An Algebraic Hardware/Software Partitioning Algorithm

QIN Shengchao (秦胜潮)[1], HE Jifeng (何积丰)[2], QIU Zongyan (裘宗燕)[1]
and ZHANG Naixiao (张乃孝)[1]

[1] *Department of Informatics, School of Mathematical Sciences, Peking University
Beijing 100871, P.R. China*

[2] *UNU/IIST, The International Institute for Software Technology
The United Nations University, Macau, P.R. China*

E-mail: qinshc@pubms.pku.edu.cn; jifeng@iist.unu.edu; {zyqiu,naixiao}@pku.edu.cn

Received March 5, 2001; revised June 13, 2001.

Abstract Hardware and software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of the co-design process is to decompose a program into hardware and software. This paper proposes an algebraic partitioning algorithm whose correctness is verified in program algebra. The authors introduce a program analysis phase before program partitioning and develop a collection of syntax-based splitting rules. The former provides the information for moving operations from software to hardware and reducing the interaction between components, and the latter supports a compositional approach to program partitioning.

Keywords hardware/software co-design, hardware/software partition, program algebra

1 Introduction

The design of a complex computerized product like a nuclear reactor control system is ideally decomposed into a progression of the related phases. It starts with an investigation of the properties and behaviors of the process evolving within its environment, and an analysis of the requirement for its safety performance. From these is derived a specification of the electronic or program-centered components of the system. The project then may go through a series of design phases, ending with a program expressed in a high-level language. After being translated into the machine code of a chosen computer, it is executed at a high speed by electronic circuitry. In order to achieve the time performance required by the customer, additional application-specific hardware devices may be needed to embed the computer into the system which it controls.

Industry interest in the formal verification of embedded systems is gaining ground since an error in a widely used hardware device can have significant repercussions on the stock value of the company concerned. In principle, the proof of correctness of a digital device can always be achieved by making a comparison of the behavioral description of the circuit with its specification. But for a large system this would be impossibly laborious. What we need is a useful collection of proven equations and other theorems, which can be used to calculate, manipulate and transform the specification formulae to the product.

Hardware/software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of the co-design process is to partition the specification (the source program) into hardware and software. This paper proposes a partitioning method whose correctness is verified using the algebraic laws developed for the high level programming language. To meet performance goals and reduce the communication between components, our approach combines the program analysis technique with the syntax-based splitting rules to move heavy-weight

The work is partially supported by the National Natural Science Foundation of China under grant Nos.69873003 and 69983001.

operations from software to hardware. The allocation of variables is also based on the data flow analysis of the source program. One of the advantages of our method is the integration of the splitting phase with the joining phase of the partitioning process (in [1]). It optimizes the underlying target architecture, and facilitates the reuse of hardware devices.

The algebraic approach advocated in this paper to verify the correctness of the partitioning process has been successfully employed in the **ProCoS** project on "Provably Correct Systems"[2,3]. Sampaio showed how to reduce the compiler design task to one of program transformation by program algebra[4]. Ian Page *et al.* made rapid advance in the development of hardware compilation techniques using an Occam-like language targeted towards Field Programmable Gate Arrays[5], and He Jifeng *et al.* provided a formal verification of the hardware compilation scheme within the algebra of Occam programs[6].

Recently, some researchers have suggested the use of formal methods for the partitioning process[1,7,8] Balboni *et al.* adopted Occam as an internal model for system exploration and partitioning strategies. Cheung pursued the structural transformation and verification within the functional programming framework. However, neither has provided a formal proof for the correctness of the partitioning process. In [1], Silva *et al.* provided a formal strategy for carrying out the splitting phase automatically, and presented an algebraic proof for its correctness. However, the splitting phase delivers a large number of simple processes, and leaves the hard task of clustering these processes into hardware and software components to the clustering phase and the joining phase. Furthermore, additional channels and local variables introduced in the splitting phase to accommodate a huge number of parallel processes actually increase the data flow between hardware and software components.

The remainder of this paper is organized as follows. Section 2 describes our partitioning strategy. Section 3 introduces the programming language we adopt and explores its algebraic laws. Section 4 poses the static analysis that we perform on the source program. Section 5 investigates the underlying target architecture of hardware/software components. Section 6 provides the syntax-based hardware/software splitting rules in both bottom-up and top-down styles.

2 Partitioning Strategy

Our partitioning strategy is described as follows. Suppose a source program has been coded by the programmer in the source programming language based on the customer's requirements. A static analysis[9] is performed on that program to obtain useful statistical data, such as quantitative information concerning the occurrences of expressions and variables, distributive information with respect to those variables occurring in expressions.

Based on the analysis, the programmer marks those parts of the program that are worthy to be implemented by hardware and leaves others to software, and as well divides the set of the variables employed by the program to two disjoint parts. Program marking and variable division are conducted by the following general guidelines, where the word "busy" is explained in Section 4.

- Generally, busy expressions should be marked and implemented by hardware, to gain a high performance.
- Analogously, busy variables should be allocated to hardware, to make high-speed access available, whereas the remaining variables and large scale data structures, such as arrays, should be left to software, to achieve lower cost.
- The number of interactions between software and hardware should be minimized since they incur high costs.
- In addition, the customer's demands concerning performance and cost should also be taken into account.

We take such a marked program as the input of our hardware/software splitting process and generate as output a program comprising two concurrent processes representing software and hardware components respectively. We investigate a collection of syntax-based rules to perform the splitting task. All those rules are well-defined and their correctness is verified in the algebra of programs.

They play a vital role in our partitioning algorithm and can be used to undertake the splitting task automatically.

