Variable Mapping of System Level Design

Lukai Cai and Daniel Gajski

CECS Technical Report 02-32
Oct 08, 2002

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{lcai, gajski}@ics.uci.edu

Variable Mapping of System Level Design

Lukai Cai and Daniel Gajski

CECS Technical Report 02-32
Oct 08, 2002

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425,USA
(949) 824-8059

{lcai, gajski}@ics.uci.edu

Abstract

This report presents a variable-memory mapping approach of the system level design, which maps the variables of the system behavior to the memories of the system architecture. It first introduces a novel memory size model to compute the required minimal memory sizes when allowing variables with un-overlapped lifetime to share the same memory portion. It then presents variable mapping algorithms for different design assumptions. The variable mapping algorithms are applied before obtaining some implementation details, such as bus topology and bus protocols, which moves the variable mapping to the earliest design stage.

Contents

List of Figures

List of Tables

Variable Mapping of System Level Design

Lukai Cai and Daniel Gajski
Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine

Abstract

This report presents a variable-memory mapping approach of the system level design, which maps the variables of the system behavior to the memories of the system architecture. It first introduces a novel memory size model to compute the required minimal memory sizes when allowing variables with un-overlapped lifetime to share the same memory portion. It then presents variable mapping algorithms for different design assumptions. The variable mapping algorithms are applied before obtaining some implementation details, such as bus topology and bus protocols, which moves the variable mapping to the earliest design stage.

1. Introduction

In order to handle the ever increasing complexity and time-to-market pressures in the design of system-on-chips(SOCs) or embedded systems, the design has been raised to the system level to increase productivity. Figure 1 illustrates extended Gajski and Kuhn's Y chart representing the entire design flow, which is composed of four different levels: system level, RTL level, logic level, and transistor level. The thick arc represents the system level design. It starts from the behavior specification representing the designs' functionality (also called application or system behavior), which is denoted by point S. The behavior specification contains a set of functional blocks (also called behavior). It also contains a set of variables that reserve the data transferred between intra-block operations or inter-block operations. The system level design then synthesizes the behavior specification to the system architecture denoted by point A. A system architecture consists of a number of PEs (processing elements) and a number of global memory connected by buses. Different PEs can belong to different PE types. Each PE implements a number of functional blocks of the behavior specification.

We divide the systhesis process of system level design to three steps: behavior-PE mapping, variable-memory mapping, and channel-bus mapping. **Behavior-PE mapping** selects PEs to assemble the system architecture and maps the behaviors (functional blocks) in the behavior specification to PEs. **Variable-memory mapping** selects sizes of local memories of PEs and the global memories on the system architecture and maps the variables of behaviors to the memories. **Channel-bus mapping** selects bus topology of the system architecture, selects bus protocols and maps the communication among behaviors to the selected buses. In this report, we focus on the variable-memory mapping step.

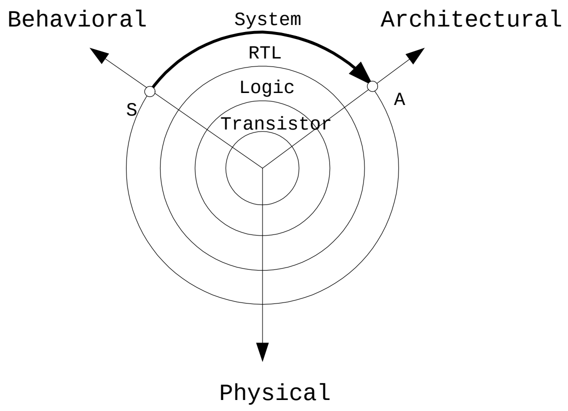


Figure 1. Extended Gajski and Kuhn's Y chart

Variable-memory mapping is critical because it heavily affects the chip area and the execution time of design. 70% of the chip area is dedicated to memory, which is determined by the memory size. Furthermore, different variable-mapping alternatives will produce different amount of traffic among PEs and global memories, which determines the communication time of design. Taking the chip area and the execution time into consideration, we have two objectives during variable-memory mapping: to minimize the size of memories and to minimize either the amount of traffic or the communication time on the system architecture.

This report presents a novel but straightforward variable-mapping approach, which contains three parts: memory size computation, traffic amount computation, and variable-memory mapping algorithm. During memory size computation, we analyze lifetime of variables and compute the required minimal local memory sizes of PEs and global memory sizes. During traffic amount computation, we compare the produced amount of traffic for different variable mapping alternatives. Finally, we introduce a straightforward variable-mapping heuristics.

Compared with other works introduced later in section 2, we aim to handle the very complex design at the earliest design stage. As a result, we tailor our approach to fulfill the need of very complex design by considering following three issues:

1. We adopt the hierarchy specification as input, which ensures system behavior's scalability.
2. We implement variable-memory mapping in the earliest design stage, even before deriving implement details such as bus topology and bus protocols.
3. We adopt a constructive algorithm, which reduces the algorithm's complexity.

This report is organized as followed: section 2 describes the related work and our contribution. Section 3 describes when variable-memory mapping is implemented in the system design flow. Section 4 introduces the system model. The design flow of variable mapping is given in section 5. Section 6 introduces the model for memory size computation. Section 7 defines the detailed variable mapping problems and provides corresponding solutions. Experimental result is described in section 8. Finally, section 9 gives the conclusion.

2. Related Work

Lots of research has been done on system level design for years. Some of them took the memory issue into account.

Research in [13][10] involves cache issues. [13] solves the application-specific multiprocessor synthesis problem to optimize cache hierarchy size of periodic real time systems. [10] maps variables into Scratch-Pad memory and off-chip DRAM accessed through data cache, to maximize the performance.

Research in [11] implements behavior-architecture mapping for the application-specific multiprocessor systems by using mixed integer linear programming. It adds the cost of memories determined by the amount of memories to the cost equation of design. It chooses message-passing communication mechanism. Thus, the variable through which two tasks in different PEs are communicated has a local copy at the local memory of each PE. It also assumes that the tasks mapped to one PE are executed sequentially. Therefore the amount of memory required by the PE must equal to the largest amount that is required by any of the task mapped to that PE.

Research in [12] uses the similar memory model as [11] to implement behavior-architecture mapping. The difference between [12] and [11] is that when two tasks communicate through a variable, [12] reserves the memory for the variable in the sending task until the value of the variable is transferred to the receiving task. [12] adopts constraint logic programming paradigm as the basic algorithm.

The goal of [9] is the same as our goal, which is to implement variable-memory mapping. Its targeted architecture model contains a set of PEs with local memories and a global memory. Unlike [11][12], it choose shared-memory mechanism for communication. Therefore, the variable is mapped to either one of the local memories of PEs or the global memory. It computes communication time by adding up the read access time and write access time of variables. It computes the memory size by adding up the size of variables mapped to the memories. It uses integer linear program to produce the optimal mapping result.

The register allocation and variable-register binding problem has been discussed in the field of high level synthesis. [5] introduces some related algorithms including clique partitioning, left-edge algorithm, and weighted-bipartite-matching algorithm. We notice that the difference between variable-memory mapping problem in the system synthesis domain and variable-register binding problem in the high level synthesis domain can be classified into the following two aspects:

1. Size of variable. In the high level synthesis domain, the sizes of variables are same, which is determined by the memory/register width. In the system synthesis domain, the sizes of variables can be ranged from 1 bit of bit type to 10000 byte of structure type.
2. Estimation accuracy of lifetime of variable. In the high level synthesis domain, the lifetime of variable is computed based on clock cycles. Therefore, it can be accurately estimated according to the finite state machine. However, in the system synthesis domain, the lifetime of variable is computed based on the system estimation model. If the system estimation model is not accurate, then the estimation of lifetime of variable is not accurate, which will make lifetime analysis useless. Unfortunately, the system level estimation model such as VCC[2] never guarantees accuracy.

Our work don't consider cache optimization, which is different from [13] and [10]. In comparison to research in [11][12][9], our work has the following contributions:

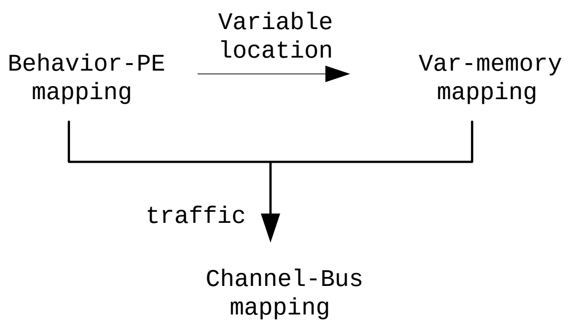


Figure 2. The task dependencies between behavior-PE mapping, variable-memory mapping, and channel-bus mapping

1. Our input system behavior is modeled hierarchically. On the other hand, the system behaviors in [11][12][9] are flat models.

2. We support message-passing mechanism and shared-memory mechanism, while [11][12] do not support shared-memory mechanism and [9] does not support message-passing mechanism.

3. We analyze the lifetime of variables independent of the system estimation model. We also take preemptive RTOS into consideration. On the other hand, [9] does not analyze lifetime of variables. The lifetime analysis in [11][12] is much easier than our analysis because they don't support hierarchical behavior model, shared-memory communication mechanism, and preemptive RTOS.

4. We implement variable mapping according to the amount of traffic, rather than the communication time used in [11][12][9]. Therefore, we can implement variable mapping before channel-bus mapping, while [11][12][9] implement variable mapping after channel-bus mapping.

5. We use straight-forward heuristics to solve the problem, rather than use time-consuming ILP adopted by [11][12][9]. This enables us to solve the variable-mapping problem for very complex designs within affordable time.

3. Variable Mapping in System Design Flow

Before implementing variable mapping, we must understand when the variable mapping should be applied in the design flow.

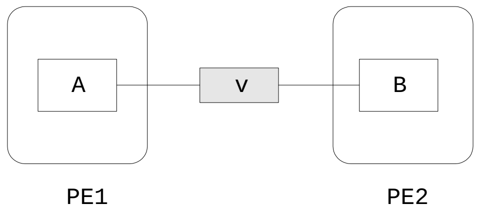


Figure 3. A simple example of variable mapping problem

Figure 2 shows task dependencies among *behavior-PE mapping*, *variable-memory mapping*, and *channel-bus mapping* steps.

First, *behavior-PE mapping* determines computation time of behaviors. In addition to that, it also determines the variable location: whether a variable is used only by a certain PE, or by more than one PE. As explained later in section 5, this knowledge is required by *variable-memory mapping*. Therefore, *variable-memory mapping* must be implemented after *behavior-PE mapping*.

It is obvious that the *channel-bus mapping* must be implemented after *behavior-PE mapping* because the bus only handles the communication between behaviors mapped to different PEs, which is determined by *behavior-PE mapping*. However, the amount of traffic on bus is not only influenced by *behavior-PE mapping*, but also influenced by *variable-memory mapping*. For example, in Figure 3, behavior *A* in *PE1* is communicated with behavior *B* in *PE2* through variable *v*. Behavior *A* reads *v* 100 times and behavior *B* writes *v* once. In this case, if we map variable *v* to the local memories of *PE1* and *PE2*, the amount of traffic is 1. On the other hand, if we map variable *v* to a global memory , then the amount of traffic is 101. Because the amount of traffic on bus is influenced by *variable-memory mapping* and is used as input of *channel-bus mapping*, we implement *variable-memory mapping* before *channel-bus mapping*.

The Figure 4(a) displays the system design flow used in [11] [12] [9]. The Figure 4(b) displays our system design flow. We move the *variable-memory mapping* from the position after *channel-bus mapping* to the position before *channel-bus mapping*.

The *variable-memory mapping* in design flow (a) and (b) is in different level of accuracy when estimating the traffic. The former one estimates the communication time because the bus protocol is known. However, the latter one estimates amount of traffic because the bus protocol is undecided. To extend the variable-memory mapping to a more accurate level, we also extend the design flow in Figure 4(b) to the design flow in Figure 4(c). In in Figure 4(c), the *variable-memory mapping* is based on the amount of traffic, while *variable-memory re-mapping* is based on the communica-

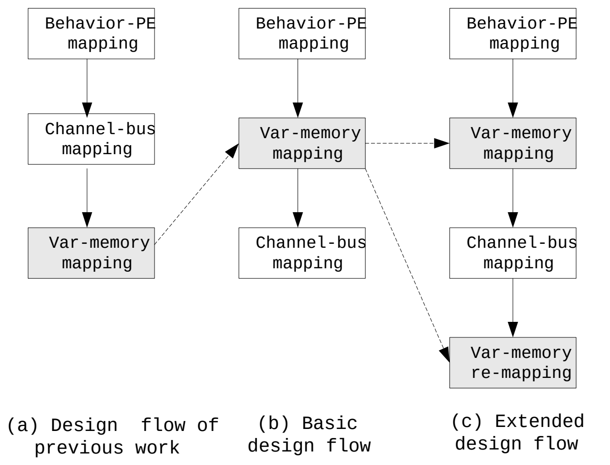


Figure 4. Design flow of system synthesis

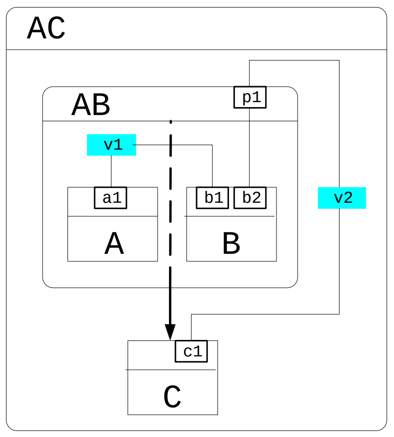


Figure 5. System behavior example 1

tion time. The *variable-memory re-mapping* is introduced in section 7.4

4. System Model

4.1 Behavior Model

We use SpecC language[6] to model the system behavior. SpecC language is a C-language based system level design language that supports behavior hierarchy, behavior concurrency, behavior pipeline, and state transitions in behavior level.

