An Optimal Approach to Hardware/software Partitioning for Synchronous Model

Pu Geguang†§⋆, Dang Van Hung†, He Jifeng†⋆⋆, and Wang Yi‡

†International Institute for Software Technology
United Nations University, Macau.
‡ Department of Computer Systems, Uppsala University, Sweden.
§ LMAM, Department of Informatics, School of Math.
Peking University, Beijing, China 100871
`yi@docs.uu.se, {ggpu,dvh,jifeng}@iist.unu.edu`

Abstract. Computer aided hardware/software partitioning is one of the key challenges in hardware/software co-design. This paper describes a new approach to hardware/software partitioning for synchronous communication model. We transform the partitioning into a reachability problem of timed automata. By means of an optimal reachability algorithm, an optimal solution can be obtained in terms of limited resources in hardware. To relax the initial condition of the partitioning for optimization, two algorithms are designed to explore the dependency relations among processes in the sequential specification. Moreover, we propose a scheduling algorithm to improve the synchronous communication efficiency further after partitioning stage. Some experiments are conducted with model checker UPPAAL to show our approach is both effective and efficient.

Keywords: hardware/software partitioning, timed automata, reachability, scheduling algorithm.

1 Introduction

Computer system specification is usually completely implemented as software solution. However, some strong requirements for performance of the system demand an implementation fully in hardware. Consequently, in between the two extremes, Hardware/software Co-design [25], which studies systematically the design of systems containing both hardware/software components, has emerged as an important field. A critical phase of co-design process is to partition a specification into hardware and software components.

One of the objectives of hardware/software partitioning is to search a reasonable composition of hardware and software components which not only satisfies

⋆ Partially supported by NNSFC No. 60173003

⋆⋆ On leave from East China Normal University. The work is partially supported by the 973 project 2002CB312001 of the ministry of science and technology, and the 211 project of the ministry of Education of China.

the constraints such as timing, but also optimized desired quality metrics, such as communication cost, power consumption and so on.

Several approaches based on algorithms have been developed, as described, for example, in [2, 20, 21, 24, 26]. All these approaches emphasize the algorithmic aspects, for instance, integer programming [20, 26], evolution algorithm [24] and simulated annealing algorithm [21] are respectively introduced to the partitioning process in the previous researches. These approaches are applied to different architectures and cost functions. For example, in [26], Markus provided a technique based on integer programming to minimize the communication cost and total execution time in hardware with certain physical architecture. The common feature of these approaches is that the communication cost is simplified as a linear function on data transfer or the relation between adjacent nodes in task graph. This assumption is reasonable in asynchronous communication model, but is unreasonable in synchronous communication model in which the cost of waiting time for communication between processes is very high. In order to manage the synchronous model which is common in many practical systems, a new approach should be introduced into the partitioning problem.

A number of papers in the literature have introduced formal methods into the partitioning process [17, 23, 3]. Some of them adopted a subset of the Occam language [16] as specification language [17, 23]. In [23], for example, Qin provided a formal strategy for carrying out the partitioning phase automatically, and presented a set of proved algebraic laws of the partitioning process. In that paper, he did not deal with the optimization of the partitioning.

Few approaches deal with the analysis of the specification for exploring the hidden concurrency to relax the initial condition of the partitioning for optimization. In [17], Iyoda *et al.* provided several algebraic laws to transform the initial description of the system into a parallel composition of a number of simple processes. However, the method delivers a large number of processes and communication channels, which not only poses a complicated problem of merging those small processes, but also raises the communication load between the hardware and software components.

In this paper, we present an optimal automatic partitioning model. In this model, we adopt an abstract synchronous architecture composed of a coprocessor board and a hardware device such as FPGAs, ASIC etc, where the communication between them is synchronized. By means of our approach, the following goals will be achieved

- Explore the hidden concurrency, i,e, find the processes which could be executed in parallel from the initial sequential specification.
- Obtain the optimal performance of the overall program in terms of the limited resources in hardware. The communication waiting time between software and hardware components is considered as well.
- Improve the communication efficiency by re-ordering the commands to reduce the communication waiting time between hardware and software components after partitioning.

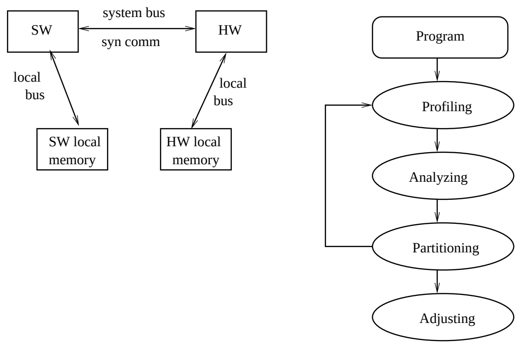

Fig. 1. Architecture and Partitioning Flow

Given a specification, system designers are required to divide the specification into a set of basic processes (blocks) which are regarded as candidate processes for the partitioning phase. In general, the next step is to select and put some processes into hardware components to obtain the best performance. On account of the parallel structure of software and hardware components, the hidden concurrency among the processes will relax precedence condition of the partitioning, that is, an optimal solution will be obtained from a larger search space. We design two algorithms to explore the control and data flow dependency. To allocate the processes into the software and hardware components, we transform the partitioning to a reachability problem of timed automata [10], and obtain the best solution by means of an optimal reachability algorithm. Since the synchronous communication model is adopted in our target architecture, to reduce the communication waiting time further, we adjust communication commands in the program of each component by applying a scheduling algorithm.

The paper is organized as follows. Section 2 presents the overview of our technique. Section 3 explores the dependency relation between processes. Section 4 describes a formal model of hardware/software partitioning using timed automata. In Section 5, we propose a scheduling algorithm to improve the communication efficiency. Some partitioning experiments are conducted in Section 6. Finally, Section 7 is a short summary of the paper.

