软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(7):1980−2002 [doi: 10.13328/j.cnki.jos.005754]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种同步语言多线程代码自动生成工具*

杨志斌[1,2], 袁胜浩[1], 谢 健[1], 周 勇[1], 陈 哲[1], 薛 垒[3], Jean-Paul BODEVEIX[4], Mamoun FILALI[4]

[1](南京航空航天大学 计算机科学与技术学院,江苏 南京 211106)

[2](软件新技术与产业化协同创新中心,江苏 南京 210093)

[3](上海航天电子技术研究所,上海 201109)

[4](IRIT-University of Toulouse, Toulouse 31062, France)

通讯作者: 杨志斌, E-mail: yangzhibin168@163.com

摘 要: 随着安全关键系统对计算性能要求的日趋提高,能够提供更强计算能力而又减少电子设备的体积、重量和功耗的多核处理器将在安全关键领域得到广泛应用.同步语言能够表达确定性并发行为且具有精确时间语义等特性,适用于安全关键软件的建模和验证.目前,同步语言 SIGNAL 编译器主要支持串行代码生成,较少关注多线程代码生成.提出一种同步语言 SIGNAL 多线程代码生成工具.首先将 SIGNAL 程序转换为经过时钟演算的 S-CGA 中间程序;之后将 S-CGA 中间程序转换为时钟数据依赖图以分析依赖关系;然后对时钟数据依赖图进行拓扑排序划分,并针对划分结果提出优化算法和基于流水线方式的任务划分方法;最后将划分结果转换为虚拟多线程结构并进一步生成可执行多线程 C/Java 代码.通过在多核处理器上的实验,验证了所提方法的有效性.

关键词: 同步语言;同步多时钟卫式动作;多线程代码生成

中图法分类号: TP311

中文引用格式: 杨志斌,袁胜浩,谢健,周勇,陈哲,薛垒,Jean-Paul BODEVEIX,Mamoun FILALI.一种同步语言多线程代码自动生成工具.软件学报,2019,30(7):1980−2002. http://www.jos.org.cn/1000-9825/5754.htm

英文引用格式: Yang ZB, Yuan SH, Xie J, Zhou Y, Chen Z, Xue L, Bodeveix JP, Filali M. Multi-threaded code generation tool for synchronous language. Ruan Jian Xue Bao/Journal of Software, 2019,30(7):1980−2002 (in Chinese). http://www.jos.org.cn/1000-9825/5754.htm

Multi-threaded Code Generation Tool for Synchronous Language

YANG Zhi-Bin[1,2], YUAN Sheng-Hao[1], XIE Jian[1], ZHOU Yong[1], CHEN Zhe[1], XUE Lei[3], Jean-Paul BODEVIX[4], Mamoun FILALI[4]

[1](School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

[2](Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210093, China)

[3](Shanghai Aerospace Electronic Technology Institute, Shanghai 201109, China)

[4](IRIT-University of Toulouse, Toulouse 31062, France)

* 基金项目: 国家自然科学基金(61502231); 国家重点研发计划(2016YFB1000802); GF基础科研重点项目(JCKY2016203B011); 江苏省自然科学基金(BK20150753); 中央高校基本科研业务费专项资金(NP2017205); 国家自然科学基金委员会-中国民航局民航联合研究基金(U1533130); 南京航空航天大学研究生创新基地(实验室)开放基金(kfjj20181603)

Foundation item: National Natural Science Foundation of China (61502231); National Key Research and Development Program of China (2016YFB1000802); National Defense Basic Scientific Research Project under Grant of China (JCKY2016203B011); Natural Science Foundation of Jiangsu Province, China (BK20150753); Fundamental Research Funds for the Central Universities (NP2017205); Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China (U1533130); Foundation of Graduate Innovation Center in NUAA (kfjj20181603)

本文由“软件形式化验证”专题特约编辑贺飞副教授、张立军研究员推荐.

收稿时间: 2018-07-15; 修改时间: 2018-09-28; 采用时间: 2018-12-13; jos 在线出版时间: 2019-03-28

CNKI 网络优先出版: 2019-03-29 09:14:23, http://kns.cnki.net/kcms/detail/11.2560.TP.20190329.0914.006.html

Abstract: Multi-core processors are being widely used in safety-critical systems, due to the demands of higher computation performance when designing these systems, and strengths of multi-core processors, such as faster computation and SWaP (size, weight, and power) properties. Synchronous languages are suitable for modeling and verification of safety-critical software due to their abilities, e.g. the description of concurrency behaviors and precise timing semantics. At present, the SIGNAL compiler supports to generate the sequential code from synchronous specification. The existing studies pay a little attention to the generation of parallel code from SIGNAL specification. The paper presents a multi-threaded code generation tool for synchronous language. Firstly, the SIGNAL specification is transformed into the intermediate program S-CGA and is carried out the clock calculus. After that, the S-CGA program is transformed into CDDG (clock data dependency graph). Then, the CDDG is partitioned by topological sort, after which an optimized algorithm and a partition algorithm are respectively proposed based on pipeline-style. Finally, the partition results are transformed to VMT (virtual multi-threaded) code which is then transformed into executable multi-threaded C/Java program. The experiment running on multi-core CPUs is given to verify the effectiveness of the proposed methodology.

Key words: synchronous language; synchronous clocked guarded action; multi-threaded code generation

安全关键软件(safety-critical software)[1]是指应用于航空、航天、交通、能源等领域的安全关键系统中,且其运行情况可能引起系统处于危险状态,从而导致财产损失、环境破坏或者人员伤害的一类软件.随着功能需求的发展,能够提供更强计算能力而又能减少电子设备的体积、重量和功耗(size,weight and power,即 SWaP 特性)的多核处理器将在安全关键领域得到广泛应用.例如,在航空领域,风河公司(WindRiver)提出了支持多核处理器的 Vxworks 653 操作系统 V3.0 版本[2];而为了满足将来航天应用的复杂需求,欧空局(European Space Agency,简称 ESA)开发了基于四核 LEON4 的下一代微处理器试验板(LEON4-N2X)[3].

近年来,模型驱动(model-driven),尤其是采用形式化模型驱动的安全关键软件设计与开发方法逐渐受到重视,并被工业界认为是切实可行的重要手段.例如,国际民航领域使用的机载系统适航审定中的软件开发标准 DO-178C[4]就将模型驱动和形式化方法(即 DO-331[5]和 DO-333[6])作为其核心标准的重要技术补充.模型驱动开发方法将模型作为整个软件开发过程的核心元素,在设计阶段就建立软件模型,并尽早进行验证和分析.同时,模型的重用以及基于模型的自动代码生成,都有助于降低软件开发时间和成本.

安全关键系统,是一类反应式系统(reactive system),它不断地和环境进行交互,即从环境中得到输入,经过计算,然后输出给环境,并重复这一过程.其中环境包括:被系统控制的物理设备、系统的操作人员或其他的反应式系统.同步语言基于同步假设理论(synchronous hypothesis)来表达系统的功能行为.目前,有 ESTEREL[7]、LUSTRE[8]、SCADE[9]、SIGNAL[10]、QUARTZ[11]等同步语言,这些同步语言可以看作是对同步假设理论的不同实现,而且同步语言在安全关键系统领域得到实际应用.例如,空客使用 SCADE 对 A350、A380 的飞控子系统[12]进行建模和代码生成.ESTEREL、LUSTRE 和 QUARTZ 语言使用 perfect synchrony 模式(即纯同步模式),即存在一个全局时钟,而 SIGNAL 语言使用 Polychrony 模式(即多态同步模式),可能不存在一个全局时钟,可以更加自然地表达分布式系统.现有的 SIGNAL 编译器 Polychrony 目前主要支持串行代码生成和仿真分析,较少考虑在多核处理器上的多线程代码生成.

在同步语言多线程代码生成方面[13−15],目前研究主要考虑基于全局异步局部同步(globally asynchronous locally synchronous,简称 GALS)的多线程代码生成,即系统由具有不同时钟的多个进程组成,而每个进程是单时钟的.然而,在同步语言模型中,单个进程内部也往往存在潜在的并行执行信息.尤其是可以自然表达分布式系统的 SIGNAL 语言,其语法的基本结构中包含多时钟操作,可以比较自然地支持在进程内部描述并行执行操作.

本文研究基于 SIGNAL 语言表达的单进程程序的多线程代码生成方法,并给出一种基于函数式编程语言 OCAML 实现的同步语言 SIGNAL 多线程代码生成工具.该工具是 SIGNAL 串行代码生成工具 Min SIGNAL[16−18]在多线程代码生成方面的扩展.整个生成器分为前端和后端:首先,已有 MinSIGNAL 编译器前端对输入的 SIGNAL 程序进行用户程序标准化,转换为中间表达式即同步多时钟卫式动作(synchronous clocked guarded actions,简称 S-CGA),并进行时钟演算,生成 S-CGA 中间程序.其次,基于扩展的代码生成器后端输出可执行的多线程 C/Java 代码.扩展的代码生成器后端延续 MinSIGNAL 的模块化设计思想,主要包括:构建时钟数据依赖图(clock data dependency graph,简称 CDDG),以分析 S-CGA 程序中变量(全局的时钟变量和数据变量)之

间的依赖关系;在使用拓扑排序划分算法分析 CDDG 的基础上,分别提出拓扑排序优化算法和基于流水线方式的任务划分方法以提高最终目标代码执行速度;基于划分结果生成虚拟多线程代码,进一步生成可执行多线程 C/Java 程序,并在多核处理器上进行实验.

本文第 1 节简要介绍 SIGNAL 语言的基本概念和我们的一些前期研究.第 2 节给出同步语言多线程代码生成方法的总体研究框架.第 3 节介绍同步语言多线程代码生成方法和步骤,包括多线程代码生成器前端、构造时钟数据依赖图、任务划分算法和多线程代码生成.第 4 节给出基于 OCAML 的工具实现,并基于案例进行可执行多线程代码实验和分析.第 5 节给出相关工作比较.第 6 节是本文的总结和展望.

1 研究背景

1.1 同步语言 SIGNAL

SIGNAL 是由 Le Guernic、Benveniste 和 Gautier 等人提出的一种声明式数据流同步语言.基于同步假设理论,SIGNAL 语言中定义了一类对象,称为信号(signal),是一种带类型的无限长的值序列.在给定逻辑时刻下,信号可以是“存在”状态并具有对应的数值,或者是“缺失”状态,记为⊥.信号处于存在状态对应逻辑时刻组成的集合构成信号对应的逻辑时钟(简称时钟).在 SIGNAL 中,两个信号同步当且仅当其逻辑时钟相同.

SIGNAL 语言通过数据流等式进行建模,它提供 4 类基本的数据流等式结构:(1) 瞬时函数($y:=f(x_1,x_2,\ldots,x_n)$);(2) 延迟($y:=x_1$ \$ *init c*);(3) 条件采样($y:=x_1$ when x_2);(4) 确定性合并($y:=x_1$ default x_2).

此外,按照时钟关系划分,4 类基本结构可以被分为单时钟操作(瞬时函数和延迟)和多时钟操作(条件采样和确定性合并):前者要求所有信号都是同步的,而后者允许信号可以不同步.例如,在确定性合并中,给定逻辑时刻下 x_1 或者 x_2 处于存在状态即可以保证 y 处于存在状态.

SIGNAL 不仅提供基本结构,还提供一些扩展结构以支持显式的时钟操作.例如,时钟同步:$x_1\hat{}=x_2$;时钟并运算 $x:=x_1\hat{}+x_2$、时钟交运算 $x:=x_1\hat{}*x_2$、时钟补运算 $x:=x_1\hat{}-x_2$;时钟小于运算 $x_1\hat{}<x_2$;时钟大于运算 $x_1\hat{}>x_2$.本质上,扩展结构是由基本结构进行复合操作来定义[19].

