AN ALGEBRAIC APPROACH TO HARDWARE/SOFTWARE PARTITIONING

Qin Shengchao[1,2,*] and He Jifeng[1]
[1]UNU/IIST, P.O.Box 3058, Macau, China
[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]. Sampaio 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-

*Partially supported by NSFC No. 69873003

grammable Gate Arrays [12], and He Jifeng *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 source 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 concerned 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:

$S ::= PC$ (primitive command)
$\quad\mid S; S$ (sequential composition)
$\quad\mid S \lhd b \rhd S$ (conditional) $\mid b * S$ (iteration)
$\quad\mid (g\ S)\ [\!]\ (g\ S)$ (guarded choice)
$\quad\mid \mathbf{var}\ x \bullet S$ (variable declaration)

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:

$P ::= S \mid P \parallel P$

In later discussions, we adopt *Var*(P) and *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 $P \sqsubseteq Q$, if $(skip\ P)[\!](skip\ Q) = P$ 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 $complex : Expr \to N$ is inductively defined on the structure of expressions:
$complex(v) =_{df} 1$, for any variable v,
$complex(c) =_{df} 0$, for any constant c, and
$complex(\mathbf{op}(e_1, \ldots, e_n)) =_{df}$
$\quad \sum_{i=1}^{n} complex(e_i) + complex(\mathbf{op})$,
where $\mathbf{op}$ is any operator used to construct expressions in

the source language, and *complex*($\mathbf{op}$) is defined by the programmer in accordance with the complexity of $\mathbf{op}$. □

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

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

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. From this table, the designer can figure out those busy expressions.

The analysis produces a table

$\Psi(S) \;=\; \{(v, eset(v)) \mid v \in \mathit{Var}(S)\}$

for variables employed by S, where $eset(v)$ 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 example 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 confining hardware and software components to specially chosen forms. To synchronize their activities, we introduce a simple handshaking 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, comprising processes C with $\mathit{Chan}(C) \supseteq B$ and one of the following forms.

(1). a sequential process not using channels in B.

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

(3). $C_1;\ C_2$, or $C_1 \lhd b \rhd C_2$, 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).

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

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

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 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 $C \in \mathit{CP}(B)$, then

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

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 those of its constituents. We establish the correctness of those rules by using the algebraic laws we have explored.

We introduce a predicate *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:

$$\begin{aligned}
&\mathit{Split}_B(S, C, D) =_{df}\\
&\quad S \sqsubseteq (C \parallel D) \;\wedge\; \mathit{Var}(C) \cap \mathit{Var}(D) = \emptyset \;\wedge\\
&\quad \mathit{Chan}(C) \cap \mathit{Chan}(D) \;=\; B \;\wedge\\
&\quad \mathit{Chan}(C).\mathit{Input} \,\cap\, \mathit{Chan}(D).\mathit{Input} \;=\; \emptyset \;\wedge\\
&\quad \mathit{Chan}(C).\mathit{Output} \,\cap\, \mathit{Chan}(D).\mathit{Output} \;=\; \emptyset \;\wedge\\
&\quad C \in \mathit{CP}(B) \;\wedge\; D \in H(B)
\end{aligned}$$ □

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, whereas the latter assembles both the hardware and software 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 Composition

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

Proof

$$\begin{aligned}
& S_1\ ;\ S_2 && \{;\ \mathit{is\ monotonic}\}\\
\sqsubseteq\ & (C_1 \parallel D);\ (C_2 \parallel D) && \{\mathit{Th.\ 5.1}\}\\
=\ & (C_1;\ C_2) \parallel D &&
\end{aligned}$$ □

Bottom-up Rule for Iteration

$$\frac{\mathit{Split}_B(S,\ C,\ D),\ \mathit{Var}(b) \subseteq \mathit{Var}(C)}{\mathit{Split}_B(b * S,\ b * C,\ D)}$$

Before presenting the *top-down* splitting rules, we introduce the notion of *interface-consistency* on hardware components:

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

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

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

Top-down Rule for Conditional

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

Top-down Rule for Guarded Choice

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

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

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

$Split_B(u := e(v),\ C,\ D)$, where
$C =_{df} (r_j\,!\,v;\ a_j\,?\,u)$, and
$D =_{df} (r_j\,?\,x;\ y := e(x);\ a_j\,!\,y)$ $\square$

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, where 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 the program partitioning. To synchronize software and hardware components, and reduce the complexity of their interface, we introduce a simple handshaking protocol, and propose a normal form for the hardware components. The correctness of the splitting process is verified using the algebraic laws of the source language.

REFERENCES

[1] A. Balboni *et al*, "Partitioning and Exploration Strategies in the TOSCA Design Flow", In *Proceedings of Fourth International Workshop on Hardware/Software Codesign*, 62–69, IEEE Computer Society Press, (1996).

[2] T. Cheung, "A Multi-level Transformation Approach to Hardware/Software Co-design", In *Proceedings of Fourth International Workshop on Hardware/Software Codesign*, 10–17, (1996).

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

[4] He Jifeng, I. Page and J. Bowen, "A Provable Hardware Implementation of Occarn", *Lecture Notes in Computer Science* 711, 693–703, (1993).

[5] He Jifeng and J. Bowen, "Specification, Verification and Prototyping of an Optimised Compiler", *Formal Aspect of Computing* 6, 643–658, (1994).

[6] He Jifeng *et al*, "Provably Correct Systems", *Lecture Notes in Computer Science* 863, 288–335, (1994).

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

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

[9] C.A.R. Hoare *et al*, "Laws of Programming", *Communications of the ACM*, Vol 30(8): 672-686, 1987.

[10] Mathematics of Program Construction Group, "Fixed-point Calculus", *Information Processing Letters*, 53(1995) 131-136.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, *Principles of Program Analysis*, Springer-Verlag, 1999.

[12] Ian Page and Wayne Luk, "Compiling Occam into FPGAs", in *FPGAs*, eds., Will Moore and Wayne Luk, 271-283, Abingdon EE&CS books, 1991.

[13] Qin Shengchao and He Jifeng, "An Algebraic Approach to Hardware/software Partitioning", *Technical Report 206*, UNU/IIST, June, 2000.

[14] Qin Shengchao and He Jifeng, "Partitioning Program into Hardware and Software", draft paper, UNU/IIST, August, 2000.

[15] A.W.Roscoe and C.A.R. Hoare, "Laws of Occam Programming", *Theoretical Computer Science*, Vol 60: 177-229, 1988.

[16] Augusto Sampaio, "An Algebraic Approach to Compiler Design", *World Scientific*, (1997).

[17] L. Silva, A. Sampaio and E. Barros, "A Normal Form Reduction Strategy for Hardware/software Partitioning", *Formal Methods Europe (FME) 97, LNCS, 1313*, 624-643, (1997).