2 Overview of Our Partitioning Approach

In this section we present the overview of our approach to hardware/software partitioning problem. The partitioning flow is depicted as Figure 1.

In profiling stage, a system specification is divided into a set of basic candidate processes which could never be split further. However, there is a trade-off between the granularity of the candidates and the feasibility of optimization. The smaller the candidate processes are, the greater the number of different partitions is. The large number of partitions will increase the time dramatically in the computation of the optimum. Furthermore, smaller candidate processes will bring heavy communication cost. On the other hand, larger candidates will restrict the space of possibilities, therefore, may reduce the concurrency and increase the waiting time for communication. We leave this choice to the designers to repeat the profiling process as long as the partitioning results are not satisfied with according to the current granularity of candidate process. Once the designer decides the granularity, the initial specification is transformed into a sequential composition of candidate processes, that is, $P_1; P_2; \ldots; P_n$, where P_i denotes the ith process.

The analyzing phase is to explore the control and data flow dependency among the sequential processes $P_1; P_2; \ldots; P_n$. The data flow dependency is as important as the control flow dependency and helps to decide whether data transfer occurs between any two processes. The details are discussed in Section 3.

Our goal is to select those processes which yield the highest speedup if moved to hardware. More precisely, the total execution time is minimized in terms of the limited resources in hardware. The overhead of the required communication between the software and hardware should be included too. The synchronous waiting time will be considered in the performance of the partitioning as well. In fact, we will see that this partitioning is a scheduling problem constrained by precedence relation, synchronous communication and limited resources. We transform the scheduling problem into a reachability one of timed automata(TA) [10] and obtain the optimal result using an optimal reachability algorithm. TA is the finite state automata with clock variables. It is a useful formalism to model real-time systems [18], where the system verification is usually converted to checking of reachability properties, i.e. if a certain state is reachable or not. Automatic model checking tools for timed automata are available, such as UPPAAL [19], KRONOS [7] and HyTEch [13]. We will use the UPPAAL as our modelling tool to conduct some partitioning experiments in Section 7.

When the partitioning process is finished, we get the following form:

$$P_{S_1}; P_{S_2}; \ldots; P_{S_m} \parallel P_{H_1}; P_{H_2}; \ldots; P_{H_l}$$

where P_{S_i} $(1 \leq i \leq m)$ denotes a process allocated in the software component, and P_{H_i} $(1 \leq i \leq n)$ denotes a process allocated in the hardware component.

In the end of partitioning phase, communication commands are added to support data exchange between the software and hardware. To reduce the waiting time, we reorganize the software and hardware components by re-ordering the communication commands. For example, let us consider the following partitioned processes $P \parallel Q$.

$$\begin{array}{ll}
P: & Q: \\
P1(x); & Q1(z); \\
y := f(x); & q := g(z); \\
P2(x); & Q2(z); \\
C!y; & C?q; \\
P3(x); & Q3(z);
\end{array}$$

Suppose process P is implemented in software and Q is implemented in hardware. Where $C!y$ denotes the output and $C?q$ as the input. In the process P, moving the action $C!y$ before $P2(x)$ or after $P3(x)$, and in the process Q moving the action $C!q$ before $Q2(x)$ or after $Q3(x)$ will not effect the result of the program $P||Q$. We assume that the estimate of the execution time of P_1, $P2$ and $P3$ is $2, 2$, and 2 respectively, and the estimate of the time for the execution of Q_1, $Q2$ and $Q3$ is $1, 1$ and 1 respectively. Then moving $C!y$ to the line in between $y := f(x)$ and $P2(x)$ will make the program run faster. We propose a general algorithm which is applicable to more than two parallel processes to improve the communication efficiency.

3 Exploring Dependency Relations between Processes

3.1 Dependency Relations

Let $P_1; P_2; \ldots ; P_n$ be the initial sequential specification produced by the system designer in the profiling stage. In this section we explore the dependency relations between any two processes. This is an important step in analyzing phase. Our intention is to disclose the control and data flow relations of processes in the specification. These relations will be passed to the next step as the input for partitioning using timed automata model. Moreover, through the analysis of control relation among processes, we will find those processes that are independent so that they can be executed in any order on one component or in parallel on two components without any change to the computation result specified by the original specification.

Let for process P_i, $Wr(P_i)$ and $Re(P_i)$, respectively, denote the set of variables modified by P_i and the set of variables read by P_i.

The control flow dependency is represented by the relation $\prec$ over two processes defined as follows.

Definition 1

$$\begin{aligned} P_i \prec P_j =_{def} & \ (Wr(P_i) \cap Re(P_j) \neq \emptyset \vee \\ & Re(P_i) \cap Wr(P_j) \neq \emptyset \vee \\ & Wr(P_i) \cap Wr(P_j) \neq \emptyset) \wedge \\ & i < j \end{aligned}$$

We call P_i as a control predecessor of P_j. If $P_i \prec P_{i+1}$ does hold, then process P_{i+1} can not start before the completion of P_i. Otherwise, P_{i+1} can be activated before P_i leaving the behaviour of the whole program unchanged.

Theorem 1
$P_i;\ P_{i+1}\ =\ P_{i+1};\ P_i\quad \text{if}\quad \neg(P_i \prec P_{i+1})$

Proof. To prove formally this property we follow the convention of Hoare and He [14] that every program can be represented as a *design*. A design has the form $pre \vdash post$, where pre denotes the precondition and $post$ denotes the postcondition. Sequential composition is formally defined as follows [14]:

$\mathbf{P}(v,v');\mathbf{Q}(v,v') \;=_{def}\; \exists m \bullet \mathbf{P}(v,m) \wedge \mathbf{Q}(m,v')$,
Where variable lists v and v' stand for initial and final values respectively, and m is a set of fresh variables which denote the hidden observation.

