

國立交通大學

電子工程學系 電子研究所碩士班

碩士論文

針對在單晶片網路系統中任務群組之研究

On Task Clustering for Network-on-Chip Designs

研究生：雷永群

指導教授：周景揚 博士

中華民國九十四年一月

針對在單晶片網路系統中任務群組之研究
On Task Clustering for Network-on-Chip Designs

研 究 生：雷永群

Student: Yung-Chun Lei

指導教授：周景揚 博士

Advisor: Dr. Jing-Yang Jou

國立交通大學

電子工程學系 電子研究所碩士班



Submitted to Institute of Electronics
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE
in
Electronics Engineering

January 2005
Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 四 年 一 月

針對在單晶片網路系統中任務群組之研究

研究生：雷永群 指導教授：周景揚 博士

國立交通大學

電子工程學系 電子研究所碩士班



單晶片網路系統已成為研究晶片上系統之通訊的一個新趨勢。我們為單晶片網路系統發展了一個完整的設計流程，而本篇論文著重在單晶片網路設計流程中之任務群組部份。此任務群組問題需要考慮多重的限制，包括運算量，記憶體，輸出入以及整體通訊頻寬。在滿足所有的限制條件下，試圖減少整體群組之數量。我們導入一個演算法來解決這個問題。此演算法已被寫為一個自動化的程式，而我們也得到一些實驗結果。經由這個程式我們可以在數分鐘內減少群組數目並滿足所有條件限制。

On Task Clustering for Network-on-Chip Designs

Student: Yung-Chun Lei

Advisor: Dr. Jing-Yang Jou

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University

Abstract

The Network-on-Chip (NoC) has become a new trend in on-chip system communication design. We have developed a complete design flow for NoC system design. In this thesis we focus on the task clustering in our NoC design flow. The task clustering problem in the NoC system design needs to consider multiple constraints, which include computing power, memory, I/O and communication bandwidth. We introduce an algorithm to minimize the number of clusters in the result with all constraints being met. The algorithm has been written as an automatic program and some experimental results are presented. Using our program, we can reduce the number of clusters with all constraints satisfied in few minutes.

Acknowledgment

I would like to express my sincere gratitude to my advisor Professor Jing-Yang Jou for his suggestions and guidance throughout these years. Also, I would like to thank Cheng-Yeh Wang, Liang-Yu Lin for their help on the development of this work. Special thanks to EDA lab members, for their company and friendship. Finally, I would like to express my sincere gratitude to my parents and my wife Meng-Fang Tsai, who have always helped and encouraged me a lot.

Contents

摘要.....	I
ABSTRACT	II
ACKNOWLEDGMENT	III
CONTENTS.....	IV
LIST OF FIGURES.....	VI
LIST OF TABLES	VII
CHAPTER 1 INTRODUCTION	1
CHAPTER 2 PRELIMINARY	5
2.1 HARDWARE ARCHITECTURE	5
2.2 PE ARCHITECTURE	6
2.3 SWITCH ARCHITECTURE	7
2.4 SYSTEM DEVELOPMENT ENVIRONMENT.....	9

CHAPTER 3	A TASK CLUSTERING ALGORITHM.....	15
3.1	THE ALGORITHMIC FLOW	18
3.2	THE WEIGHT ANALYSIS STAGE	20
3.3	THE INITIAL SOLUTION GENERATION STAGE.....	23
3.4	THE DECOMPOSITION STAGE.....	25
3.5	THE REFINEMENT STAGE.....	28
CHAPTER 4	EXPERIMENTAL RESULTS.....	31
4.1	THE EXPERIMENTAL ENVIRONMENT	31
4.2	THE EXPERIMENTAL RESULTS	32
CHAPTER 5	CONCLUSIONS.....	37
REFERENCE	39
VITA	41



List of Figures


FIGURE 1. THE NoC ARCHITECTURE	6
FIGURE 2. THE PE ARCHITECTURE.....	7
FIGURE 3. THE SWITCH ARCHITECTURE.....	8
FIGURE 4. THE EXAMPLE OF VIRTUAL CHANNELS	9
FIGURE 5. A NoC SYSTEM DESIGN FLOW	10
FIGURE 6 THE FLOW CHART OF THE MAPPING PROCESS	11
FIGURE 7. AN TASK CLUSTERING EXAMPLE	17
FIGURE 8. THE FLOW OF OUR HEURISTIC ALGORITHM.....	18
FIGURE 9. THE PSEUDO CODE OF WEIGHT ANALYSIS STAGE	20
FIGURE 10. THE PSEUDO CODE OF THE DECOMPOSITION STAGE.....	25
FIGURE 11. THE PSEUDO CODE OF REFINEMENT STAGE.....	29
FIGURE 12. THE MEASURE POINTS OF OUR EXPERIMENT.....	33
FIGURE 13. THE EXPERIMENTAL RESULT	34

List of Tables

TABLE 1. THE EFFECT OF VARIANCE	35
TABLE 2. LIST OF RESULTS.....	錯誤! 尚未定義書籤。



Chapter 1 Introduction



With the advent of IC manufacturing technology, design of System-on-Chip (SoC) faces some challenges. While more and more functions are integrated into the SoC design, the architecture exploration becomes more complex than before. Because of the exponentially uprising of mask cost, the hardware modification is a critical concern. On the other hand, the global synchronization of clocks and signals will be difficult because that the wire delay has dominated the total delay. All these problems lead to a much longer design time and much higher design cost than before [1].

A technology called Network-on-Chip (NoC) was introduced in recent years [2][3][9]. In the NoC architecture, the chip is mainly composed of a set of processing elements. There is a switch connected to every processing element. And the network is constructed by the connection between the switches. By transmitting data across the network, processing elements can communicate with each other in a

NoC system.

The NoC system has several benefits making us believe that it can be a good choice of solving the problems of SoC design.

- 1) The NoC system is scalable. The same processing elements can be duplicated into an $N \times N$ array according to the system's requirement. Hardware design effort can be reduced, and the system design environment can be reused for different system scales [5][6].
- 2) The NoC system is reconfigurable. Most functions are implemented by software, thus the system can be adjusted without hardware modification. This can greatly reduce the cost of hardware development [3][5].
- 3) The NoC system is a Globally Asynchronous Locally Synchronous (GALS) system. The difficulty on wiring global synchronous signals can be avoided [2][12].
- 4) The NoC system has faster data transfer rate than the traditional bus architecture because of the parallel communication capability [8].

We have developed a complete design flow for NoC system design [13][14]. Like the multiprocessor design, the system design of NoC needs to parallelize the application code into tasks to be mapped to distributed processors. Compared to traditional parallel systems, the processors in NoC architecture have limited

resources, especially the memory. In order to maximize the utilization of NoC resources, considering all of the resource constraints while mapping the tasks into PEs is an important problem.

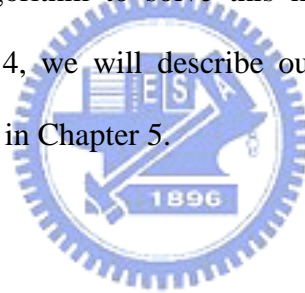
In this thesis, we focus on the task grouping step of our NoC development system. In this problem, we consider the constraints of the computing power, memory, I/O between processor and switch, and total communication. We try to minimize the number of clusters of the result with all constraints satisfied.

