = Access(c) \times \lceil Bits(c) \div Width(b) \rceil \times Delay(b)$$

where *Access(c)* represents the number of times of data transferred over channel *c* during its lifetime. *Bits(c)* is the type of data being accessed over channel c.

For the bus with the block protocol, the synchronization time should be taken into account. Therefore, the communication delay of blocked protocol equals

$$BusTime(c,b) = SynTime(c,b) + Access(c) \times \lceil Bits(c) \div Width(b) \rceil \times Delay(b)$$

where *SynTime(c,b)* is the channel *c*'s synchronization time over bus *b*. In order to simplify the problem, we assume that the *SynTime(c,b)* is zero.

In our approach, we use pre-computed $Delay(b,t)$ to replace the $Width(b)$ and $Delay(b)$ in the formulation. The converted formulation is

$$BusTime(c,b) = (Access(c) \times Delay(b, DataType(c)))$$

where *DataType(c)* is the type of data being accessed over channel c. In our project, *Access(c)* is computed by the profiler [2].

Furthermore, we compute the total bus delay of PE *P* over bus *b*, which is formulated as,

$$CommTime(P,b) = \sum_{c \in Channel(P,b)} BusTime(c,b)$$

where *Channel(P,b)* is the set of *P*'s channels that are mapped to bus *b*.

9.2.2 PE Data Preparation Delay

When a PE sends/receives data over buses, it consumes time for the data preparation . The data preparation delay may be caused by data packaging, or caused by internal buffer-/memory accessing. We estimate the PE preparation delay for channel c by

$$PETime(c,P) = Access(c) \times Delay(P, DataType(c))$$

where *Delay(P,t)* is the preparation delay of PE *P* per preparation of the data with type *t*. *Delay(P,t)* is precomputed and stored in the PE library.

9.2.3 Transducer Delay

During communication, if data transfers through a transducer between two buses, an additional delay of transducer is added to the total communication time. We estimate this delay by

$$TransTime(c,t) = Max(BusTime(c,b1), BusTime(c,b2)) \times 3$$

where c represents the channel that goes through transducer *t*. *B1* and *b2* are the two buses to which transducer *t* connect. We derive this formulation based on the our experience of transducer synthesis.

9.2.4 Total Communication Delay

For each channel c, the total communication time is formulated as,

$$CommTime(c) = \sum_{P \in PE(c)} PETime(c,P) + \sum_{b \in Bus(c)} BusTime(c,b) + \sum_{t \in Transduers(c)} TransTime(c,t)$$

where *PE(c)* is the set of PEs to which channel *c* connect. *Bus(c)* is the set of buses to which channel *c* are mapped. *Transducer(c)* is the set of transducers over which channel *c* transfer through.

In each PE, a number of processes are executed. The total communication time of any process *p* is formulated as :

$$CommTime(p) = \sum_{c \in Channel(p)} CommTime(c)$$

where *Channel(p)* is the set of channels through which process *p* communicates.

Finally, the total communication time of any PE *P* is formulated as:

$$CommTime(P) = \sum_{c \in Channel(P)} CommTime(c)$$

where *Channel(P)* is the set of P's channels through which PE *P* communicates.

9.3 Design Constraint

During the design, we take two types of design constraints into account: the constraint of entire design denoted by *Cstr(Design)*, and the constraint for a certain process *p* denoted by *Cstr(p)*.

Because that each process has been mapped to a certain PE, the execution time of process *p* denoted by *CompTime(p)* has been computed by the profiler [2]. Therefore, the communication constraint of process *p* can be computed by the formulation

$$CommCstr(p) = Cstr(p) - CompTime(p)$$

We also compute the computation time of PE. Since all the processes are executed sequentially on a PE, we formulate the computation time of PE as

$$CompTime(P) = \sum_{p \in Process(P)} CompTime(p)$$

where *Process(P)* is the set of processes executed on PE *P*.

