为关系操作 $y:=f(x_1,\ldots x_n)$.由于 y 以及 x_1 到 x_n 的信号是同步的,当 y 的时钟为真时,x_1 到 x_n 一定存在.

对于延迟操作,由于信号 y 和 x 之间没有数据依赖关系,因此不生成对应的 NDG 节点.但是记忆信号 y 可以作为其他节点的右值.

对于欠采样操作,y 对应于节点的 L;右值 x 与 z 对应于节点的 R;B 中的 G 为信号 y 的时钟 C_y,动作为赋值操作 $y:=x$.可以看出,若要动作 $y:=x$ 得到执行,需要保证 x 和 z 的值已经得到,并且 C_y 的值为真.

对于合并操作,将生成两个节点,分别对应于 y 对 x 及 z 值的选择.当 x 的时钟 C_x 为真时,y 将被赋值为 x;当 x 的时钟为假并且 z 的时钟为真时,y 将被赋值为 z.由于两个节点执行动作的守卫条件是互斥的,因此不需要考虑共享变量更新的问题.

对于输入信号的 NDG,其中,由于输入信号从环境中取值,因此 R 为空.当 L 值信号时钟为真时,做读取操作.

EDG 实质上是一种包含 SIGNAL 程序全局数据依赖关系的有向无环图,这里给出方程依赖图 EDG 的定义.

定义 2. EDG 为一个二元组$\langle SNDG,\rightarrow\rangle$,其中,$SNDG$ 为节点 NDG 的集合;$\rightarrow$为节点之间的依赖关系:若节点 v_1,v_2 满足二元关系 $v_1\rightarrow v_2$ 当且仅当 $v_1.L\in v_2.R$.

通过分析数据流方程集合可以得到 $SNDG$,其中,每个节点中守卫条件时钟由对应的时钟等价类表示.EDG 则通过分析 $SNDG$ 之间左右值的数据关系得到.需要说明的是,对于方程右值集合中出现记忆信号(即延迟操作左值信号)的情况,由于延迟操作特殊的语义,可以保证当节点中的其他数据依赖以及守卫条件 G 得到满足时,从记忆信号得到的值也是正确的.

2.3 基于 EDG 的并行任务划分方法

通过分析 EDG 获得方程间的数据依赖关系,并将其划分为可以并行执行以及必须串行执行的部分.首先,将通过拓扑排序的方法对 EDG 进行并行任务划分;之后,将时钟信息加入到并行任务中.

2.3.1 基于拓扑排序的 EDG 划分

在 EDG 的定义中,二元关系"$\rightarrow$"定义了节点间的依赖关系,即对于两个节点 v_1、v_2,如果 v_1 对应信号定义方程 eq_1 的左值信号出现在 v_2 对应信号定义方程 eq_2 的右侧,则说明 eq_2 依赖 eq_1.由于数据依赖关系满足严格偏序关系,可以定义其传递闭包,记作"$\searrow$".若程序的两个数据流方程 eq_1 和 eq_2 之间不存在此关系,这两个方程可以并行执行.为了从 SIGNAL 程序生成并行仿真代码,需要 EDG 作进一步划分.在此,我们给出并行任务(parallel task)的定义.

定义 3. 并行任务为一个二元组$\langle T,\nearrow\rangle$,其中,

- T 是对 EDG 中节点集合的划分,即(1) 对于任意 $t\in T,t\subseteq EDG.SNDG$;(2) 对于任意两个节点 t_1、$t_2\in T,t_1\cap t_2=\varnothing$;(3) T 中所有元素的并集为 EDG 的节点集合,我们称元素 t 为任务节点(task node).
- $\nearrow$是定义在 T 上的二元关系,称为优先关系(precedence relation).

目标平台的不同会导致并行任务划分方法对最终程序的运行性能产生很大的影响.本文采取拓扑排序的方法 EDG 进行划分.其基础是定义划分 T 和优先关系的性质.

- 对于任意 $t\in T,t$ 是定义在$\searrow$上的反链(anti-chain),即任意两个节点 n_1、$n_2\in t,n_1\searrow n_2$ 以及 $n_2\searrow n_1$ 恒为假.
- 对于任意 $t_1,t_2\in T,t_1\nearrow t_2$ 当且仅当 t_2 中至少存在一个节点 n 使得 t_1 中存在一个节点 m,且有 $m\searrow n$.

