软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(3):611−632 [doi: 10.13328/j.cnki.jos.004984]

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

同步语言的时间可预测多线程代码生成方法*

杨志斌[1,2,3], 赵永望[2], 黄志球[1,3], 胡 凯[2], 马殿富[2], 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 210016)

[2](软件开发环境国家重点实验室(北京航空航天大学),北京 100191)

[3](软件新技术与产业化协同创新中心,江苏 南京 210016)

[4](IRIT, Université de Toulouse, Toulouse, France)

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

摘 要: 能够提供更强计算能力的多核处理器将在安全关键系统中得到广泛应用,但是由于现代处理器所使用的流水线、乱序执行、动态分支预测、Cache 等性能提高机制以及多核之间的资源共享,使得系统的最坏执行时间分析变得非常困难.为此,国际学术界提出时间可预测系统设计的思想,以降低系统的最坏执行时间分析难度.已有研究主要关注硬件层次及其编译方法的调整和优化,而较少关注软件层次,即,时间可预测多线程代码的构造方法以及到多核硬件平台的映射.提出一种基于同步语言模型驱动的时间可预测多线程代码生成方法,并对代码生成器的语义保持进行证明;提出一种基于 AADL(architecture analysis and design language)的时间可预测多核体系结构模型,作为研究的目标平台;最后,给出多线程代码到多核体系结构模型的映射方法,并给出系统性质的分析框架.

关键词: 安全关键系统;多核处理器;时间可预测;同步语言;AADL(architecture analysis and design language)

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/4984.htm

Time-Predictable Multi-Threaded Code Generation with Synchronous Languages

YANG Zhi-Bin[1,2,3], ZHAO Yong-Wang[2], HUANG Zhi-Qiu[1,3], HU Kai[2], MA Dian-Fu[2], Jean-Paul BODEVEIX[4], Mamoun FILALI[4]

[1](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

[2](State Key Laboratory of Software Development Environment (BeiHang University), Beijing 100191, China)

[3](Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210016, China)

[4](IRIT, Université de Toulouse, Toulouse, France)

* 基金项目: 国家自然科学基金(61502231); 国家重点基础研究发展计划(973)(2014CB744904); 江苏省自然科学基金(BK20150753); 软件开发环境国家重点实验室开放课题(SKLSDE-2015KF-04); 航空科学基金(2015ZC52027); 中国博士后科学基金

Foundation item: National Natural Science Foundation of China (61502231); the National Key Basic Research Program of China (973) (2014CB744904); Natural Science Foundation of Jiangsu Province (BK20150753); the Project of the State Key Laboratory of Software Development Environment of China (SKLSDE-2015KF-04); the Avionics Science Foundation of China (2015ZC52027); China Postdoctoral Science Foundation

收稿时间: 2015-07-15; 修改时间: 2015-10-20; 采用时间: 2015-11-27; jos 在线出版时间: 2016-01-05

CNKI 网络优先出版: 2016-01-05 16:39:58, http://www.cnki.net/kcms/detail/11.2560.TP.20160105.1639.010.html

Abstract: Multi-core processors are being widely used in safety-critical systems. Unfortunately, the introduction of performance-enhancing architectural elements, such as pipelines, out-of-order execution, dynamic branch prediction, caches and inter-cores resource-sharing, make WCET (worst-case execution time) analysis of a system become more difficult. Thus, time-predictable system design is established to meet the challenge of building systems for which WCET can be statically and easily analyzed. At the software level, this paper proposes a time-predictable multi-threaded code generation based on synchronous-model development. At the platform level, it presents a time-predictable multi-core architecture model in AADL (architecture analysis and design language), and then maps the multi-threaded code to this model. Real-time specifications propagate down in the system hierarchy. As a result, the proposed method integrates time predictability across several design layers, and finally reduces the complexity of WCET analysis.

Key words: safety-critical system; multi-core processor; time predictability; synchronous language; AADL (architecture analysis and design language)

安全关键系统(safety-critical system)广泛应用于航空电子、航天器、核能、汽车控制等关键信息领域.由于功能和非功能需求的发展,这类系统对处理器计算性能的要求日趋提高.相比单核处理器,能够提供更强计算能力的多核处理器将在安全关键系统中得到广泛应用.然而,在安全关键系统中使用多核处理器仍然面临诸多挑战,例如安全性(safety)、保密性(security)以及时间可预测性(time predictability)等.保证程序执行时间的可预测性是其中一个主要挑战问题[1−4],这是因为安全关键系统对实时性约束有非常严格的要求,系统的关键任务如果不能满足时限约束(deadline),可能导致灾难性的后果.而所谓时间可预测,即应用程序在目标处理器上的最坏执行时间(worst-case execution time,简称 WCET)可以线下分析(off-line analysis),或者说可以静态确定,从而使得在设计阶段就可以确定任务的时限约束是否得到满足[1].

一个应用程序的 WCET 往往受到多个系统层次的影响,例如应用程序、编译器、操作系统、处理器等.而现代处理器所使用的性能提高机制,如 Cache、流水线、乱序执行、动态分支预测等,使得 WCET 分析变得更加复杂和困难.业界已有一些单核处理器上的 WCET 分析基本方法和工具[3]:分析方法大致分为两类——静态分析和测量;而 WCET 分析工具包括商业化工具 aiT[5],RapiTime[6],Bound-T 以及学术界研究与开发的工具 OTAWA[7],Choronos[8],Heptane,Sweet,Metamoc,TuBound 等.多核处理器的资源共享机制(共享 Cache、共享总线及共享主存)进一步增加了 WCET 分析的难度,这是因为执行在多核处理器上的任务的时间行为与共享资源的仲裁机制以及其他任务的资源占用情况有关.例如:由于总线带宽的访问冲突,一条指令执行的延迟可能会增加甚至增加到不可界定(unbounded);由于共享 Cache 的访问冲突,可能导致非常多次的 Cache 不命中.我们称此为时间干扰问题(timing interferences).

为解决多核处理器带来的问题,学术界产生了两种研究思路:

1) 延续传统 WCET 计算方式,研究由于访问冲突带来的时间延迟的计算方法,并试图扩展已有的 WCET 分析工具.例如,新加坡国立大学提出了一些针对性的计算方法[9,10],并扩展了其 WCET 分析工具 Chronos,以支持多核处理器.但由于共享资源可能导致非常庞大的硬件状态,使得 WCET 分析面临状态空间爆炸问题;

2) 设计时间可预测多核处理器,这样可以降低运行在此类处理器上的程序的 WCET 分析难度.2004 年,欧洲科学院院士 Wilhelm 给出了构造时间可预测嵌入式系统(基于单核处理器)的基本建议[1],如超标量流水线使用按序执行而不使用乱序执行、使用静态分支预测而不使用动态分支预测、使用 LRU(least recently used,最久未使用)Cache 替换策略等;2007 年,Berkely 大学和 Columbia 大学提出了多硬件线程单核处理器 PRET(精确时间机器)[11],该处理器提供专门的时间机器指令,用于控制线程的执行不能超过给定的时限约束;2010 年开始,欧盟 FP7 框架(7th framework programme)相继支持了 PREDATOR[12],MERASA[13],T-CREST[14],parMERASA[15]等项目,这些项目都是围绕时间可预测多核处理器以及一些诸如考虑 WCET 的编译方法[16]以及时间可预测操作系统[17,18]的研究.

第 2 种研究思路是目前国际上的主流和热点问题.

与 WCET 对应,时间可预测性也体现在多个系统层次上.大多数已有研究主要关注硬件层次(即多核处理器)及其上的编译方法的调整和优化,而较少关注软件层次,即时间可预测多线程代码的构造方法.同步语言[19]能够表达确定性并发行为,具有精确的时间语义,非常适合可预测系统设计.因此,本文提出一种基于同步语言

(SIGNAL)模型驱动的时间可预测多线程代码生成方法 TPCodeGen(time-predictable multi-threaded code generation from synchronous programs),主要包括:

1) 考虑代码生成器的可扩展性,即,将来可以与其他同步语言集成,提出了一种新的中间表达式 S-CGA (synchronous clocked guarded actions),给出了 S-CGA 的形式语法和语义;给出了 SIGNAL 语言到 S-CGA 的转换规则,即代码生成器前端 SIGNAL2SCGA,并证明转换的语义保持;

2) 给出了 S-CGA 到多线程代码的转换,即代码生成器后端 SCGA2TPCode,包括时钟演算、构造数据依赖图、任务划分方法、生成多线程代码等步骤;

3) 多线程代码在多核处理器上的执行时间是可预测的,需要时间可预测多核处理器的支持.AADL (architecture analysis and design language)[20−22]是一种复杂嵌入式实时系统体系结构建模语言标准,对应用软件、运行时环境、硬件平台以及软硬件映射具有丰富的表达和分析能力,且具有可扩展性.因此,提出了一种基于 AADL 的时间可预测多核体系结构模型,作为本文的目标平台.同步语言模型生成时间可预测 AADL 多线程代码,而目标平台也为 AADL 模型,我们将使用 AADL 建模与分析工具 OSATE(http://www.aadl.info/aadl/currentsite/tool/osate-down.html)对该模型进行编译和分析.

确定性并发行为和实时规约贯穿于同步模型、S-CGA 以及多线程代码这 3 个层次.同时,目标平台为时间可预测的多核体系结构模型,并且对时间可预测的软硬件映射进行表达和分析.因此,时间可预测性在多个层次上得到保证,从而降低程序的最坏执行时间的分析难度.

本文第 1 节简要介绍同步假设理论和 SIGNAL 语言的基本概念.第 2 节给出时间可预测代码生成器的总体框架.第 3 节介绍时间可预测代码生成器前端 SIGNAL2SCGA,包括 SIGNAL 语言的抽象语法和形式语义、中间表达式 S-CGA 的语法和形式语义、SIGNAL 到 S-CGA 的转换规则及其语义保持证明.第 4 节介绍时间可预测代码生成器后端 SCGA2TPCode.第 5 节给出时间可预测多核体系结构模型及软硬件映射方法,并给出系统性质分析框架.第 6 节给出相关工作比较.第 7 节是本文的总结和展望.

1 同步语言简介

1.1 同步假设

安全关键系统一般也称为反应式系统(reactive system),因为这类系统会不断地和外部环境进行交互,即,不断地从环境中得到输入、计算,并输出给环境.外部环境包括被系统控制的物理设备、人或其他反应式系统.而同步语言[19]是一种重要的反应式系统设计与分析方法.

