计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220730
61(1): 98−119, 2024

基于 RPU 的 TTI 程序设计和分析方法

高银康　陈香兰　龚小航　蒋滨泽　李　曦　周学海
（中国科学技术大学计算机科学与技术学院　合肥　230027）
（中国科学技术大学苏州高等研究院　江苏苏州　215004）
（gaoyinkang@mail.ustc.edu.cn）

Design and Analysis Method of TTI Program Based on RPU

Gao Yinkang, Chen Xianglan, Gong Xiaohang, Jiang Binze, Li Xi, and Zhou Xuehai
（*School of Computer Science and Technology*, *University of Science and Technology of China*, *Hefei* 230027）
（*Suzhou Institute for Advanced Research*, *University of Science and Technology of China*, *Suzhou*, *Jiangsu* 215004）

Abstract　Real-time embedded system (RTES) needs to guarantee not only logical correctness of the calculation results, but also temporal correctness of the interaction with outside world, so low-level programs must be able to accurately express time behavior in upper-level models. TTI instruction set (time-triggered instruction set) is proposed to try to solve the problem of the lack of timing semantics at the computer instruction set architecture level, and the realization of real-time processing unit (RPU) based on TTI instruction set proves the feasibility and effectiveness of TTI instruction set. However, the current work lacks research on design and analysis method of TTI program. Therefore, based on TTI instruction set and RPU, we propose four types of timing semantics that TTI instruction set can express, give the design paradigm of TTI program. Then, we define the representation of TTI program time behavior — TFG+, which is an extension of TFG. TFG+ distinguishes timing semantic instructions and the common code segments in the TTI program, and TFG+ can represent the control flow information of TTI program, the time behavior specified by the user, and the time attributes related to hardware platform. Finally, we propose time analysis method of TTI program and time safety checking method, which provide the basis for the design and deployment of TTI program.

Key words　real-time embedded system (RTES); timing predictability; timed instruction set; real-time processor; WCET analysis

摘　要　实时嵌入式系统不仅要保证计算结果的逻辑正确性，还要确保与外界交互的时序正确性，所以底层程序要能精确表达上层模型中的时间行为. TTI 指令集（time-triggered instruction set）的提出尝试解决计算机指令集体系结构层次缺少时间语义的问题，并且基于 TTI 指令集实现的实时处理单元（real-time processing unit, RPU）证明了 TTI 指令集的可行性和有效性. 但是目前的工作缺少对于 TTI 程序设计和分析方法的研究. 所以，基于 TTI 指令集和 RPU，提出了 TTI 指令集可以表达的 4 种时间语义，给出了 TTI 程序的设计范式. 并且构建了 TTI 程序时间行为的表示方法——TFG+，TFG+是对 TFG 的扩展，TFG+区分了 TTI 程序中时间语义指令和普通代码段，可以表示 TTI 程序的控制流信息、用户规定的时间行为和 TTI 程序平台相关的时间属性. 最后，提出了 TTI 程序的时间分析方法以及时间安全性检查方法，为 TTI 程序的

收稿日期：2022-08-22；**修回日期**：2023-04-19
基金项目：国家重点研发计划项目 (2017YFA0700900)；国家自然科学基金项目 (62102383)
This work was supported by the National Key Research and Development Program of China (2017YFA0700900) and the National Natural Science Foundation of China (62102383)
通信作者：李曦（llxx@ustc.edu.cn）

设计和部署提供了依据.

关键词 实时嵌入式系统；时间可预测性；时间语义指令集；实时处理器；WCET 分析

中图法分类号 TP332

实时嵌入式系统（real-time embedded system, RTES）通常应用于安全关键领域，此类系统需要实时和外界环境交互，因此不仅要保证计算结果的逻辑正确性，还要确保与外界交互的时序正确性.

实时领域的程序设计中，高层的编程模型含有时间语义，如逻辑执行时间（logical execution time, LET）编程模型[1]，而在底层的指令集架构（instruction set architecture, ISA）层面缺乏时间语义，导致编程人员无法直接控制底层程序的时序行为. 现阶段实时系统设计和分析的主要方法是，计算底层程序的执行时间，并以此作为程序时序控制的依据，但是由于程序中不确定的分支跳转指令，以及硬件体系结构（流水线、Cache、分支预测等）影响，传统程序的执行时间分析，总是悲观地分析程序在最坏情况下的执行时间（worse case execution time, WCET），而为了保证程序时序行为的正确，就必须依据 WCET 进行资源的过度分配，使高层的编程模型不能精确且自然地映射至底层程序.

造成这一问题的原因是通用体系结构的 ISA 没有可以表达时间语义的指令，所以和 LET 等实时编程模型的表达能力存在差距. 针对该问题，最有效的方法是规定程序中每一条指令的执行时间，但是这样会限制底层硬件体系结构的发展[2]. 为平衡时间语义表达能力和执行性能的问题，Lee 等人[3] 提出 PRET（precision timed）概念，并提出了限定代码段执行时间的指令，一定程度上提升了底层指令集的表达能力. 文献 [4] 提出了 TTI 指令集（time-triggered instruction set），与 Lee 小组工作相比，增加了时间和操作相结合的定时指令，使指令集的表达能力进一步提升，本文的分析和验证都依据 TTI 指令集进行，并且将使用了 TTI 指令集的程序称为 TTI 程序.

Lee 等人[3] 和文献 [4] 中对于指令集扩展的工作，仅给出了时间语义指令集的定义和实现方案，并没有说明指令集可以表达的时间语义和相应的时间语义程序设计范式，导致实时系统设计者在进行上层模型至底层程序的映射时缺乏理论依据. TTI 指令集通过不同指令组合可以表达丰富的时间语义，为了映射上层设计，需要给出 TTI 程序的设计范式，明确 TTI 程序可以表达的语义类型.

TTI 程序运行时，时间语义指令的执行会受到前驱代码段执行时间的影响，为了保证指令的时间语义在执行时能够得到正确表达，需要对 TTI 程序进行时间行为分析. 例如，对于 TTI 指令集中定时输出指令 ttoat，指令语义要求在预设时间点完成输出操作，而 ttoat 指令会在前驱代码段执行完成后开始执行，所以需要计算 ttoat 指令前驱代码段的 WCET，保证该 WCET 小于 ttoat 指令预设的时间点，以此保证定时语义的正确表达. 并且传统程序的时间行为分析，会以整个程序（任务）为单位，给出程序（任务）的 WCET，而 TTI 程序需要计算时间语义指令间程序段的 WCET，并结合时间语义指令的语义要求进行时间安全性检查，验证指令运行时的正确性，所以需要改进传统程序的 WCET 分析方法，提出针对时间语义程序的时间行为分析方法.

本文依据文献 [4] 所述工作，首先提出了 TTI 程序的设计范式，说明了 TTI 指令集的表达能力，明确 TTI 程序可以表达的 4 种语义. 同时，对 Timed C[5] 系列工作中提出的程序时间行为的表示 TFG[6] 进行扩展，提出了 TTI 程序的表示 TFG+，描述了 TTI 程序的控制流信息、用户规定的时间行为和平台相关的时间属性. 最后，根据 TFG+，基于实时处理单元（real-time processing unit, RPU），采用基于执行图[7] 的低级分析方法，分析了 TTI 程序的时间行为，并且对 TTI 程序的时间语义指令进行时间安全性检查，验证其语义在执行时能够得到正确表达. 本文提出的 TFG+ 构建方法、时间行为分析方法以及时间安全性检查方法适用于 RPU 和其他可以运行 TTI 指令集的处理器架构，若采用除 RPU 外的其他架构，只需要修改执行图的构建方法和执行图节点属性的计算方法即可.

本文的主要贡献有 3 个方面：

1）提出 TTI 程序的设计范式，明确 TTI 程序可以表达的 4 种时间语义，使编程人员能够精确自然地表达上层的设计.

2）提出 TFG+，以表示 TTI 程序，TFG+可以表达 TTI 程序的控制流信息、用户规定的时间行为以及平台相关的时间属性，为 TTI 程序的分析和时间安全性检查提供了依据.

3）提出了 TTI 程序的时间行为分析方法和时间

安全性检查方法,保证 TTI 程序运行时指令的语义能够正确表达.

1 相关工作

相关工作分为时间语义指令集、程序抽象表示和程序时间行为分析 3 个方面进行介绍.

1.1 时间语义指令集

Lee 在文献 [8] 中从“实时”概念出发,总结了多种对于实时性的定义. Lee[8] 认为根据不同的需求和角度,“实时”概念有 6 种定义:快速计算、优先调度、对数据流的计算、有界执行、程序的时间语义和网络的时间语义. 其中,在对程序的时间语义的阐述中,Lee[8] 认为实时系统设计和部署时,由于现在的通用计算 ISA 中不存在时间语义,需要在更高层次重新引入时间语义,但是通用计算 ISA 和高层次模型关于时间语义表达能力上存在的差距,会导致实际系统设计时的诸多问题,所以需要在实时系统设计的各个抽象层次引入时间语义.

根据上述观点,Lee 等人[2] 认为通用计算机指令集隐藏了底层体系结构中精确的数字电路时钟,为了控制程序的时间行为,计算机要向高层次模型提供这一精确时间. 对于时序的控制,Lee 等人[2] 定义了 ISA 层需要具备的 4 种能力:1)限定程序块执行时间下界;2)程序块执行超过限定时间,程序块完成后分支跳转;3)程序块执行超过限定时间,立即分支跳转;4)限定程序块执行时间的上界. 文献 [2] 较早地提出了 ISA 层在时序控制方法需要具有的时间语义,但是没有说明上层模型到底层指令的映射关系,没有给出相应的设计范式,同时,Lee 等人[2] 认为时间语义指令集只需要“限定代码段执行时间”,没有关于点时间语义的论述.

Lee 等人[2] 对通用计算指令集进行扩展,增加指令 DU(delay until)保证代码执行时间下界,指令 MTFD(meet the final deadline)保证代码执行时间上界. 并且依据此设计了处理器 PTARM[9] 和 FlexPRET[10]. 此方法可以限定代码的执行时间,但是无法直接表达操作发生在某个时刻的语义,所以该指令集的表达能力依旧有限.

文献 [4] 从实时理论出发,类比通用计算中图灵机为通用计算系统的理论基础[11],冯•诺依曼机是通用计算机系统统一的实现模型[12],提出了具有时间语义的实时计算机体系结构模型实时机(real-time machine, RTM),RTM 在冯•诺依曼机 5 大部件的基础上扩展了标准时钟(stdClk)部件,至此处理器时钟(cpuClk)负责实时处理器的值域控制,标准时钟负责实时处理器的时域控制. 基于 RTM 的理论指导,文献 [4] 结合时间触发自动机(time-triggered automata, TTA)理论[13],定义了时间语义指令集 TTI. 相比于 Lee 等人[2] 提出的限定代码执行时间的时间语义指令集,TTI 指令集不仅具有限定代码执行时间上下界的 delay, mtfd 指令,还增加了时间和操作相结合的定时指令 ttiat, ttoat, 以及任务并发管理指令,进一步扩展了指令集的表达能力. 下面详细介绍 TTI 指令集的定义和实现.

TTI 指令集包含时间语义寄存器和时间语义指令 2 部分. 表 1 给出了时间语义寄存器. 时间粒度寄存器(time granularity register),用来设置 TTI 程序的时间粒度,以满足不同系统或不同工作模式的时间粒度要求;时间寄存器(current time register),用来表示系统时间,根据时间粒度寄存器生成,即标准时钟的上升沿累计值等于时间粒度寄存器时系统时间寄存器的值加 1;时间戳寄存器(time stamp register),用来保存上一次时间触发操作的系统时间. 如表 2 所示,TTI 指令集中共有 10 条时间语义指令,按照功能分为 4 部分. 文献 [4] 中对于指令的语义进行了详细说明.

Table 1 Time Semantic Registers

表 1 时间语义寄存器

寄存器名称	助记符	描述
时间粒度寄存器	tgr	系统时间粒度
时间寄存器	ctr	系统当前时间
时间戳寄存器	tsr	保存上一个时间触发操作发生的时间

Table 2 Time Semantic Instructions

表 2 时间语义指令

类型	汇编指令
系统时间管理指令	settg *rs*1 getti *rd* setti *rs*1 getts *rd*
时间触发操作指令	ttiat *rd*, (*rs*2), *rs*1 ttoat *rs*3, (*rs*2), *rs*1
执行时间约束指令	delay *rs*1 mtfd *r*1
任务并发管理指令	tkend addtk *rs*1, *rs*2

目前时间语义指令集的相关工作仅说明了每条指令的语义和对应的硬件实现方案,缺乏时间语义指令集表达能力的说明,同时,没有给出时间语义程序的设计范式,导致无法直接自然地定义上层模型中时间语义与底层时间语义程序的映射关系.

文献 [4] 基于 TTI 指令集的定义，扩展 RISC-V 指令集[14]，实现了可以执行 TTI 指令集的 RPU. RPU 基于经典 5 段流水[15] 实现，即流水线分为取指段、译码段、执行段、访存段和写回段 5 个阶段. 下面给出 TTI 指令集的实现.

1）系统时间管理指令. 此类指令的功能是对标准时钟进行管理，生成系统时间. 执行段，settg 指令设置 tgr，setti 指令设置 ctr. 写回段，getti 指令读取 ctr 中保存的当前系统时间到目的寄存器，getts 指令读取 tsr 保存的上一时间触发操作发生时间到目的寄存器.

2）时间触发操作指令. 此类指令的特点是操作和时间绑定，规定操作在指定系统时间触发执行. 执行段，比较 ctr 的值和指令预设时间，若 ctr 小于指令预设时间，产生 stall 信号停顿流水线，直到 ctr 等于指令预设时间，执行时间触发操作指令. 后续指令 ttiat/ttoat 在访存段和写回段的操作和普通 load/store 指令相同.

3）执行时间约束指令. 此类指令的功能是保证代码段执行时间的上下界. delay 指令用于约束代码执行时间的下界；mtfd 指令用于约束代码执行时间的上界. 执行段，delay 指令会比较 ctr 的值和指令预设时间，若 ctr 小于指令预设时间，表示未到延时时间，产生 stall 信号停顿流水线，直到 ctr 等于指令预设时间，才继续指令的执行. 对于 mtfd 指令，RPU 没有相关实现，mtfd 指令只作为代码中的检查点，用于时间安全性检查.

4）任务并发管理指令. 此类指令的功能是摒弃传统的使用中断的软件任务并发管理，提供确定的硬件线程管理任务并发的能力. 本文不考虑多线程 TTI 程序，所以省略任务并发管理指令的实现.

1.2 程序的表示

传统程序进行 WCET 分析时，可以将程序表示为控制流图（control flow graph, CFG），在 CFG 中程序以基本块（basic block）为单位进行划分，基本块间的依赖关系通过 CFG 中基本块间的边进行表达. 后续许多程序表示都是对 CFG 的扩充.

文献 [16] 中提出了 CKAG（concurrent kep assembler graph），用来表示使用了处理器 KEP（kiel Esterel processor）指令集的程序，KEP 是可以运行同步语言 Esterel[17] 的专用处理器，而 CKAG 的提出是为了计算同步语言程序最坏情况反应时间（worst case reaction time, WCRT）. 同步语言 PRET-C[18] 为了计算 WCRT 也提出了 PRET-C 程序的表示 TCCFG（timed concurrent control flow graph）[19].

