Grouping-Based Architecture Exploration of System-Level Design

Lukai Cai
Daniel D. Gajski

CECS Technical Report 02-31
8/19/2002

Center for Embedded Computer Systems
University of California
Irvine, CA 92697, USA
{lcai, gajski} @cecs.uci.edu

Abstract

This report introduces the grouping-based architecture exploration of the system level design. The grouping-based architecture exploration selects processing elements (PEs) to assemble the system architecture according to the design's functionality. Furthermore, it maps design's functional blocks to the selected PEs to pursue the shortest execution time. This report outlines the design flow of the grouping-based architecture exploration and describes a list-scheduling based algorithm for the grouping-based architecture exploration.

Index

List of Figure

List of Table

Grouping-Based Architecture Exploration of System-Level Design

Lukai Cai, Daniel D. Gajski
Center for Embedded Computer Systems
University of California
Irvine, CA 92697, USA
{lcai, gajski} @cecs.uci.edu

Abstract

This report introduces the grouping-based architecture exploration of the system level design. The grouping-based architecture exploration selects processing elements (PEs) to assemble the system architecture according to the design's functionality. Furthermore, it maps design's functional blocks to the selected PEs to pursue the shortest execution time. This report outlines the design flow of the grouping-based architecture exploration and describes a list-scheduling based algorithm for the grouping-based architecture exploration.

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[1] 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 specification representing the designs' functionality (also called application or system behavior), which is denoted by point S. The system level design then synthesizes the specification to the system architecture denoted by point A. A system architecture consists of a number of PEs (processing elements) connected by buses. Different PEs can belong to different PE types. Each PE implements a number of functional blocks in the specification.

One of main tasks of the system level design is architecture exploration (also called design space exploration). Architecture exploration contains three tasks: PE selection, behavior-architecture mapping, and behavior scheduling. PE selection selects appropriate PEs from PE library to assemble the system architecture. Behavior-architecture mapping determines which allocated PE that each behavior will be assigned to. Behavior scheduling determines when the behaviors are executed on the mapped PEs.

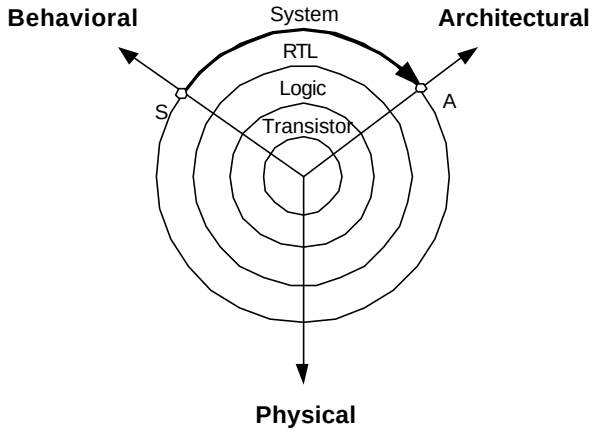


Figure 1: Extended Gajski and Kuhn's Y chart

The above tasks have been studied for a decade in different scopes, which will be reviewed in section 2. Most of previous work pursue optimal solutions therefore the complexity of them are relatively high. However, with the increase of the design complexity, designers need a method with relatively low complexity to produce a satisfactory result rather than an optimal result, to shorten the design time. As a result, we carefully design an algorithm to tailor the designers' needs.

This report is organized as follows. Section 2 reviews the previous work. Section 3 defines the problem scope. Section 4 introduces the design

flow of grouping-based architecture exploration. Section 5 introduces behavior grouping which groups the functional blocks in design. Section 6 introduces PE selection for each group. Section 7 describes behavior mapping which reassigns the functional blocks to the selected PEs to shorten the execution time. Section 8 introduces the PE number selection. The experimental result is given in section 9. Finally section 10 concludes the report.

2 Previous Work

The *behavior-architecture mapping* problem has been well studied for multi-processor architectures containing the same type of processors[2][3][4][5]. Studies in [2][4] show that the CPM(critical path method) list scheduling algorithm produces near-optimal solutions for this type of problems and the complexity of CPM is O(nlog(n) + log(p)), where n denotes the number of tasks and p denotes the number of processors. Therefore, in this report, we choose the CMP list scheduling as our *behavior-architecture mapping* algorithm.

Based on the study of HW/SW co-design, such as [6][7], the research on the architecture exploration for multi-PE architectures containing a number of different types of PEs are popular in these years. These research can be classified to two types: the system synthesis[12][13][14][15] starting from *PE selection*, or the platform-design[8][9][10][11] without considering *PE selection*. Since we believe that the *PE selection* gives designers a high flexibility to produce the application specific SoC, in this report, we take the *PE selection* into account.

Among the research considering *PE selection*, Prakash and Parker[12] formulated the problem as an integer linear program(ILP). They could simultaneously allocate and schedule processes while designing the underlying PEs. However, their ILP formulation is time consuming and sometimes requires hours to execute. D'Ambrosio and Hu[14] use simulation to judge the feasibility of a schedule, then screen those candidates for feasibility by simulation. Simulation is both time-consuming and does not guarante to prove feasibility. Wolf[13] developed a heuristic algorithm that gives results comparable to ILP in many cases. The complexity of its work is much better, which is $O(p^3n^2P)$, where p is the number of PEs in the system, n is the number of behaviors, and P is the number of PE types in the PE library. During architecture exploration, to remove the least unutilized PE from the system, it moves behavior on that PE to other PE in the system architecture. However, it uses utilization as an approximation to evaluate the feasibility of these movements, rather than re-scheduling, which makes the result incorrect in some cases. Yen[15] extended Wolf's work. He evaluated displacement vector when moving one behavior from current mapped PE to any other possible PEs, and then performed the movement that has the highest sensitivity. The complexity of its work is $O(pn^3Plog(n))$, if the complexity of the sensitivity evaluation is O(nlog(n)). In comparison to above research, the complexity of our algorithm is $O(pPnlog(n)) + O(np^2log(n))$, which is the lowest.

3 Scope Definition

The purpose of this report is to outline the grouping-based architecture exploration. In this report, we limit our interests to following aspects.