Therefore, the communication constraint of PE *P* is computed by

$$CommCstr(P) = Cstr(Design) - CompTime(P)$$

9.4 Channel Rate

Furthermore, we compute average channel rate and peak channel rate for each bus *b*.

We compute the average channel rate of any bus *b*, which is formulated as:

$$AvgChanRate(b) = \sum_{c \in Channel(b)} (Access(c) \times Bits(c)) / Cstr(Design)$$

where *Channel(b)* is the set of channels that are mapped to *b*.

We also compute the peak channel rate on a bus *b*, which is formulated as:

$$PeakChanRate(b) = \max_{p \in Process(Design)} ((\sum_{(c \in Channel(p)) \cap (c \in Channel(b))} (Access(c) \times Bits(c))) / (CommCstr(p)))$$

where *Process(Design)* is the set of processes in design, *Channel(p)* is the set of channels through which process *p* communicates, and *Channel(b)* is the set of channels that are mapped to *b*.

9.5 Bus Selection Inequations

9.5.1 Bus Rate Inequations

We divide the bus type selection process to two steps. In this first step, we select bus types according to the maximum bus rate and the channel rate. According to [9]

$$BusRate(b) \geq AvgChanRate(b)$$

$$BusRate(b) \geq PeakChanRate(b)$$

As shown in formulations described before, in this step, the process of bus type selection for different buses in the interconnection topology are independent to each other.

9.5.2 Design Constraint Inequations

In the second step, we select bus types which enable the design to meet the design/process constraint by making the following inequations true,

$$CommCstr(P) \geq CommTime(P)$$
$$CommCstr(p) \geq CommTime(p)$$

for any process *p* and PE *P* in the design. Because each channel may be mapped to multiple buses, the communication time of each channel is influenced by the bus type selection of all the buses to which it is mapped. As a result, in this step, the process of bus type selection for different buses in the interconnection topology are inter-dependent.

9.6 Bus Type Selection Algorithm

In this section, we introduce the bus type selection algorithm.

First, for each bus *b* in the interconnection topology, we select all of the bus type candidates which both have the same communication protocol with *b* and make the two inequations described in Figure 9.5.1 true.

Second, we select bus types according to the inequations in section 9.5.2. Because the bus type selections of different buses in this step are inter-dependent, we need to determine the bus type selection sequence. We attempt to first select bus type for the bus which is less inter-influenced with other buses. Figure 11 represents part of a simple topology tree. *C1*.. *Cn* contains a set of different PEs respectively. For any cluster *Ci*, it inter-influences with *Cj* if and only if there exists a PE in *Ci* that communicates with a PE in *Cj*. On

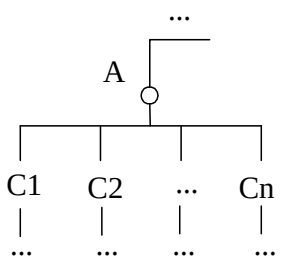


Figure 11. The example of inter-influence relations of clusters in the topology tree.

the other hand, cluster A is inter-influenced with *Ci* if there exists PE in *Ci* communicate with a PE in any other child of *A*. Furthermore, if a PE in *Ci* that communicates with PE that is beyond the subtree led by *A*, the communication path must cross the cluster *A*. As a result, we conclude that a child cluster is less inter-influenced with other clusters than its parent cluster. Therefore,we select bus type for a child cluster before its parent cluster in the topology tree.

Furthermore, we found that the main system bus, which is represented by the top-level cluster, such as *Bus1* in Figure 10(a), greatly effects the total communication time. Therefore, at each outer iteration, we assign one bus type candidate to the main system bus before the bus selection for any other buses in the interconnection topology.