The following lemma is taken from [14]:

Lemma 1
$(\mathbf{P}_1 \vdash \mathbf{Q}_1);(\mathbf{P}_2 \vdash \mathbf{Q}_2) \;=\; \mathbf{P}_1 \wedge \neg(\mathbf{Q}_1;\; \neg\mathbf{P}_2) \vdash \mathbf{Q}_1;\mathbf{Q}_2$

Because processes P_i and P_{i+1} do not satisfy relation $P_i \prec P_j$, we can easily obtain $Wr(P_i) \cap (Wr(P_{i+1}) \cup Re(P_{i+1})) = \emptyset$ and $Wr(P_{i+1}) \cap (Wr(P_i) \cup Re(P_i)) = \emptyset$. For simplicity, Assume $Wr(P_i) = \{y\},\ Wr(P_{i+1}) = \{z\},\ Re(P_i) \cap Re(P_{i+1}) = \{x\}$, where variables x, y, z stand for list variables respectively. Let $P_i = pre_1 \vdash post_1$, $P_{i+1} = pre_2 \vdash post_2$. We could note $pre_1, post_1, pre_2$ *and* $post_2$ as follows:

$$\begin{aligned}
&Pr_1 = pre_1(x,y)\\
&Pr_2 = pre_2(x,z)\\
&Po_1 = post_1(x,y,y') \wedge x = x' \wedge z = z'\\
&Po_2 = post_2(x,z,z') \wedge x = x' \wedge y = y'
\end{aligned}$$

From the above, we could easily obtain:
$P_i = pre_1 \vdash post_1 = Pr_1 \vdash Po_1$
$P_{i+1} = pre_2 \vdash post_2 = Pr_2 \vdash Po_2$

$$\begin{aligned}
&(1)\ Po_1;\ Po_2 && \{rewrite\ Po_1\ and\ Po_2\}\\
&\quad = post_1(x,y,y') \wedge x = x' \wedge z = z'\ ;\\
&\qquad post_2(x,z,z') \wedge x = x' \wedge y = y' && \{def\ of\ ;\}\\
&\quad = \exists m_1, m_2, m_3 \bullet post_1(x,y,m_2) \wedge\\
&\qquad x = m_1 \wedge z = m_3 \wedge post_2(m_1,m_3,z') \wedge\\
&\qquad m_1 = x' \wedge m_2 = y' && \{predicate\ calculus\}\\
&\quad = \exists m_1, m_2, m_3 \bullet post_1(x,y,m_2) \wedge\\
&\qquad m_1 = x' \wedge m_2 = y' \wedge post_2(m_1,m_3,z') \wedge\\
&\qquad x = m_1 \wedge z = m_3 && \{predicate\ calculus\}\\
&\quad = post_1(x,y,y') \wedge post_2(x,z,z') \wedge x = x'
\end{aligned}$$

$$\begin{aligned}
&(2)\ Po_2;\ Po_1 && \{rewrite\ Po_1\ and\ Po_2\}\\
&\quad = post_2(x,z,z') \wedge x = x' \wedge y = y'\ ;\\
&\qquad post_1(x,y,y') \wedge x = x' \wedge z = z' && \{def\ of\ ;\}\\
&\quad = \exists m_1, m_2, m_3 \bullet post_2(x,z,m_3) \wedge\\
&\qquad x = m_1 \wedge y = m_2 \wedge post_1(m_1,m_2,y') \wedge\\
&\qquad m_1 = x' \wedge m_3 = z' && \{predicate\ calculus\}\\
&\quad = \exists m_1, m_2, m_3 \bullet post_2(x,z,m_3) \wedge\\
&\qquad m_1 = x' \wedge m_3 = z' \wedge post_1(m_1,m_2,y') \wedge\\
&\qquad x = m_1 \wedge y = m_2 && \{predicate\ calculus\}\\
&\quad = post_2(x,z,z') \wedge post_1(x,y,y') \wedge x = x'
\end{aligned}$$

(3) From (1)(2), we establish

$Po_1;\ Po_2 = Po_2;\ Po_1$

In the same way, we can prove the following equation

$Pr_1 \wedge \neg(Po_1; \neg Pr_2) = Pr_2 \wedge \neg(Po_2; \neg Pr_1)$

From Lemma 1, the theorem is proved. $\square$

Let set S_j $(1 \leq j \leq n)$ store all the control predecessors of P_j, and constant max_j be the maximum index of processes in S_j. To uncover the hidden concurrency among the processes, we have the following corollary.

Corollary 1

$$\begin{array}{l} P_1; \ldots; P_{max_j};\ P_{max_j+1}; \ldots; P_j;\ \ \ldots; P_n \\ =\ P_1; \ldots; P_{max_j}; P_j;\ P_{max_j+1}; \ldots;\ \ \ldots; P_n \\ if\ \ max_j < j-1 \end{array}$$

Proof. Apply Theorem 1 $(j - 1 - max_j)$ times. $\square$

This corollary shows each process P_k between P_{max_j} and P_j could be executed in parallel with P_j. If P_k and P_j are allocated in the software and hardware respectively, it should reduce the execution time of the whole program.

To be more concrete on the data flow specified by the initial specification, we introduce the relation $\overset{d}{\prec}$ between processes which is exactly the relation "read-from" in the theory of concurrency control in databases.

Definition 2

$$\begin{array}{rl} P_i \overset{d}{\prec} P_j = i < j \wedge \exists x \bullet (x \in Wr(P_i) \cap Re(P_j) \wedge \\ \forall k \bullet (i < k < j \Rightarrow x \notin Wr(P_k))) \end{array}$$

If processes P_i and P_j satisfy relation $\overset{d}{\prec}$, there is direct data transfer between them in any execution. We call P_i as a data predecessor of P_j. Through this relation, we know the data from which process, a process may need and estimate the communication time between them.