文献 [6] 中提出 TFG 的概念，用来表示 Timed C[5] 程序，进行 Timed C 编译器正确性检查，Timed C 程序的 TFG 在 CFG 的基础上，添加了表达段时间语义的时间点（timing point, TP）. 另外，FG 定义了图中每个节点的时间属性，至此 TFG 不仅可以表达程序的控制流信息，还可以表达用户规定的时间行为，以及平台相关的时间属性，但是 Timed C 程序的 TFG 受限于 Timed C 语言，只是表达了段时间语义. 本文结合 TTI 程序提出了 TFG+，是对 TFG 的扩充，表达了更为丰富的时间语义，例如点时间语义.

1.3 程序时间行为分析

现有的程序 WCET 分析方法分为基于测量的 WCET 分析和静态 WCET 分析. 静态 WCET 分析，由流分析（flow analysis）、低级分析（low-level analysis）和 WCET 计算（WCET computation）组成.

1）流分析. 首先通过分析程序结构，构建程序的表示，例如 CFG[20]；然后由程序员手动[21] 或通过自动分析[22] 给出程序循环边界（loop bound）等流事实（flow fact）.

2）低级分析. 目的是结合硬件体系结构分析 CFG 中每个基本块的执行时间，常用的方法有抽象解释（abstract interpretation）[23]、基于执行图（execution graph）[5, 24-25] 的分析方法. 抽象解释方法的 WCET 分析工具有 aiT[26]，而基于执行图的 WCET 分析工具有 Chronos[27]，OTAWA[28].

3）WCET 计算. 目的是计算整个程序最坏情况下的执行时间，现有基于语法树的 WCET 计算方法[29-30] 和基于 CFG 的 WCET 计算方法[31]. 基于 CFG，整个程序的执行时间可以表示为每个基本块执行时间的加权和，权值为每个基本块的执行次数. 通过分析 CFG 节点依赖关系和流事实，给出对每个基本块执行次数的约束，采用整数线性规划（integer linear program, ILP）方法，最大化整个程序的执行时间表达式，该方法被称为隐式路径枚举技术（implicit path enumeration technique, IPET）.

2 TTI 程序设计流程

本节给出 TTI 程序相关概念，以及设计 TTI 程序的基本流程.

2.1 相关概念

由 1.1 节 TTI 指令集的实现可知，TTI 指令集中时间语义指令在 RPU 流水线执行段的行为会受到指令预设系统时间和当前系统时间寄存器 ctr 的值的

影响, 编程人员需要依据系统时间设计 TTI 程序的时间行为. 下面给出 TTI 程序中“系统时间”这一概念的定义.

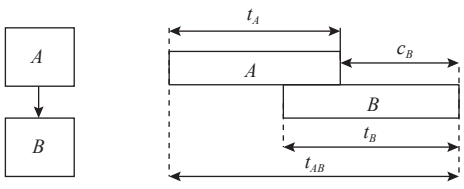
定义 1. 系统时间. TTI 程序中系统时间是以系统时刻为单位的离散值, 系统时刻指的是一段实时间, 系统时刻的大小由实时应用需要的时间粒度决定, TTI 程序时间语义指令中的预设时间均以系统时刻为单位, 根据不同需求, 不同 TTI 程序的系统时刻也不相同.

如果实时应用需要的时间粒度为 1 ms, 即 1 个系统时刻为 1 ms, 那么系统时间 20 对应的实时间为 20 ms. RPU 使用标准时钟(stdClk), 结合时间粒度寄存器 tgr, 更新系统时间寄存器 ctr 的值. 对于实时应用规定的时间粒度, TTI 程序可以使用 settg 指令设置时间粒度寄存器 tgr, tgr 的值应该设置为 $tg \times f$, 其中 tg 为实时应用需要的时间粒度, f 为 stdClk 的频率, 假设 stdClk 的频率为 1 MHz, 而实时应用需要的时间粒度为 1 ms, 则时间粒度寄存器 tgr 应该设置为 0.001 s×1 MHz, 即 tgr 值为 1000. 假设实时应用的时间粒度需求变得更为紧致, 为 0.1 ms, 那么 tgr 应该设置为 100.

除了控制 TTI 程序时序行为的 stdClk, RPU 存在驱动数据通路各个功能部件同步运行的 cpuClk, 根据文献 [4] 中给出的“stdClk 和 cpuClk 关系定理”可知, stdClk 和 cpuClk 存在同步和异步 2 种关系, 在本文中只考虑同步情况, 即 stdClk 的周期 T_{stdClk} 和 cpuClk 的周期 T_{cpuClk} 的关系为 $T_{\text{stdClk}} = n \times T_{\text{cpuClk}}$.

TFG+使用不同节点表示 TTI 程序中的时间语义指令和普通指令构成的基本块. 由于 RPU 流水线中同时存在多条指令, TFG+中 2 个连续节点对应的时间语义指令或基本块的执行会有所重叠. 如图 1 所示, 基本块 A 的执行时间为 t_A, 基本块 B 的执行时间为 t_B, 基本块 A 和基本块 B 的整体执行时间为 t_{AB}, $t_A + t_B > t_{AB}$. 文献 [5] 给出了执行开销的概念, 现在结合 TFG+进行说明.

定义 2. 执行开销. v_j 为 TFG+中节点, v_i 为 v_j 的直接前驱节点, v_j 对应程序段的执行开销为 v_i 最后一条指令完成时间到 v_j 最后一条指令完成时间的间隔. 例如, 在图 1 中, 基本块 B 的执行开销为 c_B.



(a) 程序示意图 (b) 程序执行时间图

Fig. 1 Example of execution cost

图 1 执行开销示例

本文采用执行图[7]的方法进行低级分析, 以计算 TFG+各个节点的执行开销. 根据定义 2, 构建 TFG+各节点执行图时需要增加前驱节点的最后一条指令的执行信息.

2.2 TTI 程序设计流程

TTI 程序的设计流程有 3 步:

1)对于给出的高层编程模型, 分离其时间语义和功能语义, 对于上层模型的时间语义, 选择合适的 TTI 程序设计范式进行表达, 再结合相应的功能代码, 给出对应的 TTI 程序.

2)构建 TTI 程序的 TFG+, 表达 TTI 程序中的控制流信息、用户规定的程序时间行为以及平台相关的时间属性.

3)基于 RPU 或其他可以运行 TTI 指令集的处理器, 结合 TFG+中控制流信息, 构建 TFG+每个节点的执行图, 基于执行图分析 TFG+中每个节点平台相关的时间属性, 并根据 TFG+用户规定的程序时间行为, 进行时间安全性检查.

由于 RPU 采用的是顺序取指、顺序执行的模式, ttiat, ttoat, delay, mtfd 指令的执行会受到前驱代码段执行时间的影响, 时间安全性检查是为了保证系统运行时, 这些指令的时间语义能正确表达. 时间安全性检查会分析时间语义指令前驱代码段最坏情况下的执行时间, 如果前驱代码段的最坏情况下执行时间较长, 执行到时间语义指令时系统时间大于预设时间, 此时 mtfd 指令约束代码段执行时间上界的语义被违反, 且 ttiat, ttoat 指令在规定系统时刻完成输入输出操作的语义也被违反; 如果前驱代码段执行时间较短, 执行到时间语义指令时, 系统时间小于指令预设时间, 显然满足 mtfd 指令约束代码段执行时间上界的语义, 而 ttiat, ttoat, delay 指令会停滞流水线, 直到 RPU 的系统时间等于指令预设的时间, 此时满足 delay 指令约束代码段执行时间下界的语义, 由于 ttiat, ttoat 指令语义要求在规定系统时刻完成输入输出操作, 所以还需要验证 ttiat, ttoat 指令的输入输出操作会在规定的系统时刻内完成.

3 TTI 程序设计范式

TTI 指令集可以通过不同的指令组合表达丰富的时间语义, 用来满足用户对于程序时间行为的控制需求. 现在给出 TTI 指令集能够表达的 4 种时间

语义：

1）绝对点时间语义（absolute timed semantic）. 相对于系统 0 时刻的点时间约束，即规定自系统启动到某一确定时间点的系统行为.

2）绝对段时间语义（absolute timing semantic）. 相对于系统 0 时刻的段时间约束，即规定自系统启动，某一段时间的系统行为.

3）相对点时间语义（relative timed semantic）. 相对于某一时刻的点时间约束，即自系统某一时刻到规定时间点的系统行为.

4）相对段时间语义（relative timing semantic）. 相对于某一时刻的段时间约束，即自系统某一时刻，规定时间段内的系统行为.

相对时间语义中的“某一时刻”，分为 2 种情况：

1）设计时确定的时刻. 在系统设计时，已知该时刻的系统时间. 例如，相对时刻是规定的绝对点时间，那么这里相对时间语义中的相对时刻就是系统设计时已知，此时的相对时间语义可以转化成绝对时间语义，即绝对时间等于相对的系统时间加上相对语义预设的系统时间.

2）运行时确定的时刻. 在系统设计时，不能确定该时刻的系统时间，系统时间需要根据系统实际运行情况分析. 例如，相对时刻是系统某一数据输入时刻，该数据输入对应的指令执行完成时间会受到前驱代码段的影响，数据输入的系统时间只有在系统运行时才能确定. 此时的相对时间语义，不能转化成绝对时间语义.

下面介绍 TTI 指令集对上述时间语义的实现，给出 TTI 程序设计范式和 TTI 程序设计规则.

3.1 绝对点时间

TTI 指令集中，可以通过 setti 设置 ctr 寄存器来设置系统时间. 整个系统运行前的准备工作完成后，需要使用 setti *x*0 指令，将系统时间置为 0，表示系统开始运行，ttiat/ttoat 指令提供了定时输入/输出语义，可以使用 setti *x*0 指令和 ttiat/ttoat 指令的组合，完成绝对点时间语义的表达.

图 2 给出绝对点时间的表达示例和实际运行的时序. 如图 2（a）所示，init 负责初始化系统，设置时间粒度寄存器 tgr 等，完成后使用 setti *x*0 将 ctr 设置为 0，表示系统启动. task1 中，设计时需要在系统时间为 20 时进行输出，所以使用 ttoat 指令，并将指定触发时间的寄存器（图 2（a）中寄存器 *a*0）设置为 20.

如图 2（b）所示，init 任务初始化系统，并通过 setti 指令将 ctr 寄存器设置为 0，系统启动，task1 在某一时刻调度执行，且 tast1 中 ttoat 指令在系统时间为 20 时生效，即 ttoat 指令的输出操作发生在系统时间 20 和 21 之间. 如果将 ttoat 指令替换成 ttiat 指令，则表示输入操作的发生在系统时间 20 和 21 之间.

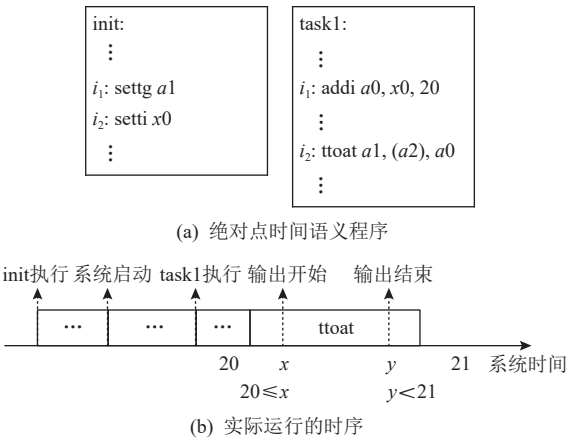


Fig. 2 Example of absolute timed semantic

图 2 绝对点时间语义示例

3.2 绝对段时间

绝对段时间和绝对点时间类似，在系统初始化完成后，通过 setti *x*0 指令将系统时间置为 0，表示系统启动，然后在需要限制的代码段后使用 delay/mtfd 指令，表达绝对段时间约束.

图 3 给出绝对段时间的表达示例和实际运行的时序. 如图 3（a）所示，init 负责初始化并启动系统. task2 中，设计时规定 delay 指令前程序执行时间最少为 20，所以将寄存器 *a*0 设置为 20，使用 delay 指令，控制程序的绝对段时间行为.

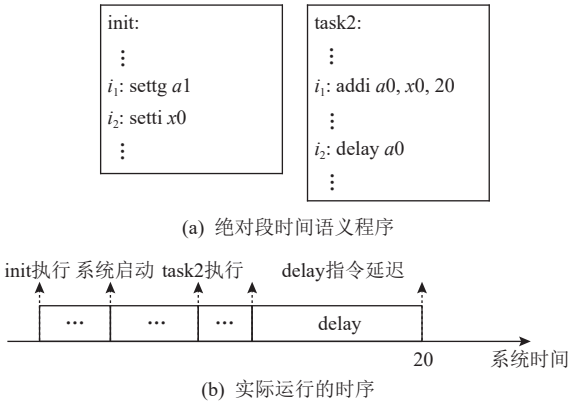


Fig. 3 Example of absolute timing semantic

图 3 绝对段时间语义示例

如图 3（b）所示，task2 在某一时刻调度执行，且 task2 中 delay 指令需要保证前驱代码段的执行时间大于等于 20. 如果将 delay 指令替换成 mtfd 指令，需要保证前驱代码段的执行时间小于 20.

3.3 相对点时间

用户需要表达相对于系统某一时刻的点时间行为. 这里的“某一时刻”, 可以是设计时确定的系统时刻, 这样相对点时间语义就可以转化为绝对点时间语义, TTI 指令的实现可以转化为上述绝对点时间语义的实现方式. 所以这里主要分析的“某一时刻”, 指的是系统运行前无法确定系统时间的时刻, 可以使用 TTI 指令集中提供的 getti 指令和 ttiat/ttoat 指令组合的方式表达. 图 4 给出此种情况下, 相对点时间的表达示例和实际运行的时序.

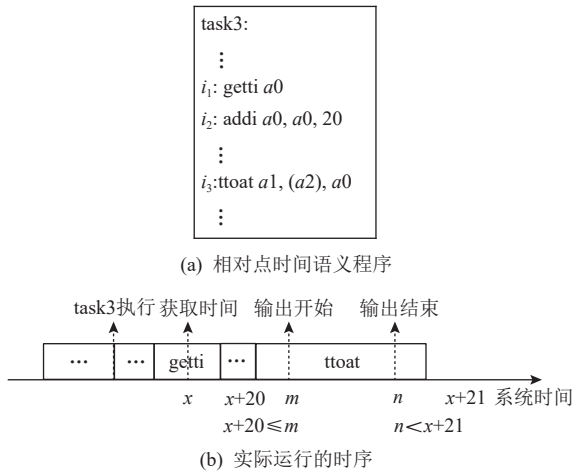


Fig. 4 Example of relative timed semantic

图 4 相对点时间语义示例

如图 4(a)所示, 在 task3 中, 用户需要相对 getti 指令 20 个时间单位执行输出操作. TTI 指令对此种语义的实现为: 设置 getti 指令, 获取当前系统时间, 存放在 $a0$ 中, 再通过 addi 指令加上相对时间 20, 作为 ttoat 指令的触发时间.

如图 4(b)所示, 由于 getti 前程序段的执行时间不确定, 所以 getti 获取的系统时间也需要根据实际运行情况确定, 假设的 getti 执行时系统时间为 x, ttoat 指令也会在相对于 getti 指令 20 个时间单位的 x+20 生效.

