Parallelization Optimization of System-Level Specification

Lukai Cai
Daniel D. Gajski

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

Abstract

This paper introduces the parallelization optimization of system-level specification, which explores maximal parallelism among functional blocks of the design. We introduce two tools, spec profiler and spec optimizer, to implement the parallelization optimization automatically.

Index

List of Figures

List of Tables

Parallelization Optimization of System-Level Specification

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

Abstract

This paper introduces the parallelization optimization of system-level specification, which explores maximal parallelism among functional blocks of the design. We introduce two tools, spec profiler and spec optimizer, to implement the parallelization optimization automatically.

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 design's functionality, 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. Each PE implements a number of functional blocks in the specification. The system level design contains a series of tasks including *PE allocation* and *behavior binding*. *PE allocation* selects PEs for the architecture. *Behavior binding* maps different function blocks in the specification to different PEs.

In addition to tasks in existing system level design, we add task *specification tuning* to the system design flow, which is denoted by the dotted circle around point S. *Specification tuning* not only reduces the complexity of the specification, but also explores maximal parallelism existing in the specification, which are used for tasks *PE allocation* and *behavior binding* in later steps. For example, if only two functional blocks can be executed in parallel in the specification, then *PE allocation* will choose no more than two PEs in the architecture because of parallel execution. For the same reason, *behavior binding* also maps the two functional blocks to different PEs.

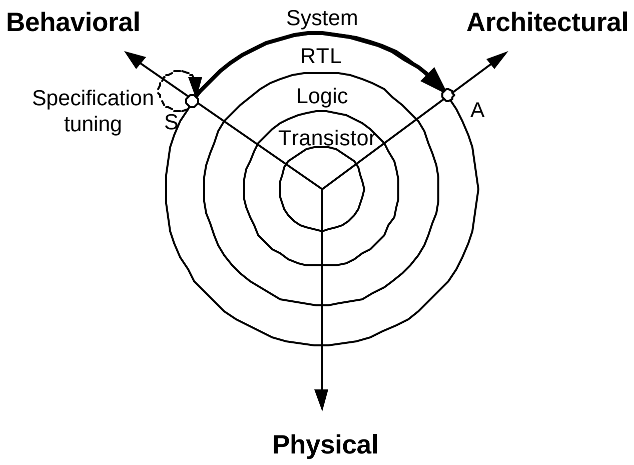

Figure 1: Extended Gajski and Kuhn's Y chart

This paper introduces the *parallelization optimization* of *specification tuning*, which exploits maximal parallelism among functional blocks of the design's specification. Designers can implement *parallelization optimization* manually. In general, designers start modeling the specification from existing C/C++ code. Since C/C++ language does not support parallelism, designers must manually find the parallelism by analyzing the code or designs' algorithms, which is time-consuming.

After finding the parallelism among the functional blocks in the specification, designers must determine the hierarchical parallel structure of the specification. After parallelization optimization, one original specification may produce different hierarchical parallel structures. For example, in Figure 2(a), functional blocks A, B, C, and D are executed sequentially. In Figure 2(b), the dependencies among the functional blocks are displayed. Block C can only be executed after the execution of A, while block D can only be executed after the execution of B. In Figure 2(c) and (d), two possible hierarchical parallel structures are shown. The functional blocks separated by dotted line represent parallel executed blocks. In Figure 2(c), block C and D are executed parallel after the parallel execution of A and B. In Figure 2(d), block C is executed after A while block D is executed after B. The execution of A and C is parallel with the execution of B and D. Because one original specification may produce different hierarchical parallel structures, we prefer implementing parallelization optimization structurally by tools rather than randomly by hand.

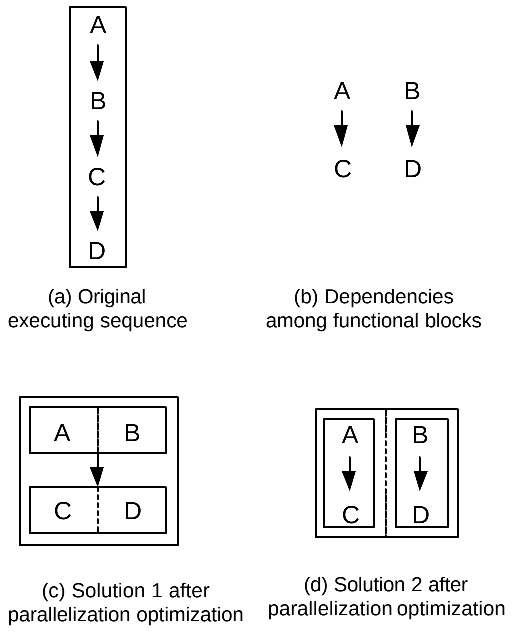
A
B
C
D
(a) Original executing sequence
A
B
C
D
(b) Dependencies among functional blocks
A
B
C
D
(c) Solution 1 after parallelization optimization
A
B
C
D
(d) Solution 2 after parallelization optimization

Figure 2: Example 1 of parallelization optimization

Therefore we make two tools, spec profiler and spec optimizer, to implement parallelization optimization automatically: spec profiler analyzes the dependencies among functional blocks; spec optimizer finds out the hierarchical parallel structure with maximal parallelism. We compared the manual parallelization with the automatic parallelization and concluded the automatic parallelization produced better results in terms of design time and hierarchical parallel structures.

We use SpecC language[2][3] to model the specification. In contrast to other system level design languages such as SystemC[4], SpecC language is a synthesis-based design language because it provides keywords such as *par* and *pipe* to model parallel and pipeline executing relations among functional blocks. Explicitly specifying the executing relations enables system-level synthesis tools to recognize the hierarchical parallel structures, which make it possible for them to implement *PE selection* and *behavior binding* automatically.