1. Behavior model
   The behavior model of design consists of a number of behavior entities called *behaviors*, which represents the functional blocks. *Behavior*s are composed hierarchically either in sequential or in parallel sequence. However, behaviors cannot be composed in a pipeline sequence.

2. Architecture model.
   The system architecture contains a number of PEs connected by buses. Different PE can belong to different PE types. During the architecture exploration, we have two objectives. The first is to determine the number of PEs in the system architecture. The second is to select suitable PE type for each PE. Bus selection and memory selection are not taken into account in this level.

3. Performance Estimation.
   During architecture exploration, we only use the execution time and the system cost as our criteria. During the execution time estimation, we only compute the execution time of behaviors, while the communication

time between behaviors is ignored. The system cost is equal to the sum of the cost of PEs in the system architecture.

4 Design Flow

Figure 2(a) presents the design flow of the grouping-based architecture exploration, which contains three steps: *specification tuning*, *PE selection*, and *behavior mapping*.

First, *specification tuning* changes the behavior model of design which makes it suitable for architectural exploration. For example, it explores maximal parallelism existing in the behavior model and reduces the hierarchy depth while keeping all the parallelism. *Specification tuning* is introduced in [16].

Second, if the system architecture is not pre-defined, then *PE selection* selects PE from PE library to assemble the system architecture. The generated system architecture must ensure that the implementation meet the given time constraint when behaviors are mapped onto it and the generated system architecture has the low cost.

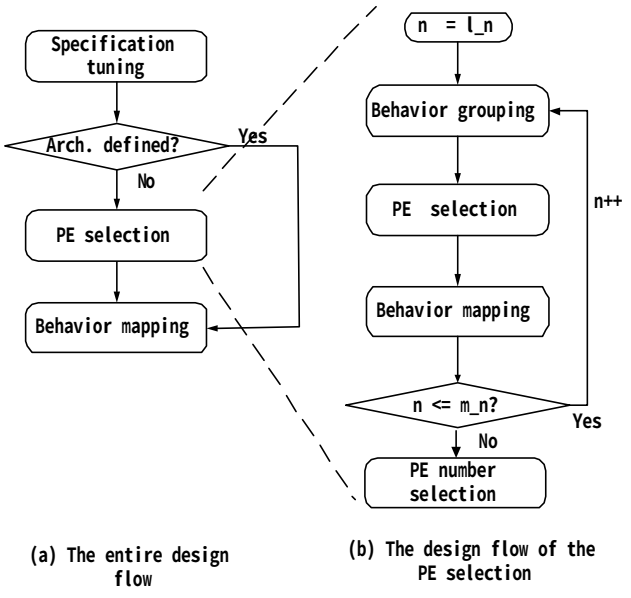


Figure 2: Design flow of the grouping-based architecture exploration

Third, *behavior mapping* maps the behavior model to the generated architecture to pursue the shortest execution time.

PE selection is illustrated in Figure 2(b). In Figure 2(b), *n* represents the number of PEs which will be selected for the system architecture. *l_n* is the lower bound of n while *m_n* is the upper bound of *n*. We will discuss the value of *l_n* and *m_n* in section 8.

During *PE selection*, *behavior grouping* first groups the *behaviors* in the system behavior model to *n* groups. During grouping, it balances the computation in each group.

After *behavior grouping*, *PE selection* selects the slowest PE for each group without violating the given time constraint. Then designers use all the selected PEs to assemble the system architecture.

Finally, *behavior mapping* maps the behavior model to the generated architecture to pursue the shortest execution time.

For each selected n, *behavior grouping*, *PE selection*, and *behavior mapping* are executed once to produce a system architecture containing n PEs . At the end of *PE selection*, *PE number selection* selects the architecture with the lowest cost among all the generated architectures containing different number of PEs.

5 Behavior Grouping

5.1 System Behavior Model

We use SpecC language[18][19] to specify the functionality of design, which is illustrated by Figure 3.

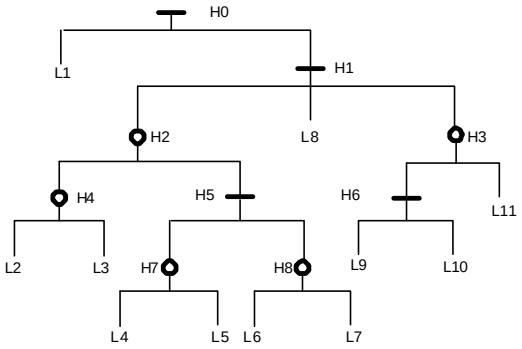


Figure 3: An example of the system behavior model

In the SpecC language, there are two types of *behavior*: leaf behavior and non-leaf behavior. A leaf behavior, which is identified by the name starting with "L", implements certain functionality of design. A non-leaf behavior, which is identified by the name starting with "H", contains a number of sequential (—) or parallel (o) executing child behaviors. If a non-leaf behavior is identified as a sequential behavior (—), then its left child behavior is always executed before its right child behavior. In Figure 3, L4, L5, L6, and L7 are four leaf behaviors. H5 is a non-leaf sequential behavior. H7 is a non-leaf parallel behavior. In H5, the executing sequence is H7 then H8.

We assume *specification tuning* has been applied on the system behavior model thus the system behavior model specifies all the possible parallelism in the design. Therefore, we don't do further dependency-analysis on the behavior model.

5.2 Behavior Estimation

We evaluate the computation cost of leaf behaviors by using spec profiler[20]. The spec profiler produces the execution number of operations during simulation for each behavior, which is used to represent behavior's computation cost.

As mentioned before, we ignore the communication cost among behaviors.

5.3 Behavior Grouping Algorithm

Behavior grouping is the first step of *PE selection*. The goal of *behavior grouping* is to map *behaviors* to *n* groups and to balance the computation cost on each group. If we treat behaviors as tasks, treat the computation cost as the execution time, and treat groups as resources, then the grouping problem can be interpreted as a scheduling problem: schedule tasks to *n* resources to pursue the shortest execution time. This scheduling problem has been well-studied [17][21]. In this report, we choose the static-list scheduling[17][22] as the grouping algorithm.