3.2 Algorithms for Exploring Dependency Relations

In this section, we present two algorithms. One is for finding control predecessors of each process, and the other is for finding data predecessors of each process. The two algorithms are intuitive, so we will omit the proof of their correctness here.

Set variables S and T are vectors with n components, and store the control and data predecessors of each process respectively. i.e, the postconditions of the two algorithms are as follows,

$$\forall j \leq n \bullet (S_j = \{P_k \mid P_k \prec P_j\})$$

Table 1. Control and Data Flow Dependency Algorithms

```
Input: $P_1; P_2; \ldots P_n$
Output: Set vector $S$
Method:
for $j := 1$ to $n$
    $S_j := \emptyset$;
end for
for $j := 2$ to $n$
    for $i := 1$ to $j - 1$
        if $(P_i \prec P_j)$
            $S_j := S_j \cup \{P_i\}$;
        endif
    endfor
endfor
```

```
Input : $P_1; P_2; \ldots P_n$
Output : Set vector $T$
Method:
for $i := n$ to 1
    $T_i := \emptyset$;
end for
for $i := n$ to 1
$j := i - 1$;
    while$(Re(P_i) \neq \emptyset \wedge j > 0)$
        $G := Re(P_i) \cap Wr(P_j)$;
        if$(G \neq \emptyset)$
            $T_i := T_i \cup \{P_j\}$;
            $Re(P_i) := Re(P_i) \setminus G$
        end if
        $j := j - 1$;
    end while
end for
```

$$\forall i \leq n \bullet (T_i = \{P_k \mid P_k \overset{d}{\prec} P_i\})$$

Obviously, $S_1 = T_1 = \emptyset$. Table 1 shows the two algorithms.

Although the control dependency algorithm is very simple, the set S_j ($1 \leq j \leq n$) discloses the hidden concurrency in the sequential specification based on the corollary in the last subsection.

The set vector variables S and T will provide us all necessary information on the temporal order between processes which will be the input for modelling the partitioning with timed automata in UPPAAL. For simplicity, let D_i be the set of indexes for the processes in T_i, i.e. $D_i = \{j \mid P_j \in T_i\}$

4 Formal Model Construction

In this section, we transform the hardware/software partitioning into a reachbility problem of timed automata. The timed behaviour of each process is modelled as a timed automaton, and the whole system is composed as a network of timed automata. The timed automata models of all processes are similar except for some guard conditions. After the model is constructed, an optimal reachbility algorithm is applied to obtain an optimal trace in which each process is sequentially recorded whether it is allocated in hardware or software components. As model checker UPPAAL has implemented this algorithm in its latest version, we use the UPPAAL as our modelling tool.

4.1 Behaviour Analysis

Here we list some key elements of the timed automata in our model.

Table 2. Notation

P_i	Process P_i
$Token_i$	The number of control predecessors of P_i
$Tcom_i$	The estimated communication time of P_i
$STexe_i$	The estimated execution time of P_i if implemented in software
$HTexe_i$	The estimated execution time of P_i if implemented in hardware
Rh_i	The estimated hardware resources needed for P_i
St_i	Variable indicating the state of P_i
X_i	Variable recording the number of the terminated control predecessors of P_i
SR	Variable indicating if the processor of software is occupied
HR	Variable indicating if the hardware is occupied
$Hres$	Constant storing the total available hardware resources
CS_i	Software clock for P_i
CH_i	Hardware clock for P_i
CC_i	Communication clock for P_i
T_i	The set of the processes from which P_i reads data from
D_i	The set of the indexes of the processes that P_i reads data from

State variables. Each process has two possible states which indicate whether the process is allocated in hardware or software. We use a global variable St_i to record the state of P_i. St_i is 1 if process P_i is implemented in software, otherwise it is 0.

Precedence constraints. It is obvious that only when all the control predecessors of a process have terminated, the process has the opportunity to be executed either in hardware or software. We use local variable X_i to denote the number of the control predecessors of P_i which have completed their tasks.

Resource constraints. There is only one general processor in our architecture, so no more than one process can be executed on the processor at any time. We introduce a global variable SR to indicate whether the processor is occupied or not. The processor is free if SR is 1, otherwise SR is 0. As far as hardware resources are concerned, the situation is a little complicated. We introduce global variable $Hres$ to record the available resources in hardware. As the processes in hardware are also sequential like in software in our architecture, we introduce a global variable HR to denote whether a process is executed in hardware. If HR is 1, it indicates that no process occupies the hardware. Otherwise HR is 0.

Clock variables. Local clock variables CH_i and CS_i represent the hardware clock and software clock for process P_i respectively. To calculate the communication time between the software and hardware we introduce local clock CC_i for process P_i.

Table 2 lists the variables used in our timed automata model together with their intended meaning. Most of these notations have been explained above.

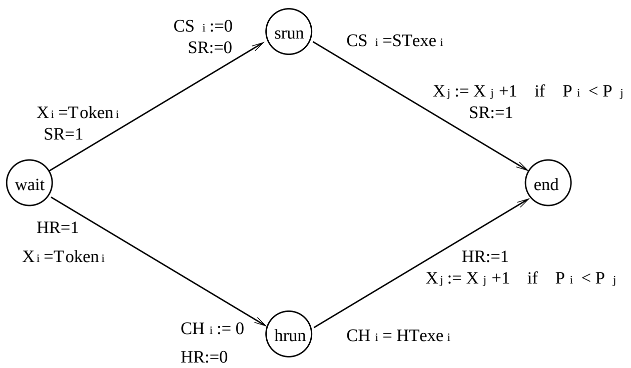

Fig. 2. The Simple Model of Process P_i

4.2 Model Construction

In this section we present two models, one is called simple model and helps to understand the behaviour of the system, and the other is the full model which takes into account all the elements including resource and communication.

