DOI:10.7544/issn1000-1239.2021.20210157
58(6): 1176-1191, 2021

一种具有时间语义的实时处理器模型

汪 超 陈香兰 章 博 李 曦 王 超 周学海
(中国科学技术大学计算机科学与技术学院 合肥 230027)
(wcwxy@mail.ustc.edu.cn)

A Real-Time Processor Model with Timing Semantics

Wang Chao, Chen Xianglan, Zhang Bo, Li Xi, Wang Chao, and Zhou Xuehai
(*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027)

Abstract Real-time embedded system (RTES) is the core of calculation and control of safety-critical equipment. The software and hardware of RTES are required to have timing determinism and timing predictability to ensure the correctness of its time behavior. However, nearly every abstraction of modern computer systems has failed to provide timing semantics, which means it cannot meet the security design requirements of hard real-time systems. In this paper, we focus on the lack of timing semantics in the infrastructure of the instruction set architecture and try to redefine the instruction set and microarchitecture of RTES. First, we propose real-time machine (RTM), a real-time computer architecture model with timing semantics. Then, referring to the theory of time-triggered automata, we construct TTI, which is a timed instruction set, as the software/hardware interface of RTM. We also discuss the completeness of the timing semantics of TTI. Finally, we design and implement the real-time processing unit (RPU) and the timing determinism of RPU is obtained by comparing theoretical analysis with experimental results. The LET programming model is a real-time programming paradigm widely recognized by academia. In this article, we illustrate the effectiveness of RTM and TTI by giving an example of running LET tasks on RPU.

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

摘 要 实时嵌入式系统是安全关键设备的计算与控制核心.为了保证系统的时间行为正确,要求其软硬件具有时序确定性和可预测性.而现代计算机系统的各个抽象层次均缺乏时间语义,无法满足硬实时安全性设计要求.针对指令集体系结构层次的基础设施缺乏时间语义的问题,尝试重新定义实时嵌入式系统的指令集和微体系结构.首先,提出一种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).接着,参考时间触发自动机理论,构建具有时间语义的指令集——TTI(time-triggered instruction set)作为RTM的软硬件接口,并讨论TTI的时间语义完备性问题.最后,设计并实现了实时处理单元(real-time processing unit, RPU),通过理论分析与实验结果的对照得出RPU的时序确定性.逻辑执行时间(logical execution time, LET)编程模型是学术界广泛认可的实时编程范式,通过给出在RPU上运行LET任务集的示例,说明RTM和TTI的有效性.

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

中图法分类号 TP332

收稿日期:2021-02-26;**修回日期**:2021-04-20

基金项目:国家自然科学基金项目(61772482)
This work was supported by the National Natural Science Foundation of China (61772482).

通信作者:李曦(llxx@ustc.edu.cn)

实时嵌入式系统(real-time embedded system, RTES)是航空航天、汽车和医用仪器等安全关键设备的计算与控制核心,需要实时地与被控物理环境进行交互,其功能的正确性不仅取决于计算结果的逻辑正确性,还取决于与外界交互的时间正确性.

为了保证 RTES 系统的时间行为正确,要求其软硬件系统具有时序确定性和可预测性[1].然而,现代计算机系统的体系结构、运行时环境、编程语言等各个抽象层次均缺乏精确的时间语义和表达时序属性的显式结构,使得系统设计者只能采用编程抽象之外的手段(如定时中断)进行时序控制,导致控制和验证程序的时序属性或处理时序冲突十分困难,并且任何软硬件改动都将对系统时序造成不可预测的影响,甚至不得不重新进行系统设计.尤为重要的是,由于上述机制内在的异步不确定性,难以建立系统的确定性抽象模型,并据此进行严谨的形式化时间行为分析,无法满足硬实时安全性设计要求.

本文重点讨论指令集体系结构(instruction set architecture, ISA)层次缺乏时间语义的问题.现代处理器的数字电路能够以纳秒级细粒度执行,但常规 ISA 屏蔽了这一宝贵资源,未提供具有时间语义的指令集,程序员只能依赖操作系统的时间服务进行粗粒度时序控制,其可预测时序粒度比硬件高几个数量级.常规 ISA 所定义的处理器行为的正确性与其时序无关,时序仅仅是计算性能指标,而不是正确性判据.

ISA 层的基础设施缺乏时间语义已经成为安全关键实时系统设计方法学的一个基础性问题,需要重新考虑实时处理器的 ISA 和微体系结构组织.

在先前的工作[2]中,我们尝试提出一种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),设计了一版具有时间语义的指令集 TTI(time-triggered instruction set),分析了其硬件实现的可行性.本文进一步完善 RTM 的定义,抽象出独立的标准时钟部件,明确中央处理器(central processing unit, CPU)时钟和标准时钟协同控制的模式.接着,本文以 RTM 为基础,结合时间触发自动机(time-triggered automata, TTA)理论,重新定义 TTI,为其增加执行时间约束指令和任务并发管理指令,并探讨了 TTI 指令集的时间语义完备性问题.最后,本文通过一款实时处理器——实时处理单元(real-time processing unit, RPU)的设计实现和应用示例,验证了 TTI 的时间语义表达能力.

加州大学伯克利分校(University of California, Berkeley, UCB)的 PRET 机[3]研究动机与我们相近.PRET 小组提出了限定代码段执行时间的 Deadline 指令[4]和 MTFD 指令[5],并设计了 FlexPRET 处理器[6].但是,他们重点关注程序时序行为的可重复性,以辅助最坏执行时间(worst-case execution time, WCET)分析,而我们以建立具有时间语义的计算机体系结构为目标,为程序员提供表达和控制程序时序行为的时间语义指令集,作为实现时序确定性系统的基础设施.

本文的主要贡献包括 3 个方面:

1) 提出支持时间语义的实时计算机体系结构模型 RTM.RTM 以冯·诺依曼机为参照,通过添加标准时钟 stdClk,建立 ISA 层次的时间模型.stdClk 作为定义系统时间语义的基础,允许程序员以实时间(real time)控制 RTM 机 I/O 操作的时间属性.同时,stdClk 与控制处理器执行的 CPU 时钟 cpuClk 协作,既最小化程序执行时间和 I/O 时序抖动,又实现了程序的 I/O 时序行为物理平台无关性.

2) 提出作为 RTM 模型软硬件接口的时间语义指令集 TTI.TTI 以 TTA 理论为基础定义最小时间语义指令集,为程序员提供直接表达和控制程序时序行为的时间语义指令.

3) 设计和实现实时处理器 RPU.RPU 基于 RTM 模型,为 RISC-V 指令集添加了 TTI 扩展,保证指令集时间语义.同时,它采用硬件粗粒度多线程技术(coarse-grain multithreading, CMT),以时间触发(time-triggered, TT)机制进行线程调度,保证任务级时间语义.

