

Application-Specific Interconnection Network Design in Clustered DSP Processors

Cathy Qun Xu Youtao Zhang Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083

Abstract

To meet increasing performance requirements of DSP applications, application specific processor designs, e.g. function unit (FU) duplication and register file (RF) distribution, are widely used in the design of DSP processors. In this paper, an application specific approach is proposed for the design of interconnection network in such DSP processors. By extracting the scheduling information of DSP applications, we decide the minimal number of required partially connected buses. Without impacting the performance and increasing the hardware cost, it provides optimized future scheduling flexibility. Our results show that reductions of 40% in the number of required global buses and 60% in wire segments can be achieved.

Key words: Clustered processors, Architecture, Interconnection network.

1 Introduction

Signal processing applications become increasingly complicated. To meet their tight performance requirements, high-end DSP processors use function unit (FU) duplication to exploit the maximal instruction level parallelism in these programs, i.e. multiple copies of one type of FU exist in the system and thus multiple instructions of the same type can be executed at the same time. However, these FUs may access operands from the register file (RF) simultaneously and incur great contentions for the RF. To simplify the hardware design, these FUs are grouped into clusters and accordingly the system level RF is divided into several subbanks. While each cluster can read/write its associated RF subbank with low cost, an inter-cluster access passes through an interconnection network and thus is much more expensive.

Figure 1 shows the block diagram of TI TMS320C62x [5]. It contains two datapaths (clusters) and the register file is divided into two subbanks. FUs on datapath (cluster) A can access register file A directly within one cycle. If these FUs want to access register file B, one pipeline stall is inserted by the hardware automatically. Inter-cluster data transfers in TI TMS320C62x use two buses ($1\times$, $2\times$). How-

ever, at any given time, only one data item can be transferred on each bus, e.g. from register file A to datapath B. The buses in TI TMS320C62x connect 6 out of 8 functional units.

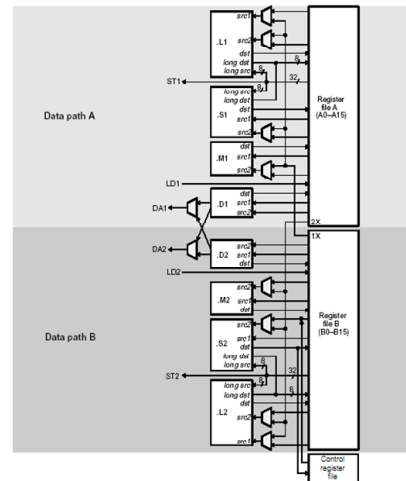


Figure 1. TI TMS320C62x Block Diagram.

The interconnection network between clusters can become a bottleneck and hurt the overall performance. As we will see in this paper, unrolling the kernel loop can exploit the parallelism in the program. However, inter-cluster data transfers increase significantly and limit the peak performance that the processor can achieve. While compilation based approaches have been recently proposed to re-schedule the program and reduce the inter-cluster data transfers [6, 7], we take a hardware approach in this paper, that is, an application specific interconnection network design is proposed to reduce the hardware cost without performance degradation. As we know, to reduce the overhead of scheduling on clustered DSP processors, DSP applications are usually statically compiled with explicit scheduling information embedded in the code. For example, using VLIW code, each instruction contains 4, 8 or more operations which explicitly specify the function units and associ-

ated registers. On the other hand, many embedded systems are designed to run a limited number of applications, application specific processor designs can thus take advantage of these applications to reduce the hardware cost.

The interconnection network between function units and register files in different clusters could be crossbar network, i.e., there is a physical connection between each function unit and each register file. While it provides the maximal transfer flexibility, its high hardware cost prohibits its practical use. On the other hand, a fully connected bus that connects each dedicated register file and all function units in other data paths, has the potential to become a bottleneck. In addition, the bus speed is limited by the number of function units that are connected to it and thus the fully connection slows down the performance.