图 2 给出了 EDG 划分算法.算法输入为 EDG,输出为表示并行任务的有序列表 TaskList.首先将 EDG 中的所有节点加入到 NodeSet 集合中(第 3 行),之后进入迭代过程(第 4 行).第 5 行~第 10 行描述了拓扑排序的过程:首先找到 EDG 中没有入度的节点(第 5 行),这里的入度是针对数据依赖关系"$\rightarrow$"而言的.获得的无入度的节点集合 setNoDegree 将组成 Task 中的一个元素,记作 tempTask(第 6 行).将 tempTask 加入到 TaskList 后(第 7 行),需要将 setNoDegree 中所有节点以及以其为出度的二元关系从 NodeSet 集合中移除(第 9 行及第 8 行).迭代的结束条件是 NodeSet 为空,即图中所有节点都已被划分到 TaskList 中.

对于任意 EDG 图,经拓扑排序得到的序列 TaskList=$\{t_{n1},t_{n2},\ldots,t_{nk}\}$,满足以下性质.

- 任意 $t\in T$ 是定义在$\searrow$上的反链.
- 对于任意整数 $x,y\in\{n_1,\ldots,n_k\}$,t_x 和 t_y 满足关系 $t_x\nearrow t_y$ 或 $t_y\nearrow t_x$.

- 对于任意两个整数 $n_x,n_y,x,y\in\{1,\ldots,k\}$,若 $x<y$,对于任意节点 $n\in t_{ny}$,不存在节点 $m\in t_{nx}$,使得 $n\searrow m$ 成立.

得到的任务序列 TaskList 是并行任务中的一种特殊情况,优先关系↗为任务节点间的全序关系.这里,进一步定义 TaskList 序列间的相邻关系:对于任务序列 TaskList=$\{t_{n1},t_{n2},\ldots,t_{nk}\}$以及 $x,y\in\{n_1,\ldots,n_k\}$,若 $y=x+1$,则称任务节点 t_x 左相邻于任务节点 t_y,记作 $t_x\Rightarrow t_y$.

以上算法适用于 EDG 中没有出现循环依赖的情况.若存在信号 s 满足 $s\searrow s$,则无法对 EDG 进行划分.需要说明的是,多时钟操作符中不同时钟下的数据依赖可以不同.例如,对于合并操作 $c:=d$ default e,在 d 的时钟为真时,c 依赖于 d;而在 d 的时钟为假而 e 的时钟为真时,c 依赖于 e.因此,对于数据依赖链 $n_1\xrightarrow{c_1}n_2\xrightarrow{c_2}\ldots\xrightarrow{c_{k-1}}n_1$,若有 $c_1\wedge c_2\wedge\ldots\wedge c_{k-1}$ 的值恒为 0,则循环依赖条件永远不会成立,被称为伪循环依赖,本文中则假设程序没有循环依赖或伪循环依赖.

```
1:Input: EDG
2:Output:TaskList%有序表,先后顺序表示了线性关系
3:NodeSet←EDG.Nodes%初始值 NodeSet 包含了 EDG 中的所有节点
4:while NodeSet≠∅
5:      setNoDegree←findNodeWithNoIndegree(NodeSet)%找到 NodeSet 中没有入度的节点
6:       tempTask←createT(setNoDegree)%建立 Task 中的元素 t
7:       addTask(TaskList,tempTask)%将新建元素加入到 TaskList 的末尾
8:       removeRelation(NodeSet,setNoDegree)%删除与 setNoDegreee 集合中节点相关的二元关系
9:       removeNode(NodeSet,setNoDegree)%从 NodeSet 删除 setNoDegree 中的所有节点
10:endwhile
11:return TaskList
```

Fig.2 Partition of EDG based on topological sorting

图 2 基于拓扑排序的 EDG 划分算法

2.3.2 时钟信息与并行任务的合并

通过对 EDG 进行任务划分,可以得到任务序列 TaskList,其元素将根据序列顺序执行,而元素内部的节点由于不存在数据依赖关系可并行执行.然而,此结构中缺少对时钟如何进行计算的信息,因此需要将规约范式方程集合 RNFS 中的时钟定义方程放入到任务序列中正确的位置.在此,首先给出需满足的性质.

- 所有信号定义必须在其激活时钟被计算后执行.
- 所有时钟定义必须在方程右值时钟或取值变量被计算后执行.

为了满足上述性质,本文通过遍历 TaskList,在其二元相邻关系“⇒”中加入时钟定义方程集合,记作 $t_1\overset{\text{ClockDef}}{\Rightarrow}t_2$. 其中,ClockDef 为任务 t_1 和 t_2 之间的时钟定义方程集合.表示的含义是任务 t_2 执行前必须首先执行 ClockDef 中的方程;而 ClockDef 中的方程可能将依赖 t_1 中的计算结果.

由 endochrony 性质可知,根时钟的值恒为真,因此,从 TaskList 的第 2 个元素开始进行有序遍历.首先,获取当前 task 中的信号定义;之后从 RNFS 找出之前没有被加入到二元关系中并依赖这些信号定义方程的 RNF 集合 ClockDef;最后将 ClockDef 加入到二元关系 task.pre⇒task 之间,形成 task.pre $\overset{\text{ClockDef}}{\Rightarrow}$ task.