同步语言,基于同步假设理论(synchronous hypothesis),如图 1 所示,在离散时间上表达和分析系统的功能行为:在每个逻辑时刻,系统的输入-计算-输出时间为 0,即,将物理时钟抽象,仅仅验证系统的功能(输入/输出关系)是否正确;同时,多任务之间采用同步并发,即,通信时间为 0,多任务并发执行,系统会按照确定性顺序执行,每次结果都相同,也称为确定性并发.因此,同步语言是在平台无关层次上表达、分析和验证系统的功能行为.另外,同步语言不允许使用不可界定的循环和递归,在每个时刻,以非递归的方式表达系统的功能行为,循环则是通过逻辑时刻的次数来表达的,因此,循环次数是可界定的.以上特征使得同步语言非常适合可预测系统设计.另外,也使得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(http://www.esterel-technologies.com/products/scade-suite/)(同步语言 LUSTRE[23]工业界版本及其开发环境)对 A350,A380 的飞控子系统进行建模和代码生成.

目前,有 ESTEREL[24],LUSTRE[23],SIGNAL[25],QUARTZ[26]等同步语言,其中,

- ESTEREL,LUSTRE 以及 QUARTZ 语言使用纯同步模式(perfect synchrony),即,存在一个全局时钟(单时钟);
- 而 SIGNAL 语言使用多态同步模式(polychrony),即,多时钟,能够更自然和方便地表达分布式系统,尤其是全局异步局部同步系统(globally asynchronous locally synchronous,简称 GALS).

随着多核处理器的广泛使用,多态同步模式得到更多关注.如,文献[27−29]给出 SIGNAL 语言的多线程代码

生成方法,但这些研究还没有考虑时间可预测性质.

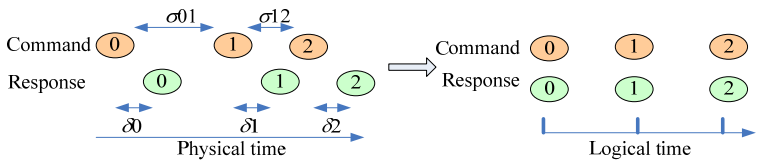

Fig.1 Synchronous hypothesis

图 1 同步假设

1.2 SIGNAL语言

SIGNAL 是由法国 INRIA-Rennes 实验室提出的一种声明式数据流同步语言.按照同步假设理论,在每个逻辑时刻,系统做输入-计算-输出.因此,一个输入或输出为一个取值序列,SIGNAL 语言称之为信号(signal).给定一个逻辑时刻,信号 s 可以存在(present)并带有一个取值 v,或者缺失(absent,记为⊥).信号 s 的逻辑时钟则是该信号存在的所有逻辑时刻的集合,记为 $\hat{s}$.两个信号 s_1 和 s_2 同步,当且仅当在相同的逻辑时刻两个信号同时存在和同时缺失.例如,3 个信号 in_1,in_2 和 out,其时钟分别为 $\widehat{in_1}$,$\widehat{in_2}$ 和 $\widehat{out}$.这里,三者不同步.

in_1	3	⊥	6	⊥	…
in_2	⊥	9	1	8	…
out	⊥	⊥	5	⊥	…

有了输入/输出信号,SIGNAL 语言通过数据流等式的方式来表达信号之间的功能关系(计算)和时钟关系(控制).

首先,SIGNAL 提供了 4 种基本结构(primitive constructs),包括瞬时函数(instantaneous function)、延迟(delay)、条件采样(undersampling)以及确定性合并(deterministic merging).表 1 给出了 4 种基本结构的语法和语义.瞬时函数中的 f 代表各种运算,如代数运算或布尔运算,这些运算都定义在开发工具的运算库中,为 SIGNAL 语言提供丰富的计算表达能力.

Table 1 Syntax and semantics of the dataflow equations

表 1 数据流等式基本结构语法和语义

名称	语法	语义
Instantaneous function	$y:=f(x_1,x_2,\ldots,x_n)$	当 $x_1,\ldots,x_n$ 同时存在,y 的值为 $f(x_1,x_2,\ldots,x_n)$;否则,y 缺位
Delay	$y:=x_1$ \$ *init* c	y 的取值为 x_1 在前一个逻辑瞬间的值,初值为 c,x_1 缺位,则 y 缺位
Undersampling	$y:=x_1$ when x_2	当 x_1 存在,x_2 存在且取值为 true,y 的值为 x_1 的值;否则,y 缺位
Deterministic merging	$y:=x_1$ default x_2	当 x_1 存在,y 的取值为 x_1;当 x_1 不存在而 x_2 存在,y 的取值为 x_2; 当 x_1 和 x_2 同时存在,y 的取值为 x_1;否则,y 缺位

SIGNAL 的基本结构除了表示信号之间的功能关系,还隐含了信号之间的时钟关系.对于瞬时函数和延迟操作,所有信号是同步的,这两种操作也被称为单时钟操作(monoclock operator);对于条件采样,信号 y 存在当且仅当 x_1 存在以及 x_2 存在并且 x_2 的值为真(记做[x_2]);对于合并操作,信号 y 存在当且仅当 x_1 存在或 x_2 存在.由于后两种操作中信号的时钟可以不同,因此被称为多时钟操作(multiclock operator).

其次,SIGNAL 还提供了显式的时钟操作等扩展结构(extended construct),以简化 SIGNAL 程序的表达.如:

- 时钟同步操作:x_1^=x_2;
- 时钟集合操作:并集 x:x_1^+x_2、交集 x:x_1*x_2、补集 x:x_1^–x_2;
- 时钟比较操作: x_1^<x_2, x_1^>x_2.

但所有扩展结构的语义都可以通过基本结构来表达.因此在本文研究中,我们仅考虑基本结构.

最终,SIGNAL 将输入、输出以及用于表达计算的数据流等式组合起来,构成一个进程(process).进程包含两个操作:组合(composition)与局部声明(local declaration),见表 2.

Table 2 Syntax and semantics of the process

表 2 进程基本结构的语法和语义

名称	语法	语义
Composition	$P\|Q$	P 和 Q 为进程,$P\|Q$ 的行为是 P 和 Q 行为的组合
Local declaration	P where $t\_1\ s_1$; $t\_2\ s_2$; … $t\_n\ s_n$; end;	P 为进程,s_1 到 s_n 为定义在 P 中的信号,即,在进程 P 外不可见

另外,SIGNAL 语言有多种形式语义,如:基于踪迹的指称语义(trace semantics)[30,31],即,在全序的时间序列上分析信号之间的功能和时钟关系;基于标签模型的指称语义(tagged model semantics)[31],即,在偏序的时间模型上分析信号之间的功能和时钟关系;结构化操作语义以及基于同步变迁系统(synchronous transition systems,简称 STS)[32]的操作语义等.在文献[33]中,我们基于定理证明器 Coq[34]证明了 SIGNAL 语言的踪迹语义和标签模型语义的等价性.

2 TPCodeGen 总体框架

图 2 给出了时间可预测多线程代码生成器 TPCodeGen 的总体框架.AADL 对应用软件、运行时环境、硬件平台以及软硬件映射具有丰富的表达和分析能力,因此:在软件层次,基于同步语言模型驱动自动生成时间可预测的 AADL 多线程代码;在硬件层次,扩展 AADL 属性集,给出时间可预测 AADL 多核体系结构模型;并基于 AADL 语言对软硬件映射进行表达和分析.最终,构成完整的时间可预测多核嵌入式系统模型.

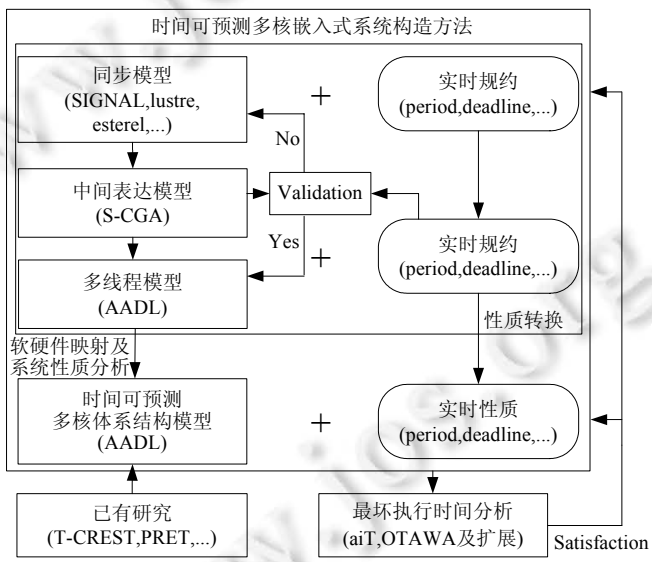

Fig.2 Overview of TPCodeGen

图 2 TPCodeGen 总体框架

软件层次则进一步分为 3 个阶段.

- 同步模型

确定性并发特征使得同步语言非常适合可预测系统设计,而相比其他同步语言,SIGNAL 语言使用多态同步模式,即多时钟,能够更自然和方便地表达分布式系统,尤其是全局异步局部同步系统.

因此,我们选择 SIGNAL 作为主要建模语言.

- 中间模型

近年来,同步语言之间的集成成为一个热点研究问题[35].德国 Kaiserslautern 大学的 Schneider 教授提出同步卫式动作(synchronous guarded actions)[36]:$\langle\gamma\Rightarrow A\rangle$,作为其同步语言 QUARTZ 编译器的中间表达式,以能够支持多种纯同步模式的同步语言,其中,布尔条件 γ 称为卫式,A 称为动作.为了同时支持纯同步模式、多态同步模式以

及异步模式(例如 CAOS(concurrent action-oriented specification)及 SHIM(software/hardware integration medium)等语言),他们进一步提出多时钟卫式动作(clocked guarded action)[37],并基于模型检测工具 SMV 以及 SystemC 进行验证和仿真分析.在 SIGNAL 语言中,信号变量只能在某些逻辑时刻获得取值,即所谓的时钟.而且每个信号变量可以具有自己的时钟,即多时钟.CGA 具有多时钟特征,但是和 SIGNAL 语言不同的是,CGA 在时钟缺失的情况下仍然能够获得信号变量的取值.因此,我们提出 CGA 的一种变种,即 S-CGA.S-CGA 的语义更接近 SIGNAL 语言,使得更容易验证两者转换的语义保持.而 SIGNAL 到 S-CGA 转换的语义保持,可使确定性并发行为得到保持.

- 多线程代码

为了进一步保持确定性并发行为,我们使用 AADL 同步执行模型(synchronous execution model)[22,38](即 AADL 同步子集,周期性线程或线程组及数据端口通信)来表达多线程代码.

因此,确定性并发行为和实时规约贯穿于同步模型、S-CGA 以及 AADL 多线程代码这 3 个层次.同时,目标平台为时间可预测的多核体系结构模型,并且对时间可预测的软硬件映射进行了表达和分析.因此,时间可预测性在多个层次上得到保证,从而降低系统的最坏执行时间分析难度.