SpecC uses a keyword *behavior* to represent a functional block. Each *behavior* contains a number of methods that define the functionality, a set of ports that connect it with other *behaviors,* and a number of *behavior instances* to support *behavior* hierarchical modeling.

The paper is organized as follows: Section 2 describes the implementation of the automatic parallelization; Section 3 introduces the specification modeling process with the automatic parallelization; Section 4 gives experimental results. Finally, the conclusion is made in Section 5.

2 Implementation of Parallelization Optimization

2.1 Parallelization Optimization Tasks

In this paper, we parallelize sequential behaviors. A sequential behavior is defined as the behavior that only contains a number of sequential executing behavior instances.

The parallelization optimization contains three tasks shown in Figure 3. The first task, *sequential behavior searching*, finds all the sequential behaviors in the specification. The second task, *dependency analysis*, computes the dependencies among behavior instances of the sequential behaviors. Finally, the third task, *instance structure optimization*, finds the hierarchical parallel structure for each sequential behavior according to the dependencies.

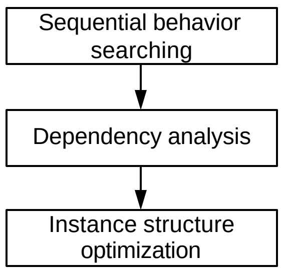
Sequential behavior searching
Dependency analysis
Instance structure optimization

Figure 3: Parallelization optimization tasks

2.2 Sequential Behavior Searching

We first find all the sequential behaviors in the specification. Sequential behaviors are identified by internally attributes of SpecC internal representing format.

2.3 Dependency Analysis

2.3.1 Definition

A sequential behavior A contains behavior instances B and C, a set of local variable V_i, and a set of port P_j. B is executed before C. If there exists a V_i or P_j that

(a) B write to V_i/P_j and C reads from V_i/P_j, or
(b) Both B and C write to V_i/P_j, or
(c) There exists a behavior instance D of A such that D depends on B and C depends on D,

then behavior instance C depends on behavior instance B.

If Behavior instance C depends on behavior instance B, then C must be executed after the execution of B. Otherwise, Behavior instances B and C can be executed parallel.

2.3.2 Dependency analysis

We compute the dependencies among behavior instances by analyzing the port traffic of behavior instances.

First, we use a spec profiler [5] to produce the specification statistics. Spec profiler generates the static traffic and dynamic traffic of behavior ports. Static traffic of the port refers to the number of ports of leaf behaviors to which it is connected. Leaf behavior is the behavior containing only a set of methods without any behavior instances, which is used as the instance of other behaviors. Dynamic traffic of the port refers to the number of port access during simulation. If the port is an input port, and static/dynamic traffic is greater than 0 for that port, we conclude that the behavior statically/dynamically read from the port. Likewise, if the port is an output port and static/dynamic traffic is greater than 0, we conclude that the behavior statically/dynamically write to the port. The "inout" port can be treated in a similar way.

Second, we analyze the port connections of behavior instances. If behavior instances B and C of sequential behavior A meet the conditions (a) or (b) in 2.3.1 statically or dynamically, then C statically or dynamically depends on B.

Finally, we find the static/dynamic behavior dependencies based on the condition (c) in 2.3.1.

After dependency analysis, designers can determine whether one behavior instance depends on another based on either static dependency or more greedy, dynamical dependency.

2.4 Instance Structure Optimization

2.4.1 Hierarchical Parallel Structure

Instance structure optimization changes the instance structure from one-level pure-sequential structure to multi-level hierarchical parallel structure. Figure 4 gives an example of the hierarchical parallel structure. After instance structure optimization, the produced hierarchical parallel structure has three levels shown in Figure 4(c). In the first level, D and E are parallel executed. In the second level, B is executed before the execution of D and E. In the third level, A and C(C is executed after A) are executed parallel with B, D, and E. Note that two of three levels are parallel structure.

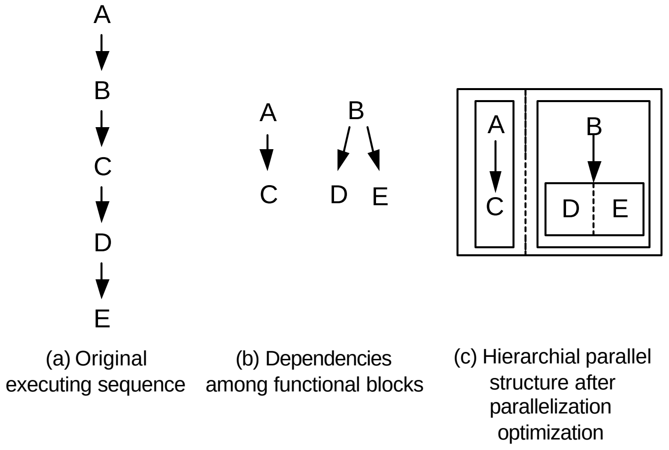
A
B
C
D
E
(a) Original executing sequence
A
C
B
D
E
(b) Dependencies among functional blocks
A
C
B
D
E
(c) Hierarchial parallel structure after parallelization optimization

Figure 4: Example 2 of parallelization optimization

2.4.2 Goals of Instance Structure Optimization

During instance structure optimization, we want to achieve two goals.

(a) Minimize the number of added dependencies among behavior instances.

After instance structure optimization, some independent behavior instances will be changed to dependent behavior instances because of overuse parallelism. For example, the solution shown in Figure 2(c) adds two pairs of dependencies: D depends on A and C depends on B, while do not exist in Figure 2(b). Adding dependencies among behavior instances are unavoidable; therefore we choose minimizing the number of added dependencies as the first goal.