对于一个任务序列 $t_1\Rightarrow t_2\Rightarrow t_3\ldots\Rightarrow t_n$,我们可以得到一个序列 $t_1\overset{\text{ClockDef1}}{\Rightarrow}t_2\overset{\text{ClockDef2}}{\Rightarrow}t_3\overset{\text{ClockDef3}}{\Rightarrow}\ldots\overset{\text{ClockDefn-1}}{\Rightarrow}t_n$ 满足以下性质.

- 对于任意二元关系 $t_k\overset{\text{ClockDefk}}{\Rightarrow}t_{k+1}$,ClockDefk 中的方程计算不依赖于节点 $t_{k+1},\ldots,t_n$.
- 任意 t_k 中信号定义方程需要的时钟 $C_1,\ldots,C_m$ 的值一定已经在时钟定义方程 ClockDef1,…,ClockDef$k$–1 中被计算出来.

以上性质将保证序列保持 SIGNAL 程序的数据依赖关系以及时钟关系.

2.4 并行任务到 OpenMP 并行结构的映射

OpenMP 是一种多平台的共享内存的并行编程技术,所支持的编程语言包括 C、C++以及 FORTRAN 等.它提供了包括编译制导、应用编程接口以及环境变量等多种机制,支持用户在高层次对并行算法进行描述.前文已将 SIGNAL 程序转换为并行任务序列,并且执行动作均为简单数学、逻辑计算或赋值.因此,本文选择编译制导语法 parallel sections 作为映射的对象,其语法如下所示.

```
#pragma omp sections [clause[[,] clause] ...]
{
[#pragma omp section]
structured-block
[#pragma omp section ]
structured-block
...
}
```

制导语句#pragma omp sections 所包含的块内为一个并行域.而由制导语句#pragma omp section 包含的代码块 structured-block 之间是并行执行的,代码块内部则是串行执行.根据并行任务序列的语义,可以得到从任务序列元素到 sections 语法的映射规则,见表 5.

Table 5 Mapping from TaskList to sections

表 5 任务序列元素到 sections 的映射

任务序列元素	sections 语法
任务节点	#pragma omp sections{ }

信号定义方程 eq	#pragma omp section{
	if(eq.Class==true){
	eq.A
	}}

任务序列 $t_1 \Rightarrow t_2 \Rightarrow t_3 \ldots \Rightarrow t_k$	#pragma omp sections{
	t1 %t1 对应的 sections 代码块
	}
	…
	#pragma omp sections{
	tk %tk 对应的 sections 代码块
	}

任务序列中的每个任务将被映射为一个由制导指令#pragma omp sections 所包围的代码块.而任务中每个节点的信号方程将被映射为一个由制导指令#pragma omp section 所包围的代码块.根据 sections 的语义,所有处于一个任务内的信号定义方程将可以并行执行.表中第 3 行表示任务序列的顺序执行关系将映射为 sections 代码块间的顺序执行关系.此外,在 OpenMP 编译中,并行任务需要编译器开启多个线程,而多线程的操作需要消耗时间及资源.因此若一个任务中只含有一个节点,则不再生成 sections 以及 section 代码块,而直接生成对应的动作,以提高程序的效率.

除了任务以及任务节点以外,在映射中还需要处理对时钟定义方程的映射.时钟定义方程本质上也是赋值运算,因此可以采用相似的映射方法.为了增加程序的并行度,对于任务序列中的二元关系 $t_1 \overset{\text{ClockDef1}}{\Rightarrow} t_2$,首先对时钟定义方程集合 ClockDef1 进行处理.

- 若 ClockDef1 中只含有一个方程,则直接生成对应的赋值动作.
- 若 ClockDef1 有多个方程,并且方程间不存在数据依赖关系,则将此方程集合也看作一个任务,采用相同的方法生成 sections 以及 section 代码块.
- 若多个方程间存在数据依赖关系,则根据数据依赖关系生成串行执行序列.

通过以上映射方法,可以得到满足 SIGNAL 程序数据及时钟依赖关系并且具有一定并行度的 OpenMP+C 仿真代码.在对目标平台的特点(如处理器核数)以及 OpenMP 编译器实现作进一步了解后,可以进一步对映射进行优化,例如根据 section 代码块的个数设定生成线程数以及 section 间代码的合并等等.

本文采用 SIGNAL 编译器 Polychrony 所使用的迭代模式对 SIGNAL 程序的行为进行仿真:无限循环模拟程序与环境的交互过程,每次循环代表一个逻辑瞬间中的执行过程.根据前文提出的方法得到的 OpenMP 并行结构将作为迭代的核心部分,而对记忆信号值的更新则被放在每次迭代结束之后进行.