There are two kinds of traditional problem similar to the problems just mentioned before. One is called bin packing problem. The other is called partitioning problem [11]. The bin packing problem is defined as to put the most objects in the least number of fixed space bins, or to find a least number of bins to hold a set of objects. If we only consider computing power and memory constraints, then the problem can be reduced to the bin packing problem. However we need to consider the I/O and communication constraints, and these constraints based on the edge connection between the tasks can not be resolved by using bin packing algorithm.

The partitioning algorithm is used in parallel processing domain. It includes decomposition stage and assignment stage. In the decomposition stage, a program is separated into several tasks. In the assignment stage, tasks are put into processes. Our problem is also a kind of partitioning problem. However traditional partition problems are also very complex to be solved exactly. Heuristic algorithms are generally used.

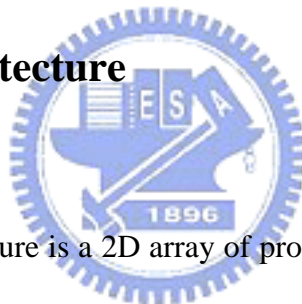
Few works are focus on the task clustering problem of NoC in recent years. A previous work of us tries to solve the task binding problem by simulated annealing [14]. In this work, we only consider the computing power and communication constraint. Another work mapped the task graph into a NoC architecture, but the clustering problem is not considered in it [4]. A work about task mapping for high-performance computing system focuses on communication but other constraints are not mentioned in [10].

In Chapter 2, we will first introduce our NoC architecture and the developing environment. The task clustering problem is then described in Chapter 3. We also introduce our heuristic algorithm to solve this multi-constraint problem in this chapter. Then, in Chapter 4, we will describe our experimental results. Finally, some conclusions are given in Chapter 5.



Chapter 2 Preliminary

2.1 Hardware architecture



Our hardware architecture is a 2D array of processing elements (PE), and each PE is accompanied by a switch as shown in Figure 1. This architecture is commonly used because it meets the physical characteristic of IC technology. For each PE, there is a processor in it and it can run program independently to any other processors. Every switch connects 4 neighbor switches, and the network is constructed by the connection of these switches.

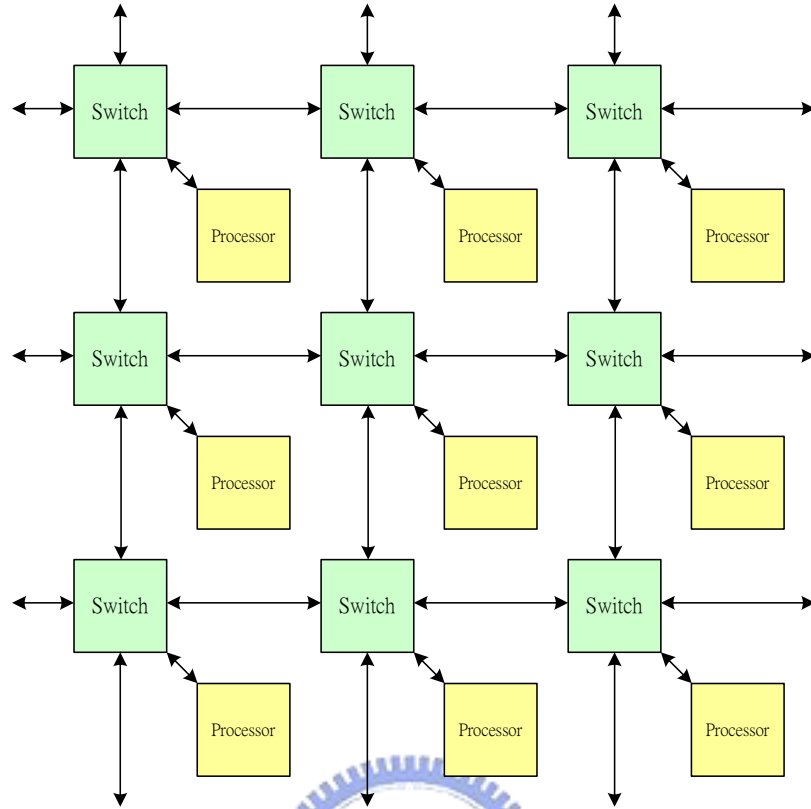


Figure 1. The NoC architecture

2.2 PE architecture

The PE of our architecture is composed of three components and is shown in Figure 2. The first one is a processor. The processor is a general purpose processor and can run a compiled machine code. All of the tasks assigned to the PE will be run one by one by the processor. The next component is the memory. It includes the program memory, data memory and buffer. The memory is addressable to the processor, thus the purpose of the memory is defined by the program code. The third component is an I/O function block. This block is used to transfer data from PE to switches. This block can be controlled by the processor. Like DMA, the

processor only need to setup some registers, then the I/O block can transfer data one by one from memory to switch or from switch to memory.

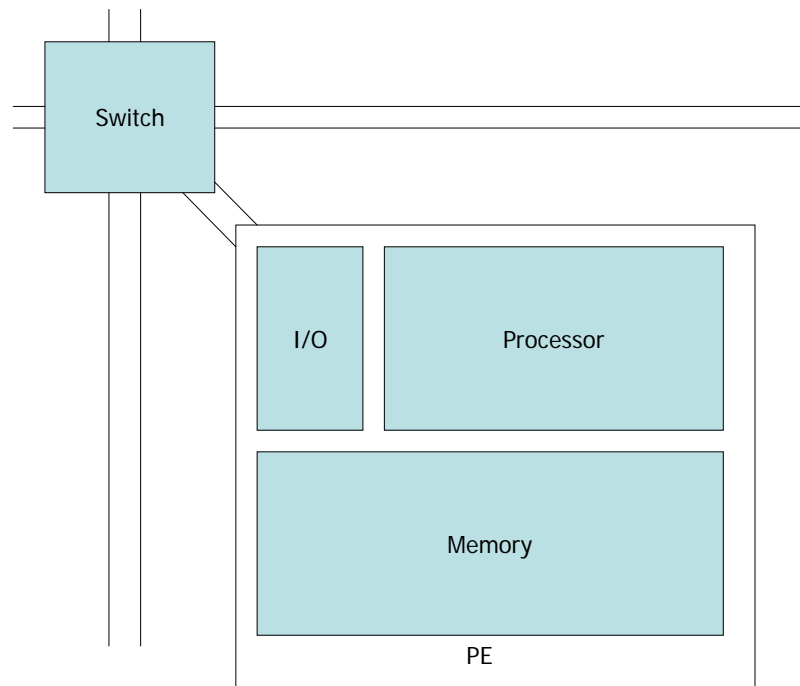


Figure 2. The PE architecture

2.3 Switch architecture

The switch of our architecture belongs to the class of circuit switching. There are 5 ports in our switches; four of them are connected to the neighbor switches and the other one is connected to the processing element attached to it. For every port, the output is selected from the input of the other ports as shown in Figure 3.