3 Preliminaries

The language we select to perform hardware/software partitioning is a subset of Occam which was designed for constructing communicating systems. It consists of the following two parts.

1) Sequential Process:

$$\begin{aligned} S ::= {} & PC \text{ (primitive command)} \mid S; S \text{ (sequential composition)} \\ & \mid S \lhd b \rhd S \text{ (conditional)} \mid S \sqcap S \text{ (non-deterministic choice)} \\ & \mid b * S \text{ (iteration)} \mid (gS) \, [\!] \, (gS) \text{ (guarded choice)} \end{aligned}$$

where $PC ::= (x := e) \mid skip \mid \perp \mid c!e \mid c?x$, and g is *skip* or a communication event $c!e$ or $d?x$.

2) Parallel Program:

$$P ::= S \mid P \parallel P \mid \mathbf{var}\ x \bullet P$$

In the later discussion, we adopt $Var(P)$ and $Chan(P)$ to denote the sets of variables and channels employed by P.

As a subset of Occam, the language enjoys a rich set of algebraic laws presented in [10–14]. Here we only explore those algebraic laws which will be employed within the proofs in the following sections.

Successive assignments to the same variable can be combined to one assignment.

L1 $x := e; x := f = x := f[e/x]$ (*comb ass*)

Sequential composition is associative, and has left zero $\perp$ and unit *skip*. It distributes backward over internal and external choices and conditional.

L2 $(P; Q); R = P; (Q; R)$ (*seq assoc*)

L3 $\perp; P = \perp$ ($;-zero \perp$)

L4 $skip; P = P; skip = P$ ($; -unit\ skip$)

L5 $(P \sqcap Q); R = (P; R) \sqcap (Q; R)$ ($; - \sqcap\ distr$)

L6 $(g\ P) \, [\!] \, (h\ Q); R = (g(P; R)) \, [\!] \, (h(Q; R))$ ($; - [\!]\ distr$)

L7 $(P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R)$ ($; -cond\ distr$)

Assignment distributes forward over conditional.

L8 $v := e; (P \lhd b(v) \rhd Q) = (v := e; P) \lhd b(e) \rhd (v := e; Q)$ (*ass − cond distr*)

The input and output events can be renamed as follows.

L9 $c?x = \mathbf{var}\ lx \bullet (c?lx; x := lx)$ (*input renaming*)

L10 $c!e = \mathbf{var}\ lx \bullet (lx := e; c!lx)$ (*output eval*)

Iteration is subject to the fixed point theorem.

L11 $b * P = (P; b * P) \lhd b \rhd skip$ (*fixed point*)

Parallel operator is symmetric and associative, and has $\perp$ as zero.

L12 $P \parallel Q = Q \parallel P$ ($\parallel$ *comm*)

L13 $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$ ($\parallel$ *assoc*)

L14 $\perp \parallel P = \perp$ ($\parallel -zero \perp$)

Parallel operator also distributes over conditional. It is disjunctive.

L15 $(P \lhd b \rhd Q) \parallel R = (P \parallel R) \lhd b \rhd (Q \parallel R)$, provided $Var(b) \cap Var(R) = \emptyset$. ($\parallel -cond\ distr$)

L16 $(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$ ($\parallel$ *disj*)

Local variable declaration enjoys the following laws.

L17 $\mathbf{var}\ x \bullet (x := e) = skip$, provided x does not occur in e. (*dec skip*)

L18 $\mathbf{var}\ x \bullet (P \lhd b \rhd Q) = (\mathbf{var}\ x \bullet P) \lhd b \rhd (\mathbf{var}\ x \bullet Q)$, provided x is not free in b. (*dec−cond distr*)

L19 $\mathbf{var}\ x \bullet (P; Q) = P; \mathbf{var}\ x \bullet Q$, provided x is not free in P. (*dec elim1*)

L20 $\mathbf{var}\ x \bullet (P; Q) = \mathbf{var}\ x \bullet P; Q$, provided x is not free in Q. (*dec elim2*)

The following expansion law deals with assignment expansion.

L21 $(x := e; S) \parallel T = x := e; (S \parallel T)$ (*ass exp*)

The following law is one of the general expansion laws of Occam[10], which deals with the case where two processes in parallel are guarded choice constructs.

L22 Let $P = []_{i=1}^{n}(g_i\, P_i)$, $Q = []_{j=1}^{m}(h_j\, Q_j)$, where each g_i or h_j has one of the forms $c!e$, $c?x$ or *skip*, then $P \parallel Q = []_{r=1}^{N}(k_r\, R_r)$, where the pairs $\langle k_r, R_r\rangle$ are precisely all possibilities from the following:

(i) $R_r = P_i \parallel Q$ and ($k_r = g_i = skip$ or $k_r = g_i = c!e$ or $k_r = g_i = c?x$), where $c \notin Chan(Q)$,

(ii) $R_r = P \parallel Q_j$ and ($k_r = h_j = skip$ or $k_r = h_j = c!e$ or $k_r = h_j = c?x$), where $c \notin Chan(P)$,

(iii) $R_r = x := e; (P_i \parallel Q_j)$ and

$k_r = skip$ and ($g_i = c!e$ and $h_j = c?x$ or $g_i = c?x$ and $h_j = c!e$). (*gc exp*)

We explore a corollary from L22, which will be very useful in later sections.

Corollary 3.1.