逻辑执行时间(logical execution time, LET)编程模型[7]是一种硬实时软件模型.本文通过 LET 任务集的设计示例,验证了上述工作的有效性.

1　相关工作

本节对当前实时计算设计方法、时间属性和行为建模、实时处理器设计 3 个方面的研究工作进行梳理.

1.1　实时计算设计方法

传统的实时嵌入式系统设计方法关注计算过程的时序可预测性,通过设计阶段进行时序分析来满足系统的时间约束.可调度性分析(schedulability analysis)[8]和端到端分析(end-to-end analysis)[9]都以任务作为最小分析单位,以任务的 WCET 作为输入.然而,现代计算机系统通过引入计算并行化和

访存缓存化等优化机制来提升平均性能,导致系统出现严重的时序不确定性问题,任务的 WCET 难以紧致确定[10].本质上,保证任务的 WCET 不是实时系统的目标,现有方法必须要在系统设计的同时进行 WCET 分析以保证系统的时序正确性,这使系统开发变得复杂.另一方面,将任务作为最小单位的设计分析方法也无法保证任务内/指令级操作的时间语义,即只能表达指令在一段时间内执行的语义,不能直接表达指令在某个时刻执行的语义.

20 世纪 90 年代以来,研究者逐步认识到实时系统诸多问题的根源:实时计算难以依托在通用计算之上,实时系统领域需要重新定义计算机系统的各个层次[11-12].研究表明:采用构件化软件体系结构、时间触发执行机制[13]和 LET 编程模型[7]构建时序关键系统是实现实时系统时序隔离(temporal isolation)、可预测(predictability)、可组合(composability)和可扩展(extensibility)的可行技术路线.DCW[14]在现代处理器上实现了反应式可预测的 LET 编程框架,但受通用计算机体系结构和任务执行机制的限制,LET 模型系统执行效率低下,需要具有时间语义的 ISA 和时序行为确定的体系结构支持.

1.2 时间属性和行为建模

20 世纪 50 年代,逻辑学家在数理逻辑(logic)框架中添加时间语义,建立了时态逻辑(temporal logic)框架,以表达和推理用时间修饰的谓词.Pnuel 将时态逻辑引入计算机科学领域[15].时态逻辑作为描述型的语义表达体系,适合系统建模和验证,但是难以应用于定义操作型语义的计算过程.

1991 年 Henzinger 为符号变迁系统(labeled transition system, LTS)添加时间语义,提出时间迁移系统(timed transition system, TTS)模型[16],用于建模复杂的并发实时系统.TTS 采用一个全局的虚拟时钟,时间域是实数集,时间随着事件的发生隐式地不等间距地步进.在全局时钟的度量下,TTS 约束了转移(transitions)发生时间的上下界,并用时间状态序列(timed state sequences)刻画系统的状态变化.TTS 作为 LTS,重点关注系统所处状态及其变迁,对状态变迁时发生的计算说明较少.

1991—1994 年,Alur 和 Dill 为自动机模型添加时间语义,提出和完善了时间自动机(timed automata, TA)[17],用以建模实时系统随着时间的发展而不断变化的行为.时间自动机采用离散时间模型(discrete-time model),该时间模型要求时间序列是单调递增的整数值,即时间域是正整数域.而它接受的时间词(timed words)则采用稠密时间模型(dense-time model).时间词是指一个由关联实数时间值的符号组成的无穷序列.时间自动机采用多时钟模型来表达系统的时间需求,通过系统时间与特定时刻的大于、小于和等于关系分别表达"在……之后""在……之前"和"在……时刻"的时间语义.时间自动机可以用来形式化地建模和验证实时系统的时间属性.相比于 TTS,TA 直接通过描述系统的计算过程来刻画系统的行为.另一方面,TTS 只能约束转移发生时间的上下界,而 TA 则可以直接表达操作发生于特定时刻的语义.

时间自动机基于多时钟模型和事件触发.2004 年 Krčál 和 Mokrushin 等人提出时间触发自动机(TTA)[18].TTA 基于单时钟模型,使用"$\lfloor n \rfloor$"和"(n)"表达"在 n 时刻"和"每 n 时刻"的时间语义.另外,时间触发自动机接收数字化(digitalized)的时间词.究其含义,时间自动机认为计算系统从外界接收连续的信息,而时间触发自动机认为计算系统从外部设备接收处理后得到的数字信号.

时间自动机理论和时间触发自动机理论对时间语义指令集的设计具有重要的指导意义,是研究时间语义完备性理论的重要参考.

1.3 实时处理器设计

Schoeberl 小组进行了一系列优化硬件最坏执行时间分析的研究.通用处理器的设计中,体系结构层次的新技术如流水线、高速缓存、分支预测和乱序执行等提升了系统的平均性能,但这些技术使程序的 WCET 分析越来越复杂,甚至难以进行.JOP[19]是 Java 虚拟机的硬件实现.JOP 采用简单的三段流水使 Java 程序的 WCET 分析更准确.Patmos 处理器[20]同样以优化 WCET 分析为目标,它采用为该目标定制的类 RISC 指令集.同时,Patmos 使用 Method Cache 作为指令 Cache,使用 Split Cache 作为数据 Cache,能够提前预知指令延迟和执行时间.Hahn 和 Reineke 关注 2 种时序异常,设计具有严格访存顺序的 SIC(strictly in-order core)[21].SIC 具有单调性,消除了时序异常并简化了 WCET 分析.

Roop 小组对支持同步语义的反应式处理器[22]进行研究,希望通过在指令集中增加同步语义来直接支持执行同步语言程序.同步编程语言[23]基于同步假设,通过信号管理,如轮询(polling)、发射(emission)和抢占(preemption)等进行程序的同步,保证了并发行为的确定性,易于程序验证.

HiDRA[24]是一个实时系统的软硬件协同设计方案,使用多个反应式核 REMIC[25]构造全局异步局部同步(globally asynchronous locally synchronous, GALS)的体系结构,提供类 Esterel 语言的同步语义.STARPro[26]与 HiDRA 思想一致,但 HiDRA 支持类 Esterel 的同步语义,而 STARPro 旨在直接支持 Esterel 程序执行.另外,Roop 团队提出基于 C 语言扩展的 PRET-C 同步语言,并提出支持 PRET-C 同步语义的微体系结构 ARPRET[27].ARPRET 分为 2 部分:1)GPP 实现通用计算;2)PFU 负责调度和保证功能的时间属性正确.

Lee 小组提出 PRET(precision timed)概念[3].他们认为硬件体系结构中的数字电路时钟是精确的,但是通用计算指令集抽象掉了这个精确的时间,所以为了程序的时序行为可重复,PRET 机要向上层提供这一精确的时间.该小组研制了多款 PRET 处理器[4,6,28].FlexPRET 处理器[6]从流水线、线程调度、时间指令、存储系统 4 个方面对处理器的时间属性进行规范,并用时间指令 DU(delay until)保证代码的执行时间下界,时间指令 EE(exception on expire)保证代码的执行时间上界.此方案可以约束指令序列执行时间的上下界,无法直接表达操作发生在某个时刻的语义.Wan 和 Li 等人在文献[29]中提出构建时间语义指令集的思想,拟添加时间语义和操作语义绑定的指令,但该想法处于比较早期的阶段.

