`

Introduction of Design-Oriented Profiler of SpecC Language

Lukai Cai, Daniel Gajski

Technical Report #02-07
Mar 2002

Center for Embedded Computer System
University of California
Irvine, CA 92697

(949)824-8059
{lcai, gajski}ics.uci.edu

Abstract

To start design from higher levels of abstraction and to make an architecture exploration decision at the early stage of design, designers must achieve the characteristics of specification on the higher levels of abstraction. Designers should also estimate the system performance at the higher levels of abstraction to ensure that the implemented system meets design constraints. In this report, SpecC profiler, a design-oriented profiler, accomplishes two tasks above, by evaluating the specification model of SpecC language.

Index

List of Figures

Introduction of Design-Oriented Profiler of SpecC Language

Lukai Cai, Daniel Gajski
Center for Embedded Computer System
University of California, Irvine
Irvine, CA 92697

Abstract

To start design from higher levels of abstraction and to make an architecture exploration decision at the early stage of design, designers must achieve the characteristics of specification on the higher levels of abstraction. Designers should also estimate the system performance at the higher levels of abstraction to ensure that the implemented system meets design constraints. In this report, SpecC profiler, a design-oriented profiler, accomplishes two tasks above, by evaluating the specification model of SpecC language.

1 Introduction

With the decrease of time to market and the increase of the complexity of design, the design industry has tried to start design from higher levels of abstraction. Using SpecC methodology [1][2], the design process will be smooth and efficient.

However, in the past, our design experiences on JPEG[5][6], GSM vocoder[7], and JBIG[8] projects show the difficulties of getting the satisfactory profiling results for the specification model of SpecC language, from existing profiling tools. The reason for the difficulties is that the existing profiling tools are *code-oriented*; the purpose of these tools is to find the bottleneck of the executed algorithm, thus to optimize the specification of algorithms. The characteristics of these profiling tools are listed below:

a) They are machine-dependent. For example, DSP 56600 instruction set simulator [9] only provides the cycle timing result of instructions for DSP 56600 processor. Similarly, the hierarchical profilers of Codewarrior[10] only analyzes the performance for Intel Pentium/486/AMD K6/AMD K7 processors.
b) They only analyze the performance of designs as a whole. They cannot provide characteristics of communication and computation independently, therefore limiting the usage for partitioning.
c) They only consider the sequential execution of functions, without considering the parallel execution.

Code_oriented profilers work well when evaluating the specification executed on the processor supported by profiling tools. However, because SoC design incorporates at least a programmable processor, an on-chip memory, and an accelerating function unit implemented in hardware [11], the *code-oriented profilers* cannot generate all the information required by SoC design. For example, functions of specification can be implemented in any selected processing elements (PEs). If the system components are not Intel Pentium/486/AMD K6/AMD K7 processors, then Codewarrior cannot be used. Furthermore, the traffic throughput between different PEs should be evaluated. The traffic influences not only the performance of system, but also the protocol selecting process. Finally, since PEs can be executed in parallel in SoC, the parallel execution between different functions should be specified.

Since existing profilers cannot provide enough information for SoC design, in the JBIG project, a manual approach of profiling was implemented[8]. It took 2 months for one person to generate the acceptable profiling results.

To speed up the design process, SpecC profiler, which is a *design-oriented profiler*, is developed. Compared with the *code-oriented profiler*, the *design-oriented profiler* provides sufficient profiling results for SoC design. Unlike *code-oriented profiler*, the purpose of *design-oriented profiler* is to help designers to find a good architecture exploration solution by evaluating the performance of specification and design system thus. For example, it can tell designers which function should be implemented in faster PE to achieve better system performance. Also, it can identify which two functions should be implemented in the same PE, to reduce the performance overhead for communication. Thus, with the help of SpecC profiler, designers can make architecture exploration decision more easily.

The characteristics of SpecC profilers are.

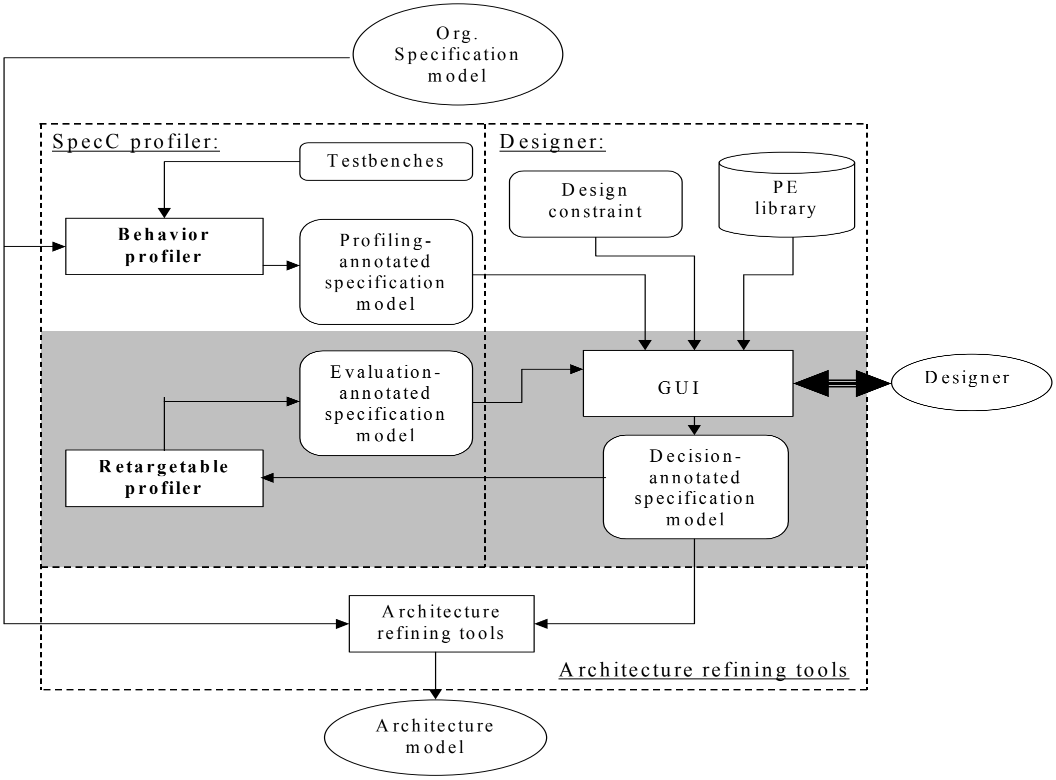

Org. Specification model
SpecC profiler:
Testbenches
Behavior profiler
Profiling-annotated specification model
Designer:
Design constraint
PE library
Evaluation-annotated specification model
GUI
Designer
Retargetable profiler
Decision-annotated specification model
Architecture refining tools
Architecture refining tools
Architecture model

Figure 1: Design flow of SpecC methodology of refining from *specification model* to *architecture model*

a) SpecC profiler is a retargetable profiler: it evaluates the characteristics of behaviors executed on any selected PEs. Moreover, it provides profiling results for the cases that different functions of specification are executed on different PEs.

b) SpecC profiler not only evaluates the characteristics of computation, but also evaluates the characteristics of traffic and the characteristics of memory for each function.

c) SpecC profiler evaluates the parallel execution among functions as well as evaluates the sequential execution.

SpecC profiler belongs to a set of tools refining the *specification model* into the *architecture model* of SpecC methodology [2].

Currently, SpecC profiler works on *the specification model*. SpecC profiler computes the profiling results based on the number of operations in specification. Therefore it is not really cycle-accurate. However, it is a good start of system design.

This report describes SpecC profiler. In section 2, the overview of SpecC profiler and the design flow of refining *the specification model* into *the architecture model* are introduced. Input and output models of SpecC profiler are introduced in section 3. Two main parts of SpecC profiler, *behavior profiler* and *retargetable profiler,* are described in section 4 and 5 respectively. Finally, Section 6 gives a conclusion.

