System Level Design Using SpecC Profiler

Lukai Cai, Daniel Gajski

Technical Report #02-08
April 2002

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

(949)824-8059

{lcai, gajski}ecec.uci.edu

Abstract

This report illustrates the use of SpecC profiler for the system level design at high levels of abstraction. SpecC profiler allows the designers to choose the granularity of specification, to select PEs for architecture, and finally to map specification to PEs based on the given design constraints. The report also provides system level design examples including JPEG encoder, JBIG encoder, and Vocoder projects. We show that the design processes are much smoother and easier with the use of SpecC profiler.

Index

List of Figures

List of Tables

System Level Design Using SpecC Profiler

Lukai Cai, Daniel Gajski
University of California, Irvine

Abstract

This report illustrates the use of SpecC profiler for the system level design at high levels of abstraction. SpecC profiler allows the designers to choose the granularity of specification, to select PEs for architecture, and finally to map specification to PEs based on the given design constraints. The report also provides system level design examples including JPEG encoder, JBIG encoder, and Vocoder projects. We show that the design processes are much smoother and easier with the use of SpecC profiler.

1 Introduction

1.1 SpecC Profiler

This report describes the usage of using SpecC profiler [1] for the system level design at the functional level of specification.

SpecC profiler [1] is a profiler for the specification model of SpecC language [2][3]. Specification model of SpecC language specifies the desired system functionality without containing any implementation information. It is the highest abstraction model in SpecC methodology. In comparison to other profilers[1], SpecC profiler has following characteristics:

a) It is retargetable. It analyzes the given specification and generates the characteristics of behaviors in the specification when behaviors are executed on the mapped processing component (PE). (In the specification model of SpecC language, behavior refers to a behavior entity that encapsulates a number of functions and connects to other behaviors by its ports). It also generates the statistics for the entire design in the case that the different behaviors of the design are executed on different PEs
b) It not only computes the characteristics of computation, but also computes the characteristics of communication and needed memory size of design.
c) It computes the characteristics of design when the behaviors of the design are run in sequential, parallel or pipeline style hierarchically.

1.2 Tasks of System Level Design for The Specification Model

The SpecC methodology is a top-down design approach. SpecC methodology contains three main tasks, *specification modeling, architecture exploration, and specification-architecture mapping,* as depicted in Figure 1. *Specification modeling* involves modeling the design functionality and determines the granularity of behaviors and the primitive behaviors for *specification-architecture mapping*. Task *architecture exploring* then involves choosing the computation and communication components from a library and assembling the selected components into a system architecture. Finally, *specification-architecture mapping* involves mapping the primitive behaviors produced by *specification modeling* to the selected components of architecture, to meet the imposed design constraints.

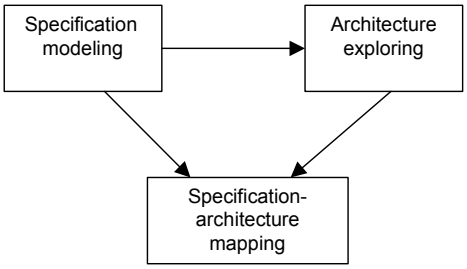


Figure 1: Tasks of system level design for the specification model

a) *Specification Modeling.*

Specification modeling composes of two tasks. First, designers specify the functionality using the specification model of SpecC language[2][3], in which parallel and pipeline execution are specified explicitly.

Second, the granularity of the specification for specification-architecture mapping is chosen. This process defines primitive behaviors for specification-architecture mapping. Each primitive behavior can be mapped to only one PE.

b) *Architecture Exploring.*
Designers select PEs and buses from PE and bus library and generate the system architecture. The PE/bus selection must guarantee that the imposed design constraints are met when the specification is implemented on the selected PEs/buses.

c) *Specification-Architecture Mapping.*
The purpose of specification-architecture mapping is to generate a design implementation to meet the design constraints with low design cost. According to the specification generated by specification modeling and the system architecture generated by architecture exploration, designers map primitive behaviors to PEs, and map communication/channels between behaviors to buses/protocols. After specification-architecture mapping, a design implementation at the high level of abstraction is generated.

SpecC profiler computes the characteristics of computation, communication, and storage size of behaviors, which help designers to understand specification and to choose the granularity for specification-architecture mapping. It also evaluates the design performance based on the given specification and the selected system architecture, which is the foundation of architecture exploration and specification-architecture mapping. Thus, system level design of the high level of abstraction using SpecC profiler becoming much easier. With the help of SpecC profiler, we have successfully modeled the design examples of JPEG encoder, JBIG encoder, and Vocoder projects.

This report is organized as follows: Section 2 introduces specification modeling. Section 3 introduces architecture exploring. The following section introduces specification-architecture mapping. Section 5 through section 7 illustrate the design processes of JPEG encoder, JBIG encoder, and Vocoder. And finally section 8 concludes the report.

2 Specification Modeling

Specification modeling uses SpecC language [2][3] to model the functionality of design. The guidelines of specification modeling are [4]:

a) Separate communication and computation.
b) Explore inherent parallelism in the system functionality.
c) Use hierarchy to group related functionality.
d) Choose the granularity of specification and determine primitive behaviors for specification-architecture mapping.
e) Use state transitions to explicitly model the computation steps.

Designers can follow guidelines a, c, and e without difficulty. However, to follow guideline b and d, designers must know behavior characteristics, such as the total amount of communication among behaviors and the total number of execution of operations in behaviors, which can be generated by SpecC profiler.

2.1 Exploring Parallelism

The total amount of communication between two behaviors produced by SpecC profiler can help designers to explore parallelism. The total amount of communication between two behaviors $T\_BB_{i,j}$ is described in section 4.3.2 of [1].

For example, in Figure 2, if behavior A and B are executed one after another, and there is no communication between A and B, then A and B can be executed in parallel instead of in sequence.

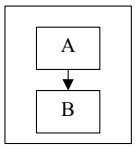


Figure 2: An example of potential parallel execution.

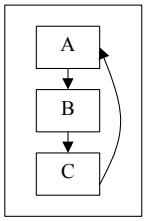


Figure 3: An example of potential pipeline execution.

Another example is illustrated in Figure 3. Behavior A, B and C are executed sequentially inside a loop. If traffic only exists from A to B, and from B to C, then behavior A, B and C can be executed in pipeline style.

2.2 Choosing Granularity

2.2.1 Behavior Hierarchy

In the specification model of SpecC, there are two types of behaviors: leaf behavior and non-leaf behavior. A leaf behavior contains a list of statements, such as *a = b + c*, or *if (a > b) { a = c;}*. A leaf behavior does not contain any behavior instantiations. A non-leaf behavior consists of a set of behavior instantiations. These instantiated behaviors are explicitly called in sequential, parallel, pipeline, or state transition style [2][3]. A good non-leaf behavior does not include any statements besides instantiated behavior calls, in order to have good modularity. A good non-leaf behavior is called as clean behavior. If all the non-leaf behaviors in the sepecification are clean behaviors, then the specification is a clean specification. The method of specification modeling described is only appropriate for clean specifications. We also illustrate the method of specification modeling for non-clean specification by using the design process of JBIG encoder described in section 6.

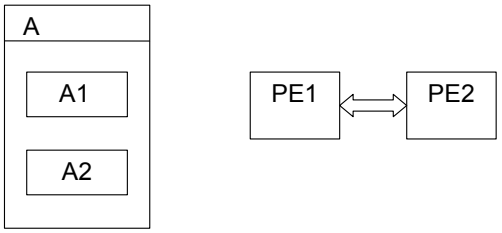


Figure 4: An example of choosing granularity for specification-architecture mapping.

In general, all the leaf behaviors reflect algorithm blocks. The design experience of Vocoder described in section 7 gives an example of specification modeling for the case that leaf behaviors don't reflect the algorithm blocks.

2.2.2 Granularity of Primitive Behaviors for Specification-Architecture Mapping

Primitive behavior for specification-architecture mapping is defined as the undivided behaviors that designers can only map it as a whole to one PE rather than map different parts of it to different PEs. Granularity of primitive behaviors determines the flexibility of specification-architecture mapping. For example, in Figure 4, if designers choose behavior A that has coarse granularity as the primitive behavior, then two mapping possibilities are:

a) Map(A) = PE1
b) Map(A) = PE2

However, if designers choose behaviors with finer granularity as primitive behaviors, i.e., choose A1 and A2 as primitive behaviors, then there exists four mapping possibility:

a) Map(A1) = PE1, Map(A2) = PE1
b) Map(A1) = PE1, Map(A2) = PE2
c) Map(A1) = PE2, Map(A2) = PE1
d) Map(A1) = PE2, Map(A2) = PE2

The smaller the granularity of primitive behaviors for mapping, the larger range specification-architecture mapping can be explored. Hence, the better mapping solution

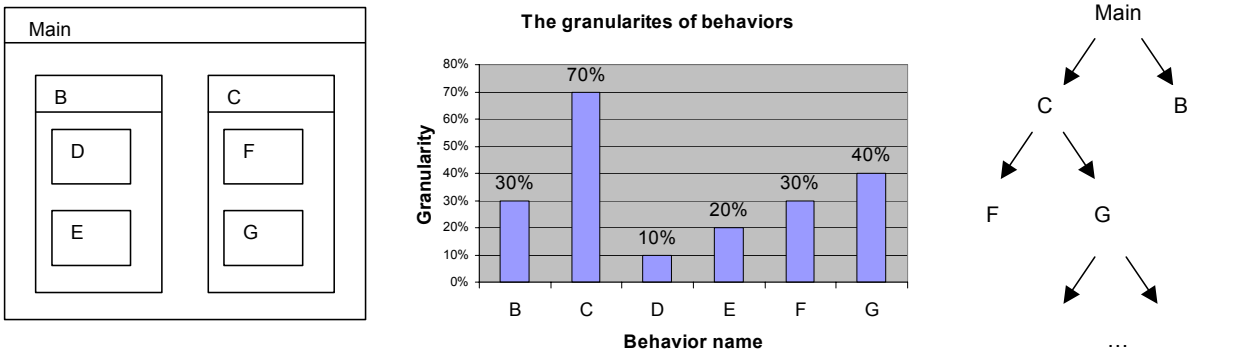
designers can produce during specification-architecture mapping. On the other hand, design with primitive behaviors of finer granularity needs more work on decomposing behaviors and on making decisions for specification-architecture mapping.

```
Algorithm DETERMINING_PRIMITIVE_BEHAVIORS

    primitive_behaviors = behaviors representing algorithm blocks.
    implementation = ARCHITECTURE_EXPLORATION__SPEC_ARCHITECTURE_MAPPING();
    while  NOT_MEET_CONSTRAINT(implementation)  do
        upperbound = MAX (G(i) ), for all i in primitive_behaviors; // G(i) is the granularity of behavior i
        upperbound = upperbound / 2;
        b = FIRST_ITEM_IN(primtive_behavior);
        while  b != NULL  do
                DECOMPOSING (b, upperbound, primtive_behavior);
                b = NEXT_ITEM_IN(primtive_behavior);
        endwhile
        implementation = ARCHITECTURE_EXPLORATION__SPEC_ARCHITECTURE_MAPPING();
    endwhile


function DECOMPOSING (i, upperbound, primtive_behavior){
    if (G(i) > upperbound) do
        If (IS_NON_LEAF_BEHAVIOR(i)) do
            REMOVE_I_FROM_LIST(i, primitive_behavior);
            for ( each j = child(i) ) do
                APPEND_J_TO_LIST( j, primtive_behavior);
            endfor
        else
            DECOMPOSING_LEAF_BEHAVIOR (i, upperbound);
        endif
    endif
```

Figure 5: The algorithm for determining primitive behaviors for specification-architecture mapping.



(a) Behavior hierarchy display

(b) Granularities of behaviors

(c) Primitive behavior searching tree

Figure 6: An example of determining primitive behaviors of specification-architecture mapping.

We define the granularity of a behavior as the percentage of the total number of execution of operations of the behavior in the total number of execution of operations of the whole specification. We use G(A) to denote the granularity of behavior A,

$$T\_OP_A = N\_B_A * OP_A$$
$$G(A) = T\_OP_A / T\_OP_{Main}$$

where N_B_A represents the number of execution of behavior A. OP_A represents the average number of execution of operations in a single execution behavior A, per behavior A's execution. T_OP_A is the total number of execution of operations in behavior A. T_OP_{Main} is the total number of execution of operations in the whole specification during simulation. OP and N_B are generated by SpecC profiler as described in section 4 of [1].

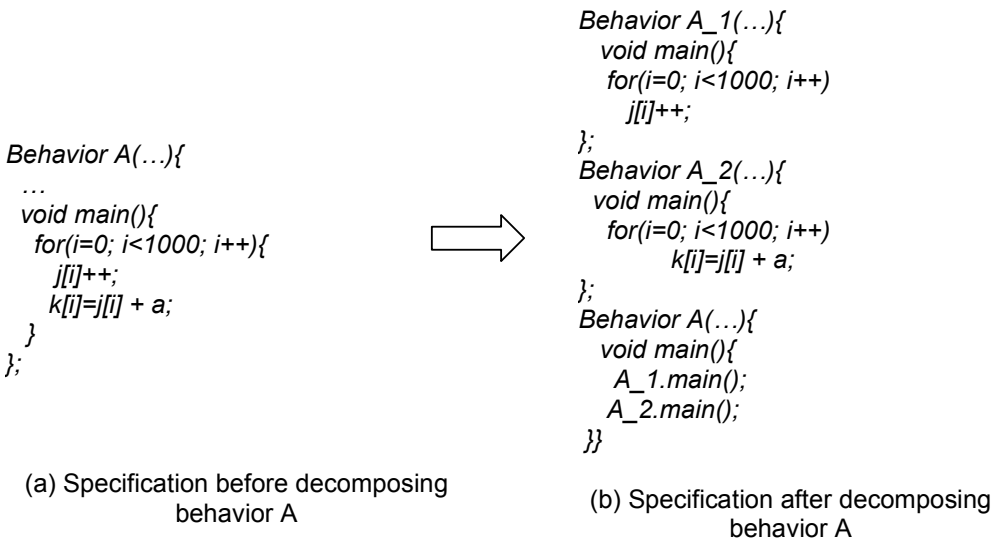
Figure 6 shows an example to illustrate the algorithm in Figure 5. Figure 6(a) displays the behavior hierarchy of the example, Figure 6(b) lists the granularities of the leaf behaviors in the example. In this example, behavior B and C reflect the algorithm blocks.

First, algorithm chooses behavior B and C as the primitive behaviors to do the architecture exploration and behavior-architecture mapping. However, the resulting implementation does not meet the imposed design constraints. Therefore, behaviors with smaller granularity must be chosen.

The algorithm computes variable *upperbound* as 35%, which is the half of MAX(G(C), G(B)). The function DECOMPOSING then decomposes all the primitive behaviors whose granularities are greater than *upperbound* into parts. For example, algorithm decomposes behavior C to behavior F and G, removes behavior C from set *primitive_behaviors* by REMOVE_I_FROM_LIST, and added behavior F and G to *primitive_behaviors* by APPEND_J_TO_LIST.

2.2.3 Determining Primitive Behaviors for Specification-Architecture Mapping

In general, we choose the behaviors representing algorithm blocks as primitive behaviors for specification-architecture mapping. However, sometimes designers cannot find implementation to meet the design constraints because the granularities of these primitive behaviors are too coarse. Therefore, to produce the implementation that meet the given design constraint, designers must decompose at least one of primitive behaviors to several new primitive behaviors and redo the architecture exploration and specification-architecture mapping with the new set of primitive behaviors. Figure 5 provides an algorithm to determine the primitive behaviors for mapping for designers.