总体而言,当前实时处理器设计研究缺乏 ISA 层理论模型支持.在通用计算中,图灵机[30]为通用计算机系统奠定坚实的理论基础,冯·诺依曼机[31]是通用计算机系统统一的实现模型.实时处理器理论基础的缺乏,导致时间语义难以表达,或表达能力边界不确定,程序的时序行为难以形式化验证.

2　背景知识

本节介绍具有时序确定性的系统构建范式时间触发范式以及按照时间触发范式构建的计算理论时间触发自动机,介绍时间模型的分类.后文定义时间模型,参考时间触发自动机理论设计了最小时间语义指令集 TTI,并以时间触发范式实现了实时处理器 RPU.

2.1　时间触发范式

事件触发范式和时间触发范式是 2 种截然不同的系统设计范式,本节参考文献[13],对 2 种范式进行介绍.2 种系统设计范式的核心区别是:在事件触发系统(event-triggered systems, ET 系统)中,重大事件的发生导致了响应活动的启动;在时间触发系统(time-triggered systems, TT 系统)中,响应活动是在预定的时间点实时启动的.

ET 系统使用中断机制感知外部事件的发生,2 类主要的事件是 P 类事件(predictable events)和 C 类事件(chance events).对于 P 类事件,可以提前保留将来传输和处理它们所需的资源.而 C 类事件可能在短时间内集中发生,这将超出 ET 系统的处理能力,系统必须有对这类事件的存储能力,并适当限制事件的感知.在 ET 系统中,判断任务集满足时间约束和找出可行的调度算法都是 NP 难的问题.因此在实践中,ET 系统往往采用静态优先级算法进行运行时调度,这导致 ET 系统的时间行为是运行时上下文相关的,且不具有单调性,无法保证系统的时间可预测性.设计者只有通过模拟负载进行广泛的系统测试才能判断 ET 系统的正确性.然而,C 类事件的存在导致难以对 ET 系统进行全面的测试.

TT 系统使用轮询机制获取外部实体的状态.轮询是周期性的,轮询发生的时间序列被称为观测格(observation grid);系统根据状态进行响应,响应发生的时间序列称为响应格(action grid).TT 系统需要事件传感器对外部事件进行处理,形成系统需要的状态信息.在 TT 系统的设计阶段,设计者首先确定系统的观测格和时间关键任务的最坏执行时间,然后在编译时期通过调度策略生成确定的调度表,在运行时期执行简单的表查找来调度任务.因此,TT 系统的时间行为是运行时上下文无关的,具有确定性.通过仔细的设计,TT 系统的设计者可以精确地预测任务的时序行为和系统的时序行为.另一方面,观测格和响应格决定了 TT 系统的时基是离散的,离散时基确保每一种输入情况都可以被观察和再现,因而 TT 系统的测试更加可构造和系统化.

2.2　时间触发自动机

时间触发自动机[18]是基于事件的非即时可观察性,对时间自动机进行简化而形成的计算模型.TTA 是在时间触发模式下运行的有限状态自动机,接受定时语言的数字化版本.本质上,TTA 是数字计算机(或数字控制器)的时间表,描述了在给定时间点该计算机应执行的操作,可以转换为计算机上的可执行程序.

TTA 采用一个全局参考时钟,时间域为整数集 N,它采用隐式的时间模型,把状态变化的时刻作为

隐式参考点.它接受的时间词是数字化后的时间词,更符合现代数字计算机的特征.TTA 只能在满足时序约束的整数时间点接收时间词(消耗外部事件)和发生转移.时序约束集合 Θ 包含周期性的和瞬时的(也称非周期性的)2 种时序约束,分别由 $a(n)$ 和 $a[n]$ 来表示.标有 $a(n)$ 的转移表示每 n 个时间单位应该检查 a 是否已发生;标有 $a[n]$ 的转移表示第 n 个时间单位应该检查 a 是否已发生.

定义 1. 时间触发自动机[18].时间触发自动机 A_{TTA} 定义为四元组 $\langle \Sigma, S, s_0, T \rangle$,其中:

1) Σ 是有限字符表,字符代表事件;

2) S 是有限状态集合;

3) $s_0 \in S$ 是初始状态;

4) $T \subseteq S \times \Sigma^* \times \Theta \times S$ 是转移边的有限集合.

注:Σ^* 是离散化后的 Σ.

定义 2. 时间触发自动机行为语义[18].初始状态为 $(s_0, 0)$ 的时间触发自动机 $\langle \Sigma, S, s_0, T \rangle$ 的语义由 3 个规则定义:

1) $(s_1, t) \xrightarrow{\omega} (s_2, 0)$,如果 $s_1 \xrightarrow{\omega[n]} s_2$ 且 $n = t$;

2) $(s_1, t) \xrightarrow{\omega} (s_2, 0)$,如果 $s_1 \xrightarrow{\omega(m)} s_2$ 且 $t \bmod m = 0, t > 0$;

3) $(s, t) \rightarrow (s, t + d)$,如果 $t + d \leqslant \lfloor t + 1 \rfloor$.

其中,使用 $s_1 \xrightarrow{\omega[n]} s_2$ 表示 $(s_1, \omega, [n], s_2) \in T$,使用 $s_1 \xrightarrow{\omega(n)} s_2$ 表示 $(s_1, \omega, (n), s_2) \in T$,使用 $\lfloor n \rfloor$ 表示不超过 n 的最大正整数.定义 2 中的第 3 条规则确保转移的发生不会导致有整数时间点被跳过.

图 1 是时间触发自动机的一个示例,其中的 null 符号表示接收长度为 0 的时间词,即不消耗事件发生转移.在初始状态 S_0 时,自动机在第 1 个时间单位检测开始命令 b 是否发生,如果 b 发生,自动机将接收 b 并迁移到运行状态 S_1.在运行状态 S_1 时,自动机每 1 个时间单位检测停止命令 e 是否发生,如果 e 发生,自动机将接收 e 并迁移到终止状态 S_3;每 3 个时间单位检测正常信号 r 是否发生,如果 r 发生,自动机将接收 r 并仍迁移到运行状态 S_1;每 3 个时间单位检测出错信号 w 是否发生,如 w 发生,自动机将接收 w 并迁移到故障状态 S_2.在故障状态 S_2,系统会进行复位,并在第 5 个时间单位迁移到 S_0 状态.注意,如果在 S_1 状态未检测到正常信号 r 的发生,状态机也会停留在 S_1 状态,但不会接收一个 r 事件.