(b) Minimize the length of critical path of produced hierarchical parallel structure.

The length of the critical path of hierarchical parallel structure is defined as the number of behavior instances on the longest path from the first starting behavior instance to the last ending behavior instance, while parallel-executed instances can be executed simultaneously.

2.4.3 Algorithms for Instance Structure Optimization

We implemented two algorithms for instance structure optimization: ASAP(as soon as possible) algorithm and constructive algorithm.

2.4.3.1 ASAP Algorithm

Algorithm 1 outlines the ASAP algorithm for instance structure optimization for each sequential behavior. *B* is an instance group that contains a set of behavior instances in sequential behaviors. *Hier_Struct* denotes the generated hierarchical parallel structure containing a link of groups, each of which is executed sequentially from the head to the tail of the link. The function *DependentOnB(b)* returns Φ if no behavior instance on which b depends is in *B*, otherwise it returns the first instance on which *b* depends. *CurGroup.Append(b)* inserts b to a group *CurGroup*. All the instances in *CurGroup* are executed parallel. After the execution of *for* loop each time, *CurGroup* records a set of parallel executing behavior instances. *Hier_Struct.Append(CurGroup)* then appends the current *CurGroup* at the end of the link of *Hier_Struct*. Figure 2(c) is the hierarchical parallel structure generated by ASAP algorithm.

The ASAP algorithm has only goal (b), which is to minimize the length of the critical path. It gives the optimal solution in terms of the critical path but may add a large amount of dependencies among behavior instances.

Algorithm 1: ASAP Algorithm.

```
B = {all the behavior instances};
Hier_Struct = {};

while B ≠ Φ  do
   CurGroup = {};
   for each instance b_i ∈ B do
      if  DependentOnB(b_i ) = Φ  then
         CurGroup = CurGroup.Append(b_i , Par);
         B  = B - {b_i );
      do
   endfor
   Hier_Struct = Hier_Struct.Append(CurGroup,
Sequ);
do
```

2.4.3.2 Constructive Algorithm

Besides ASAP algorithm, we also implemented a constructive algorithm. The constructive algorithm schedules one behavior instance at a time in the order of the execution sequence in the original sequential behavior and produces a temporal hierarchical parallel structure. It is constructive because it constructs the hierarchical parallel structure without performing any backtracking, i.e. changing the previously produced temporal structure. The constructive algorithm has both goals (a) and (b) during instance structure optimization. Figure 2(d) is the hierarchical parallel structure generated by the constructive algorithm.

2.4.3.2.1 Data Structure of Hierarchical Parallel Structure

Before introducing the constructive algorithm, we first specify the data structure for the hierarchical parallel structure in the algorithm. Each hierarchical parallel structure is represented by a data structure *ParGroup*. Each *ParGroup* contains a set/link of child *ParGroups*, or a link of behavior instances. The items in the links are executed sequentially from head to tail of the link. The items in the set are executed parallel. Figure 5 shows three types of *ParGroups. Flat ParGroup* contains a behavior instance link without any child *ParGroups. Par ParGroup* contains a set of child *ParGroups. Sequ ParGroup* contains a link of child *ParGroups.*

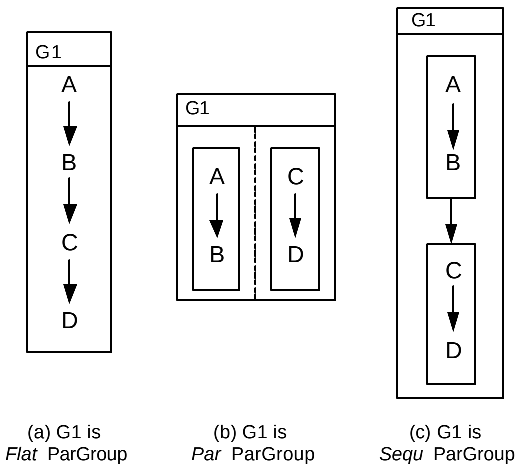

Figure 5: Three types of ParGroup

2.4.3.2.2 Algorithm Overview

Algorithm 2.1 outlines the constructive algorithm. The behavior instance link *B* contains all of the behavior instances of the sequential behavior, which are saved in the order of execution sequence of the sequential behavior. Starting from the head of link *B*, an instance of *B* is inserted into a *ParGroup Hier_Struct* at a time by function *Insert*. Function *Insert* calls different inserting functions according to the type of Hier_Struct. After all of the instances in *B* are inserted, *Hier_Struct* represents the final hierarchical parallel structure.

Algorithm 2.1: The Constructive Algorithm

```
B = {all the behavior instances}
Hier_Struct = {};

for each instance b_i ∈ B do
   Insert(Hier_Struct, b_i);
endfor


Function Insert(Hier_Struct, b)
switch Type(Hier_Struct) do
case FLAT: Hier_Struct = InsertToFlat(Hier_Struct,
b_i);
break;
case PAR: Hier_Struct = InsertToPar(Hier_Struct, b_i);
break;
case SEQU: Hier_Struct = InsertToSequ(Hier_Struct,
b_i);
endswitch
```