(a) Specification before decomposing behavior A

(b) Specification after decomposing behavior A

Figure 7: An example of leaf behavior decomposing.

Function DECOMPOSING tests all the new generated primitive behaviors, i.e., behavior F and G. It decomposes behavior G by function DECOMPOSING_LEAF_BEHAVIORS because granularity of behavior G is greater than *upperbound* and G is a leaf behavior. Decomposing leaf behaviors is quite complex and is explained in 2.2.4. Figure 6(c) displays the decomposing tree of this example of Figure 6(a).

2.2.4 Decomposing Leaf Behavior

To decompose a leaf behavior, we group the statements of this behavior to several sets, each of which becomes a new leaf behavior. Note that during the decomposing, the computational and communication overheads may be generated. For example, in Figure 7, behavior A is decomposed into two behaviors, A_1 and A_2. Since behavior A contains a *for* loop, each of newly formed behaviors A_1 and A_2 contains one *for* loop. Hence a additional *for* loop forms the computation overhead. Also, since both behavior A_1 and A_2 access array j, decomposition increases communication overhead if we map behavior A_1 and A_2 to different PEs. Hence, if decomposing a leaf behavior produces heavy computational and communication overheads, instead of decomposing the leaf behavior we still use it as primitive behavior.

SpecC profiler can compute the computational and communication overheads for leaf behavior decomposing. For example, in Figure 7, SpecC profiler generates the average number of execution of operations (OP) of behavior A before and after decomposing. By comparing the two, designers can obtain the computation overhead, which is ($T\_OP_{new\_A}$ - $T\_OP_{old\_A}$). T_OP is described in section 2.2.2

In Figure 7, the communication overhead equals to ($T\_BB_{A1,A2}$ * N_B_A), while T_BB represents the traffic between behavior instances A1 and A2 and N_B represents the number of execution of behavior A. T_BB and N_B is described in section 4 of report [1].
Note that in some cases, designers cannot decide whether a leaf behavior should be decomposed or not entirely on the basis of the resulting overhead. If new generated behaviors can be executed in a parallel/pipeline style instead of in a sequence style, a certain amount of overhead can be tolerated.

3 Architecture Exploration

The purpose of the system level design is to produce an implementation with satisfactory performance and with low cost. However, architecture model cannot determine the cost and the performance of implementation before performing specification-architecture mapping. Therefore, we divide the task of the architecture exploration into two steps. In the first step, we limit the range of the architecture exploration without performing specification-architecture mapping, according to the characteristics of specification and imposed design constraints. In the second step, we select the architecture for specification-architecture mapping. This section only introduces the first step of the architecture exploration. The step two which is called architecture selection will be introduces in next section.

In this report, we use execution time as the performance of design. In Figure 8, the shaded circle at the top-left corner represents the processor with the lowest performance and with the lowest cost among processors on PE library. This single processor represents the simplest architecture.

There are two directions to improve the performance of design by changing the architecture. In the horizontal direction, the performance can be improved by using multiple PEs instead of single PE, if parallelism exists in given specifications. In the vertical direction, the performance can be improved by using faster PE to replace slower PE. With the improvement of the performance, the design cost also increase, which is denoted by single-line arrows in Figure 8.

With the given specification and the imposed time constraint, there is an upperbound on the amount of PEs in the architecture. Increasing the amount of PEs in the selected architecture does not improve the design performance if the amount of PEs in the architecture is already not less than the upperbound on the amount of PE. Similar, there is an upperbound on PE speed for single-PE architecture. If PE speed is lower than the upperbound, time constraint cannot be met when all behaviors are executed on this PE. In Figure 8, the upperbound on the amount of PEs concerns the performance gain in the horizontal direction; while the upperbound on PE speed concerns the performance gain in the vertical direction.

In Figure 8, the dotted area represents the architecture exploration range.

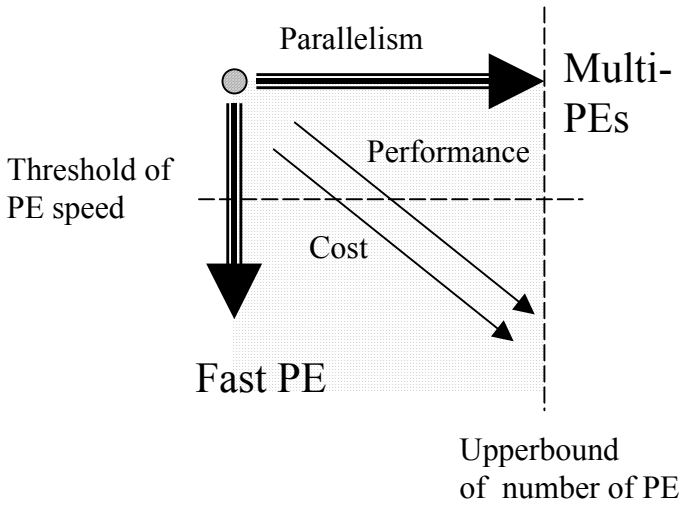
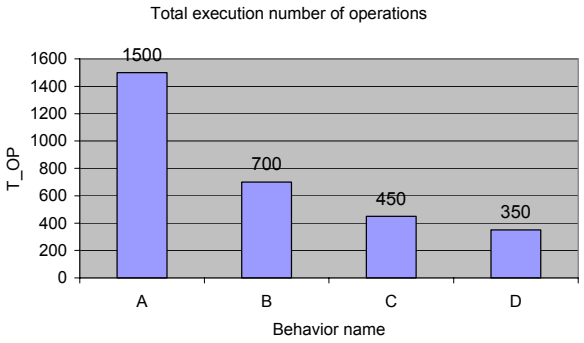
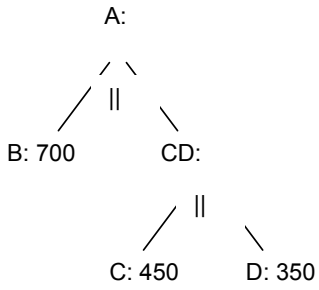


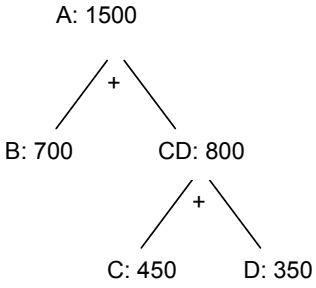
Figure 8: The range of architecture exploration



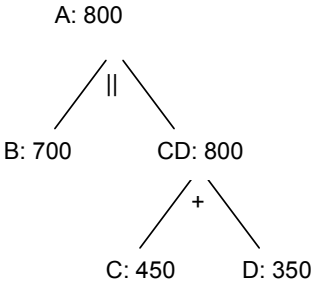
(a) Total execution number of operations of behaviors



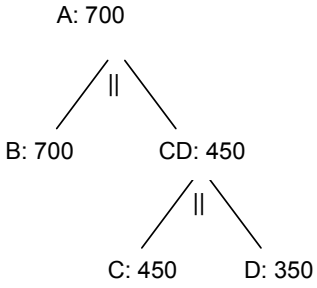
(b) Hierarchical construction of behavior A



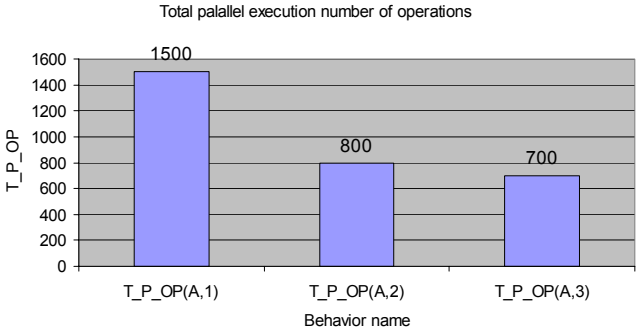
(c) T_P_OP(A, 1)



(d) T_P_OP(A, 2)



(d) T_P_OP(A, 3)



(e) Total parallel execution number of operations of behavior A

Figure 9: An example of calculating T_P_OP.

3.1 Upperbound on Amount of PEs

In most cases, the more PEs the architecture contains, the higher the cost of the implementation is, and the more complex the design process is.

The maximal amount of parallelism in the specification determines the upperbound on the amount of PEs in the architecture. Adding a new PE to the architecture cannot improve design performance if the amount of PEs in the architecture has been no less than the maximal amount of parallelism in the specification. Thus, the maximal amount of parallelism in the specification is the upperbound on amount of PEs.

We further compute the total parallel amount of execution of operations of behavior according to the selected amount of PE but not PE performance. The total parallel number of execution of operations of behavior equals to the sum of T_OP of its child behaviors if the child behaviors are in same PE and equals to the maximum of T_OP of its child behaviors if the child behaviors are in different PEs. Since total parallel number of executions of operations of behavior do not only depend on the given specification and the selected architecture, but also depend on the specification-architecture mapping, we use T_P_OP to represent the smallest value of total parallel number of execution of operations of behaviors.

The example in Figure 9 illustrates our method of calculating T_P_OP. Behavior A contains three parallel executing child behavior B, C, and D, the total number of execution of operations of which are displayed in Figure 9(a).

T_P_OP(A, n) denotes T_P_OP for behavior A when n PEs is in the architecture. Our algorithm first builds the hierarchy tree of A's child behaviors according to hierarchical clustering algorithm[11]. Two child behaviors that have the smallest T_OP will be clustered to one hierarchical node at a time until only one hierarchical node left at the highest level. Figure 9(b) displays the hierarchical tree of behavior A. Two "||" symbols represents that the child nodes of node A and CD can be executed in the parallel style.

Second, we assign parallel execution relation to hierarchical nodes from the top to the bottom in the hierarchical tree until the amount of assigned parallel execution is one smaller than the amount of PE in architecture. A node is assigned a parallel execution only when the child nodes of it can be executed in parallel as indicated by "||" in the hierarchical tree and there is a parallel execution available for assignment.

Third, we compute T_P_OP for each hierarchical node in the tree. If the execution relation of hierarchical node is a parallel execution, then T_P_OP of it is equal to the maximum of T_P_OP of its child node. If the execution relation of hierarchical node is a sequential execution, then T_P_OP of it is equal to the sum of T_P_OP of its child nodes.

For example, if we select only one PE for the examples in Figure 9(a), then the amount of assigned parallel execution is 1-1 = 0. Therefore the execution relations of node A and CD are both sequential execution as indicated by "+" in Figure 9(c), and T_P_OP(A,1) equals to 1500, which is the sum of T_P_OPs of behavior B, C, and D, as displayed in Figure 9 (c). If two PEs are in the architecture, then we assign a parallel execution to node A. Since the amount of assigned parallel execution is 2-1 = 1 and a parallel execution has been assigned to node A, the execution relation of CD has to be sequential. In this case, T_P_OP(A,2) equals to MAX(T_OP(B), (T_OP(C) + T_OP(D))) =800. If three PEs are in the architecture, then the execution relations of node A and CD are both parallel execution and T_P_OP(A,3) = MAX((T_OP(B), MAX(T_OP(C), T_OP(D))) = 700 .

After T_P_OP is calculated, we compute PE_N to represent the gain of T_P_OP by adding one PE to the architecture. We use PE_N to evaluate the efficiency of parallelism.

PE_N can be defined as:

$$PE\_N(n,m) = \frac{T\_P\_OP(Design,n)}{T\_P\_OP(Design,m)}$$

	PE_N(2,1)	PE_N(3,2)	PE_N(4,3)
Main	0.53	0.88	1

Table 1: PE_N table for the example in Figure 9.

For example, PE_N(n+1, n) = 99% indicates that adding one new PE to n PE architecture cannot achieve more than 1% of the gain of T_P_OP, if the added PE is not faster than the slowest PE in architecture. The usage of PE_N is explained in section 4.

Table 1 is the PE_N table for the example in Figure 9. It indicates that no more than 3 PEs should be selected for the architecture.

3.2 Upperbound on PE Speed

With a given specification and a given time constraint, we can compute the million operations per second (MOPS) for behaviors. If the given time constraint for behavior A is TC(A), then

$$MOPS(A) = T\_OP_A / TC(A)$$

Where T_OP_A is described in section 2.2.2.

Each PE has a property MOP(PE) to denote the average million operations per second executed on this PE. For any Behavior A and PE x, if MOPS(A) ≤ MOPS(x), then behavior A can meet its time constraint when it is executed on PE x. Therefore, when we select PEs for the specification, the MOPS of selected PE must be greater than MOPS of behavior executed on that PE.

4 Specification-Architecture Mapping

Specification-architecture mapping maps the behaviors of the given specification into PEs of the selected architecture. The purpose of it is to find a mapping solution to meet the time constraint with low cost. In this section, we introduce the algorithm of architecture selection and specification-architecture mapping.

First, we introduce how to compute the cost and the execution time of implementation. Then we introduce an iterative improvement algorithm for selecting architecture as well as specification-architecture mapping.

4.1 Estimating Design Cost

Design cost contains two parts: material cost and processing cost.

design cost = material cost + processing cost

The design cost is evaluated by designers or other EDA tools. SpecC profiler cannot provide statistics for design cost.

4.1.1 Material Cost

The material cost of design is:

material cost = ΣPE cost + Σmemory cost

PE cost refers to production cost of processors or manufacturing cost of custom HW. Memory cost refers to production cost of memories. The costs of PE and memory are saved in PE library.

4.1.2 Processing Cost

Processing cost reflects the cost required during design process. Processing cost consists of:

a) The cost of hardware design.
b) The cost of communication design.

The cost of hardware design refers to the cost for designing started from functional specification to layout specification. The cost of hardware design is evaluated based on how many primitive behaviors for specification-architecture mapping are implemented in

hardware and how difficult it is to design these primitive behaviors on custom HW.

The cost for communication design contains the cost of bus and protocol design. It is evaluated based on whether transducer [2] should be inserted among PEs, how difficulty it is to develop transducers, and how difficult it is to design communication protocols.

4.2 Estimating Execution Time

We define the execution time of design as,

execution time = computation time + communication time

communication time = data transfer time + synchronization time

With SpecC profiler, we can coarsely estimate computation time and data transfer time. If behavior A is executed on PE x, then we first select the PE weight table representing PE x from PE library. Then, we use weighted operations $Total\_P_A(x)[1]$ to represent execution time of behavior A when it is executed on PE x. Similar, we use weighted communication $C\_D_A(x)[1]$ to estimate data transfer time for behavior A when behavior A communicates with other behaviors on different PEs through bus x. Total_P and C_D are described in section 5 of report [1].

SpecC profiler cannot estimate synchronization time. Designer should provide synchronization time among behaviors.

4.3 Specification-Architecture Mapping Algorithm

We design an iterative improvement algorithm for architecture selection and specification-architecture mapping.

Initially, all behaviors are mapped to a single-processor architecture. The MOPS of the processor in the architecture must be greater than the MOPS of design. If there are more than one processor that can satisfy above condition in the PE library, then the one with the lowest cost is selected. If the cost of selected processor is smaller than the cost constraint, then mapping process is finished and system architecture and specification-architecture mapping solution are found. Otherwise, if the cost of selected processor is greater than the cost constraint, then architecture selection and specification-architecture mapping should be redone. In this report, we don't consider the case that the cost constraint is smaller than any processor cost.

If there is no processor whose MOPS is greater than the MOPS of design, then the iterative improvement algorithm will be performed.

