AN ALGEBRAIC APPROACH TO HARDWARE/SOFTWARE PARTITIONING

Qin Shengchao\textsuperscript{1,2,*} and He Jifeng\textsuperscript{1}
\textsuperscript{1}UNU/IIST, P.O.Box 3058, Macau, China
\textsuperscript{2}Peking University, Beijing, China
\{qsc, jifeng\}@iist.unu.edu

ABSTRACT: Hardware and software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of co-design process is to decompose a program into hardware and software. This paper proposes an algebraic partitioning method whose correctness is verified in the algebra of programs. We introduce the program analysis phase before program partitioning and develop a collection of syntax-based splitting rules, where 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 the program partitioning.

1 INTRODUCTION

The design of a complex software product like a nuclear reactor control system is ideally decomposed into a progression of related phases. It starts with an investigation of the properties and behaviours of the process evolving within its environment, and an analysis of 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 in a program expressed in a high level language. After translation into a machine code of the chosen computer, it is executed at 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.

With chip size reaching one million transistors, the complexity of VLSI algorithms is approaching that of software algorithms. However, the design methods for circuits resemble the low level machine language programming methods. Selecting individual gates and registers in a circuit like selecting individual machine instruction in a program. State transition diagrams are like flowcharts. These methods may have been adequate for small circuit design when they were introduced, but they are not adequate for circuits that perform complicated algorithms. 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, 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 co-design process is to partition a 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 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. 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". The original ProCoS project [6] concentrated almost exclusively on the verification of standard compiler of a high-level programming language based on Occam down to a microprocessor based on Transputer [5]. Sampao showed how to reduce the compiler design task to one of program transformation; his formal framework is also a procedural language and its algebraic laws [16]. Towards the end of the first phase of the project, Ian Page et al made rapid advance in the development of hardware compilation technique using an Occam-like language targeted towards Field Pro-
grammable Gate Arrays [12], and He Jieng et al provided a formal verification of the hardware compilation scheme within the algebra of Occam programs [4].

Recently, some works have suggested the use of formal methods for the partitioning process [1, 2, 17]. Balboni et al adopt Occam as an internal model for the system exploration and partitioning strategy. Cheung pursues 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 [17], Silva et al provide a formal strategy for carrying out the splitting phase automatically, and present 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 huge number of parallel processes actually increase the data flow between the hardware and software components.

The remainder of this paper is organized as follows. Section 2 describes the splitting 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. A simple conclusion is followed in section 7.

2 SPLITTING STRATEGY

Our partitioning strategy is described as follows. Suppose a process program has been coded by the programmer in the source programming language out of the customer’s requirements. A static analysis [11] is performed on that program to obtain useful statistical data, such as quantitative information concerned with occurrences of expressions and variables, distributive information with respect to those variables occurring in expressions. Based on this analysis, the programmer marks those parts of the program that are worth to be implemented by hardware and leaves others to software, and as well divides the interface of the program to two disjoint parts. The program marking and interface partitioning are conducted by the following guidelines:

- Generally busy expressions should be marked out and implemented by hardware, to gain 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 cost.

Additionally, the customer’s demands concern with the performance and the cost should be taken into account.

We take such a marked program as input of our hardware/software splitting process, which generates as output a program comprising of two concurrent processes representing software and hardware components respectively.

3 PRELIMINARIES

The language we select to perform hardware/software partitioning is a subset of Occam which was designed for constructing communicating systems.

1. Sequential Process:
   
   \[ S ::= \text{PC (primitive command)} \mid S; S \mid S \bowtie S \mid S \bowtie b \bowtie S \mid S \bowtie \varnothing \mid (g S) \mid (g S) \mid \varnothing S \mid \text{variable declaration} \]

   where \( PC ::= (x := e) \mid \text{skip} \mid \varnothing \mid c ! e \mid c ? x \) and \( g \) is \( \text{skip} \) or a communication event \( c ! e \) or \( d ? x \).

2. Parallel Program:
   
   \[ P ::= S \mid P \mid P \]

In later discussions, we adopt \( \text{Var}(P) \) and \( \text{Chan}(P) \) to denote the set of variables and channels employed by \( P \).

As a subset of Occam, the language enjoys a rich set of algebraic laws presented in [15, 3, 7, 9, 8]. We explore a collection of algebraic laws ([13]) which will be employed within the proofs in the following sections. Here we omit them to meet the limit of space.