SpecC uses **behavior** to represent a functional block. Among behaviors, **leaf behavior** represents the undivided unit. One leaf behavior can only be mapped to one PE. It contains a number of hierarchically called functions but no sub-behavior instances. In addition to leaf behavior, **hierarchy behavior** consists of a number of sub-behavior instances that are executed in parallel, sequential, pipeline, or, FSM fashion [6]. We call that sub-behaviors instances are the children of the hierarchy behavior and the hierarchy behavior is the parent of the sub-behavior instances. In Figure 5, *A*, *B*, and *C* are leaf behaviors. *AC* and *AB* are hierarchy behaviors. In behavior *AB*, leaf behavior *A* and *B* are executed parallel, which is denoted by a thick doted line. In behavior *AC*, behavior *AB* is executed before the execution of behavior *C*, which is a denoted by thick arrow.

Each behavior has a set of ports to connect with other behaviors. It also has a set of variables that connect the ports of its sub-behaviors instances. In Figure 5, variable is denoted by shaded rectangle and port is denoted by framed rectangle. For example, behavior *AB*'s variable *v1* connects port *a1* of behavior *A* and port *b1* of behavior *B*. The ports of behaviors can also connects to its parent behavior's ports. For example, port *b2* of behavior *B* is connected to port *p1* of behavior *AB*. Since the behaviors are in different hierarchy levels, variables are also declared in different hierarchy levels.

Although we adopt SpecC as our input modeling language, the idea introduced in this report can also be applied to other system level design languages such as SystemC[1]. This is because the behavior hierarchy is supported by most of system level design languages.

4.2 Architecture model

We choose multi-PE architecture as the system architecture, which is shown in Figure 6. Each PE in the architecture contains a micro-processor, a custom hardware, a virtual component, or an IP, It also contains a local memory. The above two parts of PE are connected to each other by a local bus. If a PE contains a microprocessor, we call it **SW PE**. If a PE contains a custom hardware, we call it **HW PE**. The system architecture also contains a set of global memories. A global memory may be either single port, dual port, or special purpose memory such as FIFO. The interconnection network consists of several buses. The PEs and global memories are connected by one or more system buses via corresponding interfaces. The system bus is associated with a well-defined protocol with the components on the bus have to respect. The protocol may be standard, such as itVME bus, or custom. An interface bridges the gap between a local bus of a PE/memory and system buses.

Each PE communicates with every other PE within the multi-PE system, using one of the two communication mechanisms. In a **shared-memory mechanism**, all the PEs are connected to a common shared memory through an interconnection network, which means that each processor can access any data in the shared memory. In a **message-passing mechanism**, on the other hand, each processor tends to have a large local memory, and sends data to

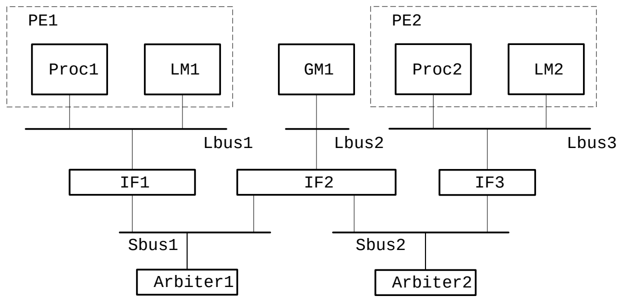


Figure 6. A general system architecture

other processors in the form of messages through an interconnection network.

4.3 Variable Classification

We classify variables in the behavior specification to three types: function variable, static behavior variable, and dynamic behavior variable.

4.3.1 Function Variable

Function variable is a variable declared in functions of leaf behaviors. The function variable in SpecC is similar to the function variable in C language. The lifetime of function variable equals to the lifetime of the function in which it is declared. Since the lifetime of any two sequential executing functions is un-overlapped, any two function variables declared in two sequential executing functions can share the same portion of the memory.

In our behavior model, *hierarchy behavior* is not allowed to contain any function variables. For *leaf behavior*, we compute the sum of its function variables as a whole. We call it **stack size** of leaf behavior.

Since functions in leaf behaviors can be called hierarchically, we compute the stack size of function f $S_{stack}(f)$ following the equation:

$$S_{stack}(f) = \max_{f_i \in S\_C(f)} (S_{stack}(f_i)) + \sum_{v_j \in S\_V(f)} (S(v_j))$$

where $S\_C(f)$ is the set of function calls in f and $S\_V(f)$ is the set of declared variables in f. The stack size equals to the largest stack size of its calling functions plus the sum of sizes $S(v_j)$ of variables declared in f .

The *stack size of leaf behavior* $S_{stack}(A)$ of behavior *A* equals to the stack size of *A*'s *main* function, which represents the root function of leaf behavior *A* in SpecC.

4.3.2 Static Behavior Variable

Static behavior variable is a behavior variable declared inside behavior and outside any functions. **Behavior variable** is the variable usually declared in the *hierarchy behavior* to connect the ports of sub-behavior instances. The static behavior variable in SpecC is similar to the static variable in C language. The lifetime of static behavior variable is the same as the lifetime of the entire system behavior. As a result, any two static behavior variables cannot share the same portion of the same memory.

4.3.3 Dynamic Behavior Variable

Dynamic behavior variable is also a behavior variable declared inside behavior and outside any functions. Unlike the static behavior variable, the lifetime of dynamic behavior variable equals to the execution interval of the behavior in which the variable is declared. If we treat the behavior in SpecC as the function in C language, the dynamic behavior variable is similar to function variable in C. Two dynamic variables declared in two sequential executing behaviors can share the same portion of the same memory.

4.4 Variable-Memory Mapping Mechanism

There are three mechanisms of variable-memory mapping: map variable to each local memory of its connecting PEs, map variable to only one local memory of its connecting PEs, and map variable to the global memory. We define that v is connected to behavior ω if and only if v is declared in behavior ω or v is connected to the port of behavior ω. We define that a variable v is connected to PE ρ if and only if any v's connecting behavior ω is mapped to PE ρ.

4.4.1 Mapping to Each Local Memory

In **mapping to each local memory** mechanism, we map a variable to each local memory of its connecting PEs. This mechanism is applied if we choose message-passing mechanism for communication. This mapping mechanism ensures that each connecting PE of the variable has a variable's local copy.

In *mapping to each local memory* mechanism, the read access of variable is achieved by accessing the local memory of PE. Therefore, it doesn't produce any traffic on the interconnection network. On the other hand, when write access of the variable connecting to several PEs happens, the system must update local copy of the variable in all of its connecting PEs. As a result, write access produces traffic on the interconnection network.

4.4.2 Mapping to One Local Memory

In **mapping to one local memory** mechanism, we map a variable to only one local memory of its connecting PEs. This mechanism is applied when we choose shared-memory mechanism for communication and the variable is mapped to a local memory of PE rather than a global memory.

In *mapping to one local memory* mechanism, assuming a variable v is mapped to the local memory of its connecting PE ρ. The read/write access of variable v from/to the behaviors mapped to PE ρ doesn't produce any traffic on the interconnection network. The read/write access of variable v from/to the behaviors mapped to the PEs other than ρ produces the traffic.

4.4.3 Mapping to a Global Memory

In **mapping to a global memory** mechanism, we map a variable connecting to more than one PE to a global memory. This mechanism is applied when we choose shared-memory mechanism for communication and the variable connecting to more than one PEs is mapped to a global memory.

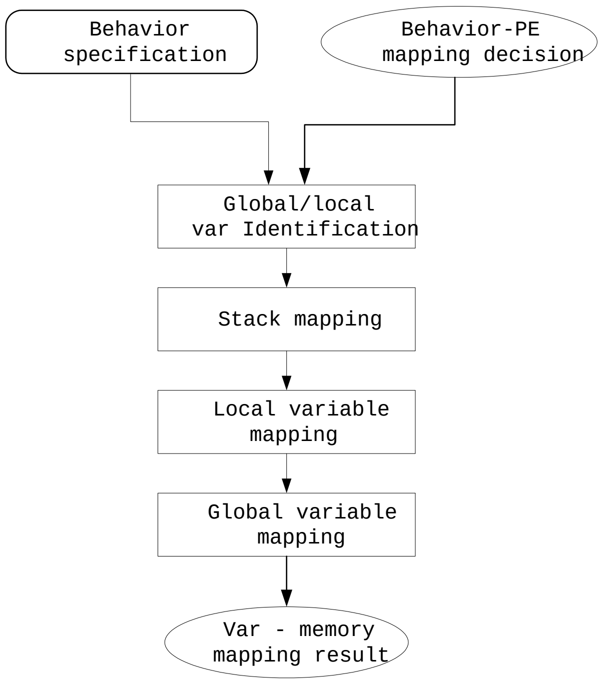
In *mapping to a global memory* mechanism, every read-/write access of the variable mapped to the global memories produces traffic on the interconnection network.

5. Design Flow of Variable Mapping

Figure 7 describes the design flow of variable-memory mapping, which consists of four steps: *global/local variable identification*, *stack mapping*, *local variable mapping*, and *global variable mapping*.

5.1 Input

Variable-memory mapping requires *behavior specification* and *behavior-PE mapping decision* as input. We use behavior model described in section 4.1 as the behavior specification. It only reflects the functionality of design, which is independent of implementation. It is an un-timed model. Behavior-PE mapping decision contains the system architecture and behavior-PE mapping information. It includes: which types of PEs and how many PEs are selected in the system architecture; which behaviors are mapped to which PEs; whether the system architecture contains a global memory; what are the sizes of local memories of PEs. Since variable-memory mapping is implemented before channel-bus mapping, the topology of interconnection network and the bus protocols are unknown. In our project, we specify behavior-PE mapping decision in the format of annotation of the behavior specification.



Behavior specification
Behavior-PE mapping decision
Global/local var Identification
Stack mapping
Local variable mapping
Global variable mapping
Var - memory mapping result

Figure 7. The design flow of variable-memory mapping

5.2 Global/Local Variable Identification

With the behavior specification and behavior-PE mapping decision as inputs, the first step of variable-memory mapping is *global/local variable identification*. The global/local variable identification identifies whether a static/-dynamic behavior variable is a global variable or is a local variable. If a variable's connecting behaviors are mapped to one PE, then the variable is a **local variable** of that PE. Otherwise, if a variable 's connecting behaviors are mapped to more than one PEs, then it is a **global variable**.

Global/local variable identification is the step of linking the implementation-independent behavior specification to the behavior-PE mapping decision. For the same behavior specification, the behavior-PE mapping decision determines the variable identification. For example, as shown in Table 1, if different behavior-PE mapping decisions are made for the behavior specification described in Figure 5, then the variables $v1$ and $v2$ have different identifications. If we map the behavior AC to $PE1$ as shown in the raw 1 of Table 1, then $v1$ and $v2$ become the local variables of $PE1$. However, if we map behavior AB to $PE1$ and map behavior C to $PE2$ as shown in the raw 2 of Table 1, then variable $v1$ becomes the local variable of $PE1$ and $v2$ becomes the global variable. This is because that $v2$ is connected to both behavior AB and behavior C, which are mapped to differ-

Behavior-PE mapping		Variable identification		
PE1	PE2	Local (PE1)	Local (PE2)	Global
AC	-	v1, v2	-	-
AB	C	v1	-	v2
A, C	B	-	-	v1, v2
A	B, C	-	v2	v1

Table 1. Behavior-PE mapping decisions and the produced variable identification for behavior example 1.

ent PEs.

5.3 Stack Mapping

The second step of variable-memory mapping is *stack-mapping*, which maps the stack of leaf behavior to the local memory of the PE to which the leaf behavior is mapped. The stack cannot be mapped to the global memory or the local memories of PEs other than the one to which the leaf behavior is mapped. Otherwise, each stack access will produce traffic on the interconnection network. As a result, the size of local memory of PE must be greater than the stack size of the leaf behavior that is mapped to that PE.

5.4 Local Variable Mapping

The third step of variable-memory mapping is *local variable-mapping*, which attempts to map the local behavior variables to the local memories of the variables' connecting PEs. If the local memories of PEs are not large enough to store all the local behavior variables, we either ask designers to increase the local memory size, or map some local behavior variables to the global memory, according designers' assumption. The different design assumptions are discussed in section 7.

5.5 Global Variable Mapping

The fourth step of variable-memory mapping is *global variable mapping*, which chooses different variable-memory mapping mechanisms introduced in section 4.4 to map global behavior variables to either local memories or global memories. We will discuss the global variable mapping under different design assumptions in section 7.

5.6 Output

After global-variable mapping, we annotate the variable mapping results into the behavior specification which allows the architecture refinement tool to refine the behavior specification to the specification reflecting the system architecture.

In the design flow, we implement *stack mapping* before *local variable mapping*, implement *local variable mapping* before *global variable mapping*. We choose this mapping sequence because we observed that stack must be mapped to the local memories of PEs, the local variables prefer to be mapped to the local memories of PEs, and the global variables can be mapped to either local memories or global memories, which are described in Table 2. Since sometimes the local memory sizes of PEs are pre-defined, following this mapping sequence helps us to first check the required local memory sizes.

It should be noted that we don't compute the memory required for storing the code instruction when behaviors are mapped to software PEs. The size of code instruction is not changed during the execution of design. Therefore, for the software PEs, we assume the size of local memory of PE used in this report equals to the original size of local memory subtracted by the size of memory required to store the code instruction of behaviors mapped to that PE.

Following the variable-mapping design flow, we make the variable-memory mapping decisions without changing the behavior specification(the behavior specification annotation only add notes to the specification). The complete separating the design space exploration from the behavior specification avoid tedious and time-consuming specification updating, which makes the design space exploration flexible and fast.