During the process of the iterative improvement, we can apply four specification-architecture-mapping actions:

a) Reduce the computation time of a primitive behavior by moving this primitive behavior from slower PE to faster PE.
b) Schedule sequential executed primitive behaviors to run them in parallel on different PEs if feasible.
c) Reduce data transfer time by changing communication protocols.
d) Re-schedule behaviors to reduce the synchronization time.

In this report, we only concern the first three actions.

In 4.3.1, we first introduce the performance gain of specification-architecture-mapping actions. In 4.3.2, we describe the cost-adding of the actions. In 4.33, we introduce performance gain/cost-adding ratio to characterize each mapping action. Finally, in 4.4.4, we give the iterative improvement algorithm.

4.3.1 Performance Gain Of Specification-Architecture Mapping Actions

4.3.1.1 Performance Gain Of Moving Behavior to Faster PE

If we move behavior A from PE x to PE y, then the Performance-gain(PG) is:

$$PG(MOV, A, x, y) = Total\_P_A(x) + C\_D_A(x) - Total\_P_A(y) - C\_D_A(y)$$

Above equation is valid only for the behaviors that are not run in parallel with any other behaviors.

If behavior A run on PE x is executed in parallel with another behavior B run on PE z, and there are no other behaviors run in parallel with A on PE y, then

$$PG(MOV, A, x, y) = Total\_P_A(x) + C\_D_A(x) - Max(Total\_P_B(z) + C\_D_B(z), Total\_P_A(y) + C\_D_A(y));$$

If behavior A run on PE x is executed in parallel with another behavior B run on PE y, then

$$PG(MOV, A, x, y) = Total\_P_A(x) + C\_D_A(x) - (Total\_P_B(y) + C\_D_B(y) + Total\_P_A(y) + C\_D_A(y));$$

4.3.1.2 Performance Gain Of Mapping Behaviors for Parallel Execution

If there are two behaviors A and B executed on PE x, and behavior A and B can be run in parallel on PE x and PE y, then the execution time may be improved by mapping behavior A and behavior B to different PE, ex. PE x and PE y, and running behavior A and B in parallel. If behaviors A and B are on PE x at the beginning, and there are no behaviors run in parallel with A and B on PE y, then

$$PG(PAR, A, B, x, y) = Total\_P_A(x) + C\_D_A(x) + Total\_P_B(x) + C\_D_B(x) - Min(Max(Total\_P_A(x) + C\_D_A(x), Total\_P_B(y) + C\_D_B(y)) , Max(Total\_P_B(x) + C\_D_B(x), Total\_P_A(y) + C\_D_A(y)))$$

If there are some behaviors run in parallel with A and B on PE y already, then PG(PAR, A, B, x, y) should be computed case by case.

4.3.1.3 Performance Gain Of Changing Communication Protocols

If a slow bus/protocol x is replaced by a faster bus/protocol y, then the performance gain will be:

$$PG(BUS, x, y) = \sum_i (C\_D_i(x) - C\_D_i(y))$$

i refers to the behaviors that using bus/protocol x for communication.

4.3.2 Cost-Adding Of Specification-Architecture Mapping Actions

Besides computing performance gain for each specification-architecture-mapping action, designers should also compute the cost adding for each specification-architecture-mapping action., based on the description in 4.1.

4.3.2.1 Cost-Adding Of Moving Behavior to Faster PE

If behavior A is moved from processor PE1 to custom HW PE2, then the processing cost of A for PE2 should be added to cost-adding.

If before moving, PE2 is not in the architecture, then the material cost of PE2, the material cost of communication (transducer), and the processing cost of communication are added to the cost-adding.

4.3.2.2 Cost-Adding Of Mapping Behaviors for Parallel Execution

Before mapping, both behavior A and B are executed on PE1. After mapping, behavior A and B are run in parallel on PE1 and PE2, respectively.

If before mapping, PE2 is not in the architecture, then the material cost of PE2, the material cost of communication (transducer), and the processing cost of communication are added to the cost-adding.

If PE2 is custom HW, then the processing cost of PE2 for behavior B is added to the cost-adding.

If PE1 is custom HW, then the processing cost of PE2 for behavior B is subtracted from the cost-adding.

4.3.2.3 Cost-Adding Of Changing Communication Protocols

If changing communication protocols will add transducer to system, then the material cost for transducer is added to the cost-adding.

If changing communication protocols will remove transducer from system, then the

material cost for transducer is subtracted from the cost-adding.

The processing cost for new protocol is added to the cost-adding. The processing cost for old protocol is subtracted from the cost-adding.

4.3.3 Performance-Gain / Cost-Adding Ratio

The performance-gain/cost-adding ratio(TCR) for each specification-architecture mapping action reflects the efficiency of mapping actions. For architecture mapping action i,

TCR(i) = PG(i) / Cost_adding(i).

where PG(i) is the performance gain of action i. Cost_adding(i) is the cost adding of action i.

If TCR(i) is greater than TCR(j), then action i will get more performance-gain at the same cost.

4.3.4 Iterative Improvement Algorithm

If the initial single-processor design described at the beginning of 4.3 cannot meet the given design constraint, then multi-PE architecture must be selected. There are two different types of architecture: multi-processor architecture, and processor-custom HW architecture. The multi-processor architecture don't need the implementation of custom HW, while the processor-custom HW architecture has better performance. In this subsection, we first test whether multi-processor architecture design or HW-SW co-design can produce an implementation satisfying the given constraint. Then starting from the initial single-processor design, we use an iterative improvement algorithm to produce the implementation.

4.3.4.1 Multi-Processor Architecture Design

There are three tasks in multi-processor architecture design: selecting PE type, amount of PE, and mapping behaviors to PE. We select the processor with the fastest speed called PE1 out of PE library as the processor type. We select the amount of PE n according to following two inequations,

$$PE\_N(n,1) < \frac{Time\_Constra\,\text{int}}{Total\_P_{Main}(PE1)}$$

and $$n < \frac{Cost\_constra\,\text{int}}{Cost(PE1)}$$

If there is a result of n satisfying above two inequations, then n-PE1 architecture can meet the given time constraint and the given cost constraint. In this case, we select the smallest number of n satisfying the inequations as the amount of PE allocated in the architecture. Specification-architecture mapping will be implemented by performing our iterative improvement algorithm. Otherwise, n-PE1 architecture will not meet the design constraints. Therefore, HW-SW co-design must be implemented.

4.3.4.2 HW-SW Co-Design

Another solution is HW-SW co-design. HW-SW co-design is more complex than multi-processors architecture design because the PE types allocated in the architecture are different. If a custom HW called HW is the fastest PE in the PE library and there is an integer n that satisfy following inequation,

$$PE\_N(n,1) < \frac{Time\_Constra\,\text{int}}{Total\_P_{Main}(HW)}$$

then mapping implementation on n-HW architecture can meet the given time constraint. Therefore, n-PE architecture can meet the time constraint. Otherwise, a faster custom HW has to be chosen as PE type.

Unlike multi-processor architecture design that have chosen PE type and amount of PE before behavior-architecture mapping, during the HW-SW co-design, we decide the PE type, amount of PEs in the architecture and behavior-architecture mapping at the same time during the process of our iterative-improvement algorithm.

4.3.4.3 Algorithm Description

Figure 10 describes our iterative improvement algorithm. First, we select the fastest processor out of PE library as the unique component of the single-PE architecture. We map all the behaviors to this architecture. If the initial implementation meet the given time constraint and cost constraint, then the initial implementation is final solution. Otherwise, if the cost constraint is not meet, then we replace the processor in the architecture with a processor with lower cost and with lower performance

in the library. We produce new implementation by mapping all behaviors to this processor. The processor replacing will be continued until the new implementation meet the cost constraint and time constraint. If no implementation of mapping all the behaviors to one processor can meet both the time and cost constraints, then no implementation can be found based on our PE library and specification. This process is completed by function *Init_Solution*.

If the initial implementation does not meet the time constraint, then *Update_TCR* will compute the performance gain, cost_addiing, and performance_gain/cost_adding ratio(TCR) of each behavior-architecture mapping action for each behavior. If designers want implementing multiple-processor architecture design, then next possible PE in the architecture is always the fastest processor in the library. If designers want implementing SW-HW co-design, then next possible PE in the architecture can be all of the PEs in the PE library. Designers also can limit the range of PE selection when they compute for mapping actions. With the statistics for each mapping action, *Action_Selection* selects the action with the greatest TCR as the next mapping action. If the selected mapping action add new PE to the architecture, then *Update_PE_List* add this PE to PE list *Selected_PE*. *Apply_Action* updates the current architecture and behavior-architecture mapping solution according to the selected action. Finally, algorithm compares the cost of updated implementation with the given cost constraint and compares the execution time of updated implementation with the given time constraint. The process of *Action_Selection, Update_PE_List, and Apply_Action* will be repeated until the implementation meets the given time constraint and the given cost constraint, or no implementation meeting design constraint can be found at all. Design examples are described in section 5, 6, and 7.

```
Algorithm Arachitecture_Exploration

        Init_Design = Init_Solution();
        if (T(Init_Design) > T(constraint) ) do
                Selected_PE = {Fastest_processor};
                 Design = Init_Design;
                 Update_TCR(Design);
                  while ( T(Design) > T(constraint) ) do
                            Action = Action_selection(Design);
                            Update_PE_List(Selected_PE);
                            Design = Apply_Action(Design, Action);
                            Update_TCR(Design);
                           if (Cost(Design) > Cost(constraint) ) do
                              There is no solution.
                              exit();
                          endif
                  endwhile
        else
                  if ( Cost(Design) < Cost(constraint)) do
                            Design = Init_Design
                  else
                            Try_SlowerPE(Design);
                  endif
        endif
}
```

Figure 10: The iterative improvement algorithm

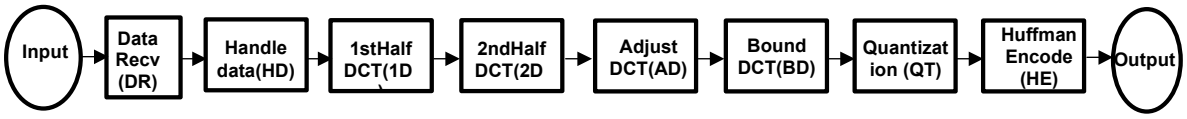


Figure 11: The block diagram of JPEG encoder.

5 Design Experience -- JPEG Encoder

5.1 JPEG Encoder

5.1.1 Block Diagram

JPEG [9] is an image compression standard. It is designed for compressing either full-color or grey-scale images of natural scenes. Figure 11 shows the block diagram of the DCT based encoder for a gray scale image.

5.1.2 Testbench

We use a bitmap (bmp) file containing 180 8×8 pixel blocks as the input of design. The given time constraint for this input is 90ms, which is denoted as TC(Design).

5.2 PE Types and Bus Protocols

There are two types of PE in the PE library:

a) Motorola ColdFire processor
b) Custom HW

We use two weight tables to represent both PEs. The weight table contains computation weights for different operation types and data types representing the executed clock cycles on corresponding PE. Both of the weight tables also contain the communication weights for data types, which represent the required executed clock cycles of the normal model of Coldfire master bus protocol[4]. In this protocol, the transfer time for 4 bytes is 2 clock cycles. Weight tables also contains the memory sizes for data types. The frequency of both ColdFire and custom HW is 66 MHz.

We compute MOPS for ColdFire and custom HW. Since the executed clock cycles of most operation types are between 1 and 9 clock cycles. Thus, the lowerbound of MOPS(ColdFire) = 66/9 = 7.3, the upperbound on MOPS(ColdFire) = 66/1 = 66.

The MOPS of custom HW is more difficult to compute from the custom HW weight table. In custom HW's weight table, more than one operation can be performed in one clock cycle. Therefore, the weights for some operation types are 0 when we set the weights for the operations executed in the same clock cycle with them as 1. Since the required executed clock cycles for most operation types of HW are between 0 and 9 clock cycles, we can have the lowerbound of MOPS(HW), which is 7.3.

5.3 Specification Modeling

5.3.1 Exploring Parallelism

Figure 12 shows the parallelism of JPEG encoder behaviors. Two sets of parallel behaviors are:

a) Behavior DR and encodeStripe are executed in a pipeline style.
b) Behavior HD, 1D, 2D, AD, BD, QT and HE are executed in a pipeline style.

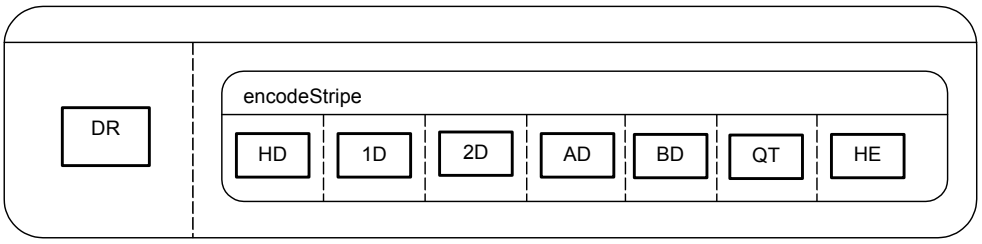
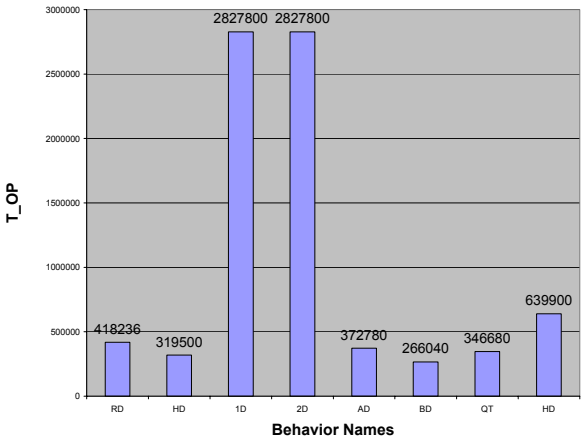
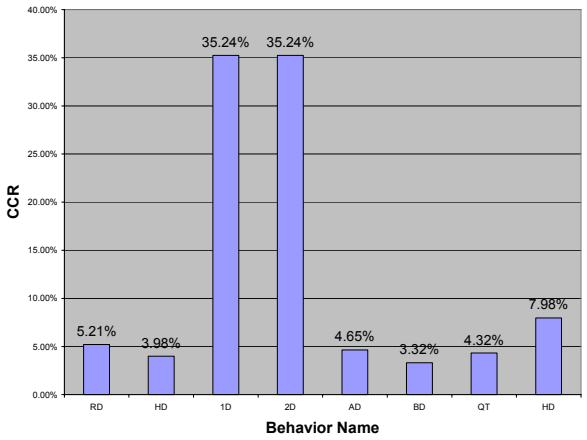


Figure 12 : Parallelism of JPEG encoder behaviors.



(a) Total execution number of operations of behaviors in JPEG encoders



(b) The granularity of behaviors in JPEG encoder

Figure 13: T_OP and granularity of behaviors in JPEG encoder.

SpecC profiler demonstrates traffic between behaviors, which indicates whether behaviors can be executed in a parallel/pipeline model, or not. For example, the result of SpecC profiler shows that the traffic only exists from HD to 1D, from 1D to 2D, from 2D to AD, from AD to BD, from DB to QT, and from QT to HE. There is no traffic from HE to HD. Furthermore, above behaviors are called sequentially in one loop. Thus, according to section 2.1, behavior HD, 1D, 2D, AD, BD, QT, and HE can be run in a pipeline style.

5.3.2 Choosing Granularity

The leaf behaviors in JPEG encoder reflect algorithm blocks. The total numbers of execution of operations of leaf behaviors(T_OP) are displayed in Figure 13(a). The granularity G() of behaviors are displayed in Figure 13(b).