2 SpecC Profiler

SpecC profiler works on *the specification model* of SpecC language [1][2]. *The specification model* is the model having the highest level of abstraction. It is an accurate model of the system in terms of pure functionality but does not reflect system structure or timing.

SpecC Profiler consists of two parts: *behavior profiler* and *retargetable profiler*. Figure 1 displays these two parts under SpecC methodology of refining *the specification model* to *the architecture model*.

At the beginning, designers specify *an original specification model* of SpecC and select executable testbenches. Then *behavior profiler* inserts statements into the *original specification model* to collect execution numbers of basic blocks in specification during simulation-time. After simulating selected testbenches, *behavior profiler* analyzes the specification based on generated execution numbers of basic blocks. *Behavior profiler* creates two sets of results: *behavior statistics*, and *behavior dependency* (In SpecC language, *behavior* is a class that encapsulates related functions and connects to other *behaviors* by its ports). For each behavior, *behavior statistics* contains the execution number of behavior, average execution number of operations per behavior execution, the size of needed memory, and the average traffic per behavior execution. *Behavior dependency* contains calling/called relations and sequence/parallel/parallel executing relations among behaviors. Since *behavior statistics* and *behavior dependency* are not related to the system architecture, they are implementation independent statistics.

Behavior profiler annotates *behavior statistics* and *behavior dependency* into *the original specification model* thus produces *the profiling-annotated specification,* which is then sent to GUI. GUI is a graphical user interface that helps designers to comprehend the profiling results by providing graphs of profiling results and to annotate design decisions into the specification automatically [4].

By comprehending *behavior statistics* and *behavior dependency,* designers then make architecture exploration decisions based on their design experiences. The architecture exploration decision making consists of following three steps:

a) Allocation. Designers select PEs from a PE library to assemble the system architecture.

b) Partitioning. Designers map behaviors in specification to the selected PEs.

c) Scheduling. Designers determine the executing relations (parallel/sequential/pipeline) among behaviors.

After designers make architecture exploration decisions, they use GUI to annotate the decisions into *profiled-annotated specification model*. After the annotation, *the decision-annotated specification model* is created.

Retargetable profiler reads *the decision-annotated specification model* as input and produces re-profiling statistics. The re-profiling statistics are implementation-dependent statistics based on designers' architecture exploration decisions. The re-profiling statistics include performances of behaviors, and memory sizes of behaviors. *Retargetable profiler* produces *the evaluation-annotation specification model* by annotating re-profiling statistics into *the decision-annotated specification model.*

GUI reads *the evaluation-annotation specification model* and displays the re-profiling statistics to designers. Designers then evaluate whether the performance of design based on the previous design decisions meets design constraints. If it does not meet the constraints, designers will make new architecture exploration decisions and reuse *retargetable profiler* to produce new re-profiling statistics. The processes of designers' decision making and re-profiling are continued until design constraints are met. This process is shown as the shaded part in Figure 1. As long as the final architecture exploration decisions are made, *the evaluation-annotation specification model* will be sent to the architecture-refining tool, which will refines the specification model into the corresponding architecture model automatically.

In general, with the help of SpecC profiler, designers control the design process and make the architecture exploration decisions based on their design experience. [14] describes guidelines of the design process using SpecC profiler and describes design examples of JPEG, JBIG, and Vocoder projects.

Besides the characteristics of SpecC profilers described in Section 1, SpecC profiler has two more advantages.

a) SpecC profiler analyzes *the behavior dependency*, which provides designers more flexibility of scheduling. This part will be illustrated in Section 4.

b) *Behavior profiler* and *retargetable profiler* are independent to each other. During the design process, *behavior profiler* and the design simulation are only executed once. On the other hand, *retargetable profiler* can be executed as many times as needed for different sets of architecture exploration decisions. Since the process of profiling and the specification simulation are slow, and the process of re-profiling is fast, this advantage enables more system alternatives explored during the architecture exploration than traditional methodologies.

3 Input and Output Models of SpecC Profiler

As shown in Figure 1, there are four specification models: *original specification model, profiling-annotated specification model, decision-annotated specification model*, and *evaluation-annotated specification model.* All the specification models are modeled in the format of.*SIR* file, which is the internal representation of SpecC model. Designers can transform *SIR* files to and from *.SC* files by using SpecC compiler [3].

Among these models, *original specification model* and *decision-annotated specification model* are input models of SpecC profiler. *Profiling-annotated model* and *evaluation-annotated specification model* are output models of SpecC profiler. The only differences among these models are annotations. *Original specification model* has no annotation; *profiling-annotated specification model* has annotations of *behavior dependency* and *behavior statistic; decision-annotated specification model* has annotations of architecture exploration decisions; *evaluation-annotated specification model* has annotations of re-profiling statistics.

As shown in Figure 1, GUI is a graphical user interface that helps designers to comprehend the profiling results by providing graphs of profiling results and to annotate design decisions into the specification automatically [4]. Besides using GUI, designers can also use SpecC profiler's command-line tools to produce textural files containing profiling statistics and to annotate architecture exploration decisions into specification.

4 Behavior Profiler

4.1 Design Flow of Behavior Profiler

Figure 2 shows the design flow *of behavior profiler*.

First of all, the task *instrument for profiling* analyzes *original specification model* and inserts statements into *original specification model* to compute execution numbers of basic blocks during simulation. It produces an internal model *instrumented specification model*, as shown in Figure 2.

By using SpecC compiler, *instrumented specification model* is compiled to an executable file. After designers simulate the executable file with selected testbenches, the instrument result is generated. The instrument result consists of two files, *_basicblock_counter.dpr* and *_funcmem_counter.dpr.* File *_basicblock_counter.dpr* contains the execution numbers of basic blocks. File *_funcmem_counter.dpr* contains the heap size of functions. Based on the instrument result, the task *"behavior statistics analysis*" and the task "*behavior dependency analysis*" analyze *original specification model* and produce *behavior statistics* and *behavior dependency*. Finally, the task "*behavior statistics annotation*" annotates the *behavior statistics* and *behavior dependency* into *original specification model.* This process will produce *profiling-annotated specification model.*

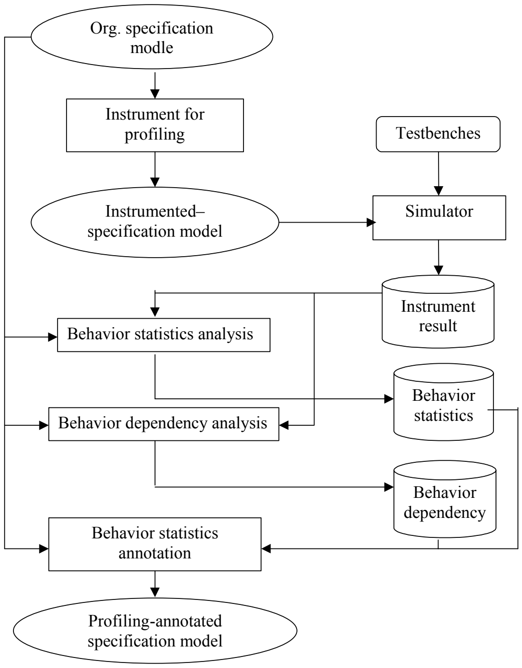

Org. specification modle
Instrument for profiling
Testbenches
Instrumented– specification model
Simulator
Instrument result
Behavior statistics analysis
Behavior statistics
Behavior dependency analysis
Behavior dependency
Behavior statistics annotation
Profiling-annotated specification model

Figure 2: Design flow of behavior profiler.