3 时间可预测代码生成器前端 SIGNAL2SCGA

本节将给出 SIGNAL 语言的抽象语法和形式语义、中间表达式 S-CGA 的语法和形式语义以及 SIGNAL 到 S-CGA 的转换规则,并基于定理证明器 Coq 对转换的语义保持进行证明.

3.1 SIGNAL抽象语法

如第 1.2 节所述,代码生成器会将 SIGNAL 语言的扩展结构转换为基本结构.因此在研究当中,我们仅考虑基本结构,并称之为 kSIGNAL(kernel SIGNAL).kSIGNAL 抽象语法如下所示:

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

SIGNAL 语言有多种形式语义,这里给出基于踪迹的指称语义(trace semantics),即,在全序的时间序列上分析信号之间的功能和时钟关系.我们先给出每个基本结构对应的踪迹集合定义.

踪迹语义规则 1. 瞬时函数 $x:=f(x_1,\ldots,x_n)$的踪迹语义定义如下:

$$
\forall \tau \in \mathbb{N}, x_\tau = \begin{cases} \bot, & \text{if } x_{1\tau} = \ldots = x_{n\tau} = \bot \\ f(x_{1\tau},\ldots,x_{n\tau}), & \text{if } x_{1\tau} \neq \bot \wedge \ldots \wedge x_{n\tau} \neq \bot \end{cases}.
$$

在每个逻辑时刻τ,所有信号要么同时存在,要么同时缺失,即,所有信号都是同步的,记为 $x\hat{}=x_1\hat{}=\ldots\hat{}=x_n$.当所有信号存在时,$x_\tau$获得$f(x_{1\tau},\ldots,x_{n\tau})$的取值.

踪迹语义规则 2. 延迟 $x:=x_1 \$ \mathit{init}\ c$ 的踪迹语义定义如下:

$$
\begin{aligned}
&(\forall \tau \in \mathbb{N}) x_{1\tau} = \bot \Leftrightarrow x_\tau = \bot,\\
&\{k \mid x_{1k} \neq \bot\} \neq \varnothing \Rightarrow x_{\min\{k \mid s_{1k} \neq \bot\}} = c,\\
&(\forall \tau \in \mathbb{N}) x_{1\tau} \neq \bot \wedge \{k > \tau \mid x_{1k} \neq \bot\} \neq \varnothing \Rightarrow x_{\min(k > \tau \mid x_{1k} \neq \bot\}} = x_{1\tau}.
\end{aligned}
$$

我们对原始语义定义[31]做了调整,使其更精确.min(X)代表非空自然数集合的最小值.类似的,信号 x 和 x_1 具有相同的时钟,即 $x\hat{}=x_1$.在逻辑时刻τ,x 获得 x_1 的前一个取值,而 x 的初始值为 c.

踪迹语义规则 3. 条件采样 $x:=x_1$ when x_2 的踪迹语义定义如下:

$$\forall \tau \in \mathbb{N}, x_\tau = \begin{cases} x_{1\tau}, & \text{if } x_{2\tau} = \text{true} \\ \perp, & \text{otherwise} \end{cases}.$$

这里,x 和 x_1 具有相同的数据类型,而 x_2 为 boolean 类型.x 的时钟为 x_1 的时钟和 x_2 的时钟并且 x_2 取值为真的交集,并记为 x:x_1^*[x_2].其中,[x_2]=$\hat{x}_2 \wedge x_2$ 表示 x_2 的存在并且取值为真.

踪迹语义规则 4. 确定性合并 x:=x_1 *default* x_2 的踪迹语义定义如下:

$$\forall \tau \in \mathbb{N}, x_\tau = \begin{cases} x_{1\tau}, & \text{if } x_{1\tau} \neq \perp \\ x_{2\tau}, & \text{otherwise} \end{cases}.$$

信号 x,x_1,x_2 具有相同的数据类型.x 的时钟为 x_1 的时钟和 x_2 的时钟的并集,并记为 x:=x_1^+x_2.在给定的逻辑时刻 τ,x_τ合并 $x_{1\tau}$和 $x_{2\tau}$的取值,而且 $x_{1\tau}$具有更高优先级.

定义 1(sprocess). 给定一个 SIGNAL 进程,应用踪迹语义规则 1~规则 4,其踪迹语义记为 Sprocess,包括进程的定义域即信号变量的集合以及踪迹的集合.

定义 2(踪迹等价). 设踪迹 tr_1 和 tr_2,两者之间存在踪迹等价关系(trace equivalnce),当且仅当它们具有相同的信号变量集合和相同的踪迹集合.

3.2 中间表达式S-CGA

我们先给出 S-CGA 的语法,并基于踪迹模型给出 S-CGA 的指称语义.

3.2.1 S-CGA 语法

S-CGA 重用了 CGA 的语法结构,但具有不同的语义.

定义 3(S-CGA). 一个 S-CGA 系统是定义在信号变量集合 X 上的卫式动作⟨$\gamma \Rightarrow A$⟩的集合.卫式动作可以是如下形式:

(1) $\gamma \Rightarrow x=\tau$ (immediate);

(2) $\gamma \Rightarrow next(x)=\tau$ (delayed);

(3) $\gamma \Rightarrow assume(\sigma)$ (assumption).

其中,

- 卫式γ是定义在信号变量 X、信号变量的逻辑时钟($x \in X$,其时钟记为 $\hat{x}$)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、延迟赋值和约束定义:
  1) 立即赋值:当γ中所有信号变量都存在,并且γ的取值为 true,x 和τ同时存在,则将τ的取值立即赋值给 x;
  2) 延迟赋值:当前时刻 t_1,γ中所有信号变量都存在,且γ的取值为 true,x 和τ在 t_1 时刻同时存在,在下一时刻 t_2 中,如果 x 存在,那么将在 t_1 时刻τ的取值赋值给 t_2 时刻的 x;
  3) 约束定义:当γ中所有信号变量都存在,并且γ的取值为 true,那么σ存在并且为 true.所有执行都需要满足约束定义.

  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

多个 S-CGA 组合采用同步组合操作||,因此和 SIGNAL 语言一样,能够表达确定性并发行为.

要注意的是:这里,我们使用符号⇒将卫式和动作分开,后面将采用→表示逻辑蕴含.

在 CGA 中,信号变量时钟缺失仍然可以赋值,这是为了同时支持异步并发.而在 S-CGA 语义中,我们做相应修改,与 SIGNAL 语义一致,即,每次都需要检查信号变量的时钟.在此基础上,将基于踪迹模型给出 S-CGA 的指称语义.

为精确定义 S-CGA 的形式语义,我们先给出卫式 Guards(γ)、约束 Assumptions(σ)以及表达式 Expressions(τ)的语法定义.

$$\gamma,\sigma ::= init(\hat{x}) \mid x \mid \hat{x} \mid f(\gamma,...,\gamma),$$
$$\tau ::= x \mid f(\tau,...,\tau).$$

这里考虑卫式和约束具有相同的语法结构.在此定义中,f能够表达常量(如 true,false 等)、算术操作、逻辑操作以及用户子定义函数.

3.2.2 S-CGA 形式语义

首先,分别给出卫式 Guards(γ)、约束 Assumptions(σ)以及表达式 Expressions(τ)的时钟定义以及取值定义.

定义 4. 给定踪迹 S 以及逻辑时刻 i,我们定义如下函数:

- $[\![\widehat{\gamma}]\!]_{S,i}$ 定义了$[\![\gamma]\!]_{S,i}$的域.$[\![\widehat{\gamma}]\!]_{S,i}$ 为真,当且仅当在S上的逻辑时刻i,γ中的所有信号变量都存在,即,它们的时钟都不缺失.注意,$[\![\widehat{\hat{x}}]\!]_{S,i}$ 被定义为真,因为一个时钟可以在任意逻辑时刻被读取:

$$[\![\widehat{\gamma}]\!]_{S,i}:\begin{cases}[\![\widehat{init(\hat{x})}]\!]_{S,i}=\text{true}\\ [\![\widehat{f(\gamma_1,....,\gamma_n)}]\!]_{S,i}=[\![\widehat{\gamma_1}]\!]_{S,i}\wedge...\wedge[\![\widehat{\gamma_n}]\!]_{S,i}\\ [\![\widehat{x}]\!]_{S,i}=(S(i,x)\neq\perp)\\ [\![\widehat{\hat{x}}]\!]_{S,i}=\text{true}\end{cases};$$

- $[\![\gamma]\!]_{S,i}$ 为一个偏函数,当 $[\![\widehat{\gamma}]\!]_{S,i}$ 为真,计算γ的取值(真或假):

$$[\![\gamma]\!]_{S,i}:\begin{cases}[\![init(\hat{x})]\!]_{S,i}=S(i,x)\neq\perp\wedge\forall j<i,S(j,x)=\perp\\ [\![f(\gamma_1,...,\gamma_n)]\!]_{S,i}=f([\![\gamma_1]\!]_{S,i},...,[\![\gamma_n]\!]_{S,i})\\ [\![x]\!]_{S,i}=S(i,x)\\ [\![\hat{x}]\!]_{S,i}=S(i,x)\neq\perp\end{cases};$$

- $[\![\widehat{\tau}]\!]_{S,i}$ 定义了$[\![\tau]\!]_{S,i}$ 的域.$[\![\widehat{\tau}]\!]_{S,i}$ 为真,当且仅当在 S 上的逻辑时刻 i,τ中的所有信号变量都存在,即,它们的时钟都不缺失:

$$[\![\widehat{\tau}]\!]_{S,i}:\begin{cases}[\![\widehat{f(\tau_1,....,\tau_n)}]\!]_{S,i}=[\![\widehat{\tau_1}]\!]_{S,i}\wedge...\wedge[\![\widehat{\tau_n}]\!]_{S,i}\\ [\![\widehat{x}]\!]_{S,i}=(S(i,x)\neq\perp)\end{cases};$$

- $[\![\tau]\!]_{S,i}$ 为一个偏函数,当 $[\![\widehat{\tau}]\!]_{S,i}$ 为真,计算τ的取值(真或假):

$$[\![\tau]\!]_{S,i}:\begin{cases}[\![f(\tau_1,...,\tau_n)]\!]_{S,i}=f([\![\tau_1]\!]_{S,i},...,[\![\tau_n]\!]_{S,i})\\ [\![x]\!]_{S,i}=S(i,x)\end{cases}.$$

由于卫式和约束具有相同的语法结构,$[\![\widehat{\sigma}]\!]_{S,i}$ 和$[\![\sigma]\!]_{S,i}$ 分别具有 $[\![\widehat{\gamma}]\!]_{S,i}$ 和$[\![\gamma]\!]_{S,i}$ 相同的定义.

其次,基于踪迹模型,我们定义 S-CGA 的踪迹语义.