Note that T_OP or G() of leaf behavior 1D and 2D are much larger than other leaf behaviors, as shown in Figure 13. As described in 2.2.4, whether we should decompose 1D and 2D for producing new leaf behaviors with finer granularity depends on the computation and communication overhead. Note that both behaviors 1D and 2D perform matrix multiplication. For example, Figure 14 is the specification model of behavior 1D. Obviously, decomposing matrix multiplication to two or more independent leaf behaviors will produce heavy computation and communication overhead. As a result, we use 1D and 2D as primitive behaviors for mapping, even though their granularity is much larger than other behaviors.

```
behavior 1D(in int x[BLOCKSIZE], out int y[BLOCKSIZE]){
      void main(){
            int i, j, k;
            for( i = 0; i<BLOCKROW; i++) {
              for ( j = 0; j< BLOCKROW; j++) {
                y[ i*BLOCKROW + j] = 0;

                for( k = 0; k< BLOCKROW ; k++)
                  y[ i*BLOCKROW + j] += x[ i* BLOCKROW + k] * COSBlock[
k*BLOCKROW + j];
          }
      }

      }
};
```

Figure 14: The specification model of behavior 1D

5.4 Architecture Exploration -- PE Selection

5.4.1 Selecting PE Speed

SpecC profiler provides the total number of execution of operations of JPEG design T_OP(Design), which is 8023749. According to section 3.2,

MOPS = T_OP(Design) / TC(Design) = 8023749 / 90ms = 89 MOPS

Where TC(Design) is the time constraint of the design. Since the upperbound on MOPS(ColdFire) is 66, a single-ColdFire architecture cannot meet the given time constraint.

5.4.2 Selecting Amount of PEs

Table 2 displays the parallel efficiency PE_Ns of JPEG. It indicates that implementing JPEG encoder in 4 ColdFires instead of 3 ColdFires cannot achieve any performance gain. Thus, the upperbound on PE Amount in architecture is 3.

PE_N(2,1)	PE_N(3,2)	PE_N(4,3)
0.6476	0.5448	1

Table 2 : PE_N for JPEG encoder

5.5 Specification-Architecture Mapping

5.5.1 Estimating Design Cost

In our example, we use 5$ as one automatic cost unit.

Assuming that 1000 pieces of chips will be produced, the design material cost of ColdFire is estimated as 15$ (3 unit)[7], the design material cost of custom HW is 10$ (2 unit)[8].

For each behavior, the design processing cost for ColdFire is 0, the design processing cost for custom HW is 5$(1 unit). Design process cost for communication is 5$(1 unit).

We assume that cost constraint of design is 50$(10 unit).

5.5.2 Estimating Execution Time

5.5.2.1 Estimating Computation Time

	ColdFire(ms)	Custom HW(ms)	Performance_gain(ms)
RD	28.00873	3.007273	25.00145
1D	199.1536	21.03818	178.1155
2D	199.1536	21.03818	178.1155
AD(ms)	18.68455	4.701818	13.98273
BD	7.022727	1.052727	5.97
QT	16.44818	4.02	12.42818
HE	18.67323	2.389064	16.28416

Table 3 : Estimated computation time of leaf behaviors of JPEG encoder.

	1D(ms)	2D(ms)	AD(ms)	BD(ms)	QT(ms)	HE(ms)
RD	0.000349					
1D		0.002793				
2D			0.000349			
AD				0.000349		
BD					0.000349	
QT						0.000349

Table 4: Estimated communication time between leaf behaviors of JPEG encoder.

Table 3 displays the estimated computation time of leaf behaviors of JPEG encoder. Column 2 represents the estimated execution time of behaviors for ColdFire. Column 3 represents the estimated execution time of behaviors for custom HW. Column 4 is the difference between column 2 and 3, which represents the performance_gain achieved by moving behavior from ColdFire to custom HW, without considering communication.

SpecC profiler compute the computation time of behaviors in terms of clock cycle when behaviors are mapped to either ColdFire or custom HW,. The execution time displayed in column 2 and 3 is achieved by dividing total clock cycles by 66M(Hz).

5.5.2.2 Estimating Communication Time

Table 4 displays the estimated communication time between leaf behaviors. SpecC profiler produces total communication between behavior instances in terms of clock cycles. The communication time is achieved by dividing total clock cycles by 66M(Hz).

One exception is that the communication time between RD and 1D is computed manually instead of using SpecC profiler. This is because the data is transferred by address reference. SpecC profiler cannot compute this type of communication.

5.5.3 Specification-Architecture Mapping Process

5.5.3.1 Initial Design: Pure ColdFire Design

We start our design with pure ColdFire solution. We use single ColdFire processor as the system architecture. SpecC profiler indicates that the execution time is 487 ms.

The design cost is equal to the material cost of ColdFire, which is 3 cost units,. The behavior display is shown in Figure 15.

5.5.3.2 Specification-Architecture Mapping for Multi-Processor Architecture

For JPEG encoder, the upperbound on the amount of PE is 3. PE_N(3,1) = PE_N(2,1) * PE_N(3,2) = 0.3528. Thus, the execution time for 3-ColdFire architecture is equal to Total_P(Initial Design) * PE_N, which is 171ms, without considering communication. Compared it with the given time constraint of 90ms, multi-ColdFire solution cannot meet the time requirement. Therefore, SW-HW co-design is required.

Mapping_Action	Performance_Gain	Cost_Adding	TCR
MOV, RD, SW,HW	25.00	4	6.25
MOV, 1D, SW,HW	178.11	4	44.53
MOV, 2D, SW,HW	178.11	4	44.53
MOV, AD, SW,HW	13.98	4	3.50
MOV, BD, SW,HW	5.97	4	1.49
MOV, QT, SW,HW	12.43	4	3.11
MOV, HE, SW,HW	16.28	4	4.07
PIPE 1D,2D,SW,HW	**199.15**	**4**	**49.79**

Table 5: Table of performance_gain, cost_adding, and TCR for improved solution 1.

Mapping Action	Performance Gain	Cost_Adding	TCR
MOV, RD, SW,HW	25.00	1	25.00
MOV, 2D, SW,HW	**157.0**	**1**	**157.0**
MOV, AD, SW,HW	13.98	1	13.98
MOV, BD, SW,HW	5.97	1	5.97
MOV, QT, SW,HW	12.43	1	12.43
MOV, HE, SW,HW	16.28	1	16.28

Table 6: Table of performance_Gain, cost_adding, and TCR for improved solution 2.

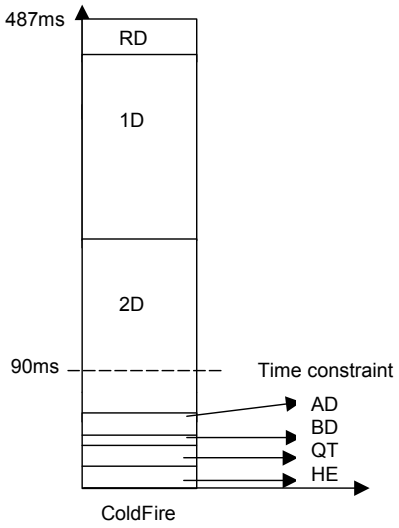


Figure 15: Behavior display of the initial design of JPEG encoder.

5.5.3.3 Improved Solution 1: ColdFire-Custom HW(1D) Design

Table 5 displays the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action. Performance_gain is calculated based on data in 5.5.2.1 and 0, and based on equations in section 4.3.1. Note that there is PIPE mapping action. PIPE mapping action is special type of PAR mapping action. The performance_gain of PIPE is computed following the equations for PAR action. Then the adjustment for it is performed based on case by case. Cost_adding for all actions is 4, which includes 1 cost unit for the processing cost of custom HW, 2 cost units for the material cost of custom HW, and 1 cost unit for the processing cost of communication.

Among those actions, the action of moving behavior 1D to custom HW from ColdFire and to run 1D and 2D in a pipeline style gives the biggest TCR, which is 49.79. It is displayed as the bold row in table 5. We select this action and update the architecture and specification-architecture mapping solution.

Improved solution 1 maps 1D to custom HW and maps other behaviors to Coldfile, and makes 1D and 2D run in parallel. The estimated design time is 288 ms, as shown in Figure 17. The estimated cost is 7. The shaded area indicates that the PE status is idle. In Figure 16, how the behaviors are executed in a pipeline style is illustrated.

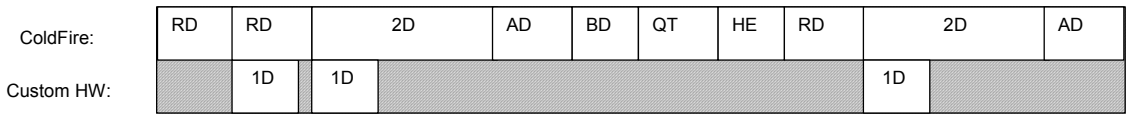


Figure 16: Display of pipeline execution of behaviors for improved solution 1.

Mapping action	Performance gain	Cost_adding	TCR
MOV, RD, SW,HW	25.00	1	25.00
MOV, AD, SW,HW	13.98	1	13.98
MOV, BD,SW,HW	5.97	1	5.97
MOV, QT, SW,HW	12.43	1	12.43
MOV, HE, SW,HW	16.28	1	16.28
PIPE, RD, 1D, SW, HW	**20.88**	**0**	**100000**
PIPE, AD, 1D, SW, HW	18.58	0	100000
…			

Table 7: Table of performance_gain, cost_adding, and TCR for improved solution 3.

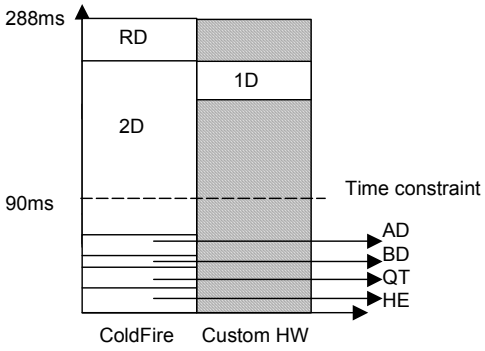


Figure 17: Behavior display of improved solution 1.

5.5.3.4 Improved Solution 2: ColdFire-Custom HW(1D, 2D) Design

Since the improved solution 1 cannot meet the time constraint, more specification-architecture mapping action should be applied.

Table 6 describes the updated Performance_Gain, Cost_Adding, and TCR of mapping actions. 1 cost unit comes from processing cost of leaf behavior for custom HW. Performance_Gain is computed in the similar way as described in section 5.5.3.3.

Among those actions, the action of moving 2D from ColdFire to custom HW gives the greatest TCR, which is 157, which is displayed in bold row. We select this action and update the architecture and specification-architecture mapping solution.

Improved solution 2 maps 1D and 2D to custom HW and maps other behaviors to Coldfire. The estimated execution time is 131 ms which is shown in Figure 18. The estimated cost is 8.

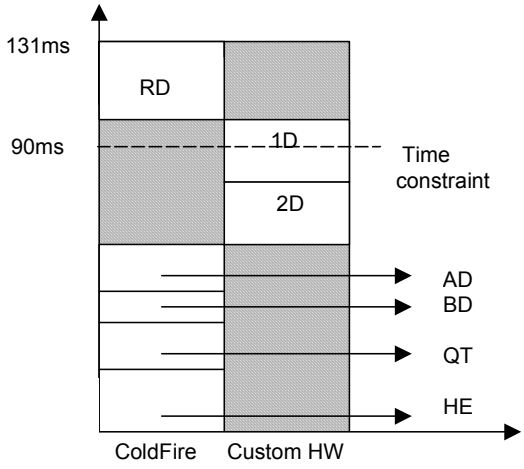


Figure 18: Behavior display of improved solution 2.

5.5.3.5 Improved Solution 3: ColdFire-Custom HW(1D, 2D) Design

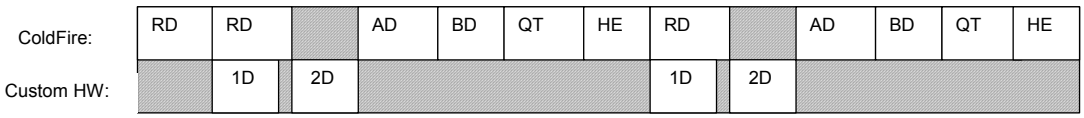


Figure 19: Display of pipeline execution of behaviors for improved solution 3.

Mapping action	Performance gain	Cost_adding	TCR
MOV, RD, SW,HW	4.12	1	4.12
MOV, AD, SW,HW	13.98	1	13.98
MOV, BD, SW,HW	5.97	1	5.97
MOV, QT, SW,HW	12.43	1	12.43
MOV, HE, SW,HW	16.28	1	16.28
PIPE, HE, 2D,SW, HW	16.56	0	100000
PIPE, AD, 2D, SW, HW	18.58	0	100000
PIPE, (RD +AD), (1D +2D) , SW, HW	**20.88**	**0**	**100000**

Table 8 : Table of performance_gain, cost_adding and TCR for improved solution 4.

Since the improved solution 2 cannot meet the time constraint, more specification-architecture mapping action should be applied.

Table 7 describes the updated Performance_Gain, Cost_Adding, and TCR for mapping actions. 1 cost unit comes from processing cost of leaf behavior for custom HW. Since the cost adding for parallel execution is 0, we assign 10000 as the value of TCR to represent the upperbound on Performance Gain. Performance_Gain is computed in the similar way as described in section 5.5.3.3.

Among actions in Table 7, the action of pipelining 1D in custom HW and RD in ColdFire gives the greatest TCR, since the cost_adding is 0. We select this action and update the architecture and specification-architecture mapping solutions.

Improved solution 3 maps 1D and 2D to custom HW, and executes 1D and RD in a pipeline style. The estimated execution time is 110.12ms, as indicated in Figure 20. The estimated cost is 8. Figure 19 illustrates how the behaviors are executed in a pipeline style.

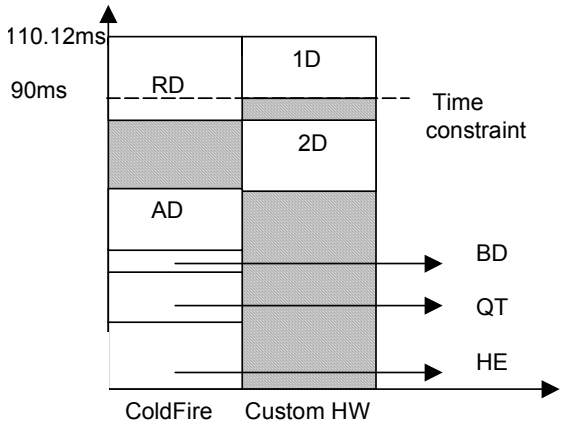


Figure 20: Behavior display of improved solution 3.

5.5.4 Improved Solution 4: ColdFire-Custom HW(1D, 2D) Design

Since the improved solution 3 cannot meet the time constraint, more specification-architecture mapping action should be applied.

Table 8 describes the updated Performance_Gain, Cost_Adding, and TCR for mapping actions. 1 cost unit comes from processing cost of leaf behavior for custom HW.

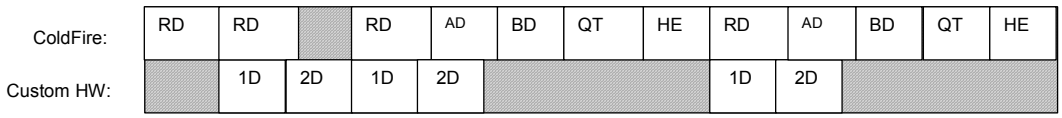


Figure 21: Display of pipeline execution of behaviors for improved solution 4.