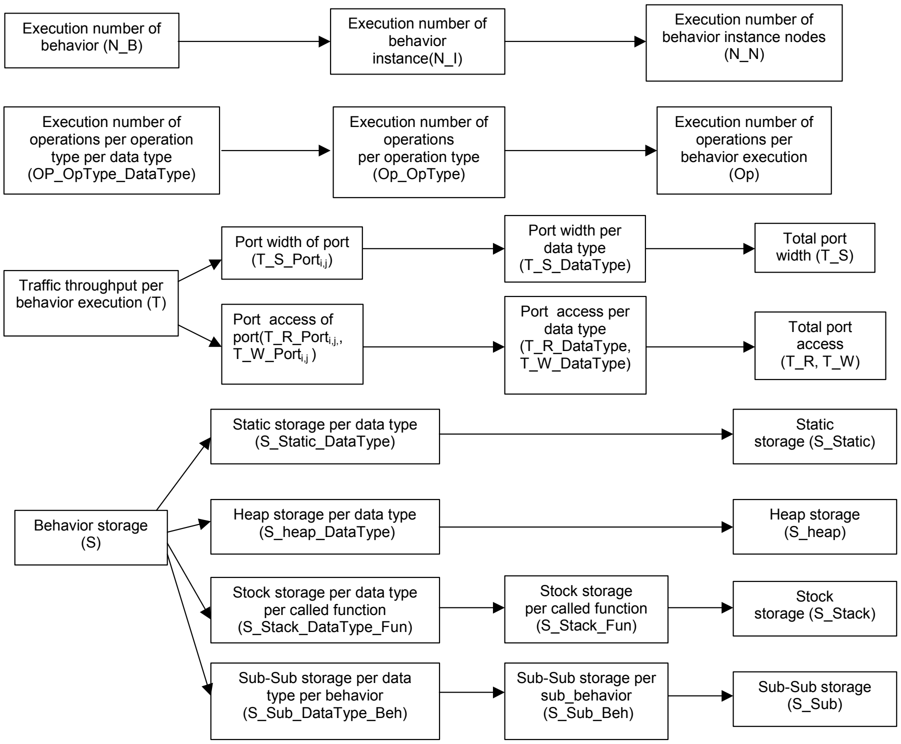

Execution number of behavior (N_B)
Execution number of behavior instance(N_I)
Execution number of behavior instance nodes (N_N)
Execution number of operations per operation type per data type (OP_OpType_DataType)
Execution number of operations per operation type (Op_OpType)
Execution number of operations per behavior execution (Op)
Traffic throughput per behavior execution (T)
Port width of port (T_S_Port$_{i,j}$)
Port width per data type (T_S_DataType)
Total port width (T_S)
Port access of port(T_R_Port$_{i,j,}$, T_W_Port$_{i,j}$)
Port access per data type (T_R_DataType, T_W_DataType)
Total port access (T_R, T_W)
Behavior storage (S)
Static storage per data type (S_Static_DataType)
Static storage (S_Static)
Heap storage per data type (S_heap_DataType)
Heap storage (S_heap)
Stock storage per data type per called function (S_Stack_DataType_Fun)
Stock storage per called function (S_Stack_Fun)
Stock storage (S_Stack)
Sub-Sub storage per data type per behavior (S_Sub_DataType_Beh)
Sub-Sub storage per sub_behavior (S_Sub_Beh)
Sub-Sub storage (S_Sub)

Figure 3: Behavior statistics

4.2 Behavior Statistics Description

For each behavior, *behavior profiler* produces four types of behavior statistics.

a) The execution number of the behavior.
b) The average execution number of operations per behavior execution
c) Average traffic throughput per behavior execution
d) Behavior storage.

The detailed information about statistics of above four types is displayed in Figure 3.

4.2.1 The Execution Number of the Behavior

Behavior profiler computes three types of execution numbers of behavior: the execution number of behavior, the execution number of behavior instance, and the execution number of behavior instance node.

First, *behavior* profiler computes the execution number of behavior, which refers to how many times a behavior is executed during simulation. It is used to compute the total execution number of operations of behaviors representing behavior complexity.

Second, *behavior profiler* computes the execution number of behavior instance. Each behavior can contain a number of behavior instances. Different behavior instances of a behavior can be mapped to different PEs during the architecture exploration. Therefore, designers can use the execution number of behavior instance to compute the total execution number of operations of each behavior instance on each PE.

Third, *behavior profiler* computes the execution number of behavior instance nodes. Behavior instance node can be defined in Figure 4. There are three behaviors in Figure 4(a): behavior X, A, and B. Behavior X contains behavior instance A1 and A2, both of which are the instantiations of behavior A. Similarly, behavior A contains behavior instance B1 and B2, both of which are the instantiations of behavior B. We use a hierarchical calling tree, called *behavior calling tree*, to illustrate these instantiations, as displayed in Figure 4(b). The tree contains seven nodes: X, A$_1$(X), A$_2$(X), B$_1$(X_A1), B$_2$(X_A1), B$_1$(X_A2), B$_2$(X_A2). For example, B$_1$(X_A1) represents the behavior instance B1 of behavior instance A1 of behavior instance X. We call nodes in the behavior calling tree as *behavior instance nodes.*

The execution number of behavior instance node is useful for the architecture exploration. As shown in Figure 4(c), designers map four *behavior instance node*s to different PEs. If we want to compute the total execution number of operations of *behavior instance node*s based on this partitioning, the execution number *of the behavior instance node* B_1(X_A1), B_2(X_A1), B_1(X_A2), B_2(X_A2) is needed.

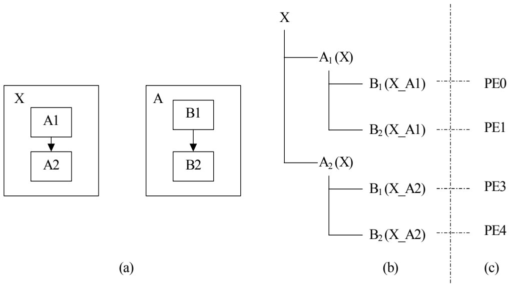


Figure 4: An example of behavior instance nodes

Behavior profiler computes three types of execution numbers of behavior by using execution numbers of combinatorial /basic blocks in the specification, which is generated by *instrument for profiling.*

a) The execution number of behavior (*N_B).*

For behavior i, N_B_i equals to the execution number of the combinatorial block in the specification representing behavior i's main function.

b) The execution number of behavior instance($N\_I_{i,j}$:).

If behavior instance j is instantiated in behavior i, $N\_I_{i,j}$ represents the average execution number of behavior instance j per execution of behavior i.

If N_B_i is not equal to zero, and if the basic block containing the calling statement of behavior instance j has executed X times, then

$$N\_I_{i,j} = X / N\_B_i\ ;$$

If N_B_i is equal to zero, $N\_I_{i,j} = 0$;

c) The execution number of *behavior instance node i*(N_N_i)

N_N_i represents the total execution number of *behavior instance node i*.

N_Ni can be computed as below:

i. For the behavior instance node representing Main behavior, $N\_N_{Main} = 1$;
ii. If the parent of behavior instance node A is behavior instance node B, then

$$N\_N_A = N\_N_B * N\_I_{B,A.}$$

4.2.2 Average Execution Number of Operations

The average execution number of operations per behavior execution is another essential behavior statistic. In SpecC language, there are 56 operation types and 29 data types. Therefore, the execution number of operations can be calculated hierarchically in three levels. If we use *OpType* to represent a chosen operation type and use *DataType* to represent a chosen data type, these three levels can be defined as:

a) $OP\_OpType\_DataType_i$:

It represents the average execution number of operations for the selected operation type *OpType* and for the selected data type DataType, per behavior i's execution.

Behavior profiler computes $OP\_OpType\_DataType_i$ according to the number of operations in each basic block and the total execution number of each basic block. The number of operations in each basic block per block execution can be directly derived from *original specification model.* It is denoted as $BB\_OpType\_DataType_{K,}$, for the operation type *OpType* and the data type DataType in the basic block k. The $OP\_OpType\_DataType_i$ can be computed by

$$OP\_OpType\_DataType_i = \Sigma_K(BB\_OpType\_DataType_k * (\text{The execution number of basic block k}))$$

b) OP_OpType_i:

It represents the average execution number of operations for selected operation type *OpType* and for all the data types, per behavior i's execution.

$$OP\_OpType_i = \Sigma_{DataType} OP\_OpType\_DataType_i.$$

c) OP_i:

It represents the average execution number of operations for all operation types and all data types, per behavior i's execution.

$$OP_i = \Sigma_{OpType}\ OP\_OpType_i.$$

Above equations are correct under the assumption that there are no behavior instances and function calls in behaviors. If a behavior contains behavior instances or function calls, then the execution numbers of operations of its behavior instances or called functions should be added to the execution number of operations of this behavior. The algorithm for this case is described in 4.4.1.