Several previous works solve the connection problem on datapaths in single processor and thus are related to our research. [4] addressed the interconnection synthesis problem similar to ours. The performance-constrained algorithms are proposed to minimize the number of buses. Integer linear programming models are also proposed in the paper to solve the problem optimally. However, since we are using different processor models, the data transfer in their model is bound to a particular cycle while it is flexible in our model. [2] explored the interconnect architecture between the processor and the memory. [1] focused on operation assignment on a given VLIW processor and made the assumption that the interconnection network is given. All of these works are at the single cluster level. Compilation based approaches [6, 7] are recently proposed to take existing interconnection network constraints and optimize the scheduling under these constraints. Our approach, however, removes the interconnection constraints and increases the flexibility in software scheduling. As shown in this paper, our approach is orthogonal to compilation optimizations and thus can be used together with advanced scheduling techniques.

The rest of the paper is organized as follows. We model DSP applications as well as target processor architectures in section 2. The proposed approach is motivated and briefly discussed in section 3 and discussed in great more details in section 4. The experimental results are given in section 5. Section 6 concludes the paper.

2 Fundamentals

2.1 Modeling DSP applications

DSP applications spend most of their execution time on loops [6]. For simplicity, this paper focuses on basic blocks of kernel loops. Our proposed approach can be extended to kernels by taking control flows into consideration.

A basic block in a DSP application is modeled as a *data flow graph* (DFG) in this paper. It is a directed weighted graph $G = (V, E, d, t)$ where V is the set of computation nodes (instructions); E is a set of edges that defines the data

dependency relations, i.e. $E = f : V \rightarrow V$; $d(e)$ is a number of delays (registers) for an edge $e \in V$. The set of edges without delay composes a Directed Acyclic Graph, which represents data dependencies within the same iteration.

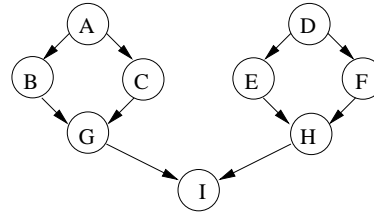


Figure 2. A Directed Acyclic Graph G

For illustration purpose, a simple DAG is shown in Figure 2 and we use it throughout this paper. This DAG example consists of 9 operations. Each operation is represented by a node labeled as A, B, etc. All these operations are additions in this example although they can be additions, multiplications or other operations in a real program. A directed edge represents a data dependency from the operation that generates a data item to the operation that consumes the data item. As it is shown in the Figure 2, operation G depends on the outputs of operations B and C. Since there are no control flow edges, the DAG is thus acyclic.

2.2 Modeling Clustered DSP Processors

Figure 3 illustrates the target processor model which contains a set of n DSP clusters. Each cluster contains multiple functional units and one register file. The functional units in one cluster can access its register file with low cost. However, when a FU wants to access a remote register, the content is transferred from an interconnect network(CN). It is more expensive and slow.

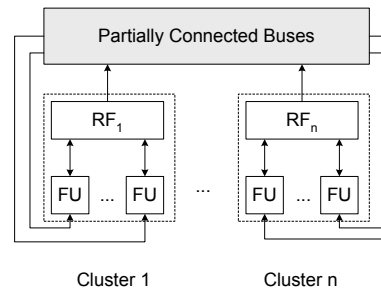


Figure 3. Target Architecture

In this paper, our goal is to optimize the interconnection network for given pre-scheduled DSP applications. In a pre-scheduled DSP application, all operations are explicitly assigned to FUs and they are executed in their program order. Multiple inter-cluster data transfers might occur at the same time and the interconnection network needs to be carefully designed to eliminate the possible delays. For example, while FU_1 in cluster 2 fetches an operand from RF_1 , FU_3 may want to fetch an operand from RF_2 . Two straightforward connection schemes exist to satisfy the transfer re-

quirement. The first one is to connect RF_1 to FU_1 in cluster 2 and RF_2 to FU_3 in cluster 2 respectively. The general form of this approach is to connect each RF and each FU using a different connection segment, i.e. a crossbar network. Clearly there would be too many connections and the high cost would prohibit its practical use.

The second design is to use buses. Since we have two simultaneous transfers, two buses are needed. Each bus connects all register files and all functional units. In general, the number of buses is the maximum number of inter-cluster data transfers that can occur in a scheduled program. While this number is not very big, there are too many connections points on each bus and results in longer bus delay and high physical area cost.