5.3.1 List Scheduling

List scheduling are class of implementable schedules in which tasks are assigned priorities and placed in a list ordered in decreasing magnitude of priority. Wherever executable tasks contend for processors, the selection of tasks to be immediately processed is done on the basis of priority with the higher priority tasks executable being assigned processors first. If there is more than one task of a given priority, ties are broken randomly. List schedules may be preemptive or non-preemptive, dynamic or static.

List scheduling contains three tasks at each iteration.

1. Task selection
   It selects task according to the priority list.

2. Processor selection
   It selects suitable resource for the selected task.

3. State update
   It updates the priority list and resource statistics.

In this report, we choose a static, non-preemptive list scheduling algorithm.

5.3.2 Basic Grouping Algorithm

```
Algorithm 1 PRIORITY_LIST_GENERATION(){
        COMPUTE_ADJUSTED_OP();
        prioirty_list = SORT_LEAF_BEHAVIOR();
}

Algorithm 2 BASIC_GROUPING (n, priority_list)

groups = RESET_GROUP(n);
RESET_BEHAVIOR();
behavior = FIRST(priority_list);
while behavior do
        early_start = ∞;
        group = FIRST(groups);
        while group do
                if(group.earliest_time < early_start){
                        mapped_group = group;
                        early_start = group.earliest_time;
                }
                group = NEXT(groups);
        endwhile
        behavior.start_time = max( early_start, behavior.earliest_time );
        behavior.end_time = behavior.start_time + behavior.op;
        behavior.earliest_time = behavior.start_time;
        INSERT_TO_GROUPS(behavior, group, start_time, end_time);
        behavior = NEXT(priority_list);
endwhile
```

Algorithm 2 outlines the basic grouping algorithm which is a variation of the static list scheduling. Before implementing basic grouping, *priority_list* is produced by algorithm 1. *COMPUTE_ADJUSTED_OP* first computes the *adjusted_op* of leaf behaviors. The *adjust_op* of a leaf behavior equals to the sum of the leaf behavior's computation cost (Op) and all of its successors' Op. The successors of a leaf node can be found according to the behavior model illustrated in Figure 3: if an ancestor *B* of the leaf node *A* is a sequential behavior identified by ━, then all the leaf behaviors in the children of *B* right to the child of *B* containing *A* are *A*'s successors. For example, in Figure 3, H1 is one of L4's ancestors and H1 is a sequential behavior. Therefore, L8, L9, L10, and L11 are L4's successors. We use a post-order tree walk to compute *adjusted_op* for all the leaf behaviors, in the complexity of O(m), while m is the number of behaviors in the model. After COMPUTE_ADJUSTED_OP, SORT_LEAF_BEHAVIOR sorts leaf behaviors and saves them in *priority_list* in the decreasing order of adjust_op. Ordering in this way, the predecessors of a behavior are always sorted before the behavior.

After *priority_list* is generated, basic grouping algorithm first creates an set of group *groups* containing n groups and set the variable *earliest_time* of each group to 0 by *RESET_GROUP(n). RESET_BEHAVIOR* then sets the variable *earliest_time* of each leaf behavior to 0.

Then, during each iteration, the algorithm schedules one leaf behavior at a time, starting from the behavior in the beginning of priority_list which has the largest adjust_op. At a time, the scheduled behavior will be mapped to the group having the smallest *earliest_time*. Then *start_time* and *end_time* of the behavior are computed: *start_time* equals to the maximum of the *earliest_time* of *behavior* and *earliest_time* of *group*. The *end_time* equals to the sum of *start_time* and behavior's op. The *earliest_time* of *behavior* is equal to the *start_time* of *behavior*.

Finally, INSERT_TO_GROUPS inserts *behavior* to *group*. The earliest_time of *group* is updated to *end_time*. Furthermore, all the *earliest_time* of *behavior's* immediate successors are updated to the maximum of *end_time* and their previous *earliest_time*.

For example, Table 3 shows the computation cost(op) and adjust_op of leaf behaviors in Figure 3. Figure 4 shows the corresponding result of behavior grouping.

Table 1: Op and adjust_op of leaf behaviors in the example of Figure 3.

	L1	**L2**	**L3**	**L4**	**L5**	**L6**
Op	24	30	12	20	28	36
Adjust_Op	240	106	88	146	154	112
	L7	**L8**	**L9**	**L10**	**L11**	
Op	14	10	18	26	22	
Adjust_Op	90	76	44	26	22	

5.4 Behavior Re-Grouping

The result of basic grouping algorithm is excellent in terms of the length of the critical path because of the characteristics of the list scheduling. The critical path refers to the path from starting point *S* to the ending point *E* displayed in Figure 4. The length of the critical path equals to the sum of the computation costs of behaviors on the critical path. The length of the critical path in Figure 4 is 142.

After applying the basic grouping algorithm, *behaviors* on the critical path may be mapped to

different groups. For example, in Figure 5(a), behaviors *A, B,* and *C* on the critical path are mapped to group 1, 2, and 3 respectively.

In order to execute the behaviors on the critical path on the fastest PE in the system architecture to pursue the fastest execution time, we must map the critical-path behaviors to one group. The ideal grouping result is displayed in Figure 5(b): critical path behaviors *A,B, and C* are mapped to group1; The "second critical path" behaviors D and E are mapped to group2; The "third critical path" behavior *F* is mapped to group 3. The "*k*th critical path" refers to the critical path when mapping behaviors to (n – k +1) groups, after 1,2.., (k-1)th critical paths have been found and the behaviors in 1, 2, .., (k-1)th critical path have been removed, where n is the number of group. Grouping in this way, we can map group 1 to the fastest PE, map group 2 to the second fastest PE, and finally map group 3 to the slowest PE in the system architecture.

Therefore, we implement behavior regrouping algorithm to produce the ideal grouping result, based on the result of basic grouping, which is displayed in algorithm 3.