定义 5(S-CGA 踪迹语义). 一个 S-CGA 系统的踪迹语义是踪迹的集合,即:

$$[\![SCGA]\!]=\{S\mid\forall scga\in SCGA,[\![scga]\!]_S=\text{true}\},$$

且有如下语义规则:

(1) $[\![\gamma\Rightarrow x=\tau]\!]_S=\forall i\in\mathbb{N},[\![\widehat{\gamma}]\!]_{S,i}\wedge[\![\gamma]\!]_{S,i}\rightarrow([\![\widehat{x}]\!]_{S,i}\wedge[\![\widehat{\tau}]\!]_{S,i}\wedge[\![x]\!]_{S,i}=[\![\tau]\!]_{S,i})$;

(2) $[\![\gamma\Rightarrow next(x)=\tau]\!]_S=\forall i_1<i_2\in\mathbb{N},((\forall i'\in\mathbb{N},i_1<i'<i_2\rightarrow\neg[\![\widehat{x}]\!]_{S,i'})\wedge[\![\widehat{\gamma}]\!]_{S,i_1}\wedge[\![\gamma]\!]_{S,i_1})\rightarrow$
$([\![\widehat{x}]\!]_{S,i_1}\wedge[\![\widehat{\tau}]\!]_{S,i_1}\wedge([\![\widehat{x}]\!]_{S,i_2}\rightarrow[\![x]\!]_{S,i_2}=[\![\tau]\!]_{S,i_1}))$;

(3) $[\![\gamma\Rightarrow assume(\sigma)]\!]_S=\forall i\in\mathbb{N},[\![\widehat{\gamma}]\!]_{S,i}\wedge[\![\gamma]\!]_{S,i}\rightarrow[\![\widehat{\sigma}]\!]_{S,i}\wedge[\![\sigma]\!]_{S,i}$.

规则 1. 当γ中所有信号变量都存在,且γ的取值为真,x 和τ中的所有变量都存在,那么 x 获得τ的取值.

规则 2. 在逻辑时刻 i_1,当γ中所有信号变量都存在,且γ的取值为真,x 和τ中的所有变量都存在,i_2 为 i_1 的下一个逻辑时刻,在 i_2 上,x 存在,则 x 获得τ在 i_1 时刻的取值.

规则 3. 当γ中所有信号变量都存在,且γ的取值为真,则,σ中所有信号变量都存在,且σ的取值为真.

S-CGA 的组合语义为

$$[\![scga_1||scga_2]\!]_S=[\![scga_1]\!]_S\wedge[\![scga_2]\!]_S.$$

3.3 SIGNAL2SCGA转换规则

SIGNAL 语言的基本结构,即 kSIGNAL,可以一一转换到 S-CGA 的表示,即,将 SIGNAL 程序所表达的功能关系和时钟关系都转换为 S-CGA 程序.这些转换规则类似于 SIGNAL 语言基本结构到 CGA 的转换,但是为了遵守 S-CGA 的语义,我们作了相应的修改.

kSIGNAL		S-CGA
(1) $x:=f(x_1,\ldots,x_n)$	$\Rightarrow$	$\hat{x}\Rightarrow x=f(x_1,\ldots,x_n)$ $\|\|\ \hat{x}_1\Rightarrow assume(\hat{x})$ $\|\|\ \ldots$ $\|\|\ \hat{x}_n\Rightarrow assume(\hat{x})$
(2) $x:=x_1$ \$ *init* c	$\Rightarrow$	$init(\hat{x})\Rightarrow x=c$ $\|\|\ \hat{x}\Rightarrow next(x)=x_1$ $\|\|\ \text{true}\Rightarrow assume(\hat{x}=\hat{x}_1)$
(3) $x:=x_1$ when x_2	$\Rightarrow$	$\hat{x}_1\wedge x_2\Rightarrow x=x_1$ $\|\|\ \hat{x}\Rightarrow assume(\hat{x}_1\wedge x_2)$
(4) $x:=x_1$ *default* x_2	$\Rightarrow$	$\hat{x}_1\Rightarrow x=x_1$ $\|\|\ \hat{x}_2\wedge\neg\hat{x}_1\Rightarrow x=x_2$ $\|\|\ \hat{x}\Rightarrow assume(\hat{x}_1\vee\hat{x}_2)$

转换规则 1. 瞬时函数应用于输入 $x_1,\ldots,x_n$,同时获得输出 x,并且所有信号变量是同步的.在 S-CGA 表示中,依据 S-CGA 的语义,立即赋值 $\hat{x}\Rightarrow x=f(x_1,\ldots,x_n)$ 蕴含了 $\hat{x}\Rightarrow\hat{x}_1,\ldots,\hat{x}\Rightarrow\hat{x}_n$,因此在约束定义中,我们仅给出另外一个方向,即, $x_1\Rightarrow assume(\hat{x}),\ldots,x_n\Rightarrow assume(\hat{x})$.

转换规则 2. 延迟结构分为两种情况:a) 当 x 存在的第 1 个逻辑时刻,这个结构所产生的第 1 个取值为初始值 c;b) 在所有其他逻辑时刻,x 获得在 $\hat{x}_1$ 的上一个非缺失逻辑时刻的取值.在 S-CGA 表示中,约束代表两个信号变量具有相同时钟.

转换规则 3. 条件采样结构,信号变量 x 存在,当且仅当输入 x_1 和 x_2 存在,并且 x_2 取值为真.在 S-CGA 表示中,基于 S-CGA 的语义规则 3,$assume(\hat{x}_1\wedge x_2)$蕴含 $\hat{x}_1\wedge\hat{x}_2\wedge x_2$.

转换规则 4. 确定性合并结构,将信号变量 x_1 和 x_2 进行合并,并且 x_1 具有更高优先级.

3.4 语义保持证明

我们使用定理证明器 Coq 对第 3 节所有定义进行了形式化.如图 3 所示,Coq 形式化表示包括 7 个模块(大约 1 300 行代码):kSIGNAL 抽象语法(10 行)、踪迹模型(250 行)、kSIGNAL 踪迹语义(100 行)、S-CGA 抽象语法(80 行)、S-CGA 踪迹语义(80 行)、转换规则(30 行)以及语义保持证明(750 行).我们用踪迹等价关系来表达两者语义的等价性.

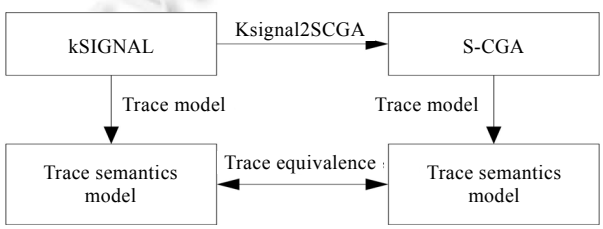

Fig.3 The proof of semantics preservation of SIGNAL2SCGA

图 3 SIGNAL2SCGA 语义保持证明

这里给出语义保持证明的主要思路:首先证明kSIGNAL的每条踪迹语义规则和转换得到的S-CGA表示的语义存在踪迹等价关系,每条语义规则的证明都包括两个方向,即两个引理;其次,基于这些引理,我们证明两个语义模型(Process2Sprocess P 和 scga2Sprocess(signal2scga P))为踪迹等价.踪迹等价性质是针对两个无穷对象,这也是采用定理证明器 Coq 来辅助证明(交互式定理证明)的原因.

4 时间可预测代码生成器后端 SCGA2TPCode

同步假设理论使得同步语言具有确定并发建模能力,但同时也使得同步语言的编译过程较为复杂.这是因为在代码生成之前,编译器需要确定信号何时计算以及如何进行计算,即,需要知道信号之间的时钟关系及数据依赖关系,分别对应时钟演算和数据依赖图构造两个阶段,其中,时钟演算又包括多个子阶段,如构造时钟关系等式系统、时钟关系等式的消解(resolution)、构造时钟树(clock hierarchy)等.我们将基于 S-CGA 来重写时钟演算和构造数据依赖图,并在两者的基础上给出任务划分方法,最终生成多线程代码.

要说明的是:一般情况下,SIGNAL 程序中的所有信号变量可以具有各自的时钟,但如果最终生成串行代码,即需要确定性的执行顺序,编译器会计算信号之间的时钟关系以及数据依赖关系,并给出一棵时钟树.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),因此,可以给出一个确定性的串行执行顺序.在同步语言中,称为 Endochrony 性质.但在具有全局时钟的情况下,我们仍然可能找到可并行执行的部分,对应生成多线程代码.为了放松对全局时钟存在的要求,研究者提出了 Weak Endochrony[39]性质的定义,但是目前还仍然处于理论定义和证明阶段.因此,本文依据 Endochrony 性质来生成多线程代码.

我们基于函数式程序设计语言 OCAML 实现了整个代码生成器原型,并采用了模块化的设计方法,即:某个模块的改动,并不影响其他模块的语义保持证明.主要包括:用户程序到 kSIGNAL 程序的转换、转换为 S-CGA 程序、基于 BDD(binary decision diagram)和 SMT(satisfiability modulo theories)的时钟演算、构造数据依赖图、任务划分、代码生成(包括串行代码和多线程代码)等模块.

图 4 给出了一个 SIGNAL 程序到 S-CGA 程序的转换示例,我们将以此 S-CGA 程序为例来介绍时间可预测多线程代码生成的过程.

```
1:  process Count=
2:  (?integer y1;
3:    boolean x1,x2;
4:    ! integer y;
5:  )
6:   (| x̂1 = x2
7:   | x̂1 = y2
8:  |y2:=(y1+1) $ init 2
9:  |s1:=y1 when x1
10: |s2:=y2 when x2
11: |x:=s1 default s2
12: |y:=x+1
13: |)
14: where integer x, y2, s1, s2;
15: end;
```

$$
\begin{aligned}
&\text{true} \Rightarrow assume(\hat{x}_1 = \hat{x}_2)\\
&\text{true} \Rightarrow assume(\hat{x}_1 = \hat{y}_2)\\
&\text{true} \Rightarrow assume(\hat{y}_1 = \hat{y}_2)\\
&init(\hat{y}_2) \Rightarrow y_2 = 2\\
&\hat{y}_2 \Rightarrow next(y_2) = y_1 + 1\\
&\hat{s}_1 \Rightarrow assume(\hat{y}_1 \wedge x_1)\\
&\hat{y}_1 \wedge x_1 \Rightarrow s_1 = y_1\\
&\hat{s}_2 \Rightarrow assume(\hat{y}_2 \wedge x_2)\\
&\hat{y}_2 \wedge x_2 \Rightarrow s_2 = y_2\\
&\hat{x} \Rightarrow assume(\hat{s}_1 \vee \hat{s}_2)\\
&\hat{s}_1 \Rightarrow x = s_1\\
&\hat{s}_2 \wedge (\neg\hat{s}_1) \Rightarrow x = s_2\\
&\hat{x} \Rightarrow assume(\hat{y})\\
&\hat{y} \Rightarrow y = x + 1
\end{aligned}
$$

Fig.4 An example of the translation from SIGNAL to S-CGA

图 4 SIGNAL 到 S-CGA 的转换示例

4.1 基于S-CGA的时钟演算

同步模型表达了功能关系和时钟关系.因此,同步语言编译器在代码生成之前,都需要做时钟演算(对应时钟关系)和数据依赖分析(对应功能关系),以确定该同步模型是可执行的,将来生成代码也更优化.