进程是 SIGNAL 的编程单元(programming unit),由输入、输出以及数据流等式组合起来构成.进程包含组合(composition)与局部声明(local declaration).

另外,SIGNAL 语言具有多种形式语义,如基于踪迹的指称语义[19,20]、基于标签模型的指称语义[19]以及基于同步变迁系统[21]的操作语义等.在文献[22]中,我们基于定理证明器 Coq[23]证明了 SIGNAL 语言的踪迹语义和标签模型语义的等价性.

我们给出一个同步语言 SIGNAL 示例:Count 进程.该进程的第 2 行、第 3 行声明输入信号 y_1,x_1 和 x_2;第 4 行声明输出信号 y;第 5 行、第 6 行数据流等式属于扩展结构,第 7 行~第 11 行数据流等式属于基本结构;第 13 行是局部声明.我们将此程序贯穿全文用于介绍整个多线程代码生成器的工作过程.

Example 1 Count process in SIGNAL

例 1 Count 进程

```
01:  process Count=                          08:  |s1 :=y1 when x1
02:  (? integer y1;                          09:  |s2 :=y2 when x2
03:      boolean x1,x2;                      10:  |x :=s1 default s2
04:    ! integer y;)                         11:  |y:=x+1
05:  (|x1^=x2                                12:  |)
06:  |x1^=y2                                 13:  where integer x, y2, s1, s2;
07:  |y2 :=(y1+1) $ init 2                   14:  end;
```

1.2 前期研究

在分析 SIGNAL 语言的已有编译器 Polychrony 的基础上,我们基于 OCAML 实现了一种新的 SIGNAL 串行代码生成工具 MinSIGNAL[16–18],其编译步骤如图 1 所示.

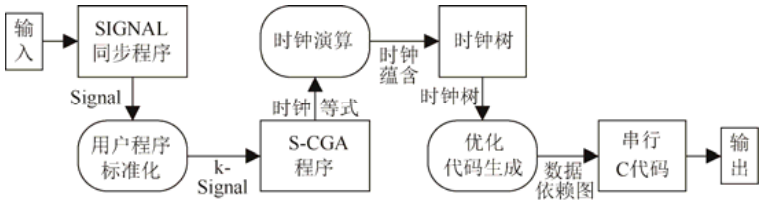

Fig.1 Compiliation steps of the MinSIGNAL sequential code generator

图 1 MinSIGNAL 串行代码生成器编译步骤

相比 Polychrony,MinSIGNAL 具有的特征包括:

(1) MinSIGNAL 支持的输入为 SIGNAL 的子集(称为 MinSIGNAL),包括 SIGNAL 语言中的基本结构、扩展结构、组合、局部声明等,不包括 Polychrony 工具库;

(2) 考虑将来支持多种同步语言的集成,提出了一种新的中间表达式,即同步多时钟卫式动作 S-CGA;

(3) 基于 Coq 完成代码生成器前端 SIGNAL 到 S-CGA 之间的语义保持证明.

本文是 MinSIGNAL 在多线程代码生成方面的扩展,而 MinSIGNAL 的最终目标是生成一个新的 SIGNAL 验证编译器(verified compiler),即在定理证明器 Coq 中规约并证明编译规则的语义保持,然后从证明中自动抽取编译器的 OCAML 实现.目前基于 OCAML 的工具实现是在定理证明器 Coq 中进行规约的基础,并将和自动生成的验证编译器进行对比分析.

2 MinSIGNAL 多线程代码生成研究框架

同步语言 SIGNAL 多线程代码生成器的总体框架如图 2 所示.整个生成器包括前端和后端两个部分:编译器前端对输入的 SIGNAL 程序进行用户程序标准化、转换为 S-CGA 程序、进行时钟演算等步骤生成 S-CGA 中间程序.其次,基于扩展的代码生成器后端输出可执行的多线程 C/Java 代码.扩展的代码生成器后端延续 MinSIGNAL 的模块化设计思想,主要包括:

(1) 定义并构建时钟数据依赖图 CDDG,以分析 S-CGA 程序中变量(全局的时钟变量和数据变量)之间的依赖关系;

(2) 在使用拓扑排序划分算法分析 CDDG 的基础上,分别提出拓扑排序优化算法和基于流水线方式的任务划分方法,以提高最终目标代码执行速度;

(3) 基于划分结果生成虚拟多线程代码(平台无关),进一步生成可执行多线程 C/Java 程序(平台相关).

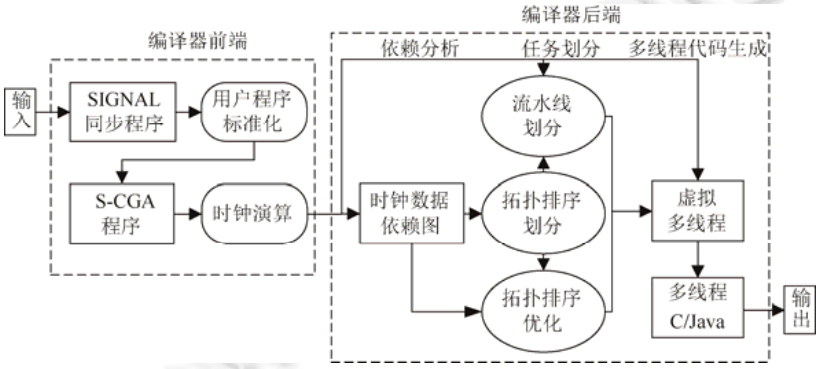

Fig.2 The structure of the MinSIGNAL parallel code generator

图 2 MinSIGNAL 多线程代码生成器结构

串行代码生成过程主要由时钟树驱动完成,多线程代码生成过程主要由数据依赖图驱动完成.即在串行代码生成过程中,通过分析时钟树上包含的时钟关系得到串行代码的控制结构,分析数据依赖图中包含的数据依赖关系确定控制结构中语句的执行顺序;在多线程代码生成过程中,还需要定义 CDDG 以便同时显式地考虑时钟和数据之间的依赖关系.

3 MinSIGNAL 多线程代码生成方法和步骤

MinSIGNAL 多线程代码生成器由前端和后端两部分组成.首先简要介绍代码生成器前端,即已有 MinSIGNAL 串行代码生成器前端及时钟演算.然后,按照编译步骤依次介绍时钟数据依赖图、任务划分和多线程代码生成.

3.1 多线程代码生成器前端

由于此代码生成器前端主要是重用已有的MinSIGNAL串行代码生成器前端(参见文献[16–18]),这里,我们仅简要给出其主要步骤:用户程序标准化、中间语言 S-CGA 和时钟演算.

(1) 用户程序标准化

如第 1.1 节和第 1.2 节所述,MinSIGNAL 支持的输入为 SIGNAL 子集(称为 MinSIGNAL),包括 SIGNAL 语言中的基本结构、扩展结构、组合、局部声明等,不包括 Polychrony 工具库.所谓用户程序标准化,即代码生成器在对用户程序进行词法和语法分析之后,将用户程序中使用的扩展结构转换为 SIGNAL 语言的基本结构.我们称基本结构为 kSIGNAL(kernel SIGNAL).kSIGNAL 的抽象语法如下所示.

$$
\begin{aligned}
P ::= {} & x := f(x_1,...,x_n) && \text{instantaneous function}\\
| {} & x := x_1 \ \$ \text{ init } c && \text{delay}\\
| {} & x := x_1 \text{ when } x_2 && \text{undersampling}\\
| {} & x := x_1 \text{ default } x_2 && \text{deterministic merging}\\
| {} & P \mid P' && \text{composition}\\
| {} & P / x && \text{local declaration}
\end{aligned}
$$

我们在文献[16–18]中给出了用户程序标准化的转换规则.这里,针对例 1 给出部分转换示例.

对于扩展结构,如时钟同步操作 x_1^=x_2,首先分别定义 x_1 和 x_2 的时钟变量 $C\_\_\_1$ 和 $C\_\_\_2$,然后通过两个时钟变量相等表示时钟同步操作^=.

时钟同步操作	标准化后结果
x_1^=x_2	$C\_\_\_1=x_1==x_1$ $C\_\_\_2=x_2==x_2$ $C\_\_\_3=C\_\_\_1==C\_\_\_2$

另外,还需对复合的基本结构进行处理,目的是为了降低源程序到中间语言的转换难度.例如,y_2 :=(y_1+1) $ init 2 的标准化结构如下.

复合基本结构	标准化后结果
y_2 :=(y_1+1) $ init 2	$K\_\_\_8=K\_\_\_8$ $ init 1 $F\_\_\_7=y_1+K\_\_\_8$ $y_2=F\_\_\_7$ $ init 2

(2) 中间表达式 S-CGA

随着同步语言的发展,学术界已提出多种同步语言,导致集成不同同步语言成为热点研究问题[24,25],因为这不仅可以更深刻地理解不同同步计算模型之间的区别,更重要的是,可以提供或重用统一的验证、仿真和代码生成工具.集成的解决方案主要是给出一种通用的中间表达式.例如,Polychrony 中使用 HCDG[26]、L2C 项目中使用 S-Lustre*AST[27,28]、QUARTZ 编译器使用 CGA[29,30]以及本文使用的 S-CGA 等中间语言.

定义 1(同步多时钟卫式动作(S-CGA)). 在 S-CGA 中卫式动作的形式为$\gamma \Rightarrow A$,其中,γ为卫式,A 为要执行的动作.其直观语义为,如果卫式γ为真,则执行动作 A.$\gamma \Rightarrow A$ 主要包括 5 种形式,见表 1.

Table 1 Form of guarded actions

表 1 卫式动作形式

卫式动作名称	卫式动作形式
立即动作	$\gamma \Rightarrow x=\tau$
延迟动作	$\gamma \Rightarrow next(x)=\tau$
约束定义	$\gamma \Rightarrow assume(\sigma)$
输入动作	$\gamma \Rightarrow$read x
输出动作	$\gamma \Rightarrow$write x

其中,

• 卫式γ是定义在信号变量 X(源 SIGNAL 中的输入、输出和局部变量)、X 的逻辑时钟($x \in X$,其时钟为 $\hat{x}$)及其初始时钟($init(\hat{x})$)之上的布尔条件.

• 动作 A 包括:立即赋值、延迟赋值、约束定义、输入动作和输出动作.其中,τ是定义在 X 上的表达式,σ则是定义在 X 及其逻辑时钟上的布尔表达式.

(1) 立即赋值,给定逻辑时刻 t,当γ中所有信号变量都处于存在状态且γ为 true、x 和τ都处于存在状态时,则将τ的取值赋给 x;

(2) 延迟赋值,给定逻辑时刻 t_1,γ中所有信号变量都处于存在状态,且γ为 true,x 和τ在 t_1 时刻都处于存在状态,在下一逻辑时刻 t_2 中,如果 x 处于存在状态,则在 t_2 时刻将 t_1 时刻τ的取值赋值给当前时刻的 x;

(3) 约束定义,如果γ中所有信号变量同时处于存在状态且γ为 true,那么σ处于存在状态并且为 true.所有 S-CGA 程序中的操作都需要满足约束定义;

(4) 输入动作,当γ=true 时,从环境中读取输入变量 x;

(5) 输出动作,当γ=true 时,将计算结果 x 输出到环境.

多个 S-CGA 组合采用同步组合操作||.因此,S-CGA 具有与 SIGNAL 语言一致的表达确定性并发行为的能力.

表 2 给出 kSIGNAL2S-CGA 的转换规则.基于转换规则,SIGNAL 语言中的基本结构(即 kSIGNAL)所表达的功能关系和时钟关系一一映射为 S-CGA 中的对应表示.SIGNAL 到 S-CGA 之间具体的转换规则解释以及语义保持证明思路参考文献[16].