6. Memory Size Model

One objective of variable-memory mapping is to minimize memory sizes. To accomplish this objective, we map the variables which have un-overlapped lifetime to the same portion of the memory.

In this section, we first analyze the lifetime of variables on the base of data dependencies among behaviors. According to the lifetime analysis, we then compute the memory size model of each memory. For each memory, the memory size model records the used memory size and the remaining memory size for each behavior. The used memory size of behavior ω on memory ρ denotes the size of memory ρ occupied by the variables that are declared in behavior ω and have been mapped to memory ρ. The remaining memory size of behavior ω on memory ρ denotes the size of memory ρ that can be allocated to the variables that are declared in behavior ω and will be mapped to ρ . By computing the memory size model after mapping each variable, we not only derive the required memory size for local/global memories, but also determine whether the next unmapped variable can be mapping to the local memories of its connecting PEs or the global memory, without exceeding the memory

	Local mem of connecting PEs	Local mem of un-connecting PEs	Global memory
Stack	Mandatory	No	No
Local variable	Prefer	No	Yes
Global variable	Yes	Yes	Yes

Table 2. The variable-memory mapping alternatives for stack, local variable, and global variable.

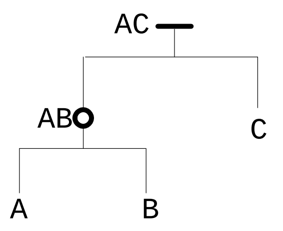


Figure 8. Behavior hierarchy tree of example 1

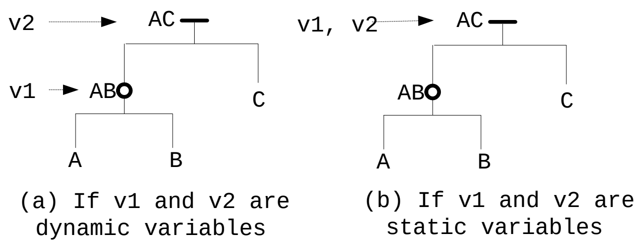

v2
AC
v1
AB
C
A
B
(a) If v1 and v2 are dynamic variables
v1, v2
AC
AB
C
A
B
(b) If v1 and v2 are static variables

Figure 9. Lifetime of behavior variables in the example 1

size limit.

In this section, we first introduce the behavior hierarchy tree representing the behavior hierarchy in section 6.1. Then we analyze the lifetime of variables in section 6.2. In section 6.3, we introduce the memory size model.

6.1 Behavior Hierarchy Tree

We use *behavior hierarchy tree* to describe the behavior hierarchy. The behavior hierarchy tree of behavior example 1 in Figure 5 is displayed in Figure 8.

Behavior hierarchy tree consists of a set of node, each of which represents a behavior. If the node does not have child nodes, such as behavior *A*, *B*, and, *C* in Figure 8, then it represents a leaf behavior. Otherwise, it represents hierarchy behavior, such as behavior *AB* and *AC*. The child behaviors of behavior ω are denoted by the child nodes of the node representing the behavior ω. For example, nodes A and B are the child nodes of node AB. In the following sections, we use both name *node* and *behavior* to represent a behavior in the specification or a node in the hierarchy tree interchangeably We use $S\_B(w)$ to denote the set of child behaviors of behaviors ω. In Figure 8, $S\_B(AC) = \{AB, C\}$.

Child behaviors of behavior ω can be executed either sequentially or concurrently, which is denoted by $parallel(\omega)$. If the child behaviors are executed concurrently, as described by symbol "**o**" in Figure 8, then $parallel(\omega) = 1$. If the child behaviors are executed sequentially, as described by symbol "**-**" in Figure 8, then $parallel(\omega) = 0$. For example, behavior AB contains two parallel executing child behaviors A and B, therefore $parallel(AB) = 1$. In behavior AC, behavior AB is executed before the execution of behavior C, therefore $parallel(AC) = 0$.

We also use $S\_V(\omega)$ to indicate a set of variables associated with behavior/node ω. A behavior/node ω is a **associated behavior/node** of variable v if and only if the lifetime of the variable v and the behavior/node ω are the same. According to the definition, all of the static behavior variables are associated with the behavior representing the entire system behavior, which is denoted by the root node. A dynamic behavior variable is associated with the behavior in which it is declared. For example, if $v1$ and $v2$ in Figure 5 are dynamic variables, then $S\_V(AB) = \{v1\}$, $S\_V(AC) = \{v2\}$, which is shown in Figure 9(a). On the other hand, if $v1$ and $v2$ are static variables, then $S\_V(AB) = \{\}$, $S\_V(AC) = \{v1, v2\}$, which is shown in Figure 9(b).

6.2 Lifetime Analysis

6.2.1 Lifetime of Behavior

First, we define the lifetime of behavior/node. The lifetime of behavior ω is defined as the duration between ω's starting execution time to its ending execution time. Rather than using the exact time to describe the lifetime, we use the lifetime of leaf behaviors to represent the lifetime of all the behaviors. For example, for the behaviors in Figure 8:

$$\begin{aligned}
lifetime(A) =& lifetime(A);\\
lifetime(B) =& lifetime(B);\\
lifetime(C) =& lifetime(C);\\
lifetime(AB) =& lifetime(A) \bigcup lifetime(B);\\
lifetime(AC) =& lifetime(AB) \bigcup lifetime(C)\\
=& lifetime(A) \bigcup lifetime(B) \bigcup\\
& lifetime(C)
\end{aligned}$$

If both two nodes in the behavior hierarchy tree contain the $lifetime$ of the same leaf node, then they are overlapped. For example, node B and node AB are overlapped because both of them contains $lifetime(B)$. However, if two nodes don't contain the lifetime of the same leaf node, we can not conclude that they are un-overlapped. This is because whether two leaf nodes are overlapped are not determined.

Based on the fact that two sequential executing behaviors are un-overlapped and two parallel executing behaviors are overlapped, we define the following three rules to determine whether two nodes are overlapped to each other.

Rule1: For a node ω, if $parallel(\omega) = 0$(sequential), then all of its child nodes are un-overlapped to each other.

Rule2: If nodes $\omega1$ and $\omega2$ are un-overlapped nodes, then any node in the sub-tree lead by $\omega1$ is un-overlapped to the any node in the sub-tree lead by $\omega2$.

Rule3: If two nodes do not satisfy the rule 1 & 2, they are overlapped.

For example, in Figure8, nodes C and AB are un-overlapped because they satisfy the Rule1. Nodes C and A are also un-overlapped because they satisfy the Rule2.

6.2.2 Lifetime of Stack

The lifetime of stack of leaf node ω is equal to the lifetime of node ω. For example, in Figure 8,

$$\begin{aligned}
lifetime(stack(A)) &= lifetime(A);\\
lifetime(stack(B)) &= lifetime(B);\\
lifetime(stack(C)) &= lifetime(C);
\end{aligned}$$

6.2.3 Lifetime of Dynamic Behavior Variable

The lifetime of a dynamic behavior variable v is equal to the lifetime of v's associated node ω. For example, for the example in Figure 5, if $v1$ and $v2$ are dynamic variables as shown in Figure 9(a), then

$$\begin{aligned}
lifetime(v1) &= lifetime(AB)\\
lifetime(v2) &= lifetime(AC)
\end{aligned}$$

6.2.4 Lifetime of Static Behavior Variable

The lifetime of the static behavior variable v is equal to the lifetime of v's associated node ω. For example, for the example in Figure 5, if $v1$ and $v2$ are static variables as shown in Figure 9(b), then

$$\begin{aligned}
lifetime(v1) &= lifetime(AC)\\
lifetime(v2) &= lifetime(AC)
\end{aligned}$$

6.3 Memory-Size Model

In this seciton, we define the memory size model.

For each pair (ω, ρ), where ω denotes a node in the behavior hierarchy tree, and ρ denotes a local memory of PE or a global memory, we define the memory size model as

$$M(\omega, \rho) = (\alpha(\omega, \rho), \beta(\omega, \rho), \lambda(\omega, \rho))$$

where α denotes the self-used memory size, β denotes the hierarchically-used memory size, and λ denotes remaining memory size.

6.3.1 Self-Used Memory Size α

$\alpha(\omega, \rho)$ denotes node ω's self-used memory size in memory ρ, which represents the occupied memory size of ρ by the ω's associated variables and stacks that have been mapped to ρ.

$a(\omega, \rho)$ is defined as:

If ω is a leaf node, then

$$\alpha(\omega, \rho) = size(stack(\omega), \rho) + \sum_{v_i \in S\_MV(\omega,\rho)} size(v_i, \rho)$$

If ω is a non-leaf node, then

$$a(\omega, \rho) = \sum_{v_i \in S\_MV(\omega,\rho)} size(vi, \rho)$$

where $S\_MV(\omega, \rho)$ denotes the set of ω's associated variables which have been mapped to ρ.

6.3.2 Hierarchically-Used Memory Size β

$\beta(\omega, \rho)$ denotes ω's hierarchically-used memory size in ρ. It does not only contain self-used memory size $\alpha(\omega, \rho)$, but also contains the used memory size of the nodes in the sub-tree lead by ω.

$\beta(\omega, \rho)$ is defined as:

If $parallel(\omega, \rho) = 1$, then

$$\beta(\omega, \rho) = \alpha(\omega, \rho) + \sum_{\theta \in S\_B(\omega)} \beta(\theta, \rho)$$

If $parallel(\omega, \rho) = 0$, then

$$\beta(\omega, \rho) = \alpha(\omega, \rho) + \max_{\theta \in S\_B(\omega)} \beta(\theta, \rho)$$

where $S\_B(\omega)$ denotes the set of ω's child nodes. $parallel(\omega, \rho)$ represents whether the child nodes of ω are executed concurrently(=1) or sequentially(=0) in memory ρ, which is explained in section 6.3.4.

The formulation is achieved according to the following fact: if the child nodes of ω are executed concurrently, then the lifetime of them are overlapped. Therefore, the memory required by ω's child nodes equals to sum of hierarchically-used memory sizes of ω's child nodes.

On the other hand, if child nodes of ω are executed sequentially, then the lifetime of them are un-overlapped. Therefore, The child nodes can share the same port of memory ρ. Therefore, the memory required by ω's child nodes equals to the largest hierarchically-used memory size of ω's child behaviors.

6.3.3 Remaining Memory Size λ

$\lambda(\omega, \rho)$ denotes ω's remaining memory size in ρ, which represent the available memory size of ρ for unmapped variables associated with ω. It can be computed only when memory size of ρ $size(\rho)$ is known.

$\lambda(\omega, \rho)$ is defined as:

If ω is the root node, then

$$\lambda(\omega, \rho) = size(\rho) - \beta(\omega, \rho)$$

Otherwise,

if $parallel(Parent(\omega), \rho) = 1$,

$$\lambda(\omega, \rho) = \lambda(parent(\omega, \rho))$$

if $parallel(Parent(\omega), \rho) = 0$,

$$\lambda(\omega, \rho) = \lambda(Parent(\omega), \rho) + \beta(Parent(\omega), \rho) - \alpha(Parent(\omega), \rho) - \beta(\omega, \rho).$$

where $Parent(\omega)$ denotes the parent node of node ω.

The formulation is achieved according to the following fact: if ω is the root node, then remaining memory size equals to the total memory size $size(\rho)$ subtracted by used memory size $\beta(\omega, \rho)$.

If ω is not the root node, then there are two possibilities. First, if $parallel(Parent(\omega)) = 1$, then ω is concurrently executed with $parent(\omega)$'s other child nodes. In this case, node ω needs to reserve memory not only for itself, but also for $parent(\omega)$'s other child node because the lifetime of them are overlapped. As a result, $\lambda(\omega, \rho)$ equals to $\lambda(parent(\omega, \rho))$.

On the other hand, if $parallel(parent(\omega)) = 0$, then ω is executed sequentially with $parent(\omega)$'s other child nodes. In this case, ω and $parent(\omega)$'s other child behaviors can share the same memory portion. Because $parent(\omega)$ reserves memory for the largest hierarchically-used memory size of its child nodes and ω only reserves the memory to store hierarchically-used memory size of itself, the difference between them is added on $\lambda(parent(\omega, \rho))$ to represent $\lambda(\omega, \rho)$. Since the largest hierarchically-used memory size of $parent(\omega)$'s child nodes equals to $\beta(parent(\omega), \rho)$ subtracted by $\alpha(Parent(\omega), \rho)$. Therefore, $\lambda(\omega, \rho) = \lambda(Parent(\omega), \rho) + \beta(Parent(\omega), \rho) - \alpha(Parent(\omega), \rho) - \beta(\omega, \rho)$.

6.3.4 Memory Type Model

There are three types of memories, local memory of SW PE (named *SW*), local memory of HW PE (named *HW*), and global memory (named *Global)*. We compute $parallel(\omega, \rho)$ according to the type of ρ. If $parallel(\omega, \rho)$ equals to 0, then it indicates that at any time ρ only reserves memory for one of ω's child nodes that are mapped to ρ. Otherwise, ρ must reserve memory for all of ω's child nodes that are mapped to ρ.