基于 S-CGA 的语义,针对每一个 S-CGA 表达式提取时钟关系等式,这样就能获得一个时钟关系等式集合.

基本产生规则为:

S-CGA	Clock equations
$\gamma \Rightarrow x=\tau$	$\hat{y} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow next(x)=\tau$	$\hat{y} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow assume(\sigma)$	$\hat{y} \wedge \gamma \rightarrow \hat{\sigma} \wedge \sigma$
	$init(\hat{x}) \rightarrow \hat{x}(\forall x \in X)$

每个时钟等式都被视为谓词.例如,在图 4 的 S-CGA 程序中,经过迭代计算,有时钟关系等式:

$$\hat{x} = (\hat{x}_1 \wedge x_1) \vee (\hat{x}_2 \wedge x_2).$$

左值 $\hat{x}$ 称为时钟变量,右边等式为该时钟变量的定义.在时钟关系等式集合中,如果多个时钟关系等式具有相同的时钟,那么生成的代码将会执行多次相同的控制条件,代码执行不够优化.因此,我们需要对整个时钟关系等式集合进行消解(resolution),即:将右边等式中的时钟变量用其定义进行替换,并不断地递归,最终通过 BDD 或 SMT 来检查两个时钟变量定义的等价性.如果相同,则将对应的时钟变量放入相同的时钟等价类中.相比之下,SIGNAL 已有编译器 Polychrony(http://www.irisa.fr/espresso/Polychrony)仅支持 BDD,我们将在代码生成器中同时支持 BDD 和 SMT.BDD 或 SMT 还进一步用于计算各时钟等价类之间的时钟蕴含关系,从而构建时钟树.

我们给出 S-CGA 示例程序的时钟等价类.例如,x_1,x_2,y_1 以及 y_2 是同步的,因此这些信号变量属于同一个时钟等价类(即 C_0).

$$C_0 : \{\hat{x}_1, \hat{x}_2, \hat{y}_1, \hat{y}_2\},$$
$$C_1 : \{\hat{y}_2 \wedge x_2, \hat{s}_2\},$$
$$C_2 : \{\hat{x}, \hat{y}\},$$
$$C_3 : \{\hat{y}_1 \wedge x_1, \hat{s}_1\},$$
$$C_4 : \{\hat{s}_2 \wedge \neg\hat{s}_1\}.$$

其时钟树如图 5 所示.在图中,*clk_x* 代表 $\hat{x}$.

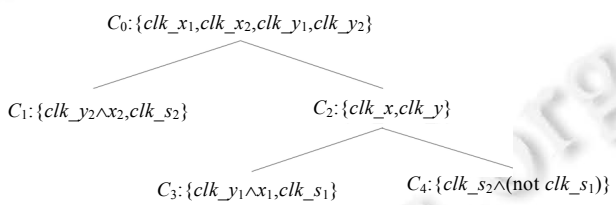

Fig.5 The clock hierarchy of the S-CGA example

图 5 S-CGA 示例程序的时钟树

时钟树的性质定义如下:

(1) 每个节点为一个时钟等价类;

(2) 有且仅有一个全局时钟等价类,即 C_0;

(3) 子节点和父节点之间存在时钟蕴含关系(基于 BDD 和 SMT 计算).例如,$\hat{y}_2 \wedge x_2 \rightarrow \hat{y}_2$ 意味着子节点 $\hat{y}_2 \wedge x_2$ 为真,父节点 $\hat{y}_2$ 永为真.因此,时钟树上的所有时钟等价类都可以用全局时钟等价类来定义.

4.2 基于S-CGA的数据依赖图构造

在串行代码生成中,我们将属于同一个时钟等价类的 S-CGA 等式放在同一个时钟节点上,并对这些等式按照读写数据依赖关系进行排序(如果为偏序,则采用拓扑排序).从而按照时钟树的顺序,生成串行代码的控制结构.而在控制结构内,按照经过排序的 S-CGA 等式生成对应的程序代码.

在多线程代码生成中,还需要显式构造数据依赖图,以找到更大并行度.

首先,为了使得生成的代码更加优化,我们在构造数据依赖图的过程中直接加入了时钟树信息.正如前面所述,所有时钟等价类都可以用全局时钟等价类来定义,因此,基于全局时钟等价类和非时钟变量给出各时钟等价类的定义.例如在生成的代码中,不需要每次都检查 C_0,从而代码控制结构会更优化.

$$C_0:\{\hat{x}_1,\hat{x}_2,\hat{y}_1,\hat{y}_2\},$$
$$C_1:\{\hat{y}_2\wedge x_2,\hat{s}_2\}:=C_0\wedge x_2,$$
$$C_2:\{\hat{x},\hat{y}\}:=C_0\wedge(x_1\vee x_2),$$
$$C_3:\{\hat{y}_1\wedge x_1,\hat{s}_1\}:=C_0\wedge x_1,$$
$$C_4:\{\hat{s}_2\wedge\neg\hat{s}_1\}:=C_0\wedge(x_2\wedge\neg x_1).$$

其次,基于变量的读/写依赖建立 S-CGA 表达式之间的关系,从而构建数据依赖图.对于读/写依赖,例如执行 $\gamma\Rightarrow x=\tau$,先要知道γ和τ中变量的取值,这称为读依赖变量,读变量的集合用 *RdVars* 表示.而此表达式对 x 进行赋值,即可以将 x 的值传给其他表达式,因此称为写依赖变量,写变量的集合用 *WrVars* 表示.对应的,可以定义读依赖动作集合和写依赖动作集合,分别用 *RdActs* 和 *WrActs* 表示.

定义 6(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:

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

而变量到动作的读写依赖定义如下:

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

要注意的是,$\gamma\Rightarrow assume(\sigma)$主要用于定义时钟约束关系,因此不用于构造数据依赖图.

在数据依赖图中,一个动作能够执行当且仅当所有读变量都获得取值.对应的,一个变量能够获得取值当且仅当所有写动作都已经完成计算.

基于以上定义,我们给出数据依赖图的定义及其构造规则.

定义 7(数据依赖图).数据依赖图为有向图$\langle V,\varepsilon\rangle$,$V$ 代表卫式动作,$\varepsilon\subseteq V^+\times V$ 代表卫式动作之间的依赖关系.并且在依赖关系中,如果有多条输入(multi-edges),那么只有其中一条会触发输出.其构造规则如下:

- Construct 规则.依据读写依赖(定义 6)构造卫式之间的关系:对任意的动作 $a\in A$,并对任意的变量 $x\in RdVars(a)$增加一条边:从 $WrActs(x)$到 a;
- Expand 规则.将卫式动作的卫式中的时钟等价类用对应的定义替代;
- Replace 规则.将全局时钟等价类用 true 替代;
- Simplify 规则.对卫式进行化简,对任意的边 $E=(V_1,V_2)$,设 V_1 为$\langle\gamma_1\Rightarrow A_1\rangle$,$V_2$ 为$\langle\gamma_2\Rightarrow A_2\rangle$:(1) 当$\gamma_1=\gamma_2$,则$\gamma_2$ 在 V_2 中删除;(2) 当$\gamma_2=\gamma_1\wedge\gamma_{1,2}$,则将$\gamma_2$ 替换为$\gamma_{1,2}$,以避免重复检测γ_1.

依据构造规则,我们给出 S-CGA 程序示例的数据依赖图,如图 6 所示.

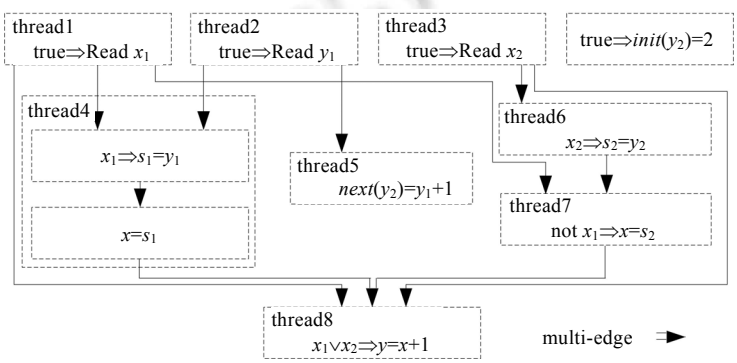

Fig.6 The data-dependency graph of the S-CGA example

图 6 S-CGA 程序实例的数据依赖图

对于延迟结构,true⇒$init(y_2)$=2 在初始化时只执行一次,而其他卫式动作可以无穷次执行.而且$next(y_2)=y_1+1$和 $x_2 \Rightarrow s_2=y_2$ 不存在直接的数据依赖关系,因为前者在当前步骤计算 y_2 取值,后者在下一步骤使用.另外,其中 $x=s_1$,not $x_1 \Rightarrow x=s_2$ 以及 $x_1 \vee x_2 \Rightarrow y=x+1$ 存在多条输入和输出的关系,因此只有一条输入能够触发输出.

4.3 任务划分方法

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).任务划分方法有很多种,例如垂直划分(vertical partition)[40]以找到并行执行,或水平划分(horizontal partition)[41]以支持流水线执行.我们的思路是仅定义什么是合法的任务划分,这样,不同的任务划分方法并不影响语义保持证明.

定义 8(合法的任务划分). 对于一个划分 P,A_1 和 A_2 是 P 中的两个 S-CGA 表达式,$\sqsubseteq$为关系 R 的自反和传递闭包,且 $R \subseteq A \times A$:$(A_1,A_2) \in R \Leftrightarrow WrVars(A_1) \cap RdVars(A_2) \neq \{\}$.我们称 P 为合法划分当且仅当$\sqsubseteq$为一个偏序.

要注意的是:当 A_1 是 A_2 中的读取变量的延迟赋值,那么 $WrVars(A_1)$和 $RdVars(A_2)$的交集为空.

图 6 已经给出 S-CGA 程序示例的划分,分为 8 个划分.这里,我们进一步给出划分方法:

1) 数据依赖中的每个节点被看做一个划分或一个线程;
2) 合并划分.例如:设两个划分 P_1 和 P_2,如果 P_2 为 P_1 的唯一子节点,而 P_1 为 P_2 的唯一父节点,那么将两者合并为一个划分;
3) 在每个划分当中,基于时钟等价类来对卫式进一步合并.例如,thread4 的两个卫式属于同一个时钟等价类,因此将两者的卫式进行合并.在生成的代码中,它们将处于相同的控制条件内.