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

**Definition 3.3 (Refinement)**

Given programs \( P, Q \), we say \( Q \) is a refinement of \( P \), denoted as \( P \models Q \), if \( (\text{skip} P) || \text{skip} Q = P \) is algebraically provable.

4 THE STATIC ANALYSIS

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

First, we introduce a function complex for expressions, which specifies the complexity of expressions.

**Definition 4.1** \( \text{complex} : \text{Expr} \rightarrow \mathbb{N} \) is inductively defined on the structure of expressions:

- \( \text{complex}(v) =\_df 1 \), for any variable \( v \),
- \( \text{complex}(c) =\_df 0 \), for any constant \( c \), and
- \( \text{complex}(\text{op}(e_1, \ldots, e_n)) =\_df \sum_{i=1}^{n} \text{complex}(e_i) + \text{complex}(\text{op}) \),

where \( \text{op} \) is any operator used to construct expressions in...
the source language, and complex(op) is defined by the
programmer in accordance with the complexity of op.

An expression is regarded as a busy expression if it oc-
curs often in the program or owns an intricate structure.
The analysis generates a table to record the occurrence
frequency of expressions.

\[ \Phi(S) = \{ (e, n(e)) \mid e \in \text{Expr}(S) \} \]

where \( S \) is a program, \( n(e) \) represents the number of oc-
currences of the non-trivial expression \( e \) in \( S \), i.e., \( e \) is
neither a single variable nor a constant. From this table, the
designer can figure out those busy expressions.

The analysis produces a table

\[ \Psi(S) = \{ (v, \text{eset}(v)) \mid v \in \text{Var}(S) \} \]

for variables employed by \( S \), where \( \text{eset}(v) \) is the set of
expressions containing the variable \( v \). Such a table can
provide helpful information for the variable partitioning.

We propose an algorithm to generate the results of
the analysis, which is omitted here because of the limit of
space. Readers can refer to [13, 14], where the former
presents the details of the simple analysis, the latter poses
a more detailed analysis which takes data types and
procedures/subroutines into account and illustrates an exam-
ple with respect to the design of an ATM switch.

5 THE HARDWARE/SOFTWARE
TARGET ARCHITECTURE

This section describes the target architecture of our
partitioning approach by refining hardware and soft-
ware components to specially chosen forms. To synchro-
nize their activities, we introduce a simple handshake
protocol to streamline communications between them.

Suppose \( B = \{ r_j, a_j \mid j \in I \} \) is a set of channels, we
define \( CP(B) \) as a subset of the source language, com-
prising processes \( C \) with \( \text{Chan}(C) \supseteq B \) and one of
the following forms.

1. a sequential process not using channels in \( B \).
2. \( r_j \vdash v; C; a_j \vdash x \), where \( C \) is a member of \( CP(B) \)
   not interacting via any channel in \( B \).
3. \( C_1; C_2 \), or \( C_1 \not< b \supset C_2 \), or \( (g_i, C_1)[(g_2, C_2)] \),
   where both \( g_i \) and \( C_i \) lie in \( CP(B) \), for \( i = 1, 2 \).
4. \( b \star C \), where \( C \) is a member of \( CP(B) \).

To simplify the interface design, we confine the inter-
actions between the hardware and software compo-
nents to the communications along the channels from the set
\( B \). Our partitioning rules will select the software com-
ponents from the set \( CP(B) \), and organize the hardware
component in the form of

\[ D = \mu X \star \{ \ell_1 \vdash (r_j \vdash x; M_j; \alpha_j[g]; X); \text{skip} \} \]

where none of \( M_j \) mentions channels in \( B \). The
communicating process \( D \) represents a digital device which
provides 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** For any \( C_1, C_2 \) in \( CP(B) \), we have

\[ (C_1 ; C_2) \parallel D = (C_1 \parallel D); (C_2 \parallel D). \]

The proof (presented in [13]) is omitted here because of
the limit of space.

**Corollary 5.2** If \( C \in CP(B) \), then

\[ (b \star C) \parallel D = b \star (C \parallel D). \]

6 SYNTAX-BASED SPLITTING
RULES

This section is devoted to the design of program split-
ting 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 those of its constituents. We establish the correct-
ness of those rules by using the algebraic laws we have
explored.

We introduce a predicate \( \text{Split} \), which will be of great
help in formalizing the decomposition rules.