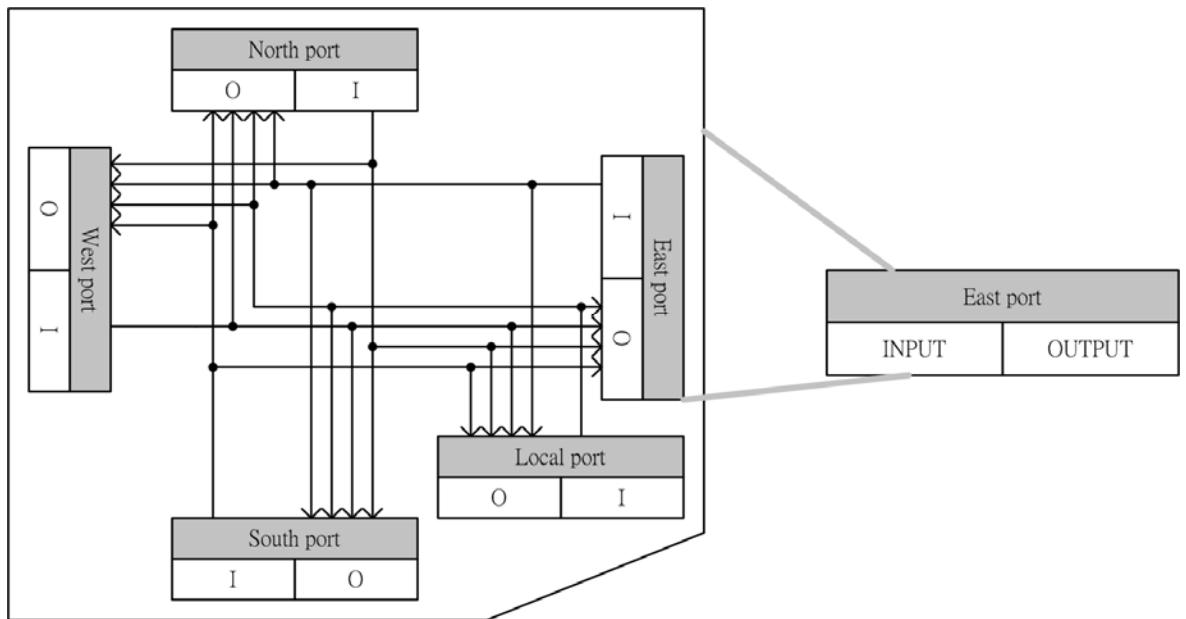


Figure 3. The switch architecture

Our switch doesn't need a lot of memory and it can support real-time applications. We also employ the concept of virtual channel, within which a physical channel can be divided into many virtual channels. Figure 4 is an example showing the way virtual channel works. There are three virtual channels named as A, B and C. Three buffers are used to store the data of these virtual channels. The selection of these channels is decided by a state machine. By proper arranging of the order, the virtual channel can share the bandwidth and solve the blocking problem of the network.

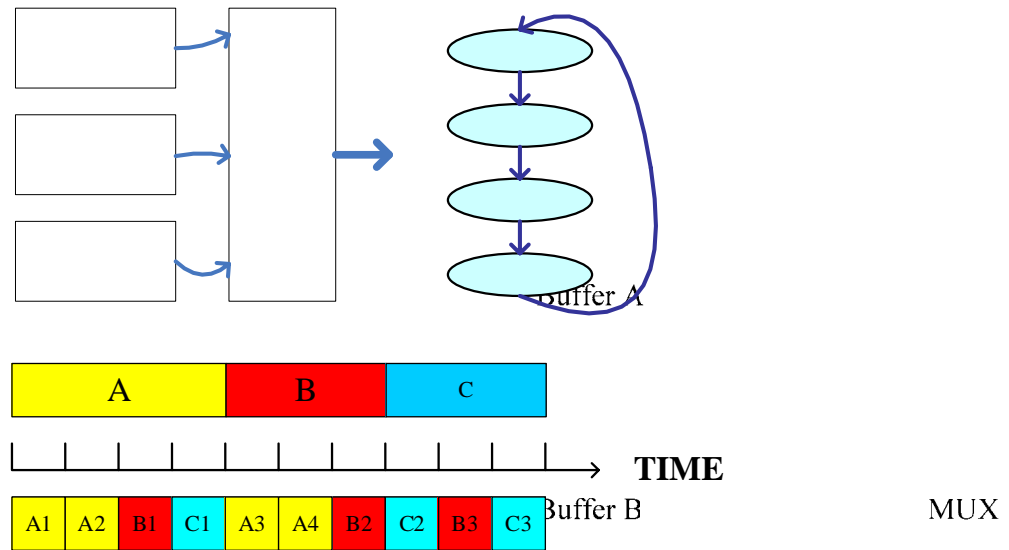


Figure 4. The example of virtual channels

Buffer C

2.4 System development environment

The system development environment of NoC system is quite important because it has great effect on the total system performance. This is a very complex problem with many details to be considered. We developed a flow to solve this problem, and try to find a feasible solution of it.

Figure 5 is the NoC system design flow. The applications are the target applications we want to implement in the NoC system, and it can be developed by using system modeling languages. The architecture platform modeling is the model of our hardware architecture. For example, the processor can be modeled by ISS and the switches can be modeled by system model languages. These models are analyzed and then the flow goes to the mapping process stage. The mapping process will partition the applications into tasks, and then map them to the physical

processing elements. The next step is to analyze the performance and then the result is given. Then, we can make sure the applications can work well under the NoC architecture.