Table 2 Rules of kSIGNAL2S-CGA

表 2 kSIGNAL 到 S-CGA 的转换规则

kSIGNAL	S-CGA
$x := f(x_1,...,x_n)$	$\begin{cases} \hat{x} \Rightarrow x = f(x_1,...,x_n) \\ \| \ \hat{x}_1 \Rightarrow assume(x_1) \\ \| \ ... \\ \| \ \hat{x}_n \Rightarrow assume(x_n) \end{cases}$
$x := x_1$ \$init c	$\begin{cases} init(\hat{x}) \Rightarrow x = c \\ \| \ \hat{x}_1 \Rightarrow next(x) = x_1 \\ \| \ \text{true} \Rightarrow assume(\hat{x} = \hat{x}_1) \end{cases}$
$x := x_1$ when x_2	$\begin{cases} \hat{x}_1 \wedge x_2 \Rightarrow x = x_1 \\ \| \ \hat{x} \Rightarrow assume(\hat{x}_1 \wedge x_2) \end{cases}$
$x := x_1$ default x_2	$\begin{cases} \hat{x}_1 \Rightarrow x = x_1 \\ \| \ \hat{x}_2 \wedge \neg\hat{x}_1 \Rightarrow x = x_2 \\ \| \ \hat{x} \Rightarrow assume(\hat{x}_1 \vee \hat{x}_2) \end{cases}$

(3) 时钟演算

同步模型除显式地表达功能关系外,还隐式地表达了时钟关系.因此,在代码生成之前,一般同步语言编译器都需要经过时钟演算,从而确定该同步模型可执行,并优化生成后的代码.首先,根据 S-CGA 语义,从 S-CGA 表达式中生成时钟关系等式集合,其基本产生规则见表 3.

Table 3 Rules of S-CGA2Clock equations

表 3 S-CGA 到时钟关系等式的转换规则

S-CGA	时钟关系等式
$\gamma \Rightarrow x=\tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow next(x)=\tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow assume(\sigma)$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{\sigma} \wedge \sigma$
$\gamma \Rightarrow$read x	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge x$
$\gamma \Rightarrow$write x	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge x$
	$init(\hat{x}) \rightarrow \hat{x}\ (\forall x \in X)$

针对例 1 给出部分转换示例及其时钟关系等式如下.

SIGNAL	部分 S-CGA	部分时钟关系等式
s_1:=y_1 when x_1 s_2:=y_2 when x_2 x:= s_1 default s_2	$\hat{s}_1 \Rightarrow assume(\hat{y}_1 \wedge x_1)$ $\hat{s}_2 \Rightarrow assume(\hat{y}_2 \wedge x_2)$ $\hat{x} \Rightarrow assume(\hat{s}_1 \vee \hat{s}_2)$ ...	$\hat{x} = (\hat{y}_1 \wedge x_1) \vee (\hat{y}_2 \wedge x_2)$ ...

等式左侧 $\hat{x}$ 为时钟变量,而右边等式是对应的定义,时钟关系等式 $x \rightarrow y$ 和 $y \rightarrow x$ 可表示为 $x=y$.在时钟关系等式集合中,需要对具有相同时钟的不同时钟关系等式进行优化以保证生成后的代码执行效率更高.时钟关系等式集合的消解原理为:将等式右侧中的时钟变量用其定义进行替换,并不断地迭代,最后检查不同时钟变量之间是否等价(基于 BDD 或 SMT).如果等价,则归为同一个时钟等价类,时钟演算之后的程序中时钟等价类用于替代 S-CGA 对应的时钟变量.

针对例 1,代码生成器前端生成的部分 S-CGA 程序如图 3 所示.S-CGA 程序中包括输入动作、输出动作、立即动作和延迟动作.此外,约束定义只用于时钟演算,不参与之后的任务划分等步骤.

SIGNAL 程序	S-CGA 程序(经过时钟演算)
01: process Count= 02: (? integer y_1; 03: boolean x_1,x_2; 04: ! integer y;) 05: ($\|x_1$^=x_2 06: $\|x_1$^=y_2 07: $\|y_2$:=(y_1+1) $ init 2 08: $\|s_1$:=y_1 when x_1 09: $\|s_2$:=y_2 when x_2 10: $\|x$:=s_1 default s_2 11: $\|y$:=x+1 12: |) 13: where integer x, y_2, s_1, s_2; 14: end;	01: *ID*_4⇒read y_1 02: *ID*_4⇒read x_1 03: *ID*_4⇒read x_2 04: *ID*_4⇒write y 05: *ID*_4⇒*C*___1=x_1==x_1 06: *ID*_4⇒*C*___2=x_2==x_2 07: *ID*_4⇒*C*___3=*C*___1==*C*___2 08: *ID*_4⇒*C*___4=y_2==y_2 09: *ID*_4⇒*C*___5=*C*___1==*C*___4 10: *init*(*ID*_4)⇒y_2=2 11: *init*(*ID*_4)⇒*K*___8=1 12: *ID*_4⇒*next*(*K*___8)=*K*___8 13: *ID*_4⇒*F*___7=y_1+*K*___8 14: *ID*_4⇒*next*(y_2)=*F*___7 15: *ID*_4⇒*ID*_5=x_1 16: *ID*_4⇒*ID*_6=*ID*_4 and *ID*_5 17: *ID*_6⇒s_1=y_1 18: *ID*_6⇒x=s_1 19: *ID*_4⇒*ID*_15=*ID*_8 diff *ID*_6 20: *ID*_15⇒x=s_2 21: *ID*_4⇒*ID*_10=*ID*_6 or *ID*_8 22: *ID*_4⇒*ID*_7=x_2 23: *ID*_4⇒*ID*_8=*ID*_4 and *ID*_7 24: *ID*_8⇒s_2=y_2 25: *init*(*ID*_10) ⇒*K*___9=1 26: *ID*_10⇒y=x+*K*___9 27: *ID*_10⇒*next*(*K*___9)=*K*___9

Fig.3 The generated S-CGA repsentation of the Count process

图 3 Count 进程到 S-CGA 的转换示例

3.2 时钟数据依赖图

SIGNAL 程序中的所有信号变量可具有各自的时钟,如果生成串行代码,即需要确定性的执行顺序,编译器基于时钟等式和时钟等价类,构造出一棵时钟树(clock hierarchy).能够构造时钟树,代表所有时钟都隶属于一个全局时钟(master clock),即可以给出一个确定性的串行执行顺序,在同步语言中,称其为Endochrony性质.在同步语言串行代码生成中,根据时钟等价类之间的时钟蕴含关系构建时钟树,基于遍历时钟树结构,生成串行代码的控制结构,并根据等式之间的读写依赖关系生成执行顺序,依次填写到对应的控制结构中.而在多线程代码生成中,需要进一步构造数据依赖图,从而分析出更多的并行信息.

首先,使用时钟等价类替换对应的所有时钟变量以简化程序.其次,基于 S-CGA 表达式中的变量间读写依赖关系构建数据依赖图.例如,对于卫式动作$\gamma\Rightarrow x=\tau$,只有当获得γ和τ中变量的取值后,才可以执行该卫式动作对 x 进行赋值,此时,γ和τ为读依赖变量,x 为写依赖变量.*RdVars*/*RdActs* 用于表示读依赖变量/动作集合,*WrVars*/*WrActs* 用于表示写依赖变量/动作集合.

定义 2(读写依赖). 设 $FV(\tau)$为表达式τ中的自由变量的集合.S-CGA 等式中的动作到变量的依赖定义为

- $RdVars(\gamma \Rightarrow x = \tau) := FV(\gamma) \cup FV(\tau)$
- $WrVars(\gamma \Rightarrow x = \tau) := \{x\}$

而变量到动作的读写依赖定义如下.

- $RdActs(x) := \{\gamma \Rightarrow A \mid x \in RdVars(\gamma \Rightarrow A)\}$
- $WrActs(x) := \{\gamma \Rightarrow A \mid x \in WrVars(\gamma \Rightarrow A)\}$

MinSIGNAL 多线程代码生成器采用基于周期的目标代码执行方式:(1) 初始化程序;(2) 计算程序体;(3) 更新变量.然后进入下一个周期,迭代执行(2)和(3).我们主要分析程序体中的依赖关系.含有初始时钟值 $init(\hat{x})$ 的立即动作主要用于程序初始化,而延迟动作用于更新变量,$\gamma \Rightarrow assume(\sigma)$ 则主要用于定义时钟约束关系,因此,这 3 类卫式动作不用于构造数据依赖图.

在数据依赖图中,只有当一个动作的所有读变量都有取值时,该动作才可以被执行.类似地,只有当一个变量的所有写动作都已经完成计算后,该变量才能够有取值.例如,在上述 S-CGA 程序中,$ID\_4\Rightarrow ID\_5=x_1$ 依赖于 $ID_4\Rightarrow$read x_1 获得的 x_1 值.

定义 3(时钟数据依赖图(CDDG)). 时钟数据依赖图 CDDG 为一个有向图$\langle V,DR\rangle$,V 代表卫式动作集合,$DR\subseteq V\times V$ 代表卫式动作之间的依赖关系集合.

时钟数据依赖图的构造规则如下.对任意的动作 $a\in A$ 以及变量 $x\in RdVars(a)$,在 $WrActs(x)$对应的卫式动作到 a 对应的卫式动作之间增加一条有向边.

图 4 为 Count 进程对应的 S-CGA 程序依据构造规则生成的时钟数据依赖图 CDDG.

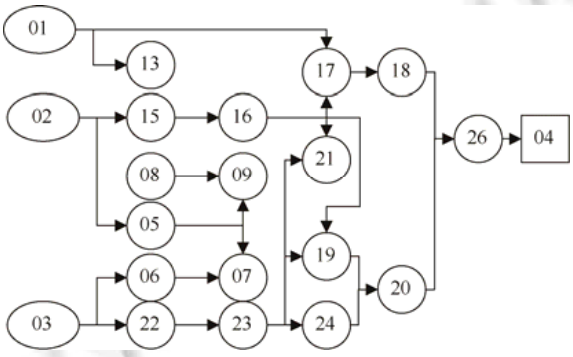

Fig.4 The CDDG of the Count process

图 4 Count 进程的 CDDG

我们使用 S-CGA 程序中的行号表示对应的卫式动作,使用不同的形状区分卫式动作.其中,椭圆表示输入动作,矩形表示输出动作,圆形表示立即动作.而箭头→表示依赖关系,例如,22→23 表示 $ID\_4 \Rightarrow ID\_8 = ID\_4$ and ID_7 依赖 $ID\_4 \Rightarrow ID\_7 = x_2$ 获得的 ID_7 值.

3.3 任务划分

任务划分用于分析 CDDG 中包含的并行执行信息,直接影响目标代码的并行度,是多线程代码生成器的核心步骤.首先,CDDG 是一个有向图,根据拓扑排序思想对 CDDG 进行任务划分;其次,基于划分结果分别提出优化算法和基于流水线方式的任务划分,其中前者用于处理划分结果中可能存在丢失依赖关系和线程数目过多的问题,后者用于尝试寻找划分中更大的并行度.

3.3.1 基于拓扑排序的任务划分

多线程代码生成是基于已经融合时钟关系的数据依赖图,并需要在数据依赖图中进行任务划分(partition).CDDG 是有向无环图,我们基于拓扑排序方法对 CDDG 进行任务划分.