3 实例分析

本节将以一个典型的 SIGNAL 程序为例介绍生成 OpenMP 并行仿真代码的过程.首先,将通过分析数据流

方程获取时钟信息;之后通过数据依赖分析生成 EDG 并进行任务划分;最终将并行任务序列映射到 OpenMP 并行结构中.

3.1 示例 SIGNAL 程序

本文选择同步语言中经典示例程序 ABRO[25]作为转换实例 ABRO 自动机表示,如图 3 所示.ABRO 有 3 个输入信号,A、B、R 以及一个输出信号 O.在初始状态,ABRO 等待输入信号,当 A、B 信号按照任意次序读入后,ABRO 输出信号 O.信号 R 用于重置 ABRO 的状态.根据 ABRO 的功能特点,文献[16]给出了相应的 SIGNAL 程序的 SIGNAL 程序,本文给出的修改版本如图 4 所示.

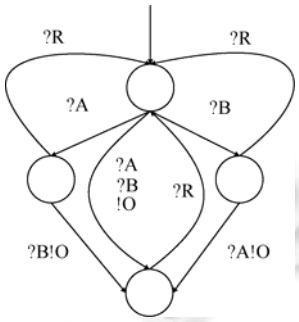

Fig. 3 State machine of ABRO

图 3 ABRO 的状态机表示

```
1:process ABRO=
2:(? boolean A, B, R;! boolean O;)
3:(|A^=B^=R
4:|A^=A_received
5:|A_received^=B_received^=after_R_until_O
6:|nR:=not R
7:|RT:=nR when R
8:|A_received:=RT default AR
9:|AT:=A when A
10:|AR:=AT default Adelay
11:|Adelay:=A_received $ init false
12:|BT:=B when B
13:|B_received:=RT default BR
14:|BR:=BT default Bdelay
15:|Bdelay:=B_received $ init false
16:|nO:=not O
17:|from_R_before_O:=nO default RR
18:|RR:=Re default after_R_until_O
19:|Re:=R when R
20:|after_R_until_O:=from_R_before_O $ init true
21:|O:=true when ABR
22:|ABR:=A_received when Arr
23:|Arr:=B_received when after_R_until_O|)
24:where boolean nR, nO, A_received, B_received,
from_R_before_O ,Adelay,Bdelay,AR,BR,RR,ABR,Arr,after_R_until_O,AT,BT,RT,Re; end;
```

Fig.4 SIGNAL program of ABRO

图 4 ABRO 的 SIGNAL 程序

3.2 时钟信息获取

通过对时钟方程进行解析后得到时钟等价类.图 5 展示了对时钟变量进行等价类划分的结果.等价类 *C*_2 为程序的根时钟.图 6 展示了对应的规约范式方程.可以看到,只有时钟 *C*_2 出现在方程的右侧,程序满足 endochrony 属性并且 *C*_2 为程序的根时钟.

C_2={^R,^A_received,^B_received,^after_R_until_O,^A,^B,^R,^nR,
^RR,^Adelay,^Bdelay,^from_R_before_O,^AR,^BR}
C_6={^RT,^R_true,^Re}
C_75={^A_received_default,^B_received_default,^RR_default}
C_13={^A_true,^AT}
C_26={^B_true,^BT}
C_77={^AR_default}
C_79={^BR_default}
C_84={^ABR,Arr_true}
C_92={^from_R_before_O_default}
C_82={^after_R_until_O_true,^Arr}
C_86={^ABR_true,^nO,^O}

Fig.5 Partition of clock equivalence classes for ABRO

图 5 对 ABRO 时钟等价类的划分

C_6=C_2∧R
C_75=C_2 \ C_6
C_13=C_2∧A
C_77=C_2 \ C_13
C_26=C_2∧B
C_79=C_2 \ C_26
C_82=C_2∧after_R_until_O
C_84=C_82∧Arr
C_86=C_84∧ABR
C_92=C_2 \ C_86

Fig.6 RNFS of ABRO

图 6 ABRO 的规约范式方程集合

3.3 EDG 的生成及并行任务划分

通过分析 ABRO 的数据流方程及数据依赖关系可以得到 EDG,如图 7 所示.需要说明的是,尽管图中 EDG 没有加入时钟信息,但在分析数据依赖关系时考虑了时钟关系,例如对于方程 ABR:=A_received when Arr,信号定义方程 ABR=A_received 同时依赖于 A_received 的值与 Arr 的值.此外,对于合并操作,这里将两个互斥的动作放在一个节点中,在一次迭代中,这两个动作将不会同时执行.得到的 EDG 将通过拓扑排序生成并行任务序列,并加入时钟信息,如图 8 所示.图中的实线箭头代表并行任务执行的线性关系,而同一个任务中的多个方程(实线方框中的椭圆方程)可以并行执行.