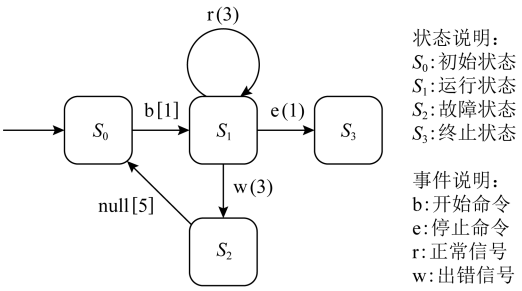


Fig. 1 An example of time-triggered automaton

图 1 时间触发自动机示例

2.3 时间模型

时间模型是实时系统的理论核心,从时间线的构成、时间步进规则和事件观测方法等 3 个方面刻画和定义系统的时间行为.本文参考文献[32],对描述时间模型 3 个方面的维度进行说明.

描述时间模型中时间线构成的维度有:

1) 离散的或者是稠密的.根据采用的时间域是离散集合还是稠密集合,时间模型分为离散时间模型和稠密时间模型.按照数学上集合连续性的概念,稠密时间模型又进一步分为连续时间模型和非连续时间模型.如果系统同时采用了离散时间模型和稠密时间模型,则系统的时间模型为混合时间模型.

2) 有界的或者是无穷的.按照系统生命周期内的行为是否为周期性的,可以分为有界时间模型和无穷时间模型.

3) 定序的或者是定量的.只能描述序的模型,能定序地定义时间关系,称为定序时间模型;能够描述序和时刻的模型,能定量地定义时间关系,称为定量时间模型.

4) 线性的或者是分支的.线性时间模型为线性结构,能够表示具有唯一未来行为的属性.分支时间模型为树结构,允许用户陈述具有分支未来行为的属性.已有研究表明线性时间模型和分支时间模型是可以相互转化的.

描述时间模型中时间步进规则的维度有:

1) 物理的或者是逻辑的.物理时间是指物理设备提供的时间,随实时间步进;逻辑时间是指使用某种逻辑关系对事件进行排序形成的虚拟时间,随事件发生步进.

描述时间模型中事件观测方法的维度有:

1) 显示的或者是隐式的.显示时间模型使用时域范围内的显示术语,显示表达时刻,有量词可以对时间进行描述;隐式时间模型无法描述时域量词,

隐含当前时刻的概念,相对于当前时刻对事件进行排序,没有量词进行度量.

2) 绝对的或者是相对的.绝对时间是指使用点时间(即时刻)进行描述的时间模型;相对时间则是指使用相对于当前时间的段时间进行描述的时间模型.

另外,在分布式的场景下,时间有全局的或者是局部的之分.全局时间是指所有节点统一同步后产生的时间;局部时间是指每个节点内自己使用的时间.

3　实时机模型

本文提出 RTM 机模型以解决当前处理器体系结构层缺乏时间语义的问题.RTM 机以冯•诺依曼机为基础,引入标准时钟 stdClk,并增加时间语义指令集.TTI 指令集参考 TTA 模型定义最小时间语义指令集,允许程序员以实时间概念定义 RTM 机 I/O 操作的时间行为属性.实时程序可以在基于 RTM 机的硬件平台之间进行时间语义一致的移植.

3.1　实时机模型 RTM

计算机科学领域的时间概念相当原始.图灵机和冯•诺依曼机模型基于顺序控制抽象,指令一条接一条执行,时间先后关系(temporal succession)是当前机器语言级唯一可用的时序关系.虽然定义冯•诺依曼机的程序逻辑行为无需显式地引用量化时间概念,但无法满足实时系统所需要的实时间约束定义.

定义 3. RTM 机.RTM 机对冯•诺依曼机进行扩充,如图 2 所示:

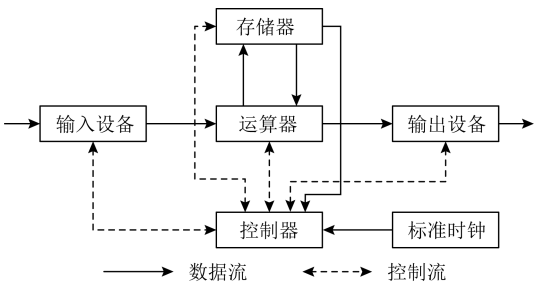


Fig. 2　RTM machine architecture

图 2　RTM 机结构模型

1) 计算机系统由运算器、控制器、存储器、标准时钟和 I/O 设备等功能部件构成.

2) 标准时钟用于操作定时,在保持冯•诺依曼机指令的逻辑功能不变的同时,增加约束特定操作的时序和限定操作序列(动作)的执行时间等时序功能,支持构建具有时间语义的指令集.

3) RTM 机的指令字由操作码、地址码和定时码构成.操作码定义操作的功能,地址码指定操作数和下一条指令的地址,定时码指定操作的时间属性和约束.

由于各个硬件平台的时钟设备不同,即使程序具有相同的时间语义,也无法得到相同的时间行为.RTM 机引入标准时钟,并据此定义其指令的时间语义,使具有时间语义的程序得到同一时间模型的支持,实现了程序的 I/O 时序行为物理平台无关性.

定义 4. RTM 机时间模型.RTM 机的时间模型定义为:

1) 时间线的构成是离散的、定量的、线性的时间域 N.

2) 时间步进规则是物理的,随牛顿时间步进.

3) 事件观测方法是显示的、绝对的.

RTM 机以支持紧致时间设计为目标,其核心思想不是简单地在 CPU 中增加标准时钟,而在于明确了时间行为控制抽象,并通过提供时间语义指令集,实现了时域控制的原子性和可组合性,并最小化(周期精确)定时抖动.

时间语义指令集的完备性问题是 RTM 机模型的一个基础性问题,限定了 RTM 机的时间语义表达能力边界.需要进一步基于实时逻辑和时间自动机等理论框架对其进行研究.我们给出一种支持时间触发的 RTM 机时间语义指令集 TTI.参考 TTA 模型,TTI 指令集至少需要支持 3 种操作:

1) 时间管理操作.获取全局参考时间并设置时间参考点.

2) 定时触发输入操作.无论是$[n]$还是(n),都需要定时对事件进行采样.

3) (n)原语的周期性操作.

参考以上需求,本文在 3.3 节中给出了 TTI 的定义.

RTM 机以“关注分离(separation of concerns)”和“构建正确(correct-by-construction)”为设计原则,区分逻辑控制与时域控制,为实现从时序可预测设计到时序确定性设计转变奠定了理论基础,利于实时程序代码自动生成和化简实时系统验证的复杂度.

