Introduction of design-oriented profiler of SpecC language

Technical Report ICS– 00-47
June 2001

Lukai Cai, Dan Gajski
Information and Computer Science
University of California, Irvine
Irvine, CA 92697

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

Index

List of Figures

Introduction of design-oriented profiler of SpecC language

Abstract

To design from higher level of abstraction andto make architecture exploration decision at early stage, designer should know the characteristics of specification on the higher level of abstraction. Designers should also have a way to evaluate the system in early stage, to ensure that the system meets the constraint requirement. In this report, SpecC profiler, a design-oriented profiler, is introduced to complete above two tasks, by evaluating the specification model of SpecC language.

Index Terms-- behavior profiler, retargetable profiler, specification model, design-oriented, SpecC

1 Introduction

With the requirement of time to market and the increase of complexity of design, the design industry has tried to make the design decisions in earlier stage of design flow. 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] system show the difficulty to get the satisfied profiling result, from existing profiling tools, for the specification model of SpecC language. The reason is that the existing profiling tools are *code-oriented*: the purposes of existing profiling tools are to find the bottleneck of the executed algorithm, thus to optimize specification of algorithms. The characteristics of these profiling tools are:

a) They are machine-limited: For example, DSP 56600 instruction set simulator [9] only provides cycle timing result of instruction for DSP 56000 processor. Similarly, Hierarchical profilers of Codewarrior[10] only analyzes performance for Intel Pentium/484/AMD K6/AMD K7 processors.
b) They can only analyze performance of the design. Traffic and needed size of memory of system components are not concerned.
c) They only consider sequential execution of functions, without considering the parallel execution.

The *code_oriented profilers* work well in case of evaluating the specification executed on the processor that the profiling tools support. However, because SoC design incorporates at least a programmable processor, on chip memory, and accelerating function units implemented in hardware [11], the need of SoC design is more complex than that *code-oriented profilers* can generate. For example, functions of specification can be implemented in any selected PEs. If the system components (PEs: processing elements) are not Intel Pentium/484/AMD K6/AMD K7 processors, Codewarrior can not be used. Furthermore, The traffic between different PEs also should be evaluated. The traffic does not only influence the performance of system, but also influence the protocol selection. Finally, since PEs can be executed in parallel in SoC, the parallel execution between different functions should be considered.

Since the existing profilers cannot help us to implement SoC design, in the JBIG project, a manual approach of profiling were implemented. It took 2 months one person to generate acceptable profiling result.

To fasten the design process, SpecC profiler, which is a *design-oriented profiler*, is developed. Compared with the *code-oriented profiler*, the *design-oriented profiler* provides needed profiling result for SoC design. Unlike *code-oriented profiler*, the purpose of *design-oriented profiler* is to help designers to evaluate specification and design system thus to find good architecture exploration solution. For example, it can tell designers that which function should be implemented in faster PE to improve the performance of system. Also, it can identify that two functions should be implemented in 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.

a) SpecC profiler is retargetable profiler: it can evaluate the characteristics of behavior that is executing on any selected PEs. Moreover, it can provide profiling result for the case that different

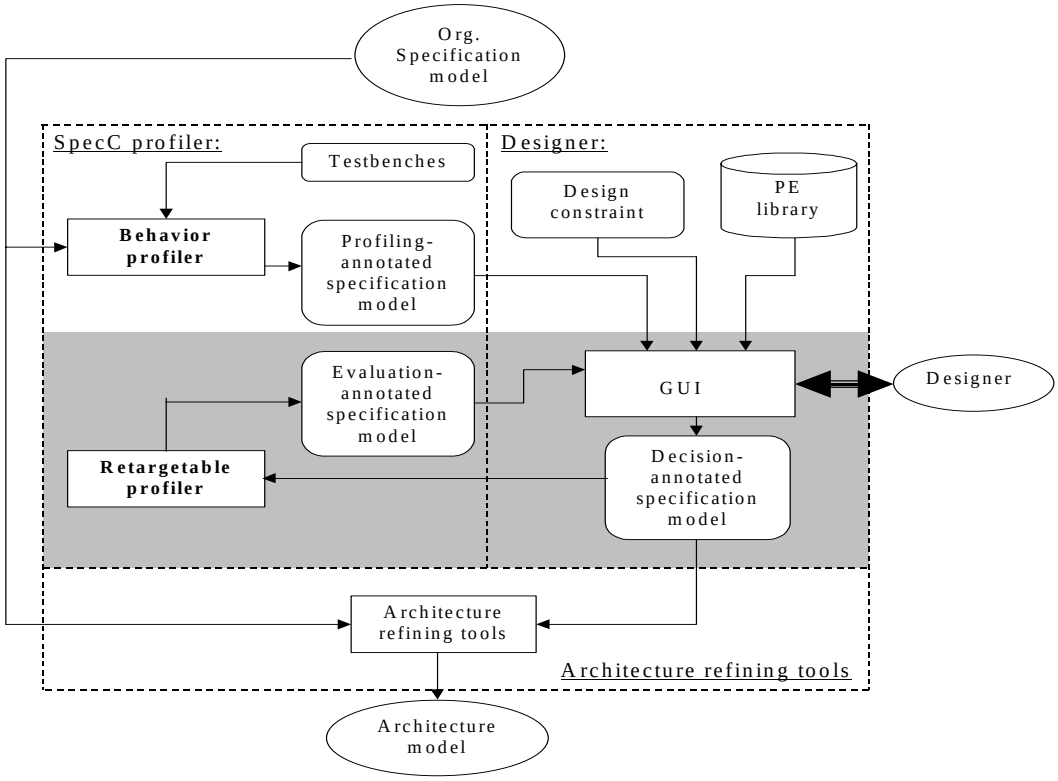


Figure 3: Design flow of SpecC methodology of refining from specification model to architecture model

functions of specification are executed on different PEs.

b) SpecC profiler not only evaluates performance, but also evaluate the traffic, needed memory size, for each function.
c) SpecC profiler evaluates parallel execution among functions as well as sequential execution.

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

SpecC profiler works on the specification model of SpecC language. The profiling results are based on the number of operations of specification, therefore it is not real cycle-accurate. However, it is good enough to give designers a first look of system of design.

This report describes SpecC profiler. In section 2, the overview of SpecC profiler as well as the design flow of refining from specification model into architecture model, are illustrated. The input and output models of the 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. In section 6, a simple methodology and a example are given to teach designers how to use SpecC profiler. Finally, a conclusion is made in section 7.

2 SpecC profiler

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

SpecC Profiler consists of two parts: *behavior profiler* and *retargetable profiler*. In Figure1, these two parts are illustrated in the environment of SpecC methodology, for refining the specification model into the architecture model .

At the initial stage of design, as soon as *original specification model* is semantically and syntactically correct and an executable testbench is selected, *behavior profiler* can be performed.

At the beginning, the first part of SpecC profiler, *behavior profiler*, inserts statements into the original specification model, to collect execution number of basic blocks of specification, at simulation-time. After simulating with selected testbenches, *behavior profiler* analyzes the specification, based on the execution numbers of basic blocks that are generated during simulation. *Behavior profiler* creates two 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). *Behavior statistics* consists of the statistics of behaviors, including execution number of behaviors, average execution number of operations per behavior execution, the size of needed memory of behavior, and the average traffic of behavior, per behavior execution. *Behavior dependency* describes the executing relation between behaviors, including calling/called relations and sequence/parallel execution relations. *Behavior statistics* and *behavior dependency* are not related to system architecture, therefore, it is implementation-independent.