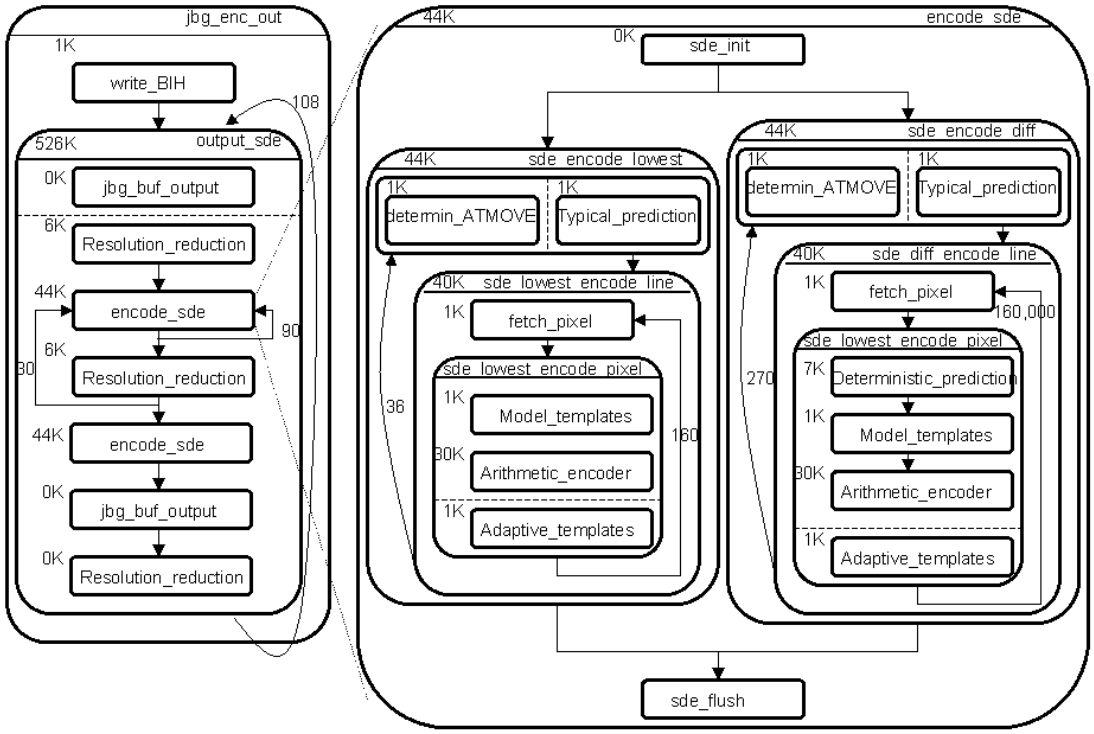


Figure 22: JBIG encoder block diagram.

Among those actions, the action of parallelizing (1D+2D) in custom HW and (RD +AD) in ColdFire gives the greatest TCR, since the cost_adding is 0. We select this action and update the architecture and specification-architecture mapping solutions.

Improved solution 4 maps 1D and 2D to custom HW, maps other behaviors to Coldfire, and executes (1D + 2D) and (RD+AD) in a pipeline style. The estimated design time is 89.24ms, as indicated in Figure 23. The estimated cost is 8. In Figure 21, how the behaviors are executed in a pipeline sytle is illustrated.

Improved solution 4 is the solution that meet the time requirement with the lowest design lost. The process of system level design is completed.

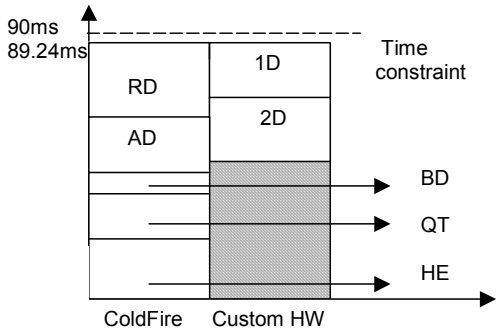


Figure 23: Behavior display of improved solution 4.

6 Design Experience -- JBIG Encoder

6.1 JBIG Encoder

6.1.1 Introduction and Block Diagram

JBIG[10] stands for "Joint Bi-level Image experts Group". JBIG is one of the image compression/decompression standards.

Specifically, JBIG is a lossless progressive encoding of a bi-level image (an image that has only two colors, like black-and-white). It is lossless because the decoded image is identical to the original without any distortion. The progressive capability enables the transmission of the image with different resolutions over networks.

The block diagram of JPIG encoder is specified in Figure 22.

6.1.2 Testbench

A pbm file with 108-layer [10] is selected as the input of design. The time constraint is 1.5s. The cost constraint is 20 cost units.

6.2 PE Types and Bus Protocols

PE Types and Bus protocols are the same as JPEG encoder example, which is described in 5.2.

6.3 Specification Modeling

6.3.1 Exploring Parallelism

There are three sets of parallel behaviors:

a) In behavior *output_sde, jpg_bug_out* is parallel executed with others child behaviors of output_sde.

b) In behavior *sde_encode_lowest* and behavior *sde_encode_diff*, behavior *determine_ATMOVE* and *Typical_prediction* are run parallel.

c) In behavior *sde_lowest_encode_pixel* and *sde_diff_encode_pixel*, *adaptive_template* and (*deterministic_prediction* + *model_templates* + *arith_encode*) are run parallel.

We get above parallelism information by analyzing the algorithm

6.3.2 Choosing Granularity

Compared with JPEG encoder specification, there are two problems in the JBIG encoder specification. The first one is that JBIG encoder specification is not a clean model. The second one is that some non-leaf behaviors are quite similar: they share large percentage of functionality but only contain small percentage of different functionality. These two problems and their corresponding solutions will be described in section 6.3.2.1 and section 6.3.2.2.

6.3.2.1 Pseudo Behavior for Non-Clean Specification

JBIG encoder is not a clean specification: non-leaf behaviors not only contain child behavior instance calls, but also contain direct statement computation such as addition.

For example, *sde_diff_encode_line* is a non-clean behavior, whose granularity is 86%. As shown in Figure 24. 38.88% of its computation is original from execution of direct statements. Other 47.22% of its computation is original from execution of statements in its child behaviors.

However, making behavior *sde_diff_encode_line* clean is time-consuming work. There are four levels of hierarchical loops in *sde_diff_encode_line*. All the child behavior instance calls of *sde_diff_encode_line* are distributed in these loops. Therefore, we don't change the specification to a clean model.

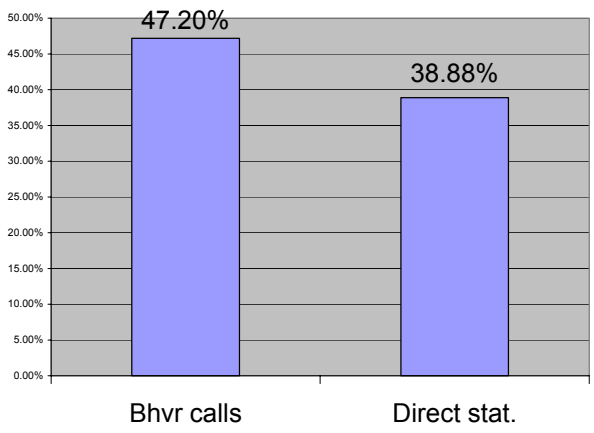


Figure 24: GRANULARITY distribution of behavior sde_diff_encode_line.

Because JPIG encoder is a non-clean specification, it is difficult to determine the primitive behaviors for mapping. To make the problem solved, we group all the direct statements in non-clean behavior and treat is as a pseudo primitive behavior. This pseudo behavior does not really exist in the specification but we can use the sum of statistics of the direct statement as the statistics of the pseudo behavior.. For example, in behavior *sde_diff_encode_line*, the granularity of pseudo primitive behavior *sde_diff_encode_line_stat* is 38.88%.

Using pseudo behaviors for computation operations in non-clean behavior, the correctness of specification-architecture mapping is guaranteed without revising specification.

6.3.2.2 Pseudo Behavior for Behavior Merging

JBIG supports progressive encoding. Designer can choose either 1 layer model or multi-layer model[10].

In JPIG encoder, there are two different non-leaf behaviors to implement above two models. Behavior *sde_encode_lowest* implements the functionality of 1 layer encoding. Behavior *sde_encode_diff* implements the functionality for multi-layer model. For the same input, the granularity of behavior *sde_encode_lowest* and *behavior sde_encode_diff* are different for different selected encoding models. If encoding model is 1 layer model, G(*sde_encode_lowest*) is 83%, and G(*sde_encode_diff*) is 0. However, if encoding model is 5 layer model, G(*sde_encode_lowest*) is 0.09%, G(*sde_encode_diff*) is 91.89%. Since both behavior *sde_encode_lowest* and *sde_encode_diff* may take large percentage of the whole design, how to implement these two behaviors will heavily influence the performance and the cost of the design.

By analyzing the specification, we find that both *sde_encode_diff* and *sde_encode_lowest* are "similar" because most of their child behavior instance calls are the same: both of them have the instantiations of child behavior *arith_encode, model_templates, determine_ATMOVE,* and *adaptive_template*. On the other hand, they also have some difference. For example, behavior *sde_encode_diff* calls behavior *deterministic_prediction* while *sde_encode_lowest* does not.

Because the importance and similarity of both behavior *sde_encode_diff* and *sde_encode_lowest*, we believe merging two behaviors to one can save processing cost of HW by avoiding double work, when they are mapped to custom HW. Furthermore, the behavior merging ensures that specification-architecture mapping for both *sde_encode_diff* and *sde_encode_lowest* are the same, which is reasonable from the view of algorithm.

However, merging behavior *sde_encode_diff* and *sde_encode_lowest* to one is difficult because both of them are non-clean behaviors. To bypass this difficulty, we didn't really merge them but group them as one pseudo behavior "*sde_encode*", which is used as a primitive behavior for architecture selection and specification-architecture mapping. Since SpecC profiler provides statistics for behavior *sde_encode_diff* and *sde_encode_lowest*, we can estimate the statistics of the pseudo behavior sde_encode by adding up the statistics of *sde_encode_diff* and *sde_encode_lowest,* even though the behavior *sde_encode* does not really exist.

After specification-architecture mapping, if *sde_encode* is mapped to ColdFire, then behavior merging is not necessary because ColdFire's operation system can directly

execute *sde_encode_lowest* or *sde_encode_diff* without any processing cost. However, if pseudo behavior *sde_encode* is mapped to custom HW, then we will make behavior merging.

Using pseudo behaviors for behavior merging, the correctness of specification-architecture mapping is guaranteed without revising specification.

6.3.2.3 Choosing Granularity

The total number of execution of operations and granularity of leaf behaviors of JBIG encoder are displayed in Figure 25 and Figure 26. The granularity of leaf behaviors are various from 0.09% to 39.03%. *Sde_encode_stat* represents the direct statement computation in the pesudo behavior *sde_encode* mentioned in section 6.3.2.2.

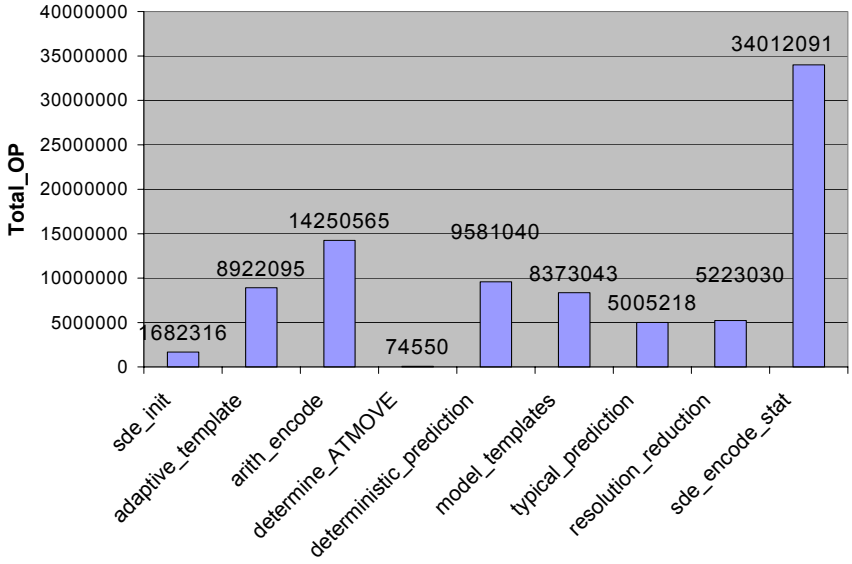


Figure 25: The total number of execution of operations of primitive behaviors in JBIG encoder.

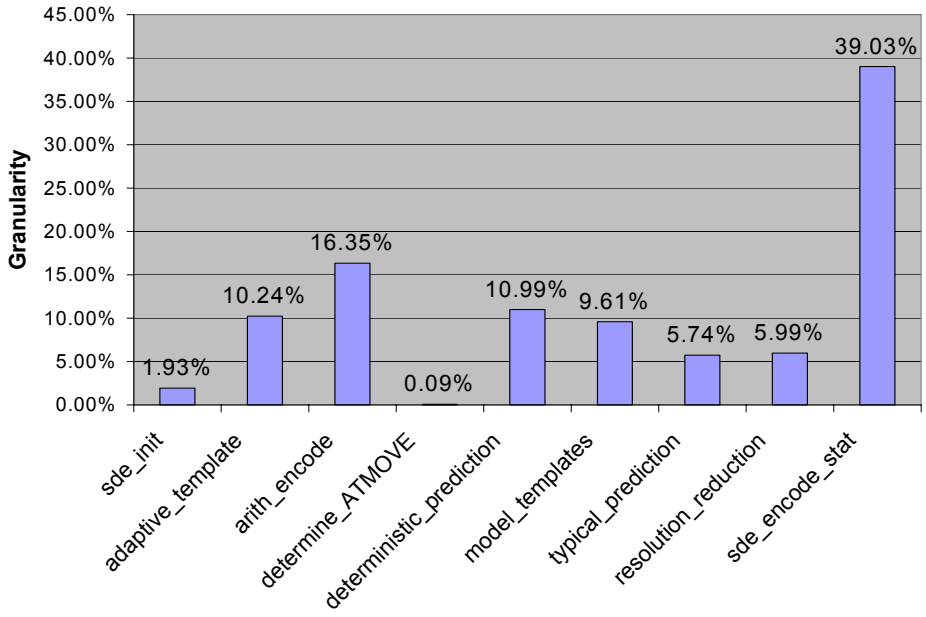


Figure 26: granularity of primitive behaviors in JBIG encoder.

Note that the granularity of Sde_encode_stat is too large compared with other behaviors. Therefore, we should first decide whether decomposing behavior *sde_encode_stat* is necessary. One solution of decomposing is to group a number of direct operations to form some new leaf behaviors. However, the direct statements in *sde_encode_stat* are tightly coupled with its child behavior instance calls. There will be more than 20% communication overhead added if we generate new leaf behaviors for direct statements. An alternative solution is to merge the statements with their neighbor child behavior instance calls in the same basic block. This solution is also not feasible. We found that around 5% of granularity in *sde_encode_stat* is caused by parameter assignment of behavior instance call, which must be specified out of behavior instance calls. Furthermore, other 34% granularity in *sde_encode_stat* is not in the same block with any child behavior instance calls. Therefore, we cannot merge direct statements to *sde_encode_stat*'s child behaviors instance calls.