Simple Model Figure 2 shows the simple model. It expresses the timed behaviour of process P_i. There are four states in this model. The *wait* state denotes that process P_i is waiting. The states *srun* and *hrun* denote that P_i is allocated in software or in hardware respectively. When P_i finishes its computation task, it will be in the state *end*. Our purpose is to find the fastest system trace in which all processes reach their *end* states.

If the control predecessors of process P_i are all terminated, i.e $X_i = Token_i$ is satisfied, P_i is enabled to be executed in hardware or software. When both of the components are free, it will choose one of them nondeterministically. If both components are occupied by other processes, P_i will still be in state *wait*.

Suppose process P_i is decided to run in software. Once P_i has occupied the software, it sets the global variable SR to 0 to prevent other processes from occupying the processor. It sets the software clock CS_i to 0 as well. The transition between the *srun* state and *end* state can only be taken when the value of the clock CS_i equals $STexe_i$. As soon as the transition is taken, variable X_j will be added one if P_i is the control predecessor of process P_j. At the same time, P_i releases the software processor as well. The situation is similar if P_i is implemented in the hardware.

This simple model shows that every P_i may be implemented in software or in hardware. When all the processes reach its *end* state, the reachability property of the system is satisfied. To obtain the minimal execution time of the whole system, we use a global clock in UPPAAL tool. When the optimal reachability trace is found the global clock will show the minimal execution time.

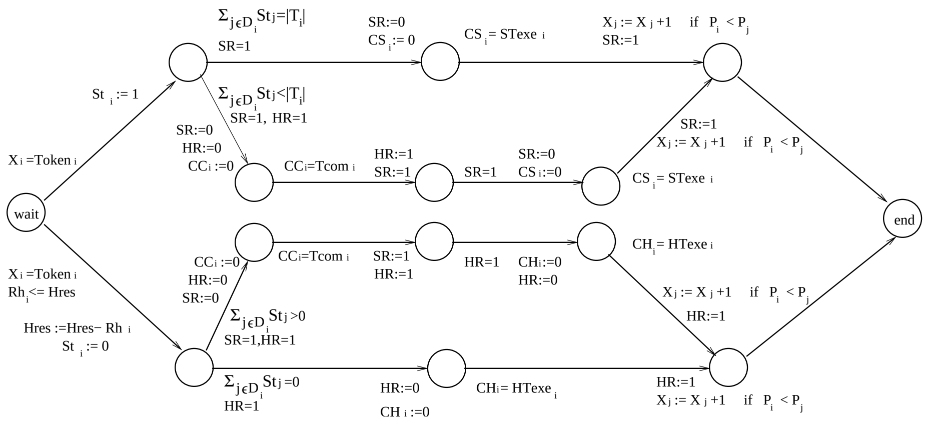

Fig. 3. The Full Model of Process P_i

Full Model Now we present the full model taking into account the communication and resource, etc. The full model is depicted in Figure 3.

In addition to the states the simple model introduced, we should solve two problems which are not involved before. One is how to simulate the resource allocation in the hardware component, and the other is how to simulate the data transfer between the hardware and software.

The first problem is simple. When considering allocation of process P_i in hardware, the automata tests not only variable *HR* but also variable *Hres* to check if the hardware resources are sufficient for P_i. If the resources there are enough, P_i may be put into the hardware. Though there exist reusable resources in hardware, such as *adder* and *comparator*, we do not need to consider them here because the processes are implemented sequentially in hardware in our target architecture. When process P_i terminates, it releases the reusable resources for other processes.

In order to model the communication between the software and hardware, the data dependency among processes has to be considered. When process P_i uses the data which is produced by other processes, there is data transfer between them. If they are all in the same component, the communication time could be ignored. For example, when the processes communicated with each other are all in software, they exchange data via shared memory. Supposing that process P_i is in the software, and at least one process communicating with P_i is allocated in the hardware, the communication between them will take place by means of the bus or other physical facilities. In this case, the overhead of communication between the software and hardware can not be negligible, and should be taken it into account in the model.

Recall that variable St_i is introduced to denote that process P_i is implemented in the hardware or software components. For example, when P_i is allocated in software, St_i is set to 1. P_i then checks the processes that will transfer data to it (i.e. the processes in T_i) are in software or hardware. If at least one of them is in the hardware, the communication should be taken into account. In Figure 3, the condition $\Sigma_{j \in D_i} St_j < |T_i|$ is a guard to express that at least one process that P_i reads data from is in hardware. The situation is the same when P_i is located in hardware and $\sum_{j \in D_i} St_j > 0$.

Next, when the communication really occurs between the software and hardware, it should occupy both the software and hardware components. That is to say, no other processes would be performed until the communication is finished. According to this, the variables SR, HR are set to 0 simultaneously as long as the communication takes place. The clock CC_i is set to 0 when the communication begins. For the communication time T_{comm_i} depends on the states of P_i's data predecessors, there are two methods to estimate the value of T_{comm_i}. One is to adopt T_{comm_i} as the probability average value of all the communication combinations. On the other hand, we can calculate the values of all possible communication in advance, then choose T_{comm_i} as one of them according to the current states of process P_i's data predecessors.

Once a communication action of process P_i finishes, the control of the hardware and software is released immediately. Process P_i will compete hardware or software resources with other ready processes at that point.

It is worthwhile to point out that even if process P_k is one of the data predecessors of P_j, it is not necessary that there will be a non negligible time consuming communication between processes P_j and P_k. Other process P_i may be as a delegate to transfer the data for them. The data will not be modified by process P_i in terms of the data dependency defined before. For example, if both processes P_i and P_j are implemented in the hardware and they have to transfer data to the process P_k which is allocated in the software. Process P_i or P_j will be a delegate to send all the data to P_k. Although more than one process will communicate with process P_k, the communication between the hardware and software occurs only once.

4.3 Discussion

We have showed that the hardware/software partitioning is formulated as a scheduling problem, which is constrained by precedence relation, limited resources, etc. In the partitioning model, we need not only to check all the processes could reach their *end* states, but also to obtain a shortest accumulated delay trace. This is regarded as the optimal reachability problem of model checking in timed automata.