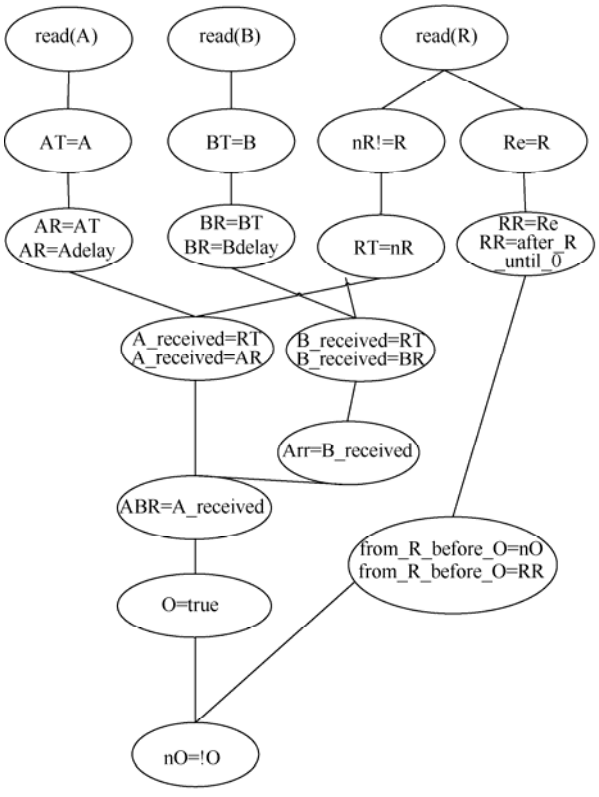

Fig.7 EDG of ABRO

图 7 ABRO 的 EDG 表示

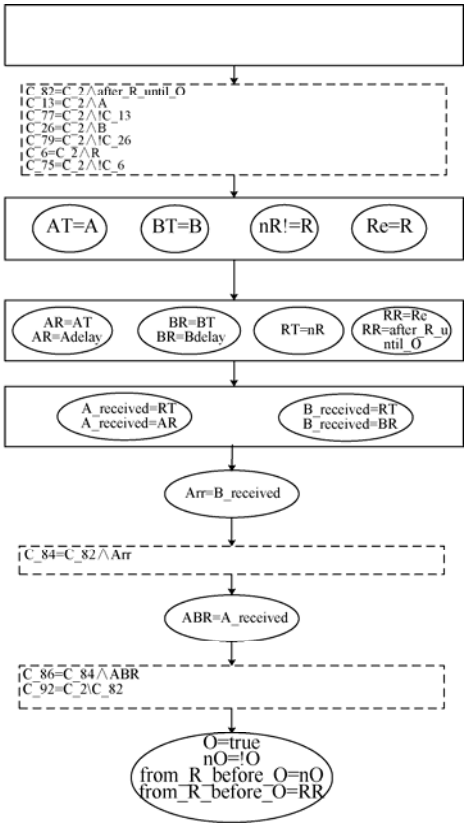

Fig.8 TaskList of ABRO

图 8 ABRO 的任务序列

3.4 OpenMP 并行仿真代码生成

图 9 给出了生成的 OpenMP 并行代码片段.可以看到,对 3 个输入信号 A、B、R 的读取可以同时进行,可生成的代码与其结构有一一对应关系.

```
1:int core(){
2:      int mark=0;
3:      #pragma omp parallel sections
4:      {
5:              #pragma omp section
6:              {
7:                      if(read_A()==EOF)
8:                              mark=1;
9:              }
10:             #pragma omp section
11:             {
12:                     if(read_B()==EOF)
13:                             mark=1;
14:             }
15:             #pragma omp section
16:             {
17:                     if(read_R()==EOF)
18:                             mark=1;
19:             }
20:     }
…
```

Fig.9 Fragment of OpenMP code

图 9 OpenMP 代码片段

3.5 代码的执行测试

我们基于 Eclipse 平台开发了 SIGNAL 代码生成工具.工具采用右键插件菜单的形式.通过在 Eclipse 下 Package Explorer 视图中在对应 SIGNAL 程序源文件位置点击右键选择进行代码生成,得到并行仿真代码.生成的仿真代码在 VS 2010 环境中进行测试,如图 10 所示.程序通过读取文本文件获得输入信号,并在输出信号得到值后向对应的文本文件进行输出.输入信号的文件中为一个序列,每一次迭代时,如果当前输入信号的时钟为真,则从序列中读取一个值.在 ABRO 程序示例中,输入信号有 A、B、R,输出信号为 O.对应的文本文件分别为 A.txt、B.txt、R.txt 以及 O.txt,3 个输入信号文件的信号值如图 11 所示.