In any SW PE, all the nodes mapped to it are executed sequentially. However, if SW PE's operation system supports preemptive schedule, then one behavior may be preempted by another. In this case, local memory ρ of the SW PE reserves memory not only for the preempting node, but also for the preempted node. Also, one node can only be preempted by its overlapped nodes. Therefore, we define,

```
if ($\rho = SW$) then
    if ($os\_preemptive(\rho) = 1$)
        then if $parallel(\omega) = 1$
            then $parallel(\omega, \rho) = 1$;
        if $parallel(\omega) = 0$
            then $parallel(\omega, \rho) = 0$;
    else // ($os\_preemptive(\rho) = 0$)
        $parallel(\omega, \rho) = 0$;
```

In any HW PE, all the nodes mapped to it are executed sequentially. Since no preemptive schedule is allowed. Therefore, we define,

if $\rho = HW$, then $parallel(\omega, \rho) = 0$;

The parallel analysis for global memory is a little more complex. In general, if the lifetime of child nodes of node ω are overlapped in terms of functionality, the global memory ρ must reserve the memory for all the variables of these child nodes that are mapped to it. This is because that different child nodes may be mapped to different PEs, and nodes in different PEs can be executed concurrently if the functionality allows concurrent execution. However, all the nodes in the same HW PE have been sequentialized. As a result, we disable the concurrency existing in the sub-tree which is mapped to the same HW PE. This rules can also be applied to the node for the same SW PE with non-preemptive schedule. Therefore, we define:

```
if ($\rho = Global$)
    then if $parallel(\omega) = 0$
        then $parallel(\omega, \rho) = 0$;
        else // $parallel(\omega) = 1$
            if all the nodes in the sub_tree lead by $\omega$ are mapped to the same HW PE,
                then $parallel(\omega, \rho) = 0$;
                else if all the nodes in the sub_tree lead by $\omega$ are mapped to the same SW PE $\rho$ and $os\_preemptive(\rho) = 0$,
                    then $parallel(\omega, \rho) = 0$;
                    else $parallel(\omega, \rho) = 1$;
```

We illustrate $parallel(\omega, \rho)$ computation by Figure 10. Figure 10(a) shows the original behavior hierarchy tree reflecting the system behavior. For any node ω, "**o**" denotes $parallel(\omega) = 1$ while "**-**" denotes $parallel(\omega) = 0$. Assume we map the nodes in dotted circle to a HW PE $PE2$ and map other nodes to a SW PE $PE1$ with preemptive schedule. Figure 10(b),(c), and (d) display the value of $parallel(\omega, \rho)$ in the behavior hierarchy tree, for the local memory of $PE1$(SW), the local memory of $PE2$(HW), and the global memory respectively. In these figures, "**o**" denotes $parallel(\omega, \rho) = 1$ while " **-**" denotes $parallel(\omega, \rho) = 0$.

6.3.5 Variable-Memory Mapping Judgement

After defining memory size model, we define two judgements for variable-memory mapping.

Judgement1: The required memory size of memory ρ for node ω equals to $\beta(\omega, \rho)$. The required memory size of memory ρ for design equals to $\beta(\varpi, \rho)$, where ϖ denotes the root node .

Judgement2: Assuming an unmapped variable v is associated with node ω. For any memory ρ, if $\lambda(\omega, \rho) < size(v, \rho)$, then variable v cannot be mapped to the memory ρ. Otherwise, it can be mapped to memory ρ.

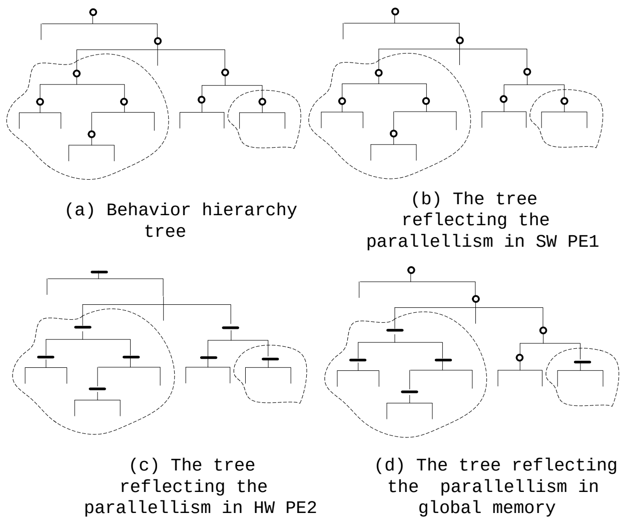

(a) Behavior hierarchy tree
(b) The tree reflecting the parallellism in SW PE1
(c) The tree reflecting the parallellism in HW PE2
(d) The tree reflecting the parallellism in global memory

Figure 10. Example of $parallel(\omega, \rho)$ computation for local memory of SW PE, local memory of HW PE and global memory.

7. Problem Definition and Solution

Different designs have different assumptions for system behavior and system architecture. They also have different given constraints such as memory size constraint and time constraint. In this section, we derive five variable-memory mapping problems according to the most common applied design assumptions and given design constraints. We solve these problems based on the memory size model introduced in section 6.

To illustrate the problems and the solutions, we use a system behavior described in Figure 11 as an design example. Figure 12 displays its behavior hierarchy tree.

During the design process, we adopt a system architecture described in Figure 13, which contains two PEs. $PE1$ is a SW PE. $PE2$ is a HW PE. Each PEs has a local memory LM. The global memory in the system architecture is denoted by GM.

In the system behavior, there are four behavior variables $v1$, $v2$, $v3$, and $v4$. There are six stacks of leaf behaviors, $stack(A)$, $stack(B)$, $stack(C)$, $stack(D)$, $stack(E)$, and $stack(F)$. When a variable or a stack is mapped to different memories, the size of its occupied memory is different. For example, an integer variable may occupy 16 bit in 16-bit microprocessor but occupies 32 bit in 32-bit microprocessor. Table 3 and Table 4 displays the occupied memory size of variables and stacks of leaf behaviors on different memories. We use spec profiler[3] to compute the occupied memory sizes.

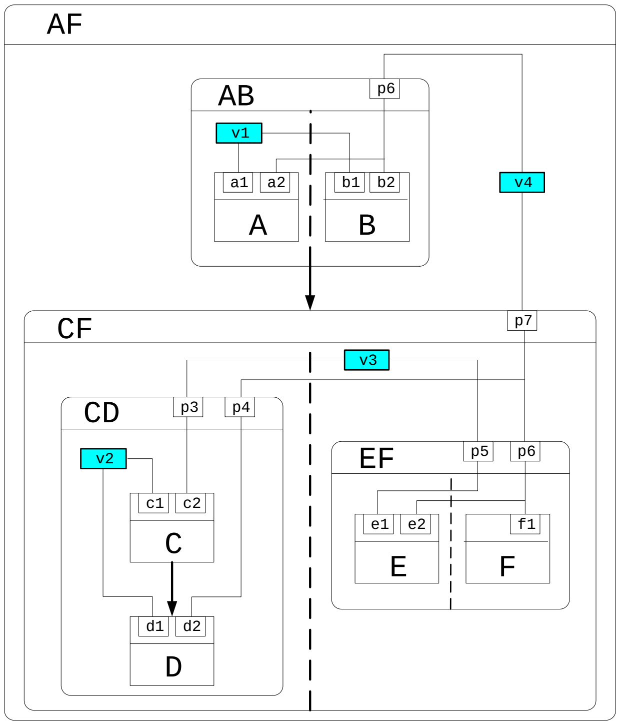


Figure 11. System behavior example 2

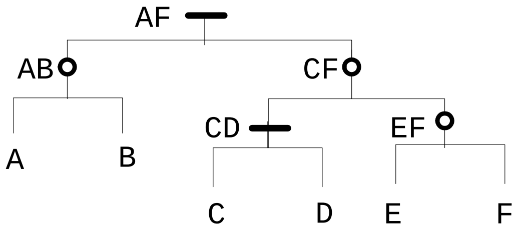


Figure 12. Behavior hierarchy tree of example 2

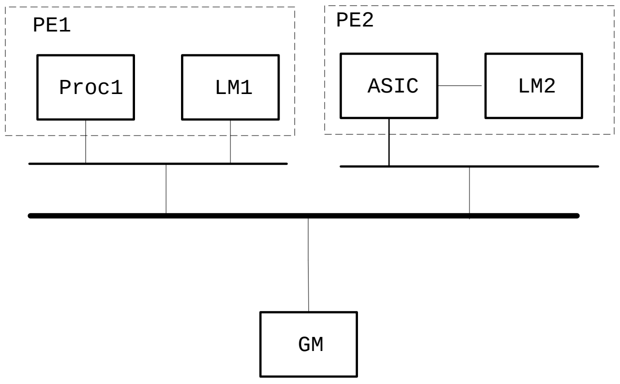


Figure 13. The system architecture for behavior example 2

size (byte)	LM in PE1	LM in PE2	GM
v1	7	14	14
v2	10	20	20
v3	9	18	18
v4	5	10	10

Table 3. Occupied memory sizes of variables in example 2

size (byte)	LM in PE1	LM in PE2	GM
A	1	2	2
B	2	3	4
C	3	5	6
D	4	6	8
E	5	8	10
F	6	10	12

Table 4. Occupied memory sizes of stacks of leaf behaviors in example 2

7.1 Design Problem 1

7.1.1 Design Assumption

Problem 1 allows two types of variables: function variable and static behavior variable. Problem 1 allows only message-passing mechanism. Each variable has a local copy in the local memory of connecting PEs. Since problem 1 doesn't allow shared-memory mechanism, there is no global memory in the system architecture.We assume that the sizes of local memories of PEs are unknown.

7.1.2 Problem Definition

The goal of problem 1 is to find the minimal sizes of local memories of PEs.

7.1.3 Solution

We follow the design flow described in section 5 to solve problem 1.

Global/Local Variable Identification We first identify global/local variables following the approach introduced in section 5.2. In the example, if we map behavior A, B, and C to $PE1$, and D, E, and F to $PE2$, as shown in Figure 14, then $v1$ is a local variable of $PE1$ and $v2$, $v3$, and $v4$ are the global variables which are connected to both $PE1$ and $PE2$. The variable-PE connection table is displayed in Table 5.

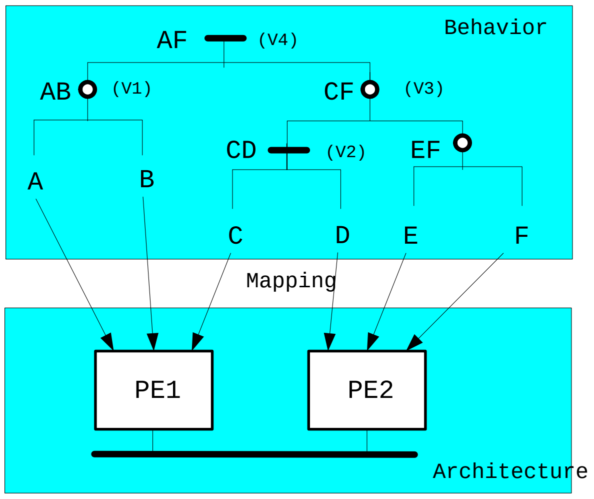


Figure 14. Behavior-PE mapping solution of the example 2.

Variable	v1	v2	v3	v4
PE1	Yes	Yes	Yes	Yes
PE2	No	Yes	Yes	Yes

Table 5. Variable-PE connection table in example 2

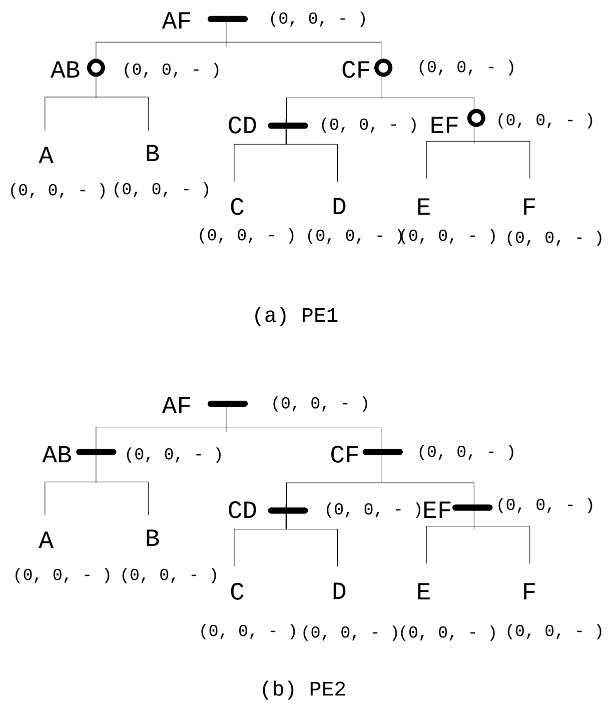


Figure 15. Initial memory size models for PE1 and PE2.

Stack Mapping First, we generate the behavior hierarchy tree for the local memories of *PE1* and *PE2*. Symbol ”**-**” represents $parallel(\omega, \rho) = 0$ and symbol ”**o**” represents $parallel(\omega, \rho) = 1$. Initially, we assign $(0, 0, -)$ to each (α, β, λ) in the memory size models . The value of λ is ”-”, which refers to ”not consider”. This is because for design problems 1, the local memory sizes are unknown. The initial behavior hierarchy trees and the memory size models are displayed in Figure 15.

Second, we add the stack size to the self-used memory size α of the memory models. Since behaviors A, B, and C are mapped to *PE1*, their stacks sizes are added to the memory size model of *PE1*. Similarly, the stacks of behaviors D, E, and F are added to the memory size model of *PE2*. The self-used memory size α and the hierarchically-used memory size β are computed accordingly. The resulting memory size models are displayed in Figure 16.