Because both solutions don't work, we still leave *sde_encode_stat* as one primitive behavior for specification-architecture mapping.

6.4 Architecture Exploration -- PE Selection

6.4.1 Selecting PE Speed

SpecC profiler compute that the total number of execution of operations for Design (T_OP(Design)) is 87141176. Thus, based on section 3.2,

MOPS = T_OP(Design) / TC(Design) = 87141176 / 3 = 29 MOPS

Since the upperbound on MOPS(ColdFire) is 66, we cannot decide whether pure ColdFire can meet the time constraint. In this case, we need to estimate the performance of design and directly compare it with the given time constraint.

6.4.2 Selecting Amount of PEs

PE_Ns for JPEG encoder are as shown in Table 9. Table 9 indicates that implementing JBIG encoder in 3 ColdFire instead of 2 ColdFire cannot achieve any performance gain. Thus, the upperbound on amount of PE in architecture is 2.

PE_N(2,1)	PE_N(3,2)
0.8976	1

Table 9: PE_N for JBIG encoder.

6.5 Specification-Architecture Mapping

6.5.1 Estimating Design Cost

The cost assumption in JBIG encoder is the same at in JPEG encoder, which is displayed in section 5.5.1

Since the statements in pseudo behavior *sde_encode_state* is around 5 times as great as the statements in other leaf behaviors, the processing cost of *sde_encode_stat* for custom HW is 5 cost units.

6.5.2 Estimating Execution Time

6.5.2.1 Estimation Computation Time

Table 10 displays the estimated execution time for leaf behaviors of JBIG encoder. The second column represents the abbreviation names of the leaf behaviors. The third column represents the estimated execution time of behaviors for ColdFire. The fourth column represents the estimated execution time of behaviors for custom HW. The fifth column indicates the difference between columns 3 and column 4, which represents the performance_gain of moving behavior from ColdFire to custom HW without considering communication.

SpecC profiler computes the computation time for ColdFire and custom HW in terms of clock cycles. The execution times displayed in column 3 and 4 are computed by dividing total clock cycles by 66M(Hz).

6.5.2.2 Estimation Communication Time

	Abbreviation	ColdFire(s)	CustomHW(s)	Performance_gain(s)
sde_init		0.056529	0.007129	0.0494
adaptive_template	AT	0.451013	0.072188	0.378825
arith_encode	AE	0.532604	0.059325	0.473279
determine_ATMOVE	ATMOVE	0.003546	0.000611	0.002935
deterministic_prediction	DP	0.453648	0.09102	0.362628
model_templates	MT	0.385605	0.078989	0.306616
Typical_prediction	TP	0.215551	0.036481	0.179071
resolution_reduction	RR	0.270361	0.048544	0.221817
sde_encode_stat	SES	1.373767	0.217014	1.156753

Table 10: Estimated execution time for leaf behaviors of JBIG encode

	sde_encode_stat
adaptive_template	<0.30s
arith_encode	0.008s
determine_ATMOVE	<0.001s
deterministic_prediction	0.048s
model_templates	0.045s
Typical_prediction	0.005s
Resolution_reduction	0.01s
Sde_init	0.0003s

Table 11: Estimated communication time between sde_encode_stat and other leaf behaviors.

*Sde_en*code is the parent behavior of behavior *adaptive_template, arith_encode, deterine_ATMOVE, deterministic_prediction, model_templates* and *typical_prediction*. Since child behavior instance calls of *sde_encode* are not close to each other, all of them only communicate with *sde_encode_stat*. There is also communication between *sde_encode_stat* and *sde_init*. The communication time between *sde_encode_stat* and other leaf behaviors are displayed in Table 11. Additionally, communication time between behavior *resolution_reduction* and *encode_sde* is 0.01ms.

Some of communication is implemented by address reference. In these cases, SpecC profiler cannot provide estimation result. However, we can still produce the upperbound on communication time by analyzing the specification. For example, we estimated the communication time of behavior *adaptive_template* and *typical_prediction* based on following facts.

a) In behavior *adaptive_template*, if the value of variable mx is 50, then the communication time between *adaptive_template* and *sde_encode_state* is 0.30s.
b) In behavior *typical_prediction*, the traffic is smaller than one tenth of the total number of execution of operations of behavior *typical_prediction*.

6.5.3 Specification-Architecture Mapping Process

6.5.3.1 Initial Design: Pure ColdFire Design

We start our design with pure ColdFire solution. One ColdFire processor is selected from PE library. SpecC profiler indicates that execution time is 4.35 second when the entire JBIG encoder is mapped to the single ColdFire processor, as shown in Figure 27. The design cost is 3 cost units.

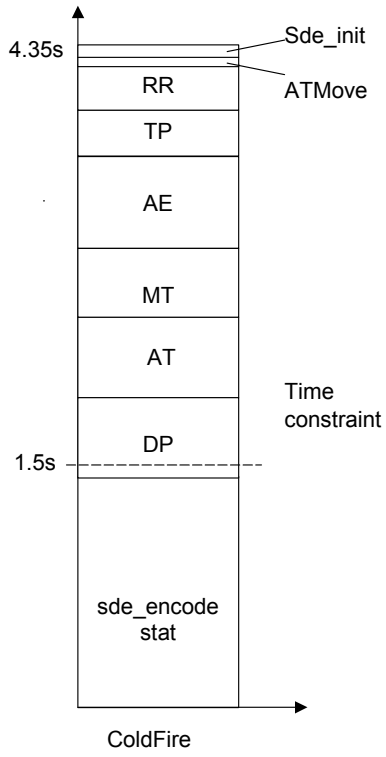


Figure 27: Behavior display of the initial design of JBIG encoder

6.5.3.2 Specification-Architecture Mapping for Multi-Processor Architecture

For JPIG encoder, the upperbound on the amount of PE is 2. PE_N(2,1) is 0.8976. Therefore, the execution time for 2-ColdFire architecture can be estimated by equation Total_P(487) * PE_N(2,1), which is 3.90s. Compared with 1.5s time constraint, the execution time of multi- coldFire solution is too large. Therefore, SW-HW co-design is required.

6.5.3.3 Improved Solution 1: ColdFire-Custom HW(AE) Solution

Table 12 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action. Cost_adding for all actions is 4, which includes 1 cost unit for the processing cost of HW, 2 cost unit for the material cost of custom HW, and 1 cost unit for the processing cost of communication.

Among those actions, the action of moving behavior AE to custom HW from ColdFire and running AE and AT parallel gives biggest TCR, which is 0.5326s. We select this action and update the architecture and specification-architecture mapping solutions.

Improved solution 1 maps AE to the custom HW and maps other behaviors to Coldfile. It executes AT and AE parallel. The estimated design time is 3.82s, as shown in Figure 28. The estimated cost is 7.

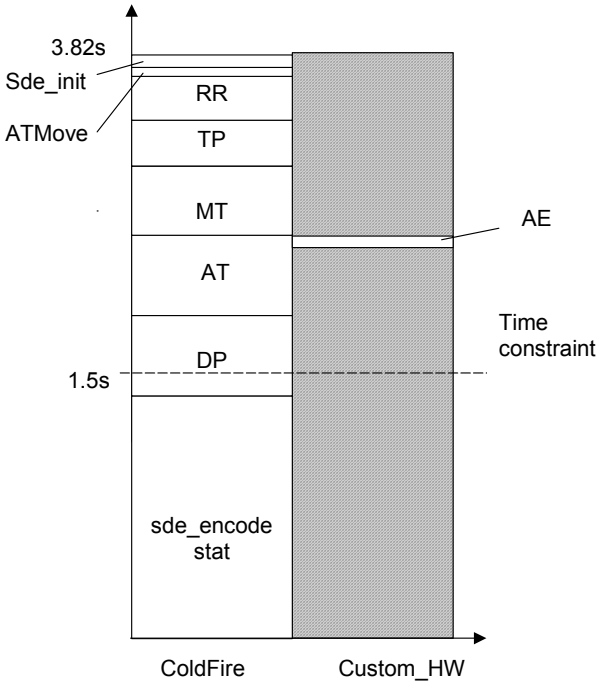


Figure 28: Behavior display for improved solution 1 for JBIG encoder.

Mapping action	Performance gain	Cost_adding	TCR
MOV, sde_init, SW,HW	0.049	4	0.012
MOV, AT, SW,HW	0.078	4	0.031
MOV, AE, SW,HW	0.465	4	0.116
MOV, DA, SW,HW	0.002	4	0.0005
MOV, DP, SW,HW	0.315	4	0.0786
MOV, MT, SW,HW	0.262	4	0.0655
MOV, TP, SW,HW	0.174	4	0.044
MOV, RR, SW,HW	0.212	4	0.053
MOV, SES, SW, HW	0.748	8	0.0935
PAR, AT, AE, SW,HW	**0.5326**	**4**	**0.1331**

Table 12: Table of performance_gain, cost_adding and TCR for improved solution 1.

Mapping action	Performance gain	Cost_adding	TCR
MOV, sde_init, SW,HW	0.049	1	0.049
MOV, AT, SW,HW	0.011	1	0.011
MOV, DA, SW,HW	0.002	1	0.002
MOV, DP, SW,HW	0.315	1	0.315
MOV, MT, SW,HW	0.262	1	0.262
MOV, TP, SW,HW	0.174	1	0.174
MOV, RR, SW,HW	0.212	1	0.212
MOV, SES, SW, HW	0.756	5	0.151
PAR, AT, DP, SW,HW	**0.454**	**1**	**0.454**

Table 13: Table of performance_gain, cost_adding and TCR for improved solution 2.

Mapping action	Performance gain	Cost_adding	TCR
MOV, sde_init, SW,HW	0.049	1	0.049
MOV, AT, SW,HW	0	1	0
MOV, DA, SW,HW	0.002	1	0.002
MOV, MT, SW,HW	0.262	1	0.262
MOV, TP, SW,HW	0.174	1	0.174
MOV, RR, SW,HW	0.212	1	0.212
MOV, SES, SW, HW	0.804	5	0.161
PAR, AT, MT, SW, HW	**0.386**	**1**	**0.386**

Table 14: Table of performance_gain, cost_adding and TCR for improved solution 3.

6.5.3.4 Improved Solution 2: ColdFire-Custom HW(AE, DP) Solution

Table 13 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action based on improved solution 1.

Among those actions, the action of moving behavior DP to custom HW from ColdFire and running DP and AT parallel gives biggest TCR, which is 0.454s. We select this action and update the architecture and specification-architecture mapping based on it.

Improved solution 2 maps AE and DP to the custom HW and maps other behaviors to the Coldfire. It executes AT and (AE+DP) parallel. The estimated design time is 3.37s second, as shown in Figure 29. The estimated cost is 8.

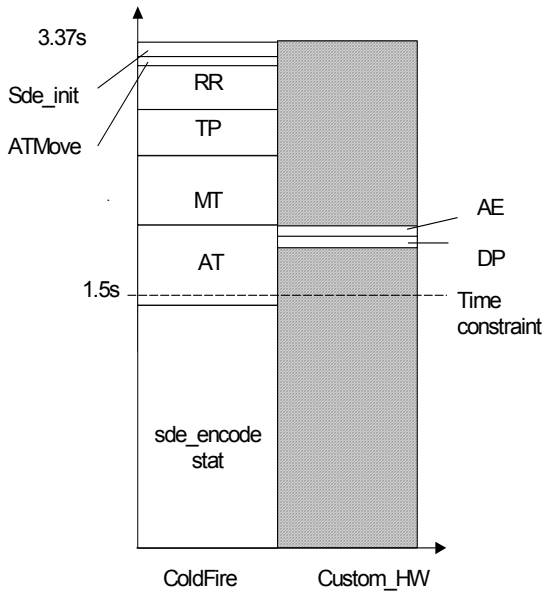


Figure 29: Behavior display for improved solution 2 for JBIG encoder.

6.5.3.5 Improved Solution 3: ColdFire-Custom HW(AE, DP, MT) Solution

Table 14 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action.

Among those actions, the action of moving behavior MT to custom HW from ColdFire and runing MT and AT parallel gives biggest TCR, which is 0.386s. We select this action and update the architecture and specification-architecture mapping solutions.

Improved solution 3 maps AE, DP, and MT to the custom HW and maps other behaviors to the Coldfire. It executes AT and (AE+DP+MT) parallel. The estimated design time is 2.98 second, as shown in Figure 30. The estimated cost is 9.

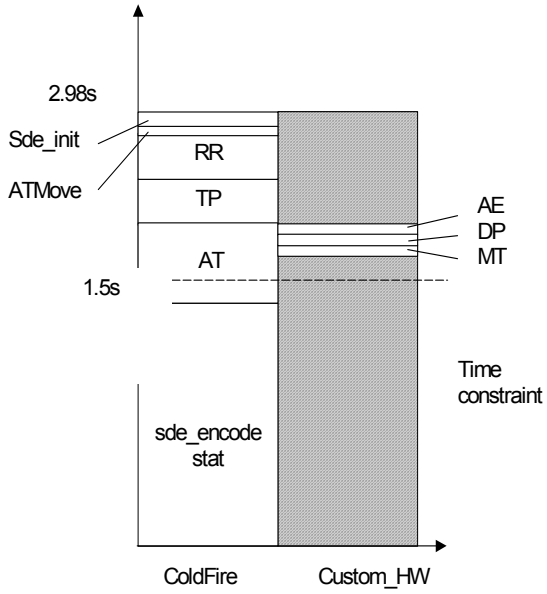


Figure 30: Behavior display for improved solution 3 for JBIG encoder.

Mapping action	Performance gain	Cost_adding	TCR
MOV, sde_init, SW,HW	0.049	1	0.049
MOV, AT, SW,HW	0	1	0
MOV, DA, SW,HW	0.002	1	0.002
MOV, TP, SW,HW	0.174	1	0.174
MOV, RR, SW,HW	**0.212**	**1**	**0.212**
MOV, SES, SW, HW	0.849	5	0.170
PAR, ATMOVE, TP, SW,HW	0.177	1	0.177

Table 15: Table of performance_gain, cost_adding and TCR for improved solution 4.

Mapping action	Performance gain	Cost_adding	TCR
MOV, sde_init, SW,HW	0.049	1	0.049
MOV, AT, SW,HW	0	1	0
MOV, DA, SW,HW	0.002	1	0.002
MOV, TP, SW,HW	0.174	1	0.174
MOV, SES, SW, HW	0.849	5	0.170
PAR, ATMOVE, TP, SW,HW	**0.177**	**1**	**0.177**
…			

Table 16: Table of performance_gain, cost_adding and TCR for improved solution 5.

6.5.3.6 Improved Solution 4: ColdFire-Custom HW(AE, DP, MT, RR) Solution

Table 15 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action.

Among those actions, the action of moving behavior RR to custom HW from ColdFire and running RR and AT parallel gives biggest TCR, which is 0.212s. We select this action and update the architecture and specification-architecture mapping solution.

Improved solution 4 maps AE, DP, MT, and RR to the custom HW and maps other behaviors to the Coldfire. It executes AT and (AE+DP+MT+RR) parallel. The estimated design time is 2.77 second, as displayed in Figure 31. The estimated cost is 10.

6.5.3.7 Improved Solution 5: ColdFire-Custom HW(AE, DP, MT,RR, TP) Colution

Table 16 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action.

Among those actions, the action of moving behavior TP to custom HW from ColdFire and runningOVE and TP parallel gives biggest TCR, which is 0.177s. We select this action and update the architecture and specification-architecture mapping solution.