4.2.3 Average Traffic Throughput

The traffic (T) represents the statistics of behavior communication. SpecC profiler provides two types of traffic: port width and port access:

4.2.3.1 Port Width

For a behavior, the port width refers to the bit width of all the behavior ports. It consists of three level statistics:

a) $T\_S\_Port_{i,j}$ represents the bit width of port j of behavior i.

b) $T\_S\_DataType_{i:}$ represents the bit width of all the ports of data type *DataType,* of behavior i.

c) T_S_i represents the bit width of all the ports of behavior i.

$$T\_S_i = \Sigma_{DataType}\ T\_S\_DataType_i$$
$$= \Sigma_j\ T\_S\_Port_{i,j}$$

Behavior profiler computes port width by directly analyzing *original specification model.*

4.2.3.2 Port Access.

Port access refers to the average access number of behavior ports per behavior execution. The port accesses consist of read access and write access. For example, if x is a port of behavior, executing "x = x +x" once includes two read accesses and one write access.

Similar to port width, port access consists of three levels:

a) $T\_R\_Port_{i,j}$ represents the average number of read access for port j of behavior i, per behavior i's execution.

$T\_W\_Port_{i,j}$ represents the average number of write access for port j of behavior i, per behavior i's execution.

b) $T\_R\_DataType_i$ represents the read access of all the ports whose data type is *DataType,,* of behavior i, per behavior i's execution.

$T\_W\_DataType_i$ represents the write access of all the ports whose data type is *DataType,,* of behavior i, per behavior i's execution.

c) T_R_i represents the read access of all the ports of behavior i, per behavior i's execution.

T_W_i represents the write access of all the ports of behavior i, per behavior i's execution.

The approach of getting port access will be illustrated in 4.4.2.

Besides communicating through ports, behaviors also can communicate through their channels. The concept of channel is defined in [1]. A behavior uses a channel in terms of function calls. For example, if a behavior reads data from channel A and saves it into local variable b, then the behavior constains the statement *b = A.receive().* On the other hand, if a behavior writes a value of variable b into the channel A, then the behavior includes the statement *A.send(b).*

Behavior profiler computes the total number of argument access of channel functions as the read access. *Behavior profiler* computes the total execution number of channel functions having return value as the write access.

A limitation of traffic throughput computation is that traffic through ports of pointer types cannot be evaluated accurately.

4.2.4 Size of Behavior Storage

There are four types of storage: static storage, stack storage, heap storage, and sub_behavior storage.

4.2.4.1 Static Storage

For a behavior, the static storage consists of variables defined in the behavior but outside any functions, variables defined in the main function of behavior, and behavior's ports. When a behavior is executed, the static storage must be allocated and the size of static storage will not be changed during behavior execution.

Static storage consists of two levels:

a) $S\_Static\_DataType_i$ represents the total amount of the static storage for data type *DataType* of behavior i.

b) S_Static_i represents the total amount of the static storage of all the data types of behavior i.

$$S\_Static_i = \Sigma_{DataType}\ S\_Static\_DataType_i$$

For examples in Figure 5,

S_Static_{Parent} =sizeof(A) +sizeof(B) + sizeof(C) +sizeof(*D).

Behavior profiler computes static storage by directly analyzing *original specification model.*

4.2.4.2 Heap Storage

The heap storage refers to storage allocated by "malloc" statements and freed by “free” statements.

In this project, heap storage contains two levels:

a) $S\_Heap\_DataType_i$ represents the total amount of the heap storage for data type *DataType* of behavior i.

b) S_Heap_i represents the total amount of the heap storage of all the data types of behavior i.

$$S\_Heap_i = \Sigma_{DataType}\ S\_Heap\_DataType_i$$

For the example in Figure 5, S_Heap_{Parent} = sizeof(D).

Unlike static storage, *Behavior profiler* computes heap storage based on instrument result *_funcmem_counter.dpr* described in 4.1.

4.2.4.3 Stack Storage

The stack storage is the first hierarchical storage concerned. It refers to the storage used in called functions of behaviors. The storage of function contains function’s local variables, function’s stack storage and function’s parameters. Since all the functions are called sequentially in behavior, only currently called function storage is needed at a time during simulation. Therefore, we use the maximum storage of called functions as the behavior stack storage.

```
Behavior Sub_1( int K, int K2){
          void main(){
                    int M;
                    K = K2;
          }
};

Behavior Sub_2(int K3, int K4){
          void main(){
                    k4 = k3+1;
          }
};

Behavior Parent(int A){
          int B;
          Sub_1 Inst1(B, A);
          Sub_2 Inst2(A, B);
          Sub_3 Inst2(A, B);

          void F0(){
                    int X;
                    X=0;
          }
          void F1(int L){
                    int E;
                    F0();
                    E=L++;
          }
          void F2(){
                    int X;
                    X=0;
          }

          void main(){
                    int C, *D;
                    F1(C);
                    F2();
                    Inst1.main();
                    Inst2.main();
                    D = (int*) malloc (5);
          }
};
```

Figure 5: An example of specification model

The stack storage of behavior contains three levels:

a) $S\_Stack\_DataType\_Fun_{i,j}$ represents the total amount of storage for data type *DataType* of called function j of behavior i.

b) $S\_Stack\_Fun_{i,j}$ represents the total amount of the storage of called function j of behavior i.

$$S\_Stack\_Fun_{i,j} = \Sigma_{DataType}\ S\_Stack\_DataType\_Func_{i,j}$$

c) *S_Stack* represents the largest $S\_Stack\_Fun_i$ of all the called functions, of behavior i.

$$S\_Stack_i = Max_j(S\_Stack\_Fun_{i,j})$$

In the example in Figure 5,

$$\begin{aligned} S\_Stack_{parent} &= \text{Max } (\text{S\_Stack\_Fun}_{\text{parent, F1}}, \text{S\_Stack\_Fun}_{\text{parent, F2}}) \\ &= \text{S\_Stack\_Fun}_{\text{parent, F1}} \\ &= \text{sizeof(E)} + \text{sizeof(L)} + \text{S\_Stack\_Fun}_{\text{parent, F0}} \\ &= \text{sizeof(E)} + \text{sizeof(L)} + \text{sizeof(X)}. \end{aligned}$$

Behavior profiler computes stack storage by hierarchically analyzing *original specification model.*

4.2.4.4 Sub_Behavior Storage

Sub_behavior storage is the second hierarchical storage concerned. Unlike stack storage, two behavior instances can be executed in parallel. Thus sub_behavior storage is defined as the sum of the storage of its sub_behavior instances. The storage of sub_behavior instance contains instance's static, stack, heap, and sub_behavior storage.

Sub_behavior storage contains three levels:

a) $S\_Sub\_DataType\_Beh_{i,j}$ represents the total amount of the sub-behavior storage for data type *DataType* of sub_behavior instantiation j of behavior i.

b) $S\_Sub\_Beh_{i}$ represents the total amount of the sub-behavior storage of sub_behavior instantiation j of behavior i.

$$S\_Sub\_Beh_{i,j} = \Sigma_{DataType}\ S\_Sub\_DataType\_Beh_{i,j}$$

c) S_Sub_{i} represents the total amount of the sub-behavior storage of behavior i.

$$S\_Sub_{i} = Sum_{j}(S\_Sub\_Beh_{i,j})$$

For the example in Figure 5,

$$S\_Sub_{Parent} = \mathrm{S\_Sub\_Beh}_{\mathrm{Sub\_1}} + \mathrm{S\_Sub\_Beh}_{\mathrm{Sub\_2}} = sizeof\ (K) + sizeof(K2) + sizeof\ (K3) + sizeof\ (K4).$$

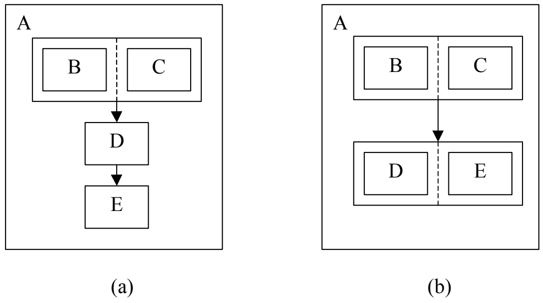
Similar to stack storage, *behavior profiler* computes sub_behavior storage by hierarchically analyzing *original specification model.*

4.3 Behavior Dependency Analysis

Besides the *behavior statistics*, relations of behaviors are also very useful for making design decisions. For example, if there is no traffic between behavior D and E as shown in Figure 6(a), then the behavior D and E can be executed in parallel instead of in sequential, as shown in Figure 6(b). It will improve the performance.