Local Variable Mapping We map local variables to the local memories of their connecting PEs. Since all the variables are static variables, they are associated with the root node. As a result, we adds them into the α of the root node. In our example, since Table 5 tells that $v1$ is the local variable of $PE1$, in the memory size model of $PE1$,

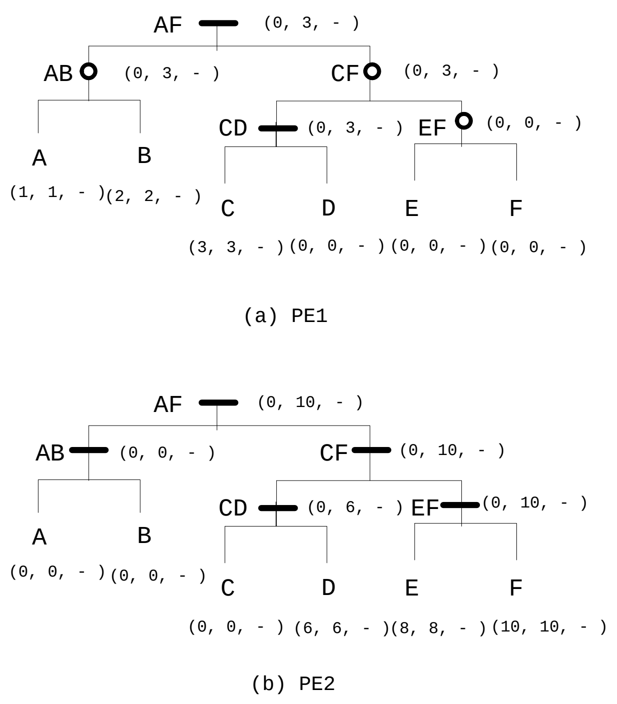


Figure 16. Memory size models of PE1 and PE2 after stack memory mapping.

$$\begin{aligned}\alpha(AF, PE1) =&\alpha(AF, PE1) + size(v1, PE1)\\ =&0 + 7 = 7(byte)\end{aligned}$$

After computing α, we update β in the memory size model. In our example, since $parallel(AF, PE1) = 0$,

$$\begin{aligned}\beta(AF, PE1) =&\alpha(AF, PE1)+\\ &\max(\beta(AB, PE1), \beta(CF, PE1))\\ =&7 + \max(3, 3) = 10(byte)\end{aligned}$$

The resulting memory size models are displayed in Figure 17.

After local variable mapping, the required memory sizes of PE1 and PE2 are both 10 bytes.

Global Variable Mapping We map the global variables to the local memories of its connecting PEs. Similar to local variable mapping, since the all the variables are static variables, they are associated with the root node. We first add them to α of the root node. In our example, Table 5 tells that global variables $v2$, $v3$, and $v4$ are connected to both $PE1$ and $PE2$,

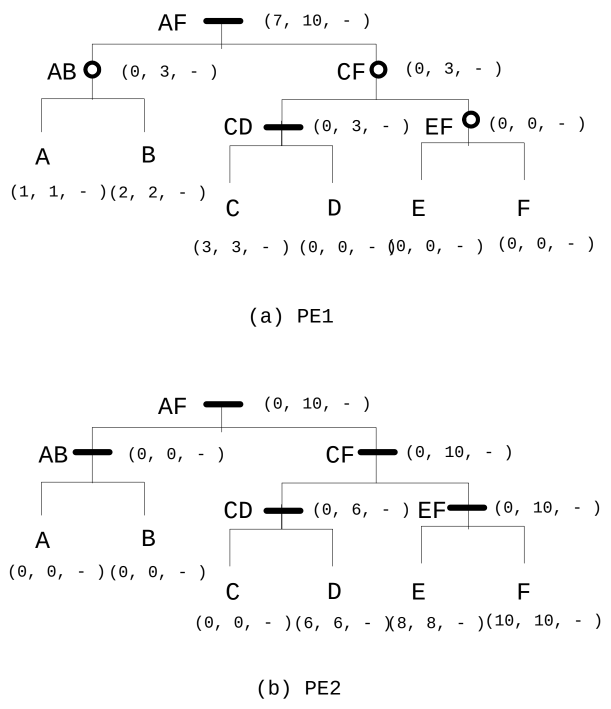


Figure 17. Memory size models of PE1 and PE2 after local variable mapping in problem 1.

$$\begin{aligned}\alpha(AF,PE1) =& \alpha(AF,PE1) + size(v2,PE1) + \\ & size(v3,PE1) + size(v4,PE1) \\ =& 7 + 10 + 9 + 5 \\ =& 31(byte)\end{aligned}$$

$$\begin{aligned}\alpha(AF,PE2) =& \alpha(AF,PE2) + size(v2,PE2) + \\ & size(v3,PE2) + size(v4,PE2) \\ =& 0 + 20 + 18 + 10 \\ =& 48(byte)\end{aligned}$$

After updating α, we update β in the memory size model,

$$\begin{aligned}\beta(AF,PE1) =& \alpha(AF,PE1) + \\ & \max(\beta(AB,PE1),\beta(CF,PE1)) \\ =& 31 + \max(3,3) \\ =& 34(byte)\end{aligned}$$

$$\begin{aligned}\beta(AF,PE2) =& \alpha(AF,PE2) + \\ & \max(\beta(AB,PE2),\beta(CF,PE2)) \\ =& 48 + \max(0,10) \\ =& 58(byte)\end{aligned}$$

The resulting memory size models are displayed in Figure 18.

Result The required minimal sizes of local memories of PEs equal to β of the root nodes in memory size model of ρ. In the example, the minimal local memory size of *PE1* is 34 byte. The minimal local memory size of *PE2* is 58 byte. Based on the computed minimal local memory sizes, designers can select the local memory sizes accordingly.

7.2 Design Problem 2

7.2.1 Design Assumption

The design assumption of problem 1 and problem 2 are the same except the supported variable types. In comparison to problem 1 that allows function variable and static behavior variable, problem 2 allows function variable and dynamic behavior variable.

The difference between static behavior variable and dynamic behavior variable are their lifetime and their associated nodes. If a variable is static, it associates with the root node. On the hand, if a variable is dynamic, it associates with the node that it is declared in.

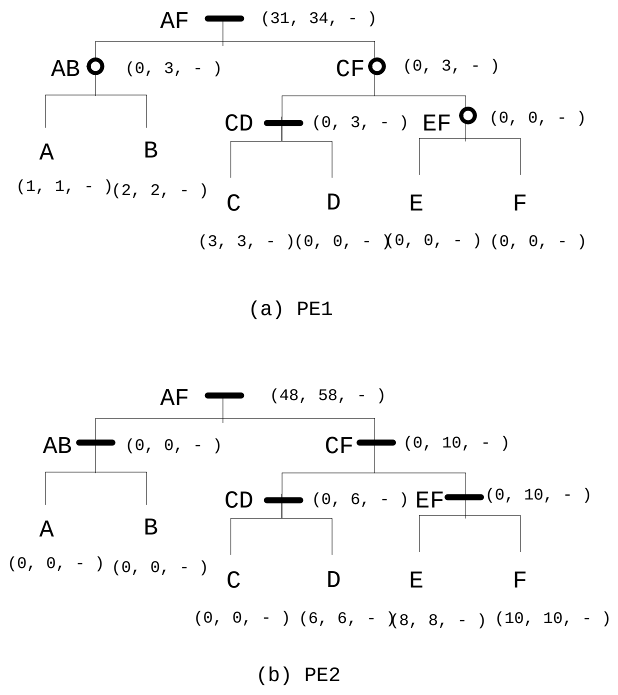


Figure 18. Memory size models of PE1 and PE2 after global variable mapping in problem 1.

7.2.2 Problem Definition

The goal of problem 2 is to find the minimal sizes of local memories of PEs.

7.2.3 Solution

Similar to problem 1, we follow the design flow described in section 5 to solve this problem.

Global/Local Variable Identification The step of global/local variable identification is the same as it in the problem 1.

Stack Mapping The step of stack mapping is the same as it in the problem 1.

Local Variable Mapping We first reserve local memories for local variables in their connecting PEs. For example, Table 5 tells that $v1$ is the local variable of $PE1$, and $v1$ is associated with node AB

$$\begin{aligned}\alpha(AB,PE1) &= \alpha(AB,PE1) + size(v1,PE1) \\ &= 0 + 7 = 7(byte)\end{aligned}$$

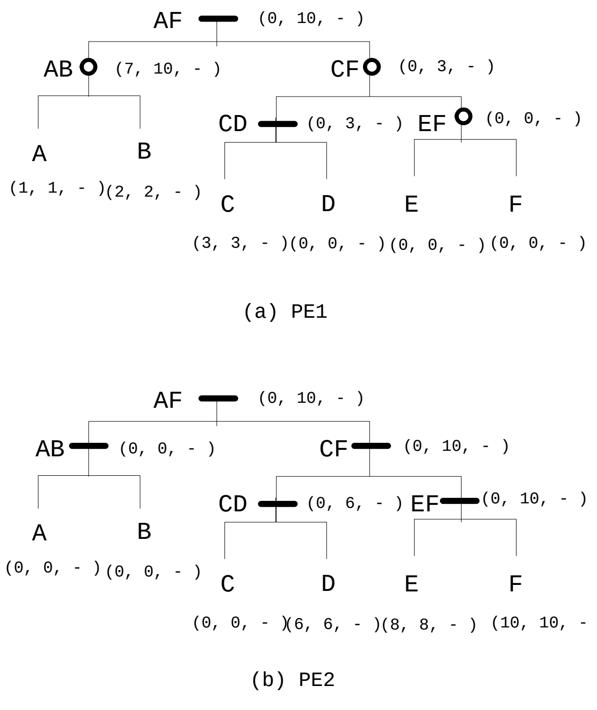


Figure 19. Memory size models of PE1 and PE2 after local variable mapping in problem 2

After computing α, we update β in the memory size model according. In the example,

$$\begin{aligned}\beta(AB, PE1) =&\alpha(AB, PE1)+\\ &(\beta(A, PE1) + \beta(B, PE1))\\ =&7 + (1 + 2) = 10(byte)\end{aligned}$$

$$\begin{aligned}\beta(AF, PE1) =&\alpha(AF, PE1)+\\ &\max(\beta(AB, PE1), \beta(CF, PE1))\\ =&0 + \max(10, 3) = 10(byte)\end{aligned}$$

The resulting memory size models are displayed in Figure 19.

After local variable mapping, the required memory sizes of *PE1* and *PE2* are both 10 bytes.

Global Variable Mapping We map the global variables to the local memories of it connecting PEs . In the example, variables *v2*, *v3*, and *v4* are connected to both *PE1* and *PE2*. We compute α in the memory size models of PE1 and PE2 as follows:

$$\alpha(CD, PE1)+ = size(v2, PE1) = 10;$$
$$\alpha(CF, PE1)+ = size(v3, PE1) = 9;$$
$$\alpha(AF, PE1)+ = size(v4, PE1) = 5;$$

$$\alpha(CD, PE2)+ = size(v2, PE2) = 20$$
$$\alpha(CF, PE2)+ = size(v3, PE2) = 18$$
$$\alpha(AF, PE2)+ = size(v4, PE2) = 10$$

We them compute β for nodes in *PE1*,

$$\begin{aligned}\beta(AB, PE1) =&\alpha(AB, PE1) + (\beta(A, PE1)+\\ &\beta(B, PE1))\\ =&7 + (2 + 1) = 10(bytes)\\ \beta(CD, PE1) =&\alpha(CD, PE1)+\\ &\max(\beta(C, PE1), \beta(D, PE1))\\ =&10 + max(3, 0) = 13\\ \beta(CF, PE1) =&\alpha(CF, PE1) + (\beta(CD, PE1)+\\ &\beta(EF, PE1))\\ =&9 + (13 + 0) = 22;\\ \beta(AF, PE1) =&\alpha(AF, PE1)+\\ &\max(\beta(AB, PE1), \beta(CF, PE1))\\ =&5 + \max(10, 22) = 27\end{aligned}$$

The β for nodes in PE2 are also computed accordingly. The updated memory size models are displayed in Figure 20.

Result In problem 2, the required memory sizes of *PE1* is 27 byte, the memory size of *PE2* is 54 byte, which are 7 bytes and 4 bytes smaller than the result computed in problem 1. This is because the some of the global variables can share the same portions of memories. After computing required memory sizes, designers can select the local memory accordingly.

7.3 Design Problem 3

7.3.1 Design Assumption

Problem 3 allows three types of variables: function variable, static behavior variable, and dynamic behavior variable. Furthermore, problem 3 not only allows message-passing communication mechanism, but also allows shared-memory communication mechanism. Local variables must be mapped to the local memory of its connecting PE. A global variable can be either mapped to the local memory of each of its connecting PEs, or to the global memory.

In the example, $v4$ is the static variable. $V1$, $v2$, and $v3$ are dynamic variables.

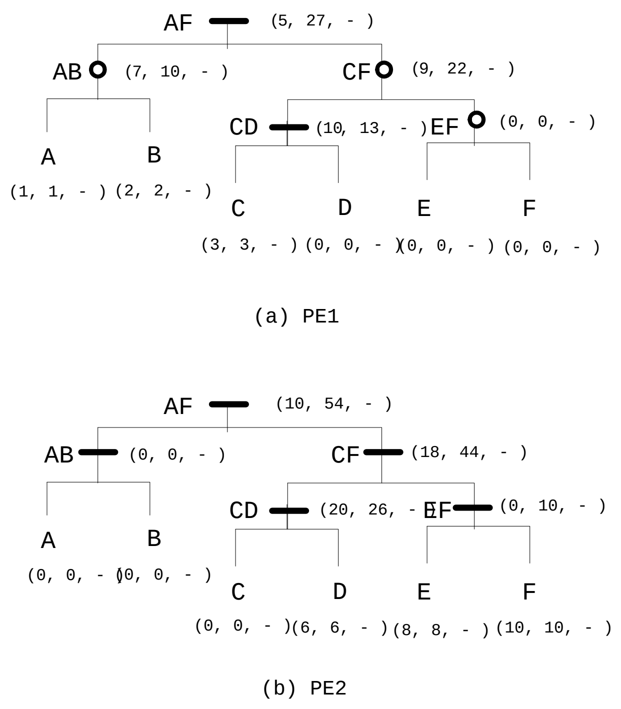


Figure 20. Memory size models of PE1 and PE2 after global variable mapping in problem 2.