3.4 相对段时间

用户需要表达相对于系统某一时刻在一段时间内的程序行为. “某一时刻”和上述相对点时间中的一样, 这里主要分析在运行时确定的系统时刻. 图 5 给出此类情况下, 相对段时间的表达示例和实际运行的时序.

如图 5(a)所示, 在 task4 中, 用户需要 getti 指令后的代码段执行至少 20 个时间单位. TTI 指令对此种语义的实现和相对点时间语义类似, 只需要把 ttoat 指令改为 delay 指令.

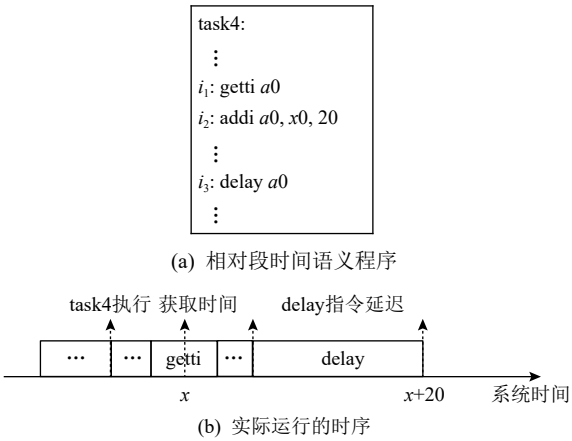


Fig. 5 Example of relative timing semantic

图 5 相对段时间语义示例

如图 5(b)所示, 可以表达 getti 指令和 delay 指令间的代码段至少执行 20 个时间单位.

3.5 TTI 程序设计规则

考虑时间语义指令实际使用场景和 TTI 程序时间安全性检查的可行性, TTI 程序需要遵守 2 个规则:

1) 为保证 TTI 程序的时间行为的确定性, 时间语义指令集不会出现在程序的某一分支中.

2) 对于运行时确定的相对时间语义, getti 指令和对应的时间语义指令之间只有通用计算指令, 不会出现其他时间语义指令.

4 TFG+定义

TTI 程序时间行为分析和时间安全性检查, 需要构建整个程序的抽象表示. 本文对 TFG 扩展, 以 TFG+表示 TTI 程序, TFG+可以表达 TTI 程序的控制流信息、用户规定的时间行为以及平台相关的时间属性. 如表 3 所示, TFG+由 3 部分组成: 图结构、程序属性、平台相关属性.

1)图结构. 使用不同节点表示多种时间语义和 TTI 程序其他组成部分, 主要分为: ①绝对时间语义节点, 包括绝对段时间语义节点(absolute timing point)、绝对点时间语义节点(absolute timed point); ②运行时确定的相对时间语义节点, 首先是相对时间语义的相对基准点(relative base point), 然后分别是相对段时间语义节点(relative timing point)、相对点时间语义节点(relative timed point); ③普通指令构成的基本块(basic block)节点; ④程序的开始(*start*)节点. 最后, 由 TTI 程序中的指令执行顺序, 构成 TFG+中的边. ttiat, ttoat 指令根据表达的语义, 对应绝对(相对)点时间语义节点; delay, mtfd 指令根据表达

Table 3 TFG+ Definition

表 3 TFG+定义

组成部分	具体构成
图结构	ATP，绝对段时间语义节点集合. $ATdP$，绝对点时间语义节点集合. RTP，相对段时间语义节点集合. $RTdP$，相对点时间语义节点集合. RBP，相对基准点集合. BB，基本块节点集合. $start$，开始节点. $V=\{start\}\cup ATP\cup ATdP\cup RTP\cup RTdP\cup RBP\cup BB$，节点集合. $E=V\times V$，边集合.
程序属性	t_{tg}，表示时间粒度寄存器的值. t_{start}，$\{start\}\to\mathbb{N}$，映射开始节点到任务开始执行的系统时间. t_{ATP}，$ATP\to\mathbb{N}$，映射 ATP 中节点到系统时间. t_{ATdP}，$ATdP\to\mathbb{N}$，映射 $ATdP$ 中节点到系统时间. t_{RTP}，$RTP\to\mathbb{N}$，映射 RTP 中节点到系统时间. t_{RTdP}，$RTdP\to\mathbb{N}$，映射 $RTdP$ 中节点到系统时间.
平台相关属性	c_f，$BB\to\mathbb{N}$，映射基本块节点到执行开销. c_t，$ATP\cup ATdP\cup RTP\cup RTdP\to\mathbb{N}$，时间语义节点对应指令的前驱代码段最坏执行情况的前提下，映射时间语义节点到执行开销. c_d，$ATP\cup ATdP\cup RTP\cup RTdP\to\mathbb{N}$，时间语义节点对应指令的前驱代码段最坏执行情况的前提下，映射时间语义节点到延迟时间. c_{off}，$\{start\}\cup ATdP\to\mathbb{N}$，映射开始节点到任务开始时间相对于某一系统时刻的偏移时间；绝对点时间语义节点对应指令的前驱代码段最坏执行情况的前提下，映射绝对点时间语义节点到节点对应指令执行完成相对于某一系统时刻的偏移时间.

的语义，对应绝对(相对)段时间语义节点；getti 指令对应相对基准点.

2)程序属性. 表示 TTI 程序平台无关的时间属性，TFG+程序属性中提到的时间均为系统时间. 程序属性包括时间粒度寄存器的值、TTI 程序的开始时间、绝对(相对)点(段)时间语义节点对应的预设系统时间.

3)平台相关属性. TTI 程序平台相关的时间属性，TFG+平台相关属性中提到的执行开销和时间均以 cpuClk 周期为单位. 对于普通指令构成的基本块，需要计算每个基本块的执行开销. 对于时间语义节点，该节点对应的时间语义指令需要根据前驱代码段的运行时间，决定时间语义指令的执行开销，为了保证该类指令的时间语义在运行时的正确表达，现在考虑在指令前驱代码段最坏执行情况的前提下，时间语义指令需要的延迟时间和执行开销. 由于 TTI 程序的时序依赖于 stdClk，而逻辑控制依赖于 cpuClk，且 stdClk 周期 T_{stdClk} 为 cpuClk 周期 T_{cpuclk} 的整数倍，所以 1 个系统时刻内包含多个 cpuClk 周期，所以对于绝对点时间语义节点，需要给出在该节点对应的绝对点时间语义指令前驱代码段最坏执行情况的前提下，绝对点时间指令执行完成相对于某一系统时刻的偏移时间，同理，需要给出任务开始执行相对于某一系统时刻的偏移时间. 如图 6 所示的平台相关属性 c_{off} 说明，假设存在绝对点时间语义节点 $v_{atdp}\in ATdP$，且 v_{atdp} 对应的指令为 i_{atdp}，在 i_{atdp} 的前驱代码段最坏执行情况的前提下，i_{atdp} 在系统时间 x 执行完成，而实际上 i_{atdp} 会在系统时间 x 内的某一时间点完成，$c_{off}(v_{atdp})$ 为图 6 中所示 v_{atdp} 对应指令执行完成相对于系统时间 x 的偏移.

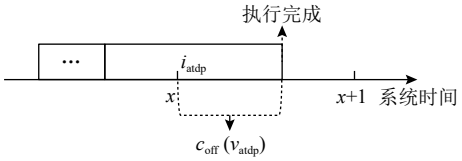


Fig. 6 Description of platform dependent property c_{off}

图 6 平台相关属性 c_{off} 说明

如算法 1 所示，给出 TFG+构建算法.

算法 1. TFG+构建算法.

输入：TTI 程序 $prog$；

输出：时间语义节点集合 ATP, $ATdP$, RTP, $RTdP$, 相对基准点集合 RBP, 基本块节点集合 F, 边集合 E.

① $(N, E)\leftarrow f_{CFG}(prog)$；

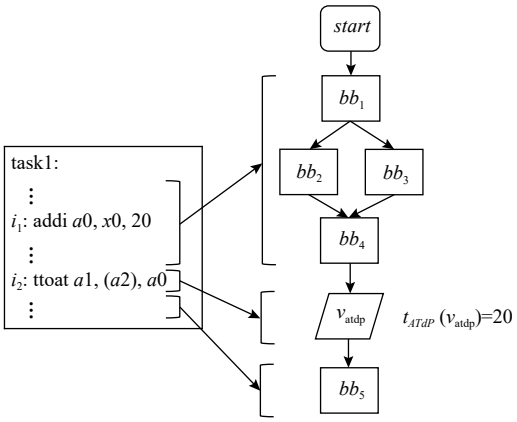
② for N 中的每个节点 i

③ $(ATP, ATdP, RTP, RTdP, RBP, F, E)\leftarrow partition(i)$；

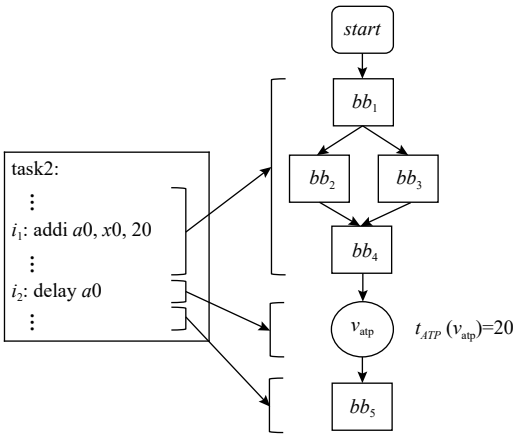
④ end for

算法 1 中，行①对应的 $f_{CFG}(prog)$，将 $prog$ 中时间语义指令当作普通指令构建 CFG，输出 CFG 中的节点集合 N 和边集合 E. 行③中的 $partition(i)$，对 CFG 节点进一步划分，节点 i 中的时间语义指令(ttiat, ttoat, delay, mtfd)以及获取系统时钟指令(getti)划分为单独节点，并将划分后的节点和边加入对应集合，输出 i 划分后的各种时间语义节点集合、相对基准点集合、基本块节点集合以及边集合.

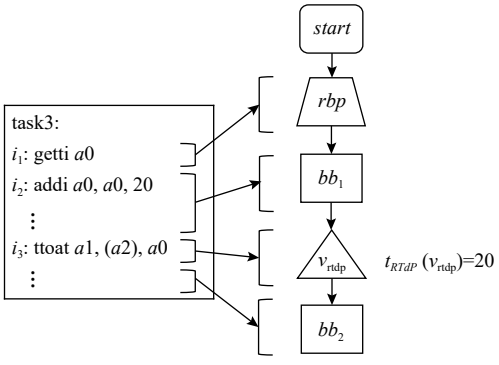
图 7 为具有不同时间语义的 TTI 程序片段和对应的部分 TFG+. 图 7(a)TTI 程序中使用 ttoat 指令，表达绝对点时间语义，在 TFG+中，ttoat 指令对应绝对点时间语义节点，使用平行四边形表示，普通代码段根据程序结构，划分成基本块，对应基本块节点，使用矩形表示. 图 7(b)使用 delay 指令，表达绝对段时间语义，使用圆形表示. 图 7(c)使用 getti 指令和 ttoat 指令组合，表达运行时确定的相对点时间语义，getti 对应相对基准点，使用梯形表示，ttoat 指令对应相对点时间语义节点，使用三角形表示. 图 7(d)使用 getti 指令和 delay 指令组合，表达运行时确定的相对段时间语义，getti 对应相对基准点，delay 对应相对段时间语义节点，使用菱形表示.



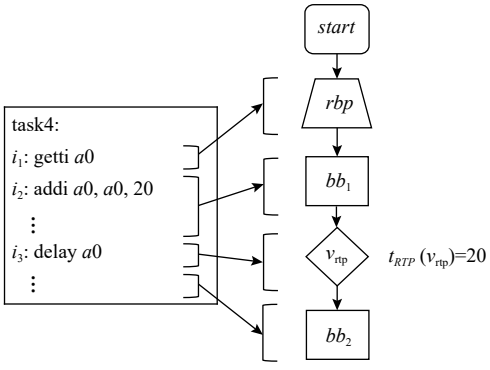
(a) 绝对点时间语义



(b) 绝对段时间语义



(c) 相对点时间语义



(d) 相对段时间语义

Fig. 7 Examples of TFG+

图 7 TFG+示例

传统程序的 CFG 只能表达程序中的控制流, 即指令执行的时间先后关系(temporal succession); TFG 在 CFG 的基础上扩展时间点(timing point), 可以表达代码段的时间行为; TFG+相比 TFG 加入了点时间语义节点, 可以表达时间和操作相结合的点时间行为. 所以 TFG+的表达能力强于 TFG 和 CFG, 可以表达更为丰富的时间语义.

5 时间行为分析和时间安全性检查

针对文献 [4] 提到的 RPU 的硬件结构, 使用基于执行图的低级分析方法, 计算 TFG+基本块节点和时间语义节点的平台相关属性, 同时对时间语义节点进行时间安全性检查.

5.1 基于 RPU 的执行图

RPU 为 5 段流水处理器, 对应的流水段的集合为 S={IF, ID, EX, MM, WB}, 分别对应取指段、译码段、执行段、访存段、写回段. TFG+中每个节点对应的指令集合为 I={i_1, i_2, …, i_n}. 执行图的节点集合为 V_{XG}, 且 $V_{XG}\subseteq S\times I$, 例如, 对于($IF$, i_j) $\in V_{XG}$, 表示指令 $i_j \in I$ 处于流水段 $IF \in S$. 执行图的边集合为 E_{XG}, 且 $E_{XG}\subseteq V_{XG}\times V_{XG}$, 表示执行图节点间的时序依赖. 执行图的边分为 2 类, 如图 8(b)所示, 为了区分这 2 类边, 在执行图中使用实线和虚线分别表示, 且 $\delta_{(v,i_m)\to(w,\,i_n)}\in\{0,1\}$表示 2 种边, 对于实线边 $\delta_{(v,i_m)\to(w,\,i_n)}=1$, 对于虚线边 $\delta_{(v,i_m)\to(w,\,i_n)}=0$.

现有 $v \in S$, $w\in S$, $i_m\in I$, $i_n\in I$, 在执行图中存在节点(v, i_m)$\in V_{XG}$, (w, i_n)$\in V_{XG}$, 且(v, i_m) $\to$ (w, i_n) $\in E_{XG}$, 基于此详细说明这 2 类边.

1) (w, i_n)只可以在(v, i_m)完成后开始, 此类边用实线表示, 且 $\delta_{(v,i_m)\to(w,\,i_n)}=1$. 具体来说分为 4 种情况. ①对于多条指令, RPU 顺序发射、顺序执行, 即(IF, i_{m-1}) $\to$ (IF, i_m), (ID, i_{m-1}) $\to$ (ID, i_m), (EX, i_{m-1}) $\to$ (EX, i_m), (MM, i_{m-1}) $\to$ (MM, i_m), (WB, i_{m-1}) $\to$ (WB, i_m), 其中 i_{m-1} 和 i_m 为 2 条连续指令. ②对于某一条指令, RPU 顺序执行, 即(IF, i_m) $\to$ (ID, i_m), (ID, i_m) $\to$ (EX, i_m), (EX, i_m) $\to$ (MM, i_m), (MM, i_m) $\to$ (WB, i_m). ③数据冒险. RPU 使用数据旁路解决大部分数据冒险, 而针对只能由延迟流水线解决的数据冒险, 即 2 条连续指令, 第 1 条是访存输入指令, 第 2 条指令需要使用第 1 条指令的访存结果, 需要在执行图中对此种情况构建相应的依赖关系, 即存在边(MM, i_{m-1}) $\to$ (EX, i_m). ④控制冒险. 采用阻塞流水线的方式解决控制冒险, 对于一条分支跳转指令, 下一条指令的取指需要延

迟到分支指令在执行段产生结果后，即存在边$(EX, i_{m-1}) \to (IF, i_m)$.