($C1$) $(c!e; P) \parallel (c?x; Q) = x := e; (P \parallel Q)$. (*int comm*)

($C2$) *Let* $P = (c!e; P_1)$, $Q = (Q_1; Q_2)$, *where* $c \in Chan(Q)$, *but no channel in* $Chan(P) \cap Chan(Q)$ *occurs in* Q_1, *then* $P \parallel Q = Q_1; (P \parallel Q_2)$. ($\parallel$ *elim*)

($C3$) *Let* $P = (S_1; c?x; S_2)$, *and* $Q = (T_1; c!e; T_2)$, *where neither* S_1 *nor* T_1 *mentions channels in* $Chan(P) \cap Chan(Q)$, *then* $P \parallel Q = (S_1 \parallel T_1); ((c?x; S_2) \parallel (c!e; T_2))$. (*sync comm*)

The proof of this corollary can be found in our previous papers[15,16].

We investigate two derived algebraic laws from those basic ones, which will be helpful to conducting hardware/software partitioning.

The test of conditional should be evaluated first.

DL1 $P \lhd b \rhd Q = \mathbf{var}\ lb \bullet (lb := b; P \lhd lb \rhd Q)$ (*cond eval*)

The proof is presented in [16].

The condition of iteration is evaluated at the beginning of every loop.

DL2 $b * P = \mathbf{var}\ lb \bullet (lb := b; lb * (P; lb := b))$ (*iter eval*)

The following lemma from [17] will support the proof of DL2.

Lemma 3.2. *Let F, G and H be functions, and let F be a lower adjoint. If* $F(G) = H(F)$, *then* $\mu H = F(\mu G)$.

Proof of DL2. Let $F(X) =_{df} \mathbf{var}\ lb \bullet (lb := b; X)$, $G(X) =_{df} (P; lb := b; X) \lhd lb \rhd skip$, $H(X) =_{df} (P; X) \lhd b \rhd skip$. It is direct from Lemma 3.2 and

$$\begin{aligned}
F(G(X)) &= \mathbf{var}\ lb \bullet (lb := b; ((P; lb := b; X) \lhd lb \rhd skip)) && \{(ass\text{-}cond\ distr)\}\\
&= \mathbf{var}\ lb \bullet ((lb := b; P; lb := b; X) \lhd b \rhd (lb := b)) && \{(dec\text{-}cond\ distr)\}\\
&= (\mathbf{var}\ lb \bullet (lb := b; P; lb := b; X)) \lhd b \rhd (\mathbf{var}\ lb \bullet (lb := b)) && \{(dec\ skip)\}\\
&= (\mathbf{var}\ lb \bullet (lb := b; P; lb := b; X)) \lhd b \rhd skip && \{lb \text{ is not free in } P\}\\
&= (\mathbf{var}\ lb \bullet (P; lb := b; lb := b; X)) \lhd b \rhd skip && \{(dec\ elim1), (comb\ ass)\}\\
&= (P; \mathbf{var}\ lb \bullet (lb := b; X)) \lhd b \rhd skip && \{def \text{ of } F, H\}\\
&= H(F(X)) && \square
\end{aligned}$$

We introduce an ordering relation between two programs before further discussion.

Definition 3.3 (Refinement). *Given programs* P, Q, *we say* Q *is a refinement of* P, *denoted as* $P \sqsubseteq Q$, *if* $P \sqcap Q = P$ *is algebraically provable.*

4 The Static Analysis

This section illustrates the simple static analysis performed on the source program, which provides primitive but useful information to the programmer to assist the appropriate hardware/software marking and variable division of the source program, aiming to gain higher performance and achieve lower cost.

First, we introduce a function *depth* for expressions to specify their complexity.

Definition 4.1. *depth*: *Expr* $\to \mathbb{N}$ *is inductively defined on the structure of expressions*:

$depth(v) =_{df} 1$, *for any variable* v,

$depth(c) =_{df} 0$, *for any constant* c, *and*

$depth(\mathbf{op}(e_1, \ldots, e_n)) =_{df} \sum_{i=1}^{n} depth(e_i) + depth(\mathbf{op})$, *where* op *is any operator used to construct expressions in the language we adopt, and* $depth(\text{op})$ *is defined by the programmer in accordance with the complexity of* op.

An expression is regarded as a busy expression if it occurs often in the program or owns an intricate structure. We will generate a table to record the occurrence frequency of expressions.

$$\Phi(S) = \{(e, n(e)) \mid e \in Expr(S)\}$$

where S is a program, $n(e)$ represents the number of occurrences of the non-trivial expression e in S, i.e., e is neither a single variable nor a constant.

The busy variable analysis produces a table $\Psi(S) = \{(v, eset(v)) \mid v \in Var(S)\}$ for the program S, where $eset(v)$ is the set of expressions which contain the variable v.

Here we introduce two merge operators applied to tables of the two kinds, respectively.

Definition 4.2. *Let* $\Phi_i = \{(e, n) \mid e \in E_i\}$, *and* $\Psi_i = \{(v, eset) \mid v \in V_i\}$, *where* E_i *and* V_i *are the set of expressions and the set of variables of interest, for* $i = 1, 2$. *We define*

$$\Phi_1 +_e \Phi_2 =_{df} \{(e, n) \mid (e, n) \in \Phi_1 \cup \Phi_2 \wedge e \notin E_1 \cap E_2\} \cup \{(e, n_1 + n_2) \mid (e, n_i) \in \Phi_i, i = 1, 2\}$$

and

$$\begin{aligned} \Psi_1 +_v \Psi_2 =_{df} &\{(v, eset) \mid (v, eset) \in \Psi_1 \cup \Psi_2 \wedge v \notin V_1 \cap V_2\} \cup \\ &\{(v, eset_1 \cup eset_2) \mid (v, eset_i) \in \Psi_i, i = 1, 2\} \end{aligned} \qquad \square$$