3.2　标准时钟 stdClk

现代计算机系统主要有 2 种时钟域:1)CPU 时钟 cpuClk.用于驱动其数据通路中各个功能部件同步

运行.不同平台的 cpuClk 时钟频率不同,与应用程序的时序控制无关.文献[12]认为该时钟是精确的,但是通用计算机的 ISA 忽略了其时间精确性.2)操作系统和应用程序提供定时中断服务的时钟.该时钟与 cpuClk 异步,且时序粒度高几个数量级,程序员依赖该时钟进行粗粒度时序控制.因此,为了精确控制系统的时序,需要定义新的时钟.

定义 5. 标准时钟 stdClk.RTM 机的标准时钟 stdClk 是基于实时间域的物理时钟,用于实时程序的时间行为控制.

标准时钟的时间粒度依赖于应用场景,对时间精度要求较高的系统可采用粒度小(如 100 ns)的标准时钟,对时间精度要求较为放松的系统可采用粒度较大(如 50 μs)的标准时钟.引入标准时钟后,现代计算机系统通过 cpuClk 驱动指令执行,通过 stdClk 提供时域控制依据.cpuClk 与 stdClk 的协作,分离了实时计算系统中的逻辑控制和时域控制,即分离了驱动计算机运行和控制计算机时序行为的时钟,有利于支持确定的时间语义,便于形式化建模和分析.另一方面,通过标准时钟来精确定时,保证程序员以统一的实时间抽象控制程序的时序行为,使得不同平台上的定时程序具有时间行为一致性.

基于标准时钟 stdClk,RTM 机实时程序的逻辑执行与其时域控制具有确定性的松弛同步时序关系,如定理 1,可以实现周期精确的系统时序行为.

定理 1. stdClk 与 cpuClk 的关系定理.假设 stdClk 的周期为 T_{std},cpuClk 的周期为 T_{cpu},并假设 $T_{cpu}<T_{std}$,则它们有 2 种关系:

1) 当 cpuClk 与 stdClk 同步时,$\exists n\in N$,使得 $T_{std}=n\times T_{cpu}$.

2) 当 cpuClk 与 stdClk 异步时,$\exists n\in N$,$0\leqslant t<T_{cpu}$,使得 $T_{std}=n\times T_{cpu}+t$.

证明. 由模运算可知,$\exists 0\leqslant c<b$,使得 $a=c \bmod b$.即 $\exists 0\leqslant t<T_{cpu}$,使得 $T_{std}=t \bmod T_{cpu}$.当 cpuClk 与 stdClk 同步时,cpuClk 与 stdClk 上升沿重合,即 $t=0$,$T_{std}=0 \bmod T_{cpu}$,所以 $\exists n\in N$,使得 $T_{std}=n\times T_{cpu}$.当 cpuClk 与 stdClk 异步时,cpuClk 与 stdClk 无特别关系,$0\leqslant t<T_{cpu}$,所以 $\exists n\in N$,$0\leqslant t<T_{cpu}$,使得 $T_{std}=n\times T_{cpu}+t$. 证毕.

由定理 1 我们可以得到引理 1.

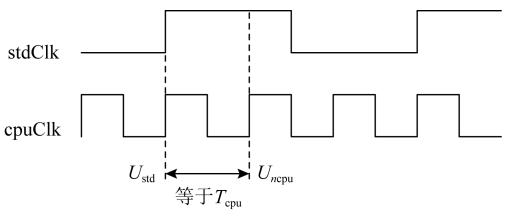
引理 1. 假设 stdClk 上升沿发生时间为 U_{std},该时间后下一个 cpuClk 上升沿发生时间为 U_{ncpu},则它们满足关系:

1) 当 cpuClk 与 stdClk 同步时,$U_{ncpu}-U_{std}=T_{cpu}$.

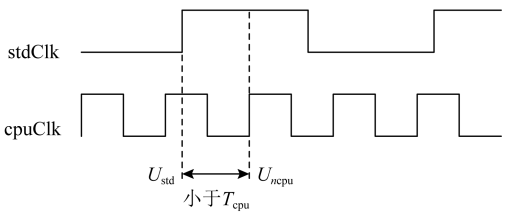
2) 当 cpuClk 与 stdClk 异步时,$0<U_{ncpu}-U_{std}<T_{cpu}$.

在同步情况下,引理 1 显然成立.异步情况下处于边界情况,即差值为 0 或者 T_{cpu} 时,异步情况转换为同步情况,可见异步情况同样成立.

定理 1 和引理 1 说明,可以将 stdClk 与 cpuClk 的最小偏差控制在一个 cpuClk 周期以内,最小化程序时序和 I/O 时序抖动.定理 1 和引理 1 展示的控制时序如图 3 所示.图 3(a)中,stdClk 与 cpuClk 为同步关系,其控制抖动为 T_{cpu};图 3(b)中,stdClk 与 cpuClk 为异步关系,其控制抖动小于 T_{cpu}.



(a) stdClk与cpuClk为同步关系



(b) stdClk与cpuClk为异步关系

Fig. 3 Timing of using stdClk for controlling

图 3 标准时钟的控制时序

实时系统技术往往应用在分布式场景下.而 RTM 系统的时间行为基于标准时钟,与运行平台无关,故在实时分布式场景中应用 RTM,只需要对标准时钟进行同步.

根据标准时钟提供的定时,设计带有时间语义的指令集作为实时处理器软硬件接口,可以提供 at, delay-until 和 timestamp 等控制时间行为的指令,支持逻辑执行时间模型和时间触发等应用模式.

3.3 时间语义指令集 TTI

RTM 机的 TTI 指令集主要包含标准时钟管理指令、时间触发操作指令、执行时间约束指令、任务并发管理指令 4 部分指令,以及时间粒度寄存器 *tg* (time granularity)、时间寄存器 *ti* 和时间戳寄存器 *ts*.其中,标准时钟管理指令对系统的标准时钟进行管理,生成系统时间,规范其应用方法;时间触发操作

指令定义操作和时间绑定的指令,保证了操作在指定时间触发执行;执行时间约束指令参考 PRET 机[12],保证一段代码的执行时间上下界;任务并发管理指令摒弃传统的使用中断的软件任务并发管理,提供确定性的硬件线程管理任务并发的能力.由于不同系统、甚至同一系统的不同运行模式需要不同的时间粒度,时间粒度寄存器 tg,用于用户自定义时间粒度;时间寄存器 ti 表示当前的系统时间,系统时间由系统根据标准时钟和系统时间粒度生成;分布式系统中,时间戳(timestamp)是支持定序的重要数据,时间戳寄存器 ts 保存系统与外界交互事件的时间戳.