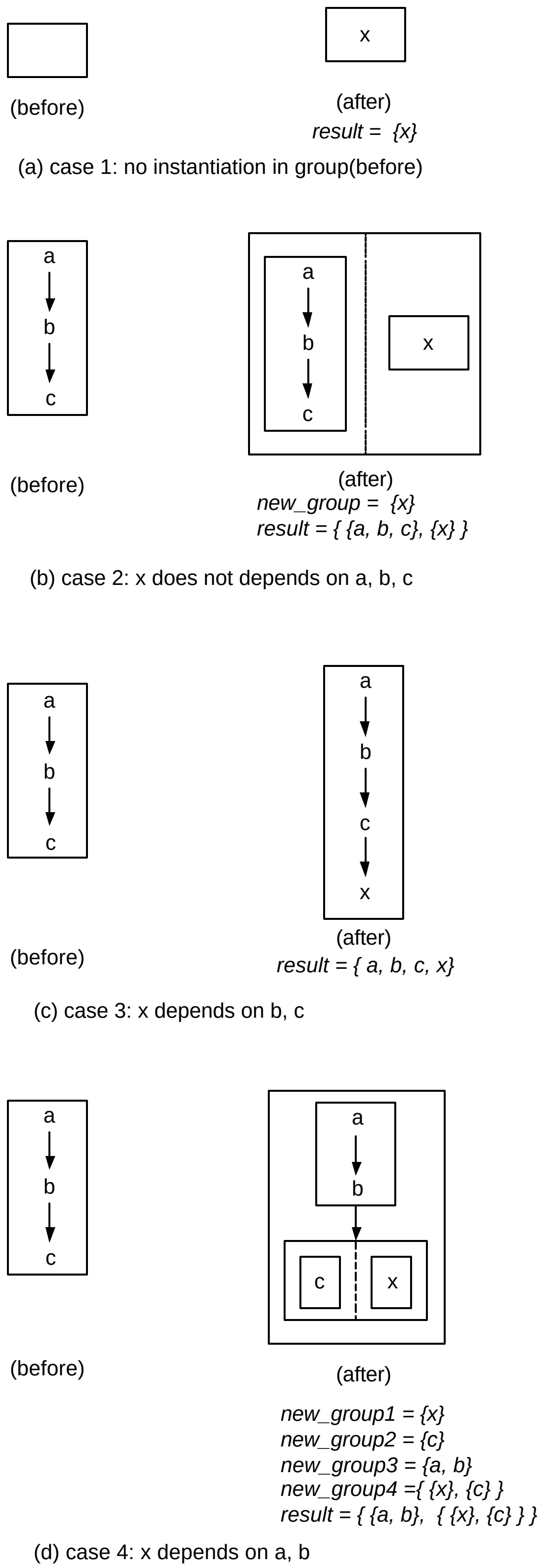
x
(before)
(after)
result = {x}
(a) case 1: no instantiation in group(before)
a
b
c
a
b
c
x
(before)
(after)
new_group = {x}
result = { {a, b, c}, {x} }
(b) case 2: x does not depends on a, b, c
a
b
c
a
b
c
x
(before)
(after)
result = { a, b, c, x}
(c) case 3: x depends on b, c
a
b
c
a
b
c
x
(before)
(after)
new_group1 = {x}
new_group2 = {c}
new_group3 = {a, b}
new_group4 ={ {x}, {c} }
result = { {a, b}, { {x}, {c} } }
(d) case 4: x depends on a, b

Figure 6: Four cases of inserting instance x to a Flat ParGroup.

2.4.3.2.3 Insert to Flat ParGroup

Algorithm 2.2 outlines the function *InsertToFlat* that inserts an instance *b* to a *Flat ParGroup Hier_Struct*. *InsertToFlat* contains four different cases according to different dependency relations between b and instances in *Hier_Struct*'s instance links. The *result* records the produced hierarchical parallel structure. The examples of the four cases are displayed in Figure 6.

```
Algorithm 2.2: InsertToFlat(Hier_Struct, b)

// Case 1
if  NoInstInGroup(Hier_Struct) = 1  do
   result = AppendInst(Hier_Struct, b);

// Case 2
else if NotDependOnGroup(Hier_Strcut, b) = 1 do
   new_group = Group(b, FLAT);
   result = Group(Hier_Struct, new_group, PAR)

// Case 3
else if DependOnLastInst(Hier_Strcut, b) = 1 do
   result = AppendInst(Hier_Struct, b);

// Case 4
else if
   d1 = FindLastDependInst(Hier_Strcut, b);
   new_group1 = Group(b, FLAT);
   new_group2 = Group( AllSucc(Hier_Struct,d1),FLAT);
   new_group3 = Group( AllPred(Hier_Struct,d1),FLAT);
   new_group4 = Group( new_group1, new_group2, PAR);
   result =     Group(new_group3, new_group4, SEQU);
endif

return result;
```

In the first case, function *NoInstInGroup* finds whether *Hier_Struct*'s instance link contains any instances. If not, *b* is inserted in to the link by function *AppendInst*. In the second case, if function *NotDependOnGroup* finds that *b* does not depend on any instances in the link, then function *Group* creates a new *Flat ParGroup new_group* containing only *b* and creates a new *Par ParGroup result* which contains *Hier_Struct* and *new_group* as its child *ParGroups*. In the third case, function *DependOnLastInst* finds whether *b* depends on the last instance in the link. If so, *AppendInst* appends *b* to the end of the instance link of *Hier_Struct*.

In the last case, function *FindLastDependInst* finds the latest instance *d1* on which *b* depends. The latest instance refers to the instance that is most close to the tail of the instance link of *Hier_Struct*. *New_group1* is a new *Flat ParGroup* containing *b*. *New_group2* is another new *Flat ParGroup* containing all the instances following *d1* in the instance link of *Hier_Struct*. The instances in *New_group2* are stored in *New_group2's* instance link in the same order as that of *Hier_Struct*. *New_group3* is the third new *Flat ParGroup* that contains all the instances in front of *d1* inclusively, saved in the same order as that of *Hier_Sturt*. *New_group4* is a *Par ParGroup* containing *new_group1* and *new_group2*. The *result* is a new *Sequ ParGroup* containing *new_group3* followed by *new_group4* in its child *ParGroup* link.