Behavior profiler computes two types of behavior dependencies: *calling dependency* and *data dependency*. *Calling dependency* analyzes the called/calling relations of behaviors; *data dependency* analyzes the traffic between behaviors.

Behavior profiler computes *behavior dependency* by hierarchically analyzing *original specification model* and by analyzing port access statistics.



(a)

(b)

Figure 6: An example of behavior dependency

4.3.1 Calling Dependency Analysis.

Calling dependency contains two parts:

(a) Parent behavior.
(b) Children behaviors and their execution relations.

For example, in Figure 6(a), Behavior A does not have any parent behavior. The children behaviors and their execution relations of behavior A can be described as (B || C) -> D -> E, while "B || C" represents parallel execution between behavior B and behavior C. "D -> E " represents D and E are executed sequentially and E is executed after D.

4.3.2 Data Dependency Analysis.

Data dependency represents the traffic between behavior instances. In the example in Figure 6, traffic between behavior instance B and C represents the data dependency of B and C.

The behavior instances are communicated through their ports. There are two ways to connect ports of behavior instances. First, the ports are connected through the global variables/channels defined in their parent behavior. For example, in Figure 5, variable B of behavior *Parent* connects behavior instance *Inst1* and behavior instance *Inst2*. We call these variables of parent behavior as *connected variables*. Second, the ports of behavior instances are connected through ports of their parent behavior. For example, in Figure 5, port A of behavior *Parent* connects behavior instance *Inst1* and behavior instance *Inst2*. We called these ports of parent behaviors as connected ports.

The port-to-port traffic is called based on following equations.

a) $T\_CV_{k,i,j}/T\_CP_{k,i,j}$

$$T\_CV_{k,i,j} = Max(T\_R\_Port_{i,\ Map(i,k)}\ , T\_W\_Port_{j,\ Map(j,k)}\ ) + Max(T\_W\_Port_{i,\ Map(i,k)}\ , T\_R\_Port_{j,\ Map(j,k)}\ );$$

$$T\_CP_{k,i,j} = Max(T\_R\_Port_{i,\ Map(i,k)}\ , T\_W\_Port_{i,\ Map(j,k)}\ ) + Max(T\_W\_Port_{i,\ Map(i,k)}\ , T\_R\_Port_{i,\ Map(j,k)}\ )$$

$T\_CV_{k,i,j}/T\_CP_{k,i,j}$ represents the total amount of traffic through connected variable/port k, between behavior instance i and j. Map(i, k) represents the port of behavior i that is mapped to connected variable/port k. For example, in Figure 5, for connected port A of behavior Parent, Map(Inst2, A) is port K3. *T_R_Port* and *T_W_Port* is the read and write access of port described in 4.2.3.2. For the traffic between behavior instance i and j through connected variable/ports, the read access of mapped port in i may not equal to the write access of mapped port in j. Therefore, the maximum of the read access of the mapped port in i and the write access of the mapped port in j is added with the maximum of the write access of the mapped port in j and the write access of the mapped port in i, to generate $T\_CV_k/T\_CP_{k.}$

For example, in behavior Parent of Figure 5,

$T\_CV_{A,Inst1,Inst2}$ = Max($T\_R\_Port_{Inst1,\ K2}$, $T\_W\_Port_{Inst2,\ K3}$) + Max($T\_W\_Port_{Inst,\ K2}$, $T\_R\_Port_{Inst2,\ K3}$).

b) $T\_CP_{\ i,j}/T\_CV_{\ i,j}$

$$T\_CP_{\ i,j} = \Sigma_k T\_CP_{k,i,j}$$
$$T\_CV_{\ i,j} = \Sigma_k T\_CV_{k,i,j}$$

$T\_CP_{\ i,j}/T\_CV_{\ i,j}$ represents the total amount of traffic for all the connected variables/ports between behavior instance i and behavior instance j.

c) $T\_BB_{i,j}$

$$T\_BB_{i,j}\ = T\_CP + T\_CV$$

$T\_BB_{i,j}$ represents the total amount of traffic between behavior instance i and j.

$T\_BB_{i,j}$ represents the data dependency between behavior instance i and j. If $T\_BB_{i,j}$ equals to 0, then the behavior instance i and j are data independent. Otherwise, they are data dependent. Furthermore, the closeness of two behavior instances is represented by the value of $T\_BB_{i,j}$

4.4 Two Algorithms in Behavior Profiler

In *behavior profiler,* several algorithms are applied to achieve *behavior statistics.* The complexity of implementing *behavior profiler* comes from the hierarchical analysis of specification. In this section, two algorithms are described. The first algorithm is for recursive function calls. The second one is for port access.

4.4.1 Algorithm for Recursive Function Calls

In 4.2.2, we introduce the computing equations for the average execution number of operations without considering function calls. In this section, the algorithm for computing the average execution number of operations considering function calls is described.

There are three types of functions in SpecC language. Local functions are the functions defined inside behaviors; global functions are the functions defined outside behaviors but in specification; library functions are the functions defined in libraries. The local functions of each behavior and can be called recursively. The global functions also can be called recursively.

The average execution number of operations of behavior equals to the average execution number of operations of the main function of behavior. Therefore, after the execution number of the main function of behavior is computed, we get the execution number of operations of behaviors.

We use local functions as our example to illustrate the method of solving recursive-calling problem. In this stage, we ignore library functions and assume the execution numbers of operations of global functions are already computed.

We computed each *OP_OpType_DataType* by using following algorithm. To add the execution number of operations of called functions into the execution number of operations of calling functions, the following equation is adopted [12]

$$A = C * A + O \qquad *$$

A is n-dimensional vector, while its item A_i is the average number of operations executed by the function i and n is the number of local functions in the behavior, including the main function. C is a square matrix, while $C_{i,j}$ denotes how many times function i calls function j. $C_{i,j}$ equals to the execution number of the basic block which contains calling statement of function j in function i divided by the execution number of function j. O is n-dimensional vector.

$$O_j = \Sigma_k((Op\_B_k + \Sigma_i Op\_Global\_Function\_i + \Sigma_l Op\_Sub_l) * BB_k) / N\_F_j$$

Op_B_k is the number of operations specified in basic block k of function j. Op_Global_Function is the average execution number of operations of called global functions in basic block k. Op_Sub is the average execution number of operations in behavior instances in basic block k. BB_k is execution number of basic block k of function j. N_F_j is the execution number of function j, which equals to the execution number of the combinatorial block that represents function j.