For model checking algorithm, it is necessary to translate the infinite state-spaces of timed automata into finite state presentation. For pure reachability analysis, symbolic states [11] of the form (l, Z) are often used, where l is a location of the timed automata and Z is a convex set of clock valuations called a *zone*. The formal definition of (l, Z) could be found in [11]. Several optimal reachability algorithms have been developed based on this presentation of symbolic state, such as [12, 5, 4].

To generalize the minimum-time reachability, in [5], a general model called *linearly priced timed automata* (LPTA), which extends the model of TA with *prices* on all transitions and locations is introduced to solve the minimum-cost reachability problem. *Uniformly Priced Timed Automata* (UPTA) [4], as a variety of LPTA, adopts a heuristic algorithm which uses some techniques such as branch-and-bound to improve the searching efficiency has been implemented in the latest version of UPPAAL. In Section 6, we will use UPPAAL to do some experiments on some hardware/software partitioning cases.

5 Improving the Communication Efficiency

After the partitioning stage is finished, we obtain two parallel processes running in software and hardware components respectively. The communication is synchronized between them. Moreover, we can improve communication efficiency further by moving the communication commands appropriately.

The idea is that we can find a flexible interval $[ASAP, ALAP]$ for each communication command in which the communication could occur without changing the semantics of the program. This interval denotes the *earliest* and *latest* time when the communication command can execute relatively to the computation time of a program. Then we apply a scheduling algorithm to decide the appropriate place of communication command to reduce the waiting time between processes. Here we propose a general algorithm which is for more than two processes in parallel.

5.1 System Modelling

Let S be a system of m processes $\{P_1, \dots, P_m\}$ running in parallel and synchronized by handshaking. All the processes start at time 0. In our partitioning problem, let m equal 2.

Description of each process P_i

- P_i has a communication trace $\alpha_i = c^i_1 c^i_2 \dots c^i_{n_i}$, $c^i_j \in \Sigma_i$, where Σ_i is the alphabet of the communication actions of P_i.
- P_i needs computation time A_i before it terminates.
- Each c^i_j has a "flexible" interval for the starting time $[a^i_j, b^i_j]$ relatively to the computation time of P_i, $a^i_j \le b^i_j \le A_i$. This means that c^i_j is enabled when the accumulated execution time of P_i has been reached a^i_j time units, and should take place before the accumulated execution time of P_i reaches b^i_j time units. b^i_j and A_i can be infinity, and a^i_j can be 0. To be meaningful, we assume that $a^i_j \le a^i_{j+1}$ and $b^i_j \le b^i_{j+1}$ for $1 \le j < n_i$.
- P_i is either running or waiting when not yet completed. It is waiting iff it is executing a communication action c^i_j for which the co-action has not been executed.

We now formulate the problem precisely. The purpose of formalization here is just to avoid ambiguity and to simplify the long text in the proof when applicable. Any formalism to model the problem must have the capacity to express the "accumulated execution time" for processes. For this reason, we take some idea from Duration Calculus (DC) ([8]) in the formalization.

For each process P_i, we introduce four state variables (which are mappings from $[0, \infty)$ to $\{0, 1\}$) $P_i.running$, $P_i.waiting$, $P_i.completed$ and $P_i.start$ to express the states of P_i. At time t, the state variable $P_i.running$ ($P_i.waiting$, $P_i.completed$ and $P_i.start$) has the value 1 iff P is running (waiting, completed and start, correspondingly) at the time. Process P_i starts at time 0 and terminates when its accumulated execution time has reached A_i. All processes stay in the state "complete" when they terminate.

Systems and Assumptions

- $\alpha_1, \alpha_2, \ldots, \alpha_m$ are assumed to be matched in the sense of handshaking synchronization. Let f be the matching function, i.e. $f(c^i_j, c^k_h) = 1$ iff c^i_j and c^k_h are matched (they are the partners for the same communication).
- Let t^i_j be starting time of c^i_j (according to the unique global clock). Then t^i_j must satisfy the constraint for c^i_j, i.e. $a^i_j \leq \int_0^{t^i_j} P_i.running.\, dt \leq b^i_j$ and $t^i_j \leq t^i_{j+1}$,
- $P_i.waiting(t) = 1$ if and only if there exists c^i_j and c^k_h such that $t^i_j \leq t \leq t^k_h$ and $f(c^i_j, c^k_h) = 1$. (P_i is waiting iff it decides to communicate and its partner is not ready).

To formalize the behaviour of communication actions as mentioned in the items above, we introduce for each c^i_j and c^k_h such that $f(c^i_j, c^k_h) = 1$, a state variable $comm(i, j, k, h)$. $comm(i, j, k, h)(t) = 1$ iff at time t, one of the partner action (either c^i_j or c^k_h) has started and the communication has not completed. Note that $comm(i, j, k, h) = comm(k, h, i, j)$.

An execution of S is a set of intervals $[t^i_j, {t'}^i_j]$ of the starting time and the ending time for communication actions c^i_j. An execution terminates at time t iff t is the termination time of the latest process.

Question: Develop a procedure for the scheduling to produce an execution of S that terminates at the earliest time.

In the following algorithm and example, we assume that the communication takes no time for simplicity. The algorithm is still correct when including the communication time.

5.2 Scheduling Algorithm

Because $\alpha_1, \ldots, \alpha_m$ are matched, we can easily construct a dependency graph G to express the synchronised computation for the system S (a Mazurkiewicz trace [1], [9]) as follows. Each node of the graph represents a synchronised action (c^i_j, c^k_h) with $f((c^i_j, c^k_h)) = 1$ (and labelled by (c^i_j, c^k_h)). There exists a directed edge from $n_1 = (c^i_j, c^k_h)$ to $n_2 = (c^{i'}_{j'}, c^{k'}_{h'})$ iff either $i = i'$ and $j' = j + 1$ or $k = k'$ and $h' = h + 1$. G is used as an additional input for the algorithm.