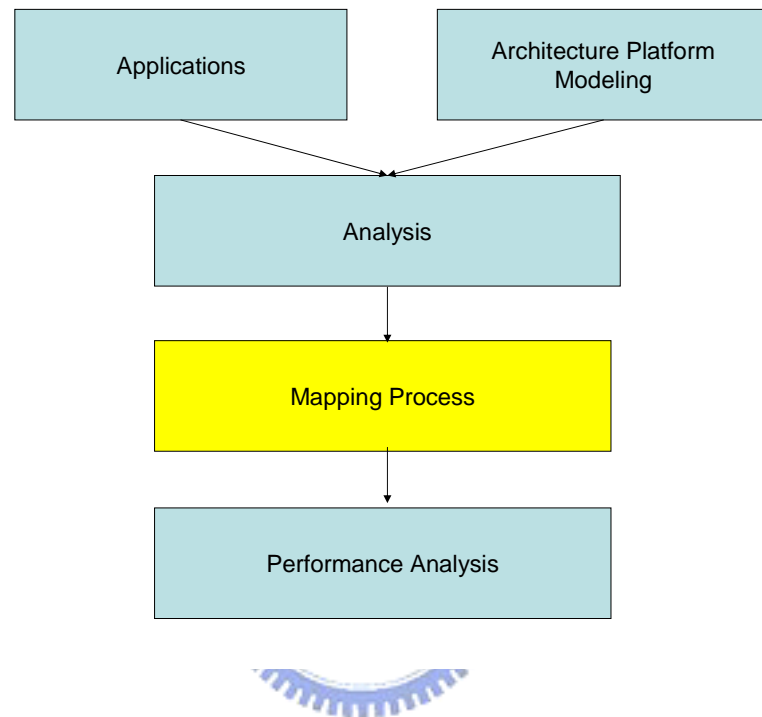


Figure 5. A NoC system design flow

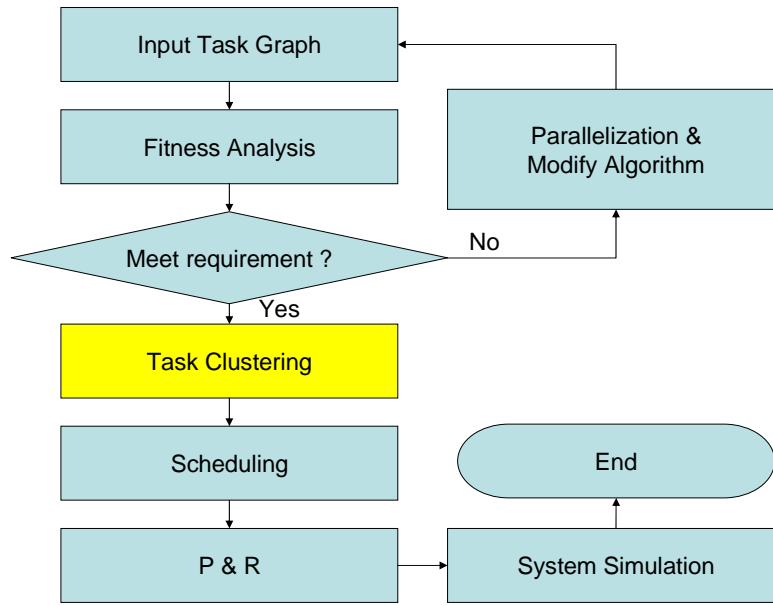


Figure 6 The flow chart of the mapping process

Figure 6 is the flow chart of the mapping process in our design flow. The first step is to transfer the algorithm into the task graph. We can use the decomposition process in parallel processing to do this job. The next step is to analyze the fitness of the task graph. In this stage we will check the iteration bound, and conduct memory optimization. Then we will check whether our design meets the iteration bound and memory requirement. If the check fails, the flow will go back to algorithm stage to do some modification. If the check passes, the next step is the task mapping step. In this step we will cluster the tasks into groups to reduce the number of nodes in the task graph. The next step is the scheduling, in which all tasks in each processing element are scheduled. Then the physical place and route step will decide the physical PE of every task groups. Then we can do a system simulation to check the feasibility of the result. If the check is ok, the flow stops.

In the algorithm development stage before the mapping process, the algorithm should be developed to be more parallelizable so that we can parallelize the system easier in the following stages.

The algorithm will be transferred into a task graph. The task graph is not scheduled in this stage, but the throughput must be decided. According to the throughput, we can get the information of computing power requirement and buffer requirement. The resource requirement of every task and every communication between tasks must be given in the generated task graph.

In the analysis stage, we will transfer the algorithm into timed Petri net model, thus we can analyze the algorithm and find out the iteration loops of it. An iteration loop can not be separated into two different tasks because of the data dependency. If a large iteration loop exists in an algorithm and the resource requirement of the iteration loop is more than a PE's capacity, then this algorithm is not feasible in our system. Thus finding the iteration loops and checking whether they meet the iteration bound are necessary. After this stage, we can make sure that every task can meet a PE's computing power capacity.

We also run the memory optimization in the analysis stage. The on chip memory is expensive, as in our hardware architecture. Thus it is important to handle the memory carefully for every task. The memory optimization is used to manage the buffer memory between tasks. And in this stage, we need to make sure that the memory requirement for each task is less than a PE's capacity.

After the analysis stage, we will check whether every task meets the resource constraints. It means that at least one solution exists such that every task being put into a single PE is feasible. If this check fails, the flow will go to the algorithm development stage. The designers should modify the algorithm to meet this requirement.

Next is the task clustering stage. In this stage, we have a task graph as our input. We want to cluster the tasks into some groups. In every group, the constraints of a PE should be met, so that every group can be put into a single PE. This thesis will focus on this problem in the following content. After this stage, we have a grouping of the tasks, and every group can be placed into a PE.

After the grouping stage, we can run the scheduling procedure. In this stage the execution order of the tasks in the same grouping will be decided. If the scheduling is not arranged properly, extra buffer memory or computing power is required to guarantee the throughput of the applications can be met.

The next stage is PnR stage. In this stage, we place the task groups into physical PEs, and decide the communication routing between PEs. Then, we can calculate the real utilization of communication because the physical mapping is decided. If we can not find a feasible solution in this stage, we will go back to task binding and try to strengthen the communication constraint.

Finally, we will run a simulation to verify the running condition of the whole

system. This is for checking the scheduling of tasks and communication conditions.

After this test passes, our system design flow is finished



Chapter 3 A Task Clustering

Algorithm



The input of the task clustering problem is a task graph which is a directed acyclic graph. Every node on this graph represents a task, and every edge on the graph represents the communication between two tasks. Our target is to group the nodes on this task graph into clusters, which will be mapped to PEs in the following place and route stage. The edges crossing clusters will become the communication data between PEs. There are some constraints must be followed in the clustering problem. We will describe these constraints in the following contents.

The first constraint is the computing power. The computing power is necessary for every task's program execution. But the computing power of the processor in a PE is limited. Thus, if we want to cluster some tasks together, we need to make sure that the summation of the computing power needed by all of the tasks in a cluster

must be smaller or equal to a PE's maximum computing power. We normalize the computing power requirement of every task by dividing it by a PE's computing power capacity. For example, if a task need 50 Mega instructions per second (MIPS) and a PE can supply 100 MIPS, the task's computing power requirement will be represented as 0.5.

The second constraint is the memory capacity. Similar to the computing power, memory is required by every task and every PE has limited memory. According to our architecture, the tasks in a PE can only utilize the build-in memory of this PE. The build-in on chip memory is very expensive, thus it is not economic to have a very large build-in memory for a PE. The memory requirement of a task will also be normalized by a PE's capacity.

The third constraint is the I/O bandwidth. When we decide to map some tasks into a PE, the communications that connect these tasks to all other tasks will need to go through the I/O port, switches, and then to the other PEs. But a PE has limited I/O bandwidth. Thus, the I/O bandwidth constraint must be taken care. The I/O requirement of an edge in the task graph will be normalized by the bandwidth of a PE to the switch attached to it.

The final constraint is the communication constraint. Although the I/O constraint has been taken care, we still need to worry about the whole network communication condition. However we can not measure it accurately because the mapping to physical PE is not yet determined in this stage. We try to use a model to estimate it. The model is similar to the wire load model in logic synthesis. We try to

estimate the edge length by PE's fan in/out. Then multiply the estimated length to the I/O data throughput. The result is a PE's communication contribution to the total system. The summation of all PE's contribution is the whole system's communication.

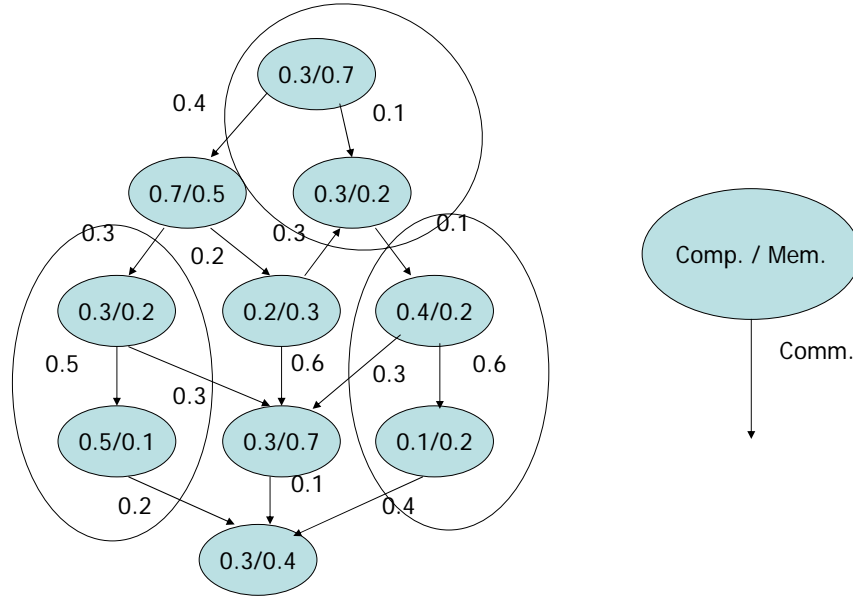


Figure 7. An task clustering example

All of the requirements of these constrains are given in the task graph. We must map these tasks into PEs and satisfy these constraints. Figure 7 shows an example of the input task graph. For every node, the value of computing power requirement and memory requirement are given. For every edge, the communication requirement is also given. When we cluster the nodes together, we need to check that the sum of computing power requirement and memory requirement of these nodes must be less than or equal to one. And the sum of all

edges crossing this cluster also must be less than or equal to one. The communication constraint will be checked after the clustering is finished.