For a program S, the table concerning the occurrence frequency of its expressions T_e and the table with respect to that of its variables T_v can be generated using the following productive rules. The predicate *Scan* used in the following is defined by

$$Scan(S, T_e, T_v) =_{df} T_e = \Phi(S) \wedge T_v = \Psi(S).$$

$$Scan(u := e(v),\ \{(e(v), 1)\},\ \{(v, \{e(v)\})\})$$
$$Scan(c\,!\,e(v),\ \{(e(v), 1)\},\ \{(v, \{e(v)\})\})$$
$$Scan(c\,?\,x,\ \emptyset,\ \emptyset)$$

$$\frac{Scan(S_i, \Phi_i, \Psi_i), \quad i = 1, 2}{Scan(S_1; S_2, \Phi_1 +_e \Phi_2, \Psi_1 +_v \Psi_2)}$$

$$\frac{\begin{array}{c} Scan(S_i, \Phi_i, \Psi_i), \quad i = 1, 2 \\ \Phi = \Phi_1 +_e \Phi_2 +_e \{(eb, 1)\} \\ \Psi = \Psi_1 +_v \Psi_2 +_v \{(v, \{eb\}) \mid v \in Var(eb)\} \end{array}}{Scan(S_1 \lhd eb \rhd S_2, \Phi, \Psi)}$$

$$\frac{\begin{array}{c} Scan(S, \Phi, \Psi) \\ \Phi' = \{(e, \infty) \mid \exists n \bullet (e, n) \in \Phi\} +_e \{(eb, \infty)\} \\ \Psi' = \Psi +_v \{(v, \{eb\}) \mid v \in Var(eb)\} \end{array}}{Scan(eb * S, \Phi', \Psi')}$$

$$\frac{\begin{array}{c} Scan(S_i, \Phi_i, \Psi_i), \quad i = 1, 2 \\ Scan(g_i, \Phi_{g_i}, \Psi_{g_i}), \quad i = 1, 2 \\ \Phi = \Phi_1 +_e \Phi_2 +_e \Phi_{g_1} +_e \Phi_{g_2} \\ \Psi = \Psi_1 +_v \Psi_2 +_v \Psi_{g_1} +_v \Psi_{g_2} \end{array}}{Scan((g_1 S_1) [\!] (g_2 S_2), \Phi, \Psi)}$$

Example 4.3. Consider the following program S.

$$\begin{aligned} S :\ & ic_1\,?\,u; ic_2\,?\,v; w := 2 \times (u + v); \\ & oc\,!\,w; ic_3\,?\,z; (z := u \times v) \lhd (z = u) \rhd skip; \\ & (u < (v \times v)) * (u := u + w; \\ & w := (u - v) \times (u + v); oc\,!\,w) \end{aligned}$$

where $Var(S) = \{w, z, u, v\}$, $Chan(S) = \{ic_1, ic_2, ic_3, oc\}$.

Provided $depth(+) = depth(-) = depth(=) = depth(<) = 1$, $depth(\times) = 2$, then the results of the analysis are listed in the following two tables.

Based on the first table, one can figure out those busy expressions. Afterward the set of variables is divided into two sets, which will be allocated respectively to hardware and software, according to the criterion that a variable should be allocated to hardware if it occurs in busy expressions more often than it does in non-busy ones.

e	$n(e)$	$depth(e)$
$2 \times (u + v)$	1	5
$z = u$	1	3
$u \times v$	1	4
$u < (v \times v)$	∞	6
$u + w$	∞	3
$(u - v) \times (u + v)$	∞	8

var	$eset$
w	$\{u + w\}$
z	$\{z = u\}$
u	$\left\{ \begin{array}{c} 2 \times (u + v), z = u, u < (v \times v), \\ u \times v, u + w, (u - v) \times (u + v) \end{array} \right\}$
v	$\left\{ \begin{array}{c} 2 \times (u + v), u < (v \times v), \\ u \times v, (u - v) \times (u + v) \end{array} \right\}$

The simple analysis described above provides helpful information to the programmer, whereas this is not a must one. It is possible to impose other reasonable static analyses on the source program, such as procedure/function analysis which yields a hardware/software marking with a large granularity.

5 The Hardware/Software Target Architecture

This section describes the target architecture of our partitioning approach by confining hardware and software components to specially chosen forms. To synchronize their activities, we introduce a simple handshaking protocol to streamline communications between them.

Inspired by the definition of the simple two-phase handshaking protocol in CSP[12,18], we construct the handshaking protocol in the algebra of Occam.

Suppose $B = \{r_j, a_j \mid j \in I\}$ is a set of pairs of channels, we define the handshaking protocol as $HP(B) =_{df}$ **var** $lv \bullet \mu X \bullet ([]_{j\in I}(r_j\,?\,lv;\ a_j\,!\,lv; X)[]skip)$.

We say a sequential process S, where $Chan(S) \supseteq B$, satisfies the handshaking protocol $HP(B)$, if $(S;R) \parallel HP(B) = (S \parallel HP(B));\ (R \parallel HP(B))$ for any sequential process R employing the same set of variables and channels as S.

Associated with B, we define a set of communicating processes $CP(B)$, which is the minimal set generated by the following rules.

(1) A communicating process C does not use any channel in B, but $Chan(C) \supseteq B$.

(2) $r_j\,!\,e;\ C;\ a_j\,?\,x$, where C is a member of $CP(B)$ not interacting via channels in B.