Table 1　Functionality of TTI's Instructions

表 1　TTI 指令功能

汇编指令	功能
settg $r1$	$tg \leftarrow r1$
setti $r1$	$ti \leftarrow r1$
getti rd	$rd \leftarrow ti$
getts rd	$rd \leftarrow ts$
ttiat $rd,r2,r1$	if ($ti==r1$) then $rd \leftarrow [r2]$ else wait
ttoat $r3,r2,r1$	if ($ti==r1$) then $[r2] \leftarrow r3$ else wait
delay $r1$	if ($ti==r1$) then pass else wait
mtfd $r1$	保证该指令执行时,$ti \leqslant r1$
tkend	当前任务执行完毕
addtk $r1,r2$	添加一个调度表项,使得处理器在 $ti=r1$ 时切换到 $r2$ 线程执行

TTI 指令的功能如表 1 所示,表 1 中 $r1$,$r2$ 和 rd 为通用寄存器.TTI 指令集包括:

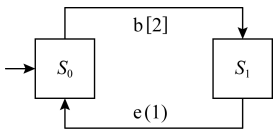
1) 标准时钟管理指令.setti/getti 指令用于设置/获取时间寄存器 ti,即系统时间;settg 指令用于设置时间粒度寄存器 tg,即系统当前时间粒度、时间向前自增的单位(time unit);getts 指令用于获取时间戳寄存器 ts,即系统上一条时间触发操作指令 ttiat/ttoat 的生效时间.

2) 时间触发操作指令.ttiat/ttoat 指令用于在指定时刻对 I/O 端口进行输入/输出;ttiat/ttoat 指令的执行代表系统与外部环境的交互事件,会设置时间戳寄存器 ts.

3) 执行时间约束指令.delay 指令用于约束代码的最短执行时间,即到该时间之前,delay 指令会阻塞后续代码执行;mtfd 指令用于约束代码的最坏执行时间,需要编译时期的检查配合完成,即保证执行该指令的时间不超过指定时间.

4) 任务并发管理指令.tkend 指令用于标志当前线程中任务的截止;addtk 指令用于向硬件时间触发调度表添加表项,即通过 addtk 指令设置好静态的时间触发调度表.

TTI 的正确执行需要保证其逻辑正确性和时间正确性.



(a) 示例 TTA

```
① S0:
②    #时间模型表达
③    getti x1
④    addi x1,x1,2
⑤    #b[2]语义表达
⑥    ttiat x2,x3,x1
⑦    …# if (x2==b) jump to S1 else loop forever
⑧ S1: # e(1)语义表达
⑨    #时间模型表达
⑩    getti x1
⑪ begin:
⑫    addi x1,x1,1
⑬    ttiat x2,x3,x1
⑭    …# if (x2==e) jump to S0
⑮    j begin
```

(b) 对应的 TTI 程序

Fig. 4　Example of translating TTA to TTI program

图 4　TTA 转换为 TTI 程序示例

3.4　TTI 的时间语义完备性分析

为了支持实时计算,需要回答 RTM 机应该具有什么时间语义,或其最小时间指令集应包含哪些指令等问题.RTM 机的逻辑功能指令集与冯•诺依曼机指令集等价,而 TTI 指令集的时间语义应至少具有与 TTA 相同的时间语义表达能力.

TTI 能够充分支持 TTA 的时间语义:1)支持 TTA 时间模型.TTA 采用一个全局的参考时钟,对应 TTI 中由标准时钟生成的系统时间 ti.时间触发自动机以进入状态作为时间参考点,TTI 可以使用 getti 获取当前系统时间,然后使用该时刻作为参考点.2)支持 $a(n)$ 语义.支持周期语义可以使用循环结合时间触发操作指令来实现.周期对应循环,获取周期开始时的系统时间 ti,使用 ttiat 在 $ti+n$ 时刻进行输入.如果输入的是 a,则发生状态迁移;否则,进入下一个周期.3)支持 $a[n]$ 语义.支持瞬时语义使用 ttiat 完成即可.

TTI 对时间触发自动机语义的充分支持,意味着所有 TTA 可以转换成 TTI 程序,即 TTI 是 TTA 时间语义完备的.一个简单的转换示例如图 4 所示,图 4(a)中 TTA 从 S_0 开始运行,在第 2 个时间单位接收 b 事件转移到 S_1;对应图 4(b) 中 S_0 代码段,S_0 代码段使用 getti 获取当前系统时间,将其加 2 作为 ttiat 的执行时刻,检查 ttiat 输入的事件是否为 b,如果是,转移到 S_1 代码段,否则永远停留在 S_0 代码段.图 4(a)在 S_1 状态时,TTA 每 1 个时间单位检测 e 事件是否发生,如果是,则接收 e 事件转移到 S_0 状态;对应图 4(b)中 S_1 代码段,S_1 代码段使用 getti 获取当前系统时间,将其加 1 作为 ttiat 的执行时刻,检查 ttiat 输入的事件是否为 e,如果是,转移到 S_0 代码段,否则,跳转到 begin 代码段,等待下一个时间单位检测 e 事件是否发生,实现周期语义.

4 实时处理单元

实时处理器 RPU 是 RTM 机的实例.4.1 节基于 RISC-V 指令集[33]定义 TTI 指令集,形成 RISC-V TTI 功能模块.4.2 节给出基于经典五段流水设计的 RPU 的原理图,RPU 添加了粗粒度多线程和时间触发执行等确保时序属性的机制.接着,本文以开源 32 位 RISC-V 处理器 CV32E40P[34]为框架对 RPU 进行实现,并以软核的方式部署到 Xilinx Artix-7 XC7A100T FPGA 上,实现时 RPU 采用的 stdClk 频率为 1 MHz,cpuClk 频率为 25 MHz. 4.3 节和 4.4 节给出 Xilinx Vivado 2019 仿真和综合得到的时序结果和资源消耗结果.

4.1 RISC-V TTI

RISC-V 是一个典型的三操作数、加载-存储形式的 RISC 指令集架构,由基础指令集和扩展指令集组成.基础指令由基本整数指令构成,扩展指令分为标准扩展和非标准扩展两大类.现阶段的标准扩展有乘法和除法扩展(M)、原子指令扩展(A)、单精度(F)和双精度(D)运算扩展等.非标准扩展作为一个高度特殊化的扩展,允许用户根据需求自定义完成.

本文基于 RISC-V 非标准扩展对 TTI 指令集进行定义,采用 RISC-V 预留的 *opcode* 域 custom-0,根据 *funct*3 区分上述四类指令,如表 2、表 3 所示.RISC-V TTI 扩展与原 RISC-V 指令集采用相同的寻址模式,有所不同的是,ttoat 指令有 3 个操作数,都是读寄存器,即要从寄存器组中读出 $r1$,$r2$ 和 $r3$ 寄存器,最后在 $r1$ 指定的时刻将 $r3$ 写入 $r2$ 表示的输出端口上.本文的实现平台,开源 RISC-V 核 CV32E40P 的通用寄存器组有 3 个寄存器读端口,因此扩展工作不会因为 ttoat 的特殊增加额外开销.