3.1 The algorithmic flow

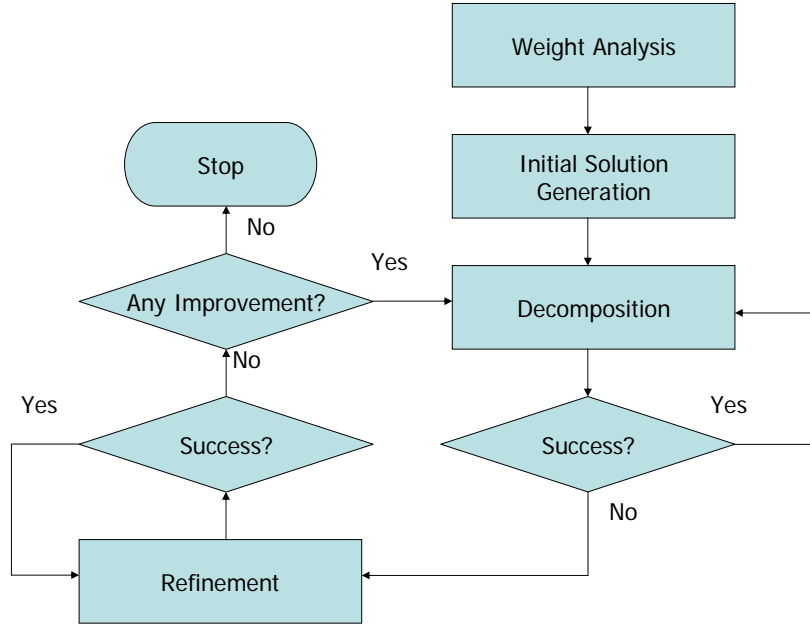


Figure 8. The flow of our heuristic algorithm

Because the complexity of finding the optimum solution is too high, we try to solve this problem by a heuristic algorithm. Our heuristic algorithm is shown in Figure 8. We can separate the flow into two parts. The first part is used to generate an initial solution. In the second part, we try to reduce the number of clusters based on the initial solution from the first part.

The first part includes “Weight Analysis” and “Initial Solution Generation”

stages, and the second part includes “Decomposition” and “Refinement” stages. We employ the two parts to handle two kinds of constraints.

In the first part we apply a fast, greedy heuristic algorithm to pack the tasks. Because we limit that all of the tasks in a PE must be connected, it is possible that one task cannot find any other un-mapped task that is connected to it. Then, the task will be grouped along. For another case, the resource requirements of each task are not the same. The resource distribution in the PEs of the initial solution might be imbalanced. This kind of imbalance also limits the reduction of PE number.

In the second part, we use two kinds of strategy to solve these problems. One is called “Decomposition” stage, and the other called “Refinement” stage. In the Decomposition stage, we find a group, and move all the tasks in it to the neighbor groups. If we succeed, one group will be eliminated. We repeat the Decomposition stage until it fails. Then we go to the Refinement stage. In this stage, we try to identify the most imbalanced group and move one of the tasks out from it. We try to make the final solution more balanced. After the solution is fully balanced and can’t be improved any further, the flow returns to the “Decomposition” again. The program will end when both stages fail.

In the following section, we will describe the details of these stages.

3.2 The Weight Analysis Stage

In this stage, we want to pack the task graph into groups fast and use the result to analyze the task graph. The weights of grade function in Initial Solution Generation stage is decided according to this result. Figure 9 shows the pseudo code of the algorithm of this stage. We do a topological sort first to decide the packing order. According to this order, we pick the first task and name it as the “seed”. The “seed” means the task that will be packed in the following steps. After selecting the seed, we try to find a cluster starting from the seed and having the highest grade. We repeat the packing until all of the tasks are packed.

Weight-Analysis(G)

- 1 Calculate-Constraints(G)
- 2 $List-of-Tasks \leftarrow$ Topological-Sort(G)
- 3 **while** $List-of-Tasks$ not empty
- 4 $Seed \leftarrow$ Top of $List-of-Tasks$
- 5 $Set-of-Clusters \leftarrow$ Find all possible clusters including $Seed$
- 6 $Max-Cluster \leftarrow$ Find the highest grade cluster from $Set-of-Clusters$
- 7 Remove-Task($List-of-Tasks, Max-Cluster$)

Figure 9. The pseudo code of weight analysis stage

In the initial stage of Weight Analysis, we will calculate the total computing power requirements and memory requirements by summing up the requirement value of every task. The results will become the weights of our grade function in later steps.

Then, we run a topological sort to keep a list of all tasks by the result of search. The order of the tasks will start from the root of the task graph and propagate to other tasks according to the distance from the root. We use breadth first search (BFS) because we want to pack the tasks from the root and passing to branches gradually.

In the following while-loop, we will select some tasks to group together. We will pick the smallest order task as the seed from the list, and try every possible cluster that includes it. We will find out all possible clusters by a branch and bound algorithm. Then we will calculate the grades of the possible clusters and select the highest grade one. After the highest grade cluster is decided, all of the nodes in this cluster will be removed from List-of-Tasks and the while loop will continues.

The branch and bound function is designed for searching all possible clusters that include the seed. This function will search nodes to be added from connected edges, and check the constraint of computing power, memory and I/O. We will pick a cluster from all by a grade function. Most task graphs will not have many edges connected to a node, thus the complexity of finding all possible solutions that include the seed is not too high.

The way we pick the best solution is by a grade function. We can handle the multi-constraint problem by using a grade function. We will enlarge the weights of the critical constraints and reduce the weights of the non-critical constraints. We analyze the input task graph to know whether every constraint is critical or not. In the Weight Analysis stage, we can only measure the computing power constraint

and memory constraint because these two constraints can be summed up to compare with a PE's capacity, and the I/O constraint and communication constraint cannot be analyzed in this way. This is because the total capacity of the I/O and communication will change if some tasks are merged and the edges between them will not count to the capacity of I/O and communication.

Because we can not analyze the I/O and communication at the Weight Analysis stage, we leave them to the Initial Solution Generation stage and only consider the measurable constraints here.

The cost function we used in the Weight Analysis is:

$$\sum C * c_i + M * m_i$$

C: total sum of computing power of all tasks in the task graph

c_i : the computing power of i_{th} task

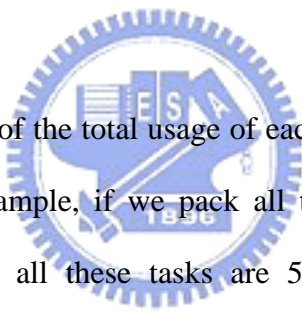
M: total sum of memory of all tasks in the task graph

m_i : the memory of i_{th} task

We use the total computing power and total memory as the weights of the two constraints. It means if one resource is used heavier than the other, it's more important to take care of it and we give it a heavier weight.