Figure 12 shows the algorithm of bus type selection. First, function *BusRateSelection* selects the bus type candidates for each buses according to bus rate inequations described in section 9.5.1 and bus protocol's compatibility. The selected bus type candidates are stored in *Link* in the decreasing order of the cost. At each outer iteration, one bus type candidate is assigned to the *main_bus* from the head of *Link* of *main_bus*. Function *InitBusSelection* then selects the fastest bus type candidates for all the other buses. At each middle iteration, algorithm reselects the bus type candidate for one bus. Function *NextSelectedBus* selects bus *bus_in_topo* for bus type reselection by traversing the topology tree using post-order scan algorithm, which selects a child cluster before its parent cluster. At each iteration of inner loop, algorithm selects a bus type *type_cur* for bus *bus_in_topo* from the head of *Link* of *bus_in_topo*. Function *MeetConstraint* then tests whether inequations described in section 9.5.2 are true. If so, *type_cur* is selected as the type of *bus_in_topo*. The process continues until the types of all the buses are reselected. Finally, function *SelectLowestCostSolution* selects the lowest cost solution among the different solutions in which *main_bus* chooses different bus type candidates.

```
BusTypeSelection{
  BusRateSelection();
  main_bus_cur = First(Link(
                   main_bus.protocol));
  while( MeetConstraint() ) do
    InitBusSelection();
    while NotFinishAllTheBuses() do
      bus_in_topo = NextSelectedBus();
      type_cur = First(Link(
               bus_in_topo.protocol));
      type_selected = NULL;
      while (type_cur) do
        if MeetConstraint() do
          type_selected = type_cur;
        endif
        type_cur = Next(Link(
               bus_in_topo.protocol));
      endwhile

      if(type_selected != NULL) do
        bus_in_topo.type = type_selected;
      enddo
    endwhile
    SelectLowestCostSolution();
    main_bus_cur = Next(Link(
             main_bus.protocol));
  endwhile
}
```

Figure 12. The algorithm of bus type selection(bus configuration).

```
SlowExploring{
  SelectedTopo = NULL;
  CurCost = MAX;
  Topo = InitClusterTree();
  while(#(Topo.root.child) > 1) do
    BusProtocolSelection(Topo);
    TransducerInsertion(Topo);
    ChannelMapping(Topo);
    BusTypeSelection(Topo);

    if(( ((MeetConstraint(Topo) &&
    ((Topo.cost < CurCost) ||
    (SelectedTopo = NULL) ) do
      SelectedTopo = Topo;
      CurCost = Topo.cost;
    endif

    Topo = ClusterGrouping(Topo);
  endwhile

  return SelectedTopo;
}
```

Figure 13. The slowly exploring algorithm.

10. Slowly Exploring Algorithm

As described in section 5, our bus topology generation algorithm produces only one single final interconnection topology. If no bus type can be selected for a certain bus in the topology to meet the design constraints, then the entire design process fails. We call this algorithm *fast exploring algorithm*.

An alternative solution is to produce a new interconnection topology after each iteration of cluster grouping or cluster flattening during bus topology generation. We call this algorithm *slowly exploring algorithm*, which is explained in Figure 13. At each iteration, *ClusterGrouping* clusters two nodes as described in the loop body of the algorithm in Figure 6. We then perform tasks *bus protocol selection* by function *BusProtocolSelection*, *transducer insertion* by function *TransducerInsertion*, *channel mapping* by function *ChannelMapping*, and *bus type selection* by function *BusTypeSelection*, for each new generated topology. The algorithm continues until the root node of the tree has only one child node. Until then, we have produced one channel-mapping solution for each generated topology. Finally, we select the topology with the lowest cost as our final channel-mapping solution by function *ClusterGrouping*.

Because of the complexity of *slowly exploring algorithm*, we only apply it after the fail of *fast exploring algorithm*.

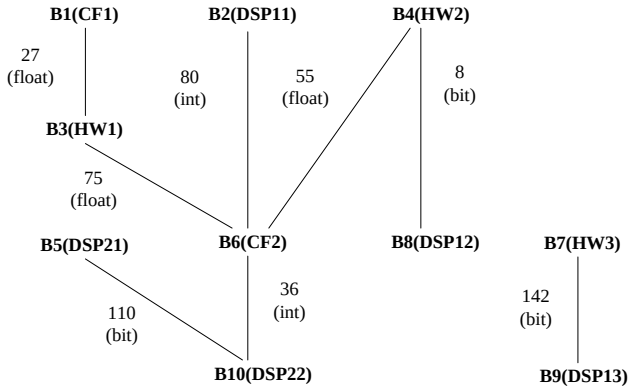


Figure 14. Behavior diagram of 10-PE example.