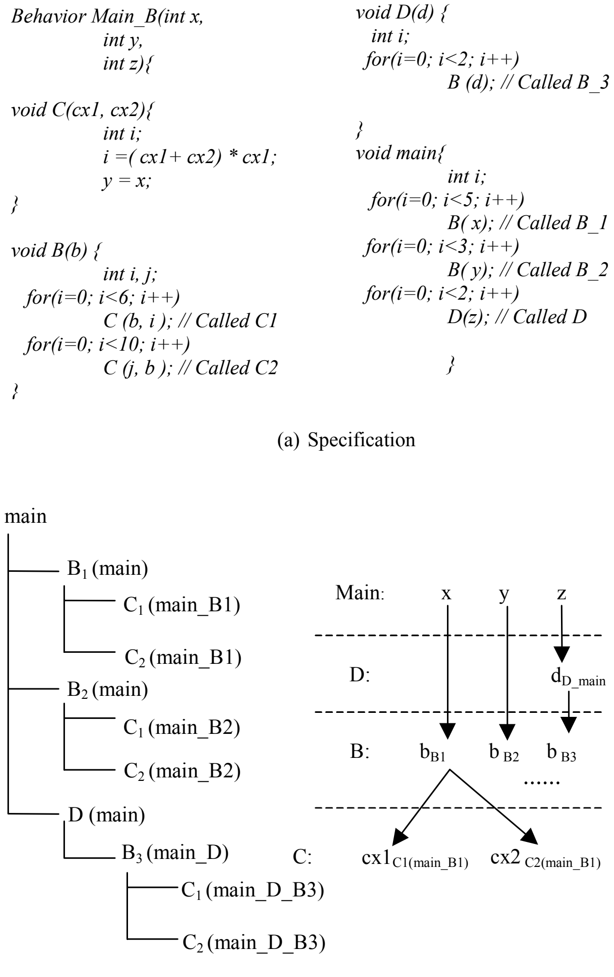
The only unknown variable in the system of liner equation * is the matrix A. Therefore, by solving the equation *, we achieves the average number of operations for all functions.

The same algorithm can solve the problems of recursive calling for traffic and memories.

4.4.2 Algorithm of Analyzing Port Access

When considering hierarchical calls, the port access of behavior consists of two parts: direct access from calling functions of behaviors, and indirect access from called functions and behavior instances.

As shown in Figure 7(a), the main function of behavior Main_B calls function B and D, while D calls function B and B calls function C. For argument cx1 of function C, there are two read accesses, per C's execution. For argument cx2 of function C, there is one read access, per C's execution. As shown in Figure 7(c), behavior Main_B's port x, y, and z are bound to argument cx1 and cx2 of called function C. Therefore, the argument access of cx1 and cx2 should be added to the port access of x, y, z. We call this types of port access *indirect port access*. On the other hand, the statement "x = y" in function C is called direct port access.



```
Behavior Main_B(int x,
            int y,
            int z){

void C(cx1, cx2){
            int i;
            i =( cx1+ cx2) * cx1;
            y = x;
}

void B(b) {
            int i, j;
  for(i=0; i<6; i++)
            C (b, i ); // Called C1
  for(i=0; i<10; i++)
            C (j, b ); // Called C2
}
```

```
void D(d) {
  int i;
 for(i=0; i<2; i++)
            B (d); // Called B_3

}
void main{
            int i;
 for(i=0; i<5; i++)
            B( x); // Called B_1
 for(i=0; i<3; i++)
            B( y); // Called B_2
 for(i=0; i<2; i++)
            D(z); // Called D

            }
```

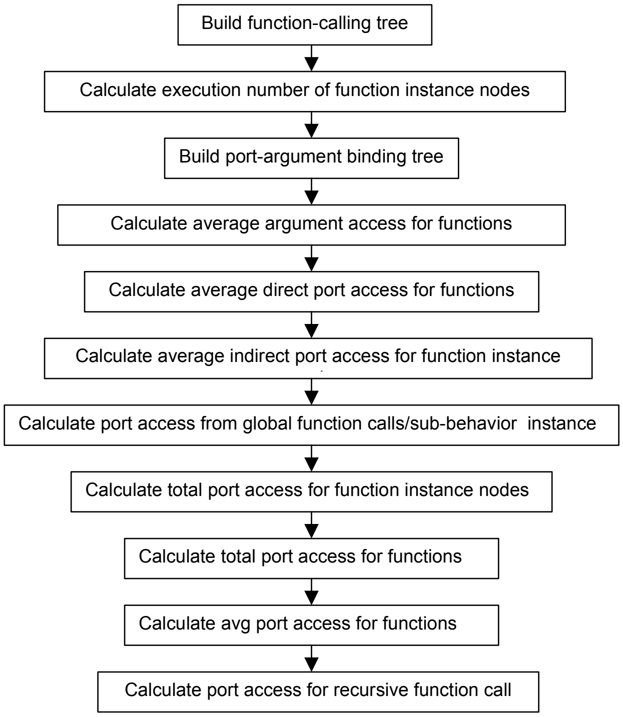
(a) Specification

(b) Function calling structure (c) Port-argument binding graph

Figure 7: An example for algorithm of port access.

Behavior profiler calculates the port access by following steps shown in Figure 8:

a) Build function-calling tree: similar to behavior calling tree in 4.2.1, *behavior profiler* builds the function-calling tree, as shown in Figure 7(b). The function-calling tree reflects the called and calling relation between function instance nodes.



Build function-calling tree
Calculate execution number of function instance nodes
Build port-argument binding tree
Calculate average argument access for functions
Calculate average direct port access for functions
Calculate average indirect port access for function instance
Calculate port access from global function calls/sub-behavior instance
Calculate total port access for function instance nodes
Calculate total port access for functions
Calculate avg port access for functions
Calculate port access for recursive function call

Figure 8: Design flow of analyzing port access

b) Calculate the execution number of function instance nodes in function-calling tree. Similar to behavior instance nodes in 4.2.1, if we define F_I (i,j) as the execution number of called function j per execution of calling function i, and if we define F_N(i) as the total execution number of function instance node i,

$$F\_N(i) = F\_N(j) * F\_I(i,j)$$

For the main function, F_N(main) = 1. F_I(i,j) equals to the execution number of the basic block that contains calling statement of function j divided by the execution number of function i.

For example, in Figure 9, the F_I(main, B1) = 5, F_I(B1, C1) = 6. Therefore, F_N(B1(main)) = F_N(main) * F_I(main, B1) = 5, F_N(C1(main_B1)) = F_N(B1(main)) * F_I(B1, C1) = 30.

c) Build port-argument binding tree for function instance nodes. The binding information is recorded in the tree if the argument of function instance node is bound to the port of behavior. For example, the port-argument binding tree of specification in Figure 7(a) is displayed in Figure 7(c). In this binding tree, the argument cx1 of function instance node C1(main_B1) is bound to port x of behavior.

d) Calculate average argument access for each function. In Figure 7, the average access for argument cx1 of function C is 2 read accesses.

e) Calculate the direct port access D_P(i,j), while i refers to function instance node i and j refers to port j.

$$D\_P(i,j) = \textit{port access}(j)\ \textit{per function execution} * F\_N(i)$$

In Figure 7, D_P(C1(Main_B1) ,x) = 1(read) * 30 = 30(read).

f) Calculate the indirect port access, based on port-argument binding tree and function-calling tree, for each port of behavior.

First, the total amount of argument accesses of function instance node D_A(i,k) is calculated. I refers to function instance node i and k refers to argument k.

$$D\_A(i,k) = \textit{argument access}(k)\ \textit{per function execution}* F\_N(i)$$

The indirect port access is:

$$I\_P(i, j) = \Sigma_k\, D\_A(i,k)$$

for all the argument k bound to port j. For example, I_P(C1(main_B1) , x) = D_A(C1(main_B1), cx1) = 2(read) * 30 = 60 (read), since argument cx1 of Main_B1_C1 is bound to port x.

g) Calculate the port access from its global function calls and behavior instances, for each port of behavior. Since the SpecC profiler computes the port accesses of child behaviors before computes the port accesses of parent behaviors, the port accesses of behavior instances are already known when computing the port accesses of parent behavior. The argument accesses of global function call are also computed before behavior port access calculation. After binding the ports of behavior to the ports of behavior instances or arguments of global functions, the port access from its global function calls and sub_behavior instances can be derived. We define it as G_P(i,j), for function instance node i and port j.

h) Calculate the total port j's accesses for function instance node i FIN_P(i,j)

$$FIN\_P(i,j) = D\_P(i,j) + I\_P(i,j) + G\_P(i,j).$$

i) Calculate the port accesses for each function, without considering local function calls.

$$F\_P(i,j) = \Sigma_k FIN\_P(k,j)$$

While k is the function instance node that have function type i. For example, in Figure 7,

F_P(C) = FIN_P(C1(main_B1)) + FIN_P(C2(main_B1)) + FIN_P(C1(main_B2)) + FIN_P(C2(main_B2)) + FIN_P(C1(main_D_B3)) + FIN_P(C2(main_D_B3)).

j) Calculate the average port accesses for each function, AF_P(i, j), without considering local function call.

$$AF\_P(i, j) = F\_P(i,j) / F\_F(i)$$

while F_F(i) is the execution number of function i.

k) Calculate the average port accesses for each function, AF_P(i, j), considering local function calls. Algorithm in 4.4.1 is used to solve this recursive problem.