3.3 The Initial Solution Generation Stage

After the Weight Analysis, we have an initial solution that considers the computing power and memory constraints. The steps of the Initial Solution Generation stage are almost the same as the Weight Analysis stage. The only difference is the grade function. In the Weight Analysis, only computing power constraint and memory constraint are considered. It doesn't mean that the I/O constraint is a less important than others. In this stage, we can use the result of Weight Analysis to be the input of the analysis. Then, the I/O constraint is considered by using a new grade function.



We use the percentage of the total usage of each constraint to be the weight of the grade function. For example, if we pack all tasks into 10 groups, and total computing power used by all these tasks are 5 PE's capacity, then the total computing power usage is 50%. By this way, the constraints can be calculated and the weight of our grade function can be obtained.

The I/O constraint is not like the computing power and memory. It can be reduced by grouping the heavy edges between tasks. We use the I/O reduction rate in the grade function instead of I/O utilization. When the source and destination tasks of an edge are clustered together, the communication of this edge is reduced. The I/O reduction rate is defined as the sum of reduced communications divided by the sum of reduced and non-reduced communications in a cluster. By setting the I/O reduction rate in the grade function, we can force the algorithm to reduce more

communications.

The new grade function of Initial Solution Generation is:

$$\sum C * c_i + M * m_i + IO * io_i$$

C: total usage of computing power of all PEs

c_i : the computing power of i_{th} task

M: total usage of memory of all PEs

m_i : the memory of i_{th} task

IO: total reduction of I/O of all PEs

io_i : the I/O of i_{th} task



According to this new grade function, the I/O constraint is also considered, thus the result will be better than the result of Weight Analysis, and becomes the initial solution of the following stage. Although the communication constraint is not listed in the grade function, the reduction of I/O bandwidth will also reduce the total bandwidth of the network communications.

3.4 The Decomposition Stage

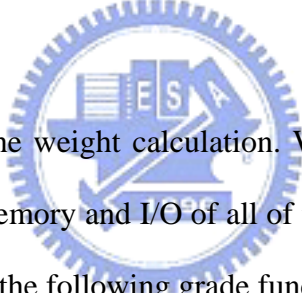
After the Initial Solution Generation stage, there might be some groups that were not fully filled. The decomposition stage is used to find out some clusters to be decomposed, and move all of the tasks of it to other groups. If this stage succeeds, one PE will be reduced and the solution will be better than before. We will repeat this stage until it can't succeed for any cluster. Then, the flows go to the refinement stage.

```
Decomposition(G)
1  Calculate-Constraints(G)
2  List-of-Packing <- Sort-by-Grade(G) // smallest first
3  for Source-Cluster = Top of List-of-Packing
4      For Each task in Source-Cluster
5          Set-of-Targets <- Find all possible target clusters to push
6          Target-cluster <- Select-Smallest-Violation(G, Set-of-Targets)
7          Add to Violation-List if Target-cluster has violation
8      For Cluster-to-Solve = Top of Violation-List
9          while Violation(Cluster-to-Solve) > 0
10             Move tasks to other clusters until no violation
11             Set-Don't-Touch(Cluster-to-Solve)
12      if Communication(G) < Communication-Constraint
13          return Success
14      Recover-Mapping(G)
15 return Fail
```

Figure 10. The pseudo code of the decomposition stage

Figure 10 shows the pseudo code of our algorithm in the decomposition stage.

Firstly, we pick a cluster with the smallest grade to be the target of decomposition. We will release all of the tasks in this cluster to neighbor clusters. However, if we move a task to other clusters, it will be very difficult not to cause any violation. Thus we allow temporary violations to these clusters, and keep a list to record the violation ones. After moving out all tasks from the target cluster, we will start the violation solving step. This step will pick a most violated cluster to be the target. We will move enough tasks from the target to other groups to make it violation free. The flow will continue to solve violations until it fails or all violations are solved. If it fails, we will go back to cluster selecting step. If all violations are solved, this stage succeeds and this stage will be repeated. If all clusters are tried and all fail, this stage stops and goes to refinement stage.



The first step is also the weight calculation. We will calculate the utilization rate of computing power, memory and I/O of all of the clusters in the input solution. The weights will be used in the following grade function.

The next step is to find a cluster to be decomposed. In this step, we try to find a cluster with the lowest grade. We use a similar grade function as in the Initial Solution Generation stage with newly calculated weights. Using the new grade function, we can calculate the grade of all clusters, and pick the lowest grade one.

The for-loop in Line 5 is the main loop of this stage. In this loop we will try to decompose a selected cluster, solve violations and check communication constraint. If all violations are solved and the communication constraint check passes, this stage succeeds. If the violation solving fails or the communication constraint

checking fails, the for-loop will continue to try next cluster to decompose.

In the for loop from line 4 to line 7, we will move all tasks in the selected cluster to the neighbor clusters. Because we restrict that all tasks in a group must be connected by edges, we also need to follow this rule when we move tasks. Thus, only those tasks that have any edge connected to other groups can be moved in the first try. We also use the branch and bound function to select the task to be moved. When moving, we will try all possible target groups, and select the one with the least violation to move. We also record the violation clusters when we push the tasks. The moving continues until all tasks are moved out and the selected cluster is empty.



After all tasks are moved out, there might be some clusters that have violations. We can not accept these violations at the final solution, thus we must try to solve them. We will select the one with the most violation from the violation list to solve. During the process, any new generated violation cluster will also be put into the list to keep this list up to date.

Solving violations is similar to moving tasks out from a group in the second step. The difference is that we don't need to move all tasks out in this step. We only need to move enough tasks to let the group become violation free and induce violations to other groups as less as possible. After a group is solved, it is marked as don't touch and any other group can not move any group back again. This restriction is used to prevent a task being kicked to and from two groups and cause endless iteration.

After violation solving stage, we will check the communication constraint. We check the communication here because only now a new solution without other violations is generated. We check it here to prevent the communication condition from becoming worse.

3.5 The Refinement Stage

After we found that all clusters can not be decomposed, the flow will go to the Refinement stage. In this stage, we try to find some imbalanced group, and adjust it to be more balanced. We will define a balance function to measure the balance of whole system, and use this function to measure the balance after we make some adjustment. If the balance is improved, we will accept this result. Otherwise, we will not accept it and recover the mapping to unadjusted one.

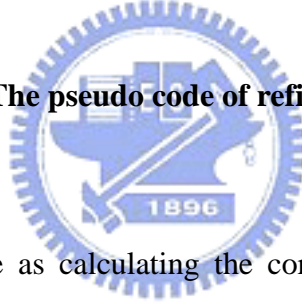
Figure 11 shows the pseudo code of the algorithm of the Refinement stage. The flow of the refinement stage is quite similar to the Decomposition stage. At the first step, we select the most imbalanced cluster. Then we try to move a task from it to the neighbor clusters. The selection of the task to be moved is based on whether it is movable and will improve the balance of the group. This moving of the task also may also generate some violations in the target group. The solving of violations is similar to the decomposition stage. If the solving of the violations succeeds, we will check the system's total balance. If the total balance is improved, this result of the refinement will be kept. Else the result will not be kept and the mapping will be recovered to the input of this stage.

```

Refinement(G)
1  Calculate-Constraints(G)
2  Input-Balance <- Balance(G)
3  List-of-Cluster <- Sort-by-Balance(G) // smallest first
4  for Source = End of List-of-Cluster
5      Find a task to be moved by balance function
6      Set-of-Targets <- Find all possible target cluster to push
7      Target-cluster <- Select-Smallest-Violation(G, Set-of-Targets)
8      Solve-violations(G)
9      if Communication(G) > Communication-Constraint
10         Recover-Mapping(G)
11         continue
12     if Balance(G) < Input-Balance
13         return Success

```

Figure 11. The pseudo code of refinement stage



We define the balance as calculating the correlation to the whole system's constraint utilization. The balance function is:

$$\frac{C * (\sum c_i) + M * (\sum m_i) + IO * (io_i)}{\sqrt{(\sum c_i)^2 + (\sum m_i)^2 + (\sum io_i)^2}}$$

C: total usage of computing power of all groups

c_i : the computing power of i_{th} task

M: total usage of memory of all groups

m_i : the memory of i_{th} task

IO: total usage of I/O of all groups

io_i : the I/O of i_{th} task

The balance function is defined as the inner product of the normalized vector of resources in a cluster and vector of global weights. If the resource distribution of a cluster is similar to the global weight, it will gain a higher score. Using this function, we can measure the balance of every cluster. If we sum up the score of all groups in a mapping, the result is the balance of this mapping.