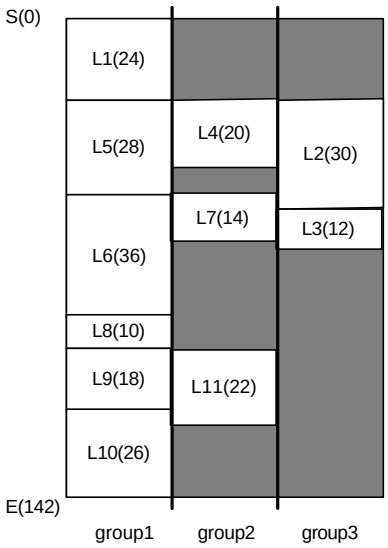
At the beginning, *RESET_EMPTY_GROUPS* creates a variable *groups* representing a set of groups. At the beginning, there is no group in *groups*. Then one group is added to *groups* at each iteration.

Algorithm 3 BEHAVIOR-REGROUPING (n, priority_list)

```
groups = RESET_EMPTY_GROUPS();
while n ≥ 0 do
        length = BASIC_GROUPING(n, priority_list);
        BACK_TRACE_SCHEDULING(n, priority_list, length);
        group = FIND_CRITICAL_PATH(bhvrs);
        ADD_GROUP(groups, group);
        UPDATING(bhvrs, group);
        n = n - 1;
endwhile
```

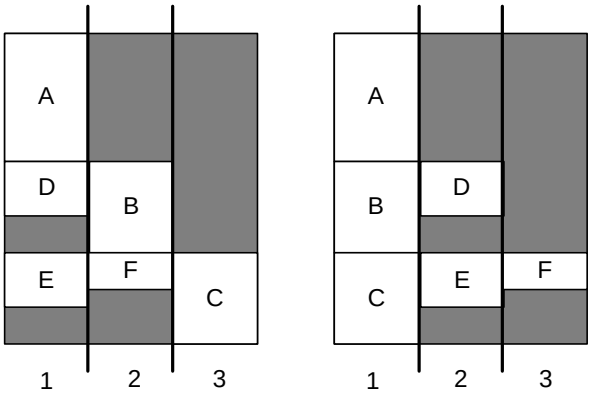
At each iteration, BASIC_GROUPING explained in algorithm 2 first produces an initial grouping solution. The returned value *length* equals to the length of the critical path in the grouping result. BACK_TRACE_SCHEDULING then reads the grouping result and reschedules behaviors in each group without moving behaviors across the groups. In contrast to BASIC_GROUPING, BACK_TRACE_SCHEDULING starts scheduling from the ending point E and ends at the starting point S of the grouping result. It also starts scheduling from the behavior at the end of the priority list, rather than from the behaviors at the beginning of the priority list. The difference between scheduling methods in BASIC_GROUPING and BACK_TRACE_SCHEDULING is similar to the difference between ASAP(as soon as possible) algorithm and ALAP(as late as possible) algorithm explained in [17]. BACK_TRACE_SCHEDULING is explained in algorithm 4. The result of BACK_TRACE_SCHEDULING for the example in Figure 3 is displayed in Figure 6.

L1, L5, L4, L6, L2, L7, L3, L8, L9, L10, L11
(a) Ordering sequence



(b) Grouping result (n = 3)

Figure 4: The basic grouping result of the example of Figure 3.



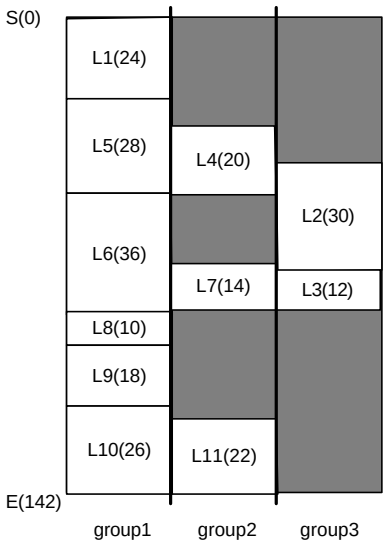
(a) Basic grouping result

(b) Ideal grouping result

Figure 5: The comparison of the basic grouping result and the ideal grouping result.

After BACK_TRACE_SCHEDULING, FIND_CRITICAL_PATH finds the behaviors on the critical path. For a behavior, if its *start_time* assigned in BACK_TRACE_SCHEDULING is equal to its *earliest_time* assigned in BASIC_SCHEDULING, then it is a critical_path_candidate. FIND_CRITICAL_PATH selects a critical_path_candidate starting from the starting point S. Then it selects the next critical_path candidate starting from the *end_time* of the previous selected critical_path_candidate. The critical_path_candidate selection continues until the ending point E is reached. All the selected critical_path_candidates are the critical path behaviors and comprises the returned *group*.

L11, L10, L9, L8, L3, L7, L2, L6, L4, L5, L1
(a) Order sequence



(b) Grouping result (n = 3)

Figure 6: The scheduling result of BACK_TRACE_SCHEDULING algorithm for the example of Figure 3.

Finally, *ADD_GROUP* adds the returned *group* to *groups* and *UPDATE* removes the behaviors in *group* from *priority_list*. The immediate predecessors and successors for left *behaviors* are also updated accordingly. Then the number of groups *n* is reduced by one and another iteration starts to select the "second critical path". This process continues until *n*th critical path is found.

Algorithm 4 BACK_TRACE_SCHEDULING (n, priority_list, length, groups)

```
for group ∈ groups do
        group.latest_time = length;
endfor
for behavior ∈ priority_list do
        group.latest_time = ∞;
endfor
behavior = LAST(priority_list);
while behavior do
        behavior.end_time = MIN( behavior.latest_time
        , behavior.group.latest_time);
        behavior.start_time = behavior.end_time – behavior.op;
        behavior.group.latest_time = behavior.start_time;
        for pred ∈ immediate predecessors of behavior do
                pred.latest_time = MIN(behavior.start_time
                , pred.latest_time);
        endfor
```

Figure 7 gives the behavior-regrouping result for the example of Figure 3.