In this paper, we use an application specific processor design (ASIP) approach and propose a partial connected bus connection scheme which (1) reduces the required buses for pre-schedule applications; (2) optimizes the connection points on each bus without introducing any performance degradation.

3 Algorithm Overview

In this section, we motivate and briefly discuss our proposed approach. The algorithm details will be discussed in the following section.

For illustration purpose, we assume that the given architecture consists of two clusters and each cluster contains one *adder* FU. Each is associated with a RF as it is shown in Figure 3. Transferring data from one to another would require a partially bus which maybe shared by multiple FUs and RFs. At any given time, only one data transfer is possible on each bus.

3.1 List Scheduling

The inputs are DSP applications scheduled by a traditional list scheduling algorithm. The algorithm first numbers all functional units and then schedules an operation to the next available FU of the same type with smallest index. It gives more priority to local FUs during the scheduling and achieves fast scheduling by using the heuristic to schedule each operation as early as possible.

The detailed list algorithm works as follows. An operation (instruction) is considered as a ready operation at control step cs if it does not depend on any other operations or all of its dependable operations have generated the results. As we scan the program, ready instructions are inserted into a *list* and ordered according to the number of its dependent operations, i.e. how many other operations that depend on it. At any control step cs , the ready node of the highest priority is selected and removed from the list. The rest are scheduled in later step.

For example, suppose node B and H in the Figure 2 are both ready for scheduling. Since node G and I are all depending on the B's output and the longest length from B to

I is 3. The node H only has one child node of I and the longest length from H to I is 1. Thus B will be scheduled before H will.

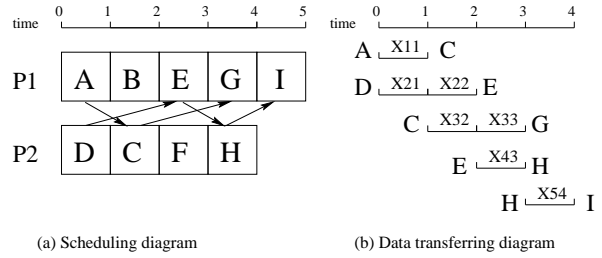


Figure 4. DAG G's Corresponding Scheduling and Data Transferring Diagram

By applying the list scheduling algorithm on the sample program, we get the scheduling scheme as Figure 4(a). Operation node A, B, E, G and I are assigned to functional unit P1 while the operation nodes D, C, F and H are assigned to P2. The lines with arrows indicate that the data transfers are required during the operations. For example, there is an arrow pointing to A from C which represents the data transfer from functional unit P1 to functional unit P2 during first control step after node A has generated its output data. Node D also produces input data for node E. Thus there is another line with arrow pointing to E from D which indicates that a data transfer is required before node E starts its computation. So the data transfer can happen either in the control step 1 after node D generated its output or in the control step 2 before node E starts its computation. Please note that no data transferring is required if two dependable nodes are scheduled on the same functional unit. For example, as it is shown in Figure 2 that there is a data dependency between node A and B, no data transfer is necessary since these two nodes are scheduled on the same functional unit.

3.2 Our Motivation

In the above scheduling, we assume unrestricted interconnection network, or, there is always an free communication channel when a data transfer takes place. While it is possible to fix the interconnection network beforehand and take it into the scheduling decision, it clearly limits the flexibility and hurts nevertheless the performance. Instead, we take another approach in this paper. The scheduling algorithm can still schedule the operations without network constraints. We take the scheduled applications as inputs and design the interconnection network with minimal hardware cost to maintain the desired performance.

Our key observation is that in the list scheduling algorithm, all data items are transferred immediately after they are generated. However, that is not necessary. Actually, it is possible to transfer the data item anytime after it is generated and before it is used. Assuming there is no register file pressure, the data item is first written to a local temporary register and could be transferred later. In this way, we can

greatly reduce the burst bus requirement by balancing the transfers over the time.

For this purpose, a data transfer diagram is built in which cycles are explicitly shown as control steps (X-axis in Figure 4(b)). We also mark operations and all control steps that a data transfer can possibly take place at that step. The line segments under each control step in Figure 4(b) represent the possible data transfers. The left operation is always a sender and a right node is always a receiver. For example, X_{11} represents the data transfer from node A to node C during the first control step. X_{21} and X_{22} represent the possible data transfer from node D to node E. Since data transferring takes only one control step, one and only one of X_{21} or X_{22} can be 1. As we discussed, if at the control step 2, the data transfers from D to E and from C to G all happen, a total of 2 buses is required to fulfill the transfer.