Chapter 4 Experimental Results

4.1 The experimental environment



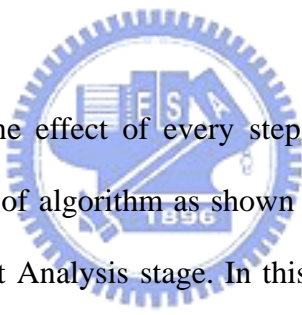
We have done some experiments to verify the efficiency of our algorithm in this problem. Our input task graph is generated by the Task Graph For Free (TGFF) [7], which is a configurable task graph generator. By setting the average and variance of each parameter in the input control file, we can generate a set of random task graphs. The generated task graph can be translated into a task graph format of our environment, and become the input of our flow.

The input parameter of TGFF is set as in the following descriptions. The average values of computing power and I/O are set as 0.1, 0.2, 0.25 and 0.3. The average value of memory is set as 0.1, 0.2, 0.25, 0.3 and 0.4. The average task number is set as 20 and 30. The average fan in/out number is set as 2, 3 and 4. The variances of all parameters are all set as 0.1. For every combination of these

parameters, a task graph is generated. All these task graphs are filtered to remove those with negative values.

Our algorithm has been developed to be an automatic tool. We only need to prepare the wire load model of the communication constraint. Then the tool will read the task graph information and run step by step. The output of our tool is a list of clusters. Each of the clusters has a list of tasks that were packed in it.

4.2 The experimental results



In order to measure the effect of every step in our algorithm, we set four measure points in our flow of algorithm as shown in Figure 12. The first measure point is set after the Weight Analysis stage. In this point we can get a first initial solution, and we will take the result of this measure point as the reference of other measure points. The second measure point is set after the initial solution generation stage. We want to examine whether the result of using different weight function can improve the design. The third measure point is set after the decomposition stage is finished, and the fourth measure point is set to check the final result. By comparing the result in third and fourth measure points, we can check whether the refinement stage can further improve our results.

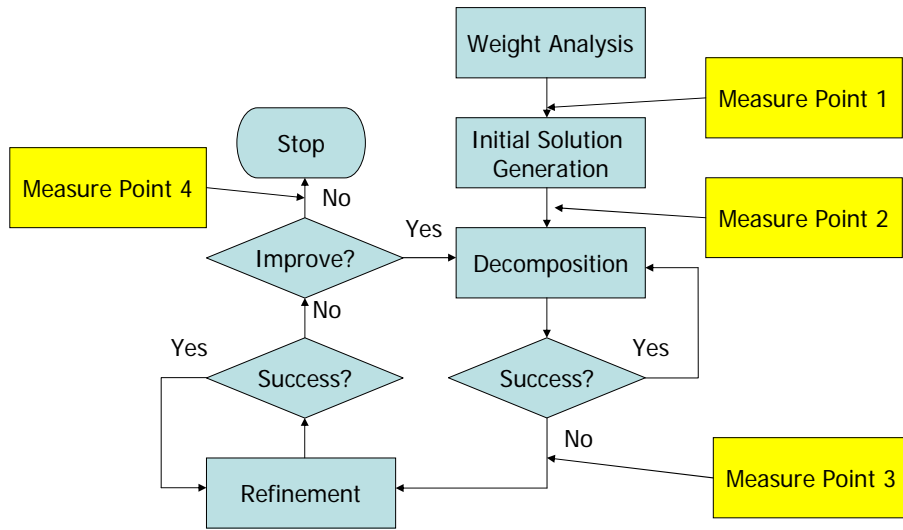


Figure 12. The measure points of our experiment

The statistics of experimental results is shown in Figure 13. The X-axis has three sets of data, the improve1, improve2 and improve3. These data are the statistical results of the comparisons of the number of clusters in those measure points. The measure point 1 is the reference, and all other measure points are normalized by the result in measure point 1. Each color bar means a different range of reduction rate. We separate the range by 0.1. The Y-axis is the sample number of each color bar. For example, the most left color bar means the sample number of reduction range from 1.0 to 0.9 of measure point 2 is 33.

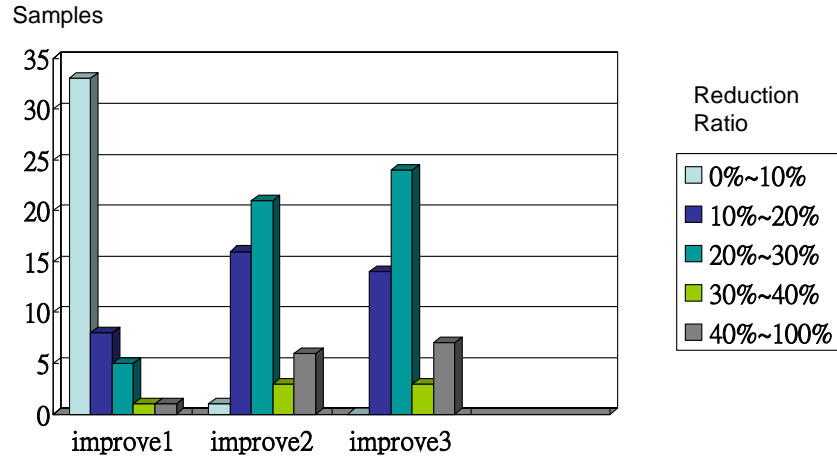


Figure 13. The experimental result

From the result we can see the reduction rate of every measure point compared with the measure point 1. From the result of improve1, we can see that most samples are not significantly improved. Only 15 out of 48 samples are improved more than 10%. In improve2, most samples are improved more than 10%. The improve2 has better improvement compared with the improve1 because the Decomposition stage can greatly reduce the cluster number. The result of improve3 is similar to the improve2, thus only some cases are improved in this stage. It means that the Refinement stage is only effective for some cases.

We also try to change the variance of the parameters. The results are shown in Table 1. When the variance becomes larger, the comparing results from measure point 3 to measure point 4 are not improved. As a result, the variance of input

parameters will not effect on the performance in the Refinement stage.

Table 1. The effect of variance

<i>Variance</i>	<i>Improve 1-2</i>	<i>Improve 1-3</i>	<i>Improve 1-4</i>
<i>0.1</i>	0.988	0.856	0.849
<i>0.2</i>	0.982	0.815	0.808
<i>0.3</i>	0.989	0.857	0.852

The detail data of all cases in our experiment are listed in Table 2. Some cases are not listed because of violations. The number of cluster in each measure point and the parameter of each constraint are listed on it.