4.4 多线程代码生成

我们将给出 AADL 多线程代码以及实时规约到 AADL 性质的转换.

为了保持同步模型的确定性并发,将使用 AADL 同步执行模型来实现多线程,即,周期性线程或线程组及数据端口通信.一般情况下,一个任务划分对应一个 AADL 线程组,线程组之间采用数据端口通信方式.每个线程组则包括 Guard 线程(用于检查 S-CGA 表达式的卫式)、Action 线程(用于表示 S-CGA 表达式的动作)以及 Sampling 线程(当卫式满足,则输出计算结果),3 个线程具有相同的周期.同时,将实时规约转换为 AADL 线程或线程组的实时性质定义,例如,实时响应可转换为对应线程组构件的流延迟定义(flow latency).

下面以 S-CGA 程序示例中的 thread8 为例.AADL 的输入数据端口(in data ports)只允许使用单一输入连接,当同一个输入数据端口需要有多个输入连接时,我们复制该输入数据端口以及对应的输入连接.例如,thread8 有两个输入数据端口用于分别从 thread4 和 thread7 获得 x 的取值.thread8 的 AADL 图形化表示如图 7 所示.

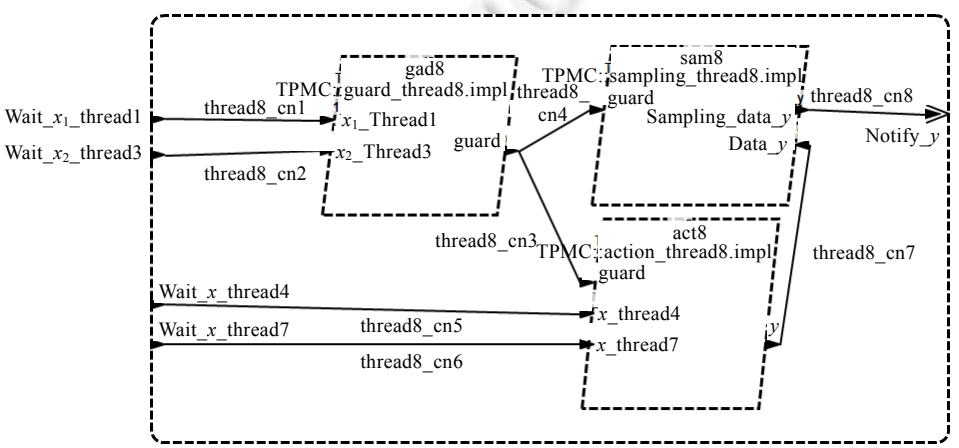

Fig.7 The AADL graphic model of thread8

图 7 thread8 的 AADL 图形化表示

线程 thread8 的 AADL 文本表示如下,包括 Guard 线程、Action 线程以及 Sampling 线程.Guard 线程用于检测 S-CGA 的卫式条件.当卫式条件满足,Action 线程执行计算.由于 x 变量来自 thread4 和 thread7,Action 线程会判断使用正确的 x 取值(见 Action 线程的 AADL 文本表示).最后,Sampling 线程输出需要的数据.这 3 个线程具有相同的周期,一般情况下可以定义为全局时钟.例如:实时规约给定 *Periodic*(x_1,20),在 AADL 线程定义为 *Period*⇒20ms.另外,实时规约中定义 *Response_time*(x_1,y,80),在 AADL 线程组中定义为,例如:

Flow1: flow path Wait_x_1_thread1→Notify_y\{*latency*⇒0ms,…,80ms;\}.

```
thread group thread8
features
  Wait_x1_thread1: in data port    Base_Types::Boolean;
  Wait_x2_thread3: in data port    Base_Types::Boolean;
  Wait_x_thread4: in data port    Base_Types::Integer;
  Wait_x_thread7: in data port    Base_Types::Integer;
  Notify_y: out event data port    Base_Types::Integer;
  flows
     Flow1: flow path Wait_x1_thread1→Notify_y{latency⇒0ms,…,80ms;};
     Flow2: flow path Wait_x2_thread3→Notify_y{latency⇒0ms,…,80ms;};
     Flow3: flow path Wait_x_thread4→Notify_y{latency⇒0ms,…,80ms;};
     Flow4: flow path Wait_x_thread7→Notify_y{latency⇒0ms,…,80ms;};
end thread8;

thread group implementation thread8.impl
 subcomponents
   gad8: thread guard_thread8.impl;
   act8: thread action_thread8.impl;
   sam8: thread sampling_thread8.impl;
 connections
   thread8_cn1: port Wait_x1_thread1→gad8.x1_Thread1;
   thread8_cn2: port Wait_x2_thread3→gad8.x2_Thread3;
   thread8_cn3: port gad8.guard→act8.guard;
   thread8_cn4: port gad8.guard→sam8.guard;
   thread8_cn5: port Wait_x_thread4→act8.x_thread4;
   thread8_cn6: port Wait_x_thread7→act8.x_thread7;
   thread8_cn7: port act8.y→sam8.Data_y;
   thread8_cn8: port sam8.Sampling_data_y→Notify_y;
 properties
   Period⇒40 ms;
   Deadline⇒80 ms;
end thread8.impl;
```

Guard 线程、Action 线程以及 Sampling 线程的 AADL 文本代码如下:

```
thread guard_thread8
features
   x1_Thread1: in data port Base_Types::Boolean;
   x2_Thread3: in data port Base_Types::Boolean;
   guard: out data port Base_Types::Boolean;
end guard_thread8;

thread implementation guard_thread8.impl
properties
   Period⇒20ms;
   Deadline⇒20ms;
annex behavior_specification {**
   states
   s0: initial complete state;
   transitions
   s0 -[on dispatch]→s0 {if (x1_Thread1) {guard:=x1_Thread1} else {guard:=x2_Thread3} end if};
**};
end guard_thread8.impl;
```

```
thread action_thread8
features
   guard: in data port Base_Types::Boolean;
   x_Thread4: in data port Base_Types::Integer;
   x_Thread7: in data port Base_Types::Integer;
   y: out data port Base_Types::Integer;
end action_thread8;

thread implementation action_thread8.impl
properties
   Period⇒20ms;
   Deadline⇒20ms;
annex behavior_specification {**
   states
   s0: initial complete state;
   transitions
   s0 -[on dispatch]→s0 {if (guard) if (x_Thread4'fresh) y:=x_Thread4+1 else y:=x_Thread7+1 end if end if};
**};
end action_thread8.impl;
```

```
thread sampling_thread8
features
   guard: in data port Base_Types::Boolean;
   Data_y: in data port Base_Types::Integer;
   Sampling_data_y: out data port   Base_Types::Integer;
end sampling_thread8;

thread implementation sampling_thread8.impl
properties
   Period ⇒20ms;
   Deadline ⇒20ms;
annex behavior_specification {**
   states
   s0: initial complete state;
   transitions
   s0 -[on dispatch]→s0 {if (guard) Sampling_data_y:=Data_y end if};
**};
end sampling_thread8.impl;
```

当卫式为全局时钟,则不生成线程组,而只生成 Action 线程.在 S-CGA 程序示例中,thread1,thread2,thread3 以及 thread5 都仅生成一个 Action 线程.

最终,4 个 Action 线程(thread1,thread2,thread3,thread5)、4 个线程组(thread4,thread6,thread7,thread8)以及线程之间的数据端口连接组成一个 AADL 进程构件 multi_threads.impl.其中,thread5 产生 y_2 的取值,并在下一个线程分发(dispatch)被 thread6 使用.因此,两个线程之间的数据端口连接为延迟连接(delayed connection),即:

{Timing ⇒Delayed}.

```
process multi_threads
features
   read_x1: in event data port Base_Types::Boolean;
   read_y1: in event data port Base_Types::Integer;
   read_x2: in event data port Base_Types::Boolean;
   write_y: out event data port Base_Types::Integer;
end multi_threads;

process implementation multi_threads.impl
   subcomponents
   T1: thread thread1.impl;
   T2: thread thread2.impl;
   T3: thread thread3.impl;
   T4: thread group thread4.impl;
   T5: thread thread5.impl;
   T6: thread group thread6.impl;
   T7: thread group thread7.impl;
   T8: thread group thread8.impl;
   connections
```

```
n_w_14: port T1.Notify_x1_thread4→T4.Wait_x1_thread1{Timing⇒Immediate;};
n_w_17: port T1.Notify_x1_thread7→T7.Wait_x1_thread1{Timing⇒Immediate;};
n_w_18: port T1.Notify_x1_thread8→T8.Wait_x1_thread1{Timing⇒Immediate;};
n_w_24: port T2.Notify_y1_thread4→T4.Wait_y1_thread2{Timing⇒Immediate;};
n_w_25: port T2.Notify_y1_thread5→T5.Wait_y1_thread2{Timing⇒Immediate;};
n_w_36: port T3.Notify_x2_thread6→T6.Wait_x2_thread3{Timing⇒Immediate;};
n_w_56: port T5.Notify_next_y2_thread6→T6.Wait_y2_thread5{Timing⇒Delayed;};
n_w_67: port T6.Notify_s2_thread7→T7.Wait_s2_thread6{Timing⇒Immediate;};
n_w_38: port T3.Notify_x2_thread8→T8.Wait_x2_thread3{Timing⇒Immediate;};
n_w_48: port T4.Notify_x_thread8→T8.Wait_x_thread4{Timing⇒Immediate;};
n_w_78: port T7.Notify_x_thread8→T8.Wait_x_thread7{Timing⇒Immediate;};
p_t_1: port read_x1→T1.Read_x1;
p_t_2: port read_y1→T2.Read_y1;
p_t_3: port read_x2→T3.Read_x2;
p_t_4: port T8.Notify_y→write_y;
end multi_threads.impl;
```

5 时间可预测多核体系结构模型及软硬件映射

通过扩展 AADL 属性集,给出时间可预测多核体系结构模型,并基于 AADL 给出简单的时间可预测软硬件映射和调度.

5.1 时间可预测多核体系结构模型

在引言中,我们简要综述了已有的时间可预测处理器研究,主要包括 Reinhard Wilhelm 提出的时间可预测处理器设计原则、时间可预测单核处理器(如 PRET 等)以及时间可预测多核处理器设计(PREDATOR,MERASA, T-CREST,parMERASA 等).基于这些已有研究,我们选取了一个功能子集作为目标平台,主要包括:(1) 支持方法 Cache(method cache)[42],以提高指令 Cache 的可预测性;(2) 将数据 Cache 分离为栈 Cache、常量数据 Cache 和静态数据 Cache(由于避免使用动态内存分配,我们不考虑堆 Cache),以提高数据 Cache 的可预测性[43];(3) 支持按序执行、静态分支预测的多流水线;(4) 支持 LRU Cache 替换策略;(5) 支持 TDMA(time division multiple access)的总线访问策略;(6) 支持 TDMA 的主存访问策略.正因为是模型,所以可以不断地改进和求精,最终构建统一的体系结构参考模型.