In the C program, the arguments of data types such as "int" can only be read but not be written. On the other hand, arguments with pointer types can be read and written through pointer access. In *behavior profiler,* we have computed the argument accesses based on this fact.

5 Retargetable Profiler

5.1 Design Flow of Retargetable Profiler

The statistics generated by *behavior profiler* are architecture-independent. Allowing designers to estimate the system performance that reflects architecture exploration decisions, we designed the *retargetable profiler*.

Retargetable profiler performs the following tasks

a) After designers map behaviors to PEs, *retargetable profiler* produces the system performance of behaviors.

b) A number of behaviors will be mapped to one PE. All of the behaviors in each PE are executed sequentially. *Retargetable profiler* produces the system performance of PEs based on the statistics of behaviors.

c) The system architecture contains a set PEs. *Retargetable profiler* produces the statistics of the entire system based on the statistics of PEs.

SpecC profiler completely separates *behavior profiler* and *retargetable profiler. Retargetable profiler* only uses the output of *behavior profiler* including *behavior statistics* and *behavior dependency* as its input. It does not directly depend on the simulation. Therefore, the execution time of executing *retargetable profiler* is very fast. For example, the execution time of *behavior profiler* for the vocoder project [7] on Sun Ultra 5 workstation is 68 seconds; the testbench simulation time is 22 seconds; the execution time of *retargetable profiler* is 5 seconds. Since executing *retargetable profiler* computes the system performance, designers can completely change architecture exploration decisions and re-profile the design using *retargetable profiler* in very short time. Thus, more system architectures can be explored.

Figure 9 shows the design flow of *retargetable profiler*, which has been explained in section 1.

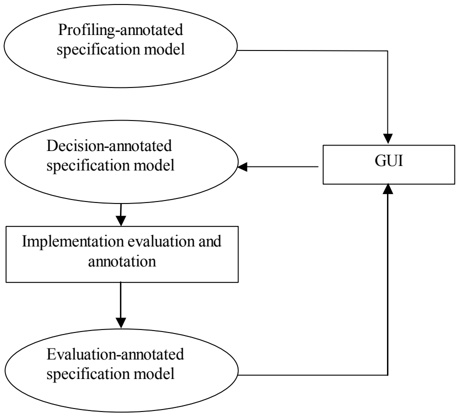

Profiling-annotated specification model
Decision-annotated specification model
GUI
Implementation evaluation and annotation
Evaluation-annotated specification model

Figure 9: Design flow of retargetable profiler

5.2 Weight Table Generation

Each weight table represents a PE. The items in weight table can be classified to three types: weight for operation, weight for traffic, and weight for memory. The weight for operation is an operation weight for certain operation type and certain data type representing the execution time of that type's operation on the PE. The weight for traffic is a traffic weight for certain data type representing the communication time of that type's data on the PE. The weight for memory is a memory weight for certain data type used representing the required memory size on the PE.

PE can be a custom ASIC, a programmable processor, or an IP. If a PE refers to IP, the IP provider should provide its weight table. If PE refers to custom ASIC, the weight table should be generated by analyzing the ASIC architecture.

If PE refers to a programmable processor, we can develop the weight table by referencing processor manuals and analyzing performance of sample codes [12].

For example, if the SpecC sample code is

```
{int c, a; c = a + 123;}.
```

After compiling on a selected PE, the target machine code is:

```
{MOVE a, R2;
MOVE #123, R1;
 ADD R1, R2;
Move R2, c}
```

Then we concluded that each integer identifier or constant contributes one MOVE instruction to the target code, each integer addition contributes one ADD instruction. There is no contribution from assignment.

The data in weight tables can be interpreted in terms of execution time, clock-cycles, number of bits/bytes, or other performance.

5.3 Output Statistics of Retargetable Profiler

The statistics of *retargetable profiler* contains computation performance for behavior instance nodes, communication performance for behavior instance nodes, and memory performance for behaviors.

5.3.1 Computation Performance

Retargetable-profiler computes average computation performance of behavior. The performance can be computed in terms of execution time, clock cycles of execution, or the number of instructions, depending on the meaning of PE weight table. The average computation performance of the behavior can be computed by adding up all of the weighted execution numbers of operations.

There are two types of behaviors, leaf behaviors and combinational behaviors. Leaf behaviors are the behaviors that do not have any behavior instances. Combinatorial behaviors are the behaviors that contains at least one behavior instance.

Retargetable profiler computes computation performance of these two types of behaviors separately according to different methods.

5.3.1.1 Leaf Behavior

The performance of each leaf behavior contains four levels:

a) $P\_OpType\_DataType_i = Op\_OpType\_DataType_i * Weight\_OpType\_DataType_i$
b) $P\_OpType_i = \Sigma_{Datatype} P\_OpType\_DataType_i$
c) $Avg\_P_i = \Sigma_{OpType} P\_OpType_i$
d) $Total\_P_i = Avg\_P_i * N\_N_i$

Op_OpType_DataType$_i$ is the average execution number for operation type *OpType* and data type *DataType*, for behavior i, which is described in 4.2.2.

The $Weight\_OpType\_DataType_i$ is the weight for operation type *OpType* and data type *DataType*, for the PE that behavior i is mapped to. N_N_i is the execution number of the behavior instance node i described in 4.2.1.

The four levels of performance counted are: $P\_OpType\_DataType_i$ represents the average computation performance for operation type *OpType* and data type *DataType* of the behavior i. P_OpType_i represents the average computation performance for operation type *OpType* and all data types of behavior i. Avg_P_i represents the average computation performance for all operation types and all data types of behavior i. $Total\_P_i$ represents the total computation performance of behavior instance node i.

5.3.1.2 Combinatorial Behavior

The performance of combinatorial behavior i can be calculated as follows:

```
if execution relations (sub_behavior) == sequential
then
   $Total\_P_i = \Sigma_{Sub\_beh}(Total\_P_{Sub\_beh})$;
else if  execution relations(sub_behavior) == parallel
then
            $Total\_P_i = Max_{Sub\_beh}(Total\_P_{Sub\_beh})$;
else if  execution relations(sub_behavior) == pipeline
then
            $Total\_P_i = Pipeline(Total\_P_{Sub\_beh})$
else if  execution relations(sub_behavior) == FSM
then
            $Total\_P_i = \Sigma_{Sub\_beh}(Total\_P_{Sub\_beh})$;
endif
```

$Total\_P_{Sub\_beh}$ represents the computation performance of sub_behavior instances, which has been calculated before calculating the computation performance of behavior i. Pipeline() is the function to calculate the computation performance when behavior instances are executed in a pipeline style.

SpecC profiler also supports the case when the execution relation of behavior instances of behaviors are a mix of sequential, parallel, pipeline, and FSM styles.

Although the computation performance of each PE is not explicitly displayed by behaviors, it is already considered by using weight tables for behaviors. The total computation performance of design can be represented by the performance of Main behavior.

5.3.2 Communication Performance

Retargetable profiler computes the communication performance for behaviors. The performance is computed in terms of the execution time, clock cycles of execution, or the number of instruction, depending on the meaning of PE weight table. Compared with port width and port access, the amount of traffic is weighted result and it is PE dependent. Based on port width and port access, *retargetable profiler* computes two types of statistics, static communication performance and dynamic communication performance.