Table 2. The list of results

Case	Measure Point 1	Measure Point 2	Measure Point 3	Measure Point 4	Communication	Computing Power	Memory	Number of Tasks	Fan In/Out
1	12	9	5	5	0.1	0.1	0.1	20	2
2	15	8	8	8	0.1	0.1	0.1	20	3
3	12	11	9	9	0.1	0.1	0.2	20	2
4	17	12	10	9	0.1	0.1	0.2	20	3
5	18	18	15	15	0.1	0.1	0.3	20	2
6	21	15	13	13	0.1	0.1	0.3	20	3
7	22	20	18	18	0.1	0.1	0.4	20	2
8	23	23	18	18	0.1	0.1	0.4	20	3
9	12	10	7	6	0.1	0.2	0.1	20	2
10	14	11	10	10	0.1	0.2	0.2	20	2
11	20	17	16	15	0.1	0.2	0.3	20	2
12	23	20	18	18	0.1	0.2	0.4	20	2
13	15	14	9	9	0.2	0.1	0.1	20	2
14	18	12	10	10	0.2	0.1	0.1	20	3
15	14	14	10	10	0.2	0.1	0.2	20	2
16	17	13	10	9	0.2	0.1	0.2	20	3
17	18	17	14	14	0.2	0.1	0.3	20	2
18	18	15	13	13	0.2	0.1	0.3	20	3
19	22	20	18	18	0.2	0.1	0.4	20	2
20	23	23	18	18	0.2	0.1	0.4	20	3
21	16	14	10	10	0.2	0.2	0.1	20	2
22	16	14	10	10	0.2	0.2	0.2	20	2
23	20	17	15	15	0.2	0.2	0.3	20	2
24	23	21	19	18	0.2	0.2	0.4	20	2
25	17	17	15	15	0.3	0.1	0.1	20	2
26	19	18	15	15	0.3	0.1	0.1	20	3
27	17	17	15	15	0.3	0.1	0.2	20	2
28	19	18	16	15	0.3	0.1	0.2	20	3
29	19	18	16	16	0.3	0.1	0.3	20	2
30	19	18	15	15	0.3	0.1	0.3	20	3
31	22	22	18	18	0.3	0.1	0.4	20	2
32	23	23	19	18	0.3	0.1	0.4	20	3
33	17	16	15	15	0.3	0.2	0.1	20	2
34	17	16	16	15	0.3	0.2	0.2	20	2
35	21	18	17	17	0.3	0.2	0.3	20	2
36	23	22	18	18	0.3	0.2	0.4	20	2
37	22	22	18	18	0.4	0.1	0.1	20	2
38	23	23	18	18	0.4	0.1	0.1	20	3
39	22	22	18	18	0.4	0.1	0.2	20	2
40	23	23	18	18	0.4	0.1	0.2	20	3
41	22	22	18	18	0.4	0.1	0.3	20	2
42	23	23	18	18	0.4	0.1	0.3	20	3
43	22	22	19	18	0.4	0.1	0.4	20	2
44	23	23	18	18	0.4	0.1	0.4	20	3
45	23	22	18	18	0.4	0.2	0.1	20	2
46	23	22	18	18	0.4	0.2	0.2	20	2
47	23	22	18	18	0.4	0.2	0.3	20	2
48	23	22	18	18	0.4	0.2	0.4	20	2

Chapter 5 Conclusions

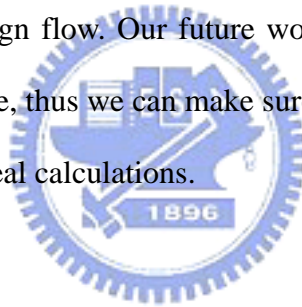
The Network-on-Chip designs will become a new trend of on-chip communication design. By developing design automation tools, the effort on system development can be reduced and design speed and quality can be guaranteed. In the flow of system development, the task clustering problem will effect on the performance and the resource requirement of the applications. Thus it is an important problem to be solved.

In this thesis we have defined the task clustering problem in Network-on-Chip system design flow, and have proposed a heuristic algorithm to solve this problem. Our problem focuses on meeting multiple constraints with the most reduction on cluster number. We use two stages to analyze the input task graph and generate an initial solution. The iteration of decomposition and refinement stages can further reduce the number of clusters. We check the computing power, memory and I/O constraints when we try to group the tasks. The communication constraint is checked

every time a new clustering solution is generated. Thus for the solution in every stage, all constraints are guaranteed to be met.

Our experimental result in Chapter 4 shows the results of every stage. We compared them with the result of the first stage and significant improvement can be seen in each stage. Most improvement can be done in the Decomposition stage and some extra improvement in the Refinement stage.

In our problem we use wire load model to calculate the utilization of communication resources. The actual result will be obtained in the Placement and Route stage later in our design flow. Our future work is to combine the placement and route stage with this stage, thus we can make sure the communication constraints will always be met through real calculations.



Reference

- [1] Alan Allan, Don Edinfeld, William H. Joyner, Jr., Andrew B. Kahng, Mike Rodgers and Yervant Zorian, “2001 technology roadmap for semiconductors”, IEEE Computer, 2002.
- [2] Luca Benini and Giovanni De Micheli, “Networks on chips: a new SoC paradigm”, IEEE Computer, 2003.
- [3] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Öberg, Kari Tiensyrjä and Ahmed Hemani, “A network on chip architecture and design methodology”, IEEE Computer Society Annual Symposium on VLSI, 2002.
- [4] Tang Lei and Shashi Kumar, “A two-step genetic algorithm for mapping task graphs to a network on chip architecture”, Euromicro Symposium on Digital System Design, 2003.
- [5] Doris Ching, Patrick Schaumont and Ingrid Verbauwhede, “Integrated modeling and generation of a reconfigurable network-on-chip”, 18th International Parallel and Distributed Processing Symposium, 2004.
- [6] Adrijean Adriahtenaina, Hervé Charlery, Alain Greiner, Laurent Mortiez and Cesar Albenes Zeferin, “SPIN: a scalable, packet switched, on-chip micro-network”, Design Automation and Test in Europe Conference and Exhibition, 2003.
- [7] Robert P. Dick, David L. Rhodes and Wayne Wolf, “TGFF: task graphs for free”, 6th

International Workshop on Hardware/Software Codesign, 1998.

[8] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-chip interconnect woes through communication-based design", Design Automation Conference, 2001.

[9] Axel Jantsch and Hannu Tenhunen, "Network on chip", KLUWER Academic Publishers, 2003.

[10] J.M. Orduña, F. Silla and J. Duato, "A new task mapping technique for communication-aware scheduling strategies", International Conference on Parallel Processing Work shop, 2001.

[11] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, "Parallel computer architecture, a hardware/software approach", Morgan Kaufmann Publishers, 1999.

[12] D. Wiklund and Dake Liu, "SoCBUS: switched network on chip for hard real time embedded systems", Parallel and Distributed Processing Symposium, 2003.

[13] Pao-Jui Huang, "A switch design for multi-processor system on chip", National Chiao Tung University, 2004.

[14] Chih-Chieh Chou, "Task binding on multi-processor system-on-chip", National Chiao Tung University, 2004.

VITA

Yung-Chun Lei was born in Keelung, Taiwan on August 02, 1974. He received the B.S. degree in Electronics Engineering from National Chiao Tung University in January 1999 and entered the Institute of Electronics, National Chiao Tung University in February 2001. His research interests include electronic design automation (EDA) and VLSI design.