**Definition 6.1** (Split)

Let \( B = \{ r_j, a_j \mid j \in I \} \). Given the source program \( S \),
its hardware/software partition \( (C, D) \) is specified by the
following predicate:

\[ \text{Split}(S, C, D) = \exists \]

\[ S \subseteq (C \parallel D) \land \text{Var}(C) \cap \text{Var}(D) = \emptyset \land \]

\[ \text{Chan}(C) \cap \text{Chan}(D) = \emptyset \land \]

\[ \text{Chan}(C), \text{Input} \cap \text{Chan}(D), \text{Input} = \emptyset \land \]

\[ \text{Chan}(C), \text{Output} \cap \text{Chan}(D), \text{Output} = \emptyset \land \]

\[ C \in CP(B) \land D \in H(B) \]

The splitting task can be undertaken in two different
approaches: the bottom-up approach and the top-down one,
where the former builds the hardware component from
the marked source program in one step, and constructs
the software component from those of its constituents,
while the latter assembles both the hardware and soft-
ware components from those of its constituents.

A complete set of splitting rules are explored in [13].
Here we only present four of them for the space limit.

**Bottom-up Rule for Sequential Structure**

\[ \text{Split}(S_1, C_1, D), i = 1, 2 \]

\[ \text{Var}(S_1) = \text{Var}(S_2), \text{Chan}(C_1) = \text{Chan}(C_2) \]

\[ \text{Split}(S_1; S_2, C_1, C_2, D) \]

**Proof**

\[ S_1 : S_2 \]

\[ \Gamma \subseteq (C_1 \parallel D); (C_2 \parallel D) \]

\[ \Gamma \subseteq (C_1, C_2) \parallel D \] (Th. 5.1)

**Bottom-up Rule for Iteration**

\[ \text{Split}(S, C, D), \text{Var}(b) \subseteq \text{Var}(C) \]

\[ \text{Split}(b \star S, b \star C, D) \]
Before presenting the top-down splitting rules, we introduce the notion of interface-consistency on hardware components:

**Definition 6.2 (Interface-consistency)**

Let $D_k = \mu X \cdot \{ \langle r_i, z_i ; M_i ; a_i | y_i ; X \rangle \} \text{skip} \}$, for $k = 1, 2, D_1$ and $D_2$ are said to be interface-consistent, denoted by $\text{Consistency}(D_1, D_2)$, if

$\var(\text{Var}(D_1)) = \var(\text{Var}(D_2))$, and

$\text{Chan}(\text{Var}(D_1)) \cdot \text{Chan}(\text{Var}(D_2)) \cdot \text{Chan}(B_i) = \text{Chan}(B_{i+1}) \cdot B_{i+2}$,

where $B_i = \mu \text{skip} \cdot \{ \langle r_j, a_j | j \in I \rangle \}$, for $i = 1, 2$.

In such a case, we define

$D = \mu X \cdot \{ \langle r_i, z_i ; M_i ; a_i | y_i ; X \rangle \} \text{skip} \}$

We present two splitting rules in the top-down approach as follows.

**Top-down Rule for Conditional**

$\text{Split}_{B_i}(S_i, C_i, D_i), \ i = 1, 2$

$\var(S_1) = \var(S_2)$, $\text{Chan}(S_1) = \text{Chan}(S_2)$

$\text{Consistency}(D_1, D_2)$

$\text{Var}(b) \subseteq \text{Var}(C_i)$

$\text{Split}_{B_i \cup B_{i+1}}(S_i \land b \land C_i \land b \land C_{i+1} \land D)$

**Top-down Rule for Guarded Choice**

$\text{Split}_{B_i}(S_i, C_i, D_i), \ i = 1, 2$

$\var(S_1) = \var(S_2)$, $\text{Chan}(S_1) = \text{Chan}(S_2)$

$\text{Consistency}(D_1, D_2)$, $D = \mu X \cdot \{ \langle r_i, z_i ; M_i ; a_i | y_i ; X \rangle \} \text{skip} \}$

Based on the results of the analysis, an assignment $u := e(\nu)$ can be decomposed to hardware and software components. We present one case here, others in [13].

Case 1: $e(\nu)$ is a busy expression, however, $u, v$ have been allocated to the software component.

$\text{Split}_{D_i}(u := e(\nu), C, D)$, where

$C = \mu \text{skip} \cdot \{ \langle r_j, y_i ; a_i | y_i \rangle \} \mu X \cdot \{ \langle r_i, z_i ; M_i ; a_i | y_i ; X \rangle \} \text{skip} \}$

$D = \mu \text{skip} \cdot \{ \langle r_i, z_i ; M_i ; a_i | y_i ; X \rangle \} \text{skip} \}$

**REFERENCES**