2）(w, i_n)可以和(v, i_m)同时开始，但是不能早于(v, i_m)，此类边用虚线表示，且$\delta_{(v,i_m)\to(w,i_n)}=0$. 此类边分为 4 类，$(ID, i_{m-1}) \to (IF, i_m)$，$(EX, i_{m-1}) \to (ID, i_m)$，$(MM, i_{m-1}) \to (EX, i_m)$，$(WB, i_{m-1}) \to (MM, i_m)$，其中 i_{m-1} 和 i_m 为 2 条连续指令.

算法 2 给出基于 RPU 的执行图构建算法.

算法 2. 基于 RPU 的执行图构建算法.

输入：程序段 *prog*；

输出：执行图节点集合 V_{XG}，执行图边集合 E_{XG}.

① $V_{XG} \leftarrow f_{vertex}(prog)$；

② for V_{XG} 中每个节点(v, i_m)

③ if $v = IF$

④ $E_{XG} \leftarrow \{(IF, i_m) \to (IF, i_{m+1}), (IF, i_m) \to (ID, i_m)\} \cup E_{XG}$；

⑤ else

⑥ $E_{XG} \leftarrow \{(v, i_m) \to (v, i_{m+1}), (v, i_m) \to (f_{nextstage}(v), i_m), (v, i_m) \to (f_{prestage}(v), i_{m+1})\} \cup E_{XG}$；

⑦ end if

⑧ if i_m 与 i_{m+1} 存在数据冒险

⑨ $E_{XG} \leftarrow \{(MM, i_m) \to (EX, i_{m+1})\} \cup E_{XG}$；

⑩ end if

⑪ if i_m 和 i_{m+1} 存在控制冒险

⑫ $E_{XG} \leftarrow \{(EX, i_m) \to (IF, i_{m+1})\} \cup E_{XG}$；

⑬ end if

⑭ end for

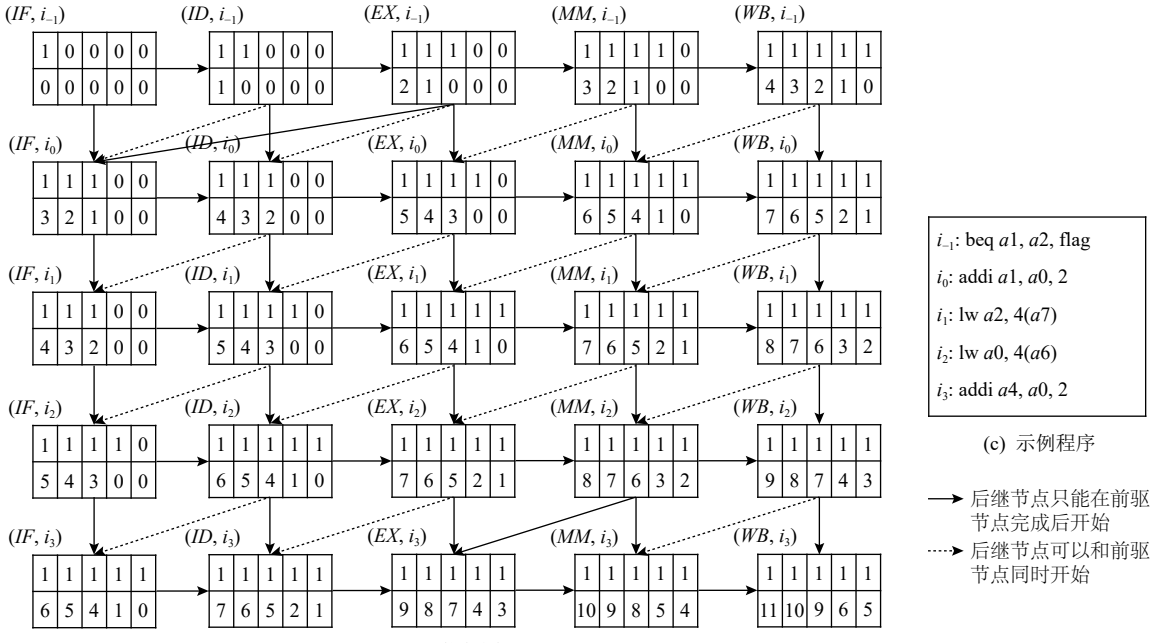
算法 2 中，行① $f_{vertex}(prog)$构建执行图中所有的节点，行④和行⑥中构建多条指令顺序发射、顺序执行以及一条指令顺序执行相关的边，行⑥中$f_{nextstage}(v)$表示流水段 v 的下一流水段，$f_{prestage}(v)$表示流水段 v 的上一流水段，并且除了$(v, i_m) \to (f_{prestage}(v), i_{m+1})$的$\delta_{(v,i_m)\to(f_{prestage}(v),\, i_{m+1})}=0$，其余边均为$\delta_{(v,i_m)\to(w,i_n)}=1$.

算法 2 仅适用于 RPU，对于其他支持 TTI 指令集的不同硬件架构，需要根据具体的架构特点对算法 2 进行修改. 例如，可以通过修改流水段集合适应不同流水线级数的处理器，对于指令间由于资源竞争而产生的结构冒险问题可以在执行图中添加相应类型的依赖边来体现等.

如图 8 所示，具体的程序段和对应的执行图，假设 TFG+中节点包含指令 i_0, i_1, i_2, i_3, 计算节点执行开

节点属性	*IF*	*ID*	*EX*	*MM*	*WB*
$e^{r}_{(v,i_j)}$	$e^{IF}_{(v,i_j)}$	$e^{ID}_{(v,i_j)}$	$e^{EX}_{(v,i_j)}$	$e^{MM}_{(v,i_j)}$	$e^{WB}_{(v,i_j)}$
$d^{r}_{(v,i_j)}$	$d^{IF}_{(v,i_j)}$	$d^{ID}_{(v,i_j)}$	$d^{EX}_{(v,i_j)}$	$d^{MM}_{(v,i_j)}$	$d^{WB}_{(v,i_j)}$

(a) 执行图节点属性



(b) 执行图　(c) 示例程序

Fig. 8　Example of execution graph

图 8　执行图示例

销需要考虑前驱节点最后一条指令 i_{-1}, 所以构建 i_{-1}, i_0, i_1, i_2, i_3 的执行图. i_{-1} 和 i_0 存在控制冒险, 所以存在边$(EX, i_{-1}) \rightarrow (IF, i_0)$, i_2 和 i_3 存在数据冒险, 所以存在边$(MM, i_2) \rightarrow (EX, i_3)$.

5.2 执行图节点属性

对于整个执行图来说, 资源集合 R={*IF*, *ID*, *EX*, *MM*, *WB*}表示和节点执行相关的所有资源的集合, $A=\{a^r \mid r \in R\}$表示对于当前执行图中所有资源的释放时间.

现在给出执行图中的每个节点的属性, 以及每个属性的计算过程, 各属性中提到的时间均以 cpuClk 周期为单位.

假设流水段 v、指令 i_m, 对应执行图中节点(v, i_m).

1) $\lambda_{(v,i_m)}$, 表示指令 i_m 在流水段 v 从就绪到完成需要花费的时间.

计算过程为: 如表 4 所示, 给出 RPU 各类指令在各个流水段花费的时间. 时间指令 ttiat, ttoat, delay, 需要在流水线的执行段判断系统时间是否到达预设的时间, 如果没有到达预设时间, 要进行延时操作, 所以在执行段存在的延迟时间τ需要根据前驱代码段的执行时间进行判断. 而 mtfd 指令没有具体实现, 所以不考虑 mtfd 在每个流水段的执行时间. RPU 采用理想的存储架构, 即访问指令存储器和数据存储器均为 1 个 cpuClk 周期, 对于其他类型的存储架构, 可以进行额外分析, 确定指令在各流水段的执行时间, 再进行后续的计算. 例如, 假设需要考虑指令 Cache 的命中或缺失, 且指令 Cache 命中时的访存时间为 1 个 cpuClk 周期, 系统设计者可以通过指令 Cache 分析, 确定每条指令的 Cache 命中情况, 若 Cache 命中, 则该指令在取指段的执行时间为 1 个 cpuClk 周期, 若 Cache 缺失, 则该指令在取值段的执行时间将会大于 1 个 cpuClk 周期, 所以表 4 中所示

Table 4 Execution Time of the Instructions in Each Pipeline Stage of RPU

表 4 指令在 RPU 各流水段的执行时间

指令类型	*IF*	*ID*	*EX*	*MM*	*WB*
通用计算指令	1^+	1	1	1	1
分支指令	1^+	1	1	1	1
访存指令	1^+	1	1	1^+	1
标准时钟管理指令	1^+	1	1	1	1
ttiat/ttoat	1^+	1	τ+1	1^+	1
delay	1^+	1	τ+1	1^+	1
mtfd					

的所有指令在取指段的执行时间为 1^+, 表示取值时间最小为 1 个 cpuClk 周期, 而具体值需要根据分析确定, 同理, 访存指令和 ttiat/ttoat 指令的在访存段的执行时间也为 1^+, 具体的访存和定时输入输出的时间需要结合实际运行情况分析确定.

2) $E_{(v,i_m)} = \{e^r_{(v,i_m)} \in \{0,1\} \mid r \in R\}$, 表示处在流水段 v 的指令 i_m 开始执行是否会受到 R 中某个资源的影响.

计算过程为: 如果节点(v, i_m)本身执行依赖于资源 $r \in R$, 那么 $e^r_{(v,i_m)}$=1. 另外, 如果节点(v, i_m)任意前驱节点(x, i_j), 即$(x, i_j) \rightarrow (v, i_m) \in E_{\mathrm{XG}}$, 存在 $e^r_{(x,i_j)}$=1, 则 $e^r_{(v,i_m)}$=1.

3) $D_{(v,i_m)} = \{d^r_{(v,i_m)} \mid r \in R\}$, 表示资源释放到指令在流水段就绪执行之间的延迟时间, 即 R 中某个资源释放与处在流水段 v 的指令 i_m 开始执行的时间间隔.

计算过程为: 对于资源 $r \in R$, 考虑(v, i_m)的所有前驱节点(x, i_j), 即$(x, i_j) \rightarrow (v, i_m) \in E_{\mathrm{XG}}$, 从资源 r 释放到节点(v, i_m)就绪的延迟时间为

$$d^r_{(v,i_m)} = \max_{(x,i_j)\rightarrow(v,i_m)} (e^r_{(x,i_j)} \times (d^r_{(x,i_j)} + \delta_{(x,i_j)\rightarrow(v,i_m)} \times \lambda_{(x,i_j)})).$$

4) $\rho_{(v,i_m)}$, 表示指令 i_m 在流水段 v 的就绪时间, 即指令 i_m 在流水段 v 可以开始执行的时间.

计算过程为: 节点(v, i_m)的就绪时间为

$$\rho_{(v,i_m)} = \max_{r \in R} (e^r_{(v,i_m)} \times (d^r_{(v,i_m)} + a^r)).$$

5.3 平台相关属性分析和时间安全性检查

整个分析和验证流程如图 9 所示, 首先分析 TFG+中由普通指令构成的基本块节点的平台相关属性; 再结合 TFG+中的控制流信息分析时间语义节点的平台相关属性; 最后, 根据每个时间语义节点的语义要求以及平台相关属性的分析结果, 对每个时间语义节点进行时间安全性检查, 验证 TTI 程序运行

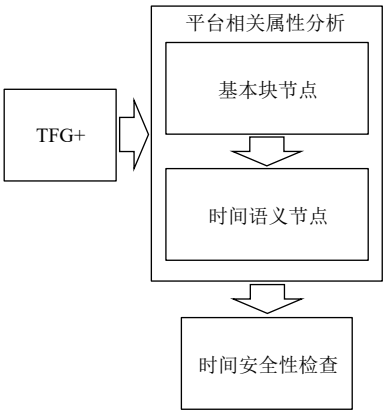


Fig. 9 Flow graph of temporal behavior analysis and semantic verification

图 9 时间行为分析和语义验证流程图

时指令的语义能否正确表达.

下面分别介绍基本块节点和时间语义节点的平台相关属性确定和时间安全性检查的流程.

1)基本块节点

对于基本块节点, 只需要确定基本块的执行开销. 首先对 TFG+中每个基本块节点构建执行图, 再根据 5.2 节中执行图节点属性, 确定基本块的执行开销.

定理 1. 基本块节点执行开销定理. 基本块节点 $bb\in BB$, 基本块最后一条指令为 i_n, 基本块前一条指令为 i_{-1}, 整个基本块的执行开销为

$$c_{\mathrm{f}}(bb)\leqslant\max_{r\in R}((d^{r}_{(WB,i_n)}-d^{r}_{(WB,i_{-1})})+(\lambda_{(WB,i_n)}-\lambda_{(WB,i_{-1})})).\quad(1)$$

证明. 基本块最后一条指令为 i_n, 基本块前一条指令为 i_{-1}, 当前基本块的执行开销为

$$c_{\mathrm{f}}(bb)=(\rho_{(WB,i_n)}+\lambda_{(WB,i_n)})-(\rho_{(WB,i_{-1})}+\lambda_{(WB,i_{-1})}),$$

即

$$c_{\mathrm{f}}(bb)=(\rho_{(WB,i_n)}-\rho_{(WB,i_{-1})})+(\lambda_{(WB,i_n)}-\lambda_{(WB,i_{-1})}).$$

假设, 对于节点(WB, i_{-1})和(WB, i_n)的关键资源分别为$\alpha\in R$, $\beta\in R$, 即

$$\rho_{(WB,i_{-1})}=d^{\alpha}_{(WB,i_{-1})}+a^{\alpha},$$

$$\rho_{(WB,i_n)}=d^{\beta}_{(WB,i_n)}+a^{\beta},$$

$$\rho_{(WB,i_n)}-\rho_{(WB,i_{-1})}=d^{\beta}_{(WB,i_n)}-d^{\alpha}_{(WB,i_{-1})}+a^{\beta}-a^{\alpha}.$$

由于 RPU 为顺序执行的 5 段流水线, 对于资源集合 $R=\{IF, ID, EX, MM, WB\}$, (WB, i_{-1})会依赖于所有的 5 个资源. 由此可得

$$d^{\alpha}_{(WB,i_{-1})}+a^{\alpha}\geqslant d^{\beta}_{(WB,i_{-1})}+a^{\beta},$$

$$a^{\beta}\leqslant d^{\alpha}_{(WB,i_{-1})}+a^{\alpha}-d^{\beta}_{(WB,i_{-1})},$$

$$\rho_{(WB,i_n)}-\rho_{(WB,i_{-1})}\leqslant d^{\beta}_{(WB,i_n)}-d^{\beta}_{(WB,i_{-1})},$$

$$c_{\mathrm{f}}(bb)\leqslant(d^{\beta}_{(WB,i_n)}-d^{\beta}_{(WB,i_{-1})})+(\lambda_{(WB,i_n)}-\lambda_{(WB,i_{-1})}),$$

$$c_{\mathrm{f}}(bb)\leqslant\max_{r\in R}((d^{r}_{(WB,i_n)}-d^{r}_{(WB,i_{-1})})+(\lambda_{(WB,i_n)}-\lambda_{(WB,i_{-1})})).$$

考虑基本块前驱指令为时间语义指令的情况, 即 i_{-1} 为 ttiat, ttoat, delay, 对于时间语义指令 i_{-1} 在执行段产生的延迟, 以$\tau_{(EX,i_{-1})}$表示, 由于指令 i_{-1} 在执行段本身存在 1 个时钟周期的执行时间, 故节点(EX, i_{-1})执行时间$\lambda_{(EX,i_{-1})}=1+\tau_{(EX,i_{-1})}$, 而执行图$(WB, i_{-1})$, (WB, i_n)节点的$D_{(WB,i_{-1})}$和$D_{(WB,i_n)}$均会受到$\tau_{(EX,i_{-1})}$的影响, 所以最后执行开销的计算不会因为延迟时间$\tau_{(EX,i_{-1})}$而产生额外的问题. 证毕.