Behavior profiler annotates *behavior statistics* and *behavior dependency* into *original specification model*. *Profiling-annotated specification model* is produced and sent to GUI. GUI is a graphical user interface that helps designers to explicitly and automatically read and write different specification models [4].

With the help of *behavior statistics* and *behavior dependency*, designers should make the architecture exploration decision, based on their design experience. Architecture exploration consists of following three items:

a) Allocation: Needed PEs are selected from PE library, for assembling the system.
b) Partitioning: Behaviors of specifiction are mapped to selected PEs.
c) Scheduling: Whether the execution relations among behaviors are parallel or sequential are decided.

The architecture exploration decision will be annotated into *profiled-annotated specification model* by GUI, thus *decision-annotated specification model* is created.

With *decision-annotated specification model* as input, the second part of the SpecC profiler, *retargetable profiler*, is used to re-profile specification. In this stage, the re-profiling results are implementation-dependent, based on designers' architecture exploration decision. The re-profiling result includes performance of behaviors, size of memory of each behavior, and the amount of traffics and the traffic time between different behaviors. The result of *retargetable-profiler* will be annotated into *decision-annotated specification model*. The new model, *evaluation-annotation specification model*, is created and displayed to designers by GUI.

After re-profiling, designers will evaluate whether the current design will meet the constraint requirement, based on *evaluation-annotation specification*. If the current implementation cannot meet the requirement, a new architecture exploration decision will be made and again annotated into *profiled-annotated specification model*. The process of designers' decision making and re-profiling is continued until the constraint requirement of design is satisfied by system, as shown in the shaded part of Figure1. As soon as the final architecture exploration decision is made, *evaluation-annotation specification model* will be sent to the architecture-refining tool. The architecture-refining tool refines the specification model into the corresponding architecture model automatically.

In general, designers control the design process and make the architecture exploration decision, based on their design experience. To help beginners to make correct architecture exploration decision, a simple design methodology of how to make architecture exploration decision with profiling result is described in Section 6 in great detail.

Besides the characteristics of SpecC profilers described in Section 1, SpecC profiler has two more advantages. First, it analyzes the behavior dependency, which provides designers more flexibility of scheduling. This part will be illustrated in Section 4. Second, since *the behavior profiler* and *the retargetable profiler* are separated, design simulation is only executed once during design process, by *behavior profiler*. The *retargetable profiler* can be executed as many times as needed, based on the result of behavior profiler, without specification simulation. Since the specification simulation and execution of *behavior profiler* are slow, while process of re-profiling is fast, this advantage makes more system alternative can be explored during the architecture exploration, than the traditional methodology.

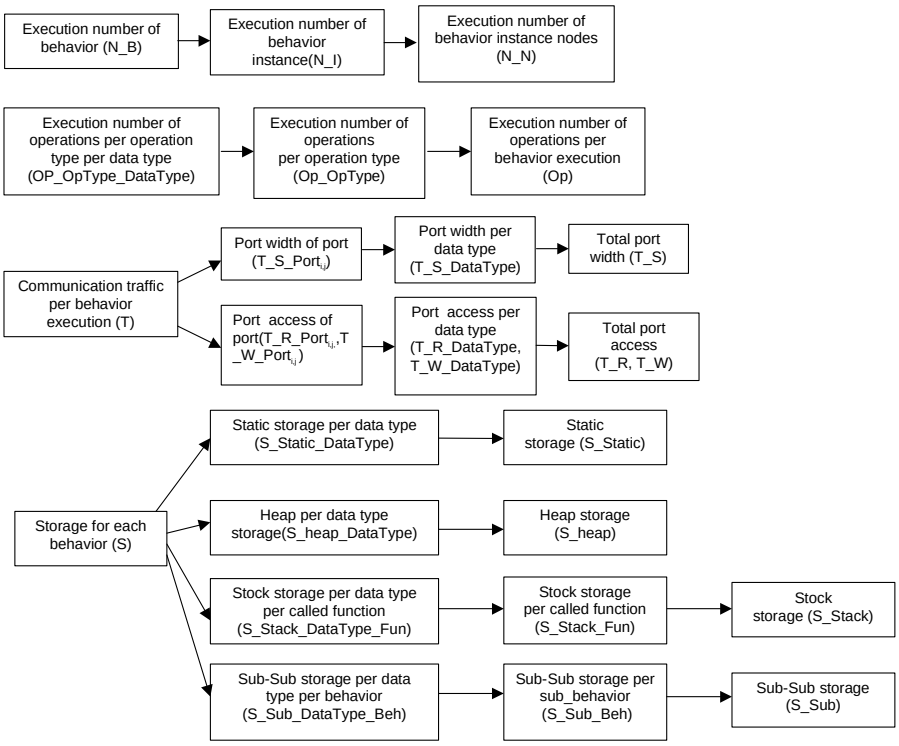


Figure 4: Display of behavior statistics

3 Input and output models of SpecC profiler

As shown in Figure1, there are four specification models: *original specification model, profiling-annotated specification model, decision-annotated specification model*, and *evaluation-annotated specification model.* All of the specification models are described in the format of the *.SIR* file. The *.SIR* file is the internal representation of SpecC model. It can be transformed to and from *.SC* file by SpecC compiler [3].

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

As shown in Figure1, GUI should be developed to help designers to read and write specification models automatically and explicitly. To make SpecC profiler an independent tool, we also provide a way for designer to read and write model by simply reading and writing texture files. This part of work is described in the profiler manual.

4 Behavior profiler

4.1 Design flow of the behavior profiler

Figure2 illustrates the design flow of the behavior profiler.

First of all, task “instrument for profiling” decomposes the original specification model into combinatorial and basic blocks, and inserts statements for counting the execution number of blocks into original specification model. After statement insertion, instrumented specification model, which is an internal model, is developed, as shown in Figure 2.

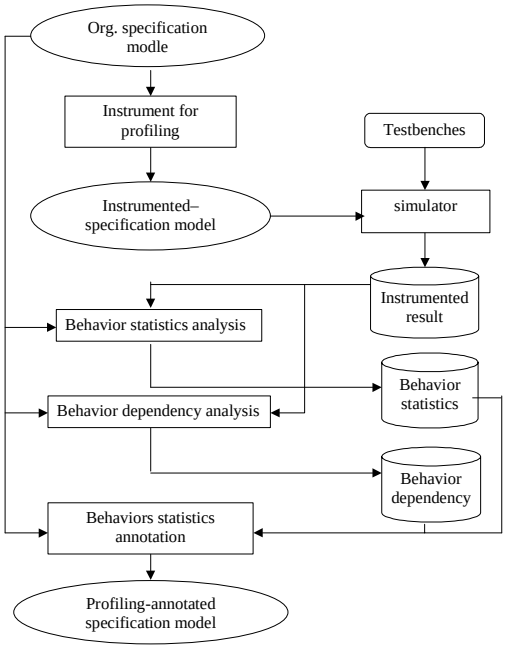


Figure 5: Design flow of behavior profiler

The *instrumented specification model* is then compiled to a executable file by SpecC compiler. After the executable file is simulated with selected testbench, two output files are created: *_BB_Counter file* contains the number of execution of basic blocks while *_Heap_Counter file* contains the heap size of functions. Based on *_BB_Counter file* and *_Heap_Counter file*, 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 "*behaviors statistics annotation*" annotates the *behavior statistics* and *behavior dependency* into *original specification model*. This process will produce *profiling-annotated specification model*.