(3) $C_1; C_2$, or $C_1 \sqcap C_2$, or $C_1 \triangleleft b \triangleright C_2$, or $(g_1\ C_1) [] (g_2\ C_2)$, where both g_i and C_i lie in $CP(B)$, for $i = 1, 2$.

(4) $b * C$, where C is a member of $CP(B)$.

By a simple structural induction on $CP(B)$, it is straightforward to know processes from $CP(B)$ satisfy the handshaking protocol.

To simplify the interface design, we confine the interactions between hardware and software components to the communications along the channels from set B. Our partitioning rules will select the software components from set $CP(B)$, and organize the hardware component in the form of

$$D = \mu X \bullet ([]_{j\in I}(r_j?x_j; M_j; a_j!y_j; X)[]skip)$$

where none of M_j mentions channels in B. The communicating process D represents a digital device which offers a set of services to its environment, each of which responds to a request from its environment on an input channel r_j by running the corresponding program M_j and delivering the result to the output channel a_j afterwards.

We denote as $H(B)$ the set of those processes which own the same form as D.

Theorem 5.1. *$(C_1; C_2) \parallel D = (C_1 \parallel D);\ (C_2 \parallel D)$, for any C_1, C_2 in $CP(B)$.*

Proof. By structural induction on C_1.

(1) No channels in B appear in C_1.

$$\begin{aligned} & LHS && \{(\parallel\ elim)\} \\ =\ & C_1;\ (C_2 \parallel D) && \{(\parallel\ elim)\} \\ =\ & RHS \end{aligned}$$

(2) $C_1 = r_j\,!\,e;\ C; a_j\,?\,x$, for some $r_j, a_j \in B$, $C \in CP(B)$, and no channel in B occurs in C.

$$\begin{aligned} & LHS && \{(int\ nomm)\} \\ =\ & x_j := e; ((C; a_j\,?\,x; C_2) \parallel (M_j; a_j\,!\,y_j; D)) && \{(sync\ comm)\} \\ =\ & x_j := e; (C \parallel M_j); ((a_j\,?\,x; C_2) \parallel (a_j\,!\,y_j; D)) && \{(int\ comm)\} \\ =\ & x_j := e;\ (C \parallel M_j);\ x := y_j;\ (C_2 \parallel D) && \{(int\ comm),\ (sync\ comm)\} \\ =\ & RHS \end{aligned}$$

(3) $C_1 = C_{01};\ C_{02}$, where $C_{01}, C_{02} \in CP(B)$.

From the definition of $CP(B)$, we know $C_{02};\ C_2 \in CP(B)$. Then

$$\begin{aligned} & LHS && \{hypothesis\} \\ =\ & (C_{01} \parallel D);\ ((C_{02}; C_2) \parallel D) && \{hypothesis\} \end{aligned}$$

$= (C_{01} \parallel D); (C_{02} \parallel D); (C_2 \parallel D)$ $\{hypothesis\}$

$= RHS$

(4) $C_1 = C_{01} \lhd b \rhd C_{02}$, or $C_1 = C_{01} \sqcap C_{02}$, or $C_1 = (g_1 C_{01})[\!](g_2 C_{02})$, where $C_{01}, C_{02} \in CP(B)$. We only demonstrate the first case here, others are similar[16].

LHS $\{(;-cond\ distr)\}$

$= ((C_{01}; C_2) \lhd b \rhd (C_{02}; C_2)) \parallel D$ $\{(\parallel -cond\ distr)\}$

$= ((C_{01}; C_2) \parallel D) \lhd b \rhd ((C_{02}; C_2) \parallel D)$ $\{hypothesis\}$

$= ((C_{01} \parallel D); (C_2 \parallel D)) \lhd b \rhd ((C_{02} \parallel D); (C_2 \parallel D))$ $\{(;-cond\ distr)\}$

$= ((C_{01} \parallel D) \lhd b \rhd (C_{02} \parallel D)); (C_2 \parallel D)$ $\{(\parallel -cond\ distr)\}$

$= RHS$

(5) $C_1 = b * C_0$, where $C_0 \in CP(B)$.

Let $F(X) = (C_0; X) \lhd b \rhd skip$, and define $\{F^n(\perp),\ n \geq 0\}$ as $F^0(\perp) =_{df} \perp$, and $F^{n+1}(\perp) =_{df} F(F^n(\perp))$, for $n \geq 0$. Then $C_1 = \mu X \bullet F(X) = \bigsqcup_{n\geq 0} F^n(\perp)$, and $F^n(\perp) \in CP(B)$, for $n \geq 0$.

LHS

$= ((\bigsqcup_{n\geq 0} F^n(\perp)); C_2) \parallel D$ $\{both \parallel and; are\ continuous\}$

$= \bigsqcup_{n\geq 0}((F^n(\perp); C_2) \parallel D)$ $\{hypothesis\}$

$= \bigsqcup_{n\geq 0}((F^n(\perp) \parallel D); (C_2 \parallel D))$ $\{both \parallel and; are\ continuous\}$

$= RHS$ □

This theorem is the theoretical foundation of the forthcoming partitioning rules. It as well indicates that the interactions between hardware and software components are well-behaved. The following corollary is straightforward from the theorem and also helpful in later discussions.

Corollary 5.2. *If $C \in CP(B)$, then $(b * C) \parallel D = b * (C \parallel D)$.*

The proof is presented in [16].

6 Syntax-Based Splitting Rules

This section is devoted to the design of program splitting rules. First we show how the static analysis affects the partition of primitive commands into hardware and software components. Secondly we demonstrate how to construct hardware and software parts of a construct from these of its constituents. We establish the correctness of those rules by using the algebraic laws given in Section 3.