7.3.2 Problem Definition

Assuming designers have selected the local memory sizes of PEs, the goal is to find a variable-memory mapping solution to minimize the required global memory size and to minimize the traffic amount on the interconnection network of the system architecture.

To help designers to select the local memory sizes of PEs, we can compute the lower-bounds and upper-bounds of the local memories sizes. The lower-bounds of local memory sizes equal to the β of root nodes after the local variable mapping step of problem 1 and problem 2, which guarantees that all the stacks and local variables are mapped to local memories. The upper-bounds of local memory sizes equal to the β of root nodes after the global variable mapping step of problem 1 and problem 2, which guarantees that all the stacks and variables are mapped to local memories. The selected local memories size must be greater than the lower-bounds. If all of the selected local memories sizes are greater than the upper-bounds, then the global memory is not needed in the system architecture.

7.3.3 Solution

Similar to problem 1, we follow the design flow described in section 5 to solve this problem.

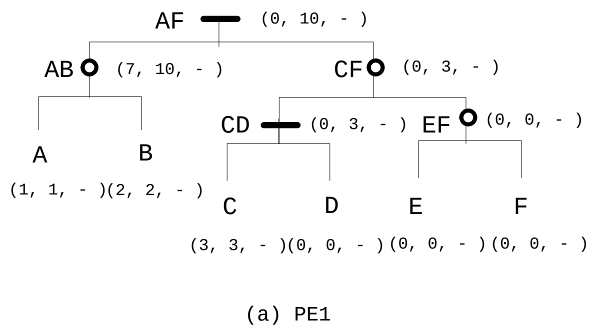
Global/Local Variable Identification The global/local variable identification in problem 3 is the same as it in problem 1.

Stack Mapping The stack mapping in problem 3 is the same as it in problem 1.

Local Variable Mapping The local variable mapping in problem 3 is the similar to it in problem 2. In the given example, the memory size models after local variable mapping are displayed in Figure 21.

Global Variable Mapping In the problem 3, the local memories may not be large enough to store every global variable. As a result, we must map some global variables to the global memory. We determine which global variables are mapped to the global memory according the amount of traffic of variables.

Global Variable Traffic Computation For each global variable v, if we choose shared-memory communication mechanism and map it to the global memory, then the amount of traffic that v generates on the interconnection network equals to



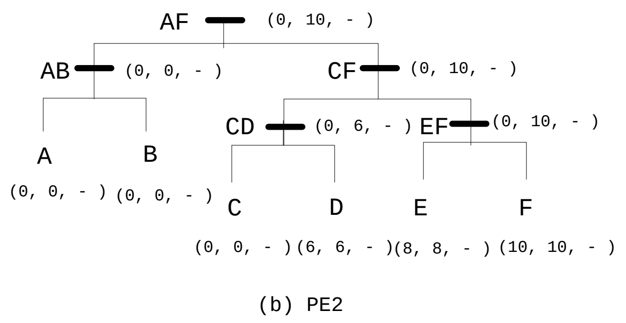


Figure 21. Memory size models of PE1 and PE2 after local variable mapping in problem 3.

$$traffic(v, global\_mem) = read(v) + write(v)$$

where $read(v)$ is the amount of v's read access and $write(v)$ is the amount of v's write access.

However, if we choose message-passing communication mechanism and map v to local memories of its connecting PEs, the amount of traffic v generates on the interconnection network equals to

$$traffic(v, local\_mem) = write(v)$$

This is because when a node ω read from v, it reads from the local memory of the PE to which ω is mapped. On the contrary, when a node ω write to v, it updates v's local copies in all the local memories of v's connecting PEs.

Obviously, the difference of the amount of traffic between two communication mechanisms is $read(v)$. As a result, we prefer mapping the global variables with larger amount of read access to local memories.

Global Variable Mapping Algorithm We first order global variables in the list $Global\_Var$ in the decreasing order of read access. Then we map one global variable v at each iteration. If all of the local memories of v's connecting PEs have enough unused memory to store v, then we map v to these local memories. Otherwise, we map v into the global memory. Whether variable v can be mapped to the local memory is based on *Judgement2* in section 6.3.5. After the mapping decision of v is made, the α, β, and λ of corresponding memory size models are updated. The global variable mapping algorithm is described in Figure 22.

Traffic	v2	v3	v4
Read(Kbyte)	60	25	35
Write(Kbyte)	14	16	12

Table 6. Variable traffic in example 2

In the given example, we assume that designers select 20-byte local memory for *PE1* and 40-byte local memory for *PE2*. The traffic of global variables $v2$, $v3$, and $v4$ are displayed in Table 6. The algorithm first sorts variables according to their read traffic. The mapping order is $v2$, $v4$, and $v3$.

Figure 23 shows the memory size models before global variable mapping. Since the local memory sizes are given, we compute the λ of the memory size models for the local memories. On the other hand, since the memory size of the global memory is unknown, we set λ of the global memory as "-", which refers "not consider".

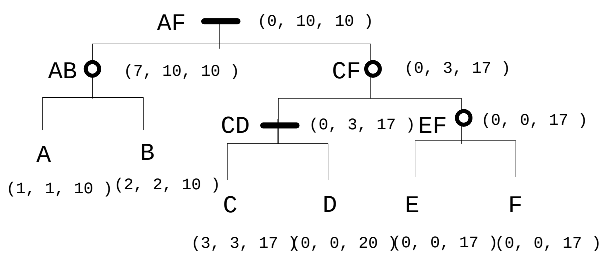
First, we map $v2$ to the system architecture. The $v2$ is associated with node CD. The $\lambda(CD, PE1)$ is 17 and $\lambda(CD, PE2)$ is 34. $Size(v2, PE1)$ is 10 and $size(v2, PE2)$ is 20. Since $\lambda(CD, PE1)$ is greater than $size(v2, PE1)$ and $\lambda(CD, PE2)$ is greater than $size(v2, PE2)$, we can map $v2$ to local memories of PEs. The updated memory size models after $v2$ mapping are displayed in Figure 24.

Second, we map $v4$ to the system archiecture. The $v4$ is decleared in node AF. The $\lambda(AF, PE1)$ is 7 and $\lambda(AF, PE2)$ is 14. The $size(v4, PE1)$ is 5 and the $size(v4, PE2)$ is 10. Since $\lambda(AF, PE1)$ is greater than $size(v4, PE1)$, and $\lambda(AF, PE2)$ is greater than $size(v4, PE2)$, we can map $v4$ to the local memories of PEs. The updated memory size models after $v4$ mapping are displayed in Figure 25.

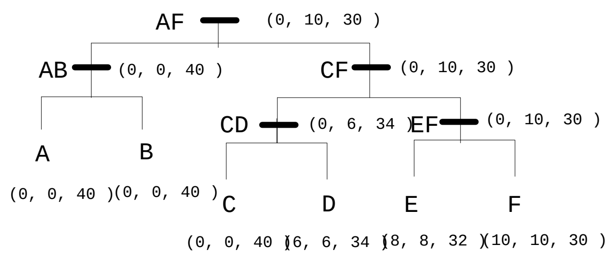
Last, we map $v3$ to the system archiecture. The $v3$ is decleared in node CF. The $\lambda(CF, PE1)$ is 2 and $\lambda(CF, PE2)$ is 4. The $size(v3, PE1)$ is 9 and the $size(v3, PE2)$ is 18. Since $\lambda(AF, PE1)$ is smaller than $size(v3, PE1)$, and $\lambda(AF, PE2)$ is smaller than $size(v3, PE2)$, we cannot map $v3$ to the local memories of PEs. Therefore, we map $v3$ to the global memory. The updated memory size models after $v3$ mapping are displayed in Figure 26. After mapping , $\beta(AF, Global\_Mem)$ is 9 byte, which equals to the required global memory size.

```
map_to_local = TRUE;
Global_Var = SortVariableBasedOnRead();

// Check whether v can be mapped to the local
//memories of all of its connecting PEs
for v in Global_Var do
    $\omega$ = AssoicateBehavior(v);
    for each $\rho$ to which v is connecting do
        if ($\lambda(\omega, \rho)$ < size(v, $\rho$) ) do
            map_to_local = FALSE
        endif
    endfor

    //Update memory size models according
    //to v’s mapping decision.
    if ( map_to_local == TRUE) do
        for each $\rho$ to which v is connecting do
            $\alpha(\omega, \rho)$ = $\alpha(\omega, \rho)$ + size(v, $\rho$) ;
            Update($\beta$, $\rho$);
            Update($\lambda$, $\rho$);
        endfor
    else
        $\alpha(\omega$, Global_Mem) = $\alpha(\omega$, Global_Mem)
                      + size(v, Global_Mem);
        Update($\beta$,Global_Mem);
        Update($\lambda$,Global_Mem);
    endif
endfor
```

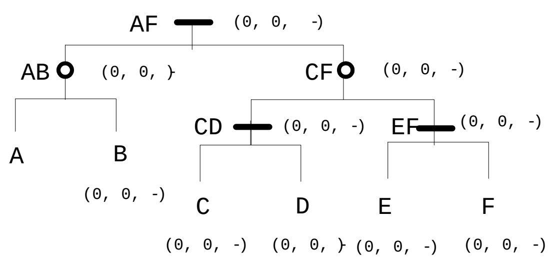
Figure 22. Algorithm 1: global variable mapping algorithm for problem 3.



(a) PE1

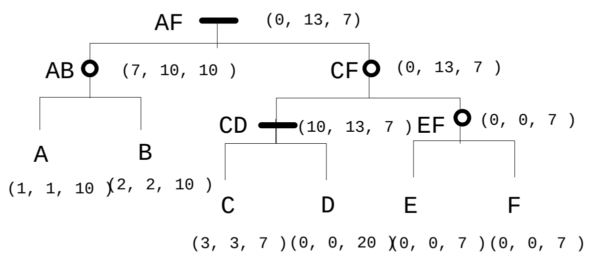


(b) PE2

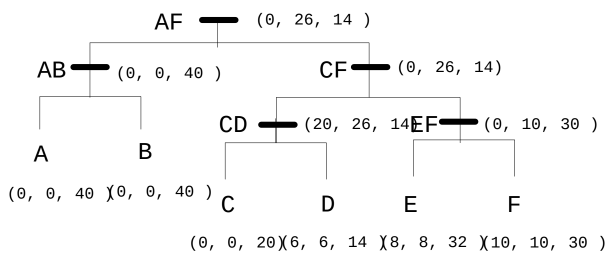


(c) Global memory

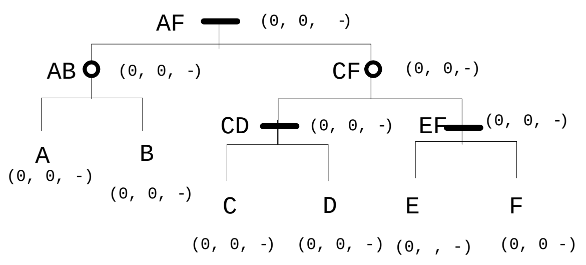
Figure 23. Memory size models before global variable mapping in problem 3.



(a) PE1



(b) PE2



(c) Global memory

Figure 24. Memory size models after mapping global variable v2 to local memories in problem 3

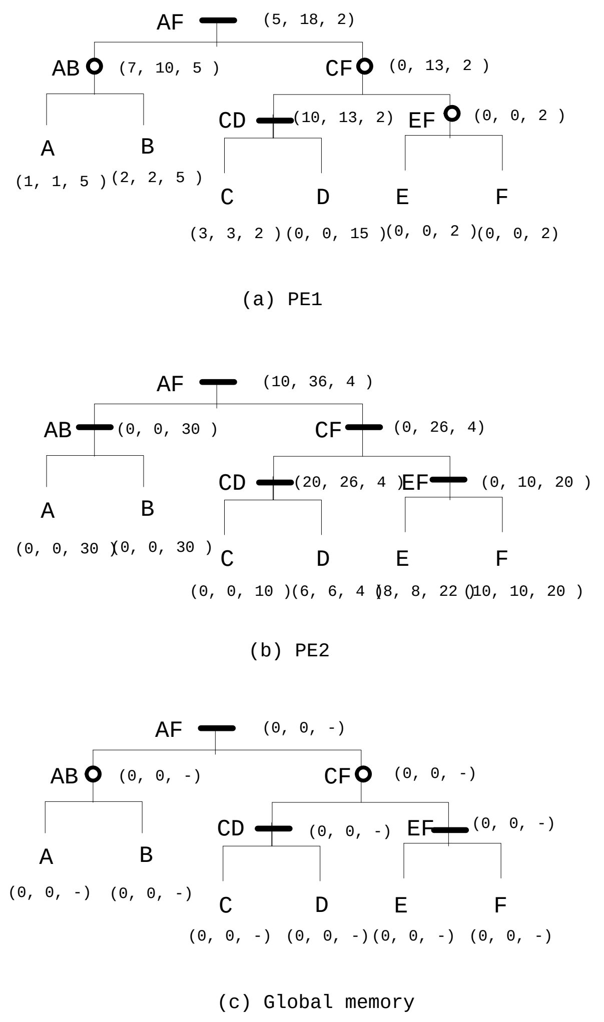


Figure 25. Memory size models after mapping global variable v4 to local memories in problem 3.