11. Experimental Result

We produce a tool called *channel-mapper* to implement our channel mapping algorithm by writing around 3000 lines of C++ code. In this section, we describe our design experience for two examples, 10-PE example and vocoder example.

11.1 10-PE Example

We first apply our algorithm on a design containing 10 processes displayed in Figure 14. During *Behavior-PE mapping*, we map different processes to different PEs. For example, process *B1* is mapped to PE *CF1*, which is illustrated by *B1(CF1)* in the Figure. The edge in the figure represents the traffic between processes. For example, the edge between *B1(CF1)* and *B3(HW1)* represents the traffic between process *B1* and *B3*. The tab *27(float)* of the edge denotes the data of *float* type is transferred 27 times.

Different PEs have different types and different compatible communication protocols, which are displayed in Table 2. Because of behavior-PE mapping is determined, we also compute the computation time of each process on the mapped PEs , which is shown in Table 3.

In the bus library, there are 9 predefined buse types as shown in Table 4. Column *Protocol* displays the communication protocols of bus types. Column *Freq(M)* displays the frequency of the bus masters. Column *Width* displays the bus widths. Column *Delay(clock)* displays the required clock cycles per bus transfer. Column *Delay(clock/word)* displays the required clock cycles per word(32bit) transfer. Column *Speed* displays the required data transfer time per word(32bit). Finally, Column *Cost* displays the cost of the buses. Above attributes of bus types are stored in the bus library. We also computed *Delay(b, t)* described in sec-

PE name	CF1	CF2	HW1	HW2	HW3	DSP11	DSP12	DSP13	DSP21	DSP22
Type	ColdFire	ColdFire	Custom hardware	Custom hardware	Custom hardware	IP	IP	IP	IP	IP
Protocol	ColdFire Protocol	ColdFire Protocol	Any	Any	Any	Hand-shake	Hand-shake	Hand-shake	FIFO	FIFO

Table 2. The communication protocols of the selected PEs in 10-PE example

Process name	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Mapped PE	CF1	DSP11	HS1	HS2	DSP21	CF2	HW3	DSP12	DSP13	DSP22
Comp. Time(ms)	1.19	3.46	0.75	0.75	10.36	21.35	3.34	0.8	13.37	13.52

Table 3. The computation time of processes in 10-PE example

Bus Name	Protocol	Freq(M)	Width(bit)	Delay(clock)	Delay(clock/word)	Speed(MWord/S)	Cost(unit)
CF1	Coldfire	256	32	2	2	128	128
BusHS1	Handshake	128	32	2	2	64	64
BusHS2	Handshake	128	16	2	4	32	32
BusHS3	Handshake	128	32	4	4	32	32
BusHS4	Handshake	128	16	4	8	16	16
BusFF1	FIFO	128	32	2	2	64	64
BusFF2	FIFO	128	16	2	4	32	32
BusFF3	FIFO	128	32	4	4	32	32
BusFF4	FIFO	128	16	4	8	16	16

Table 4. The bus types in the bus library for 10-PE example

Constraint(ms)	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	Design
Set1	2	10	2	3	15	33	8	4	18	25	120
Set2	2	10	2	3	20	35	8	4	18	25	127
Set3	2	10	2	3	20	40	8	4	18	30	137
Set4	2	15	2	3	20	40	10	4	20	30	146
Set5	2	20	2	3	20	45	10	4	25	30	161
Set6	2	25	2	3	20	55	10	4	30	35	186

Table 5. The sets of time constraints of processes in 10-PE example

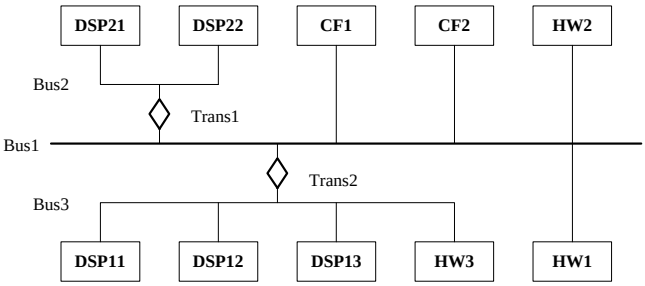


Figure 15. The generated interconnection topology for 10-PE example.