划分算法如图 5 所示:算法输入为时钟数据依赖图 CDDG,输出为任务划分结果 topoPartition.算法第 1 步分别对 NodeSet、EdgeSet 和 topoPartition 进行初始化(第 4 行~第 6 行),然后进入拓扑排序划分迭代(第 7 行).迭代过程包含以下步骤:首先从 NodeSet 中计算出所有入度为 0 的点组成的集合 P_i,i 表示迭代次数(第 8 行);其次,如果 P_i 不为空,则将 P_i 加入到 topoPartition 的末尾.如果 P_i 为空,表明 CDDG 中存在环,则算法终止,输出异常信息"CDDG 中存在环"(第 9 行);最后将 P_i 中出现的点从 NodeSet 中移除以及从 EdgeSet 中移除与这些点有关的边(第 10 行、第 11 行).当 NodeSet 为空时迭代终止.

```
01:  proceduce topo_Partition (CDDG).
02:  Input:CDDG;
03:  Output:topoPartition;
04:      NodeSet←CDDG.V %NodeSet 初始值为时钟数据依赖图 CDDG 中所有点
05:      EdgeSet←CDDG.DR %EdgeSet 初始值为时钟数据依赖图 CDDG 中所有边
06:      topoPartition←[] %将 topoPartition 初始值设置为空列表
07:      while NodeSet≠∅ do
08:          Pi←getInDegreeZeroV(NodeSet) %获取所有入度为 0 的点
09:          addTask(topoPartition, Pi) %将 Pi 加入到 topoPartition 的末尾
10:          deleteNodes(NodeSet, Pi) %从 NodeSet 中删除 Pi 中出现的所有节点
11:          delateEdges(EdgeSet, Pi) %从 EdgeSet 中删除与 Pi 中节点有关的边
12:      end while
13:      return topoPartition
14:  end proceduce
```

Fig.5 Algorithm for task partition of clock and data dependency graph based on topological sorting

图 5 基于拓扑排序的 CDDG 任务划分算法

例 1 对应的 CDDG 生成划分结果 topoPartition 如下所示.

$$
\begin{aligned}
&P_1 = 1 \parallel 2 \parallel 3 \parallel 8\\
&P_2 = 5 \parallel 6 \parallel 13 \parallel 15 \parallel 22\\
&P_3 = 7 \parallel 9 \parallel 16 \parallel 23\\
&P_4 = 17 \parallel 19 \parallel 21 \parallel 24\\
&P_5 = 18 \parallel 20\\
&P_6 = 26\\
&P_7 = 4\\
&topoPartition = [P_1;P_2;P_3;P_4;P_5;P_6;P_7]
\end{aligned}
$$

其中,$P_i = id_1 || id_2 \ldots || id_n$ 为一个划分,表示 $id_1, id_2, \ldots, id_n$ 可以并行执行,id 代表 CDDG 中对应节点的行号.以 P_1 为例,1||2||3||8 表示卫式动作 $ID_4 \Rightarrow$read y_1,$ID_4 \Rightarrow$read x_1,$ID_4 \Rightarrow$read x_2 和 $ID\_4 \Rightarrow C\_\_\_4 = y_2 == y_2$ 彼此可以并行执行.$P_i;P_{i+1}$ 表示两者之间串行执行.即当 P_i 中卫式动作执行结束,开始执行 P_{i+1} 中卫式动作.

然而,这种划分方法得到的划分结果可能存在以下问题.

首先,划分结果可能导致在最后代码生成的线程调度时,增加多余的同步开销.以 P_2 中卫式动作 $ID\_4\Rightarrow C\_\_\_1=x_1==x_1$ 为例,根据 topoPartition,该卫式动作开始执行需要等待 P_1 中所有卫式动作执行结束.但根据 CDDG 可知,$ID\_4\Rightarrow C\_\_\_1=x_1==x_1$ 开始执行依赖 $ID_4\Rightarrow$read x_1 执行结束.划分结果导致生成的线程之间也存在多余的通信信息,增加了线程之间的同步开销时间.在实际执行目标代码过程中,原本可以执行的线程必须等待其他额外的线程执行结束才可以开始执行,从而可能影响目标代码的执行效率.

其次,由于使用基于拓扑排序的划分方法,划分结果中的每个卫式动作在代码生成时将对应转换为一个线程.以 P_4 中 17(对应图 3 中 $ID\_6\Rightarrow s_1=y_1$)为例,其转换对应生成的线程的步骤包括:17 对应生成线程名 Thread_17;卫式动作转换为线程的计算部分;该线程开始执行依赖获得 P_3 对应的所有线程的执行结束信息;当该线程执行结束,则将结束信息通知 P_5 对应的所有线程.因此,当卫式动作数目过多时,导致线程数目过多,可能会增加线程间的资源竞争,从而影响目标程序的执行效率.

3.3.2 划分优化

针对划分结果 topoPartition 可能存在的问题,我们给出基于拓扑排序划分的优化算法,即记录时钟数据依赖图 CDDG 中的依赖关系以及尝试减少目标线程数目.

首先,我们给出如下定义.

定义 4(卫式动作等待). 设时钟数据依赖图 CDDG=⟨V,DR⟩,其中,V 为卫式动作集合,DR 为卫式动作间的依赖关系集合.卫式动作 b 的等待定义为 $WAITS(b)\overset{def}{=}\{a\in V \mid x\in DR \text{ and } x=a\rightarrow b\}$.

定义 5(卫式动作通知). 设时钟数据依赖图 CDDG=⟨V,DR⟩,其中,V 为卫式动作集合,DR 为卫式动作间的依赖关系集合.卫式动作 a 的通知定义为 $NOTIFYS(a)\overset{def}{=}\{b\in V \mid x\in DR \text{ and } x=a\rightarrow b\}$.

特别地,当 $WAITS(b)=\{a\}$,即 $|WAITS(b)|=1$ 时,记为 $WAITS(b)=\{a\}$.同样地,当 $NOTIFYS(a)=\{b\}$ 时,记为 $NOTIFYS(a)=\{b\}$.根据定义 4 和定义 5,遍历 CDDG 中依赖关系计算出所有卫式动作对应的等待和通知,并将其加入到优化划分结果 topoOptimization 中,从而在 topoOptimization 中记录时钟依赖图的依赖关系.此外,定义 4 和定义 5 也便于在后续步骤中找出可以合并的线程以及处理目标代码中线程之间的同步.

其次,给出尝试减少目标线程数目的方法.

如果卫式动作 a 和 b 满足 $WAIT(b)=\{a\}$ && $NOTIFY(a)=\{b\}$, 则合并 a 和 b 为一个特殊的“卫式动作”,记为 $Ma=[a;b]$,其中, $WAITS(Ma)=WAITS(a)$, $NOTIFYS(Ma)=NOTIFYS(b)$. Ma 表示卫式动作 a 和 b 在生成目标代码中将合并为一个线程.

图 6 给出基于拓扑排序划分优化算法.首先,计算出所有卫式动作对应的等待和通知并加入到优化结果中(第 4 行~第 6 行);其次,在原有并行执行部分的基础上进行线程合并(第 7 行).

```
01:  proceduce topo_Optimization (CDDG, topoPartition)
02:  Input:CDDG, topoPartition;
03:  Output:topoOptimization.
04:      Ws←getWAITS(CDDG) %计算出所有卫式动作对应 WAITS
05:      Ns←getNOTIFYS(CDDG) %计算出所有卫式动作对应 NOTIFYS
06:      addWaitNotify(topoOptimization,Ws,Ns) %将等待和通知添加到优化划分结果中
07:      topoOptimization←mergeThread(topoPartition,Ws,Ns) %合并线程
08:      return topoOptimization
09:  end proceduce
```

Fig.6 Optimization algorithm for task partition based on topological sorting

图 6 基于拓扑排序划分的优化算法

我们对第 3.3.1 节给出的划分结果进行优化,得到优化划分结果 topoOptimization 如下所示.

其中,划分 P_2 的[15;16]、[22;23],P_4 的[17;18]和 P_6 的[26;4]为合并后的卫式动作.例如,[17;18]表示在目标代码生成时,卫式动作 $ID\_6\Rightarrow s_1=y_1$ 和 $ID\_6\Rightarrow x=s_1$ 将会被分配到一个线程中顺序执行.同时,等待和通知记录依赖关系,例如 $WAITS(22)=\{3\}$表示卫式动作 $ID\_4\Rightarrow ID\_7=x_2$ 依赖于输入动作 $ID_4\Rightarrow$read x_2 得到的 x_2 值.

$P_1 = 1 \parallel 2 \parallel 3 \parallel 8$
$P_2 = 5 \parallel 6 \parallel 13 \parallel [15;16] \parallel [22;23]$
$P_3 = 7 \parallel 9$
$P_4 = [17;18] \parallel 19 \parallel 21 \parallel 24$
$P_5 = 20$
$P_6 = [26;4]$
$NOTIFYS(1) = \{13,17\}, NOTIFYS(2) = \{5,15\}, NOTIFYS(3) = \{6,22\},$
$NOTIFYS(5) = \{7,9\}0, NOTIFYS(6) = \{7\}, NOTIFYS(8) = \{9\},$
...
$WAITS(19) = \{16,23\}, WAITS(20) = \{19,24\}, WAITS(21) = \{16,23\},$
$WAITS(22) = \{3\}, WAITS(24) = \{22\}, WAITS(26) = \{18,20\}$

3.3.3 基于流水线方式的任务划分

当基于拓扑排序获得的任务划分存在并行度不高(即大部分 P_i 中可并行执行的卫式动作数目都较少)的情况下,我们提出基于流水线方式(pipeline style)对划分结果进行再划分,尝试提高执行并行度.考虑一个特殊情况为划分结果中所有 P_i 中都只有一个卫式动作,则经过任务划分生成的多线程代码执行情况实际上"等同"于串行程序执行,而通过引入流水线方式划分支持多条流水线并行执行,从而提高目标程序的执行效率.

本文的流水线方式的任务划分算法(如图 7 所示)包括 3 个步骤:首先,根据划分结果定义流水线阶段;其次,针对各流水线阶段增加流水线中间变量;最后,针对不同流水线以及同一流水线下不同流水线阶段之间增加通信函数.

```
01:  proceduce topo_Pipeline (S-CGA,topoPartition,N) %N 为流水线条数
02:  Input:S-CGA, topoPartition;
03:  Output:topoPipeline.
04:      Stage ← definePipelineStage(topoPartition) %定义流水线阶段
05:      addDelayAction (Stage,S-CGA) %添加延迟动作到 Stage 中
06:      for each Stage(i) in Stage do
07:          gas ← getGuardActionSet(Stage(i)) %确定卫式动作集合
08:          for each gas(j) in gas do
09:              addImmediateVaribles(notifys,N) %添加流水线中间变量
10:              addExchangeFunctions(notifys,N) %添加两种数据通信函数
11:          end for
12:      end for
13:      addCommunicationFunctions(Stage,N) %添加延迟通信函数
14:      return topoPipeline
15:  end proceduce
```

Fig.7 Algorithm for task partition based on pipeline style

图 7 基于流水线方式的任务划分算法

(1) 定义流水线阶段(算法第 4 行、第 5 行)

首先将划分结果 topoPartition 中每个划分 P_i 对应于一个流水线阶段 $Stage(i)$.由于 topoPartition 中并未考虑延迟动作 $\gamma \Rightarrow next(x)=\tau$,在确定流水线阶段的同时需要将所有延迟动作 $\gamma \Rightarrow next(x)=\tau$ 加入到相应的 Stage 中,便于之后处理不同流水线间的通信,并保证不同流水线执行结果的正确性.例如,对于延迟动作 $ID\_4 \Rightarrow next(y_2)=F\_\_\_7$,将其加入到 $ID\_4 \Rightarrow F\_\_\_7=y_1+K\_\_\_8$ 所在的 Stage 中.

每个 $Stage(i)$有两种不同的状态:激活状态和阻塞状态.一个 $Stage(i)$称为阻塞状态当且仅当该流水线阶段中存在需延迟动作写入的值未被更新.只有处于激活状态的流水线阶段才可以被执行.在定义流水线阶段时,每个流水线阶段默认为激活状态.