Algorithm

Input: S, G

Output: Time to start each communication action for each process and time for the synchronous communication actions (represented by each node n of G)

Method:

(1) (Initialisation) Set waiting time vector to zero $W := (0, 0, \ldots, 0)$ (no process is waiting before doing any synchronisation action). Set the last communication time vector V to zero $V = (0, 0, \ldots, 0)$.

(2) Compute the minimal slice C for G which is the set of all nodes of G with no incoming edge. If $C = \emptyset$ halt the algorithm. Otherwise, for each node $n = (c^i_j, c^k_h)$ in C:

(2.1) Compute the global real time intervals for the enabling of c^i_j and c^k_h: $I := [W_i + a^i_j, W_i + b^i_j]$ and $J := [W_k + a^k_h, W_k + b^k_h]$. Let $K := [\max\{V_i, V_k\}, \infty)$ be the interval of possible time for the synchronous communication action represented by the node n.

(2.2) (Select the earliest time t_n that (c^i_j, c^k_h) can take place). Let $B = I \cap J \cap K$.

(a) If $B \neq \emptyset$ then $t_n := \min B$. In this case, $t^i_j := t^k_h := t_n$ (no waiting time), $V_i := V_k := t_n$.

(b) If $B = \emptyset$ then $t_n := \min I \cap K$ if $\max J \cap K < \min I \cap K$, and update the waiting time $W_k := W_k + (\min I \cap K - \max J \cap K)$. In this case, $t^i_j := t_n$, $t^k_h := \max J \cap K$, and $V_i := V_k := t_n$. The case $\max I \cap K < \min J \cap K$ is symmetric.

(3) Remove all the nodes in C and the edges leaving from them from graph G.

(4) Repeat Step 2 until G is empty.

(5) Output t^i_j for each j, i, and t_n (as the scheduled time for the communication actions represented by the node n) for each node $n = (c^i_j, c^k_h)$ of G.

Example 1. Suppose there are 3 processes P_1, P_2 and P_3 to communicate each other. The communication intervals of the precesses are showed in Fig 4. The dependency graph G for S is constructed as well.

The first execution of Step 2 is on the slice $C1 = \{n1\}$, and gives $t1 = 4$, $W = (0, 0, 0)$, $V = (4, 4, 0)$ meaning that until time $t1$ for the finishing of the actions represented by $n1$, no process is waiting, and that at the action represented by $n1$ involves $P1$ and $P2$, and terminate at time 4.

The second execution of Step 2 is on the slice $C2 = \{n2\}$, and gives $t2 = 6$, $W = (0, 0, 0)$, $V = (4, 6, 6)$ meaning that until time $t2$ for the finishing of the actions represented by $n2$, no process is waiting.

The last execution of Step 2 is on the slice $C3 = \{n3\}$, and gives $t3 = 11$, $W = (1, 0, 0)$, $V = (11, 11, 6)$ meaning that until time $t3$ for the finishing of the actions represented by $n3$, P1 has to wait for 1 time unit.

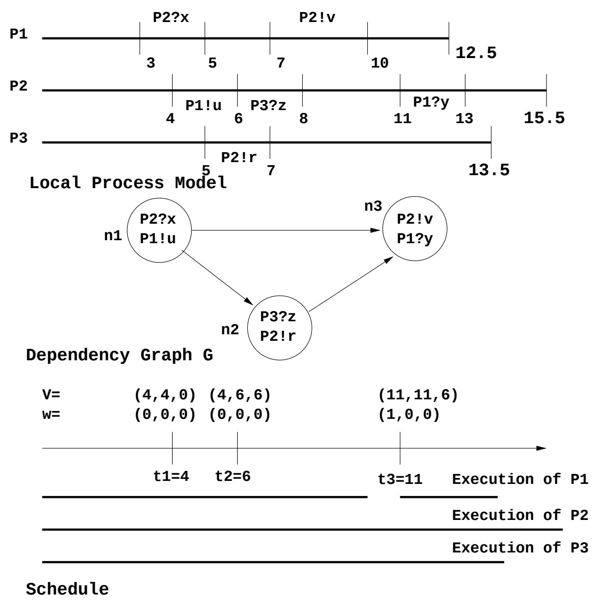

Fig. 4. The Algorithm Execution

Theorem 2. *t_n is the earliest time that the communication actions represented by n can take place. Hence, the algorithm gives a correct answer to the problem mentioned above.*

Proof. See [22]. □

6 Experiments in Uppaal

We have used the technique in the previous section to find optimal solution for some hardware/software partitioning cases. In this section we present some of our experiments in solving them with the model checker UPPAAL version 3.3.32, running in Linux machine with 256Mb memory.

After we have modelled a hardware/software partitioning problem as a network of timed automata with n processes, we input the model to the UPPAAL model checker. Then we asked the UPPAAL to verify:

```
E <> P1.end and P2.end and ... Pn.end
```

This expression in UPPAAL specification language specifies that there exists a trace of the automata network in which eventually all n processes reach their *end* states.

To let UPPAAL find out the optimal solution to our problem, we choose the breadth-first model checking algorithm (UPPAAL offers various different al-

Table 3. Experimental results of partitioning using UPPAAL

Case studies	Num. of total processes	Required gates in hardware (K)	Num. of processes in hardware after partitioning	Running time in UPPAAL(Seconds)
Huffman decoder	10	44.5	5	121
rmatrix multiplier	20	97.7	3	289
packdata	30	221.9	5	503

gorithms) and the option "fastest trace" provided by UPPAAL. A global clock variable is declared to store the execution time. When the reachability property is satisfied, the fastest trace which records the partitioning scheme will be found, and the global clock will record the minimal execution time of all the processes. This trace, after having been added with the necessary communication statements, can be passed into the software compiler and hardware compiler to for implementing.