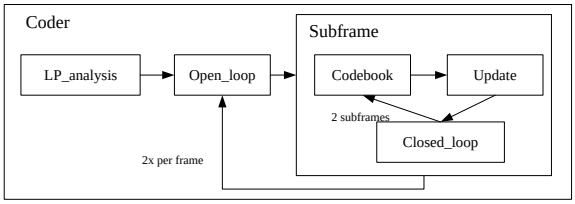


Figure 16. Block diagram of encoding part of vocoder

tion 9.1 for all the data type *t*.

There are six sets of design's time constraints, which are shown in Table 5. The sets are sorted in the increasing order of time constraint. For each set of time constraint, we generate an interconnection topology. We use the fast exploring algorithm. Therefore, the generated bus topologies for all the sets of constraints are the same, which is displayed in Figure 15. The interconnection topology contains three buses: *bus1*, *bus2*, and *bus3*. *Bus1* uses *Coldfire* communication protocol. *Bus2* uses *FIFO* communication protocol. *Bus3* use *Handshake* communication protocol. The interconnection topology also contains two transducers: *trans1* and *trans2*. We assume that the cost of each transducer is 200 units.

Finally, we select bus types from bus library for each bus in the interconnection topology. Table 6 displays the selected bus types and the added design costs for different sets of the time constraints. The added design cost contains not only the bus cost bus also the transducer cost. The percentage in the table denotes the bus utility of bus *b*, which equals the *AvgChanRate(b)* divided by *BusRate(b)*. We found that the added design cost for the set with longer time constraint is smaller than the design cost for the set with shorter time constraint. The total communication time of each process for different sets of design constraints is displayed in Table 7. The total execution time containing both computation time and communication time of each process is displayed in Table 8.

Besides implementing channel mapping automatically by using *channel-mapper*, we also implement the channel mapping manually with set1 as the time constraint. The results of automatic and manual implementation are the same. The design time of manual implementation is 2.5 hours. On the other hand, the design time of automatic implementation using *channel-mapper* is around 3 minutes, which is mainly spent on inputting the specification and linking the bus library. The simulation time of *channel-mapper* only takes less than 1 second. The speed up from 2.5 hours to 3 minutes allows designers to explore a much larger amount of *behavior-PE mapping* and *variable-memory mapping* alternatives.

11.2 Vocoder Project

The Vocoder [6] project implements the voice encoding part of the GSM standard for mobile telephony encoding standard. The block diagram of encoding part of vocoder is displayed in Figure 16. The project contains 13,000 lines of code. Vocoder encodes voice frame by frame. One frame contains two sub-frames. As shown in [6], the execution time of encoding one sub-frame must be less than 5ms. Processes *Closed_loop*, *Codebook*, and *Update* are executed once per sub-frame encoding. Process *Open-loop* is executed once when two sub-frames are encoded.

During behavior-PE mapping, we map processes *LP_analysis*, *Open_loop*, *Closed_Loop*, *Codebook*, and *Update* to different PEs as shown in Table 9. The attributes of the PEs are shown in Table 10. Since processes are executed on different PEs, we execute processes *Closed_loop*, *Codebook*, and *Update* in a pipeline fashion. We assign the design constraint to processes as shown in Table 11. The time constraint of *Closed_loop*, *Codebook*, or *Update* is 2.5ms, which ensures that total execution time for *Closed_loop*, *Codebook*, and *Update* per sub-frame is less than 3ms when we execute them in a pipeline fashion. We set the time constraint of *Open_loop* as 4ms, which ensures *Open_loop* can complete each sub-frame in 2ms. We also derive time constraint for *LP_analysis* as 7ms according to [6]. The total encoding constraint per frame is 20ms(Besides process *Coder*, processes *Pre_process* and *Post_process* are also part of encoding, which are not displayed in Figure 16).