Improved solution 5 maps AE, DP, MT, RR and TP to the custom HW and maps other behaviors to the Coldfire. It executes AT and (AE+DP+MT+RR) parallel and executes ATMOVE and TP parallel. The estimated design time is 2.29s, as displayed in Figure 32. The estimated cost is 11.

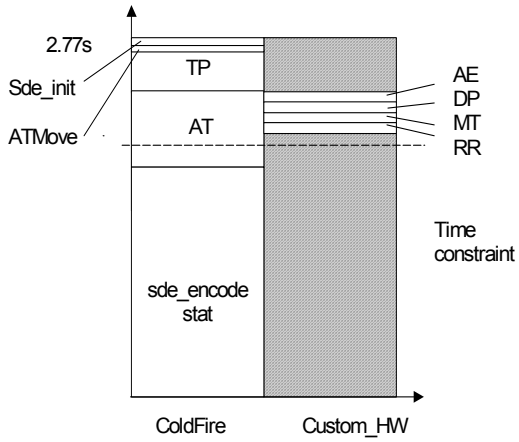


Figure 31: Behavior display for improved solution 4 for JBIG encoder.

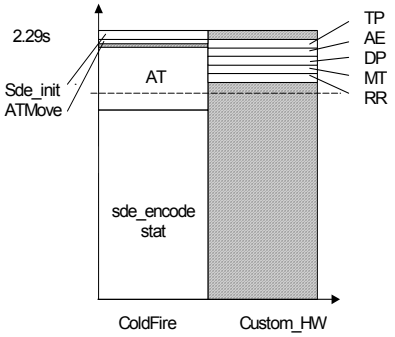


Figure 32: Behavior display for improved solution 5 for JBIG encoder.

Mapping action	Performance gain	Cost_adding	TCR
MOV, sde_init, SW,HW	0.049	1	0.049
MOV, AT, SW,HW	0	1	0
MOV, DA, SW,HW	0	1	0
MOV, SES, SW, HW	**0.854**	**5**	**0.171**
…			

Table 17: Performance_gain and TCR table for solution 6.

6.5.3.8 Improved Solution 6: ColdFire-Custom HW(AE, DP, MT,RR, TP, SES) Solution

Table 17 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action.

Among those actions, the action of moving behavior SES to custom HW gives biggest TCR, which is 0.854s. We select this action and update the architecture and specification-architecture mapping solution.

Improved solution 6 maps AE, DP, MT, RR, TP, and SES to the custom HW and maps other behaviors to the Coldfire. It executes AT and (AE+DP+MT+RR) parallel and executes ATMOVE and TP parallel. The estimated design time is 1.43 second, and the estimated cost is 16. The behavior display of improved solution 6 is displayed in Figure 33.

Improved Solution 6 is the solution that meets the time requirement with the low design cost.

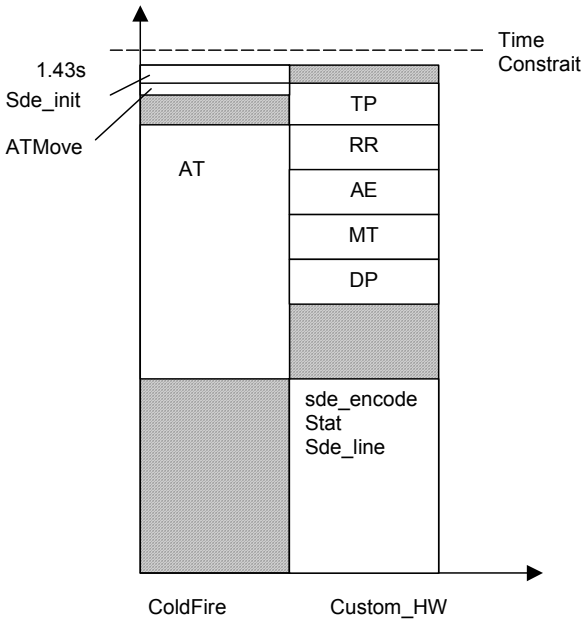


Figure 33: Behavior display for improved solution 6 for JBIG encoder.

7 Design Experience -- Vocoder

7.1 Vocoder

7.1.1 Introduction and Block Diagram

GSM vocoder is the voice encoding/decoding part of the GSM standard for mobile telephony. The block diagram of vocoder is displayed in Figure 34.

7.1.2 Testbench

We use a file containing 160 frames data as the design input.

There are two timing constraints for the vocoder example:

a) The time of executing behavior *LP_analysis, closed_loop, codebook,* and *update* once and executing open_loop half a time must be smaller than 10ms

b) The time of executing behavior *LP_analysis* and *open_loop* once and executing *closed_loop, codebook,* and *update* four times must be smaller than 20ms.

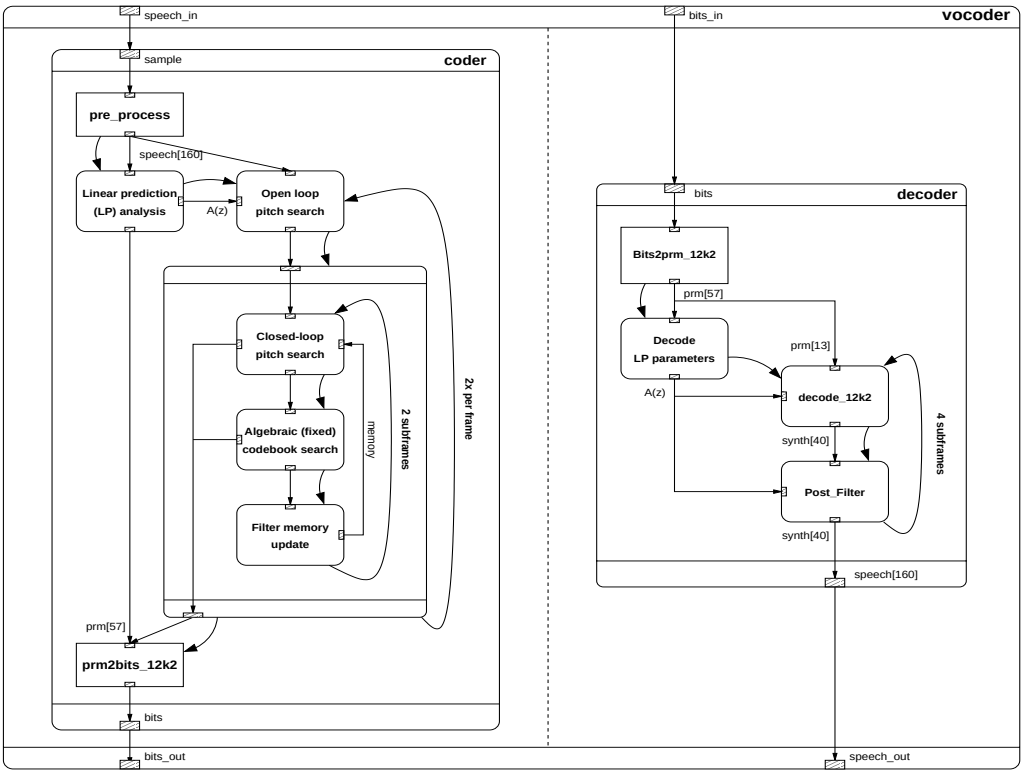


Figure 34: The block diagram of Vocoder example.

7.2 PE Types and Bus Protocol

We choose two PEs types from PE library:

a) Motorola ColdFire processor of 100 MHz
b) Custom HW of 100 MHz

We compute the weight tables of selected ColdFire processor and custom HW considering structure level pipeline technique. We also assume variables are stored in registers rather than in memories. With instruction level pipeline techniques, more than one instruction can be executed in one clock cycle. If a variable is stored in a register rather than stored in a memory, the execution time of memory access is not needed. Hence, considering these two issues, the estimated execution time will be much smaller and much closer to real simulation result. However, accurate evaluation for instruction pipeline and storing variables in registers is impossible at the operation level because accurate evaluation is determined by the compiling technique and the specification itself.

Based on our experience of comparing the estimated result of profiler and real simulator, we roughly divide the weighted data without considering pipeline and variables in registers by eight to represent the result with considering these two issues. In this case, the MOPS of chosen coldfire is 88<MOPS(ColdFirst) <800.

7.3 Specification Modeling

7.3.1 Exploring Parallelism

Five parallel executions exist in specification. There are:

a) In behavior *Open_Loop*, *Weight_Ai* and *WeightAi* are run parallel. Parallel execution *Weight_Ai* and *weightAi* gives 0.1% granularity improvement.
b) In behavior *LP_analysis*, *Find_Az1* and *Find_Az2* are run parallel. Parallel *Find_Az1* and *Find_Az2* gives 2.790% granularity improvement.
c) In behavior *LP_Analysis*, *Int_Lpc2* and *Q_Plsf_And_Intlpc* are run parallel.

Parallel *Int_Lpc2* and *Q_Plsf_And_Intlpc* gives 0.2% granularity improvement.

d) In behavior *Closed_Loop, Imp_Resp* and *Find_Targetvec* are run parallel. Parallel *Imp_Resp* and *Find_Targetvec* gives 2.7% granularity improvement.
e) In behavior Update: *Q_Gain_Code* and *Ex_Syn_Upd_Sh* are run parallel. Parallel *Q_Gain_Code* and *Ex_Syn_Upd_Sh* gives 0.39% granularity improvement

7.3.2 Choosing Granularity

Figure 35 displays the granularity of child behaviors of Main behavior in vocoder. Figure 36 to Figure 39 displays the granularity of child behaviors of behavior *LP_Analysis, Open_Loop, Close_Loop,* and *Codebook*. Figure 40 displays the granularity of child behaviors of behavior *Code_10i40_35bits*. The solid bars in figures represent non-leaf behaviors, while the doted bars represent leaf behaviors.

In Vocoder example, it is not feasible to use leaf behaviors as primitive behaviors for mapping because:

a) Leaf behaviors in vocoder do not reflect functional blocks in algorithm.
b) The granularity of leaf behaviors are various from 0.0001% to 19.66%. Therefore, leaf behaviors should be merged or decomposed to make granularity similar. The behavior merging/decomposing is helpful for mapping but difficult to implement.
c) Since there are 43 leaf behaviors in specification, the work of using leaf behaviors for mapping is heavy.

Because of above reasons, when we do behavior-specification mapping, we only consider behavior *LP_analysis, Open_loop, Closed_loop, Codebook,* and *Update*. If we cannot find an implement to meet the time constraint by mapping these behaviors to PEs, then in the next iteration of behavior-specification mapping, we will consider finer granularity behaviors.

LP_analysis, Open_loop, Closed_loop, Codebook, and *Update* are executed sequentially. Since we don't explore the parallelism inside these five behaviors, we will lose a certain possible performance improvement. As shown in 7.3.1, the total granularity improvement achieved by parallel execution is not more than 6%. It indicates that the lost performance improvement is not heavy.

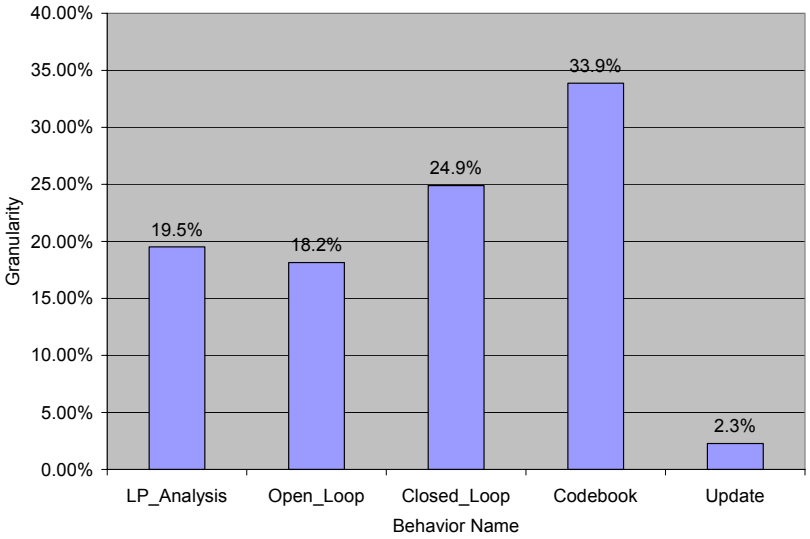


Figure 35: Granularity of top level behaviors in Vocoder.

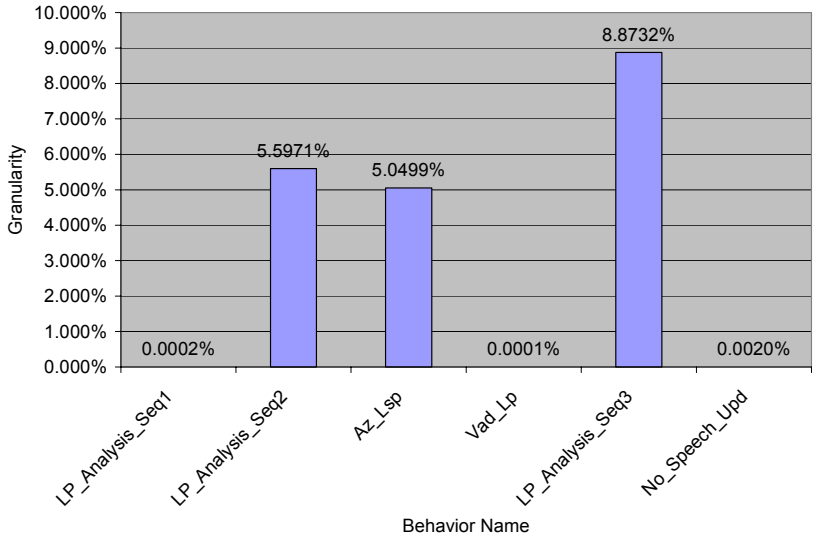


Figure 36: Granularity of child behaviors of LP_Analysis.

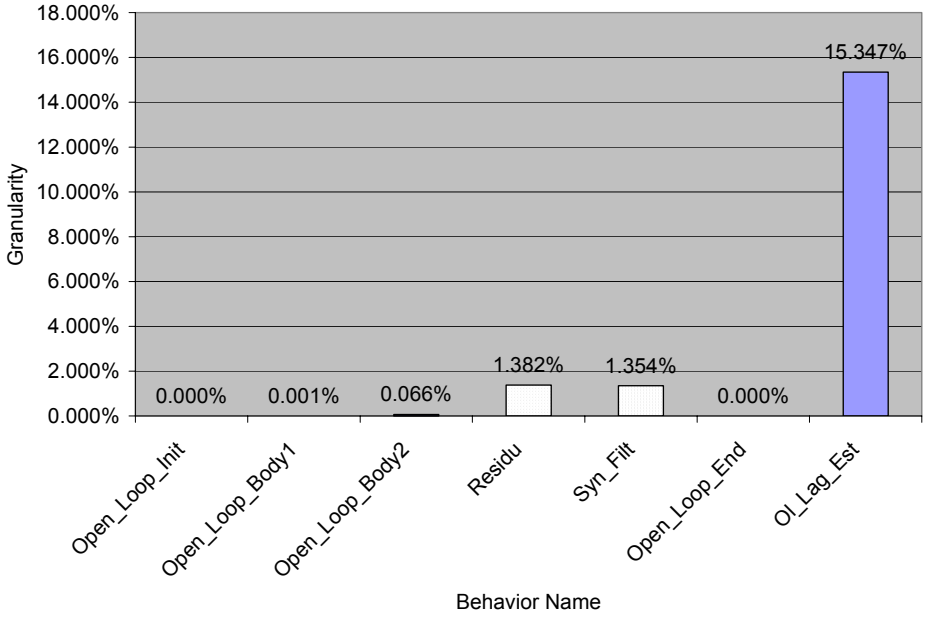


Figure 37: Granularity of child behaviors of open_loop.