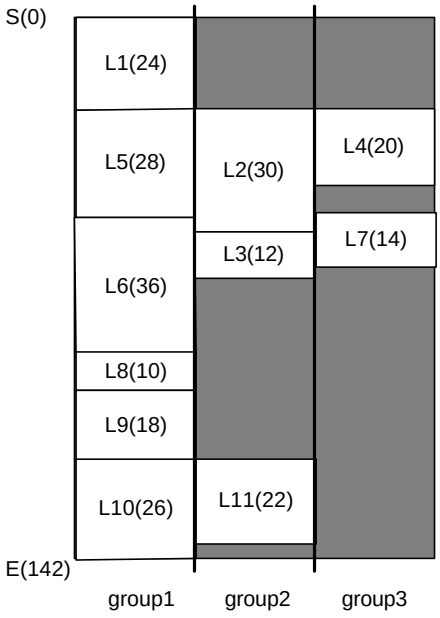


Figure 7: The behavior re-grouping result for the example of Figure 3.

6 PE Selection

Behavior grouping groups behaviors to n groups. In this section, we introduce *PE selection*, which select suitable PE for each group.

6.1 PE library

All the possible PEs are saved in PE libraries. We assume there are numerous PE libraries, each of which contains a number of PEs with compatible communication protocols. We assemble the system architecture by using the PEs in one library.

In each PE library, PEs are saved in a PE list in the decreasing order of cost. In general, the time performance of PE become worse while the cost of PE reduces. Therefore, we assume the execution time of any behavior *A* on PE *B* is always faster than the execution time on the PE after PE *B* in the PE lists.

We assume we have selected a PE library. Variable *PEs* represents the sorted PE list in the selected PE library.

6.2 Performance Estimation

When *behaviors* are executed on different types of PE, e.g. ASIC, microprocessor, or FPGA, the methods of execution time estimation are different. Furthermore, the execution time is influenced by the PE's parameters. For example, the execution time of a behavior executing on a microprocessor depends on the microprocessor's frequency.

In this report, we use the sum of weighted operations computed by spec profiler[20] as the behavior's execution time on the selected PE. Each PE has a weighted table to tell the execution time required for each type of operations, such as "+", "*", or "=". The product of the weight and the total execution number of operations for an operation type represents the execution time for that operation type in the behavior. The sum of weighted operations of all the operation types represents the execution time of the behavior executed on the PE. This type of estimation is called time-appropriate estimation. Designers can also use third-party estimation-tools to compute the estimation time.

6.3 PE Selection Algorithm

We use algorithm 5 to select PEs for groups.

First, *SELECT_FASTEST_PE* selects the fastest PE for each group. Then, at each outer iteration, we select the cheapest PE in the PE library for a

group without violating the given time constraint, while keeping the PEs for other groups unchanged. We start PE selection from group 1, which represents the critical path, and ends at group n, which represents the nth critical path. At each inner iteration, we select one PE for *behavior*, starting from the most expensive PE saved at the beginning of the PE list *PEs*. If the execution time returned by *FORWARD_TRACE_SCHEDULING* for variable *PE* and *group* is smaller than the given time constraint, then we map the group to that PE. The inner iteration continues until the returning *length* of *FORWARD_TRACE_SCHEDULING* is greater than the *Time_Constaint*.

Algorithm 5 PE Selection

```
SELECT_FASTEST_PE(groups);

group = FIRST(groups);
while group do
        PE = FIRST(PEs);
        while PE do
                length = FORWARD_TRACE_SCHEDUL(PE,
                group,  priority_list);
                If ( length ≤ Time_Constraint ){
                        group.pe = PE;
                }
                else{
                        break;
                }
                PE = NEXT(PEs);
        endwhile
        group = NEXT(groups);
endwhile
```

Algorithm 6 FORWARD_TRACE_SCHEDULING(PE, groups, priority_list)

```
for group ∈ groups do
        group.earliest_time = 0;
endfor

for behavior ∈ priority_list do
        behavior.start_time = 0;
endfor

behavior = FIRST(priority_list);
while behavior do
        behavior.exec = ESTIMATE_EXEC_TIME(behavior,
        behavior.group.pe);
        behavior.start_time = MAX( behavior.start_time
        ,  behavior.group.start_time);
        behavior.end_time = behavior.start_time + behavior.exec;
        behavior.group.ealiest_time = behavior.end_time;
        for succ ∈ immediate successors of behavior do
                succ.start_time = MAX(behavior.end_time
                , succ.start_time);
        endfor

        behavior = NEXT(behavior);
endwhile
```

FORWARD_TRACE_SCHEDUL is introduced in algorithm 6. It is also a static_list scheduling algorithm. In comparison to BASIC_SCHEDULING, it only reschedules behaviors inside the group without moving behavior across the groups. Furthermore, it uses the execution time of behavior to compute the *start_time* and *end_time* in stead of using the computation cost.

Table 2: PE attributes

	PE1	PE2	PE3
cost	$18	$10	$6

Table 3: The execution time of behaviors on different PEs

(ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
PE1	3	3.8	1.5	2.5	3.5	4.5
PE2	6	7.5	3.0	5	7	9
PE3	9.1	11.3	4.5	7.5	10.5	13.5
	L7	**L8**	**L9**	**L10**	**L11**	
PE1	1.8	1.2	2.2	3.2	2.8	
PE2	3.6	2.4	4.4	6.4	5.6	
PE3	5.4	3.6	6.6	9.6	8.4	

For the example of Figure 3, we assume the selected library contains three types of PEs. Table 2 lists the cost of PE and Table 3 lists the execution time of behaviors on PEs. We assume that the time constraint is 27ms. The processes of PE selection are displayed in Figure 8. After PE selection, we select PE1 for group1, PE2 for group2, and PE3 for group3.

7 Behavior Mapping

In section 6, we have selected PE to assemble the system architecture. During PE selection, we schedule behaviors on the selected PEs without moving behaviors across the groups. In this section, we re-map the behaviors to the selected PE without considering previous grouping.