表 6 给出了 ABRO 程序一次可能的执行结果.例如,对于逻辑瞬间 t_1,信号 A、B 同时为真而 R 为假,则输出信号 O 为真;而在逻辑瞬间 t_2,尽管 A、B 同时为真,但由于此时状态没有被重置,因此,此时 O 的值不存在;在逻辑瞬间 t_3,读入 R 的值为真,状态被重置(但此时,A、B 的值仍没有被读入,因此 O 的值不存在),因此在逻辑瞬间 t_4,A、B 同时读入后,O 取值为真.需要说明的是,由于程序仅在 O 取值为真时进行输出,因此不能输出 O 不存在时的值,即⊥.

Fig.10 Generated code in VS 2010 perspective

图 10 VS 2010 视图中的生成代码

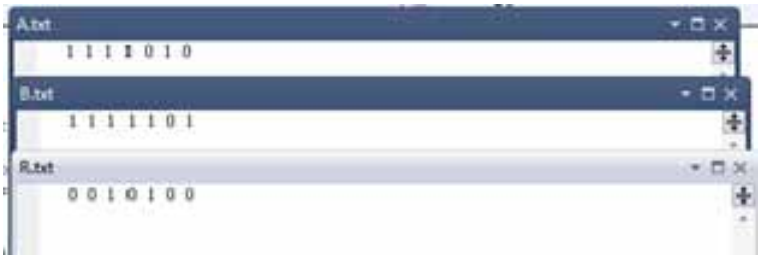

Fig.11 Example value of input signals

图 11 示例输入信号值

Table 6 A possible execution trace of ABRO

表 6 ABRO 的一次执行结果

信号	t_1	t_2	t_3	t_4	t_5	t_6	t_7
A	1	1	1	1	0	1	0
B	1	1	1	1	1	0	1
R	0	0	1	0	1	0	0
O	1	⊥	⊥	1	⊥	⊥	1

经过仿真得到 O 的输出如图 12 所示,其中的 3 个 1 与表 6 中 O 信号的输出相对应,可以看出生成的并行代码有效地对 ABRO 程序的功能行为进行了模拟执行.

Fig.12 Result of the execution

图 12 执行结果

4 相关工作介绍

针对 SIGNAL 的仿真代码生成,特别是多线程及分布式代码生成技术,有一些相关研究.但对跨平台执行的代码生产研究较少.

文献[26]介绍了面向 SIGNAL 的多线程代码生成技术,根据采取静态或动态调度方法可分为两种.静态调度方法中,编译器将根据调度图生成多个 cluster 代码块,并有一个主计算 cluster 用于进行迭代操作.每个 cluster 有自己的单根时钟树,读取输入后进入运行.在动态调度的分布式代码生成中,SIGNAL 程序中的每条可以并发执行的方程都将对应生成一个微线程(micro-thread).微线程通过 wait 等待输入信号,并在输出时发出 notify 信号.这样互不相关的过程可以并行执行.然而,生成的多线程仿真代码使用了特定的线程库,因此不能够跨平台执行.

在分布式或多核系统上,由于组件间的通信延迟等原因,导致组件间的信号不能满足同步关系,被称为全局异步局部同步(globally asynchronous locally synchronous,简称 GALS)系统.而此类系统不满足 endochrony 属性.为此,文献[27]提出了 weak endochrony 理论.就像名字所暗示的那样,weak endochrony 对于对象 SIGNAL 程序生成确定性代码所要求的属性更低:程序中可以存在多个根时钟,如果信号间满足 full-diamond 条件即可以生成确定性多线程代码.然而,检测程序是否满足 weak endochrony 属性是十分困难的.文献[28]提出了一种检测 weak endochrony 性质的方法,给出了从 SIGNAL 程序中找出可以构造出程序可能出现反应并且不会产生冲突的最小反应集合.如果此集合具有唯一性,则程序具有 weak endochrony 性质.但此种方法的缺点是需要首先将 SIGNAL 程序进行变换生成其无状态抽象,使得一些原本具有 weak endochrony 性质的程序丢失信息并导致其无状态抽象不满足该性质.文献[21]提出了基于有界模型检测以及基于 isochrony[29]属性的方法检查 weak

endochrony.然而,这两种方法也不能适用于所有 weak endochrony 程序.文献[30]提出了基于同步数据流依赖图(SFDG)的多线程代码生成方法,然而文献[30]中仍然采用文献[26]中使用的生成策略,并且没有给出 weak endochrony 属性的验证方法,也没有对方法的正确性进行验证.与前述的基于 weak endochrony 的多线程代码生成方法相比,尽管本文提出的方法目前主要支持 endochrony 属性的 SIGNAL 程序,但是通过 EDG 进行描述并将其进行并行任务划分也可以获得较好的并行度.