Table 2 Word Format of R-Type Extended TTI Instructions

表 2 R 类型 TTI 扩展指令的指令字格式

31:25	24:20	19:15	14:12	11:07	06:00	指令名
*funct*7	*rs*2	*rs*1	*funct*3	*rd*	*opcode*	
0000000	×	src	sc	×	custom-0	settg
0000001	×	src	sc	×	custom-0	setti
0000010	×	×	sc	dest	custom-0	getti
0000011	×	×	sc	dest	custom-0	getts
0000000	base	time	tio	dest	custom-0	ttiat
0000000	×	time	et	×	custom-0	delay
0000001	×	time	et	×	custom-0	mtfd
0000000	×	×	cmt	×	custom-0	tkend
0000001	id	time	cmt	×	custom-0	addtk

注:custom-0 为 0b0001011;sc 为 0b000;tio 为 0b001;et 为 0b010;cmt 为 0b011;×为未使用.

Table 3 Word Format of Custom-Type Extended TTI Instructions

表 3 自定义类型 TTI 扩展指令的指令字格式

31:25	24:20	19:15	14:12	11:07	06:00	指令名
*funct*7	*rs*2	*rs*1	*funct*3	*rs*3	*opcode*	
0000001	base	time	tio	src	custom-0	ttoat

注:custom-0 为 0b0001011; tio 为 0b001.

4.2 RPU 的设计

本节基于经典五段流水线[35]给出以 TTI 为软硬件接口的 RPU 的设计.相较于经典五段流水新增/改动部分的原理图如图 5 所示.本文采用 PLL 分频的方法产生 1 MHz 的标准时钟 *stdClk*,可参考其他文献采用更加精确的实现方法.分频的方法产生的 *stdClk* 与 *cpuClk* 周期是整倍数关系,且 *stdClk* 与 *cpuClk* 上升沿同步.由引理 1 可知,在本文的 *stdClk* 与 *cpuClk* 协同下,控制将会在 *stdClk* 生效后的 T_{cpu} 时间生效.

图 5 中的 Decode 模块增加了对 RISC-V TTI 指令的译码,Control 模块增加了相应的控制.图 5 中的 Time Unit 模块负责管理 *stdClk*,产生系统时间,如图 6 所示.时间粒度寄存器保存通过 settg 指令设置的时间粒度;*cnt* 寄存器直接接收标准时钟进行自增,当 *cnt* 寄存器增长到与 *tg* 寄存器相等的

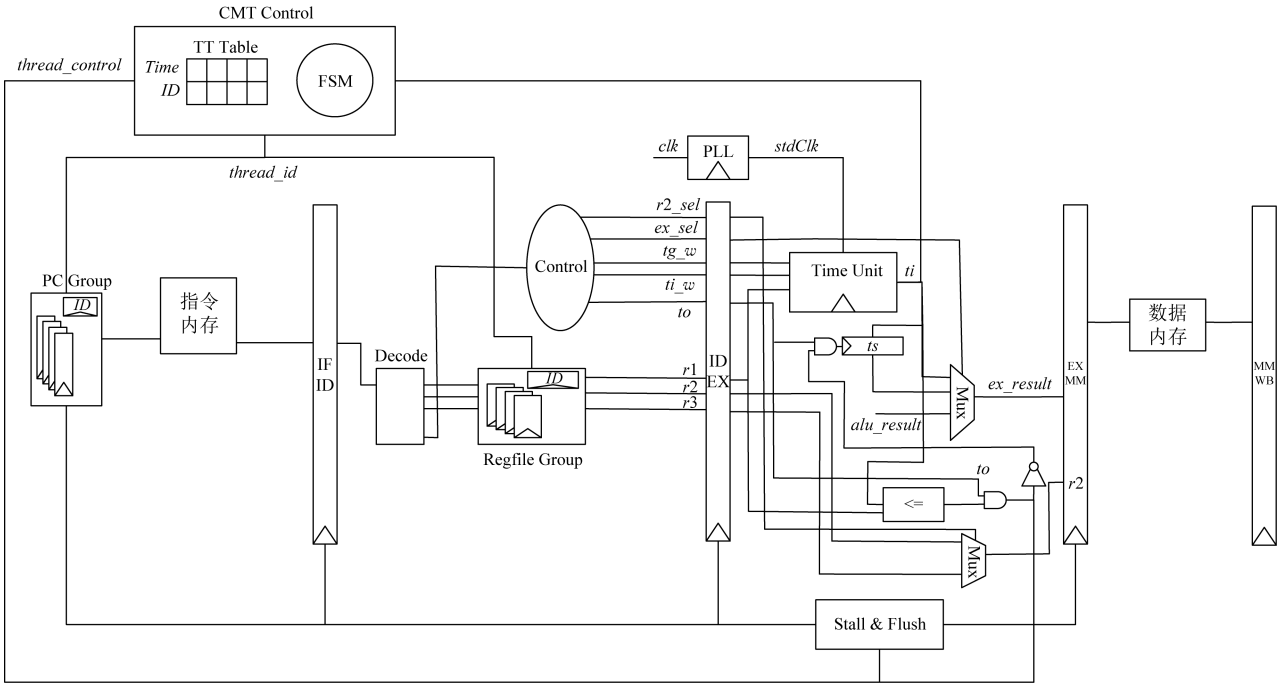


Fig. 5　Block diagram of RPU (Only incremental part of classic pipelines)

图 5　RPU 原理图:经典五段流水改动部分

时候,表示系统时间的时间寄存器 *ti* 自增;*ti* 寄存器也可通过 setti 指令进行设置,通过 getti 指令进行读取.

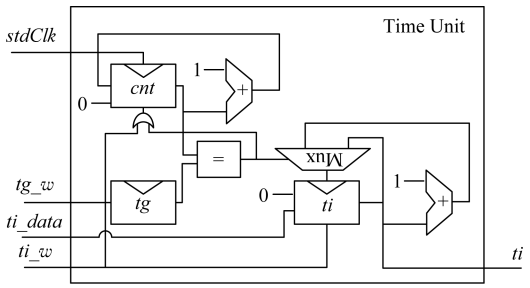


Fig. 6　Block diagram of Time Unit

图 6　Time Unit 原理图

时间触发操作指令按照 TT 机制执行.*to* 信号生效后,RPU 比较 *ti* 寄存器和 *r*1,若 $ti<r1$,表示未到触发时间,RPU 通过 Stall & Flush 模块停顿流水线;若 $ti=r1$ 寄存器,RPU 执行当前时间触发操作指令,并将当前 *ti* 寄存器值记录到时间戳寄存器 *ts*,*ts* 寄存器可通过 getts 进行读取;若 $ti>r1$ 寄存器,表示系统发生时序错误异常.对 TTI 中时间触发操作指令的实现,保证了 RPU 指令级的时间可预测属性.