In the experiments, we use a Occam-like language as our specification language, and use the hardware compiler technique [6] to estimate the required resources in hardware of each process. For simplicity, as resources we list here only the estimated required gates of each problem.

The experimental results for the three case studies are shown in Table 3. We assume there are 15,000 gates in hardware resources. The second column of Table 3 shows the resources required if the program is entirely fulfilled in hardware for each case respectively.

The first one is Huffman decoder algorithm. The second is a matrix multiplier algorithm, and the last example is a pack data algorithm in network.

7 Summary

This paper presents a noval approach to hardware/software partitioning supporting the abstract architecture in which the communication takes place synchronously. After the designer decides the process granularity of the initial specification, the partitioning process could be carried out automatically. We explore the relations among processes to find the hidden concurrency and data dependency in the initial specification. These relations are as the input of timed automata to ensure the behaviours of processes are modelled correctly. Once the formal partitioning model is constructed with timed automata, the optimal result can be obtained by means of an optimal reachability algorithm. To further improve the synchronous communication efficiency between hardware and software components, a scheduling algorithm is introduced to adjust communication commands after partitioning. The experiments in model checker UPPAAL clearly demonstrated the feasibility and advantage of our proposed approach.

Acknowledgement We are grateful to Liu Zhiming, Jin Naiyong for their helpful discussions and suggestions for the improvement of the paper.

References

1. I. J. Aalbersberg, G . Rozenberg. Theory of Traces. *Theoretical Computer Science*, Vol. 60, pp1-82. 1988.
2. S. Agrawal and R. Gupta. Dataflow-Assisted Behavioral Partitioning for Embedded Systems. *Proc. Design Automation Conf ACM, N.Y.* pp709-712, 1997.
3. E. Barros, W. Rosenstiel, X. Xiong. A Method for Partitioning UNITY Language in Hardware and Software. In *Proc. EURODAC*, September, pp220-225, 1994.
4. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in Uppaal. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pp174-188, 2001.
5. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proceedings of the 4th International Workwhop on Hybrid Systems: Computation and Control (HSCC'01)*. LNCS 2034, pp147-161, 2001.
6. J. Bowen and He Jifeng. An approach to the specification and verification of a hardware compilation scheme. *journal of Supercomputing*, 19(1):pp23-29, 2001.
7. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A modelchecking tool for real-time systems," CAV'98, LNCS 1427, pp546-550,1998.
8. Dang Van Hung. Real-time Systems Development with Duration Calculus: an Overview. Technical Report 255, UNU/IIST, P.O. Box 3058, Macau, June 2002.
9. V. Diekert and G. Rozenberg, editors. Book of Traces. World Scientific, Singapore, 1995.
10. R. Dill and D. L. Dill. A Theory for Timed Automata. In*Theoretical Computer Science*, Vol.125, pp183-235, 1994.
11. D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Proc. of Automatic Verification Methods for Finite State Systems,* LNCS 407, pp197-212, 1989.
12. A. Fehnker. Bounding and heuristics in forward reachability algorithms. Technical Report CSI-R0002, Computing Science Institute Nijmegen, 2000.
13. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybird Systems. In *Proc. of the 9th Int. Conf. on Computer Aided Verication (Orna Grumberg, ed.)*, LNCS1254, pp460-463, 1997.
14. C. A. R. Hoare and He Jifeng. Unifying Theories of Programming. Prenticel Hall, 1998.
15. T. Hune, K. G. Larsen, and P. Pttersson. Guided Synthesis of Control Programs Using UPPAAL. *Proc. of Workshop on verification and Control of Hybrid Systems III*, ppE15-E22, 2000.
16. INMOS Ltd. The Occam 2 Programming Manual. Prentice-Hall, 1988.
17. J. Iyoda, A. Sampaio, and L. Silva. ParTS: A Partitioning Transformation System. In *World Congress on Formal Methods 1999 (WCFM 99)* , pp1400-1419, 1999.
18. K. G. Larsen, P. Pettersson and Wang Yi. Model-Checking for Real-Time Systems. In *Proceedings of the 10th International Conference on Fundamentals of Computation Theory*, LNCS 965, pp62-88, 1995.
19. K. G. Larsen, P. Pettersson and Wang Yi. UPPAAL in a Nutshell. *Int.Journal of Software Tools for Technology Transfer 1*,1-2(Oct), pp134-152, 1997.
20. R. Nieman and P. Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems, special Issue : Partitioning Methods for Embedded Systems.* Vol. 2, No. 2, pp165-193, Kluwer Academic Publishers, March 1997.

21. Z. Peng, K. Kuchcinski. An Algorithm for Partitioning of Application Specific System. *IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC)*, pp316-321, 1993.
22. Pu Geguang, Wang Yi, Dang Van Hung, and He Jifeng. An Optimal Approach to Hardware/software Partitioning for Synchronous Model. Technical Report 286, UNU/IIST, P.O. Box 3058, Macau, September, 2003.
23. Qin Shengchao and He Jifeng. An Algebraic Approach to Hardware/software Partitioning. *Proc. of the 7th IEEE International Conference on Electronics, Circuits and Systems*, (ICECS 2000), pp273-276, 2000.
24. G. Quan, X. Hu and G. W. Greenwood. Preference-driven hierarchical hardware/software partitioning. In *Internatitional conference on Computer Design(IEEE)*, pp652-657, 1999.
25. J. Staunstrup and W. Wolf, editors. *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.
26. M. Weinhardt. Ingeger Programming for Partitioning in Software Oriented Codesign. *Lecture Notes of Computer Science 975*, pp227-234, 1995.
27. W. Wolf. Hardware-Software Co-Design of Embedded System. *Proc. of the IEEE*, Vol.82, No.7, pp967-989, 1994.