We use a static-list-scheduling algorithm described in algorithm 7 to map behaviors to the selected PE. The behavior mapping algorithm is similar to the behavior grouping algorithm described in algorithm 2. The difference between them is that behavior mapping algorithm use the execution time of behavior on the selected PE while behavior grouping algorithm uses the computation cost *op*. Another difference is that the behavior mapping algorithm chooses the PE having the best end time rather than the PE having the best start time. In algorithm 7, input variable *groups* records the selected PE of each group. Algorithm first removes all the behaviors from groups by EMPTY_GROUP and then re-map behaviors to the groups to pursue the shortest execution time.

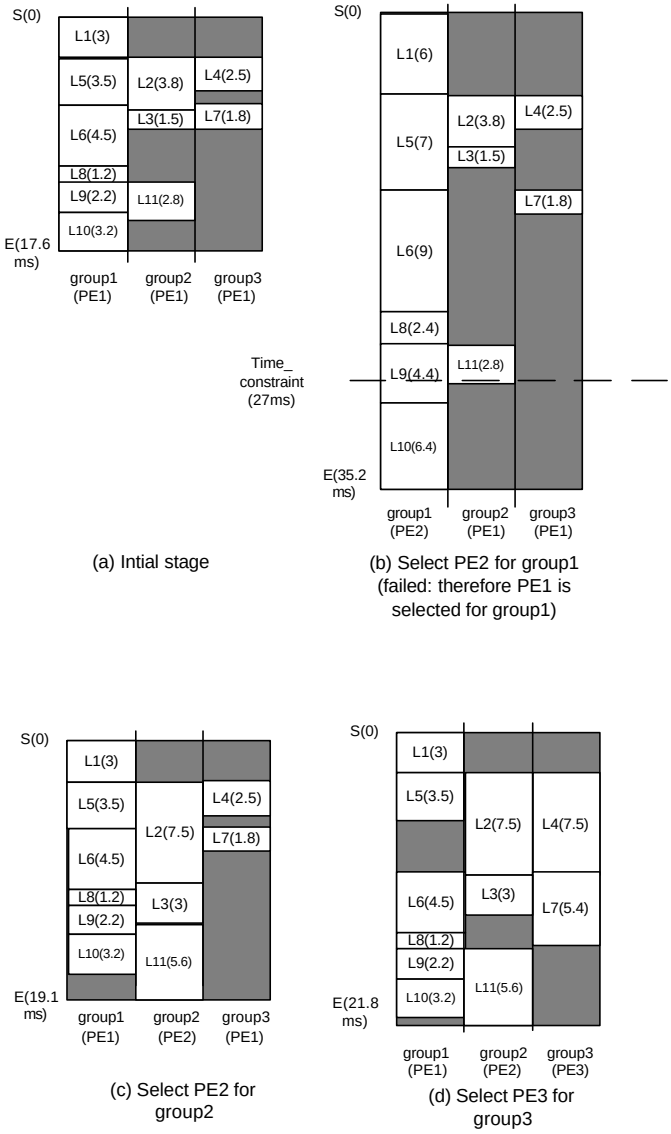


Figure 8: The processes of PE selection for the example of Figure 3. (Time constraint = 27ms)

Algorithm 7 BEHAVIOR_MAPPING (groups, priority_list)

```
EMPTY_GROUP(group);
RESET_BEHAVIOR();
behavior = FIRST(priority_list);
while behavior do
        early_end = ∞;;
        group = FIRST(groups);
        while group do
                start_time = max( group.eariest_time,
                behavior.earliest_time );
                end_time = start_time +
                ESTIMATE_EXEC_TIME(behavior, group.pe);
                if(end_time < early_end){
                        mapped_group = group;
                        early_end = end_time;
                }
                group = NEXT(groups);
        endwhile
        behavior.group = mapped_group;
        behavior.exec = ESTIMATE_EXEC_TIME(behavior,
        behavior.group.pe);
        behavior.start_time = max( behavior.group.eariest_time,
        behavior.earliest_time );
        behavior.end_time = behavior.start_time +
        behavior.exec;
        INSERT_TO_GROUPS(behavior, group, start_time,
        end_time);
        behavior = NEXT(priority_list);
endwhile
```

The result of behavior mapping for the example of Figure 3 is displayed in Figure 9. In comparison to the execution time 21.8 ms in Figure 8(d), the resulting execution time is 21.1ms.

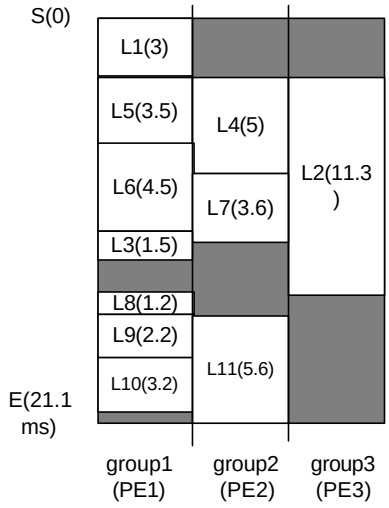


Figure 9: The behavior mapping result for the example of Figure 3.

8 PE Number Selection

Section 5 to 7 describes the algorithms of *behavior grouping*, *PE selection* and *behavior mapping* when the number of PEs in the system architecture is known. As shown in Figure 2, if designers don't know the number of PEs in the system architecture, then designers needs to define the lowerbound *l_n* and upperbound *m_n*.

n should be no less than 1 and no greater than the number of parallelism *max_p*. We define *max_p* as the largest number of behaviors that can be executed simultaneously. It is obvious that the shortest time of the design on the architecture containing *max_p* PEs is the same as the shortest execution time of the design on the architecture containing more than *max_p* PEs. Therefore, we can choose *l_n* as 1 and choose *m_n* as *max_p*.

However, if the *max_p* is too large, then it is time consuming to implement architecture exploration. Therefore, we choose another way to define *l_n* and *m_n*. First, we roughly estimate the optimal number of PE *np* as the total computation cost divided by the computation cost on the critical path. Then we define m as the range of PE number selection. Finally, we define *l_n* as n-⌈m/2⌉ and define *m_n* as n-⌈m/2⌉. In the example in Figure 3, *np* is 2. If m is 5, then *l_n* is 1 and *m_n* is 4.

After producing the system architectures containing different numbers of PEs, the designers can select the best one from them. We call this task *PE number selection*.