A carefully analysis shown that if we assign the data transfers from A to C and from D to E in control step 1, assign the data transfers from C to G in control step 2 and assign the data transfer from E to H in control step 3, we need only 2 buses. This is a simple example which can be easily analyzed. For complicated cases, we will develop a systematic approach and discuss it in details in section 4.

3.3 Optimized List Scheduling

As we discussed, scheduling optimization can greatly reduce the number of global buses in the interconnection network. However, our approach is orthogonal to scheduling and can be used in combination on existing scheduling techniques.

In this section, we propose an optimized list scheduling algorithm and it is used in our experiments. An improvement is made during the step of assigning operations. While in the traditional list scheduling, an operation is assigned to the available FU in fixed order, we instead pick up the FU which has the majority of its parent operations. This heuristic minimizes the inter-cluster data transfers and lowers the number of required global buses. For example, after applying this optimization, the scheduling and its corresponding data transfer diagram are shown in Figure 5 which reduces the number of required global buses from 2 to 1 as compared to Figure 4.

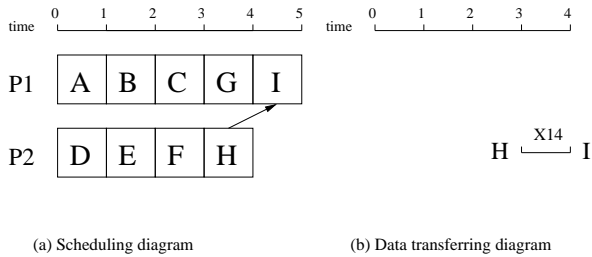


Figure 5. Optimized Scheduling and its Corresponding Data Transferring Diagram

4 Optimizing Interconnection Networks

We discuss our approach to optimize the interconnection network in this section. The approach is divided into two steps: we first model the scheduling information in an integer linear programming solver. It is then solved in polynomial time and decide the minimum number of global buses without performance degradation. We then decide the topology of the interconnection network based on the schedule model created in the first step.

4.1 Determine the Data Transfer Scheme

Our algorithm is based on the application and architecture model described in section 2. We follow the two constraints discussed in the previous section: (1) time constraint. That is, given a statically schedule DSP application, inter-cluster data transfers are flexible as long as the operands are transferred after they are produced and before they are consumed. (2) bus constraint. That is, at any given time, the number of simultaneously data transfer can take up to the number of buses.

Algorithm 1 extracts the time constraint from the pre-scheduled DSP application and create a data transfer diagram. The input is a scheduling diagram as shown in Figure 4. It generates a list of transfer variables noted as $x_{i,j}$ which indicates that the i -th data item transfer could happen at j -th control step. If $x_{i,j} = 1$, the data transfer does happen for i -th data transfer at step j . Otherwise, it is not.

Proc-Construct_Data_Transfer_Diagram

```

1  edgeList ← all edges on DAG G
2  transferList ← 0; i ← 0
3  for each edge e in the edgeList
4      PE(N1) ← Processor Unit on which
        the producing node N1 is scheduled
5      PE(N2) ← Processor Unit on which
        the consuming node N2 is scheduled
6      if PE(N1) ≠ PE(N2)
7          i increases by 1;
8          cs_start ← control step of N1 completes
9          cs_complete ← control step of N2 starts
10         for each j that cs_start ≤ j < cs_complete
11             transferList ← InsertList(xi,j, transferList)
12  return transferList

```

By applying this algorithm, we will create a list of $X_{i,j}$ as shown in Figure 4(b). For illustration purpose, we use the traditional scheduling as the input in this section. Our experiments take the optimized scheduling information as the input.