2)时间语义节点

TFG+中时间语义节点的平台相关属性分析和时间安全性检查, 沿着 TFG+控制流进行, 所以在下述分析中, 假设当前时间语义节点的所有前驱时间语义节点均完成平台相关属性分析和时间安全性检查.

首先介绍时间语义节点分析需要的映射函数.

①$next$: $\{start\}\cup ATP\cup ATdP\cup RTP\cup RTdP\cup RBP\rightarrow V$, 映射时间语义节点、开始节点、相对基准点到邻接后继节点.

② pre: $ATP\cup ATdP\cup RTP\cup RTdP\cup RBP\rightarrow V$, 映射时间语义节点、相对基准点到邻接前驱节点.

③$tpre$: $ATP\cup ATdP\cup RTP\cup RTdP\cup RBP\rightarrow ATdP\cup\{start\}$, 映射时间语义节点、相对基准点到距该节点最近的一个前驱绝对点时间语义节点, 如果没有这样的前驱绝对点时间语义节点, 则映射到开始节点.

④t_{jitter}: $ATP\cup ATdP\cup RTP\cup RTdP\cup RBP\rightarrow\mathbb{Z}$, 映射时间语义节点到该指令延迟判断时间和前一条指令执行完成时间的差值, 以及映射相对基准点到对应的 getti 指令读取 ctr 的时间和前一条指令执行完成时间的差值. 设计这一函数是为了提升 TTI 程序时间行为分析的紧致程度, 由于 RPU 底层体系结构采用流水线机制, 时间语义指令与获取系统时间指令的执行会和该指令的前驱指令重叠, 所以在计算该类指令相应操作开始执行的系统时间时, 需要考虑该类指令与其前驱指令由于重叠执行而产生的时间影响. 假设 $v\in ATP\cup ATdP\cup RTP\cup RTdP\cup RBP$, v 对应指令 i_0 和指令 i_0 的前驱指令 i_{-1}, 构建指令 i_0 和指令 i_{-1} 的执行图, 由于延迟判断、获取 ctr 均在执行段进行, 所以 t_{jitter} 相当于计算指令 i_0 在执行段就绪时间与前驱指令 i_{-1} 在写回段完成时间的差值, 即

$$t_{\mathrm{jitter}}(v)=\rho_{(EX,i_0)}-(\rho_{(WB,i_{-1})}+\lambda_{(WB,i_{-1})}).$$

根据上述基本块节点执行开销中式(1)的证明过程可得

$$t_{\mathrm{jitter}}(v)\leqslant\max_{r\in R}(d^{r}_{(EX,i_0)}-d^{r}_{(WB,i_{-1})}-\lambda_{(WB,i_{-1})}).\quad(2)$$

⑤t_{WCET}: $V\times V\rightarrow\mathbb{N}$, 映射 TFG+的 2 个节点到这 2 个节点及 2 个节点间所有节点对应程序段的 WCET. 假设有节点 v_1 和 v_2, v_1 到 v_2 所有节点的执行开销已知. TFG+给出了节点间控制流信息, 而程序循环次数等流事实可以由额外的分析或用户指定的方式给出, 最后, $t_{\mathrm{WCET}}(v_1, v_2)$可以通过 IPET 得出.

现对 TTI 程序中 4 种时间语义进行时间行为分析并进行时间安全性检查.

1)绝对点时间语义节点. 假设绝对点时间语义节点为 $v_{\mathrm{atdp}}\in ATdP$. v_{atdp} 预设系统时间和最近一个前驱绝对点时间语义节点预设系统时间的差值为

$$diff = t_{ATdP}(v_{\text{atdp}}) - t_{ATdP}(tpre(v_{\text{atdp}})).$$

如果 v_{atdp} 没有前驱的绝对点时间语义节点，那么 $tpre(v_{\text{atdp}}) = start$，所以

$$diff = t_{ATdP}(v_{\text{atdp}}) - t_{start}.$$

将单位转化为 cpuClk 周期为

$$diff_{\text{cpuClk}} = diff \times t_{\text{tg}} \times \frac{T_{\text{stdClk}}}{T_{\text{cpuClk}}}. \tag{3}$$

考虑 v_{atdp} 对应指令和前一条指令重叠执行，程序从 $tpre(v_{\text{atdp}})$ 对应指令执行完成到 v_{atdp} 对应指令的延迟判断阶段(执行段)消耗的 CPU 时钟周期为

$$cost = t_{\text{WCET}}(next(tpre(v_{\text{atdp}})), pre(v_{\text{atdp}})) + t_{\text{jitter}}(v_{\text{atdp}}). \tag{4}$$

如图 10 所示，绝对点时间语义节点 2 种执行时序图中 v_{atdp} 和 $tpre(v_{\text{atdp}})$ 对应指令分别为 i_n 和 i_{n-1}. 图 10(a)，TTI 程序执行到绝对点时间语义节点时，系统时间小于预设时间，需要延迟的执行时序图；图 10(b)，TTI 程序执行到绝对点时间语义节点时，系统时间等于预设时间，不需要延迟的执行时序图.

平台相关属性计算分 3 部分：

①节点延迟. 根据 TFG+中的平台相关属性定义，$tpre(v_{\text{atdp}})$ 对应指令执行完成相对于系统时刻开始的偏移时间为 $c_{\text{off}}(tpre(v_{\text{atdp}}))$. 如图 10(a)所示，$v_{\text{atdp}}$ 对应指令在执行段需要的延迟时间为

$$c_{\text{d}}(v_{\text{atdp}}) = \max(diff_{\text{cpuClk}} - cost - c_{\text{off}}(tpre(v_{\text{atdp}})), 0). \tag{5}$$

②执行开销. 计算前驱代码段最坏执行情况下 v_{atdp} 的执行开销. v_{atdp} 对应指令 i_0，指令 i_0 前一条指令 i_{-1}，构建 i_0 和 i_{-1} 的执行图. 此时，i_0 在执行段的延迟时间已知，所以 $\lambda_{(EX,i_0)}$ 也已知，和基本块节点的执行开销计算过程一致，由式(1)可知：

$$c_{\text{t}}(v_{\text{atdp}}) \leqslant \max_{r \in R}((d^r_{(WB,i_0)} - d^r_{(WB,i_{-1})}) + (\lambda_{(WB,i_0)} - \lambda_{(WB,i_{-1})})). \tag{6}$$

③相对于系统时刻开始的偏移. 点时间语义节点对应指令的操作时间为 t_{o}，RPU 中 ttiat(ttoat)输入(输出)操作的时间为访存段 1 个 cpuClk 周期的访存开销，加上写回段 1 个 cpuClk 周期的写回开销，所以 t_{o} 共 2 个 cpuClk 周期. 另外指令在执行段除了额外的延迟时间，还存在 1 个 cpuClk 周期的执行时间. 如图 10 所示，点时间语义节点完成执行相对于系统时刻的偏移为

$$c_{\text{off}}(v_{\text{atdp}}) = \max(cost + c_{\text{off}}(tpre(v_{\text{atdp}})) - diff_{\text{cpuClk}}, 0) + (t_{\text{o}} + 1). \tag{7}$$

时间安全性检查为：绝对点时间语义节点 v_{atdp}，要求输入输出开始和完成在 $t_{ATdP}(v_{\text{atdp}})$ 和 $t_{ATdP}(v_{\text{atdp}}) + 1$ 之间. 由于采用延迟机制，输入输出操作的开始时间一定大于等于 $t_{ATdP}(v_{\text{atdp}})$，现在只需要操作的完成时间小于 $t_{ATdP}(v_{\text{atdp}}) + 1$ 即可，所以对于 TFG+中的绝对点时间语义节点需要满足要求：

$$c_{\text{off}}(v_{\text{atdp}}) < t_{\text{tg}} \times \frac{T_{\text{stdClk}}}{T_{\text{cpuClk}}}.$$

2)绝对段时间语义节点. 假设绝对段时间语义

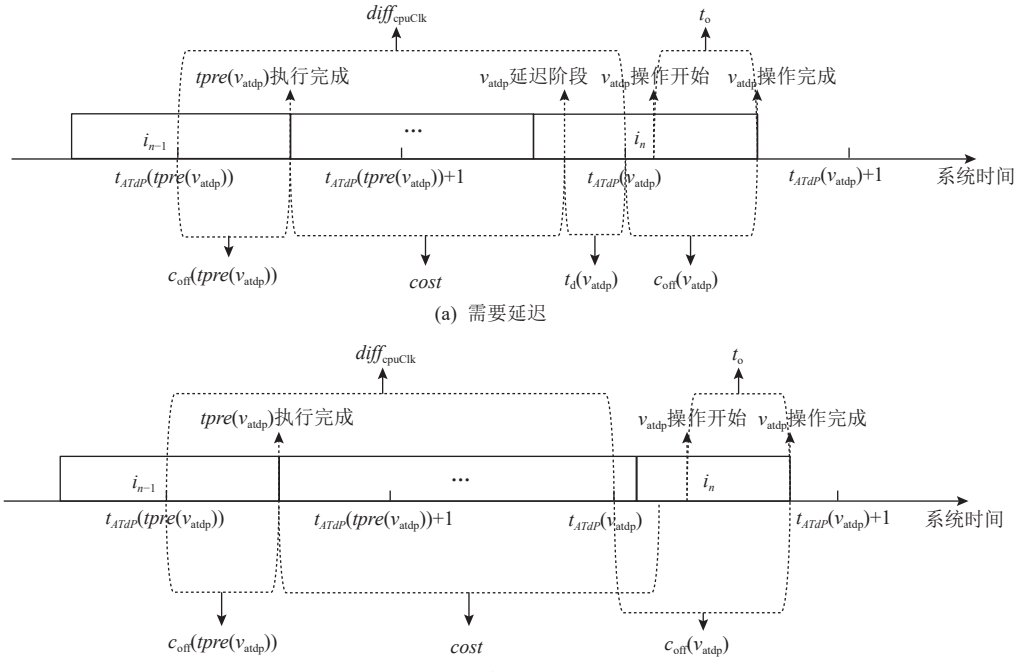


Fig. 10 Timing diagram of absolute timed semantic

图 10 绝对点时间语义时序图

节点 $v_{\text{atp}}\in ATP$. 和绝对点时间语义节点相同，首先计算 v_{atp} 预设系统时间和最近一个前驱绝对点时间语义节点预设系统时间的差值，并转化为 CPU 时钟周期，即

$$diff_{\text{cpuClk}} = diff \times t_{\text{tg}} \times \frac{T_{\text{stdClk}}}{T_{\text{cpuClk}}}.$$

如果 v_{atp} 对应指令 mtfd，那么平台相关属性均为 0，只需要验证 mtfd 语义. 而通过延迟实现的指令 delay，语义一定能得到满足，所以只需要计算平台相关属性.

下面讨论 delay 平台相关属性计算. 如图 11 所示，delay 节点 2 种执行时序图，其中 v_{atdp} 和 $tpre(v_{\text{atdp}})$ 对应指令分别为 i_n 和 i_{n-1}，图 11(a)为 delay 节点需要延迟的执行时序图，图 11(b)为不需要延迟的执行时序图. 程序从 $tpre(v_{\text{atp}})$ 执行完成到 v_{atp} 的延迟判断阶段(执行段)消耗的 cpuClk 周期为

$$cost = t_{\text{WCET}}(next(tpre(v_{\text{atp}})), pre(v_{\text{atp}})) + t_{\text{jitter}}(v_{\text{atp}}).$$

平台相关属性计算分为 3 部分：

① 节点延迟. 和绝对点时间语义节点的分析过程相同，v_{atp} 对应指令在执行段需要的延迟时间为

$$t_{\text{d}}(v_{\text{atp}}) = \max(diff_{\text{cpuClk}} - cost - c_{\text{off}}(tpre(v_{\text{atp}})), 0).$$

② 执行开销. 计算前驱代码段最坏执行情况下 v_{atp} 的执行开销. v_{atp} 对应指令 i_0，指令 i_0 前一条指令 i_{-1}，构建 i_0 和 i_{-1} 的执行图，由式(1)可知：

$$c_{\text{t}}(v_{\text{atp}}) \leqslant \max_{r\in R}((d^r_{(WB,i_0)} - d^r_{(WB,i_{-1})}) + (\lambda_{(WB,i_0)} - \lambda_{(WB,i_{-1})})).$$

下面讨论 mtfd 指令时间安全性检查. v_{atp} 对应 mtfd 指令要求指令前代码段的完成时间小于等于 $t_{ATP}(v_{\text{atp}})$，所以需要满足要求：

$$diff_{\text{cpuClk}} \geqslant t_{\text{WCET}}(next(tpre(v_{\text{atp}})), pre(v_{\text{atp}})) + c_{\text{off}}(tpre(v_{\text{atp}})).$$

3)相对点时间语义节点. 此处的相对语义，均指运行时确定的相对点时间语义. 运行时确定的相对点时间语义通过 getti 和 ttiat(ttoat)指令组合的形式表达，如图 12(a)所示，运行时确定的相对点时间语义的时序图，getti 指令获取系统时间处在 1 个系统时刻内，在系统时刻的具体位置需要根据执行情况确定. 为保证系统运行时时间语义指令的正确性，如图 12(b)所示，在时间安全性检查时，假设 getti 指令在系统时刻最后读取系统时间. 而对于相对点时间语义指令的延迟时间等平台相关属性，则根据前驱代码段最坏执行情况进行分析.