此外,文献[31]提出了一种从同步守护动作(synchronous guarded actions,一种同步语言中间表达形式)生成 OpenMP 并行代码的方法.同步守护动作首先被转换为依赖图(dependency graph),之后将从此依赖图中获取可同步执行的部分并最终生成 OpenMP 结构.然而,由于 SIGNAL 与同步守护动作在语法和语义,尤其是在时钟以及反应动作的语义方面差异很大,因此本文提出的方法与文献[31]有很大不同,尤其是在时钟分析方面,同步守护动作采用单时钟模型,不需要考虑时钟信息的获取,而本文则针对 SIGNAL 的多时钟模型,提出了获取时钟信息并将其加入到并行任务序列的方法.此外,在中间数据结构的设计上,与文献[31]提出的采用二分图的表示方法相比,本文采用的 EDG 可以更直观地描述程序全局的数据依赖关系.

与前述的研究相比,本文提出的方法主要有以下特点.

- 采用方程依赖图 EDG 可以更加全面地描述 SIGNAL 程序中方程间的数据依赖;基于拓扑排序方法,将可以并行执行的部分提取出来,仿真代码可以获得较好的并行度;
- 提出了一种面向 SIGNAL 同步规范的转换并行代码的方法,以 OpenMP 为对象,生成的并行仿真代码可以在多种系统平台上进行执行分析,具有更大的灵活性.

5 结论与未来工作

本文提出了一种针对 SIGNAL 同步规范的并行仿真代码生成方法.首先,基于布尔方程解析给出了提取 SIGNAL 程序时钟信息的时钟演算方法.之后,本文提出了方程依赖图 EDG 的概念,用于描述 SIGNAL 程序的全局数据依赖关系,并给出了基于拓扑排序方法对 EDG 进行并行任务划分的算法.最终并行任务序列被映射到 OpenMP 的 sections 结构.生成的 OpenMP 并行代码可以正确地仿真 SIGNAL 程序的功能行为,并且相对于针对特定平台线程库的仿真程序具有更大的灵活性.

未来工作将围绕以下 3 个方面进行:(1) 对并行任务划分方法进行优化,提高仿真代码并行度及执行效率.(2) 对代码生成过程作进一步的形式化,将流程进行模块分解,每一部分都将被形式化,并使用 Coq 或 Caml 等形式化语言对其正确性进行验证;在对每个模块进行验证之后,再对整体流程进行验证,以保证编译过程全程的正确性.(3) 对 weak endochrony 理论进行深入研究,研究验证 SIGNAL 程序是否满足 weak endochrony 的可行算法;在此基础上,研究从 weak endochrony 属性的 SIGNAL 程序到并行代码的生成方法.

致谢 来自法国 Toulouse Institute of Computer Science Research 的 Mamoun Filali 研究员与 Jean-Paul Bodeveix 教授在本文的写作过程中给予了很多重要的建议,在此对他们的帮助表示深深的感谢.

References:

[1] Knight JC. Safety critical systems: Challenges and directions. In: Proc. of the 24th Int'l Conf. on Software Engineering. Orlando, 2002. 547–550. [doi: 10.1145/581339.581406]

[2] Al DJ. MPI: A message-passing interface standard. Int'l Journal of Supercomputer Applications, 2009,20(2):179.

[3] The OpenMP API specification for parallel programming. http://www.openmp.org/specifications/

[4] Intel Thread Building Blocks. https://software.intel.com/en-us/forums/intel-threading-building-blocks/

[5] Benveniste A, Gérard B. The synchronous approach to reactive and real-time systems. Proc. of the IEEE, 1991,79(9):1270–1282. [doi: 10.1109/5.97297]

[6] Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, De Simone R. The synchronous languages 12 years later. Proc. of the IEEE, 2003,91(1):64–83. [doi: 10.1109/JPROC.2002.805826]

[7] Berry G, Gonthier G. The ESTEREL synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 1992,19(2):87−152. [doi: 10.1016/0167-6423(92)90005-V]

[8] Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language LUSTRE. Proc. of the IEEE, 1991,79(9):1305−1320. [doi: 10.1109/5.97300]

[9] Le Guernic P, Gautier T, Le Borgne M, Le Maire C. Programming real-time applications with SIGNAL. Proc. of the IEEE, 1991, 79(9):1321−1336. [doi: 10.1109/5.97301]

[10] Schneider K. The synchronous programming language QUARTZ. Technical Report, Internal Report 375, Kaiserslautern: Department of Computer Science, University of Kaiserslautern, 2009.