(2) 增加流水线中间变量(算法第 6 行~第 9 行)

得到流水线阶段后,需要考虑如何在每条流水线的各个流水线阶段之间加入流水线中间变量.因为每条流水线的每个阶段同时处理不同逻辑时刻的数据,所以需要一些中间变量来存储各个流水线阶段中生成的临时结果.为此,需要确定流水线中间变量在流水线阶段中加入的确切位置.

对 *Stage*(*i*)中任意的卫式动作 *a*,如果变量 $x \in WrVars(a)$,则在 *Stage*(*i*)和 *Stage*(*i*+1)之间,增加变量 *x* 对应的流水线中间变量.需要注意的是:一个流水线中间变量只适用于一条流水线.对于多条流水线,通过根据流水线条数 *N* 将流水线中间变量调整为一维流水线中间变量数组,从而保证各条流水线之间不产生数据冲突.例如,对于变量 *x* 以及流水线条数 *N*,需要增加的流水线中间变量数组为 *m_x*[*N*],其中,*m_x*[0]表示第 0 条流水线中 *x* 对应的流水线中间变量.

(3) 增加通信函数(算法第 10 行和第 13 行)

增加流水线中间变量后,需要在流水线中间变量与 S-CGA 变量之间建立起数据通信的映射关系,包括两种数据通信函数:数据输入函数 inp_exchange,负责将流水线中间变量值传输到对应的 S-CGA 变量;数据输出函数 outp_exchange,负责将已经更新的 S-CGA 变量值传输到对应的流水线中间变量中.例如,对于立即动作 $\gamma \Rightarrow (x)=\tau$,对 γ、τ调用 inp_exchange 函数,当卫式动作执行结束,调用 outp_exchange 函数更新 *x*.若变量 *x* 在某条流水线中需要的流水线中间变量为 *k* 个,设置 *N* 条流水线,则对于变量 *x* 来说,共需要中间变量为 $k \times N$ 个.

最后,需要考虑不同流水线之间的数据延迟通信问题.在多条流水线中,S-CGA 程序中延迟动作 $\gamma \Rightarrow next(x)=\tau$会影响相邻两条流水线之间的数据通信.因此,为了保证基本流水线的正确信息流,对于那些由延迟动作写入的变量(称为更新值),在每个流水线中必须保证触发一个动作来完成延迟动作.如果当前流水线中具有延迟动作中的节点对应的所有卫式动作的卫式都为 false,则需要增加一个卫式动作来保存更新值,使之保证:如果当前流水线相邻的下个流水线中需要使用延迟动作中的变量,则这个变量的值等于前个流水线中的对应的值.同时设置标志符 update 表示当前流水线中涉及更新值的阶段是否已被更新,update 为真,表示更新值已被更新,否则,该阶段处于阻塞状态.假设更新值的个数为 *t*,设置 *N* 条流水线,则一共需要 $t \times (N-1)$个通信函数,在实际情况下,通信函数的个数可能会小于上述数值.因为对于更新前后的值始终不变的更新值,即由源 SIGNAL 程序中常量生成的变量,其对应的不同流水线之间的通信函数可以进行优化而被去掉.

这里,我们给出针对第 3.3.2 节中 topoOptimization 进行流水线方式的任务划分,默认采用 3 条流水线.首先定义流水线阶段并添加延迟动作,例如 Stage(2)中添加两个延迟动作(序号 12 和序号 14);其次定义流水线中间变量,如图 8 所示,*Stage*(1)和 *Stage*(2)之间增加流水线中间变量数组 y_1、x_1、x_2 和 *C*__4;最后添加通信函数,包括数据通信函数(outp_exchange/inp_exchange)和延迟通信函数(*f_y*$_2$),如图 9 所示.

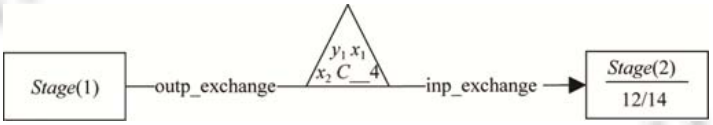

Fig.8 Example of communicating functions

图 8 通信函数示例

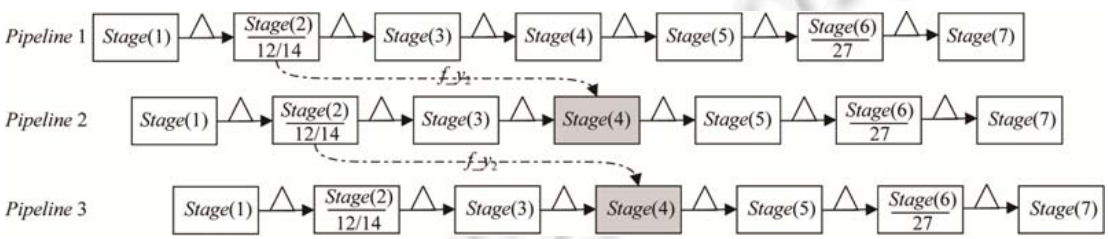

Fig.9 Example of task partition based on pipeline style

图 9 基于流水线方式的任务划分示例

Pipeline 1 中 *Stage*(4)的更新值 y_2 默认由 init 函数激活,所以设置为激活状态,而 *Pipeline* 2 和 *Pipeline* 3 中 *Stage*(4)分别由前一个 *Pipeline* 中的 *Stage*(2)触发,所以后者设置为阻塞状态(灰色).此外,序号 12 和序号 27 对应的卫式动作 *ID*_4$\Rightarrow$*next*(*K*___8)=*K*___8 和 *ID*_4$\Rightarrow$*next*(*K*___9)=*K*___9 是根据源 SIGNAL 程序中的常量而生成,可以优化其对应的延迟通信函数,因此不需要修改更新值.

3.4 多线程代码生成

本文将多线程代码生成过程分为平台相关和平台无关两个层次:首先,根据划分结果生成基于 Wait/Notify 机制的虚拟多线程 VMT(virtual multi-thread)代码;其次,从虚拟多线程代码中生成具体的可执行多线程代码,如 C 或者 Java.

3.4.1 虚拟多线程代码

优化划分中每个卫式动作或合并的卫式动作对应一个线程,如图 10 右侧线程所示,本文采用串行调度方式来确保线程间通信保持同步语义,根据优化划分结果中 *WAITS* 和 *NOTIFYS* 信息实现基于 Wait/Notify 机制的虚拟多线程代码.主函数首先执行初始化(例如,y_2=2),然后进入主循环,等待所有线程完成执行,最后处理延迟赋值操作(更新 y_2 取值).而在主循环中,每个线程(*thread* 1,…,*thread* 26)都执行一个循环:等待(wait)输入、测试卫式、产生输出、通知(notify)其他线程.

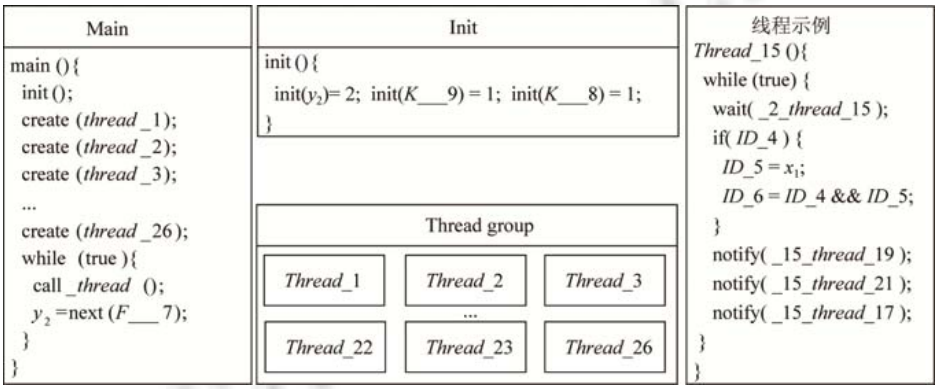

Fig.10 The structure of virutal multi-thread code

图 10 虚拟多线程结构

另外,针对基于流水线方式的任务划分结果生成虚拟多线程结构,则需要对图 10 所示结构做出如下调整.

(1) 保留虚拟多线程的通用结构,即初始化、主函数和线程组.主函数中的 *call_threads*()函数修改为 *call_threads*(*PN*),即在主循环中调用 *PN* 条流水线下的线程.延迟赋值操作从主函数转移到特定的线程中,具体线程是根据流水线阶段中延迟动作加入的位置加以确定.

(2) 增加数据交换和数据通信.数据交换用于处理单个流水线中线程与流水线中间变量的数据交换.数据通信用于处理不同流水线之间线程间的通信.

生成虚拟多线程代码的目的是方便对多线程代码进行形式化验证和仿真分析,如使用模型检测工具 UPPAAL 对多线程代码的无死锁性等性质进行验证,以及在仿真工具 Simulink 上进行仿真实验.同时,支持生成多种目标代码,增加代码生成器后端的可扩展性.

3.4.2 多线程 C 代码

在多线程 C 代码自动生成过程中,使用 POSIX 的同步机制来实现 Wait/Notify.首先,所使用的同步结构由互斥量(pthread mutex)、条件变量(pthread condition variable)以及一个线程所等待事件的数量(value)组成.例如,*thread* 13 需要等待 *thread* 1 的事件,因此,其 *value* 为 1.

```
typedef struct counter{
   pthread_mutex_t mutex;
   pthread_cond_t request;
   int value;
} counter;
```

counter_wait 用于表示 Wait 机制,即,当前线程的 *value* 值大于 0(意味着该线程的一些等待事件还没有到达),则将当前线程设为等待.

```
wait mechanism
void counter_wait(counter*c){
   pthread_mutex_lock(&c→mutex);
   while (c→value>0)
   pthread_cond_wait(&c→request, &c→mutex);
   pthread_mutex_lock(&c→mutex);
}
```

counter_notify 用于表示 Notify 机制,用于唤醒等待线程队列中的某个线程.

```
notify mechanism
void counter_notify(counter*c){
   pthread_mutex_lock(&c→mutex);
assert(c→value>0);
   if( !--c→value){
      pthread_mutex_unlock(&c→mutex);
pthread_cond_signal(&c→request);}
   else
      pthread_mutex_unlock(&c→mutex);
}
```

以图 10 中虚拟多线程为例,我们给出对应多线程 C 代码中部分线程:*thread*_15 和 *thread*_17,如图 11 所示.*thread*_15 开始执行需要等待 *thread*_2 的事件发生,即读取输入信号 x_1.条件语句中顺序执行语句 *ID*_5=x_1 和 *ID*_6=*ID*_4 && *ID*_5 是根据划分结果 topoOptimization 中 P_2 对应的[15;16]生成.当完成计算后,通知线程 *thread*_17,*thread*_19 和 *thread*_21.如果 *thread*_1 已经执行完成,则下一步可以执行 *thread*_17.

多线程 C 代码(*thread*_15)	多线程 C 代码(*thread*_17)

void**thread*_15(void*){
counter_wait(&_2_*thread*_15);
if(*ID*_4){
*ID*_5=x_1;
*ID*_6=*ID*_4 && *ID*_5;
}
counter_notify(&_15_*thread*_17);
counter_notify(&_15_*thread*_19);
counter_notify(&_15_*thread*_21);}
	void**thread*_17(void*){
	counter_wait(&_1_*thread*_17);
	counter_wait(&_15_*thread*_17);
	if(*ID*_6){
	s_1=y_1;
	x=s_1;
	}
	counter_notify(&_17_*thread*_26);
	}

Fig.11 Example of multi-thread C code

图 11 多线程 C 代码示例

3.4.3 多线程 Java 代码

在生成多线程 Java 代码阶段,我们使用线程同步栅栏的方式完成线程之间的同步,生成的 Java 文件主要包括如下两个 Java 类.