假设相对点时间语义节点为 $v_{\text{rtdp}}\in RTdP$，对应的相对基准点为 $v_{\text{rbp}}\in RBP$. v_{rbp} 对应的 getti 指令会在 EX 段读取系统时间，所以从 getti 读取系统时间(EX 段)到 v_{rtdp} 的延迟判阶段(执行段)消耗的 cpuClk 周期，还需要加上 getti 指令执行段、访存段、写回段的执行时间，对于 RPU 来说为 3 个 CPU 时钟周期，结果为

$$cost = t_{\text{WCET}}(next(v_{\text{rbp}}), pre(v_{\text{rtdp}})) + t_{\text{jitter}}(v_{\text{rtdp}}) + 3.$$

平台相关属性计算分 2 部分：① 节点延迟. 如

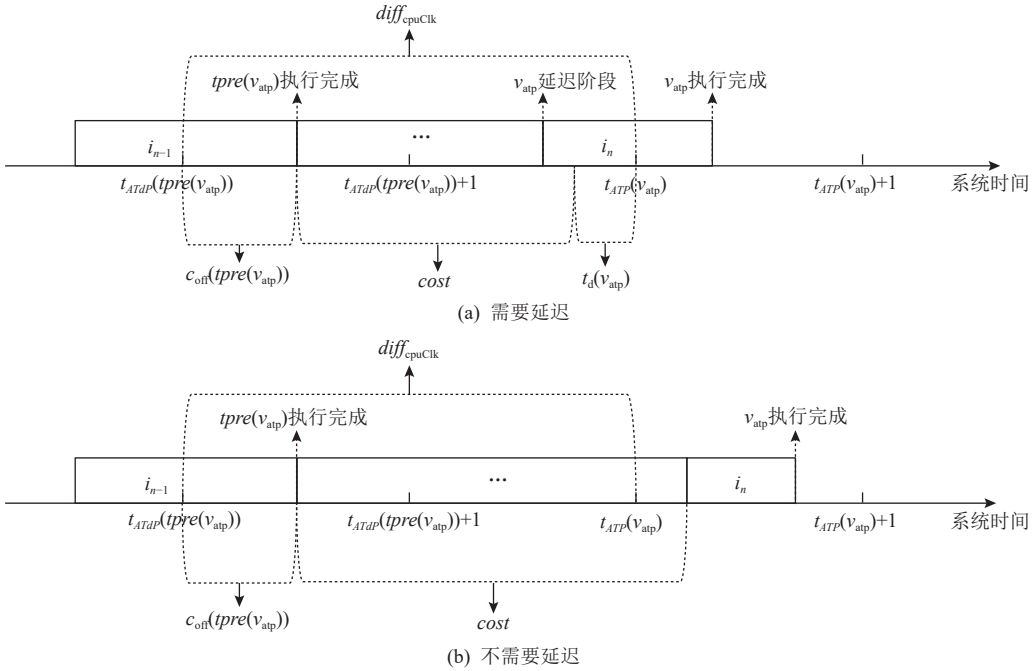


Fig. 11 Timing diagram of absolute timing semantic

图 11 绝对段时间语义时序图

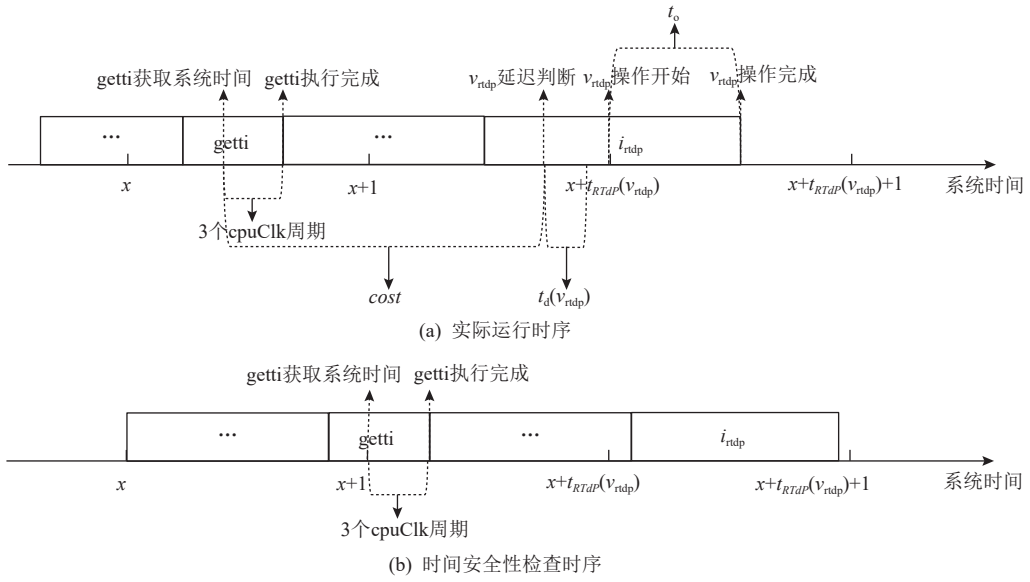


Fig. 12 Timing diagram of relative timed semantic

图 12 相对点时间语义时序图

图 13 所示, v_{rbp} 对应的 getti 指令在执行段的时间相对于系统时刻开始的偏移为 *offset*, 该偏移时间受到 v_{atp} 前一个绝对点时间语义节点或程序开始时间的影响, 图 13 中假设 getti 指令的前一个绝对点时间语义指令为 i_{n-1}. 首先, $tpre(v_{rtdp})$ 对应指令执行完成到 v_{rbp} 对应 getti 指令的执行段的时间为

$$cost' = t_{\mathrm{WCET}}(next(tpre(v_{rtdp})), pre(v_{rbp})) + t_{\mathrm{jitter}}(v_{rbp}).$$

v_{rbp} 对应的 getti 指令执行段相对于系统时刻开始的偏移时间为

$$offset = (cost' + c_{\mathrm{off}}(tpre(v_{rtdp})))- \left\lfloor (cost' + c_{\mathrm{off}}(tpre(v_{rtdp}))) \times \frac{T_{\mathrm{cpuClk}}}{T_{\mathrm{stdClk}} \times t_{tg}} \right\rfloor \times t_{tg} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}}.$$

v_{rtdp} 在执行段的延迟时间为

$$t_d(v_{rtdp}) = \max(t_{RTdP}(v_{rtdp}) \times t_{tg} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}} - cost - offset, 0).$$

②执行开销. v_{rtdp} 对应指令 i_0, 指令 i_0 前一条指令 i_{-1}, 构建 i_0 和 i_{-1} 的执行图. 由式(1)可知:

$$c_1(v_{rtdp}) \leqslant \max_{r \in R}((d^r_{(WB,i_0)} - d^r_{(WB,i_{-1})}) + (\lambda_{(WB,i_0)} - \lambda_{(WB,i_{-1})})).$$

下面讨论时间安全性检查. v_{rbp} 对应的 getti 指令获取的系统时间, 会受到前驱代码段的执行时间的影响, 只考虑前驱代码段的最坏执行情况, 不一定能保证后续相对点时间语义指令的正确性. 图 14(a)显示前驱代码最好执行情况下, v_{rbp} 和 v_{rtdp} 间程序的允许执行时间 t_b; 图 14(b)显示前驱代码最坏执行情况下, v_{rbp} 和 v_{rtdp} 间程序的允许执行时间 t_w. 图 14 中, getti 指令最好执行情况下获取到的系统时间为 y, getti 指令最坏执行情况下获取到的系统时间为 x, 存在 $y < x$, 显然 $t_w > t_b$.

如图 14(b)所示, 验证时假设 v_{rbp} 读取系统时间, 发生在系统时刻最后, v_{rtdp} 对应指令的操作时间为 t_o, 所以为保证相对点时间语义的正确性, 需满足 3 个要求:

$$cost < t_{RTdP}(v_{rtdp}) \times t_{tg} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}},$$

$$t_o + 1 < t_{tg} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}},$$

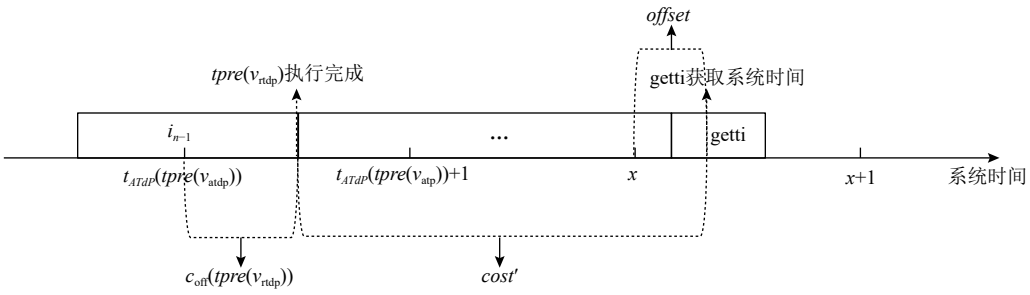


Fig. 13 Timing diagram of getti instruction execution

图 13 getti 指令执行时序图

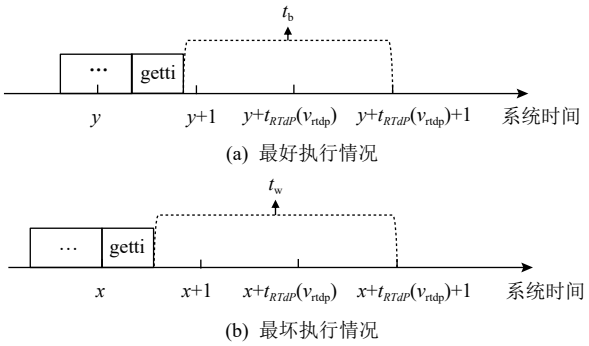


Fig. 14 Example of impact of getti instruction predecessor code fragment

图 14 getti 指令前驱代码段的影响示例

$$cost + t_{\mathrm{o}} + 1 < t_{RTdP}(v_{\mathrm{rtdp}}) \times t_{\mathrm{tg}} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}}.$$

第 1 个要求保证相关程序段的执行时间不超过限定时间；第 2 个要求保证输入输出操作的时间不超过限定时间，其中加 1 为 v_{rtdp} 对应指令在执行段除了延迟以外的 1 个 cpuClk 周期；第 3 个要求保证程序段执行以及输出输入操作的时间不超过限定时间.

4）相对段时间语义节点. 假设相对段时间语义节点为 $v_{\mathrm{rtp}} \in RTP$，对应的相对基准点为 $v_{\mathrm{rbp}} \in RBP$. 和相对点时间语义节点相同，从 getti 读取系统时间（执行段）到 v_{rtp} 的延迟判阶段（执行段）消耗的 CPU 时钟周期为

$$cost = t_{\mathrm{WCET}}((next(v_{\mathrm{rbp}})), pre(v_{\mathrm{rtp}})) + t_{\mathrm{jitter}}(v_{\mathrm{rtp}}) + 3.$$

和绝对段时间语义节点相同，如果 v_{rtp} 对应指令为 mtfd，$t_{\mathrm{jitter}}(v_{\mathrm{rtp}})=0$，且只需验证 mtfd 语义. delay 指令，只需要计算平台相关属性.

delay 指令平台相关属性计算分 2 部分：

① 节点延迟. 和相对点时间语义节点相同，$tpre(v_{\mathrm{rtp}})$ 对应指令执行完成到 v_{rbp} 对应 getti 指令的执行段的时间为

$$cost' = t_{\mathrm{WCET}}(next(tpre(v_{\mathrm{rtp}})), pre(v_{\mathrm{rbp}})) + t_{\mathrm{jitter}}(v_{\mathrm{rbp}}).$$

v_{rbp} 对应的 getti 指令执行段相对于系统时刻开始的偏移时间为

$$offset = (cost' + c_{off}(tpre(v_{\mathrm{rtp}}))) - \left\lfloor (cost' + c_{\mathrm{off}}(tpre(v_{\mathrm{rtp}}))) \times \frac{T_{\mathrm{cpuClk}}}{T_{\mathrm{stdClk}} \times t_{\mathrm{tg}}} \right\rfloor \times t_{\mathrm{tg}} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}}.$$

v_{rtp} 在执行段的延迟时间为

$$t_{\mathrm{d}}(v_{\mathrm{rtp}}) = \max(t_{RTP}(v_{\mathrm{rtp}}) \times t_{\mathrm{tg}} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}} - cost - offset, 0).$$

②执行开销. v_{rtp} 对应指令 i_0，指令 i_0 前一条指令 i_{-1}，构建指令 i_0 和 i_{-1} 执行图，由式（1）可知：

$$c_{\mathrm{t}}(v_{\mathrm{rtp}}) \leqslant \max_{r \in R}((d^{\beta}_{(WB,i_0)} - d^{\beta}_{(WB,i_{-1})}) + (\lambda_{(WB,i_0)} - \lambda_{(WB,i_{-1})})).$$

下面讨论 mtfd 指令时间安全性检查. 和相对点时间语义验证相同，考虑 getti 指令执行的最坏情况. 假设 v_{rbp} 对应 getti 指令执行段读取系统时间发生在系统时刻最后，所以为保证相对段时间语义的正确性，需满足要求：

$$cost < (t_{RTP}(v_{\mathrm{rtp}}) - 1) \times t_{\mathrm{tg}} \times \frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}}.$$

6 TTI 程序示例

本节给出 TTI 程序示例，并对 TTI 程序进行时间行为分析和时间安全性检查，最后通过在 RPU 上实际部署运行，验证时间行为分析和时间安全性检查的正确性. 本节主要有 2 个应用示例，分别为自动驾驶示例和防抱死系统示例.

6.1 自动驾驶示例

本文采用文献 [4] 中给出的自动驾驶示例. 假设一个自动驾驶系统中有 3 个任务：1）采集任务，传感器采集数据，进行数据预处理，周期为 2 ms；2）分析任务，对预处理之后的数据进行分析和处理，得到响应数据，周期为 1 ms；3）执行任务，将响应数据分发到执行器，周期为 2 ms.

使用交错 LET 语义对任务集进行实现，图 15 显示了该任务集的任务分解和建模时序，虚线箭头体现了任务的数据依赖关系. 其中，每个任务被分解为 3 个子任务，分别为 I, C, O，其中 O 和 I 这 2 个子任务被合并为一个 OI 子任务组，保证了该任务集的交错 LET 语义.

任务 1 对应采集任务，t1oi 对应采集任务的 OI 子任务组，t1c 对应采集任务的 C 任务. 任务 2 对应分析任务，t2oi1 对应分析任务第 1 个周期的 OI 子任务组，t2c1 对应分析任务第 1 个周期的 C 任务，同理，t2oi2, t2c2 分别对应分析任务第 2 个周期的 OI 子任务和 C 任务. 任务 3 对应执行任务，t3oi 对应执行任务的 OI 子任务组，t3c 对应执行任务的 C 任务. 最后自动驾驶示例的实现代码如图 16 所示，RPU 提供了

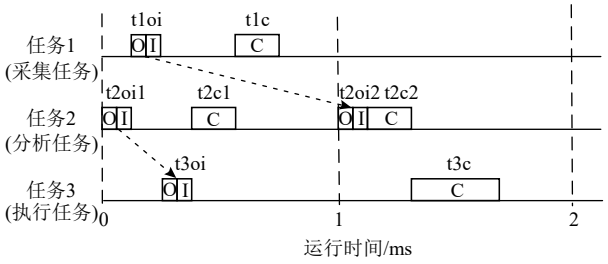


Fig. 15 Interleaved LET implementation

图 15 交错 LET 语义实现

硬件多线程机制, 所以将任务分类放置在不同的硬件线程上, 其中 0 号硬件线程(thread 0)用于初始化系统, 1 号硬件线程(thread 1)用于执行 OI 子任务, 2 号硬件线程(thread 2)用于执行 C 任务, 3 号硬件线程(thread 3)用于执行软实时任务. 可以使用 TTI 指令集中的并发管理指令, 控制硬件线程的调度时间, 从而确定硬件线程上每个任务的调度时间. 如图 16 所示, 为自动驾驶系统在 RPU 各硬件线程上的程序.