[11] Yu HF, Ma Y, Thierry G, Loïc B, Jean-Pierre T, Le Guernic P, Yves S. Exploring system architectures in AADL via Polychrony and SynDEx. Frontiers of Computer Science, 2013,7(5):627−649. [doi: 10.1007/s11704-013-2307-z]

[12] Polychrony. http://www.irisa.fr/espresso/Polychrony/

[13] Kirsch CM. Principles of real-time programming. In: Proc. of the 2002 Conf. on Embedded Software. Berlin, Heidelberg: Springer-Verlag, 2002. 61−75. [doi: 10.1007/3-540-45828-X_6]

[14] Jantsch A, Sander I. Models of computation and languages for embedded system design. In: Proc. of the IEEE of Computers and Digital Techniques—IET. 2005. 114−129. [doi: 10.1049/ip-cdt:20045098]

[15] Potop-Butucaru D, de Simone R, Jean-Pierre T. The synchronous hypothesis and synchronous languages. 2015. http://pdf.aminer.org/000/213/379/modeling_distributed_embedded_systems_in_multiclock_esterel.pdf

[16] Gamatie A. Designing Embedded Systems with the Signal Programming Language: Synchronous, Reactive Specification. Springer Publishing Company, Incorporated, 2010. [doi: 10.1007/978-1-4419-0941-1]

[17] Le Guernic P, Jean-Pierre T, Le Lann JC. Polychrony for system design. Journal for Circuits, Systems and Computers, 2003,12(3): 261−303. [doi: 10.1142/S0218126603000763]

[18] Besnard L, Gautier T, Le Guernic P. Signal v4-Inria version: Reference Manua. 2004. http://www.irisa.fr/espresso/Polychrony/document/V4_def.pdf

[19] Yang Z, Bodeveix JP, Filali M. A comparative study of two formal semantics of the SIGNAL language. Frontiers of Computer Science, 2013,7(5):673−693. [doi: 10.1007/s11704-013-3908-2]

[20] Hu K, Zhang T, Yang Z. Multi-Threaded code generation from signal to OpenMP. Frontiers of Computer Science, 2013, 7(5):617−626. [doi: 10.1007/s11704-013-3906-4]

[21] Jean-Pierre T, Ouy J, Gautier T, Besnard L, Guernic PL. Compositional design of isochronous systems. Science of Computer Programming, 2012,77(2):113−128. [doi: 10.1016/j.scico.2010.06.006]

[22] Benveniste A, Caillaud B, Le Guernic P. From synchrony to asynchrony. In: Proc. of the CONCUR. 1999. 162−177. [doi: 10.1007/3-540-48320-9_13]

[23] Hu K, Zhang T, Yang Z, Tsai W. Simulation of real-time systems with clock calculus. Simulation Modelling Practice and Theory, 2015,51:69−86. [doi: 10.1016/j.simpat.2014.10.010]

[24] Maffeïs O, Le Guernic P. Combining dependability with architectural adaptability by means of the SIGNAL language. In: Static Analysis. Berlin, Heidelberg: Springer-Verlag, 1993. 99−110. [doi: 10.1007/3-540-57264-3_32]

[25] Plotkin G, Stirling CP, Tofte M. The foundations of Esterel. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner, Vol.1. MIT Press, 2000. 425−454.

[26] Besnard L, Gautier T, Jean-Pierre T. Code generation strategies in the Polychrony environmen. 2009. http://hal.inria.fr/docs/00/37/24/12/PDF/RR-6894.pdf

[27] Potop-Butucaru D, Caillaud B, Benveniste A. Concurrency in synchronous systems. Formal Methods in System Design, 2006, 28(2):111−130. [doi: 10.1007/s10703-006-7844-8]

[28] Potop-Butucaru D, Simone RD, Sorel Y, Talpin JP. From concurrent multi-clock programs to deterministic asynchronous implementations. Fundamenta Informaticae, 2011,108(1):91−118.

[29] Benveniste A, Caillaud B, Le Guernic P. Compositionality in dataflow synchronous languages: Specification and distributed codegeneration. Information and Computation, 2000,163(1):125−171. [doi: 10.1006/inco.2000.9999]

[30] Jose BA, Shukla SK, Patel HD, Talpin JP. On the deterministic multi-threaded software synthesis from polychronous specifications. In: Proc. of the 6th ACM & IEEE Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE 2008). 2008. 129–138. [doi: 10.1109/MEMCOD.2008.4547700]

[31] Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In: Proc. of the Conf. on Design, Automation and Test in Europe. European Design and Automation Association, 2010. 949–952.

胡凯(1963－),男,湖南长沙人,博士,教授,主要研究领域为分布式系统,嵌入式实时系统.

张腾(1989－),男,硕士,主要研究领域为形式化方法,安全关键实时系统设计.

尚利宏(1971－),男,博士,副教授,CCF 专业会员,主要研究领域为嵌入式系统.

杨志斌(1982－),男,博士,CCF 专业会员,主要研究领域为形式化方法,安全关键实时系统.

Jean-Pierre TALPIN(1964－),男,博士,教授,博士生导师,主要研究领域为形式化方法,嵌入式系统.