We introduce a predicate *Split*, which will be used to formalize the splitting rules.

Definition 6.1 (Split). *Let $B = \{r_j, a_j \mid j \in I\}$. Given a sequential process S, its hardware/software partition (C, D) is specified by the following predicate:*

$$\begin{aligned} Split_B(S, C, D) =_{df} S \sqsubseteq (C \parallel D) \wedge C \in CP(B) \wedge D \in H(B) \wedge \\ Var(C) \cap Var(D) = \emptyset \wedge Chan(C) \cap Chan(D) = B \wedge \\ InputChan(C) \cap InputChan(D) = \emptyset \wedge \\ OutputChan(C) \cap OutputChan(D) = \emptyset \end{aligned}$$

We design rules in two different approaches, the *bottom-up* approach and the *top-down* one, to undertake the splitting task. The designer can select either of them in accordance with the facility.

6.1 The Bottom-Up Splitting Approach

The *bottom-up* approach builds the hardware component from a program directly from the static analysis in one step, i.e., the hardware device is to provide all the services frequently used by the program. However, it constructs the software component from its constituents using the following rules.

Bottom-Up Rule for Sequential Composition

$$\frac{Split_B(S_i, C_i, D), \quad i = 1, 2 \qquad Var(S_1) = Var(S_2), \quad Chan(C_1) = Chan(C_2)}{Split_B(S_1; S_2, C_1; C_2, D)}$$

Proof. $S_1; S_2$ $\{;\ is\ monotonic\}$

$\sqsubseteq (C_1 \parallel D); (C_2 \parallel D)$ $\{Th.\ 5.1\}$

$= (C_1; C_2) \parallel D$ $\square$

Bottom-Up Rule for Conditional

$$\frac{\begin{array}{c} Split_B(S_i, C_i, D), \quad i = 1,2 \\ Var(S_1) = Var(S_2), \quad Chan(C_1) = Chan(C_2) \\ Var(b) \subseteq Var(C_1) \end{array}}{Split_B(S_1 \lhd b \rhd S_2, \quad C_1 \lhd b \rhd C_2, D)}$$

Proof. $S_1 \lhd b \rhd S_2$ $\{cond\ is\ monotonic\}$

$\sqsubseteq (C_1 \parallel D) \lhd b \rhd (C_2 \parallel D)$ $\{(\parallel -cond\ distr)\}$

$= (C_1 \lhd b \rhd C_2) \parallel D$ $\square$

Bottom-Up Rule for Iteration

$$\frac{Split_B(S, C, D),\ Var(b) \subseteq Var(C)}{Split_B(b * S, b * C, D)}$$

Proof. $b * S$ $\{loop\ operator\ is\ monotonic\}$

$\sqsubseteq b * (C \parallel D)$ $\{Corollary\ 5.2\}$

$= (b * C) \parallel D.$ $\square$

When $Var(b) \cap Var(D) \neq \emptyset$, we will introduce a local variable lb, and rewrite the conditional and iteration into the forms

$$\textbf{var}\ lb \bullet (lb := b;\ S_1 \lhd lb \rhd S_2) \text{ and } \textbf{var}\ lb \bullet (lb := b; lb * (S; lb := b))$$

respectively by laws DL1 and DL2. The partitioning rule for $lb := b$ will be discussed later. The following rule deals with partitioning of $S_1 \sqcap S_2$.

Bottom-Up Rule for Non-Deterministic Choice

$$\frac{\begin{array}{c} Split_B(S_i, C_i, D), \quad i = 1,2 \\ Var(S_1) = Var(S_2), \quad Chan(C_1) = Chan(C_2) \end{array}}{Split_B(S_1 \sqcap S_2, \ C_1 \sqcap C_2, \ D)}$$

Proof. $S_1 \sqcap S_2$ $\{\sqcap\ is\ monotonic\}$

$\sqsubseteq (C_1 \parallel D) \sqcap (C_2 \parallel D)$ $\{(\parallel\ disj)\}$

$= (C_1 \sqcap C_2) \parallel D$ $\square$

The partitioning rule for general guarded choice constructs is presented as:

Bottom-Up Rule for Guarded Choice

$$\frac{\begin{array}{c} Split_B(S_i, C_i, D), \quad i = 1,2 \\ Var(S_1) = Var(S_2), \quad Chan(C_1) = Chan(C_2) \\ Var(g_i) \subseteq Var(C_1), \quad Chan(g_i) \subseteq Chan(C_1), \quad i = 1,2 \end{array}}{Split_B((g_1\, S_1)[\!](g_2\, S_2), (g_1\, C_1)[\!](g_2\, C_2), D)}$$

Proof. $(g_1\, S_1)[\!](g_2\, S_2)$ $\{[\!]\ is\ monotonic\}$

$\sqsubseteq ((g_1\, C_1) \parallel D)[\!]((g_2\, C_2) \parallel D)$ $\{(gc\ exp)\}$

$= ((g_1\, C_1)[\!](g_2\, C_2)) \parallel D$ $\square$

6.2 The Top-Down Splitting Approach

In this approach, both hardware and software components of the source program are assembled from those of its constituents.

Before presenting a set of *top-down* splitting rules, we introduce the notion of *interface-consistency* for hardware components.