首先,通过扩展 AADL 属性集,如 TDMA_Window,TDMA_Schedule,Branch_Predication,Execution_Order, Cache_Replacement_Policy 等属性,以支持时间可预测体系结构的表达.

```
property set MC_Properties is
-       TDMA
TDMA_Window: type record (
   AccessPoint: list of reference (access connection);
   Slot: time;
   );
TDMA_Schedule: list of MC_Properties::TDMA_Window applies to (bus);
-- Branch predication in pipeline
Allowed_Branch_Predication_Type: type enumeration (Static, Dynamic);
Branch_Predication: MC_Properties::Allowed_Branch_Predication_Type applies to (processor, virtual processor);
-- Execution order in pipeline
Allowed_Execution_Order_Type: type enumeration (In_Order, Out_of_Order);
Execution_Order: MC_Properties::Allowed_Execution_Order_Type applies to (processor, virtual processor);
-- Cache replace policies
Allowed_Cache_Replacement_Policies_Type: type enumeration (LRU, FIFO, PLRU);
Cache_Replacement_Policy: MC_Properties::Allowed_Cache_Replacement_Policies_Type applies to (memory);
-- TDMA period
TDMA_Period: Timing_Properties::Time applies to (bus);
-- Burst length
Burst_Length: aadlinteger applies to (memory);
…
end MC_Properties;
```

其次,给出时间可预测多核体系结构的表达方法:每个计算核,我们采用 Processor 构件来表达;整个多核处理器用 System 构件来表达.这样,System 构件可包括例如多个计算核、Cache(每个计算核都有独立的方法

Cache、栈 Cache、常量数据 Cache、静态数据 Cache 等,都通过 Memory 构件来表示)、Scratchpad(用于线程之间的共享,通过 Memory 构件来表示)、共享总线(通过总线构件表示,并定义 TDMA 属性)等子构件.

图 8 给出了一个简化的时间可预测多核体系结构模型,在此模型中,我们考虑了经过空间分区的 L1 Cache.同理,可以对 L2,L3 进行空间分区.AADL 文本表示中,对 4 个计算核进行了表示.

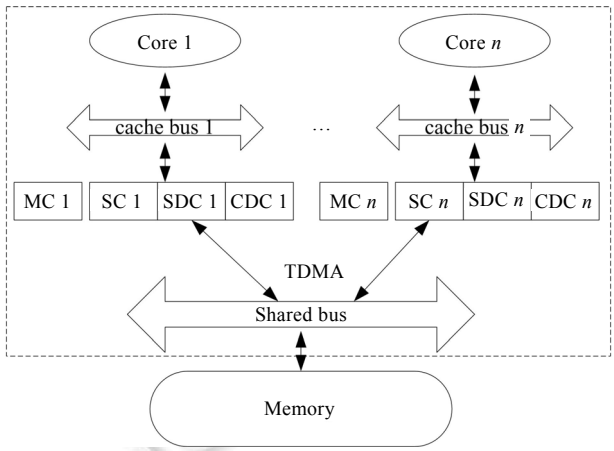

Fig.8 A simplified model of time-predictable multi-core architecture

图 8 简化的时间可预测多核体系结构模型

```
system multicore
features
ExtMem: provides bus access shared_bus.impl;
end multicore;

system implementation multicore.impl
subcomponents
Core1: processor normal_core.impl;
Core2: processor normal_core.impl;
Core3: processor normal_core.impl;
Core4: processor split_core.impl;

Cache_MC1: memory method_cache.impl;
Cache_MC2: memory method_cache.impl;
SBus: bus shared_bus.impl;

connections
Bacc1: bus access C2CBus1→Core1.MC;
Bacc2: bus access C2CBus2→Core2.MC;
Bacc3: bus access C2CBus3→Core3.MC;
Bacc4: bus access C2CBus4→Core4.MC1;
…
properties
MC_Properties::TDMA_Schedule⇒([AccessPoint⇒(reference (Bacc33), reference (Bacc34), reference (Bacc35),
reference (Bacc36), reference (Bacc37), reference (Bacc38), reference (Bacc39), reference (Bacc40),
reference (Bacc41), reference (Bacc42),reference (Bacc43), reference (Bacc44), reference (Bacc45),
reference (Bacc46), reference (Bacc47), reference (Bacc48),reference (Bacc57), reference (Bacc58),
reference (Bacc59),reference (Bacc60)); Slot⇒100μs;]) applies to SBus;
MC_Properties::TDMA_Period ⇒2ms applies to SBus; -- 20*100μs=2ms
end multicore.impl;
```

5.2 时间可预测软硬件映射方法

基于同步语言模型生成的AADL多线程代码和基于AADL表达的时间可预测多核结构模型构成一个完整的软硬件模型,并在 OSATE 工具中进行编译和分析.OSATE 是 AADL 语言的开源建模与分析工具,可以对 AADL 模型进行语法/语义检查、编译,并进行软硬件映射,从而进行性质分析.

一个线程组的多个线程之间存在确定性的执行顺序,一般会映射到同一个计算核.当多个线程组分配到同一个计算核上执行,多个线程组之间仍然可能存在资源访问干扰,如共享 Cache.

1) 当线程组执行可以访问 Cache 中的已有数据,为了避免资源访问干扰,我们将按照计算核上的线程组的数量,将计算核划分为多个虚拟处理器(virtual processor).例如在图 9 的 AADL 的文本表示中,计算核 Core1 被划分为 Core1.vcore1 和 Core1.vcore2,每个虚拟处理器具有各自的 Cache,并将线程组映射到虚拟处理器上;
2) 当线程组执行每次都是从空 Cache 开始,即,不访问 Cache 的已有数据,那么将不划分虚拟处理器.

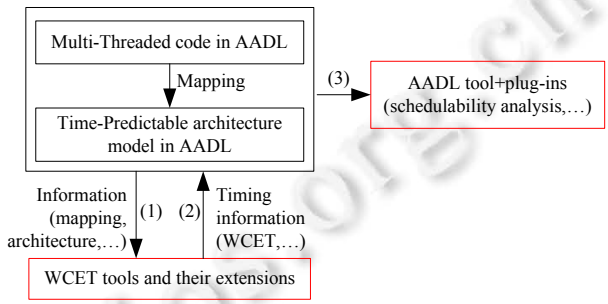

Fig.9 The framework of system-properties analysis

图 9 系统性质分析框架

为了保持可预测性,我们采用静态的软硬件映射方法.AADL 语言使用 Actual_Processor_Binding,Actual_Memory_Binding 等属性来表达静态映射,并对映射进行分析.AADL 语言也可以使用 Allow_Processor_Binding, Allow_Memory_Binding 等属性对其他可能的映射策略进行表达和分析.

```
system timepredictableProc
end timepredictableProc;

system implementation timepredictableProc.impl
subcomponents
  multiT: process multi_threads.impl;
  proc: system multicore.impl;
  mem: memory external_memory.impl;

connections
  Bacc: bus access mem.Memory_Bus↔proc.ExtMem;

properties
  Actual_Processor_Binding⇒(reference (proc.Core1)) applies to multiT.T1;
  Actual_Processor_Binding⇒(reference (proc.Core2)) applies to multiT.T2;
  Actual_Processor_Binding⇒(reference (proc.Core3)) applies to multiT.T3;
  Actual_Processor_Binding⇒(reference (proc.Core4.vcore1)) applies to multiT.T4;
  Actual_Processor_Binding⇒(reference (proc.Core2)) applies to multiT.T5;
  Actual_Processor_Binding⇒(reference (proc.Core3)) applies to multiT.T6;
  Actual_Processor_Binding⇒(reference (proc.Core4.vcore2)) applies to multiT.T7;
  Actual_Processor_Binding⇒(reference (proc.Core1)) applies to multiT.T8;
  Actual_Memory_Binding⇒(reference (mem)) applies to multiT;

end timepredictableProc.impl;
```

5.3 性质分析讨论

当确定软硬件映射之后,AADL 可以对系统的可调度性、端到端延迟等关键性质进行分析.我们给出时间可预测多核系统模型的性质分析框架(如图 9 所示):

- 首先,包含多线程及其实时性质、多核体系结构、软硬件映射等信息的时间可预测多核系统模型,能够

为最坏执行时间分析工具提供必要信息;

- 其次,最坏执行时间分析工具将 WCET 信息返回并补充给 AADL 模型;
- 最后,对具有完整信息的 AADL 时间可预测多核系统模型进行性质分析,尤其是与时间可预测机制相关的性质.例如,我们可以采用 AADL 可调度分析工具 Cheddar(http://beru.univ-brest.fr/~singhoff/cheddar/)对 TDMA 协议进行表达和仿真,最终进行任务的可调度分析.

6 相关工作

我们给出时间可预测编程规范、基于 SIGNAL 语言的多线程代码生成等方面的相关工作.

1) 时间可预测编程规范

为保证程序的时间可预测性,Wilhelm 等人提出了基本的编程规范[1],如不使用动态内存分配、不使用递归、不使用不可界定的循环(unbounded loops)等,这些规范已经体现在一些时间可预测编程语言当中.我们将时间可预测编程语言分为 3 类:增加实时约束、增加同步语言特征及单路径特征.

通常的编程语言仅仅表达功能行为,程序的时间行为是隐含的,即,到程序执行时才知道时间信息.然而,与 PRET 处理器的指令集对应,PRET 在 C 代码中增加了 Deadline 指令(*DEAD*(*t*)),即可以在编写程序时定义代码段的实时约束,这样能够同时表达程序的功能和非功能行为,我们称为 Berkeley-Columbia PRET 语言[11].基于 PRET 处理器,代码段的每条指令的执行时间可预测的,因此可以获得该代码段 Deadline 指令的取值 *t*.

新西兰奥克兰大学提出的时间可预测编程语言 PRET-C[44]是一种由同步语言和 C 语言结合的专用语言,即,在 C 语言中扩展了同步语言特征:反应式(即对外部环境的响应)和确定性并发.另外,提出了对 C 语言使用的约束:(1) 不使用指针和动态内存分配;(2) 每个循环要么包含 EOT 指令,要么给出最大的循环次数(例如,while (1) # *n* {…},*n* 即为最大循环次数),通过这种方式保证循环的时间可预测性;(3) 所有函数调用不使用递归方式;(4) 不使用 Goto 语句.

一般情况下,WCET 分析的复杂度与程序的输入数据有关.例如:针对条件分支语句,不同输入数据导致不同执行路径,每条执行路径可能具有不同执行时间.Puschner 等人提出了单路径(single-path)编程模型[45],以降低 WCET 分析的难度.该模型将依赖于输入数据的语句转换为常量时间条件表达式(constant-time conditional expressions),从而使得程序只有一条执行路径.