2.4.3.2.4 Insert to Par ParGroup

Algorithm 2.3 outlines the function *InsertToPar* that inserts an instance *b* to a Par *ParGroup Hier_Struct*. *InsertToPar* contains three different cases according to different dependency relations between b and child *ParGroups* in *Hier_Struct*'s instance. We define that an instance A depends on *a ParaGroup* B if and only if A depends on at least one instance in *ParaGroup B*. The *result* records the produced hierarchical parallel structure. The examples of the three cases are displayed in Figure 7.

```
Algorithm 2.3: InsertToPar(Hier_Struct, b)

// Case 1
if NotDependOnChildGroup(Hier_Strcut, b) = 1 do
   new_group = Group(b, FLAT);
   result = AddChildGroup(Hier_Struct, new_group);

// Case 2
else if DependOnOneChildGroup(Hier_Struct, b) = 1 do
   sub_group = FindDependChildGroup(Hier_Struct, b);
   result = Insert(sub_group, b);

// Case 3
else if
   new_group1 = Group(b, FLAT);
   new_group2 = Group( DependChildGroup(Hier_Struct,d1)
                ,PAR);
   new_group3 = Group(IndependChildGroup(Hier_Struct,d1)
                ,PAR);
   new_group4 = Group( new_group2, new_group1, SEQU);
   result =     Group(new_group3, new_group4, PAR);
endif
```

In the first case, if function *NotDependOnChildGroup* finds that *b* does not depend on any child *ParGroups* of *Hier_Struct*, then function *Group* creates a new *Flat ParGroup new_group* containing *b*. Function *AddChildGroup* then adds *new_group* into *Hier_Struct* as its child *ParGroup*.

In the second case, if function *DependOnOneChildGroup* finds that b only depends on one child ParGroup of Hier_struct denoted by sub_group, then function Insert described in Algorithm 2.1 inserts b to sub_group.

In the last case, if *b* depends on more than one child *ParGroups* of *Hier_struct*, then five new *ParGroups* will be produced. *New_group1* is a *Flat ParGroup* containing *b*. *New_group2* is a Par *ParGroup* containing all the child *ParGroups* of *Hier_struct* that *b* depends on. *New_group3* is a Par *ParGroup* containing all the child *ParGroups* of

Hier_struct that *b* does not depend on. *New_group4* is *Sequ ParGroup* containing *new_group2* followed by *new_group1* in this child *ParGroup* link. Finally, the *result* is a *Par ParGroup* containing child *ParGroup* *new_group3* and *new_group4* in its child *ParGroup* set.

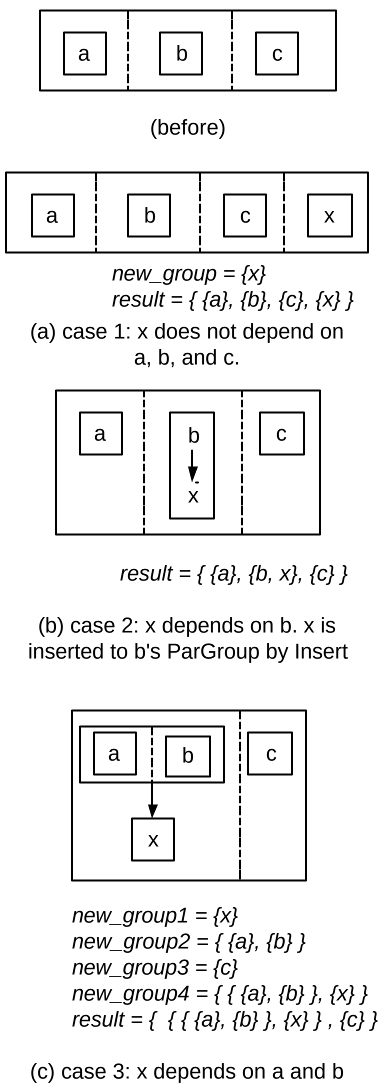
a
b
c
(before)
a
b
c
x
new_group = {x}
result = { {a}, {b}, {c}, {x} }
(a) case 1: x does not depend on a, b, and c.
a
b
x
c
result = { {a}, {b, x}, {c} }
(b) case 2: x depends on b. x is inserted to b's ParGroup by Insert
a
b
c
x
new_group1 = {x}
new_group2 = { {a}, {b} }
new_group3 = {c}
new_group4 = { { {a}, {b} }, {x} }
result = { { { {a}, {b} }, {x} } , {c} }
(c) case 3: x depends on a and b

Figure 7: Three cases of inserting instance x to a *Par ParGroup*.

2.4.3.2.5 Insert to *Sequ* ParGroup

Algorithm 2.4 outlines the function *InsertToSequ* that inserts an instance *b* to a *Sequ ParGroup Hier_Struct*. *InsertToSequ* contains three different cases according to different dependency relations between b and child *ParGroups* in *Hier_Struct*. The examples of the three cases are displayed in Figure 8.