Figure 17 displays the generated interconnection topology. PEs *DSP1*, *DSP2*, *ASIC1*, and *ASIC2* are directly connected to one bus(*Bus2*) because of their heavy inter-PE traffic. A transducer *Trans1* is inserted between *Bus1* and *Bus2*. *Bus2* has the *Handshake* protocol, while *Bus1* has the *ColdFire* protocol.

The bus types in the bus library are shown in Table 12. Our algorithm selects *CF1* for *Bus2* and *BusHS4* for *Bus1*. The bus utility for *Bus1*/*Bus2* is 9.44%/0.13% respectively.

Set	Bus1	Bus2	Bus3	Added design cost(unit)
Set1	CF1(1.82%)	BusHS1(1.18%)	BusFF1(0.55%)	652
Set2	CF1(1.72%)	BusHS1(1.12%)	BusFF2(1.04%)	620
Set3	CF1(1.59%)	BusHS1(1.03%)	BusFF4(1.93%)	604
Set4	CF1(1.49%)	BusHS3(1.94%)	BusFF2(0.91%)	592
Set5	CF1(1.40%)	BusHS3(1.82%)	BusFF4(1.70%)	576
Set6	CF1(1.17%)	BusHS4(3.05%)	BusFF4(1.42%)	560

Table 6. The bus type selection for 10-PE example

CommTime(ms)	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Set1	0.32	6.25	1.19	1.6	4.3	10.8	4.44	0.75	4.44	7.1
Set2	0.32	6.25	1.19	1.6	4.3	13.05	4.44	0.75	4.44	6.25
Set3	0.32	6.25	1.19	1.6	6.01	17.55	4.44	0.75	4.44	15.58
Set4	0.32	11.25	1.2	2.1	4.3	18.05	6.66	1.25	6.66	9.35
Set5	0.32	11.25	1.2	2.1	6.02	22.55	6.66	1.25	6.66	15.58
Set6	0.32	21.25	1.2	2.1	6.02	32.55	6.66	1.25	6.66	15.58

Table 7. The communication time of processes after the bus type selection for 10-PE example.

ExecTime(ms)	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Set1	1.51	9.71	1.94	2.35	14.66	32.15	7.78	1.55	17.81	20.62
Set2	1.51	9.71	1.94	2.35	14.66	34.4	7.78	1.55	17.81	19.77
Set3	1.51	9.71	1.94	2.35	16.37	38.9	7.78	1.55	17.81	29.1
Set4	1.51	14.71	1.95	2.85	14.66	39.4	10	2.05	20.03	22.87
Set5	1.51	14.71	1.95	2.85	16.38	43.9	10	2.05	20.03	29.1
Set6	1.51	24.71	1.95	2.85	16.38	53.9	10	2.05	20.03	29.1

Table 8. The execution time of processes after the bus type selection for 10-PE example.

Processes	LP_analysis	Open_Loop	Update	Codebook	Closed_Loop
PEs	CF1	ASIC1	DSP1	ASIC2	DSP2

Table 9. The behavior-PE mapping relations in vocoder project.

PEs	Coldfire	ASIC1	DSP1	ASIC2	DSP2
Type	Coldfire	Custom hardware	IP	Custom hardware	IP
Comm. Protocol	Coldfire	Any	Handshake	Any	Handshake
Frequency	60M	60M	60M	100M	60M

Table 10. The attributes of selected PEs in vocoder project.

Processes	LP_analysis	Open_Loop	Update	Codebook	Closed_Loop	Total
PEs	7ms	4ms	2.5ms	2.5ms	2.5ms	20ms

Table 11. The time constraints of processes in vocoder project.