After creating the list of $X_{i,j}$, we add bus constraint and formulate it as follows. (1)-(5) indicates each data transfer takes one cycle. And (6)-(9) indicates at most b simultaneous transfers are possible.

$$\begin{aligned}
x_{11} &= 1 & (1) \\
x_{21} + x_{22} &= 1 & (2) \\
x_{32} + x_{33} &= 1 & (3) \\
x_{43} &= 1 & (4) \\
x_{54} &= 1 & (5)
\end{aligned}$$

as every data transfer takes only one control step. And

$$\begin{aligned}
x_{11} + x_{21} &\leq b & (6) \\
x_{22} + x_{32} &\leq b & (7) \\
x_{33} + x_{43} &\leq b & (8) \\
x_{54} &\leq b & (9)
\end{aligned}$$

(1)-(9) can be rewritten to as follows.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_{11} \\ x_{21} \\ x_{22} \\ x_{32} \\ x_{33} \\ x_{43} \\ x_{54} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ b \\ b \\ b \\ b \end{pmatrix} \quad (10)$$

where s_1, s_2, s_3 and s_4 are positive integer values.

Suppose we define

$$Y^T = (x_{11}, x_{21}, x_{22}, x_{32}, x_{33}, x_{43}, x_{54}, s_1, s_2, s_3, s_4)$$

$$V^T = (1, 1, 1, 1, 1, b, b, b, b)$$

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$D = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$C = \begin{pmatrix} B & O \\ D & I_4 \end{pmatrix}$$

Thus equation (10) can be simplified as $CY = V$.

Solving equation (10), we obtain $b = 2$,
 $X^T = (1, 1, 0, 1, 0, 1, 1)$ where $X^T = (x_{11}, x_{21}, x_{22}, x_{32}, x_{33}, x_{43}, x_{54})$

The following summarize the general procedure of solving the minimum number of global buses and it's related CN topology.

- Derive Data Transfer Diagram(DTD) on given DAG
- Define vector $X^T = (x_{11}, \dots, x_{ij}, \dots, x_{mn})$ where m is the number of transfers and n is the number of control steps. $1 \leq i \leq m$ and $1 \leq j \leq n$.
- Define vector $S^T = (s_1, \dots, s_j, \dots, s_n)$ where S_j is non negative integer and $1 \leq j \leq n$
- Define vector $Y^T = (X^T, S^T)$
- Define vector $V = (1, \dots, 1, b, \dots, b)$. There are m 1's and n b's in the vector V.
- Construct matrix C based on the properties of the DTD
 - Construct matrix B based on each data transfer i, $\sum_j x_{ij} = 1$
 - Construct matrix D based on each control step j, $\sum_i x_{ij} \leq b$
 - $x_{ij} \in 0, 1$ for every i, j and $b \leq 0$

$$C = \begin{pmatrix} B & O \\ D & I_3 \end{pmatrix}$$

- Solving $CY = V$ for X – Integer Linear Programming in polynomial time

After modeling the constraints in matrix format, clearly it is an linear programming problem. Although a general integer linear programming problem is NP-hard, we will shown in the following that given the properties of matrix C, $CY = U$ is solvable in polynomial time.

DEFINITION 4.1 A square, integer matrix B is called unimodular (UM) if its determinant $\det(B) = \pm 1$. An integer matrix A is called totally unimodular (TUM) if every square, nonsingular submatrix of A is UM.

THEOREM 4.1 An integer matrix A with $a_{ij} = 0, \pm 1$ is TUM if no more than two nonzero entries appear in any column, and if the rows of A can be partitioned into two sets I_1 and I_2 such that:

1. if a column has two entries of the same sign, their rows are in different sets;
2. if a column has two entries of different signs, their rows are in the same set.

The matrix C in equation 10 has exactly the properties described in the above theorem since

$$\sum_{j=j}^{j+1} x_{i,j} = 1, \quad 1 \leq i \leq m \quad (11)$$

$$\sum_{i=1}^m x_{i,j} \leq b, \quad 1 \leq j \leq n \quad (12)$$

Equation (11) simply reiterates the fact that each data transfer takes one control step. Thus, for each row in the matrix C, only one variable will be 1 and the rest will all be 0s. In equation (12), b is the number of shared global buses in question. Thus matrix C is total unimodular.

THEOREM 4.2 *If A is TUM, then all the vertices of $R_1(A)$ are integer for any integer vector b.*

Thus a standard form Linear Programming with TUM matrix will always lead to an integer optimum when solved by the simplex algorithm.