Algorithm 2.4: InsertToSequ(Hier_Struct, b)

```
// Case 1
if NotDependOnChildGroup(Hier_Strcut, b) = 1 do
   new_group = Group(b, FLAT);
   result = result = AddChildGroup(Hier_Struct,
   new_group);

// Case 2
else if DependOnLastChildGroup(Hier_Struct, b)
        = 1 do
   child_group = FindLastChildGroup(Hier_Struct);
   result = Insert(child_group, b);

// Case 3
else if
   last_depend_child = FindLastDependChildGroup
                       (Hier_Struct, b);
   next_child = Next(last_depend_child);
   solution1 = Insert(last_depend_child, b);
   solution2 = Insert(next_child, b)
   result = Best(solution1, solution2);
```

The first case of *InsertToSequ* is the same as the first case of *InsertToPar*. In the second case, if function *DependOnLastChildGroup* finds that *b* depends on the tail *ParGroup* of child *ParGroup* link of *Hier_struct*, then function *Insert* described in Algorithm 2.1 inserts *b* to this child *ParGroup child_group*.

In the third case, function *FindLastDependChildGroup* finds *last_depend_group*, which is the child *ParGroup* in its child *ParGroup* link that is most closest to the tail of its child *ParGroup* link, among the child *ParGroups* on which b depends. *Next_child* is the immediate successive child *ParGroup* of *last_depend_group* in the link. Then two alternate solutions, inserting *b* in *last_depend_group* and inserting b in *next_child*, are explored. The first solution ensures that its amount of added dependencies are not greater than that of the second solution, while the second solution ensures that its length of critical path is not longer than that of the first solution. Finally, function *Best* chooses the *solution1* in the case that the length of critical path of *solution1* is not longer than that of *solution2*. Otherwise, *Best* chooses *solution2* as the result.

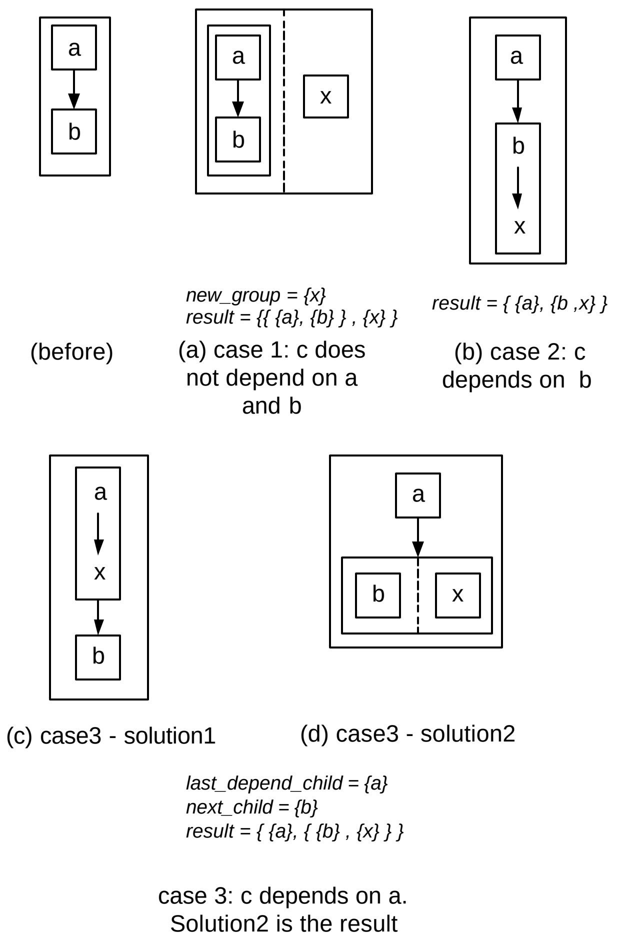
a
b
a
b
x
a
b
x
new_group = {x}
result = {{ {a}, {b} } , {x} }
result = { {a}, {b ,x} }
(before)
(a) case 1: c does not depend on a and b
(b) case 2: c depends on b
a
x
b
a
b
x
(c) case3 - solution1
(d) case3 - solution2
last_depend_child = {a}
next_child = {b}
result = { {a}, { {b} , {x} } }
case 3: c depends on a. Solution2 is the result

Figure 8: Three cases of inserting instance x to a *Sequ ParGroup*.

3 Specification Modeling Process

We introduce the process of specification modeling using the spec profiler and the spec optimizer, which contains three steps. First, designers write SpecC specification model by referencing original C/C++ code. Designers can specify top level parallelism among behavior instances according to design algorithms/standards. Second, designers use the spec profiler and the spec optimizer. The tools read SpecC specification model and generate hierarchical parallel structures in the format of textural file for sequential behaviors. Third, designers optimize the specification model based either on the result of ASAP algorithm or on the result of the constructive algorithm.

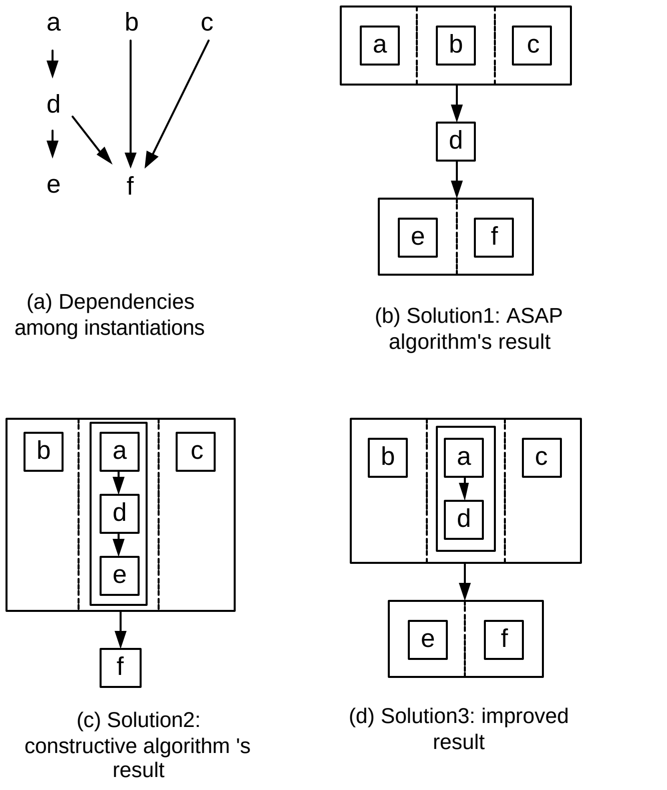
a
b
c
d
e
f
(a) Dependencies among instantiations
a
b
c
d
e
f
(b) Solution1: ASAP algorithm's result
b
a
d
e
c
f
(c) Solution2: constructive algorithm 's result
b
a
d
c
e
f
(d) Solution3: improved result