Name	Protocol	Freq(M)	Width(bit)	Delay(clock)	Delay/Word	Speed(MWord/S)	Cost(unit)
CF1	Coldfire	60	32	2	2	128	128
BusHS1	Handshake	60	32	2	2	64	64
BusHS2	Handshake	60	16	2	4	32	32
BusHS3	Handshake	60	32	4	4	32	32
BusHS4	Handshake	60	16	4	8	16	16

Table 12. The bus types in the bus library for vocoder project

Processes	LP_analysis	Open_Loop	Update	Codebook	Closed_Loop
Computation time	5.35ms	2.66ms	0.17ms	0.83	2.00
Communication time	0.06ms	0.01ms	0.05	0.03	0.20
Exec. time	5.41ms	2.67ms	0.22	0.86	2.20
Constraint	7ms	4ms	2.5ms	2.5ms	2.5ms

Table 13. The execution time of processes in vocoder project.

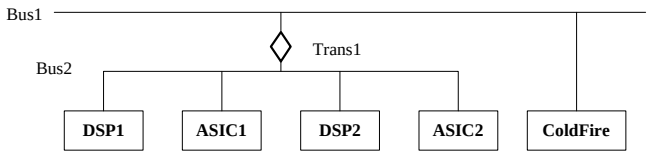


Figure 17. Interconnection topology of vocoder project

The computation time, the communication time, and the execution time per process are displayed in Table 13. The resulting bus topology generation and bus type selection meet the design constraint. The total cost of transducer and buses is 344 units, under the assumption that the cost of a transducer is 200 units.

12. Conclusion

This report introduces a novel design flow for the channel mapping problem. The design flow is tailored for the complex system architecture which contains hundreds of PEs with uncomputable communication protocols. The design flow covers the areas from the interconnection topology generation to the bus speed configuration. Especially, it takes the transducer insertion into account. The design flow has been integrated to our entire system level design flow.

We also introduce algorithms to implement channel mapping steps. Especially, during the step of *bus type selection*, our algorithm not only ensures that the maximum bus traffic rate is greater than the channel traffic rate over the bus, but also ensures that the constraint of each process can be met. During communication time computation, we take the delay of transducer into account. We produce *channel-mapper* to implement our algorithm automatically.

We first use a randomly generated 10-PE example to test our approach. This example proves that under different given design constraints, the generated channel mapping solutions are different. The costs of the generated solutions are reduced while the constraints of design are extended. It also shows that *channel-mapper* reduces the design time from hours to several minutes, thus allowing designers to explore a much larger amount of *behavior-PE mapping* and *variable-memory mapping* alternatives than without *channel-mapper*. We also apply our approach on the vocoder project, which proves the approach's correctness on the real design project.

References

[1] SystemC, OSCI[online]. Available: http://www.systemc.org/.

[2] L. Cai and D. Gajski. Introduction of Design-Oriented Profiler of SpecC Language. Technical Report ICS-TR-00-47, University of California, Irvine, June 2001.

[3] Jean-Marc Daveau, Gilberto Fernandes Marchioro, Tarek Ben-Ismail, and Ahmed Amine Jerraya. Protocol selection and interface generation for hw-sw codesign. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 136–144, March 1997.

[4] D. Gajski, N. Dutt, S. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[5] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[6] A. Gerstlauer, S. Zhao, and D. Gajski. Design of a GSM Vocoder using SpeccC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, Feb 1999.

[7] Peter Voigt Knudsen and Jan Madsen. Integrating communication protocol selection with hardware/software codesign. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1077–1095, August 1999.

[8] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Efficient exploration of the soc communication architecture design space. In *Proceedings of the International Conference on Computer-Aided Design*, 2000.

[9] Sanjiv Narayan and Daniel D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proceedings of the Design Automation Conference*, pages 468–473, June 1995.

[10] Ross B. Ortega and Gaetano Borriello. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, 1998.

[11] Ti-Yen Yen and Wayne Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, 1995.