Definition 6.2. *Let* $D_k =_{df} \mu X \bullet ([\!]_{i \in I_k}(r_i?x_i; M_i; a_i!y_i; X)[\!]skip)$, *for* $k = 1, 2$. D_1 *and* D_2 *are said to be interface-consistent, denoted by* $consist(D_1, D_2)$, *if*

$$Var(D_1) = Var(D_2) \text{ and } Chan(D_1) \backslash B_1 = Chan(D_2) \backslash B_2,$$

where $B_i =_{df} \{r_j, a_j \mid j \in I_i\}$, *for* $i = 1, 2$.

In such a case, we define

$$D = union(D_1, D_2) =_{df} \mu X \bullet ([\!]_{i \in I_1 \cup I_2}(r_i?x_i; M_i; a_i!y_i; X)[\!]skip) \qquad \square$$

We first present a basic rule, from which and the *bottom-up* rules we obtain the corresponding *top-down* rule straightforwardly in each case.

Rule for Hardware Augmentation

$$\frac{\begin{array}{c} Split_{B_1}(S, C, D_1) \\ consist(D_1, D_2), \quad Chan(C) \cap B_2 \subseteq B_1 \end{array}}{Split_{B_1 \cup B_2}(S, C, union(D_1, D_2))}$$

Cordial readers can access the proof for this rule in our previous report[16].

The following *top-down* rules are directly derived from the basic rule and those *bottom-up* rules.

Top-Down Rule for Sequential Composition

$$\frac{\begin{array}{c} Split_{B_i}(S_i, C_i, D_i), \quad i = 1, 2 \\ Var(S_1) = Var(S_2), \quad Chan(S_1) = Chan(S_2) \\ consist(D_1, D_2) \end{array}}{Split_{B_1 \cup B_2}(S_1; S_2, C_1; C_2, union(D_1, D_2))}$$

Top-Down Rule for Conditional

$$\frac{\begin{array}{c} Split_{B_i}(S_i, C_i, D_i), \quad i = 1, 2 \\ Var(S_1) = Var(S_2), \quad Chan(S_1) = Chan(S_2) \\ consist(D_1, D_2), \quad Var(b) \subseteq Var(C_1) \end{array}}{Split_{B_1 \cup B_2}(S_1 \lhd b \rhd S_2, \quad C_1 \lhd b \rhd C_2, \quad union(D_1, D_2))}$$

Top-Down Rule for Guarded Choice

$$\frac{\begin{array}{c} Split_{B_i}(S_i, C_i, D_i), \quad i = 1, 2 \\ Var(S_1) = Var(S_2), \quad Chan(S_1) = Chan(S_2), \quad consist(D_1, D_2) \\ Var(g_i) \subseteq Var(C_1), \quad Chan(g_i) \subseteq Chan(C_1), \quad i = 1, 2 \end{array}}{Split_{B_1 \cup B_2}((g_1\, S_1)[\!](g_2\, S_2), (g_1\, C_1)[\!](g_2\, C_2), union(D_1, D_2))}$$

As a special case of the guarded choice construct, the *top-down* rule for non-deterministic choice is omitted here, readers can find it in [16].

6.3 Splitting Primitive Commands

This section demonstrates how to partition an assignment $u := e(v)$. We will focus on those cases where both hardware and software participate in the evaluation of $e(v)$ and the update of u.

Case 1. $e(v)$ is a busy expression, and the variable v has been allocated to the hardware component.

$Split_B(u := e(v), C, D)$, where

$C =_{df} (r_j\,!\,1;\ a_j\,?\,u)$, and $D =_{df} \mu X \bullet ((r_j\,?\,x;\ y := e(v);\ a_j\,!\,y;\ X)\,[\!]\ skip)$.

Case 2. $e(v)$ is a busy expression, however, v has been allocated to the software component. $Split_B(u := e(v), C, D)$, where

$C =_{df} (r_j\,!\,v;\ a_j\,?\,u)$, and $D =_{df} \mu X \bullet ((r_j\,?\,x;\ y := e(x);\ a_j\,!\,y;\ X)\,[\!]\ skip)$.

Case 3. $e(v)$ is not a busy expression, but v is allocated to the hardware component.

$Split_B(u := e(v), C, D)$, where

$C =_{df} (\mathbf{var}\ lv \bullet (r_j\,!\,1;\ a_j\,?\,lv;\ u := e(lv)))$, and

$D =_{df} \mu X \bullet ((r_j\,?\,x;\ y := v;\ a_j\,!\,y;\ X)\,[\!]\ skip)$.

More intricate case of assignment $u := e(v, w)$, where v and w have respectively been allocated to the software component and the hardware one, will be converted to several successive assignments with the form we have dealt with above, by the algebraic law with respect to assignments.

6.4 The Example Revisited

Consider the source program S illustrated in Example 4.3. Suppose the programmer has marked S as follows based on the results of the static analysis and other concerns:

$$\begin{aligned} S^M : \; & ic_1\,?\,u; ic_2\,?\,v_H; w := 2 \times (u + v_H);\\ & oc\,!\,w; ic_3\,?\,z; (z := u \times v_H) \lhd (z = u) \rhd skip;\\ & (u < (v_H \times v_H))_H * (u := u + w; w := ((u - v_H) \times (u + v_H))_H; oc\,!\,w) \end{aligned}$$

where the variable v has been allocated to hardware, and as well there are two expressions ($u \times v$, $(u - v) \times (u + v)$) left to hardware.

Either by the *bottom-up* approach or via the *top-down* one, we obtain the same hardware component and the same software component as follows.

The hardware component:

$$\begin{aligned} & req_1\,?\,x; v := x; ack_1\,!\,y\\ [\!] & req_2\,?\,x; y := v; ack_2\,!\,y\\ [\!] & req_3\,?\,x; y := x < (v \times v); ack_3\,!\,y\\ [\!] & req_4\,?\,x; y := (x - v) \times (x + v); ack_4\,!\,y \end{aligned}$$

where $B = \{req_i, ack_i \mid 1 \le i \le 4\}$ is the internal channels shared by the software and hardware components.