Figure 9: An example of designers ’ improvements on the results of constructive algorithm

In the third step, designers can also change the result of constructive algorithm by referencing ASAP algorithm for shorter critical path, which is illustrated in Figure 9. By referencing the result of ASAP algorithms shown in Figure 9(b), designers can change the result of constructive algorithm shown in Figure 9(c), to parallel execute instance *e* and *f*. As shown Figure 9(d), the improved solution has the shorter critical path than the solution in Figure 9(c). Table 1 gives overview of three solutions.

Table 1 : Overview of three solutions for the example in Figure 9

	Added Dependency	Length of critical path
ASAP	4	3
Constructive	1	4
Improved	2	3

4 Experimental results

We evaluate the efficiency of the spec profiler and the spec optimizer in terms of design time, the length of the critical path of the resulting hierarchical parallel structure, and the added dependencies of the resulting structure.

We chose four sets of testing examples. First, we chose three examples for comparing the manual parallelization with the automatic parallelization. Second, we chose behaviors with no more than 10 instances. Third, we chose behaviors with more than 20 instances. Finally, we chose real project examples.

4.1 Manual Parallelization vs. Automatic Parallelization

First, we evaluate the efficiency of the spec profiler and the spec optimizer in terms of design time. We randomly generated three sequential behaviors, two of which contains 10 behavior instances, the rest of which contains 20 behavior instances. The required design time for the manual parallelization is listed in Table 2. Table 2 also shows that the manual parallelization cannot analyze dynamic dependency.

Table 2: Design time of the manual parallelization.

	Design time (mins)		
Manual design tasks	**Ex. 1 (10 inst.)**	**Ex. 2 (10 inst.)**	**Ex. 3 (20 inst.)**
Analyze static dependency	6	7	16
Analyze dynamic dependency	Not aval.	Not aval.	Not aval.
ASAP	2	2	6
Constructive	3	2	17
Total	11	11	39

In contrast to 11/11/39 minutes required by the manual parallelization in the examples, the automatic parallelization took less than 3 seconds for each example, which is 220/220/780 times faster than the time for the manual parallelization. As the complexity of a design increases, designers can save more time by using the tools.

4.2 Results for 10 Instance Examples

Table 3: Results for 10 instance examples

	Ex1	**Ex2**	**Ex3**	**Ex4**	**Ex5**
Original					
Num. instantiation	8	10	10	10	10
Num. depedency	14	34	13	22	36
Length. CP	8	10	10	10	10
Constructive Algorithm					
Num. depedency	18	39	15	23	37
Length. CP	4	8	4	5	7
Added dependency (%)	28.57%	14.71%	15.38%	4.55%	2.78%
Reduced CP (%)	50.00%	20.00%	60.00%	50.00%	30.00%
ASAP Algorithm					
Num. Dependency	24	42	32	38	40
Length. CP	4	7	4	5	6
Added Dependency (%)	71.43%	23.53%	146.15%	72.73%	11.11%
Reduced CP (%).	50.00%	30.00%	60.00%	50.00%	40.00%

We randomly generated 5 behaviors, each of which contains no more than 10 instances. Table 3 shows the results of parallelization optimization for the examples. *Length. CP* represents the length of critical path. *Num. dependency* represents the number of static dependencies among instances of behavior. *Added dependency (%)* is equal to the difference between *Num. dependency*(Original) and *Num. dependency*(Constructive/ASAP algorithm) divided by *Num. dependency*(Original). *Reduced CP(%)* is equal to the difference between *Length. CP* (Original) and *Length. CP* (Constructive/ASAP algorithm) divided by *Length. CP*(Original).

Table 3 shows that the average *Added dependency* for the constructive algorithm is 13.2%, while for ASAP algorithm is 65.0%. Therefore, constructive algorithm is much better than ASAP algorithm in terms of goal (a) described in 2.4.2. On the other hand, the average *Reduced CP* for the constructive algorithm is 42%, while for the ASAP algorithm is 46%, both of which are similar. By considering the number of dependency as well as the length of the critical path, we conclude that the constructive algorithm is better for 10 instance behaviors.

4.3 Results for 20 Instance Examples

We generated another five examples shown in Table 4, each of which has no less than 20 instances. Ex6 and Ex9 do not have locality attribute, while Ex7, Ex8, and Ex10 have. For a behavior without locality attribute, the instance has the same probability of having dependent relations with any other instances. For a behavior with locality attribute, the instance has larger probability of having dependent relations with instances close to it than instances not close to it. The closeness between two instances is equal to the number of instances between them during the execution of original sequential behavior. Attribute locality exists in most designs. In this paper, when the closeness of two instances is no more than 4, we called them close instances. For Ex7, Ex8, and Ex10, each instance can depend on any close instances, but can depend on only one un-close instance.