综上所述,3 类编程语言分别从不同角度来考虑时间可预测性,例如增加实时约束、增加确定性并发以及减少输入数据依赖带来的复杂度.首先,它们的语义规定最终多线程的执行仍然为串行执行,因此较难真正支持多核处理器结构;其次,缺乏一种自动生成方法,不方便最终用户使用.

2) 基于 SIGNAL 语言的多线程代码生成

基于同步语言的串行代码生成方法已经比较成熟,随着多核处理器的出现,基于同步语言的多线程代码生成方法成为研究的热点.

文献[27]介绍了 SIGNAL 语言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.

文献[28]提出了非侵入式的多线程代码生成方法,即,不改变已有的 SIGNAL 编译器(串行代码生成),通过增加程序 glue 文件来生成多线程代码.

文献[29]定义了一个完整的设计流程,即,从高层领域建模语言(例如 Simulink,SCADE,AADL,SysML,UMLMarte,SystemC 等)转换到 SIGNAL 模型,然后生成多线程代码.

这些研究还没有考虑时间可预测性质:多线程代码在多核处理器上的执行时间是可预测的,需要时间可预测多核处理器的支持;其次,多线程代码到多核处理器的映射和调度也需要时间可预测的.

在时间可预测多核系统设计以及同步语言等方面,目前国内研究还较少.清华大学在同步数据流语言LUSTRE 编译器验证方面开展了一些研究[46].

7 总结与展望

时间可预测多核系统设计是安全关键系统领域的重要挑战问题之一,也是目前国际学术界的研究热点.同步语言能够表达确定性并发行为,非常适合可预测系统设计.本文提出了一种基于同步语言(SIGNAL)模型驱动的时间可预测多核代码生成方法,并划分为时间可预测代码生成器前端 SIGNAL2SCGA、时间可预测代码生成器后端 SCGA2TPCode 以及时间可预测多核体系结构模型及其支持的软硬件映射这 3 个部分.

在软件层次,确定性并发行为和实时规约贯穿于同步模型、S-CGA 及 AADL 多线程代码这 3 个层次;硬件层次为时间可预测的多核体系结构模型.另外,对时间可预测的软硬件映射进行了表达和分析.因此,时间可预测性在多个层次得到保证.本文研究可为安全关键领域构造时间可预测嵌入式系统提供理论基础和方法支持.

在未来的研究当中,代码生成器后端的语义保持证明还需进一步完成.即:在定理证明器 Coq 中对时钟演算、数据依赖图构造、任务划分以及多线程代码生成进行形式化表示,并证明相关定理,最终形成完整的 SIGNAL 语言验证编译器原型.其中,目标语言,即 AADL 的语义形式化已经完成,见文献[38,47].目前,本文主要侧重方法性研究,我们将基于航天领域的飞行控制子系统进行实例性验证.另外,本文的研究都在模型层次,今后我们将时间可预测多核体系结构模型在 FPGA 上实现.

References:

[1] Axer P, Ernst R, Falk H, Girault A, Grund D, Guan N, Jonsson B, Marwedel P, Reineke J, Rochange C, Sebastian M, Von hanxleden R, Wilhelm R, Wang Y. Building timing predictable embedded systems. ACM Trans. on Embeded Computer Systems, 2014,13(4):82:1−82:37. [doi: 10.1145/2560033]

[2] Thiele L, Wilhelm R. Design for timing predictability. Real-Time Systems, 2004,28(2-3):157−177. [doi: 10.1023/B:TIME.0000045316.66276.6e]

[3] Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P. The worst-case execution-time problem: Overview of methods and survey of tools. ACM Trans. on Embeded Computer Systems, 2008,7(3):53. [doi: 10.1145/1347375.1347389]

[4] Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2009,28(7):966−978. [doi: 10.1109/TCAD.2009.2013287]

[5] aiT WCET analyzer. http://www.absint.com/aiT

[6] Rapitime. http://www.rapitasystems.com/products/rapitime

[7] OTAWA. http://www.otawa.fr/

[8] Chronos. http://www.comp.nus.edu.sg/~rpembed/chronos/

[9] Chattopadhyay S, Roychoudhury A. Scalable and precise refinement of cache timing analysis via path-sensitive verification. Real-Time Systems, 2013,49(4):517−562. [doi: 10.1007/s11241-013-9178-0]

[10] Chattopadhyay S, Chong LK, Roychoudhury A, Kelter T, Marwedel P, Falk H. A unified WCET analysis framework for multi-core platforms. In: Proc. of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2012. 99−108. http://doi.ieeecomputersociety.org/10.1109/RTAS.2012.26 [doi: 10.1109/RTAS.2012.26]

[11] Edwards S, Lee E A. The case for the precision timed (PRET) machine. In: Proc. of the Design Automation Convention. 2007. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.3179 [doi: 10.1145/1278480.1278545]

[12] Predator: Design for predictability and efficiency. http://www.predator-project.eu/

[13] MERASA. https://www.rapitasystems.com/about/research-projects/merasa

[14] T-CREST. http://www.t-crest.org/

[15] Ungerer T, Bradatsch C, Gerdes M, Kluge F, Jahr R, Mische J, Fernandes J, Zaykov PG, Petrov Z, Böddeker B, Kehr S, Regler H, Hugl A, Rochange C, Ozaktas H, Cassé H, Bonenfant A, Sainrat P, Broster I, Lay N, George D, Quiñones E, Panic M, Abella J, Cazorla F, Uhrig S, Rohde M, Pyka A. parMERASA: Multi-Core execution of parallelised hard real-time applications supporting analysability. In: Proc. of the 16th Euromicro Conf. on Digital System Design. 2013. 363−370. [doi: 10.1109/DSD.2013.46]

[16] Puschner P, Kirner R, Huber B, Prokesch D. Compiling for time predictability. In: Proc. of the Computer Safety, Reliability, and Security. LNCS 7613, 2012. 382−391. [doi: 10.1007/978-3-642-33675-1_35]

[17] Baldovin A, Mezzetti E, Vardanega T. A time-composable operating system. In: Vardanega T, ed. Proc. of the 12th Int'l Workshop on Worst-Case Execution Time Analysis (WCET 2012). 2012. 69−80. [doi: 10.4230/OASIcs.WCET.2012.69]

[18] Wolf J, Gerdes M, Kluge F, Uhrig S, Mische J, Metzlaff S, Rochange C, Cass H, Sainrat P, Ungerer T. RTOS support for execution of parallelized hard real-time tasks on the MERASA multi-core processor. Int'l Journal of Computer Systems Science & Engineering (CSSE), Special Issue on Real-Time Systems, 2011,26(6).

[19] Benveniste A, Caspi P, Edwards S, 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]

[20] SAE. Architecture analysis & design language (standard SAE AS5506). 2004. http://www.sae.org

[21] SAE. Architecture analysis & design language (standard SAE AS5506A). 2009. http://www.sae.org

[22] 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]

[23] 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]

[24] Boussinot F, de Simone R. The Esterel language. Proc. of the IEEE, 1991,79(9):1293−1304. [doi: 10.1109/5.97299]

[25] 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. [doi: 10.1016/0167-6423(91)90001-E]

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

[27] Besnard L, Gautier T, Talpin JP. Code generation strategies in the Polychrony environment. http://hal.inria.fr/docs/00/37/24/12/PDF/RR-6894.pdf

[28] 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. [doi: 10.1016/j.entcs.2008.01.006]

[29] 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. [doi: 10.1109/ESLsyn.2011.5952287]

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

[31] Gamatié A. Designing Embedded Systems with the SIGNAL Programming Language. Springer-Verlag, 2010. [doi: 10.1007/978-1-4419-0941-1]

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

[33] 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]

[34] Bertot Y, Casteran P. Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. 2004. [doi: 10.1007/978-3-662-07964-5]

[35] Brandt J, Gemunde M, Schneider K, Shukla SK, Talpin JP. Embedding polychrony into synchrony. IEEE Trans. on Software Engineering, 2013,39(7):917−929. [doi: 10.1109/TSE.2012.85]

[36] Brandt J, Schneider K. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, 2011.

[37] Brandt J, Gemunde M, Schneider K, Shukla SK, Talpin JP. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. In: Proc. of the Design Automation for Embedded Systems. 2012. 1−35. [doi: 10.1007/s10617-012-9087-9]

[38] Yang Z, Hu K, Ma D, Bodeveix JP, Pi L, Talpin JP. From AADL to timed abstract state machines: A verified model transformation. Journal of Systems and Software, 2014,93:42−68. [doi: 10.1016/j.jss.2014.02.058]

[39] 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]

[40] Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In: Proc. of the Design, Automation and Test in Europe (DATE 2010). 2010. 949−952. [doi: 10.1109/DATE.2010.5456915]

[41] Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In: Dietrich M, ed. Proc. of the MBMV. Fraunhofer Verlag, 2010. 11−20.

[42] Schoeberl M. A time predictable instruction cache for a Java processor. In: Meersman R, Tari Z, Corsaro A, eds. Proc. of the Workshops on Move to Meaningful Internet Systems (OTM 2004). LNCS 3292, Springer-Verlag, 2004. 371−382. [doi: 10.1007/978-3-540-30470-8_52]

[43] Schoeberl M, Huber B, Puffitsch W. Data cache organization for accurate timing analysis. Real-Time Systems, 2013,49(1):1−28. [doi: 10.1007/s11241-012-9159-8]

[44] Andalam S, Roop PS, Girault A, Traulsen C. PRET-C: A new language for programming precision timed architectures. Rapport Derecherchen 6922, INRIA, 2009. 38. http://citeseerx.ist.psu.edu/viewdoc/summary [doi: 10.1.1.159.9265]

[45] Puschner P, Burns A. Writing temporally predictable code. In: Proc. of the 7th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002). Washington: IEEE Computer Society, 2002. 85. [doi: 10.1109/WORDS.2002.1000040]

[46] 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]

[47] Bodeveix JP, Filali M, Garnacho M, Spadotti R, Yang Z: Towards a verified transformation from AADL to the formal component-based language FIACRE. Science of Computer Programming, 2015,106:30−53. [doi: 10.1016/j.scico.2015.03.003]

附中文参考文献:

[22] 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]

[46] 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]

杨志斌(1982－),男,江西吉安人,博士,副研究员,CCF 会员,主要研究领域为实时系统,形式验证.

马殿富(1960－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为服务计算,实时系统.

赵永望(1979－),男,博士,讲师,CCF 高级会员,主要研究领域为服务计算,实时系统.

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

黄志球(1965－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为软件工程.

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

胡凯(1963－),男,博士,副教授,主要研究领域为分布式计算,实时系统.