Table 4 : The table of PE number selection for the example of Figure 3.

PE Amount	group 1	group 2	group 3	group 4	Cost	Exec. time (ms)
1	PE1	--	--	--	$18	30
2	**PE1**	**PE2**	**--**	**--**	**$28**	**22.6**
3	PE1	PE2	PE3	--	$34	21.1
4	PE1	PE3	PE3	PE3	$36	25.6

Table 4 displays the result of PE number selection for the example in Figure 3. Since the maximal parallelism of the behavior model is four,

we produce four possible system architectures, which containing 1 to 4 PEs. The computation cost and execution time for the architectures containing different number of PEs are displayed in Table 4. As a result, designer can select 2 PE solution because of its lowest cost. The architecture contains PE1 and PE2. The cost of the architecture is $28. The execution time is 22.6ms, which meet the given time constraint 27ms.

9 Experimental Result

First of all, we implement the introduced algorithm by programming around 3000 lines of C++ code.

We then randomly generate 5 examples shown in Table 5. We assume there are three types of PEs in the PE library, which is shown in Table 2. In this section, we will discuss the advantage/limitation of our algorithms based on the exploration results of these examples.

Our research cannot be compared directly with other research mentioned in section 2 because they didn't provide the computation cost for behaviors.

The complexity of the introduced algorithm is $O(pPnlog(n)) + O(np^2log(n))$, where p is the number of PE in the system, n is the number of behavior, and P is the number of PE types in the PE library. The complexity in Wolf[13]'s work is $O(p^3n^2P)$. Furthermore, the upper-bound of p in our algorithm is the maximal parallelism while the upper-bound of p in Wolf's algorithm is n. In comparison to the previous research introduced in section 2, the complexity of our algorithm is lowest.

Table 5: The table of random generated testbenches

	Ex.1	Ex.2	Ex. 3	Ex. 4	Ex. 5
Num. of leaf behaviors	12	12	11	20	42
Max. Num. of parallelism	6	4	4	14	18

9.1 Number of Explored Architectures

The main contribution of this report is that we select suitable PEs for the system architecture with low complexity according to the system behavior model. If PE library contains three types of PEs and the system architecture contains n PEs, then during random PE selection, m^n possible system architectures may be explored. Using our algorithm, we only explore at most $(m \times n)$ system architectures, which dramatically reduced the algorithm's complexity. Table 6 lists the maximal number of explored system architectures during random PE selection and the PE selection we introduce.

Table 6: The comparsion of maximal number of explored system architectures for random PE selection and our PE selection algorithm.

Num. of explored architecture	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5
Random selection	729	81	82	4,782,969	387,420,489
Grouping based selection	18	12	12	42	54

The number of explored system architecture of Wolf[13] and Ten[15]'s work cannot be evaluated.

9.2 Design Time

We also compare the design time required by manual architecture exploration and required by automatic architecture exploration through running the C++ code we made, for examples 1, 2, and 3, which are displayed in Table 7.

Table 7: The comparison of the design time of manual exploration and automatic exploration

Design Time	Ex. 1	Ex. 2	Ex. 3
Manual exploration	≈8640s	≈9000s	≈7920s
Automatic exploration	<1s	<1s	<1s

The manual exploration takes around 8000 seconds on an average, while the automatic exploration takes less than 1 second, which is 8000 times faster than the manual one. Therefore, we conclude the automatic exploration shortens the design time dramatically.

9.3 Generated System Architecture

One goal of the grouping-based architecture exploration is to select suitable PEs for the system architecture. In this sub section, we test whether this goal is reached.

Following our design flow, the strategy of behavior grouping determines the PE selection. Different grouping strategies will produce different system architectures.

One attribute of our grouping is *group balancing*, which balances the computation among groups. Optimal *group balancing* ensures that the length of the critical path of the generated groups is the shortest one among the length of the critical path of all the possible groups. Therefore, in comparison to the fastest PE in any possible system architecture that makes the implementation meets the time constraint, the fastest PE selected based on the optimal *group balancing* is the slowest. Since the static-list scheduling produces excellent balance solution, the fastest PE in our generated system architecture is close to the slowest PE that can be selected as the fastest PE for the design.

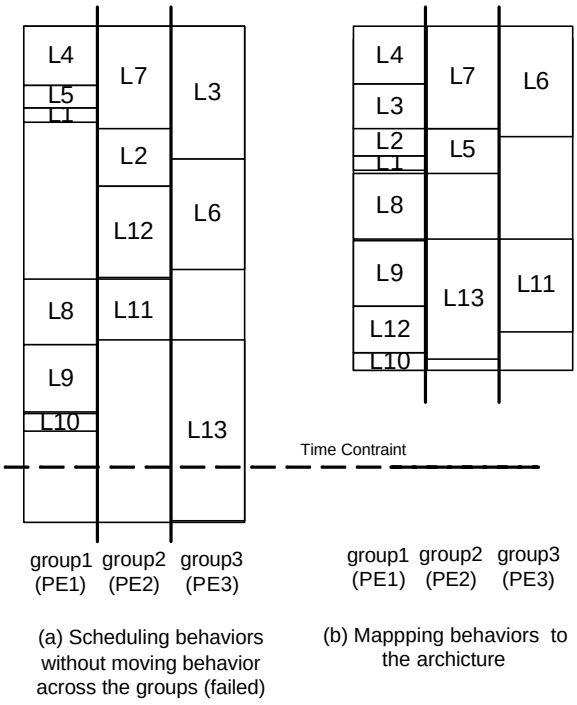


Figure 10: The example of the limitation of grouping-based PE selection

Second, behavior groups are balanced in terms of computation cost. After selecting different PEs for different groups, the groups are not balanced anymore in terms of execution time. This makes the generated system architecture not optimal. For example, in Figure 10, assume we have selected PE1 for group1 and PE2 for group. We also have proved that PE2 can be selected for group3. During testing PE3 for group3 (PE3 is slower than PE2), we fail for scheduling, which is shown in Figure 10(a). As a result, we select PE1 and two PE2 to assemble the system architecture. However, if we choose the system architecture containing a PE1, a PE2, and a PE3, and use behavior mapping algorithm in algorithm 7 to map behaviors to the architecture, the mapping meet the time constraint, which is shown in Figure 10(b). This is because the behaviors are balanced in terms of execution time rather than the computation cost. However, since the execution time cannot be known before the PE selection, it is impossible to select PEs according to the execution time of behaviors except the exhaust exploration. Therefore, the generated architecture based on our algorithms is a low cost architecture for the design but not the lowest cost one. As a

result, we add *behavior mapping* after *PE selection*. If designers want to generate the lowest cost architecture that can meet the design requirement, designers can manually optimize the generated system architecture and apply *behavior mapping* again to ensure that design can meet the given time constraint on the optimized system architecture. It should be noted that this problem also exists in Yen's work[15].