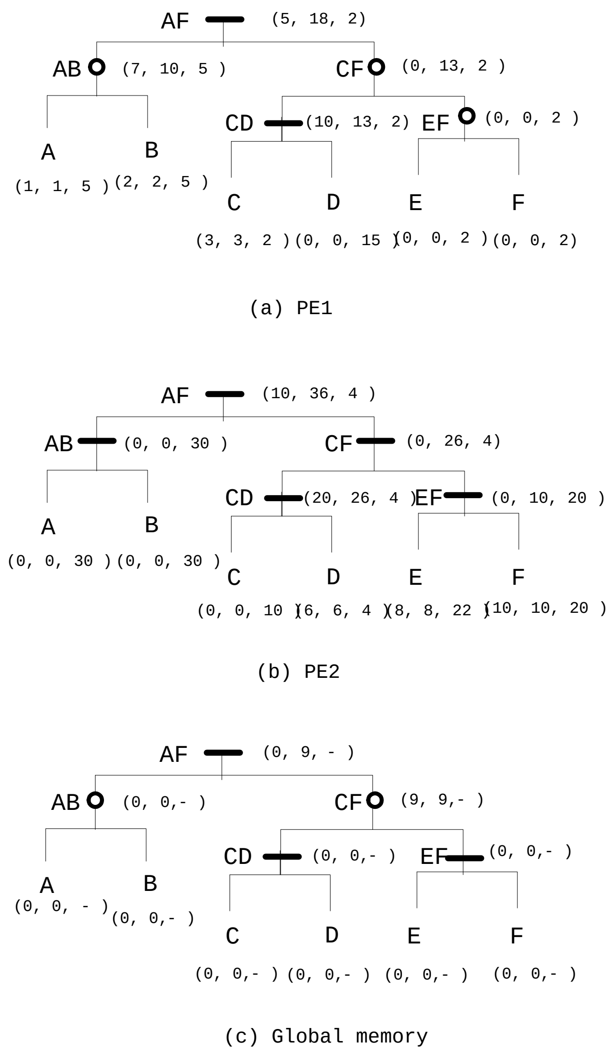


Figure 26. Memory size models after mapping global variable v3 to the global memory in problem 3.

Result In the given example, variable $v1$ is mapped to the local memory of PE1. Variable $v2$ and $v4$ are mapped to the local memories of $PE1$ and $PE2$. Variable $v3$ is mapped to the global memory.

The total amount of traffic on the interconnection network is:

$$\begin{aligned} traffic(total) =& Traffic(v1) + Traffic(v2) + \\ & Traffic(v3) + Traffic(v4) \\ =& 14 + 12 + 16 + 25 \\ =& 67(Kbyte) \end{aligned}$$

7.4 Design Problem 4

7.4.1 Design Assumption

From design problem 1 to design problem 3, the task *variable-memory mapping* is implemented before deriving all the architecture implementation details, such as the performance of PEs, the real time operation systems of SW PEs, the interconnection network topology, and the bus protocols. The decision made in variable-mapping is the basis for making such implementation decisions. For example, variable-memory mapping determines the amount of traffic, which influences bus protocol selection.

However, in some cases, designers want to reduce the communication time by *variable-memory re-mapping* introduced in Figure 7(c). The communication time is estimated according to the system estimation model, after all the related implementation details are known.

Problems 4 describes the variable-memory re-mapping problem. In problem 4, we use behavior model described in section 4.1 and use the target architecture model described in 4.2. Problem 4 allows three types of variables: function variable, static behavior variable, and dynamic behavior variable. Problem 4 not only allows message-passing communication mechanism, but also allows shared-memory communication mechanism. Problem 4 allows three variable-mapping mechanisms: a global variable is either mapped to each local memory of its connecting PEs, or to only one local memory of its connecting PEs, or to the global memory. We assume that the sizes of local memories of PEs are predefined. The size of global memory is unknown.

7.4.2 Problem Definition

For the problem 4, the goal is to produce a variable-memory mapping solution to minimize the communication time on the interconnection network of the system architecture. The communication time is defined as the summation of the communication time of all the PEs in the system architecture.

7.4.3 System Estimation Model

We estimate the communication time according to different variable-mapping mechanisms. In general, for each data transfer, we compute the communication time by:

$$\begin{aligned} comm\_time =& ready\_time + transfer\_time + \\ & arbitration\_time \end{aligned}$$

The $ready\_time$ refers to the maximin of ready time of sender and receiver. The $transfer\_time$ refers to the communication time over the interconnection network. The $arbitration\_time$ refers to the time required by bus arbiter to assign the bus to a bus master. In this report, we assume that the $arbitration\_time$ is 0.

Mapping to Each Local Memory If we map a variable v to each local memory of its connecting PEs, then v's read access do not produce any traffic over the interconnection network. However, when v's write access happens, the write access must be broadcasted to all of v's connecting PEs. Therefore, the v's communication time of any v's connecting PE ϖ is computed as:

$$\begin{aligned} ready\_time(v,\varpi) =& \max_{\theta \in S\_P(v)} ready\_time(v,\theta) * \\ & num(write, v); \\ transfer\_time(v,\varpi) =& transfer\_time(v,\tau) * \\ & num(write, v); \\ comm\_time(v,\varpi) =& ready\_time(v,\varpi) + \\ & transfer\_time(v,\varpi) \end{aligned}$$

where $S\_P(v)$ is the set of v's connecting PEs, τ is the selected bus, $transfer\_time(v,\tau)$ is the bus transfer time of bus τ per v's transmission, and $num(write, v)$ is the amount of v's write access.

Because each PE in $S\_P(v)$ takes $comm\_time(v,\varpi)$ for communication, the total communication time of variable v for all its connecting PEs is:

$comm\_time(v) = num(S\_P(v)) * comm(v,\varpi);$
where $num(S\_P(v))$ refers to the number of PEs in $S\_P(v)$.

Mapping to One Local Memory We map a global variable v to only one local memory of its connecting PEs, called ρ. In this case, both v's read access from the behaviors mapped to PE ρ and write access to the behaviors mapped to PE ρ don't produce traffic on the interconnection network. However, both v's read access and write access for the behaviors mapped to v' connecting PEs other than ρ produce traffic on the interconnection network. In this case, the v's communication time of any v's connecting PE ϖ other than ρ is computed as:

$$\begin{aligned} ready\_time(v,\varpi) =& (\max(ready\_time(v,\varpi), \\ & ready\_time(v,\rho)))* \\ & (num(v,read,\varpi)+ \\ & num(v,write,\varpi)) \\ transfer\_time(v,\varpi) =& transfer\_time(v,\tau)* \\ & (num(write,v,\varpi)+ \\ & num(read,v,\varpi)); \\ comm\_time(v,\varpi) =& ready\_time(v,\varpi)+ \\ & transfer\_time(v,\varpi) \end{aligned}$$

where $num(write,v,\varpi)/num(read,v,\varpi)$ refers to the amount of v's write/read access to/from PE ϖ. During each read/write access, both PE ρ and PE ϖ are involved in communication, therefore, the total communication time is computed as:

$$comm\_time(v) = \sum_{(\varpi \in S\_P(v)) \cap (\varpi \neq \rho)} comm\_time(v,\varpi) \\ *2$$

It should be noted that that mapping v to the local memory of different PEs produces different communication time.

Mapping to Global Memory If we map a variable v to a global memory ρ, then each read/write access of v produces traffic on the interconnection network. The v's communication time of any v's connecting PE ϖ is computed as

$$\begin{aligned} ready\_time(v,\varpi) =& (\max(ready\_time(v,\varpi), \\ & ready\_time(v,\rho)))* \\ & (num(v,read,\varpi)+ \\ & num(v,write,\varpi)) \\ transfer\_time(v,\varpi) =& transfer\_time(v,\tau)* \\ & (num(write,v,\varpi)+ \\ & num(read,v,\varpi)); \\ comm\_time(v,\varpi) =& ready\_time(v,\varpi)+ \\ & transfer\_time(v,\varpi) \end{aligned}$$

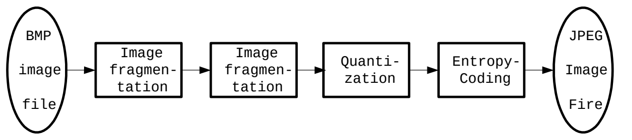


Figure 28. Block diagram of JPEG encoder

For each variable access, only one PE is involved, therefore,

$$comm\_time(v) = \sum_{\varpi \in S\_P(v)} comm\_time(v,\varpi)$$

7.4.4 Solution

We follow the design flow described in section 5 to solve this problem. The steps of global/local variable identification, stack mapping, and local variable mapping are the same as the steps in design problem 3.

Global Variable Mapping The global variable algorithm of variable-memory remapping is displayed in Figure 27.

We first order global variables in the list *Global_Var* in the decreasing order of variable's total access. Then we map one global variable at each iteration.

For each variable v, we compute the total communication time for three different types of mapping. $Total\_comm1$ refers to the communication time when mapping v to each local memory of v's connecting PEs. $Total\_comm2$ refers to the communication time when mapping v to the global memory. $Total\_comm3$ refers to the communication time when mapping v to only one local memory of v's connecting PEs. For $total\_comm1$ and $total\_comm3$, when any local memory doesn't have enough memory to store v, then the returned value equals to ∞. After communication time computation, we select the mapping mechanism that has the smallest communication time. After the mapping mechanism is selected, the corresponding memory size models are updated according.

8. Experimental Result

First of all, we implement the introduced approach by programming around 3000 lines of C++ code. We then test it on 10 random generated examples. In this section, we introduce the experimental results of two real design examples: JPEG[4] project and Vocoder[8] project.

8.1 JPEG Project

8.1.1 Introduction

JPEG is an image compression standard. It is designed for compressing either full-color or gray-scale images of natu-

```
Global_Var = SortVariable();
for v in Global_Var do
    $\omega$ = Associtate_Behavior(v);
    // Map to each local memory
    total_comm1 = ComputeTrafficMapToEachLocalMemory(v);
    // Map to global memory
    total_comm2 = ComputeTrafficMapToGlobalMemory(v);

    // Map to one local memory
    total_comm3 = $\infty$;
    mapped_pe = NULL;
    for each $\varpi \in$ S_P(v) do
        temp = ComputeTrafficMapToLocalMem($\varpi$, v);
        if (temp < total_comm3) do
            Total_comm3 = temp;
            mapped_pe = $\varpi$;
        endif
    endfor

    //Update memory size model
    switch min(total_comm1, total_comm2, total_comm3) do

    case total_comm1:
         for each $\rho \in$ S_P(v) do
              $\alpha$($\omega$, $\rho$) = $\alpha$($\omega$, p) + size(v, $\rho$) ;
              Update($\beta$, p);
              Update($\lambda$, p);
         enddo
         break;

    case total_comm2:
         $\alpha$($\omega$, Global_Mem) = $\alpha$($\omega$, Global_Mem) + size(v, Global_Mem);
         Update($\beta$,Global_Mem);
         Update($\lambda$,Global_Mem);
         break;

    case total_comm3:
         $\alpha$($\omega$, mapped_pe) = $\alpha$($\omega$, mapped_pe) + size(v, mapped_pe);
         Update($\beta$, mapped_pe);
         Update($\lambda$, mapped_pe);

    endswitch
 endfor
```

Figure 27. Algorithm 2: the global variable mapping algorithm for problem 4

without optimization	with optimization
6.466kB	4.268kB

Table 7. Required local memory size of ColdFire microprocessor in pure SW solution of JPEG project

PE	without optimization	with optimization
ColdFire	6.466kB	4.268kB
HW	0.39kB	0.39kB

Table 8. Required local memory sizes of PEs in HW-SW co-design of JPEG project

ral scenes. Figure 28 shows the block diagram of the DCT based encode for a gray scale image. It consists of four blocks: the image fragmentation block, the DCT block, the quantization block, and the entropy coding block. We model JPEG using SpecC language. The SpecC specification of JPEG contains 29 behaviors and 37 behavior variables. All the behavior variables are dynamic.

8.1.2 Pure SW solution

First, we implement the entire JPEG encoder on a Motorola ColdFire microprocessor. All the stack and behavior variables are mapped to the local memory of ColdFire processor. The estimated required memory size is displayed in Table 7.

The result for design problem 1 is displayed in column *without optimization*. In this case, we treat all the behavior variables as static variables which cannot share the same memory portion. The result of design problem 2 is displayed in column *with optmization*. In this case, we treat all the behavior variables as dynamic variables which can share the same memory portion according their lifetime. Since JPEG is data-domain application, preemptive schedule is not required. As a result, we choose the non-preemptive operation system for the ColdFire microprocessor. As shown in Table 7, we reduce the required memory size of ColdFire by 34% when we analyze the lifetime of behavior variables and allow behavior variables with un-overlapped lifetime to share the same memory portion.

8.1.3 HW-SW Codesign

Second, we implement HW-SW Codesign. The system archiecture contains two PEs: a custom hardware(*HW*) and a ColdFire microprocessor(*ColdFire*). According to [4], we map DCT behavior to *HW* and map rest of behaviors to *ColdFire*. We choose massage-passing communication mechanism. The required memory sizes for design problem 1 and design problem 2 are displayed in Table 8.

PE	lowerbound	upperbound
ColdFire	4.268kB	4.268kB
HW	0.13kB	0.39kB

Table 9. The lower-bounds / upper-bounds of local memory sizes of PEs in HW-SW co-design of JPEG project (with optimization)

In comparison to pure *ColdFire* solution described in section 8.1.2, the required memory sizes for ColdFire doesn't change. To find the reason, we analyzed the code and found that 256 Byte of memory in *HW* is for global variables and 134 Byte of memory is for the DCT's stack. $ColdFire$ reserves 256B memory for the global variable of *DCT* in both pure *ColdFire* solution and HW-SW co-design solution. Furthermore, other behaviors mapped to *ColdFire* share the same memory portion with *DCT*'s stack in both pure ColdFire solution and the HW-SW co-design solution. This proves that the result is reasonable.

We also attempt solve the design problem 3. To provide the local memory sizes of HW and $ColdFire$, we compute the lower-bound/upper-bound of local memory sizes of PEs with optimization, which is displayed in Table 9. The lowerbound and upperbound of local memory size of $ColdFire$ are the same. Therefore, the local memory size of ColdFire must be no less than 4.268kB. In this case, since all the global variables can be mapped to the local memories, the global memory is not required.