4.2 Behavior statistics description

The behavior statistics represents the statistics of each behavior based on the simulation with the selected testbenches. The behavior statistics consists of four types: execution numbers of behavior, average execution numbers of operations per behavior execution, average traffics per behavior execution, and needed behavior memory. These statistics are listed in Figure 3 hierarchically.

4.2.1 Total execution number of behavior

There are several execution numbers of behavior entities should be counted, based on the need of system evaluation.

First of all, the total execution number of behaviors should be counted. Furthermore, since each behavior can have a number of behavior instances and the behavior instances can be mapped into different PEs during architecture exploration, *behavior profiler* also provides the execution number of behavior instances. During research, we found that even the execution number of behavior instances cannot provide enough information for system evaluation. For example, in Figure 4(a), there are three behaviors: X, A, B. Behavior X contains behavior instances A1 and A2, which are both the instantiation of behavior A. Similarly, behavior A contains behaviors instances B1 and B2, which are both the instantiation of behavior B. We use a hierarchical calling tree, called *behavior calling tree*, to displayed this relation, as 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, $B_1(X\_A1)$ represent the behavior instance B1 of behavior instance A1 of behavior instance X. We call the node in behavior calling tree *behavior instance node*.

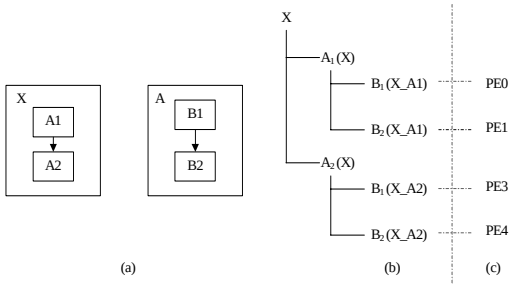


Figure 6: Example of behavior instance nodes

In Figure4(c), four leaf *behavior instance node*s are mapped to different PEs. Therefore, if we want to analyze the system performance based on this partitioning, the execution number *of behavior instance nodes* is needed.

Three types of execution numbers of behavior entities are calculated as follows:

a) *N_B:* Total execution number of behavior:

N_B_i refers to the execution number of behavior i. N_B_i equals to the execution number of combinatorial block that represents behavior i's main function.

b) $N\_I_{i,j}$*:* Average execution number of behavior instance:

If behavior instance j is declared in behavior i, it refers to the average execution number of behavior instance j per execution of behavior i.

If N_B_i is not zero, assume that the basic block which contains behavior instance j has executed X times,

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

Otherwise $N\_I_{i,j} = 0$;

c) N_N_i*:* Total execution number of *behavior instance node*:

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

N_Ni can be calculated as:

i. The top behavior instance node is Main behavior of specification. $N\_N_{Main} = 1$;
ii. If behavior instance node A 's parent is behavior instance node B,

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

4.2.2 Average execution number of operations.

Average execution number of operations per behavior execution is another essential behavior statistic, which will be used for evaluating performance of design. 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 of operation type *OpType* for data type DataType, per behavior i's execution. Since the number of operations in each basic block per block execution is fixed, this number can be derive from *original specification model.* If we write this number as $BB\_OpType\_DataType_{K,}$ for operation type *OpType* and data type DataType, in basic block k,

$$OP\_OpType\_DataType\_i = \Sigma_K(BB\_OpType\_DataType_k * (Execution\ number\ of\ Basic\ block\ K))$$

b) *OP_OpType_i:* it represents the average execution number of operations of operation type *OpType* for all the data type, 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 of all operation types for all data types, per behavior i's execution.

$$Op\_i = \Sigma_{OpType}\ OP\_OpType\_i.$$

About execution number of operations are calculated without considering function calls. If the a behavior has behavior instances or function calls, the execution numbers of operations of its behavior instance or called function should be added to the execution number of operations of the behavior. The algorithm for this case is explained in 4.4.1.

4.2.3 Traffic

The traffic (T) represents the communication throughput of behavior. SpecC profiler provides two types of traffic as follows:

4.2.3.1 Port width

The port width refers to the bit number of behavior ports. It consists of three levels:

a) $T\_S\_Port_{i,j}$ represents the bit number of port j of behavior i.
b) $T\_S\_DataType_{i:}$ represents the bit number of all the ports which have data type *DataType,* of behavior i.
c) T_S_i represents the bit number of all the ports of behavior i.

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

Port width can be directly analyzed from *original specification model.*

4.2.3.2 Port Access.

Port access refers to the average access number of each behavior's port per behavior execution. The port access includes read access and write access. . For example, if x is behavior's port, executing "x = x +x" once creates two read access 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.
   $T\_W\_Port_{i,j}$ represents the average number of write access for port j of behavior i.
b) $T\_R\_DataType_i$ represents the read access of all the ports that have data type *DataType*, of behavior i.
   $T\_W\_DataType_i$ represents the write access of all the ports which have data type *DataType*, of behavior i.
c) T_R_i represents the total read access of all the ports of behavior i.
   T_W_i represents the total write access of all the ports of behavior i.

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

Besides communication through ports, behaviors also can communicate through it channels. The concept of channel is defined in [1]. Behavior use channel in term of function calls. For example, if behavior reads from channel A and save it into local variable b, behavior includes the statement b= A.receive(). On the other hand, if behavior writes value of variable b into the channel A, behavior includes the statement A.send(b). *Behavior profiler* calculates the argument of called channel functions as read access. If there are return data of channel functions, such as A.receive(), *behavior profiler* calculated them as write access.

4.2.4 Storage

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

4.2.4.1 Static storage

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

Static storage contains two levels:

a) $S\_Static\_DataType_i$ represents the total amount of the static storage of 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.

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

In Figure 5, S_Static_{Parent} =sizeof(G) + sizeof(A) +sizeof(B) + sizeof(C) +sizeof(*D).

Static storage can be directly analyzed from *original specification model.*

4.2.4.2 Heap storage

Heap storage refers to the storage that are allocated by "malloc" statements and are 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 of 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.

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

In figure 5, S_Heap_{Parent} = sizeof(D).