10 Conclusion

This report introduces the grouping-based architecture exploration of the system level design. The grouping-based architecture exploration selects PEs to assemble the system architecture according to the designs' functionality and maps behaviors to the selected PEs.

This report has the following three contributions.

First, it defines the design flow of the grouping-based architecture exploration. After *specification tuning*, designers first group behaviors. Designers then determine the number of PE and the selected PE types in the system architecture, according to the behavior model and behavior grouping results. Finally, behaviors are reassigned to the selected PEs to shorten the execution time. This design flow generates the application specific SoC.

Second, in comparison to the complexities of the previous work, the complexity of our algorithm is the lowest, which is $O(pPn\log(n)) + O(np^2\log(n))$, where p is the number of PEs in the system, n is the number of behaviors, and P is the number of PE types in the PE library. We compare the result of our grouping-based algorithm with the best result that we can manually generate. We find that the our *PE selection* algorithm produces satisfactory results for most of randomly generated testbenches. Furthermore, our *behavior-architecture mapping* algorithm produces near to optimal solution, which is proved by [2]. Since our *PE selection* algorithm does not guarantee the optimal solution, designers can also manually improve the result of our *PE selection* algorithm to gain better PE selection solution and apply the *behavior-architecture mapping* algorithm on it.

Third, we implement our algorithm by programming around 3000 lines of C++ code. For our random generated testbenches, the implemented C++ program produces the architecture exploration results in less than 1 second, where manual architecture exploration took us 2 to 3 hours on an average. It proves that the program based on our algorithm shortens the design time dramatically.

Reference:

[1] D. Gajski " Silicon Compilers" , Addison-Wesley, 1987

[2] G. Bell, etc. " A Comparison of List Schedules for Parallel Processing Systems". Commun. ACM, 1974, 17, 685--690

[3] Yu-Kwong Kwok, Ishfaq Ahmad " Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors" ", *IEEE Trans. on Parallel and Distributed System.* May 1996 (Vol. 7, No. 5)

[4] B. Shirazi, M.Wang, and G. Pathak. "Analysis and evaluation of heuristic methods for static task scheduling" *J. of Parallel and Distributed Computing*,10(3):222– 232, Nov. 1990.

[5] Min-You Wu, Daniel D. Gajski " Hypertool: A Programming Aid for Message-Passing Systems", *IEEE Trans. on Parallel and Distributed Systems*, 1(7):330– 343, July 1990.

[6] R. Ernst, J. Henkel, and T. Benner. " Hardware software co-synthesis for microcontrollers" . IEEE Design and Test of Computers. 10(4), December 1993.

[7] R.K. Gupta and G. D. Micheli. " Hardware-software co-synthesis for digital systems" . IEEE Desgin & Test of Computers. July/September 1993 (Vol. 10, No. 3)

[8] Kurt Keutzer, Sharad Malik, A. Richard Newton, Jan M. Rabaey, A. Sangiovanni-Vincentelli, " System-Level Design: Orthogonalization of Concerns and Platform-Based Design" . IEEE transactions on computer-aiede design of integrated circuits and systems, December 2000

[9] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, " System Level Design with Spade: an M-JPEG Case Study." In Proce. of Int. Conference on Computer Aided Design. (ICCAD'01)

[10] Grant Martin, Jean-Yves Brunel " Platform-Based Co-Design and Co-Development: Experience, Methodology and Trends" The ninth IEEE/DATC Electronic Design Process workshop (EDP-2002)

[11] http://www.cadence.com/products/vcc.html

[12] S. Prakash and A. C. Parker. " SOS: synthesis of application-specific heterogenous multiprocessor systems." Journal of Parallel and Distributed Computing, 16, 1992

[13] Wayne H. Wolf "An architectural Co-Synthesis Algorithm for Distributed, embedded computing systems". IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.5, (no.2), IEEE, June 1997. p.218-29

[14] J. G. Dambrosio and X. Hu. "Configuration-level hardware/software partitioning for real-time embedded systems." In Proceeding, International Workshop on Hardwre-Software Co-Design, 1994

[15] Ti-Yen Yen and Wayne Wolf " Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems". of the Eighth International Symposium on System Synthesis, Cannes, France, 13-15 Sept. 1995.)
[16] Lukai Cai, Daniel D. Gajski "Specification Tuning of System-Level Design" Technical Report CECS-18 University of California, Irvine. April 2002
[17] Gajski, D., N. Dutt, A. wu, S. Lin, " High-Level Synthesis - Introduction to Chip and System Design", Kluwer Academic Publishers, 1993
[18] D. Gajski " Silicon compilers" , Addison-Wesley, 1987
[19] D. Gajski, J. Zhu et al. " SpecC: Specification Lanugaeg and Design Methodology" Kluwer Academic Publishers, 2000
[20] Lukai Cai, Dan Gajski, " Introduction of Design-Oriented Profiler of SpecC Language", University of California, Irvine, Technical Report ICS-00-47, June 2001
[21] B. R. Rau, J A. Fisher " Instruction-Level Parallel Processing: History, Overview, and Perspective" Journal of Supercomputing, vol.7, (no.1-2), May 1993. p.9-50.
[22] R. Jain, A. Sharma and H. Wang, " Empirical Evaluation of some High-Level Synthesis Scheduling Heuristics," 28th Design Automation Conference, 1991