(1) 输入输出类.对于 S-CGA 中每一个输入动作,对应生成一个 read 方法,每个输出动作则对应生成一个 write 方法.在线程类执行前,调用 read 方法读取输入数据,在线程类执行结束后,调用 write 方法写入输出数据.通过单独生成输入输出类,保证不同任务划分算法生成的线程类可以重用输入输出类.

(2) 线程类.线程类中包含了若干子类,如 class *thread*_15 等.子类实现了 Runnable 方法,对应于虚拟多线程中单个线程.在子类中重写 run 函数:等待所依赖的事件发生,执行条件语句,通知其他线程.

以图 10 中虚拟多线程为例,我们给出对应多线程 Java 代码中部分线程:*thread*_15 和 *thread*_17,如图 12 所示.首先 *thread*_15 调用 Wait 方法,进入阻塞状态,当所依赖的线程全部执行完之后,解除 *thread*_15 的阻塞状态.if 语句执行结束后,调用 countDown 方法通知其他线程 *thread*_15 已执行结束.类似地,如果 *thread*_1 已经执行完成,则下一步可以执行 *thread*_17.

多线程 Java 代码(*thread*_15)	多线程 Java 代码(*thread*_17)

class *thread*_15 implements Runnable{
public void run(){
try {
cdl15.await();
} catch (InterruptedException e){
e.printStackTrace();
}
if(*ID*_4){
*ID*_5=x_1;
*ID*_6=*ID*_4 && *ID*_5;
}
cdl17.countDown();
cdl19.countDown();
cdl21.countDown();}}
	class *thread*_17 implements Runnable{
	public void run(){
	try {
	cdl17.await();
	} catch (InterruptedException e){
	e.printStackTrace();
	}
	if(*ID*_6){
	s_1=y_1;
	x=s_1;
	}
	cdl26.countDown();
	}
	}

Fig.12 Example of multi-thread Java code

图 12 多线程 Java 代码示例

4 工具和实例分析

MinSIGNAL 多线程代码生成器基于 OCAML 编程实现,代码生成器采用模块化思想,包括已有串行代码生成器前端和扩展多线程代码生成器后端.本节将主要介绍 MinSIGNAL 多线程代码生成器的工具设计与实现以及多线程代码在多核处理器上的实验分析.

4.1 工具实现和分析

多线程代码生成器的主要编译步骤及其对应的 OCAML 代码数量统计见表 4.

Table 4 Main steps of MinSIGNAL parallel code generator

表 4 MinSIGNAL 多线程代码生成器主要步骤

	代码生成器步骤		执行功能	OCAML 代码(行)
前端	1.用户程序标准化		将用户输入的 SIGNAL 程序转换为 k-SIGNAL 程序	300+
	2.S-CGA		将 k-SIGNAL 程序转换为 S-CGA 程序	300+
	3.时钟演算		消解 S-CGA 中时钟等式,并生成时钟等价类,替换 S-CGA 中对应时钟	400+
后端	4.时钟数据依赖图 CDDG		将 S-CGA 转换成 CDDG 以分析 S-CGA 时钟数据依赖关系	100+
	5.任务划分	拓扑划分	基于拓扑排序思想对 CDDG 进行任务划分,得到划分结果 topoPartition	100+
		优化划分	优化 topoPartition,包括合并线程等,生成优化结果 topoOptimization	100+
		流水线划分	针对 topoPartition 定义流水线阶段,增加流水线中间变量以及通信函数,得到最终的划分结果 topoPipeline	150+
	6.虚拟多线程		从 topoOptimization/topoPipeline 生成虚拟多线程 VMT	150+
	7.多线程代码	C	根据 VMT 生成可执行多线程 C 程序	300+
		Java	根据 VMT 生成可执行多线程 Java 程序	300+

工具开发环境为基于 Eclipse 平台的 OcaIDE 插件环境.如图 13 所示,左侧为文档结构,每个文件对应一个执行功能.我们在 main.ml 文件中配置 MinSIGNAL 代码生成器执行步骤,并通过配置文件设置同步语言源文件路径、代码生成器执行文件路径等参数,运行配置文件(如图 14 所示),从而生成多线程代码.

如图 15 所示,目标程序通过从输入信号对应的文件 Input.txt 中读取数据,执行计算结束后将输出信号保存到文件 Output.txt 中.

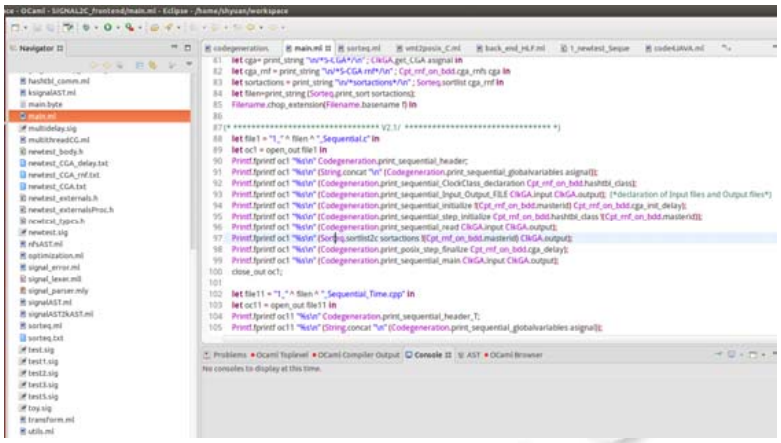

Fig.13 Code of main.ml

图 13 main.ml 代码

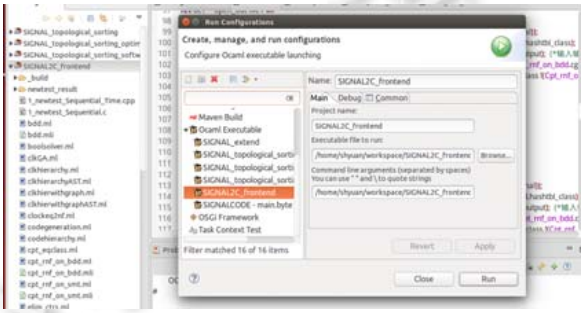

Fig.14 Configuration information of MinSIGNAL code generator

图 14 MinSIGNAL 代码生成器配置信息

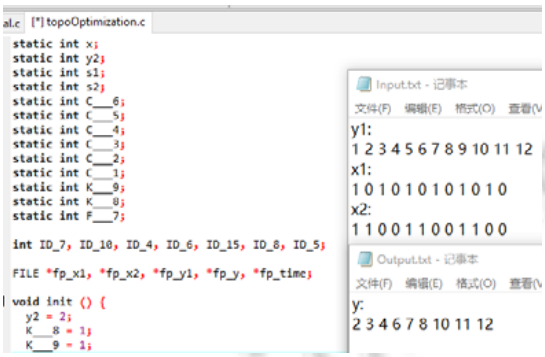

Fig.15 Example value of input signals and output signals

图 15 示例输入和输出信号值

4.2 实验与分析

本实验的目的在于通过实验分析基于拓扑排序优化算法和基于流水线方式的任务划分是否提高目标代码的执行效率,以支持比较基于不同划分方法下自动生成的程序的质量.我们测试基于不同划分算法生成的多线程代码,具体包括:使用基于拓扑排序划分生成的多线程代码(C_1/JAVA_1),以及在拓扑排序划分的基础上分别使用基于优化拓扑排序划分生成的多线程代码(C_2/JAVA_2)和基于流水线划分方式的多线程代码(C_3_*i*/JAVA_3_*i*),其中,流水线条数 *i* 分别设置为 3/4/6/12 条.C_1/JAVA_1 的运行结果作为基准,用于和后两种划分方式进行比较.整个测试环境参数包括操作系统 Win10 64bit、8 核 core i7-6700 CPU 3.40GHz、8G RAM、

C 编译环境 Dev_C++(TDM-GCC 4.8.1)和 Java 编译环境 Eclipse Oxygen(JDK1.8).

实验选取 3 个 SIGNAL 测试程序:测试程序 1 为例 1 中 Count 程序;测试程序 2 为数学中幂运算计算程序,输入两个整型(integer)参数分别是底数和指数,输出幂运算的结果;测试程序 3 为模拟采样,对输入的波形(整型值序列),根据设置不同的采样要求(假设有两种布尔条件),分别输出对应的波形.实验通过在不同 CPU 核数下运行不同 SIGNAL 测试程序生成的多线程 C 和 Java 代码,计算其循环执行 1 000 次的平均执行时间,见表 5.

平均执行时间反映多线程代码执行效率.图 16 给出表 5((1)~(3),共 3 张表)对应的折线图,横轴为 CPU 核数,纵轴为不同目标代码在给定 CPU 核数下的平均执行时间.如图 16 所示,在 CPU 核心数相同的情况下,基于流水线方式的划分算法生成的多线程代码(C/Java)执行效率最高,而采用基于拓扑排序划分优化算法次之,但仍比基于拓扑排序划分算法对应的执行效率要高.同时,由图 16 可知,在给定任务划分算法和 CPU 核数的情况下,生成的多线程 C 程序的执行效率高于生成的多线程 Java 程序.这可能是因为二者的多线程机制存在差异,C 程序直接调用 Windows 底层程序,而 Java 程序是借助于 JVM 实现多线程机制.因此,不同目标语言的选取也会影响多线程代码自动生成方法所生成目标程序的执行效率.

此外,通过分析四核 CPU 和八核 CPU 下对应的平均执行时间可知,四核 CPU 对应的执行时间更快,即并不是 CPU 核心数越高,程序执行速度越快.这可能是由于随着 CPU 核数的提升,不同核之间调度和数据交换花费的时间更多,并且 Cache 访问冲突也会造成执行时间的增加.因此,在嵌入式系统的设计过程中,在考虑软件建模的同时,也需要考虑适配硬件平台,避免造成资源浪费.

Table 5 Testing results on multi-core platform (1)

表 5 多核执行平台下的测试结果(1)

平均执行时间(ms) C/Java \ CPU 核数	单核	双核	四核	八核
C_1/JAVA_1	21366/27666	14968/24271	8765/16114	9449/17509
C_2/JAVA_2	17947/23172	11507/19503	7982/14069	8289/14625
C_3_3/JAVA_3_3	6183/8585	4696/6902	2955/5166	3072/5236
C_3_4/JAVA_3_4	5354/6400	3560/5070	2284/4121	2467/4033
C_3_6/JAVA_3_6	3673/4332	2624/3631	1593/2841	1714/2780
C_3_12/JAVA_3_12	2910/3262	1712/2048	1060/1681	1078/1610

Table 5 Testing results on multi-core platform (2)

表 5 多核执行平台下的测试结果(2)

平均执行时间(ms) C/Java \ CPU 核数	单核	双核	四核	八核
C_1/JAVA_1	18062/28369	17059/23527	10525/16797	8320/15196
C_2/JAVA_2	14838/20434	11767/17538	7482/13326	6363/12659
C_3_3/JAVA_3_3	6579/8673	4861/6815	3219/5520	2544/5000
C_3_4/JAVA_3_4	5572/6639	4465/5388	2617/4305	2216/4036
C_3_6/JAVA_3_6	4564/5058	3469/4089	1949/3319	1614/3120
C_3_12/JAVA_3_12	3564/3430	2770/2856	1491/2283	1246/2178

Table 5 Testing results on multi-core platform (3)

表 5 多核执行平台下的测试结果(3)