When an LP is formulated with inequality constraints, such as equation (12), the same result holds.

Let the corresponding polytope be

$$R_2(A) = \{x : Ax \leq b, x \leq 0\} \quad (13)$$

Then we have the next theorem.

THEOREM 4.3 *If A is TUM, then all the vertices of $R_2(A)$ are integer for any integer vector b.*

The proof of the above theorems can be found in [3]. Based on the above theorems, it's not difficult to prove that the matrix C is TUM as it satisfies the conditions specified in Theorem 4.1. Thus, determining the minimum number of global buses is not a general NP complete problem and it takes polynomial time to find the optimal solution.

4.2 Determining the Connection Scheme

Besides the minimum number of global buses we calculated from the first step, an application specific interconnection network design has to determine how these FUs and RFs are connected. We will exploit the data transfer information and determine the topology to reduce the hardware cost while achieve maximal future flexibility.

To determine if a connection from a FU or RF to a bus is necessary in interconnection network, we use the data transfer information from the first step. An algorithm is developed and shown as follows.

DETERMINE CN TOPOLOGY (G, S, DTD, b)

```

1  cs ← 0
2  Bus[i] ← 0, i from 1 to b
3  N ← total number of data transfers
4  repeat
5      for j = 1 to N
6          if  $x_{cs,j} = 1$ 
7              then FU1 ← processor of Producer( $x_{cs,j}$ )
8                  FU2 ← processor of Consumer( $x_{cs,j}$ )
9                  for i = 1, ..., b
10                     if FU1 or FU2 ∈ BUS[i]
11                         Bus[i] ← InsertList( $x_{cs,j}$ , Bus[i])
12                         continue;
13                     Bus[1] ← InsertList( $x_{cs,j}$ , Bus[1])
14             cs ← cs + 1
15 until cs > maximum control step of the scheduling length
16 output Bus[i], i from 1 to b

```

It maintains b buses and try to reduce the connection points by reusing previous ones. If a data transfer from A to B happens at some step, it checks if A or B has been to connected to a bus. If yes and that bus is idle, we can reuse it. If one of them is in a bus, we can add one more connection point. If there is no such bus or the bus is busy, then we can connect A and B to a free bus.

5 Experimental Results

We have implemented and evaluated the proposed algorithms on a set of digital signal processing applications. The optimized list scheduling algorithm (discussed in section 3) is implemented and used to generate the input scheduling scheme. The processor architecture is based on Figure 3. To exploit the maximal instruction level parallelism, we unfold the loop several times (indicated in the results) which can use up to 15 clusters. For the first 5 clusters, each contains an adder and a multiplier. For the rest, each contains an adder.

The benchmarks we used are all digital filters. 4-lat refers to 4 lattice filter while biquad refers to the biquadratic digital filter. A suffix indicates the number of times the program is unfolded, e.g. *uf2* means the loop is unfolded 2 times.

Benchmarks	Before Optimization	After Optimization
4-lat	11	3
4-lat-uf2	13	4
4-lat-uf3	10	5
biquad	2	2
biquad-uf2	4	4
biquad-uf3	6	4
er-lat-uf2	17	11
er-lat-uf3	13	10
elf-uf2	5	3
iir-uf2	5	4
iir-uf3	8	5
rls-lat-uf2	9	5
volt	5	1
volt-uf2	13	2
volt-uf3	11	3
Average	7.8	4.35

Figure 6. Number of Partially Connected Buses.

Figure 6 compares the required global buses with and without proposed interconnection network optimization. In the first column, we list benchmarks in the experiments. Column 2 reports the maximal number of buses for the pre-scheduled data transfer scheme without introducing any performance degradation. Column 3 reports the reduced number of buses after optimization. For example, 4 lattice filter, 11 and 3 buses are required before and after optimization respectively. On average, the algorithm reduces the maximal number of buses by 44.2%. Thus we conclude the proposed algorithm is very effective in removing bus contentions.