Unlike static storage, heap storage is analyzed based on simulation result *_Heap_Counter file* and *_BB_Counter_file* mentioned in 4.1

4.2.4.3 Stack storage

Stack storage is the first hierarchical storage concerned. It refers to the storage of the called functions of behaviors. The storage of function consists of function's static and stack storage and function's parameters. Since all the functions are called sequentially in behavior, only current called function storage is needed at time. Therefore, we use the maximum of storage of all the called functions of behavior as behaviors stack storage.

```
int G;

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 7: Example of specification model

Stack storage of behavior contains three levels:

(a) $S\_Stack\_DataType\_Fun_{i,j}$ represents the total amount of storage of 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 Figure 5, S_Stack_{parent}
= Max (S_Stack_Fun $_{parent, F1}$, S_Stack_Fun $_{parent, F2}$)
= S_Stack_Fun $_{parent, F1}$
= sizeof(E) + sizeof(L) + S_Stack_Fun $_{parent, F0}$
= sizeof(E) + sizeof(L) + sizeof(X).

Stack storage can be achieved 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 is defined as the sum of the storage of its sub_behavior instances. The storage of sub_behavior instance consists of instance's static, stack, heap, and sub_behavior storage.

In this project, Sub_behavior storage contains three levels:

a) $S\_Sub\_DataType\_Beh_{i,j}$ represents the total amount of the sub-behavior storage of 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 all the data of behavior i.

$$S\_Sub_i = Max_j(S\_Sub\_Beh_{i,j})$$

In Figure 5,
S_Sub_{Parent} = $S\_Sub\_Beh_{Sub\_1}$ + $S\_Sub\_Beh_{Sub\_2}$
= *sizeof (K) + sizeof(K2) +sizeof (K3) +sizeof (K4).*

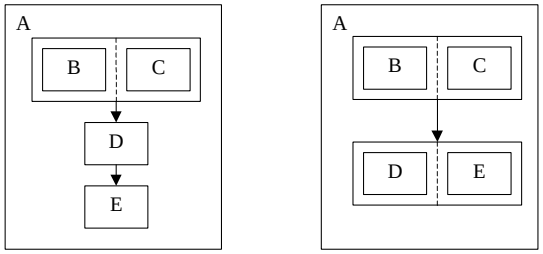
Similar to stack storage, sub_behavior storage can be achieved by hierarchically analyzing *original specification model.*

4.3 Behavior dependency analysis

Besides the statistics of each behavior, the relations among behaviors are also very useful for designers to make design decision. For example, if there is no traffic between behavior D and E as shown in Figure 6(a), the behavior D and E can be executed in parallel instead of in sequential, as shown in Figure 6(b), which may improve the performance.

Two types of behavior dependencies are analyzed: *calling dependency* analyzes the called/calling relations; *data dependency* analyzes whether there is traffic between behaviors.

Behavior dependency can be achieved by hierarchically analyzing *original specification model* and by analyzing port access statistics.



(a) (b)

Figure 8: Example of use 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 8(a), Behavior A does not have parent behavior. Its children behaviors and their execution relations 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 whether there is traffic between sub_behavior instances of the behavior. In Figure 6, data dependency represents the amount of traffic among behavior B, C, D, and E.

The sub_behavior instances are connected by their ports. There are two ways to connect ports of sub_behavior instances. First, the ports are connected through the global variables/channels that defined in the parent behavior, such as traffic between Inst1 and Inst2 through variable B, in Figure 5. We called these types of variable as connected variable, between connected behavior instances. Second, the ports are connected by the parent behavior's ports, such as traffic between Inst1 and Inst2 through port A in Figure 5. We called these types of variable as connected port, between connected behavior instances.

Based on the result of behavior statistics analysis, between any two behavior instances, the port-to-port traffics is calculated, for each connected port and each connected variable. The port-to-port traffic is called based on following equations.

a) Traffic for each connected port/variable

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

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

T_CV_k/T_CP_k represents the 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. In Figure5, for connected port A in 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 traffic between behavior instance i and j through connected variable/port, read access of mapped port in i may not equals to write access of mapped port in j. Therefore, maximum of read access of mapped port in i and write access of mapped port in j is added with maximum of write access of mapped port in j and write access of mapped port in i, which will be used as T_CV_k/T_CP_k. In Figure5, for traffic between behavior instance Inst1 and Inst2 through connected port A, T_CV_A = Max($T\_R\_Port_{Inst1,\ K2}$, $T\_W\_Port_{Inst2,\ K3}$) + Max($T\_W\_Port_{Inst,\ K2}$, $T\_R\_Port_{Inst2,\ K3}$).

b) Traffic for all connected ports/variables

$$T\_CP = \Sigma T\_CP_k$$
$$T\_CV = \Sigma T\_CV_k$$

The total traffic for all the connected variables/ports can be calculated by (15) and (16).

c) Traffic between behavior instances

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

The total traffic between behavior instance i and j are the sum of traffic for connected variables and connected ports.

If there is no traffic between two sub_behavior instance, we call this two behavior instances "data independent". Otherwise, it is called "data dependent". Furthermore, the closeness of two sub_behavior instances is represented by the traffic between sub_behavior instances.

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 these sub_section, two algorithms are described. The first one is for recursive function calls as well as operation calculation. The second one is for port access.

4.4.1 Algorithm for recursive function calls and operation calculation

In 4.2.2, we derive average execution numbers of operations of by equations (3)(4)(5), without considering function calls. In this sub-section, the algorithm for calculating average execution number of operations with considering sub_function calls is described.

There are three types of functions in SpecC. Local functions are the functions defined and used inside

behaviors; global functions are the functions defined outside behaviors but in specification; and library functions are the functions defined in libraries but used in specification. The local functions of each behavior can be called recursively. The global functions also can be called recursively.

The average execution number of operations for behavior equals to the average execution number of operations for main function of the behavior. Therefore, we can achieve the execution number of operations for behavior by only considering functions.

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

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

$$A = C * A + O$$

A is n-dimensional vector, where 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, where $C_{i,j}$ denotes how many times function i called function j. $C_{i,j}$ equals to execution number of basic block which contains calling statement divided by execution number of function j. O is dimensinal 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 in basic block k of function j, Op_Global_Function is the average execution number of operations of called global function in basic block k. Op_Sub is the average execution number of operations in the sub_behavior instance. BB_k is execution number of basic block k of function j. N_F_j is the execution number of function j.

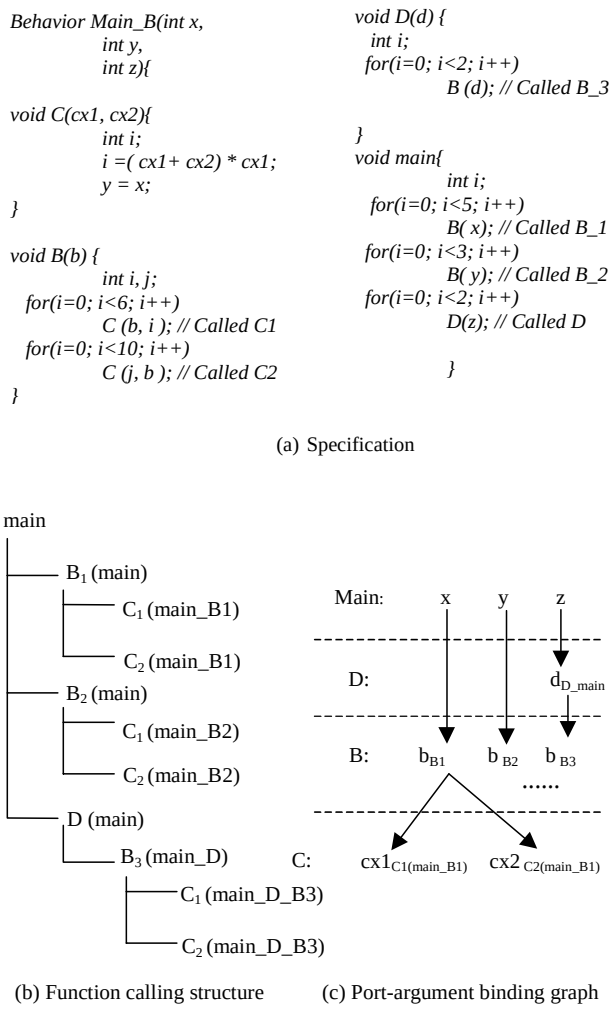


Figure 9: Example for algorithm of port access.

The only unknown in the system of liner equation (18) is the matrix A. Therefore, by solving the equation (18), we calculated 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 function of behavior, and indirect access from called functions and sub_behavior instances.

As shown in Figure 9(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 access, per C's

execution. For argument cx2 of function C, there is one read access, per C's execution. Since behavior Main_B's port x, y, and z are bound to arguments of called function C, cx1 and cx2, the arguments access of cx1 and cx2 should be counted as the port access of x, y, z. We call this port access *indirect port access*. On the other hand, the statement "x = y" in function C is called direct port access.