8.2 Vocoder Project

8.2.1 Introduction

The Vocoder[8] project implements the voice encoding part of the GSM standard for mobile telephony encoding standard. The block diagram of Vocoder is displayed in Figure 29. It contains 13,000 lines of code, 102 behaviors, and 156 dynamic behavior variables.

8.2.2 Pure SW Solution

First, we implement the entire Vocoder encoder on the Motorola DSP56600 microprocessor. All the stack and behavior variables are mapped to the local memory of DSP56600. The estimated required memory is displayed in Table 10.

Similar to JPEG design, the result for design problem 1 is displayed in column *without optimization*. The result of design problem 2 is displayed in column *with optmization*. Since Vocoder is data-domain application, preemptive schedule is not required. As a result, we choose the non-preemptive schedule mechanism for DSP56600. As shown

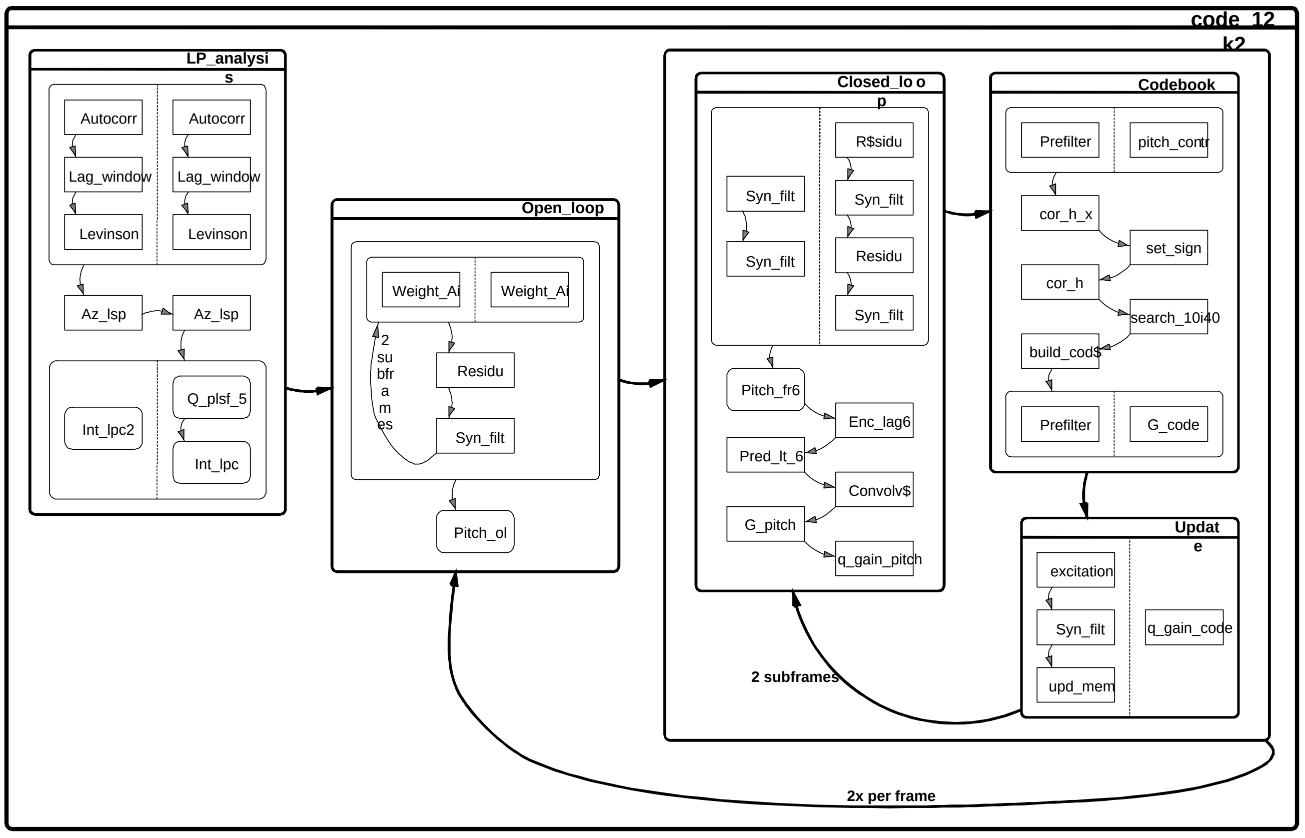


Figure 29. Block diagram of encoding part of vocoder

without optimization	with optimization
5.369kB	3.872kB

Table 10. Required memory size of Motorola DSP56600 microprocessor in pure SW solution of Vocoder project

PE	without optimization	with optimization
DSP56600	3.454kB	2.217kB
HW	2.308kB	2.308kB

Table 11. Required memory sizes of PEs in HW-SW co-design of Vocoder project

in Table 10, we reduce the required memory size of ColdFire by 28% when we analyze the lifetime of behavior variables and allow behavior variables with un-overlapped lifetime to share the same memory portion.

8.2.3 HW-SW Codesign

Mapping to Local Memory According to design project [8], a Motorola DSP 56600 microprocessor(*DSP*) and a custom hardware(*HW*) are selected to assemble the system architecture. The timing-consuming function *Codebook* is mapped to *HW* while rest behaviors are mapped to the *DSP*. The massage passing mechanism is chosen. We compute the required memory sizes for *DSP* and *HW*, which are displayed in Table 11.

We also compute the lower-bounds/upper-bounds of local memory sizes of PEs with optimization, which is displayed in Table 12. The difference between the lowerbound and the upperbound of *DSP* is 128B. The difference between the lowerbound and the upperbound of *HW* is 283B. The difference of *DSP* is smaller than the difference of *HW*. This is because that in *DSP*, the global memories shares the same memory portion with some local memories of other behaviors .

Since the total size of the global variable is 283B, we don't allocate a global memory to explore the shared-memory mechanism. As a result, we select 4k local memory for both *DSP* and *HW*.

PE	lowerbound	upperbound
DSP 56600	2.089kB	2.217kB
HW	2.015kB	2.308kB

Table 12. The lower-bounds / upper-bounds of local memory sizes of PEs in HW-SW co-design of Vocoder project (with optimization)

8.2.4 SoC Design

In the example, the selected system architecture contains four PEs. $DSP1$ and $DSP2$ are the Motorola DSP56600 microprocessors. $HW1$ and $HW2$ are the two custom hardware. One global memory Mem is also instantiated in the system architecture. We choose non-preemptive scheduler for $DSP1$ and $DSP2$. The behavior-PE mapping decision is described in Table 13.

Table 14 displays the required global memory size and generated traffic on the interconnection network for different sets of given local memory sizes. The local memory sizes in case 1(raw1) equal to the upperbound of sizes of local memories. Case 1 doesn't require mapping any variable to the global memory. Therefore, the global memory can be removed from the system architecture. The local memory sizes in case 6 equal to the lowerbounds of sizes of local memories. The upperbounds/lowerbounds of the sizes of local memories are computed according to the solution of design problem 2. In addition to case 1 and case 6, we also design four other cases, from case 2 to case 5. The difference between the local memory size of same PE in neighbor cases are the same.

Column *Decreased local mem* denotes the difference between the summation of the local memory size of the selected case and the summation of the local memory sizes of case 6. Column *Global mem* denotes the required global memory size. Column *Total added mem* equals to the sum of the values in column *Decreased local mem* and in column *Global mem*. Column *Traffic* denotes the amount of generated traffic on the interconnection network. The value in column *Global mem* and *Traffic* are computed according to the solution of design problem 3.

Figure 30 displays the values in column *Global mem*, *Total added mem*, *Decreased local mem*, and *Traffic*. As the size of local memory decreases proportionately denoted by *Decreased local memory*, the size of global memory increases unproportionately. This is because a variable is overlapped with different variables in local memories of PEs and in the global memory. The total memory size denoted by *Total added mem* is decreasing when more variables are mapped to the global memory rather than to the local memory. This is because when we map a global variable to the local memory, the global variable has a copy at each of its connecting PEs. The slope of generated traffic represented by *Traffic* increases with the decreasing of the local memory size. This proves that our approach of mapping the heavy read traffic variable to local memory is efficient.

PE	DSP1	DSP2	HW1	HW2
Bhvr	coder, LP_analys	Open_loop	Closed_loop, Update	Codebook

Table 13. Behavior-PE mapping Solution in SoC design of Vocoder project

	Input local memory sizes				Output global memory size & generated traffic			
	PE1 (Byte)	PE2 (Byte)	PE3 (Byte)	PE4 (Byte)	Added local mem(Byte)	Global mem(Byte)	Total added mem(Byte)	Traffic (KByte)
Case 1	1870	720	784	2308	810	0	810	145.7
Case 2	1858	708	699	2249	648	164	812	181.6
Case 3	1846	696	614	2190	486	214	700	220.7
Case 4	1834	684	529	2131	324	324	648	332.6
Case 5	1822	672	444	2072	162	389	551	432.7
Case 6	1810	662	361	2015	0	485	485	2016.0

Table 14. Table of required global memory sizes and generated traffic for different sets of given local memory sizes in SoC design of Vocoder project

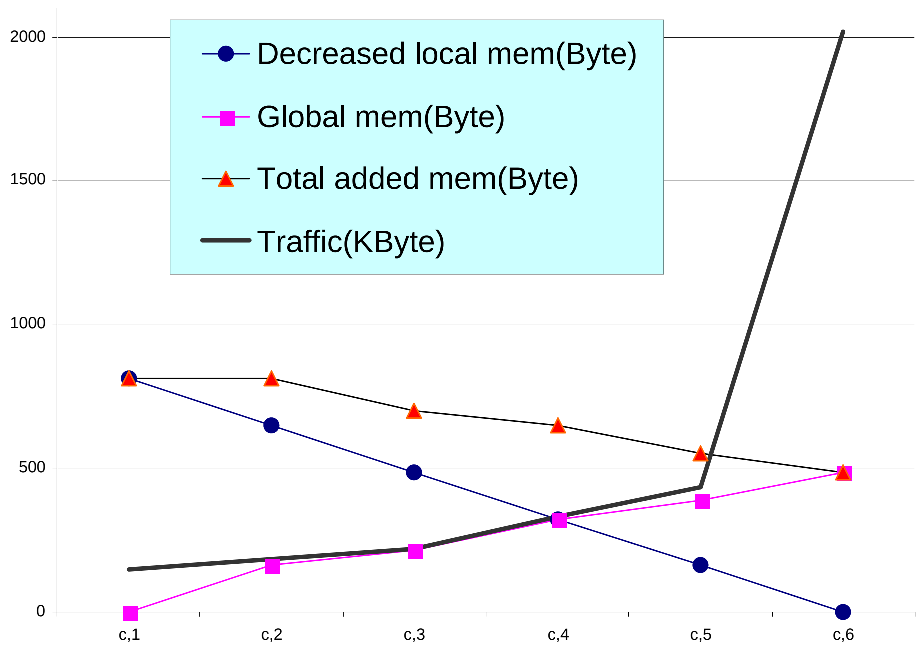


Figure 30. The chart of the require global memory size, decreased local memory size, and generated traffic for SoC design of Vocoder project

9. Conclusion

In this report, we introduce a variable-mapping algorithm of system level design. To our knowledge, this is the first variable-mapping algorithm to support hierarchical behavior specification. It is also the first to evaluate the tradeoff between message-passing communication mechanism and shared-memory mechanism, and the first to take the operation system of SW PE into account. Furthermore, we present a novel memory size model to analyze the variable lifetime in the task level. This model not only computes the required minimal memory sizes but also determines whether a memory has room for the next mapped variable. Finally, our algorithm is independent of many implementation details including bus topology of system architecture, selected bus protocols, and PE's performance. This attribute allows to move the variable-memory mapping into the earliest design stage. We apply the algorithm on the JPEG and Vocoder project. The computed memory sizes using our algorithm are 34% and 28% smaller than the sizes without using our algorithm for JPEG and Vocoder respectively.

References

[1] SystemC, OSCI[online]. Available: http://www.systemc.org/.

[2] VCC[online]. Available: http://www.cadence.com/products/vcc.html.

[3] L. Cai and D. Gajski. Introduction of Design-Oriented Profiler of SpecC Language. Technical Report ICS-TR-00-47, University of California, Irvine, June 2001.

[4] L. Cai, J. Peng, and D. Gajski. Design of a JPEG Encoding System. Technical Report ICS-TR-99-54, University of California, Irvine, Nov 1999.

[5] D. Gajski, N. Dutt, S. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[6] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[7] Lovic Gauthier, Sunjoo Yoo, and Ahmed Amine Jerraya. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, November 2001.

[8] A. Gerstlauer, S. Zhao, and D. Gajski. Design of a GSM Vocoder using SpeccC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, Feb 1999.

[9] Samy Meftali, Ferid Gharsalli, Frederic Rousseau, and Ahmed A. Jerraya. An optimal memory allocation for apllocation-specific multiprocessor system-on-chip. In *Proceedings of the International Symposium on System Synthesis*, 2001.

[10] P.R. Panda and A. Nicolau N. Dutt. *Memory Issues in Embedded System-on-chip: Optimization and exploration*. Kluwer Academic Publishers, 1999.

[11] S. Prakash and A.C. Parker. Synthesis of application-specific multiprocessor systems including memory components. *IEEE Transactions on VLSI Signal Processing*, 1994.

[12] R. Szymanek and K. Kuchcinski. Design space exploration in system level synthesis under memory constraints. In *Euromicro 25*, September 1999.

[13] Y.Li and W. Wolf. Hardware/software co-synthesis with memory hierarchies. *IEEE transaction on computer-aided design of integrated circuit and Systems*, October 1999.