平均执行时间(ms) C/Java \ CPU 核数	单核	双核	四核	八核
C_1/JAVA_1	12997/17964	11928/15514	6703/12034	5742/10838
C_2/JAVA_2	12405/17896	9832/13369	6028/10729	5164/9952
C_3_3/JAVA_3_3	6045/6703	4031/6040	2583/4606	2121/4214
C_3_4/JAVA_3_4	5005/5441	3688/4708	2312/3682	1860/3320
C_3_6/JAVA_3_6	4098/4175	3163/3501	1785/2693	1388/2558
C_3_12/JAVA_3_12	3300/3022	2529/2293	1334/1878	996/1831

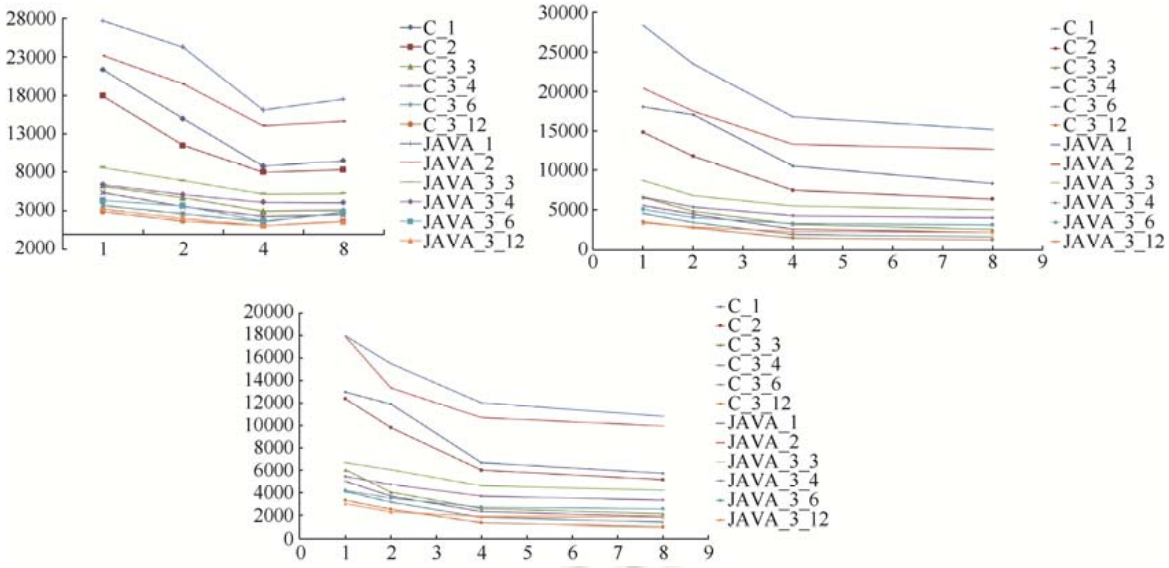

Fig.16 Testing results on multi-core platform (line chart1~3)

图 16 多核执行平台下的测试结果(折线图 1~折线图 3)

综上所述,通过分析基于不同划分算法生成的多线程 C/Java 代码在多核平台上实验的数据,可以得出以下结论.

(1) 基于拓扑排序优化算法和基于流水线方式的任务划分算法缩短了目标代码执行时间,其中,流水线划分算法较为明显地提高了执行速度.

(2) 不同目标语言的多线程代码执行速度存在差异,在实际生成代码时需要综合考虑目标语言以及多线程机制等因素.

(3) 当考虑同步模型生成多线程代码时,需要考虑部署在哪种多核平台上,以保证资源的合理利用.

本文基于 OCMAL 实现同步语言 SIGNAL 的多线程代码自动生成工具.在已有 SIGNAL 编译器研究工作中:例如,与原有 SIGNAL 编译器 Polychrony[26]相比,本文的 S-CGA 能够支持多种同步语言的集成;文献[14]主要考虑基于 GALS 的多线程代码生成,生成的线程粒度较大,本文主要考虑更细粒度的线程,以便寻找到更多的并行信息,目前已实现与体系结构分析和设计语言 AADL 的混合建模;与文献[31]相比,该文献主要考虑同步语言的跨平台代码生成,在任务划分中仅使用拓扑排序划分算法,且时钟信息需要后期单独增加,相比而言,本文考虑多种划分算法,优化目标代码的执行效率.此外,本文的未来目标是基于 Coq 多线程代码生成器的验证工作.

5 相关工作

同步语言的串行代码生成研究已经比较成熟.随着多核处理器的发展,同步语言的多线程代码生成成为研究热点.已有研究大致可以分为如下 3 类.

- 基于 GALS 的多线程代码生成,即系统由具有不同时钟的多个进程组成,而每个进程是单时钟的,目前所有同步语言基本都支持此类的多线程代码生成.
- 基于单进程的多线程代码生成,主要是针对 SIGNAL 语言以及其他单时钟同步语言的扩展,这是由于单个进程中的所有信号变量也可以具有不同时钟.
- 考虑硬件平台特性,涉及到多核架构或时间可预测处理器的执行及程序最坏执行时间分析.

(1) 基于 GALS 的多线程代码生成

同步语言支持进程间的并发操作,例如 QUARTZ 和 ESTEREL 提供同步并发运算符‘||’以支持进程间的并

发,LUSTRE 使用';'表示不同节点的并行执行,SIGNAL 中不同进程间的组合操作'|'也隐含着并行执行[32].同时,在分布式系统上,由于进程间通信受到物理载体的影响而产生不可预测的延迟[13],导致不同进程间的时钟可能不满足同步关系,被称为全局异步局部同步 GALS 系统.

文献[14]提出一种非侵入式的多线程代码生成方法,即在不改变已有 SINGAL 编译器 Polychrony 结构的基础上,利用其串行编译功能生成多个独立的进程,并基于主控函数完成调度,实现进程之间的并行执行.文献[15]提出从实际工业需求中导出功能行为等价的 Heptagon 程序[33](一种类 Lustre 同步语言)并进一步生成多线程代码.与文献[14]所提方案类似,首先基于 Heptagon 编译器生成多个串行节点,然后在集成规约中描述任务间通信和同步.通过构建节点间的依赖图,利用串行调度来保证确定性并发语义.除功能需求外,该研究还考虑了非功能性需求(如周期等实时性质)的代码生成.

此外,文献[34]提出从高层建模语言(AADL,SysML,SystemC 等)转换为 SIGNAL 同步程序并进一步生成多线程代码的方法.其中,同步程序基于 Weekly Endochronous 理论[35]:即程序中允许多个根时钟.在多线程生成过程中,采用符号化原子表达式和合并分支条件的策略约减线程数目.不过,其同步程序中仅考虑有限的数据类型(事件、布尔和枚举类型)和布尔操作(相等和取反操作).

多个进程之间并行划分的"粒度"相对较大[14],但在同步语言模型中,单个进程内部也往往存在潜在的并行执行信息.尤其是可以自然表达分布式系统的 SIGNAL 语言,其语法的基本结构中包含多时钟操作,可以比较自然地支持在进程内部描述并行执行操作.

(2) 基于单进程的多线程代码生成

文献[26]中介绍了同步语言 SIGNAL 编译器 Polychrony 中的代码生成策略,在多线程生成方面分为静态调度和动态调度.静态调度下的分簇(多线程)代码生成:将生成的代码划分到簇中来模拟调度图的结构.分簇的原则是:将调度图划分为对那些依赖于完全相同的输入集合进行计算.只要一个簇的所有输入都有确定值,那么这个簇就可以被自动执行,一个簇的时钟取决于簇中所有信号在时钟树上的公共父节点.动态调度下的分簇(多线程)代码生成:按任务实现分簇,并生成任务之间的同步信息.任务由簇来生成,主要思想是使用信号量的方式并行执行各个任务.然而,这种方式下生成的多线程数目太多,并且使用特定的线程库来生成仿真代码.

文献[29,30]是对纯同步语言 Quartz 编译过程的多线程代码的研究.文献[29]提出同步卫式动作(clocked guarded action,简称 CGA)到基于 OpenMP 的多线程 C 程序的编译方案.作者提出"垂直划分"的概念,即从数据依赖图中抽取出独立的线程,并生成对应的 C 代码.文献[30]引入软件流水线的概念,提到的软件流水线方法包含:分析卫式动作之间的依赖关系、创建流水线变量存储各阶段之间的中间结果以及卫式动作到流水线之间的转换.这种方法被称为"水平划分".

文献[31]对 SIGNAL 多线程代码生成进行了研究,提出基于方程依赖图(equation dependency graph,简称 EDG)的 OpenMP 并行代码生成方法.其中的并行代码生成算法主要是对数据依赖图进行拓扑排序,生成多个链表:多个链表之间顺序执行、单个链表内部节点并行执行.不过,文献[31]使用拓扑排序作为任务划分算法,没有考虑优化工作,而且需要在划分之后对划分结果单独加入时钟信息.而本文使用的 CDDG 中包含时钟依赖关系,即在划分过程中已经考虑时钟信息.

本文主要考虑单进程的多线程代码生成方法,同时给出优化策略和流水线方式的任务划分方法,能够较好地约减线程数目以及提高目标多线程代码执行效率.

(3) 多核架构执行及 WCET 分析

ITEA 3 计划中的(affordable safe & secure mobility evolution,简称 ASSUME)项目[15,36]提出一种用于在多核/众核架构上提供可信的嵌入式软件工程方法,包括从 LUSTRE 同步模型到自动生成并行代码并在多核/众核平台上执行.ASSUME 项目扩展了 SCADE 工具的 KCG 编译器,以支持生成面向 MPPA 众核架构[37]的并行代码,但是当前的并行信息是通过用户来驱动的,即用户指定程序中并行执行区域.此外,在代码生成过程中提供每个任务在实时调度期间所需的属性(如 WCET、内存访问的最坏次数等).

法国国家航天航空研究中心 ONERA 的 SchedMCore 环境[38,39]为形式化多处理器调度分析和实验性多核

运行时基础架构提供了一套验证和执行工具.SchedMCore将同步语言Prelude[40]编译生成一组相互通信的周期任务(包含信息有:任务的周期、WCET、首次到达时间和 Deadline),并基于工具自带的不同多核调度策略实现任务在多核架构上的调度.文献[41]中给出一个基于 SchedMCore 设计并运行在多核/众核架构上的航空经度控制器案例.

德国 Kaiserslautern 工业大学嵌入式系统小组(http://es.cs.uni-kl.de/)基于同步语言 QUARTZ 开发了一个用于并行嵌入式系统的规约、验证和实现的框架 AVEREST(http://www.averest.org/).该框架可同时生成硬件电路逻辑代码和多线程代码,并且支持软硬件分析[42].此外,他们还研究了基于指令集并行(instruction level parallelism,简称 ILP)的同步语言 SCAD 最优化代码生成,包括使用回答集编程(answer set programming,简称 ASP)处理 SCAD 代码生成过程中的最优资源/时间约束调度问题[43],以及利用 SMT 求解器生成实现最大化指令级并行(给定处理器单元数目)的最优化代码[44]等.

综上所述,同步语言多线程代码生成涉及到多核硬件执行平台以及考虑 WCET 分析.我们在已有工作[18]中给出在同步语言代码生成过程中的时间可预测多核体系结构模型及软硬件映射方法.而本文主要侧重同步语言多线程代码生成方法及其实现,提出任务划分优化策略以及基于流水线方式的划分以提高目标代码的执行速度.

6 总结与展望

同步语言广泛用于安全关键嵌入式系统建模与验证,近年来,同步语言的多线程代码生成成为学术界的一个研究热点.本文提出一种基于 OCAML 的同步语言 SIGNAL 多线程代码生成方法和工具.首先将 SIGNAL 程序转换为经过时钟演算的 S-CGA 中间程序;之后将 S-CGA 中间程序转换为时钟数据依赖图以分析依赖关系;然后对时钟数据依赖图进行拓扑排序划分,并针对划分结果提出优化算法和基于流水线方式的任务划分方法;最后将划分结果转换为虚拟多线程结构,然后进一步生成可执行多线程 C/Java 代码.通过在多核处理器上的实验验证了本文提出方法的有效性.