现在给出运行自动驾驶示例的 RPU 参数, RPU 的 cpuClk 频率为 25 MHz, T_{cpuClk} 为 0.04 μs, stdClk 频率为 1MHz, T_{stdClk} 为 1 μs, stdClk 和 cpuClk 同步, 且时间粒度为 25, 即 1 个系统时刻为 25 μs.

如表 5 所示, 给出 LET 实例中每个任务程序的属性. 任务 1、任务 2、任务 3 对应的 C 任务是普通程序, OI 子任务是 TTI 程序, 每个 OI 子任务中都需要分别表达输入和输出绝对点时间语义, 且对应不同的系统时间. 另外给出每个任务被调度执行的系统时间.

现在给出自动驾驶示例中 C 任务 WCET 分析结果, t1c 的 WCET 为 0.5 ms, t2c1 和 t2c2 的 WCET 为 0.25 ms, t3c 的 WCET 为 0.5 ms. 自动驾驶示例中每个任务的 OI 子任务的程序结构类似, 所以使用 t3oi 作为例子进行说明. 如图 17 所示, t3oi 任务对应 TTI 程序和 TFG+, 图 17 中省略部分分别表示 t3oi 在指令 ttoat 执行前的准备工作和指令 ttiat 执行后的处理工作, 此部分由普通指令构成, 所以在 TFG+中对此部分的普通基本块节点进行省略. 下面给出 TFG+中程序属性和平台相关属性.

1)程序属性. 除时间粒度, TFG+此类属性的单位均为系统时刻. TFG+中时间粒度 $t_{\text{tg}}=25$, 任务开始系统时间 $t_{start}=6$, ttoat 指令对应绝对点时间语义节点 $v_1\in ATdP$, ttiat 指令对应绝对段时间语义节点 $v_2\in ATdP$, ttoat 指令预设系统时间为 $t_{ATdP}(v_1)=7$, ttiat 指令的预设系统时间为 $t_{ATdP}(v_2)=7$.

2)平台相关属性. TFG+此类属性的单位均为 cpuClk 周期. t3oi 在指令 ttoat 前的准备工作的最坏情况下的执行开销为 200 个 cpuClk 周期, 以及指令 ttiat 后的处理工作的最坏情况下的执行开销为 300 个 cpuClk 周期. 首先, 计算节点 v_1 的平台相关属性, 并验证节点 v_1 对应 ttoat 指令的语义. 根据式(3), 计算节点 v_1 预设时间和任务开始时间的差值并转化为 cpuClk 周期:

$$diff_{\text{cpuClk}}=(t_{ATdP}(v_1)-t_{start})\times t_{\text{tg}}\times\frac{T_{\text{stdClk}}}{T_{\text{cpuClk}}}=$$
$$(7-6)\times 25\times 25=625.$$

RPU 在系统时间为 6 时执行 t3oi 任务, 任务开始相对于系统时刻开始的偏移为 0, 所以 $c_{\text{off}}(start)=0$. 计算从任务开始执行到 v_1 的延迟判断阶段消耗的 cpuClk 周期, 由式(4)可得

$$cost=t_{\text{WCET}}(next(start),pre(v_1))+t_{\text{jitter}}(v_1).$$

由于 t3oi 在执行 ttoat 前的准备工作的 WCET 为 200 个 cpuClk 周期, 即

$$t_{\text{WCET}}(next(start),pre(v_1))=200.$$

对于 v_1 延迟判断时间和前一条指令执行完成时间的差值 $t_{\text{jitter}}(v_1)$, 需要构建执行图, 如图 18 所示, 其中 τ 表示 ttoat 指令在执行段的延迟时间. 根据式(2), 执行图节点信息可得

$$t_{\text{j}}(v_1)=\max_{r\in R}(d^r_{(EX,i_0)}-d^r_{(WB,i_{-1})}-\lambda_{(WB,i_{-1})})=$$
$$3-4-1=-2,$$

```
① thread0:  # add entry for thread manager
② init:
③   addi a1, x0, 25
④   settg a1  # set time granularity to 25 us(0.025 ms)
⑤   setti x0  # reset time
⑥ add_tt_table_entry:
⑦   addi a0, x0, 2  # base time
⑧   addi a1, x0, 1  # thread id for thread1
⑨   addi a2, x0, 2   # thread id for thread2
⑩   addtk a0, a1  # task entry for first OI subtask of task2
⑪   addi a0, a0, 2  # base time + 2 × 0.025 ms
⑫   addtk a0, a1  #   task entry for OI subtask of task1
⑬   addi a0, a0, 2  # base time + 4 × 0.025 ms
⑭   addtk a0, a1  #   task entry for OI subtask of task3
⑮   addi a0, a0, 2  # base time + 6 × 0.025 ms
⑯   addtk a0, a2  # task entry for first C subtask of task2
⑰   addi a0, a0, 10  #   base time + 16 × 0.025 ms
⑱   addtk a0, a2   # task entry for C subtask of task1
⑲   addi a0, a0, 24  #   base time + 40 × 0.025 ms
⑳   addtk a0, a1  # task entry for second OI subtask of task2
㉑   addi a0, a0, 2  #   base time + 42 × 0.025 ms
㉒   addtk a0, a2  # task entry for second C subtask of task2
㉓   addi a0, a0, 10  #   base time + 52 × 0.025 ms
㉔   addtk a0, a2   # task entry for C subtask of task3
㉕   addi a0, a0, 26   # base time + 78 × 0.025 ms
㉖   addtk a0, x0   # task entry for thread0
㉗   tkend
㉘   addi a0, a0, 2  # set next base time
㉙   j add_tt_table_entry
```

(a) 0号线程代码

```
① thread1:
② addi a0, x0, 3  # init base time 3 × 0.025 ms
③ t2oi1:  # first OI subtask of task2
④ …
⑤ ttoat a4, (a3), a0  # ex time, 3 × 0.025 ms
⑥ ttiat a2, (a1), a0
⑦ …
⑧ tkend
⑨ t1oi:  # OI subtask of task1
⑩ …
⑪ addi a0, a0, 2  #  ex time, 5 × 0.025 ms
⑫ ttoat a4, (a3), a0
⑬ ttiat a2, (a1), a0
⑭ …
⑮ tkend
⑯ t3oi:  # OI subtask of task3
⑰ …
⑱ addi a0, a0, 2  # ex time, 7 × 0.025 ms
⑲ ttoat a4, (a3), a0
⑳ ttiat a2, (a1), a0
㉑ …
㉒ tkend
㉓ t2oi2:  #  second OI subtask of task2
㉔ …
㉕ addi a0, a0, 36  # ex time, 43 × 0.025 ms
㉖ ttoat a4, (a3), a0
㉗ ttiat a2, (a1), a0
㉘ …
㉙ tkend
㉚ addi a0, a0, 20  # set next base time
㉛ j t2oi1
```

(b) 1号线程代码

```
① thread2:
② t2c1:  # first c subtask of task2
③ … # run at base time + 6 × 0.025 ms
④ tkend
⑤ t1c:  # c subtask of task1
⑥ … # run at base time + 16 × 0.025 ms
⑦ tkend
⑧ t2c2:  # 2nd c subtask of task2
⑨ … # run at  base time + 42 × 0.025 ms
⑩
⑪ t3c:  # c subtask of task3
⑫ … # run at base time + 52 × 0.025 ms
⑬ tkend
⑭ j t2c1
⑮
⑯ thread3:
⑰ run some soft-real applications
```

(c) 2号和3号线程代码

Fig. 16 Example of autonomous driving program

图 16 自动驾驶程序示例

Table 5 Example of Tasks Properties

表 5 任务属性示例

任务	程序类型	语义类型和时间语义指令对应系统时间	任务开始系统时间
t1oi	TTI 程序	绝对点时间语义（输出）：5 绝对点时间语义（输入）：5	4
t2oi1	TTI 程序	绝对点时间语义（输出）：3 绝对点时间语义（输入）：3	2
t2oi2	TTI 程序	绝对点时间语义（输出）：43 绝对点时间语义（输入）：43	42
t3oi	TTI 程序	绝对点时间语义（输出）：7 绝对点时间语义（输入）：7	6
t1c	普通程序		18
t2c1	普通程序		8
t2c2	普通程序		44
t3c	普通程序		54

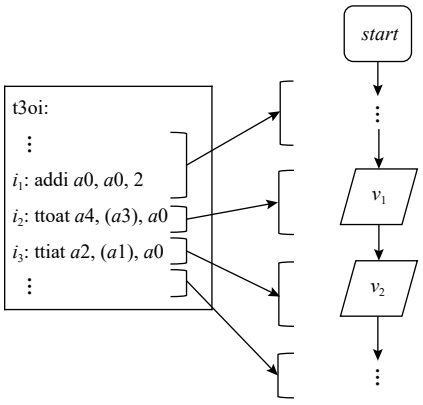


Fig. 17 Task t3oi program and TFG+

图 17 任务 t3oi 程序和 TFG+

$$cost = t_{\mathrm{WCET}}(next(start), pre(v_1)) + t_{\mathrm{jitter}}(v_1) = 200 - 2 = 198.$$

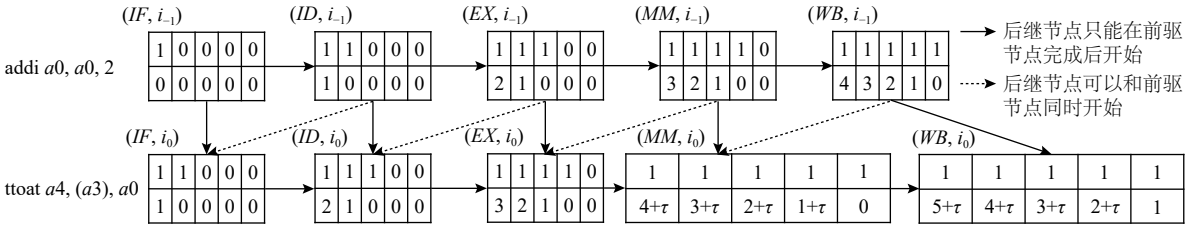
由式(5)，节点 v_1 对应指令的延迟时间为 $\tau = t_{\mathrm{d}}(v_1)$，且

$$t_{\mathrm{d}}(v_1) = \max(diff_{\mathrm{cpuClk}} - cost - c_{\mathrm{off}}(start), 0) = 625 - 198 - 0 = 427.$$

如图 17 所示，根据节点 v_1 的延迟时间和执行图

节点属性	*IF*	*ID*	*EX*	*MM*	*WB*
$e^{r}_{(v,i_j)}$	$e^{IF}_{(v,i_j)}$	$e^{ID}_{(v,i_j)}$	$e^{EX}_{(v,i_j)}$	$e^{MM}_{(v,i_j)}$	$e^{WB}_{(v,i_j)}$
$d^{r}_{(v,i_j)}$	$d^{IF}_{(v,i_j)}$	$d^{ID}_{(v,i_j)}$	$d^{EX}_{(v,i_j)}$	$d^{MM}_{(v,i_j)}$	$d^{WB}_{(v,i_j)}$

(a) 执行图节点属性



(b) 执行图

Fig. 18 Example of task t3oi execution graph

图 18 任务 t3oi 执行图示例

中节点信息, 由式(6), 节点 v_1 执行开销为

$$c_{\mathrm{t}}(v_1)=\max_{r\in R}((d^{r}_{(WB,i_0)}-d^{r}_{(WB,i_{-1})})+(\lambda_{(WB,i_0)}-\lambda_{(WB,i_{-1})}))=$$
$$(432-4)+(1-1)=428.$$

由式(7), 相对于系统时刻的偏移为

$$c_{\mathrm{off}}(v_1)=\max(cost+c_{\mathrm{off}}(start)-diff_{\mathrm{cpuClk}},0)+(t_{\mathrm{o}}+1).$$

对于 RPU 来说, 输入操作的时间 $t_{\mathrm{o}}=2$, 所以

$$c_{\mathrm{off}}(v_1)=0+(2+1)=3.$$

最后, 进行时间安全性检查, 即

$$c_{\mathrm{off}}(v_1)<t_{\mathrm{tg}}\times\frac{T_{\mathrm{stdClk}}}{T_{\mathrm{cpuClk}}},$$

节点 v_1 对应 ttoat 指令的语义可以得到保证.

对于 v_2 的平台相关属性的计算和 v_1 类似, 给出 v_2 平台相关属性的计算结果. 节点 v_2 和节点 v_1 的预设时间相同, 所以节点 v_2 的延迟时间为 $t_{\mathrm{d}}(v_2)=0$, 执行开销为 $c_{\mathrm{t}}(v_2)=1$, 相对于系统时刻开始的偏移为 $c_{\mathrm{off}}(v_2)=4$, 所以 v_2 的语义也通过验证.

至此任务 t3oi 时间行为分析和语义验证完成, 其他时间语义程序的分析和验证过程类似. 图 19 为自动驾驶示例在 RPU 上的运行时序图. 在图 19 中, 给出每个绝对点时间语义指令生效的实时间, 将表 5 中每个时间语义指令预设时间转化为实时间后, 和图 15 进行比较可知, 语义均正确表达, 同时, LET 语义也得到保证.

6.2 防抱死系统示例

防抱死系统(anti-lock braking system, ABS)是为了防止车轮在紧急刹车过程中车轮抱死而导致车辆失控的控制系统, ABS 最早于 1950 年在文献 [32] 中提出, 现在广泛应用在车辆的刹车控制系统中. 本文采用 LET 编程模型, 实现 ABS 中的控制器部分, 并且将 LET 模型映射至 TTI 程序, 通过时间行为分析和时间安全性检查验证时间语义指令的正确性, 最后部署到 RPU 上, 验证 TTI 程序的功能和时序的正确性.

如图 20 所示, ABS 的控制模型有 5 个部分: 车辆、车轮、转差、摩擦以及控制器. 车辆部分的输入为摩擦系数 *mu*, 输出为车辆速度 *vv*, 以及地面对车轮的摩

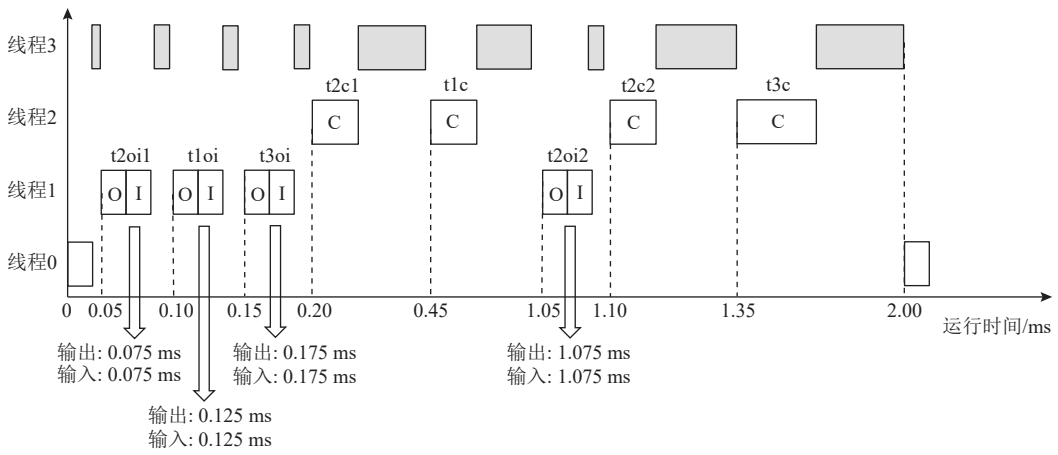


Fig. 19 Runtime time diagram of RPU threads

图 19 RPU 线程运行时序图

擦力 *Ff*; 车轮部分的输入为来自地面的摩擦力 *Ff* 以及控制器部分施加给车轮的刹车扭矩 *Tb*, 输出为车轮的角速度 *ww*; 转差部分的输入为车辆的速度 *vv*, 以及车轮的角速度 *ww*, 输出为轮胎和地面的转差 *s*; 摩擦部分的输入为转差 *s*, 输出当前车轮相对地面的摩擦系数 *mu*; 控制器部分的输入为转差 *s*, 以及车辆的速度 *vv*, 输出为应当施加给车轮的刹车扭矩 *Tb*. 各部分连接起来构成一个自反馈系统.