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

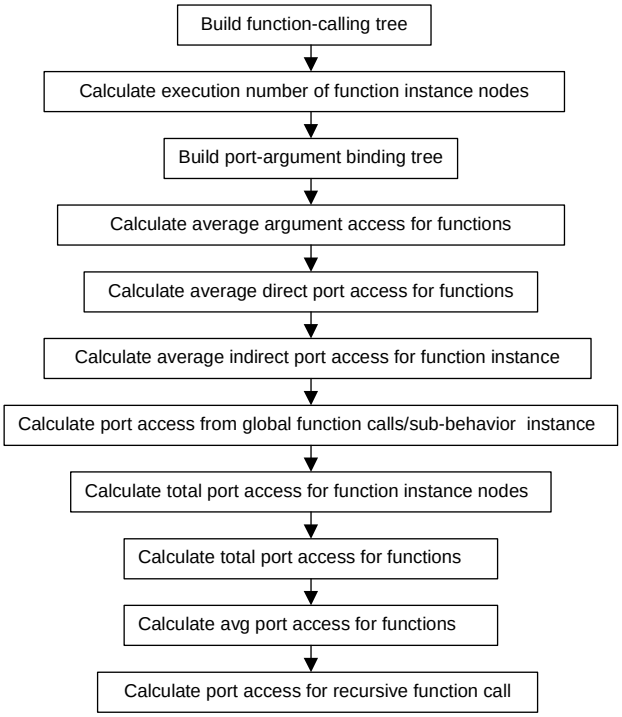


Figure 10: Design flow of analyzing port access

a) Build function-calling tree: similar to behavior calling tree in 4.2.1, *behavior profiler* analyzes the function-calling tree, as shown in Figure 9(b). The function-calling tree reflects the called and calling relation between function instance nodes.
b) Calculate 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 define F_N(i) as the total execution number of function instance node i,

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

While F_N(main) = 1. F_I(i,j) equals to execution number of the basic block that contains calling statement of function j divided by 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 each behavior instance nodes: The binding information is recorded if the argument of function instance node is bound to the port of behavior. For example, the port-argument binding tree is displayed as shown in Figure 9(c). In this binding tree, the argument cx1 of function instance node C1(main_B1) is mapped 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.
e) Calculate the direct port access D_P(i,j), while i refers to function instance node i and j refers to port j.

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

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 function instance nodes. First, the total argument access of function instance node D_A(i,k) are calculated, while i refers to function instance node i and k refers to argument k.

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

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 sub_behavior instances, for each port of function instance nodes,. Since

the port accesses of behaviors are calculated in the order from children behaviors to parent behaviors, the port accesses of sub_behavior instances are already known. The argument accesses of global function call are calculated before any behavior port calculation. After binding the ports of behavior to the ports of sub_behavior instance 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 accesses for each function instance node 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 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 the data type such as “int” can only be read but not written. On the other hand, for the pointer type, the argument can be read and written through pointer access. In *behavior profiler*, we calculated the argument access based on this fact.

5 Retargetable profiler

5.1 Design flow of the retargetable profiler

Statistics generated by *behavior profiler* are architecture-independent. Allowing users to estimate the system that reflects architecture exploration decision, we designed the *retargetable profiler*.:

Retargetable profiler did the following tasks

a) *Retargetable profiler* assigns a weight table to each behavior. The weight table represents the characteristics of operation, traffic, or memory for the PE that the behavior is mapped to. Thus, with the weight tables and the characteristics from *behavior profiler*, *retargetable profiler* generates the performance, traffic, and memory information of the behavior.

b) Each PE consists of a number of behaviors. All of these behaviors are executed sequentially. The behaviors communicate with the behaviors in the different PEs. Since the statistics of behavior can be achieved from a), the statistics of PE can also be generated.

c) The system consists of PEs. *Retargetable profiler* can generate the statistics of system based on the statistics of PEs.

SpecC profiler complete separates the *behavior profiler* and *retargetable profiler*. It makes that *retargetable profiler* does only depend on the output of *behavior profiler*, but not on the simulating result. That is, the task of *retargetable profiler* is to only use weight table, *behavior statistics*, and *behavior dependency* as input, without simulating testbenches and analyzing behaviors. Therefore, the process of executing *retargetable profiler* is very fast. For example, for the vocoder project [7] , in Sun’s Ultra 5 system, process of *behavior profiler* took 68 seconds, testbench simulation took 22 seconds, and process of *retargetable profiler* only took 5 seconds. Since the statistics of system for architecture exploration decision is calculated by executing *retargetable profiler* once, designers can complete changing the architecture exploration decision and re-profiling the design in very short time. It makes that the architecture can be explored in a very large range.

The process of *retargetable profiler* is in Figure 9, which has explained in section 1.

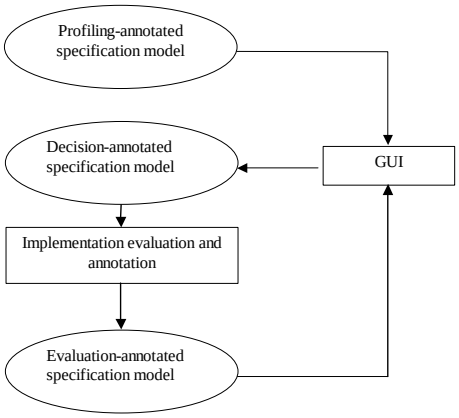


Figure 11: Design flow of retargetable profiler

5.2 Weight table generation

The items in weight table can be divided to three types: weight for operation, weight for traffic, and weight for memory. The weight for operation represents an operation weight for certain operation type and certain data type. The weight for traffic represents a traffic weight for certain data type. The weight for memory represents a memory weight for certain data type.

Each weight table represents one PE. PE can be a custom ASIC, a programmable processor, or an IP. If the PE refers to IP, the IP provider should provide related 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 reading processor manuals and analyzing based on source code generation [12]. For example, the SpecC source code is

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

After compiling, the target machine code is:

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

From these results, we concluded that each integer identifier and constant contributed with one MOVE instruction into the target code, each integer addition contributed with one ADD instruction, and there was no contribution from assignment. Using this way, the weight table can be achieved.

The number in the weight table can represents the time, clock-cycle, and number of bit/Byte, according to designers' purpose.

5.3 Output statistics of Retargetable profiler

The statistics of *retargetable profiler* include performance, traffic for behavior instance nodes and memory for behaviors. To simply the name, in this subsection, we also call behavior instance node as behavior.

5.3.1 Design Performance

Average performance of each behavior is evaluated. The performance can be counted as execution time, clock cycles of execution, or number of instruction, depending on the meaning of items in PE weight table. The average performance of behavior can be computed by adding weighted execution numbers of operations, which are the product of execution number of operations generated in 4.2.1 and their corresponding weight.

Two types of behaviors, leaf behavior and combinatorial behavior, are treated separately to achieve the performance of the design.

(a) Leaf behavior

Leaf behaviors are the behaviors that do not have any sub_behavior instances. The performance of each leaf behavior contains four levels:

$$P\_OpType\_DataType\_i = Op\_OpType\_DataType\_i * Weight\_OpType\_DataType\_i$$

$$P\_OpType\_i = \Sigma_{Datatype} P\_OpType\_DataType\_i$$

$$Avg\_P\_i = \Sigma_{OpType} P\_OpType\_i$$

$$Total\_P\_i = Avg\_P\_i * N\_N_i$$

Op_OpType_DataType_i is the average execution number of operation type *OpType* of data type *DataType*, for behavior i, as show in 4.2.2. The *Weight_OpType_DataType_i* is weight of the PE that behavior i partitions to, for operation type *OpType*

and data type *DataType*. N_N_i is the execution number of behavior instance nodes in 4.2.1.

The four levels of performance counted are: *P_OpType_DataType_i* represents the average performance of operation type *OpType* for data type *DataType* of the behavior i. *P_OpType_i* represents the average performance of operation type *OpType* for all data types of behavior i. *Avg_P_i* represents the average performance for all operation type and all data type of behavior i. *Total_P_i* represents the total performance of behavior i.

(b) Combinatorial behavior and PE

Combinatorial behaviors are the behaviors that have at least one sub_behavior. Combinatorial behavior can be used to represent the performance of PE or the whole design. The performance of combinatorial behavior i can be calculated as follows:

```
if execution relations (sub_behavior) == sequential
then
   Total_P_i = Σ_Sub_beh(Total_P_Sub_beh );
else
   if execution relations(sub_behavior) == parallel
   then
        if sub_behaviors are mapped to same PE
        then
             Total_P_i = Σ_Sub_beh(Total_P_Sub_beh);
        else
             Total_P_i = Max _Sub_beh(Total_P_Sub_beh);
        endif
   endif
endif
```

$Total\_P_{Sub\_beh}$ represents the performance of sub_behavior instances, which should be calculated before the performance of behavior i.

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

During performance estimation, we do not consider "waiting time". For example, when two behaviors A and B are executed in parallel, and A and B are mapped to different PEs, if B will not executed until B receive a input from A, there will be some waiting time for Behavior B. Designers should adjust the performance for this case, based on the profiling result.

5.3.2 Traffic

Retargetable profiler calculates the amount of traffic among behaviors. Compared with port width and port access, the amount of traffic is weighted traffic thus it is PE dependent. Based on port width and port access, two types of statistics, static traffic and dynamic traffic are generated by *retargetable profiler*.

a) Average static traffic

In some cases, when a behavior is executed, the data communication is only happened twice: right before the execution of behavior and right after the execution of behavior. During the execution, the data will be saved in local memories. In these cases, the total amount of data communication per behavior execution is called static traffic. If *C_S represents* average static traffic, static traffic 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 traffic for data type *DataType*.

b) Average dynamic traffic

If the data communication is happened whenever ports are access, the average traffic for each execution of behavior i is called average dynamic traffic, 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.

c) Total traffic of behavior

Furthermore, designer can calculate the total traffic of behavior i using average static traffic and average dynamic traffic. For example, assuming when each behavior is executed, the data communication is only

happened twice as discussed in 5.3.2 a), the total traffic

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

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 is happened whenever ports are access for each behavior, such as in 5.3.2.(b), the total traffic

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

d) Traffic between behaviors and PEs

Besides the traffic in and out a behavior nodes, the average traffic between behavior instances were generated, based on *behavior dependency* and traffic for each behavior, by calculating the port-to-port traffic between behaviors.

For example, if Behavior are executed sequentially, such as behavior D execute before behavior E as shown in *Figure 8*(a), and D and E communicate through connected ports, the traffic will be exist only when write D and read E. The amount of traffic $C_{D,\ E}$ is:

I. If D, E share a PE, there is no traffic.

II. If D, E communicate through a global memory:

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

III. If D, E communicate through a local memory and

i) If the local memory in D:

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

ii) If the local memory in E:

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

Retargetable profiler only provides the total dynamic traffic between behaviors instances. Designers should give the correct traffic based on different situation.

The traffic between two PEs are also can be calculated. The traffic between any behavior in PE1 and any behavior in PE2 are added to the traffic between PE1 and PE2. Designers can easily do this work manually.

5.3.3 Memories

Statistics of memories are weighted storages.

a) Static memory
Static memory

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

While $S\_Static\_DataType_i$ is described in 4.2.4.1 and *Weight_Memory_DataType* is the weight of memory for data type *DataType*.

b) Heap memory
Heap memory

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

While $S\_Heap\_DataType_i$ is described in 4.2.4.2.

c) Stack memory
Stack memory

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

While $S\_Stack\_DataType_i$ is described in 4.2.4.3

d) Sub_behavior memory
Sub_behavior memory

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

While $S\_Sub\_DataType_i$ is described in 4.2.4.4.

f) Total memory of behavior and PE
The total memory of behavior

$$M_i = M\_Static_i + M\_Heap_i + M\_Stack_i + M\_Sub_i$$

The total memory of PE

$$M\_P_k = Max(M_i)$$

For all behavior i in the PE k.

6 A design methodology of Using SpecC profiler

As mentioned in section 1, the designers make architecture exploration decision based on their design experience. In this section, a simple methodology of making architecture exploration decision by using *SpecC profiler* is introduced. This design methodology was developed based on our design experience and the advantages of *SpecC profiler*. It is simple and extendable. Thus designers can improve it according to different design cases. The design flow of this methodology is described in Figure 12.

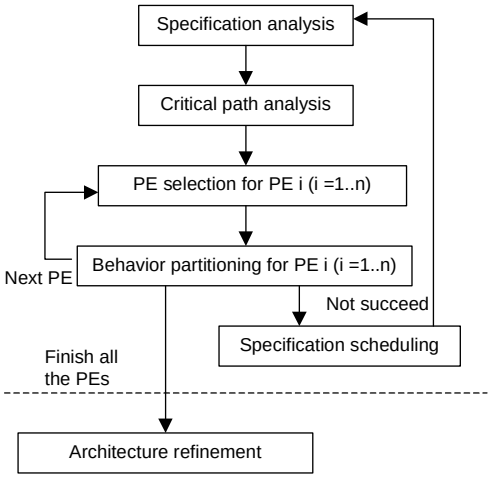


Figure 12: Design flow of the simple methodology for architecture exploration