The software component:

$$\begin{aligned} & ic_1\,?\,u; ic_2\,?\,lb; req_1\,!\,lv; ack_1\,?\,la;\\ & req_2\,!\,1; ack_2\,?\,lv; w := 2 \times (u + lv);\\ & oc\,!\,w; ic_3\,?\,z;\\ & (req_2\,!\,1; ack_2\,?\,lv; z := u \times lv) \lhd (z = u) \rhd skip;\\ & req_3\,!\,u; ack_3\,?\,b;\\ & b * (u := u + w; req_4\,!\,u; ack_4\,?\,w; oc\,!\,w; req_3\,!\,u; ack_3\,?\,b) \end{aligned}$$

where lv, la, b are the local variables used by the software.

It is worth noting that the software component can be optimized to a much simpler form by data flow analysis[9].

7 Conclusion

This paper shows how the hardware/software partitioning problem can be tackled in the algebra of programs. The partitioning task consists of the static program analysis phase and the splitting phase. The former provides the information for moving operations from software to hardware and reducing the communication between components, and the latter supports a compositional approach to program partitioning. To synchronize software and hardware components, and to reduce the complexity of their interface, we introduce a simple handshaking protocol, and propose a normal form for hardware components. The correctness of the splitting process is verified using the algebraic laws of the source language.

Our co-design approach is proposed to design embedded computer systems, which are widely used in our daily life, by coding the source program from customers' primitive requirements, and then transforming it to hardware and software components. It is worth noting that our approach is rather general in the sense that it can be used in the area of program parallelism. In our future work, we will introduce timing constraints into the source program and strengthen the program analysis phase. Meanwhile, our attentions will be paid to the reconfiguration and reusability of hardware components and the algebraic transformation from the hardware specification down to synthesizable normal forms.

References

[1] Silva L, Sampaio A, Barros E. A normal form reduction strategy for hardware/software partitioning. *Formal Methods Europe (FME) 97, Lecture Notes in Computer Science* 1313, 1997, pp.624–643.

[2] He Jifeng *et al.* Provably correct systems. *Lecture Notes in Computer Science* 863, 1994, pp.288–335.

[3] He Jifeng, Bowen J P. Specification, verification and prototyping of an optimised compiler. *Formal Aspect of Computing,* 1994, 6: 643–658.

[4] Sampaio A. An Algebraic Approach to Compiler Design. *World Scientific,* 1997.

[5] Page I, Luk W. Compiling Occam into FPGAs. In *FPGAs*, Will Moore, Wayne Luk (eds.), Abingdon EE&CS Books, pp.271–283, 1991.

[6] He Jifeng, Page I, Bowen J. A provable hardware implementation of Occam. *Lecture Notes in Computer Science* 711, 1993, pp.693–703.

[7] Balboni A *et al.* Partitioning and exploration strategies in the TOSCA design flow. In *Proceedings of Fourth International Workshop on Hw/sw Co-design,* IEEE Computer Society Press, 1996, pp.62–69.

[8] Cheung T. A multi-level transformation approach to hardware/software co-design. In *Proceedings of Fourth International Workshop on Hw/sw Co-design,* 1996, pp.10–17.

[9] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis, Springer-Verlag, 1999.

[10] Roscoe A W, Hoare C A R. Laws of Occam programming. *Theoretical Computer Science,* 1988, 60: 177–229.

[11] He Jifeng. Provably Correct Systems: Modelling of Communication Languages and Design of Optimised Compilers, McGraw-Hill Publisher, 1994.

[12] Hoare C A R. Communicating Sequential Processes, Prentice Hall, 1985.

[13] Hoare C A R *et al.* Laws of programming. *Communications of the ACM,* 1987, 30(8): 672–686.

[14] Hoare C A R, He Jifeng. Unifying Theories of Programming, Prentice Hall, 1998.

[15] Qin Shengchao, He Jifeng. An algebraic approach to hardware/software partitioning. In *Proceedings of the 7th IEEE ICECS'2K,* Lebanon, December, 2000, pp.273–276.

[16] Qin Shengchao, He Jifeng. An algebraic approach to hardware/software partitioning. UNU/IIST Research Report 206, Macau, June, 2000.

[17] Mathematics of Program Construction Group. Fixed-point calculus. *Information Processing Letters,* 1995, 53: 131–136.

[18] He Jifeng. Converting programs into delay insensitive circuits. Technical Report, Programming Research Group, Oxford University Computing Laboratory, 1994.

QIN Shengchao was born in 1974. He is a Ph.D. candidate in Department of Informatics, School of Mathematical Science, Peking University. He got his B.S. degree in the same department in 1997. His research interests include formal methods and semantics, unifying theories to programming, and formal engineering approaches.

HE Jifeng is a senior research fellow of UNU/IIST. He was a senior research fellow of the programming research group, Oxford University before 1998. He is a professor of computer science in East China Normal University since 1986, Shanghai. His research interest lies in the sound methods of specification of computer systems, communications, application of standards, and the techniques for designing and implementing those specifications in software and/or hardware with high reliability and low cost.

QIU Zongyan is a professor of computer science and deputy head of the Department of Informatics, Peking University. His research interests are formal methods, programming languages, and real-time systems.

ZHANG Naixiao is a professor of computer science in the Department of Informatics, Peking University. His research interests are formal methods and domain-specific languages.