Table 4: Results for 20-30 instance examples

	Ex. 6	Ex. 7	Ex. 8	Ex. 9	Ex. 10
Original					
Atrribute	random	locality	locality	random	locality
Num. Instantiation	20	20	21	30	30
Num. Dependency	98	95	79	193	109
Length. CP	20	20	21	30	30
Constructive Algorithm					
Num. Depedency	158	135	126	328	239
Length. CP	11	8	7	12	7
Added Dependency (%)	61.22%	42.11%	59.49%	69.95%	119.27%
Reduced CP (%)	45.00%	60.00%	66.67%	60.00%	76.67%
ASAP Algorithm					
Num. Depedency	169	173	185	400	375
Length. CP	7	8	7	10	7
Added Dependency (%)	72.45%	82.11%	134.18%	107.25%	244.04%
Reduced CP (%)	65.00%	60.00%	66.67%	66.67%	76.67%

Table 4 shows that the average *Added dependency* for the constructive algorithm is 70%, while for ASAP algorithm is 128%. Although *Added dependency* of constructive algorithm is much better than ASAP algorithm, they are much worse than the results for 10 instance behaviors. It is reasonable because later executed instances will add more dependencies than the previous ones. On the other hand, the average *Reduced CP* for the constructive algorithm is 61%, while for ASAP algorithm is 67%, both of which are similar. Obviously, *Reduced CPs* of 20 instance behaviors are greater than the results of 10 instance behaviors.

Furthermore, we do more research on example 7, 8, and 10 for behaviors with locality attribute. We find that the *Reduced CPs* of constructive algorithm and ASAP algorithm are the same for these examples. Because of this and the analysis on the constructive algorithm, we conclude that the probability of having similar length of critical path of the results of ASAP and constructive algorithm for behaviors with locality attribute is larger than the probability for behaviors without locality attribute. Therefore, constructive algorithm is more suitable for behaviors with locality attribute.

4.4 Real Project Examples

Table 5: Results for JPEG and Vocode Project Examples

	JPEG_Init (Jpeg)	Pre_ process (vocoder)	Ex_syn_ upd_sh (vocoder)	Lp_ Analysis (vocoder)
Original				
Num. Instantiation	3	3	5	9
Num. Dependency	2	2	8	30
Length. CP	3	3	5	9
Constructive Algorithm				
Num. Depedency	2	2	8	30
Length. CP	2	2	3	6
Added Dependency (%)	0.00%	0.00%	0.00%	0.00%
Reduced CP (%)	33.33%	33.33%	40.00%	33.33%
ASAP Algorithm				
Num. Depedency	2	2	8	32
Length. CP	2	2	3	6
Added Dependency (%)	0.00%	0.00%	0.00%	6.67%
Reduced CP (%)	33.33%	33.33%	40.00%	33.33%

We use the spec profiler and the spec optimizer on JPEG project[6] and Vocoder project[7]. Although designers have implemented the parallelization optimization manually for the projects, the tools still found parallelization instances existing in four sequential behaviors shown in Table 5. We updated the specification based on the produced hierarchical parallel structures and had the same simulation results with the original specifications. It proves that using the tools are more reliable than implementing parallelization optimization manually.

In addition to sequential behaviors shown in Table 5, the tools also found that sequential behavior *Coder_12k2* of Vocoder contained behavior instances executed in parallel. However, the simulation result of updated specification according to the tools is different from the simulation result of the original specification. The reason for this difference is that an address of a *Coder_12k2*'s port is assigned as a value to an address of another *Coder_12k2*'s port. Since the task *dependency analysis* could not treat read/write access of the second port as the read/write access of the first port, the tools produced a wrong result. To prevent this from happening, designers need to avoid address transfer between ports in the specification.

5 Conclusion

This paper introduces the parallelization optimization for specification tuning. Parallelization optimization is a critical optimization for design specification, which will be used for the *PE allocation* and *behavior binding*.

We introduce two tools for parallelization optimization. The spec profiler analyzes the static and dynamic dependencies among behavior instances. The spec optimizer produces the hierarchical parallel structure based on ASAP algorithm and a constructive algorithm.

In comparison to the manual parallelization, the automatic parallelization has three advantages.

First, it shortens the design time. The automatic parallelization is 200 times faster than the manual parallelization for 10-instance behaviors, 700 times faster for 20-instance behaviors. As the complexity of a design increases, the automatic parallelization can save more time.

Second, it generates required hierarchical parallel structures. ASAP algorithm produces the optimal structures in terms of the length of the critical path. Constructive algorithm produces the structures that have the similar length of the critical path as that of the ASAP algorithm and have the much smaller number of added dependencies among behavior instances than that of ASAP algorithm.

Third, it optimizes every possible parallelism in the design.

We also find that with the increase in the number of instances of behaviors, or with the loss of behavior's locality attribute, it is impossible to keep both the length of the critical path and the amount of the added dependencies to a minimum for generated structures. This is due to the nature of the problem rather than the limitation of the tools.

Reference

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

[2] D. Gajski, J. Zhu et al. "SpecC: Specification lanugaeg and Design methodology" Kluwer Academic Publishers, 2000

[3] A. Gerstlauer, R. Domer, et al. System Design: a practical guide of with SpecC. Kluwer Academic Publishers 2001

[4] www.systemc.org

[5] Lukai Cai, Dan Gajski, *Introduction of Design-Oriented Profiler of SpecC Language*, University of California, Irvine, Technical Report ICS-00-47, June 2001

[6] Lukai Cai, Junyu Peng et al. Design of a JPEG Encoding System, University of California, Irvine, Technical Report ICS-99-54, Nov. 1999.

[7] Andreas Gerstlauer, Shuqing Zhao et al. *Design of a GSM Vocoder using SpeccC Methodology,* University of California, Irvine, Technique report ICS-99-11, Feb 1999.