With the fixed number of global buses, we then compare the number of connection segments in the network with dif-

Benchmark	Fully Connected Buses	Connection in Pairs	Partially Connected Buses
4-lat	39	11	11
4-lat-uf2	56	22	22
4-lat-uf3	70	25	25
biquad	6	2	2
biquad-uf2	28	4	4
biquad-uf3	36	6	6
er-lat-uf2	88	23	23
er-lat-uf3	88	23	23
elf-uf2	9	8	8
iir-uf2	32	6	6
iir-uf3	50	9	9
rls-lat-uf2	20	10	10
volt	10	10	10
volt-uf2	28	16	16
volt-uf3	42	21	21
Average	34	12	12

Figure 7. Segments in the Network.

ferent connection topology (Figure 7). Column 1 lists the benchmarks. Column 2 reports the number of segments when using fully connected global buses. Column 3 reports the results when each connection is setup between a register file and a functional unit, i.e. in pairs. Column 4 reports the results when using partially connected global buses. We can see from the table that different topologies can greatly affect the number of connection segments in the network. On average connection in pairs or using partially connected buses can reduce 64.7% of connection segments. Since the number of segments affects the hardware layout as well as the wire connection, it is beneficial to have a small number of connection segments. Applying an application specific design approach greatly reduces this number and thus optimizes DSP processor designs.

Benchmarks	Partially Connected Buses (A)	Connection in Pairs (C)	Ratio of C over A
4-lat	480	11	0.023
4-lat-uf2	45360	22	0.00049
4-lat-uf3	209160	25	8.6e-5
biquad	2	2	1
biquad-uf2	4	4	1
biquad-uf3	65	6	0.09
er-lat-uf2	297	23	0.0774
er-lat-uf3	11292	35	0.0031
elf-uf2	27	8	0.296
iir-uf2	8	6	0.75
iir-uf3	33	9	0.27
rls-lat-uf2	147	10	0.068
volt	2e7	10	5e-7
volt-uf2	362880	16	4.41e-5
volt-uf3	2.4e8	21	7e-11

Figure 8. Network Connectivity.

From the previous experiment, we conclude the interconnection network should either take partially connected buses or have connection in pairs. We then do the experiment to compare their connection flexibility. Figure 8 compares the possible connection choice at any given time. A connection choice is a data transfer from a register file to a functional unit. With partially connected buses, any regis-

ter files can send to any functional units if both parties are connected to the bus. With pair connection, there is only one choice for each connection. All connection segments in pair connection could be used simultaneously while only one transfer can take place on each bus at any given time. However, bus connection simplify the the hardware design and the performance bottleneck has been removed through ILP solver.

6 Conclusion

In this paper, we propose an application specific approach for the design of interconnection network in clustered DSP processors. Using integer linear programming, it determines a minimal number of partially connected buses in polynomial time. The approach further determines an optimized connection scheme which increases the connection flexibility without causing further hardware cost.

References

- [1] M. Bekooij, "Phase Coupled Operation Assignment for VLIW Processors with Distributed Register Files," *Proc. ISSS'01*, pp. 118-123, October 2001.
- [2] T. V. Meeuwen, A. Vandecappelle, A. v. Zelst, F. Catthoor and D. Verkest, "System-level Interconnect Architecture Exploration for Custom Memory Organizations," *ISSS'01*, Montreal, Quebec, Canada, October 1 - 3, 2001, pp. 13-18
- [3] C. H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization, Algorithms and Complexity*, Dover Pubns, 1998.
- [4] Y.M. Jiang, T.F.Lee, T.T. Hwang and Y.L. Lin, "Performance-Driven Interconnection Optimization for Microarchitecture Synthesis," *IEEE Transactions on computer-aided design of integrated circuits and systems*, Vol 13(2), February 1994.
- [5] Texas Instruments. *TMS320C6000 CPU and instruction Set Reference Guide*, Literature number SPRU189.
- [6] Y.Qian, S.Carr and P.Sweany, "Loop Fusion for Clustered VLIW Architectures," *Proceeding of 2002 Joint Conference on LCTES and SCOPES*, pages, 112-119. 2002.
- [7] C.Kessler and A.Bednarski, "Optimal Integrated Code Generation for Clustered VLIW Architectures," *Proceeding of 2002 Joint Conference on LCTES and SCOPES*, pages, 102-111. 2002.