5.3.2.1 Static Communication Performance

In some cases, when a behavior is executed, the data communication only happens twice: before the start of execution and after the end of execution. During the execution, the communicated data will be saved in the local memories. In these cases, the total amount of communication per behavior execution is called average static communication performance. If C_S represents average static computation performance, it can be calculated by equation

$$C\_S_i = 2 * \Sigma_{DataType} (T\_S\_DataType_i * Weight\_Traffic\_DataType)$$

$T\_S\_DataType_i$ is the port width defined in 4.2.3.1 and *Weight_Traffic_DataType* is the weight of computation for data type *DataType.*

5.3.2.2 Dynamic Communication Performance

If the data communication happens whenever ports are accessed, the average communication performance for each execution of behavior i is called average dynamic communication performance, which is represented by $C\_D\_W_i$ *and* $C\_D\_R_i$

$$C\_D\_W_i = \Sigma_{DataType} (T\_W\_DataType_i * Weight\_Traffic\_DataType)$$

$$C\_D\_R_i = \Sigma_{DataType} (T\_R\_DataType_i * Weight\_Traffic\_DataType)$$

while $T\_W\_DataType_i$ and $T\_R\_DataType_i$ are write access and read access defined in 4.2.3.2.

5.3.2.3 Total Communication Performance

Designers can computes the total communication performance of behavior instance node i using average static communication performance and average dynamic communication performance. For example, if the data communication only happens twice during behavior execution as discussed in 5.3.2.1, then the total communication performance is:

$$C_i = C\_S_i * N\_N_{i.}$$

While N_N$_i$ is the total execution number of behavior instance node i in 4.2.1

On the other hand, if the data communication happens whenever ports are access for each behavior, such as in 5.3.2.2, then the total communication performance is:

$$C_i = (C\_D\_W_i + C\_D\_R_i) * N\_N_{i.}$$

5.3.2.4 Communication Performance between Behaviors and PEs

Besides the communication performance for behavior, *retargetable profiler* also computes the average communication performance between two behaviors based on *behavior dependency* and traffic of each behavior.

For example, if behavior D and E are executed sequentially as shown in Figure 6(a), and D and E communicate through connected ports, then the traffic will exist only when writing D and reading E. The communication performance C$_{D,\ E}$ is,

If D, E share a PE, there is no traffic.
If D, E communicate through a global memory:

$$C_{D,,E,} =N\_N_D*T\_S_{D,} +N\_N_E *T\_R_{E,}$$

If D, E communicate through a local memory and
The local memory is in D:

$$C_{D,,E,} =N\_N_E*T\_R_E$$

If the local memory is in E:

$$C_{D,,E,} = N\_N_D*T\_S_D$$

Retargetable profiler only provides the statics and dynamic communication performance between behaviors. Designers need to compute communication performance between behaviors in case by case.

Designers can compute the communication performance between PEs. The communication performance between PE1 and PE2 is equal to the sum of communication performance between any behavior instance node in PE1 and any behavior instance node in PE2.

5.3.3 Memory Performance

Memory performance refers to the required memory size of selected PE.

a) Static memory (M_Static$_i$)

$$M\_Static_i = \Sigma_{DataType} (S\_Static\_DataType_i * Weight\_Memory\_DataType)$$

S_Static_DataType$_i$ is described in 4.2.4.1 and *Weight_Memory_DataType* is the weight of memory for data type *DataType* based on selected PE.

b) Heap memory (M_Heap $_i$)

$$M\_Heap_{\ i} = \Sigma_{DataType} (S\_Heap\_DataType_i * Weight\_Memory\_DataType)$$

S_Heap_DataType$_i$ is described in 4.2.4.2.

c) Stack memory (M_Stack $_i$)

$$M\_Stack_{\ i} = \Sigma_{DataType} (S\_Stack\_DataType_i * Weight\_Memory\_DataType)$$

S_Stack_DataType$_i$ is described in 4.2.4.3

d) Sub_behavior memory (M_Sub $_i$)

$$M\_Sub_{\ i} = \Sigma_{DataType} (S\_Sub\_DataType_i * Weight\_Memory\_DataType)$$

S_Sub_DataType$_i$ is described in 4.2.4.4.

e) The total memory of behavior (M $_i$)

$$M_{\ i} = M\_Static_i +M\_Heap_{\ i} + M\_Stack_{\ i} + M\_Sub_{\ i}$$

f) The total memory of PE (M_P$_k$)

$$M\_P_k = Max(M_{\ i})$$

The required memory size of selected PE is equal to the maximum of memory size of behaviors mapped to it. Designers can compute it manually.

6 Conclusion

In this report, SpecC profiler is introduced. SpecC profilers contain two parts: *behavior profiler* and *retargetable profiler*. *Behavior profiler* produces the statistics of specification that is independent of design implementation. *Retargetable profiler* produces the statistics of design depending on the selected target architecture and designers' architecture exploration decisions.

SpecC profiler has following advantages.

a) SpecC profiler is retargetable profiler: it evaluates the characteristics of behavior executed on any selected PEs. Moreover, it provides the profiling result for the case that different functions of specification are executed on different PEs.

b) SpecC profiler evaluates not only computation performance, but also communication performance, and needed memory size for the specification.

c) SpecC profiler evaluates sequential execution among functions as well as parallel and pipeline execution.

d) SpecC profiler analyzes the behavior dependency, which provides designers more flexibility of scheduling.

e) Two parts of SpecC profiler, *behavior profiler* and *the retargetable profiler,* are independent. Therefore, design simulation is only executed once. It makes the process of executing *retargetable profiler* fast and makes the large range of architecture exploration possible.

This report concentrates on introducing SpecC profiler and the statistics that SpecC profiler produce. In report [14], how to use SpecC profiler to perform system level design as well as design examples is introduced.

The SpecC profiler works on the specification model of SpecC language. The evaluation result is not really cycle-accurate. However, it is first step to limit the range of architecture exploration and it outlines the profiling process on higher levels of abstraction in SoC design.

Reference:

[1] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers

[2] Andreas Gerstlauer, Rainer Doemer , J. Peng, D Gajski, System design: a practical guide of SpecC, Kluwer Academic Publishers.

[3] Rainer Doemer , SpecC SIR Internal representation, CECS Internal report IR99-03, June 1999

[4] David Berner, Development of a Visual Refinement- and Exploration-Tool for SpecC, Technical Report ICS-01-12 2000

[5] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao and D. Gajski, "Design of a JPEG Encoding System," UC Irvine, Technical Report ICS-TR-99-54, November 1999.

[6] Hanyu Yin, Haito Du, Tzu-Chia Lee, Daniel D. Gajski, "Design of a JPEG Encoder using SpecC Methodology," UC Irvine, Technical Report ICS-TR-00-23, July, 2000.

[7] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski and Arkady M. Horak, "Design of a GSM Vocoder using SpecC Methodology," UC Irvine, Technical Report ICS-TR-99-11, March 1999.

[8] Junyu Peng, Lukai Cai, Anand Selka, Daniel D. Gajski, "Design of a JBIG Encoder using SpecC Methodology," UC Irvine, Technical Report ICS-TR-00-13, June, 2000.

[9] DSP56600 16-bit Digital Signal Processor Family Manual, Motorola Inc.

[10] Rick Grehan, Code Profilers: Choosing a Tool for analyzing performance, A Metrowerks white paper

[11] Chang, Cooke, Hunt, Surviving of SoC Revolution, A guide to platform-Based Design, Kluwer Academic Publishers.

[12] Sinisa Srbljic, Mario Stefanec, Ivan Benc SpecC Profiler: Specification-level Exploration Tool

[13] D. Gajski, J. Peir Essential Issues in Multiprocessor Systems 1985 IEEE

[14] L. Cai, Dan Gajski, System Level Design Using SpecC Profiler, Technical Report , Jan, 2001