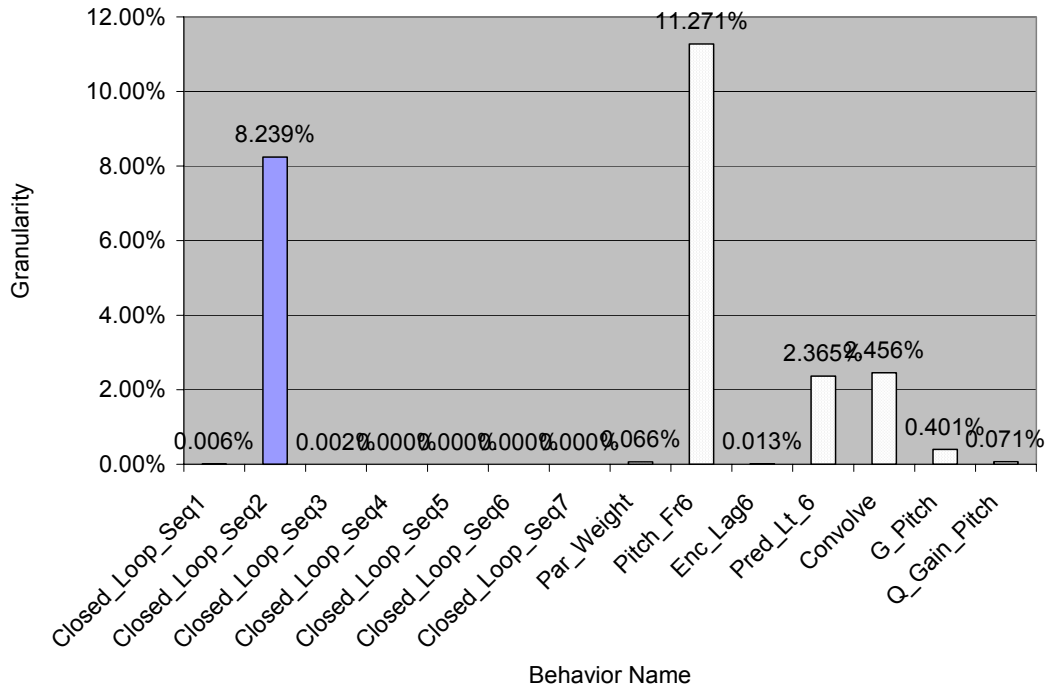


Figure 38: Granularity of child behaviors of close_loop.

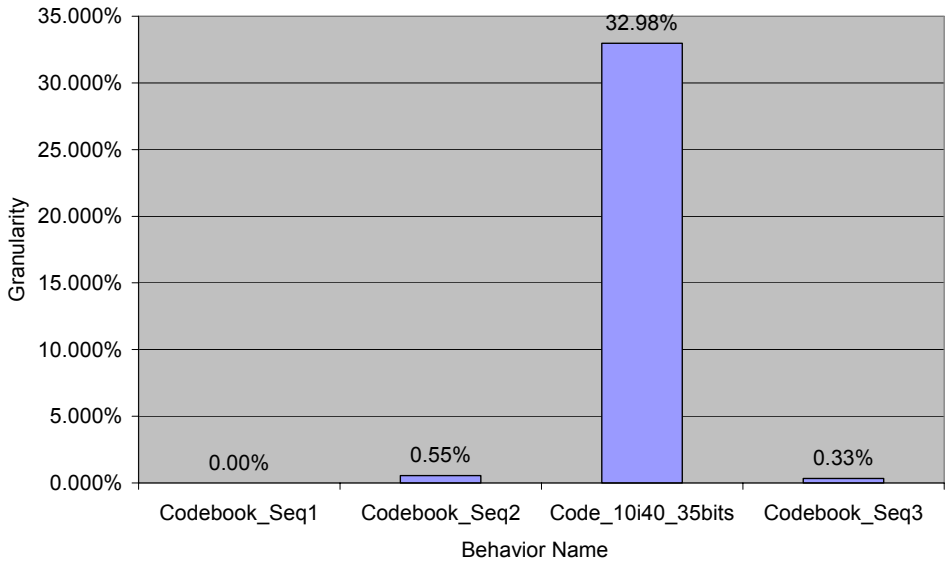


Figure 39: Granularity of child behaviors of codebook.

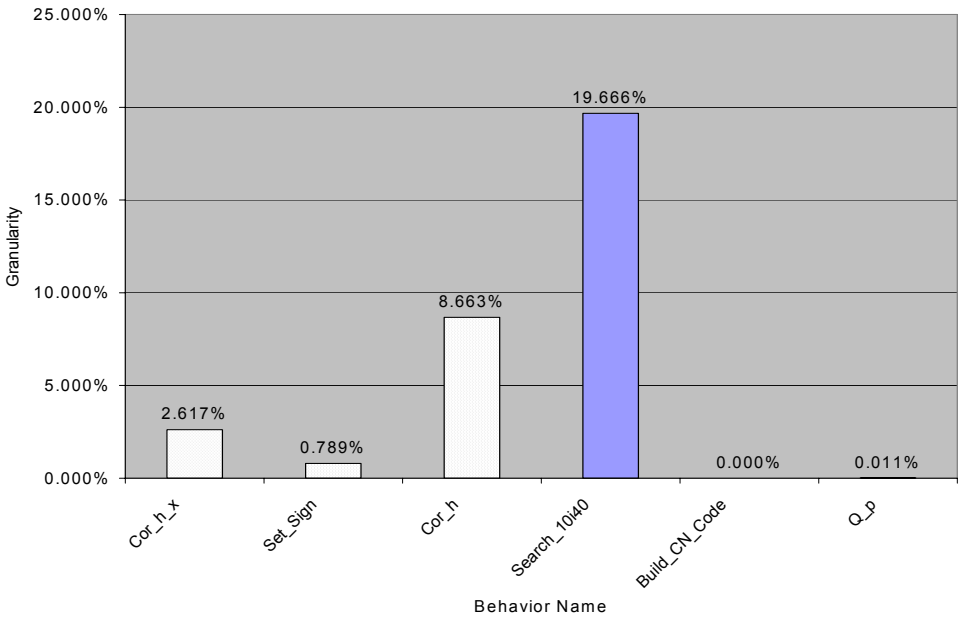


Figure 40: Granularity of child behaviors of Code_10i40_35bits.

	ColdFire(ms)	Custom HW(ms)	Performance_Gain(ms)
LP_Analysis	5.113094	0.532568	4.580526
Open_Loop	3.667963	0.378725	3.289237
Closed_Loop	7.376989	0.742122	6.634867
Codebook	10.38452	1.171317	9.2132
Update	0.743667	0.077052	0.666615

Table 18: Estimated computation time of primitive behaviors of Vocoder (per-frame)

	ColdFire(ms)	Custom HW(ms)	Performance_Gain(ms)
LP_Analysis	5.21535588	0.54321936	4.67213652
Open_Loop	3.74132226	0.3862995	3.35502174
Closed_Loop	7.52452878	0.75696444	6.76756434
Codebook	10.5922104	1.19474334	9.397464
Update	0.75854034	0.07859304	0.6799473

Table 19: Estimated execution time of primitive behaviors of Vocoder (per-frame)

7.4 Architecture Exploration -- PE Selection

7.4.1 Selecting PE Speed

The result of SpecC profiler indicates that the number of execution of operations per frame is 9878335. Therefore,

MOPS(Frame) = T_OP(Frame) / TC(Frame) = 9878335/30ms = 329 MOPS

Comparing MOPS(Frame) with the lowerbound of MOPS(ColdFire) which is 88, we cannot decide whether pure ColdFire can meet the time constraint. In this case, we need to estimate the performance of design and directly compare the estimation result with the given time constraint.

7.4.2 Selecting Amount of PEs

There is no parallel execution among selected primitive behaviors. Therefore PE_N = 1.

7.5 Specification-Architecture Mapping

7.5.1 Estimating Design Cost

Design cost assumption is the same as design cost assumption in JPEG encoder example, which is described in section 5.5.1.

7.5.2 Estimating Execution Time

7.5.2.1 Estimating Computation Time

Table 18 displays the estimated computation time of behaviors when a frame of data is process. For one frame data, *LP_Analysis* and *open_loop* are both executed once and *closed_loop, codebook*, and *update* are all executed four times. Column 2 represents the estimated execution time of behaviors per frame on ColdFire. Column 3 represents the estimated execution time of behaviors per frame on custom HW. Column 4 indicates the difference between column 2 and 3, which represents the performance_gain achieved by moving behavior from ColdFire to custom HW without considering communication.

SpecC profiler generates the computation time of behaviors for ColdFire and custom HW in terms of clock cycles. The execution time displayed in column 3 and 4 is achieved by dividing total clock cycles by PE frequency.

7.5.2.2 Estimating Computation Time

The five behaviors are communicated by address reference, which cannot be evaluated by SpecC profiler.

Based on the Vocoder standard, we notice that the number of execution of communication is always smaller than 1/50 of the numbers of execution of computation for each behavior. Therefore, for the vocoder project, we increase the execution time of computation by 1/50 to represent the total execution time for both computation and communication, which is shown in Table 19.

7.5.3 Specification-Architecture Mapping Process

7.5.3.1 Initial Design: Pure ColdFire Design

We start with the architecture containing a pure ColdFire processor and maps vocoder on this architecture. The behavior display of this mapping is shown in Figure 41.

The sum of the execution time for the behaviors corresponding to the first design constraint is equal to P_Frame(*LP_analysis*) + P_Frame(*closed_loop*)/4 + P_Frame(*codebook*)/4 + P_Frame(*update*) / 4 + P_Frame(*open_loop*) / 2 = 11.80ms. It exceeds the first design constraint, which is 10ms. The sum of the execution time for the behaviors corresponding to the second design constraint is equal to P_Frame(LP_analysis) + P_Frame(closed_loop) + P_Frame(codebook) + P_Frame(update) + P_Frame(open_loop) = 27.83ms, which also exceeds the 20ms constraints.

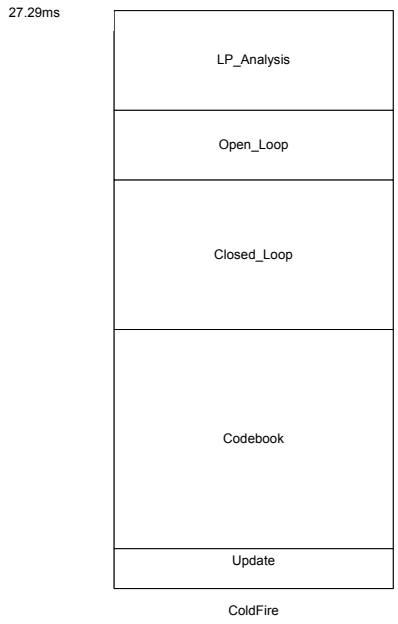


Figure 41: Behavior display of initial solution of Vocoder.

7.5.3.2 Improved Solution 1: Pure ColdFire Design

Table 20 tells the performance_gain, cost_add, and time–cost ratio(TCR) for each specification-architecture mapping action. Performance_gain is calculated based on data in Table 19 and based on equations in section 4.3.1. Cost_adding for all actions is 4, which includes 1 cost unit for the processing cost of custom HW, 2 cost units for the material cost of custom HW, and 1 cost unit for the processing cost of communication.

Mapping action	Performance gain(ms)	Cost_adding	TCR
MOV, LA, SW,HW	5.21535588	4	1.303839
MOV, Open, SW,HW	3.74132226	4	0.935331
MOV, Close,SW,HW	7.52452878	4	1.881132
MOV, CodeBook, SW,HW	10.5922104	4	**2.648053**
MOV, Update, SW,HW	0.75854034	4	0.189635

Table 20: Table of performance_gain, cost_adding, and TCR for improved 1.

Among those actions, the action of moving behavior *CodeBook* to custom HW from ColdFire gives the biggest TCR, which is 2.648. We select this action and update the architecture and specification-architecture mapping solution.

After updating, we recompute the execution time for design constraints. For design constraint 1, we compute P_Frame(*LP_analysis*) + P_Frame(*closed_loop*) / 4 + P_Frame(*codebook*) / 4 + P_Frame(*update*) / 4 + P_Frame(*open_loop*) / 2 = 9.46ms. It meets the design constraint 1.

Similar, P_Frame(*LP_analysis*) + P_Frame(*closed_loop*) + P_Frame(*codebook*) + P_Frame(*update*) + P_Frame(*open_loop*) = 18.43ms. The second design constraint is also satisfied. Therefore, this implementation is the one that meets our requirement. The Behavior display is depicted in Figure 42.

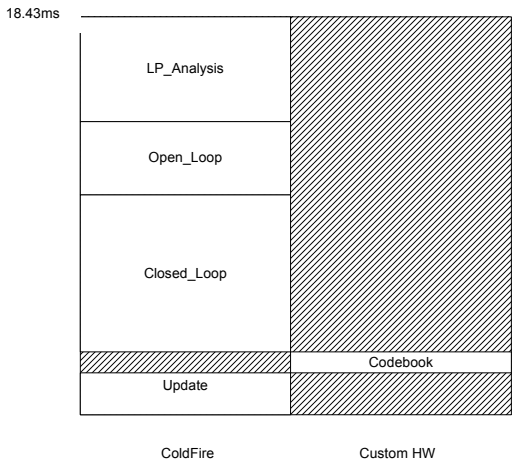


Figure 42 : Behavior display of impved solution 1

8 Conclusion

In this report, we show the use of SpecC profiler for system level design at the high level of abstraction. SpecC profiler can help designers to perform specification modeling, architecture exploration, and specification-architecture mapping tasks.

Three examples have been described in the report. JPEG encoder example describes the general design flow including specification modeling, architecture exploration, and specification-architecture mapping. JBIG encoder describes how to handle non-clean specification. Vocoder example describes the design flow with multi-time constraints. With these three design examples, we conclude that the SpecC profiler is an indispensable tool for the system level design.

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

[2] D.D. Gajski, J.Zhu, R.Domer, etc. *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers March 2000

[3] Andreas Gerstlauer, Rainer Doemer, J.Peng, D Gajski, System Design : a Practical Guide of SpecC, Kluwer Academic Publishers. 2001

[4] Andreas Gerstlauer, Shuqing Zhao etc. *Design of a GSM Vocoder using SpecC Methodology*, University of California, Irvine, Technique report ICS-99-11, Feb 1999

[5] Motorola, Inc. Semiconductors Products Sector, DSP Division, *ColdFire2 Integrated Microprocessor Designer Manual*, 1998
[6] Dan Gajski, N. Dutt, C.H.We, Y.L.Lin *High-level synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, Massachusetts, 1991.
[7] http://www.apspg.com/press/050100/coldfire.html
[8] http://www.privacy.nb.ca/cryptography/archives/coderpunks/new/1998-06/01
[9] Lukai Cai, Junyu Peng, *Design of a JPEG Encoding System*, University of California, Irvine, Technical Report ICS-99-54, Nov, 1999
[10] Jyunyu Peng, Lukai Cai, *Design of a JBIG Encoder using SpecC Methodology*, University of California, Irvine, Technical Report ICS-00-13, Jun 2000
[11] Daniel Gajski, Nikil Dutt, *High-Level Synthesis: Introduction To Chip and System Design*