6.1 Design assumption

We assume that the specification of design is specified in SpecC language, the execution relation between behavior instances in specification is either sequential or parallel. During the architecture exploration, these execution relations are not changed.

Since the architecture is assembled from its components, the component library (PE library) is already existed. For each PE, PE table includes three attributes: MIPS (Million instruction per second), MOPS (Million operation per second), and weight tables for operation, traffic and memory. For different weight tables, the items of weight tables can have different meanings. For example, if the weight for operation "+" is 3, it can be explained as that operation "+" is compiled to 3 instruction, or its execution time is 3ms.

The PE table used in our example is displayed in Table 1.

Name	MOPS	MIPS	WeightTable
SW1	2	7	WT1
SW2	5	17	WT2
HW	10	32	WT3

Table 1 : PE Table

6.2 Specification analysis

To illustrate the methodology, we use a simple example.

First of all, designers derived *behavior statistic* and *behavior dependency* by *behavior profiler*. Behavior calling tree of the example is displayed in Figure 13(a). In behavior calling tree, (--) represents the sequential execution among subbehavior instances, while (||) represents parallel execution and (Leaf) represents leaf behavior. Since there are no two behavior instances that are the instances of same behavior, we use behavior's name as behavior instance name in the behavior calling tree. In this section, we behavior instance node as behavior for sipilicity.

In table 2, the *behavior statistics* of the example is listed. To make it simple, only the total execution number of operation, total traffic, and total memory size for each behavior are chosen.

Name	Operation	Traffic	Mem
Main	56k	0	5k
AB	20k	50 Word	3k
CD	11k	50 Word	1k
A	20k	0	2k
B	8k	0	1k
C	9k	0	0.5k
D	11k	0	0.5k
E	25k	0	1k
A_1	5k	0	0.5k
A_2	15	0	0.5k

Table 2 : Behavior statisitics of example

Based on *behavior statistics and* behavior calling tree, the behavior parallel graph can be generated by designers, as shown in Figure 13(b).

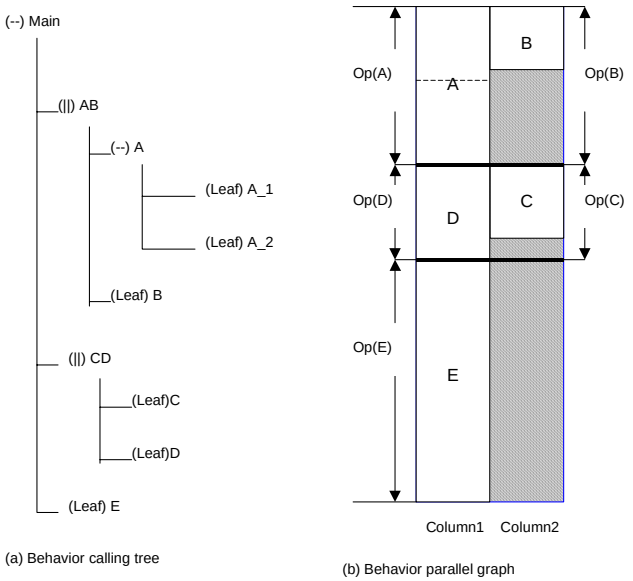


Figure 13: Specification display

In behavior parallel graph[13], each white block represents a behavior, labeled with behavior name. The behaviors in different columns are executed in parallel, such as A and B, or C and D, based on the specification. In each column, the behaviors are executed sequentially, starting from the top. If two behaviors are executed in parallel, the successor of them can not be executed until the executions of two behaviors are finished. For example, C and D will start executing after execution of A and B. In behavior parallel graph, the length of block represents the total execution number of operations (Op) for behavior. The shaded block means do nothing.

6.3 Critical path analysis

If two behaviors are executed in parallel, the one that has the greater execution time will influence the performance of the system. Therefore, critical path is first analyzed. Critical path consists of behaviors. If the behaviors are executed in parallel, the behavior that have largest Op is in the critical path, such as A and D in Figure 10. Sequential behaviors are always in the critical path, such as E in Figure10.

Designers should rearrange the behaviors in behavior parallel display, for showing the critical path. For any behavior I and J, if Op(I)> Op(J), then Column(I)< Column(J), while column(I) refers to the column number of behavior I. Therefore, the critical path is represented in column 1, as shown in Figure 13.

6.4 PE selection

The behaviors on different columns of behavior parallel graph are executed in different PEs. Therefore, PEs are first selected column by column. Since the column 1 represents the critical path, PE selection should in the order from column 1 to column n. Op of column equals to the sum of Ops of behaviors in the column. Since execution time of system is related to Op of critical path, we use critical path as an example to show our methodology.

In the beginning, designers should try to find easiest design solution: pure SW solution. In this solution, all the behaviors in critical path are implemented in one programmable processor. From 6.3, we knew the total execution number of operation of critical path, which is called Op(Critical_Path). We also know the time constraint of system, which is called T_Constraint. Thus, Op(Critical_Path) divided by T_Constraint can tell us how many operations are needed to proceed per second, which is called Need_MOPS. By comparing Need_MOPS and MOPS of PEs in PE library, we can find the suitable PE.

Using MOPS, we select PE in operation level, which is not accurate. After using *retargetable profiler*, performance of critical path, which is called P(critical_path), can be generated. If the items in weight table represent the number of instructions for each operation, P(critical_path) represents the total execution number of instructions. Similar, P(Critical_Path) divided by T_Constraint can tell us how many instructions are needed to proceed per second, which is called Need_MIPS. By comparing Need_MIPS and MIPS of selected PE, we can ensure that the PE we select based on MOPS can meet time constraint.

The algorithm of selecting the SW PE can be described as follows:

```
Need_MOPS =  Op(Critical_Path)/T_Constraint;
for test_PE = PE1 to PE n do
    if Need_MOPS > MOPS(test_PE) then
        test next PE;
    else // Have SW PE good for MOPS
        Generate P(Critical_Path), based on test_PE;
        Need_MIPS = P(Critical_Path) /
     T_Constraint;
        if Need_MIPS <= MIPS(test_PE) then
```

```
            return (test_PE);
            // Find SW PE good for MIPS
        endif
    endif
endfor

/* If no PE can be selected based on MOPS, try the
fastest PE based on MIPS, because MOPS is not
accurate for PE selection */
Generate P(Critical_Path), based on PEn;
Need_MIPS = P(Critical_Path) / T_Constraint;
if Need_MIPS <= MIPS(PEn) then
    return (PEn);
else
    return (Not_Found);
endif
```

In this algorithm, PEs refers to SW PEs in PE library. PEs are numbered in the order of MOPS, from smallest (PE1) to greatest (PEn).

For example, if the time constraint of example in Figure 11 is 16ms. Needed_MOPS = 56k / 16ms = 3.5 MOPS. From table 1, SW2 can be selected. With WT2 as our weight table, *retargetable profiler* generated P(Critical_path) as 196k. Then Needed_MIPS = 196k / 16ms = 12.25 MIPS, which is smaller than MIPS(SW2). Therefore, SW can be selected for pure SW design of system.

If the algorithm can find SW PE, all the behaviors in critical path will be implemented in selected SW PE. After selecting PE for critical path, designers can select PEs for the column 2, 3... n of behavior parallel display, in the same way.

If there is no PE that can be found for system, the behavior partitioning will be implemented.

6.5 Behavior partitioning

If SW PE can meet the requirement, SW-HW co-design should be implemented. In our methodology, designers choose fastest SW component and pre-defined custom hardware as the PE components for co-design. The purpose of partitioning is to put some behaviors into HW to ensure the whole system meet the time constraint. In our example, SW2 and HW are selected from Table 1 as the system components.

Two tasks are needed to find the suitable behaviors, The first task is to evaluate the performance based on selected PEs, the second task is to find the behavior suitable for HW implementation.

When we select behavior for HW, not only the performance of behavior should be concerned. So is the cost of behavior for HW implementation. If there are several behaviors that can be selected for HW to meet the time constraint, the one that has the lowest cost should be chosen.

Since the way of cost evaluation is not available, we evaluate the cost of behavior based on behavior hierarchy. We assume that the cost of children behaviors is always smaller than their parent behavior. The costs of any two-leaf behaviors are equal.

The behavior hierarchical relation in each column can be illustrated by behavior calling tree, by cutting the nodes that are not executed in this column. Therefore, the cost comparison between behaviors can be derived from behavior hierarchical tree. For example, for column 1 in Figure 10 (b), Main behavior include three children A, D, and E. Behavior A also includes two leaf behaviors: A_1 and A_2. Therefore, Cost(A_1) < Cost(A) < Cost(Main).

We implement behavior partitioning by following bottom-up algorithm. In this algorithm, we first select the leaf behavior that have largest P(behavior), then select its ascent if needed.

```
Selected_Beh= Null;
for i=1 to num_of_leaf_behavior do
    if P (Leaf_behavior(i) ) >  P(Selected_Beh)
    then
      Selected_Beh = Leaf_behavior(i);
    endif
endfor

P(Critical_Path) = P_REPROFILE(Critical_Path,
Selected_Beh);
while (Selected_Beh != NULL) & (P(Critical_path)
> T_constraint ) do
    P(Critical_Path) = P_REPROFILE(Critical_Path,
  Selected_Beh);
    Selected_Beh = Parent (Selected_Beh);
    Generate P(Critical_Path) ;
endwhile
return Selected_Beh;
```

In our example, Leaf_behaviors are A_1, A_2, B, C, D, and E. The Parent(A_1) is A. The purpose of P_REPROFILE is to derive the performance of critical path, in the case that the second argument Selcted_Beh is mapped to HW while the rest of behaviors are mapped to SW, for the column

represented by first argument. P_REPROFILE is implemented by *retargetable profiler.*

The advantage of this algorithm is simple. The disadvantage is that it needs to execute *retargetable profiler* several times for function P_REPROFILE.

A revised algorithm is given in the following. Firstly, *retargetable profiler* generates performance of each behavior for SW and HW, respectively. We use P_S(i) for SW performance of behavior i, while P_H(i) for HW performance of behavior i. Further more, P_Comm(i) is used to represent the performance overhead for traffic of behavior i.

To find the behavior for HW, we calculate Need_Improvement.

$$Needed\_Improvment = P\_S(Critical\_path) - T\_constraint$$

P_S(Critical_path) is the performance of critical path/system for pure SW solution. Secondly, Perf_Gain is calculated for each behavior.

$$Perf\_Gain(i) = P\_S(i) - P\_H(i) - P\_Comm(i)$$

For any behavior i, if Perf_Gain(i) > Needed_Improvment, we can say that the system can meet the time constraint by implementing behavior i in HW while rest of behaviors in SW. Thus, we need to find the behavior that can satisfy above equation.

Finally, we can get the behavior for HW by following algorithm,

```
Selected_Beh= Null;
Perf_Gain(Selected_Beh) = 0;
for i=1 to num_of_leaf_behavior do
    if  Perf_Gain ( Leaf_behavior(i) ) >
    Perf_Gain(Selected_Beh) then
      Selected_Beh = Leaf_behavior(i);
    endif
endfor

while  (Selected_Beh != NULL) &
(Perf_Gain(Selected_Beh)
< Needed_Improvement )  do
    Selected_Beh = Parent (Selected_Beh);
endwhile
return Selected_Beh;
```

In our example the performance of behaviors are listed in Table 3.

Name	P_S(ms)	P_H(ms)	P_Comm (ms)	Perf_Gain(ms)
Main	11.52	6.07	0	5.45
AB	6.49	3.44	0.50	2.55
CD	2.26	1.20	0.50	0.56
A	4.11	2.18	0	1.93
B	1.64	0.87	0	0.77
C	1.85	0.98	0	0.87
D	2.26	1.20	0	1.06
E	5.14	2.70	0	2.44
A_1	1.02	0.53	0	0.49
A_2	3.08	1.64	0	1.44

Table 3: Performance for behaviors in example

If the performance constraint of design is 10ms. Needed_Improvement = 11.52- 10 = 1.52 ms. Among leaf behavior E, A_1, and A_2, we can found behavior E has greatest Perf_Gain, which is 2.44ms. Since Perf_Gain(E) is greater than Need_Improvement, it is the behavior for HW.

If Selected_Beh is found out by this algorithm, the system can meet the constraint requirement by implementing Selected_Beh in HW.

If no behavior can be selected, and P_H(System) > T_constraint, it means even the pure HW solution can not meet the requirement. In this case, the behavior scheduling should be made.

6.6 Behavior scheduling

If the HW solution can not be meet the requirement, the specification should be scheduled by designers, to change more sequential execution to parallel execution. Designers can make this change based on *behavior-dependency*, since *behavior-dependency* can tell designers whether two behaviors are independent. For example, if the performance constraint of design is 5ms, and SpecC profiler tell designers that there are no traffic between behavior E and behavior CD or AB, designers should change the specification to make behavior E parallel executed with behavior AB and CD. The PE selection, behavior partitioning and behavior scheduling should be make executed again for new specification.

7 Conclusion

In this report, the SpecC profiler is introduced. SpecC profilers contain two parts: *behavior profiler* and *retargetable profiler*. *Behavior profiler* provides the characteristics of specification, while *retargetable profiler* provides the characteristics of system, according to designers' architecture exploration decision.

SpecC profiler has following advantages

a) SpecC profiler is retargetable profiler: it can evaluate the characteristics of behavior that is executing on any selected PEs. Moreover, it can provide profiling result for the case that different functions of specification are executed on different PEs.
b) SpecC profiler not only evaluates performance, but also evaluates the amount of traffic, needed memory size, for each function.
c) SpecC profiler evaluates parallel execution among functions as well as sequential execution.
d) SpecC profiler analyzes the behavior dependency, which provides designers more flexibility of scheduling.
e) Two part of SpecC profiler, *behavior profiler* and *the retargetable profiler*, are separated. Therefore, design simulation is only executed once. It makes the process of executing *retargetable profiler* fast and makes large range architecture exploration possible.
f) SpecC profiler is designed for SpecC language, which is worldwide-accepted system level design language. Thus, it can be used as one of tools in SpecC Engine for other Chip designers and EDA venders.

Besides SpecC profiler, a simple methodology of making architecture exploration decision is given.

The SpecC profiler works on specification model of SpecC language. The evaluation result is not real cycle-accurate. However, it is first step to limit the range of architecture exploration and it outlines profiler of abstract level 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