ABS 模型中除控制器外的其余部分在模拟物理环境以及作为传感器输入, 控制器部分通过特定算法控制整个 ABS, 而且控制器部分的算法具有硬实时性、周期性、I/O 特性等, 适合使用 LET 编程模型. 图 21 所示为控制器部分的具体构成, 控制器部分的输入为转差 *s* 和车辆速度 *vv*, 输出为刹车扭矩 *Tb*, 其中控制器部分可以根据转差的大小调整输出的刹车扭矩, 车辆速度用来判断是否需要防抱死控制, 当车辆速度小于规定的最小速度 v_{min}, 不需要进行防抱死控制. 由于车辆转差在 0.2 时, 车轮和地面产生的摩擦力最大, 刹车效果最好, 所以控制器部分的目的在于通过调整刹车扭矩的大小, 将车轮和地面的转差维持在 0.2 左右, 控制器部分会比较当前转差和 0.2 的差值, 并将结果通过一个 bang-bang 控制器输出为正负信号, 液压系统(hydraulic system)通过接收该信号来调整刹车扭矩, 最后控制系统将刹车扭矩输出, 以实现对车轮的控制.

如图 22 所示, ABS 模型中控制器部分对应的 TTI 程序, 其中控制器部分算法采用交错语义的 LET 编程模型实现, 和 6.1 节自动驾驶示例不同, ABS 控制器部分代码将 LET 中的 OI 子任务和 C 任务合并在一个 TTI 程序中.

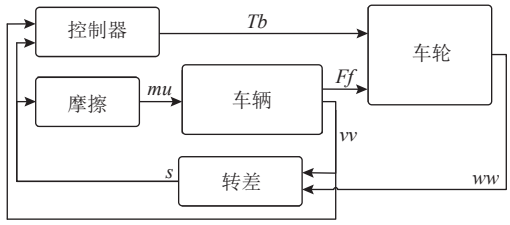


Fig. 20 ABS control model

图 20 ABS 控制模型

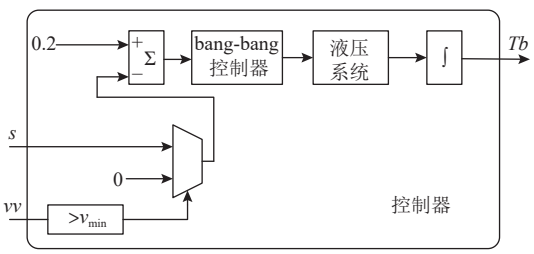


Fig. 21 ABS controller part

图 21 ABS 控制器部分

图 22 中寄存器 *a*7 中的值表示控制算法的周期, 可以根据具体的需求确定, 控制算法会在周期开始, 使用 ttoat 指令定时输出上一轮计算得到的刹车扭矩, 使用 ttiat 指令定时输入本轮计算刹车扭矩需要的转差和车辆速度, 然后经过图 22 中所示计算过程, 确定本轮需要输出的刹车扭矩.

运行图 22 所示控制器部分 TTI 程序的 RPU 参数和 6.1 节中自动驾驶示例相同. 控制算法的执行周期为 2 ms. 如图 23 所示, 控制器部分 TTI 程序对应的部分 TFG+. 该 TFG+是将 TTI 程序中循环展开构建, 对应 TTI 程序中第 1 轮循环和第 2 轮循环, 其中第 1 轮循环为图 22 中行①~⑬内容, 该轮循环中绝对点时间语义指令需要在 0 ms 完成相应操作; 第 2 轮循环为图 22 中行④~⑬内容, 且第 2 轮循环中的行⑤⑥的时间语义指令分别对应 TFG+中的绝对点时间语义节点 v_1, v_2, v_3, 这 3 个节点对应指令需要在 2 ms 完成相应操作. 显然如果图 23 中 TFG+对应的 TTI 程序可以通过时间安全性检查, 则图 22 中 TTI 程序后续循环的时序正确性也可以得到保证.

经过时间行为分析和时间安全性检查图 23 所示控制器部分 TTI 程序中时间语义指令均能正确执行, 图 24 显示控制器部分 TTI 程序第 1 轮和第 2 轮循环在 RPU 上运行的时序图. 图 24 中 O 表示定时输出指令, I1 和 I2 分别表示 2 条定时输入指令, 第 1 轮循环的时间为 0~2 ms, 第 2 轮循环的时间为 2~4 ms. 根据图 24 可知, 定时输入、定时输出操作均在规定的系统时刻内完成, 符合 ABS 中控制器部分对于时序的要求.

```
① controller:
②     …
③     addi a0, x0, 0
④ begin:
⑤     ttoat a6, (a5), a0  # output   Tb
⑥     ttiat a4, (a3), a0  # input   vv
⑦     ttiat a2, (a1), a0  # input   s
⑧     … # if ( a4 < vmin) then jump to next_round
⑨     … # if ( a2 > 0.2) then signal = 1 else signal = −1
⑩     … #   Tb = HydraulicFunction (signal, Tb)
⑪     addi a0, a0, a7
⑫ next_round:
⑬     j begin
```

Fig. 22 TTI program of controller part

图 22 控制器部分 TTI 程序

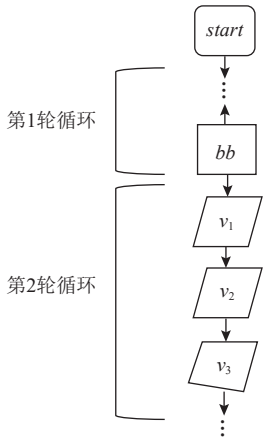


Fig. 23 TFG+ of controller algorithm

图 23 控制器算法的 TFG+

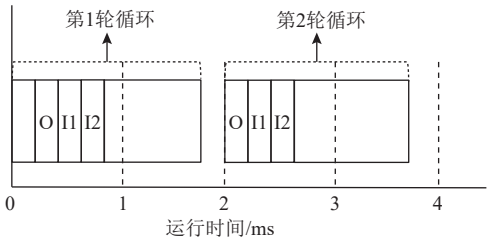


Fig. 24 Runtime time diagram of controller part

图 24 控制器部分运行时序图

7 总 结

通用计算机在指令集架构中没有提供时间语义指令, 导致上层控制模型中的时间行为无法精确自然地映射至程序代码, 控制精度大大降低, 系统的实时性受到影响.

本文在已经提出的 TTI 指令集基础上, 总结了 TTI 指令集可以表达的 4 种时间语义, 以此来描述复杂系统的时间行为. 并且提出了 TTI 程序的表示 TFG+, 同时, 基于 RPU 给出了分析 TTI 程序时间行为和时间安全性检查的方法.

未来, 我们将进一步完善 TTI 指令集相关工作, 对 TTI 指令集语义进行形式化说明, 以及扩展 TFG+ 对并发语义的支持.

作者贡献声明: 高银康提出思想、撰写论文; 陈香兰、龚小航、蒋滨泽、李曦、周学海提供论文撰写指导和修改意见.

参 考 文 献

[1] Henzinger T, Horowitz B, Kirsch C. Giotto: A time-triggered language for embedded programming[C]// Proc of the 1st EMSOF. New York: ACM, 2001: 166−184

[2] Bui D, Lee E, Liu I, et al. Temporal isolation on multiprocessing architectures[C]// Proc of the 48th Design Automation Conf. New York: ACM, 2011: 274−279

[3] Lee E, Reineke J, Zimmer M. Abstract PRET machines[C]// Proc of the 38th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 2017: 1−11

[4] Wang Chao, Chen Xianglan, Zhang Bo, et al. A real-ime processor model with timing semantics[J]. Journal of Computer Research and Development, 2021, 58(6): 1176−1191 (in Chinese)
(汪超, 陈香兰, 章博, 等. 一种具有时间语义的实时处理器模型 [J]. 计算机研究与发展, 2021, 58(6): 1176−1191)

[5] Natarajan S, Broman D. Timed C: An extension to the C programming language for real-time systems[C]// Proc of the 24th IEEE Real-Time and Embedded Technology and Applications Symp. Piscataway, NJ: IEEE, 2018: 227−239

[6] Natarajan S, Broman D. Temporal property-based testing of a timed C compiler using time-flow graph semantics[C/OL]// Proc of the 23rd Forum for Specification and Design Languages. Piscataway, NJ: IEEE, 2020 [2023-02-24].https://doi.org/10.1109/FDL50818.2020.9232935

[7] Rochange C, Sainrat P. A context-parameterized model for static analysis of execution times[M]// Transactions on High-Performance Embedded Architectures and Compilers II. Berlin: Springer, 2009: 222−241

[8] Lee E. What is real time computing? A personal view[J]. IEEE Design and Test, 2018, 35(2): 64−72

[9] Lickly B, Liu I, Kim S, et al. Predictable programming on a precision timed architecture[C]// Proc of the 11th Int Conf on Compilers, Architectures and Synthesis for Embedded Systems. New York: ACM, 2008: 137−146

[10] Zimmer M, Broman D, Shaver C, et al. FlexPRET: A processor platform for mixed-criticality systems[C]// Proc of the 19th 2014 IEEE Real-Time and Embedded Technology and Applications Symp. Piscataway, NJ: IEEE, 2014: 101−110

[11] Turing A M. On computable numbers, with an application to the entscheidung sproblem[J]. Proceedings of the London Mathematical Society, 1937, 2(1): 230−265

[12] Von Neumann J. First draft of a report on the EDVAC[J]. IEEE Annals of the History of Computing, 1993, 15(4): 27−75

[13] Knial P, Mokrushin L, Thiagarajan P S, et al. Timed vs time-triggered automata[C]// Proc of the 15th Int Conf on Concurrency Theory. Berlin: Springer, 2004: 340−354

[14] Waterman A S. Design of the RISC-V instruction set architecture[D]. Berkeley, CA: University of California, Berkeley, 2016

[15] Hennessy J L, Patterson D A. Computer Architecture: A quantitative Approach[M]. San Francisco: Margan Kaufmann, 2011

[16] Boldt M, Traulsen C, von Hanxleden R. Worst case reaction time analysis of concurrent reactive programs[J]. Electronic Notes in Theoretical Computer Science, 2008, 203(4): 65−79

[17] Berry G, Gonthier G. The Esterel synchronous programming language: Design, semantics, implementation[J]. Science of

Computer Programming, 1992, 19(2): 87−152

[18] Andalam S, Roop P, Girault A, et al. PRET-C: A new language for programming precision timed architectures[D]. Grenoble, France: Institut National de Recherche en Informatique et en Automatique, 2009

[19] Roop P S, Andalam S, von Hanxleden R, et al. Tight WCRT analysis of synchronous C programs[C]// Proc of the 12th Int Conf on Compilers, Architecture, and Aynthesis For Embedded Systems. New York: ACM, 2009: 205−214

[20] Cooper K D, Harvey T J, Waterman T. Building a control-flow graph from scheduled assembly code[R]. Texas: Rice University, 2002

[21] Kirner R, Knoop J, Prantl A, et al. WCET analysis: The annotation language challenge[C]// Proc of the 7th Int Workshop on Worst-Case Execution Time Analysis. Schloss Dagstuhl: Leibniz-Zentrum für Informatik, 2007: 83−99

[22] Asavoae M, Maiza C, Raymond P. Program semantics in model-based WCET analysis: A state of the art perspective[C]// Proc of the 13th Int Workshop on Worst-Case Execution Time Analysis. Schloss Dagstuhl: Leibniz-Zentrum fuer Informatik, 2013: 32−41

[23] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints[C]// Proc of the 4th ACM SIGACT-SIGPLAN Symp on Principles of Programming Languages. New York: ACM, 1977: 238−252

[24] Fields B, Rubin S, Bodik R. Focusing processor policies via critical-path prediction[C]// Proc of the 28th Annual International Symp on Computer Architecture. Piscataway, NJ: IEEE, 2001: 74−85

[25] Li Xianfeng, Roychoudhury A, Mitra T. Modeling out-of-order processors for WCET analysis[J]. Real-Time Systems, 2006, 34(3): 195−227

[26] Ferdinand C, Heckmann R. aiT: Worst-case Execution Time Prediction by Static Program Analysis[M]// Building the Information Society. Berlin: Springer, 2004: 377−383

[27] Li Xianfeng, Liang Yun, Mitra T, et al. Chronos: A timing analyzer for embedded software[J]. Science of Computer Programming, 2007, 69(1/2/3): 56−67

[28] Allabriga C, Cassé H, Rochange C, et al. OTAWA: An open toolbox for adaptive WCET analysis[C]//Proc of the 8th IFIP Int Workshop on Software Technolgies for Embedded and Ubiquitous Systems. Berlin: Springer, 2010: 35−46

[29] Puschner P, Koza C. Calculating the maximum execution time of real-time programs[J]. Real-Time Systems, 1989, 1(2): 159−176

[30] Shaw A C. Reasoning about time in higher-level language software[J]. IEEE Transactions on Software engineering, 1989, 15(7): 875−889

[31] Li Y T S, Malik S. Performance analysis of embedded software using implicit path enumeration[C]// Proc of the 1st ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools For Real-Rime Rystems. New York: ACM , 1995: 88−98

[32] Lawes J. Car Brakes: A Guide to Upgrading, Repair and Maintenance[M]. Marlborough, UK: The Crowood Press, 2014



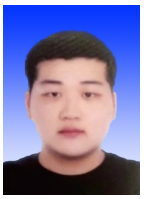
Gao Yinkang, born in 1998. Master candidate. His main research interest includes computer architecture.

高银康, 1998 年生. 硕士研究生. 主要研究方向为计算机系统结构.



Chen Xianglan, born in 1977. PhD. Her main research interest includes system software.

陈香兰, 1977 年生. 博士. 主要研究方向为系统软件.



Gong Xiaohang, born in 1999. Master candidate. His main research interest includes computer architecture.

龚小航, 1999 年生. 硕士研究生. 主要研究方向为计算机系统结构.



Jiang Binze, born in 2000. Master candidate. His main research interest includes computer architecture.

蒋滨泽, 2000 年生. 硕士研究生. 主要研究方向为计算机系统结构.



Li Xi, born in 1963. Professor. His main research interest includes computer architecture.

李　曦, 1963 年生. 教授. 主要研究方向为计算机系统结构.



Zhou Xuehai, born in 1966. Professor. His main research interest includes computer architecture.

周学海, 1966 年生. 教授. 主要研究方向为计算机系统结构.