编译器的形式化验证是一项重要工作,例如:CompCert 编译器(C 编译器)[45]、清华大学 L2C 项目(LUSTRE 编译器)[27,28]以及 Vélus 编译器(LUSTRE 编译器)[46].我们已经给出 SIGNAL 多线程代码生成器前端的语义保持证明,下一步工作将基于 Coq 完成代码生成器后端的语义保持证明.其次,扩展同步语言以支持描述实时性质,以及考虑在时间可预测多核处理器(如 Patmos[47])上执行多线程代码并进行 WCET 分析也是未来一项重要工作;最后,针对复杂嵌入式系统,我们正在开展嵌入式实时系统体系结构分析与设计语言 AADL(描述系统架构)[48]和同步语言(描述 AADL 单构件内的功能)的混合建模方法研究.

致谢 感谢匿名评审专家给予的宝贵意见.另外,感谢法国法国国家信息与自动化研究所(INRIA)Jean-Pierre Talpin 教授给予了很多重要的建议.

References:

[1] Leveson N. Engineering a Safer World: Systems Thinking Applied to Safety. MIT Press, 2011.

[2] Parkinson P. Safety, security and multicore. In: Advances in Systems Safety. London: Springer-Verlag, 2011. 215–232.

[3] Quad-core LEON4 next generation microprocessor evaluation board. 2014. http://www.gaisler.com/index.php/products/boards/gr-cpci-leon4-n2x

[4] RTCA DO-178C. Software considerations in airbone systems and equipment certification. Report, RTCA, 2011. [doi: 10.1145/1869542.1869558]

[5] RTCA DO-331. Model-based development and verification supplement to DO-178C and DO-278A. Report, RTCA, 2011.

[6] RTCA DO-333. Formal methods supplement to DO-178C and DO-278A. Report, RTCA, 2011.

[7] Boussinot F, de Simone R. The Esterel language. Proc. of the IEEE, 1991,79(9):1293–1304.

[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.

[9] SCADE. http://www.esterel-technologies.com/products/scade-suite/

[10] Benveniste A, Le Guernic P, Jacquemot C. Synchronous programming with events and relations: The signal language and its semantics. Science of Computer Programming, 1991,16:103–149.

[11] Schneider K. The synchronous programming language QUARTZ. Internal Report, Department of Computer Science, University of Kaiserslautern, 2010.

[12] Sovani S. Simulation accelerates development of autonomous driving. ATZ Worldwide, 2017,119(9):24–29.

[13] Doucet F, Menarini M, Krüger IH, *et al*. A verification approach for GALS integration of synchronous components. Electronic Notes in Theoretical Computer Science, 2006,146(2):105–131.

[14] Jose BA, Patel HD, Shukla SK, Talpin JP. Generating multi-threaded code from polychronous specifications. Electronic Notes in Theoretical Computer Science, 2009,238(1):57–69.

[15] Souyris J, Didier K, Potop D, *et al*. Automatic parallelization from lustre models in avionics. In: Proc. of the ERTS2 2018, the 9th European Congress Embedded Real-Time Software and Systems. 2018. 1–4.

[16] Yang Z, Bodeveix JP, Filali M. Towards a simple and safe objective CAML compiling framework for the synchronous language SIGNAL. Frontiers of Computer Science, 2018, 1–20.

[17] Yang Z, Bodeveix JP, Filali M, *et al*. Towards a verified compiler prototype for the synchronous language SIGNAL. Frontiers of Computer Science, 2016,10(1):37–53.

[18] Yang ZB, Zhao YW, Huang ZQ, Hu K, Ma DF, Bodeveix JP, Filali M. Time-predictable multi-threaded code generation with synchronous languages. Ruan Jian Xue Bao/Journal of Software, 2016,27(3):611−632 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4984.htm [doi: 10.13328/j.cnki.jos.004984]

[19] Gamatié A. Designing Embedded Systems with the SIGNAL Programming Language. Springer-Verlag, 2010.

[20] Besnard L, Gautier T, Le Guernic P. SIGNAL V4 Reference Manual. 2010.

[21] Pnueli A, Siegel M, Singerman F. Tanslation validation. In: Proc. of the TACAS 98. 1998. 151–166.

[22] 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.

[23] The Coq Proof Assistant. https://coq.inria.fr/about-coq

[24] Brandt J, Gemunde M, Schneider K, Shukla SK, Talpin J-P. Embedding polychrony into synchrony. IEEE Trans. on on Software Engineering, 2013,39(7):917–929.

[25] Brandt J, Gemunde M, Schneider K, Shukla SK, Talpin J-P. Integrating system descriptions by clocked guarded actions. In: Proc. of the FDL. IEEE, 2011. 1–8.

[26] Besnard L, Gautier T, Talpin JP. Code generation strategies in the Polychrony environment [Ph.D. Thesis]. INRIA, 2009.

[27] Shi G, Wang SY, Dong Y, Ji ZY, Gan YK, Zhang LB, Zhang YC, Wang L, Yang F. Construction for the trustworthy compiler of a synchronous data-flow language. Ruan Jian Xue Bao/Journal of Software, 2014,25(2):341−356 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4542.htm [doi: 10.13328/j.cnki.jos.004542]

[28] Liu Y, Gan YK, Wang SY, Dong Y, Yang F, Shi G, Yan X. Trustworthy translation for eliminating high-order operation of a synchronous dataflow language. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):332−347 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4785.htm [doi: 10.13328/j.cnki.jos.004785]

[29] Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In: Proc. of the Design, Automation & Test in Europe Conf. & Exhibition (DATE). IEEE, 2010. 949–952.

[30] Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In: Proc. of the MBMV. 2010. 11–20.

[31] Hu K, Zhang T, Shang LH, Yang ZB, Talpin JP. Parallel code generation from synchronous specification. Ruan Jian Xue Bao/Journal of Software, 2017,28(7):1698−1712 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5056.htm [doi: 10.13328/j.cnki.jos.005056]

[32] Le Guernic P, Talpin JP, Le Lann JC. Polychrony for system design. Journal of Circuits, Systems, and Computers, 2003,12(3): 261–303.

[33] The Heptagon language and compiler. 2018. http://heptagon.gforge.inria.fr

[34] Papailiopoulou V, Potop-Butucaru D, Sorel Y, De Simone R, Besnard L, Talpin JP. From design-time concurrency to effective implementation parallelism: The multi-clock reactive case. In: Proc. of the Electronic System Level Synthesis Conf. (ESLsyn). 2011. 1–6.

[35] Potop-Butucaru D, Caillaud B, Benveniste A. Concurrency in synchronous systems. Formal Methods in System Design, 2006,28(2): 111–130.

[36] ITEA 3 programme 14014 ASSUME project. 2018. https://itea3.org/project/assume.html

[37] De Dinechin BD, Van Amstel D, Poulhiès M, *et al*. Time-critical computing on a single-chip massively parallel processor. In: Proc. of the Conf. on Design, Automation & Test in Europe. European Design and Automation Association, 2014. 97.

[38] SchedMCore User's Manual. 2018. http://sites.onera.fr/schedmcore/sites/sites.onera.fr.schedmcore/files/schedmcore-user-manual.pdf

[39] Cordovilla M, Boniol F, Forget J, *et al*. Developing critical embedded systems on multicore architectures: The Prelude-SchedMCore toolset. In: Proc. of the 19th Int'l Conf. on Real-time and Network Systems. 2011.

[40] Forget J, Boniol F, Lesens D, *et al*. A multi-periodic synchronous data-flow language. In: Proc. of the 11th IEEE High Assurance Systems Engineering Symp., HASE 2008. IEEE, 2008. 251–260.

[41] Pagetti C, Saussié D, Gratia R, *et al*. The ROSACE case study: From simulink specification to multi/many-core execution. In: Proc. of the 20th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2014. 309–318.

[42] Schneider K, Brandt J. Quartz: A synchronous language for model-based design of reactive embedded systems. In: Handbook of Hardware/Software Codesign. 2017. 29–58.

[43] Dahlem M, Bhagyanath A, Schneider K. Optimal scheduling for exposed datapath architectures with buffered processing units by ASP. 2018. [doi: 10.1017/S1471068418000170]

[44] Bhagyanath A, Schneider K. Exploring the potential of instruction-level parallelism of exposed datapath architectures with buffered processing units. In: Proc. of the Int'l Conf. on Application of Concurrency to System Design. IEEE, 2017. 106–115.

[45] Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009,52(7):107–115.

[46] Bourke T, Brun L, Dagand PÉ, *et al*. A formally verified compiler for Lustre. ACM SIGPLAN Notices, 2017,52(6):586–601.

[47] Schoeberl M, Silva C, Rocha A. T-CREST: A time-predictable multi-core platform for aerospace applications. In: Proc. of the Data Systems In Aerospace (DASIA 2014). 2014.

[48] Yang ZB, Pi L, Hu K, Gu ZH, Ma DF. AADL: An architecture design and analysis language for complex embedded real-time systems. Ruan Jian Xue Bao/Journal of Software, 2010,21(5):899−915 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3700.htm [doi: 10.3724/SP.J.1001.2010.03700]

附中文参考文献:

[18] 杨志斌,赵永望,黄志球,胡凯,马殿富,Bodeveix JP,Filali M.同步语言的时间可预测多线程代码生成方法.软件学报,2016,27(3):611−632. http://www.jos.org.cn/1000-9825/4984.htm [doi: 10.13328/j.cnki.jos.004984]

[27] 石刚,王生原,董渊,嵇智源,甘元科,张玲波,张煜承,王蕾,杨斐.同步数据流语言可信编译器的构造.软件学报,2014,25(2):341−356. http://www.jos.org.cn/1000-9825/4542.htm [doi: 10.13328/j.cnki.jos.004542]

[28] 刘洋,甘元科,王生原,董渊,杨斐,石刚,闫鑫.同步数据流语言高阶运算消去的可信翻译.软件学报,2015(2):332−347. http://www.jos.org.cn/1000-9825/4785.htm [doi: 10.13328/j.cnki.jos.004785]

[31] 胡凯,张腾,杨志斌,Jean-Pierre Talpin.面向同步规范的并行代码自动生成方法研究.软件学报,2017,28(7):1698−1712. http://www.jos.org.cn/1000-9825/5056.htm [doi: 10.13328/j.cnki.jos.005056]

[48] 杨志斌,皮磊,胡凯,顾宗华,马殿富.复杂嵌入式实时系统体系结构设计与分析语言:AADL.软件学报,2010,21(5):899−915. http://www.jos.org.cn/1000-9825/3700.htm [doi: 10.3724/SP.J.1001.2010.03700]

杨志斌(1982－),男,江西吉安人,博士,副教授,CCF 专业会员,主要研究领域为安全关键嵌入式软件,形式化方法.

袁胜浩(1994－),男,学士,CCF 学生会员,主要研究领域为形式化方法,软件工程.

谢健(1988－),男,硕士生,CCF 专业会员,主要研究领域为形式化方法,系统安全性分析,服务计算.

周勇(1975－),男,博士,副教授,CCF 专业会员,主要研究领域为形式化方法,软件工程.

陈哲(1981－),男,博士,副教授,CCF 专业会员,主要研究领域为形式化方法,软件工程,软件验证.

薛垒(1982－),男,高级工程师,主要研究领域为嵌入式软件设计验证.

Bodeveix Jean-Paul(1963－),男,博士,教授,博士生导师,主要研究领域为实时系统,形式化方法.

Filali Mamoun(1957－),男,博士,高级研究员,博士生导师,主要研究领域为实时系统,形式化方法.