RPU 采用粗粒度多线程支持 TTI 的并发管理指令.CMT 需要 RPU 有多套硬件上下文,体现在图 5 中的 PC Group 和 Regfile Group 模块.线程采用 TT 机制切换体现在图 5 中的 CMT Control 模块,该模块根据由标准时钟生成的 *ti* 进行硬件线程控制.addtk 指令会向 TT Table 中增加表项,RPU 根据 TT 表控制线程切换.TT 表结构如图 7 所示,它使用图示结构模拟队列,图 7 中的 *Time* 和 *ID* 寄存器组为数组,*head* 和 *tail* 两个寄存器为头尾指针,通过 *add* 信号添加表项 *add_time* 和 *add_id*;在表头指定的时刻弹出 *head_time* 和 *head_id*,进行线程切换控制.线程也可通过执行 tkend 表示当前任务执行完成,主动释放 RPU 控制权.CMT Control 模块中的 FSM 负责线程切换控制,其时序如图 8 状态机所示.FSM 正常情况下运行在 S_0 状态,当 TT 表弹出表头或者线程执行 tkend 释放控制时,FSM

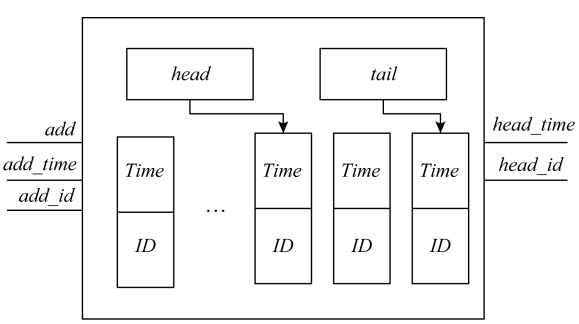


Fig. 7　Structure of TT table

图 7　TT 表结构

跳转到 S_1 状态;在 S_1 状态,RPU 会清空 IF 段、ID 段和 EX 段,保存正在执行线程的 PC,并执行 WB 段,这 3 个任务可并行完成,FSM 在下个 *cpuClk* 上升沿跳转到 S_2;在 S_2 状态,RPU 执行 MM 段,执行 MM 段与执行 WB 段要串行完成,所以需要单独一个 *cpuClk*;至此,上个线程的上下文保存完毕;在 S_3 状态,RPU 执行线程切换,更改 PC Group 和 Regfile Group 中的 *ID* 寄存器,随后进入 S_0 运行新线程.对 TTI 中并发管理指令的实现,保证了 RPU 任务级的时间可预测属性.

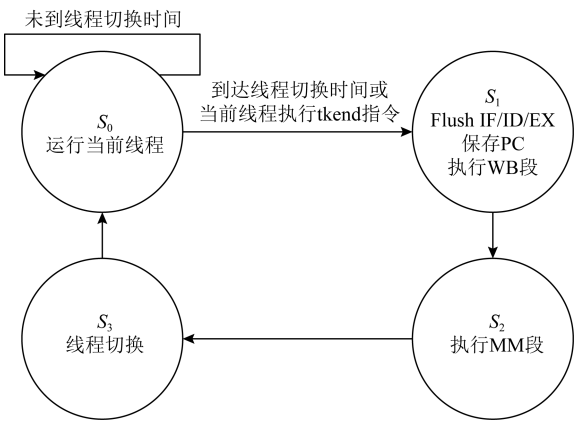


Fig. 8 FSM of thread switching

图 8 线程切换状态机

RPU 是一款指令级时间可预测和任务级时间可预测的实时处理器.它支持标准时钟,是 RTM 的实例;以 TTI 为软硬件接口,从指令集体系结构层次为上层提供了时间语义;RPU 的设计实现验证了 RTM 和 TTI 的可行性.同时,RPU 的时间触发设计理念契合时间触发通信协议,如 TTP 和 FlexRay 等.将来,在分布式场景下,可以为 RPU 添加时间触发通信协议接口,从而保证系统级时间可预测性.

4.3 时序结果

RPU 的时间可预测性体现在其具有确定性的时序上,即 TTI 的指令在运行时都具有时序确定性,图 9、图 10 和图 11 显示了系统时间生成的时序、时间触发操作指令执行的时序和线程切换的时序.

图 9 是 Time Unit 模块中对标准时钟管理的时序图.为了清楚表示时序,图中波浪部分隐去若干 *cpuClk* 周期.图 9 中系统设置的时间粒度 $tg=100$,即系统时间的单位是 100 μs.可以看出,在标为 2 的 *cpuClk*,*cnt* 与 *tg* 相等,在标为 3 的 *cpuClk*,*ti* 从原来的 2 变成了 3,与引理 1、4.1 节中的分析吻合.结合 *ti*,*ti_w* 和 *ti_data* 信号在标为 5 和标为 6 的 *cpuClk* 的变化可知 setti 指令功能正确.整体来说,标准时钟产生系统时间 *ti* 的过程时序精确,抖动总是 $T_{cpu}=40$ ns.

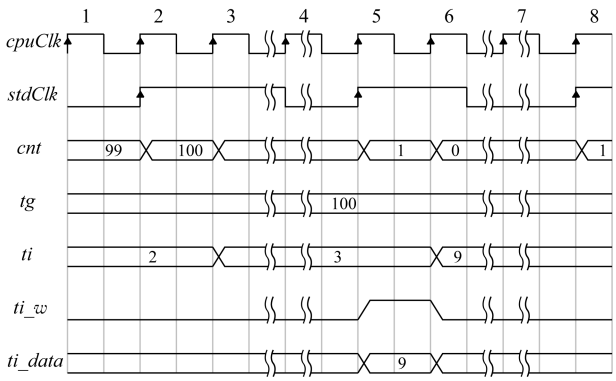


Fig. 9 Timing of Time Unit

图 9 Time Unit 时序

图 10 显示的是时间触发操作指令的运行时序,*to* 信号表示当前 EX 段有一条 ttiat 或 ttoat 指令;*idex_r*1 表示该指令的触发时间,*to_stall* 表示是否因 tio 指令产生了流水线阻塞.从图 10 可知,在标为 2 的 *cpuClk* 一条 tio 指令进入流水线,在 EX 段等待时间变为 20 时时间触发执行;在标为 7 的 *cpuClk*,即 *ti* 到达 20 的下一个 *cpuClk* 周期开始执行.时间触发操作指令的执行时序精确,抖动也是 $T_{cpu}=40$ ns.

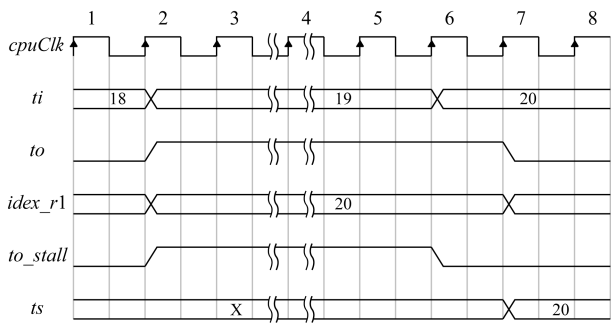


Fig. 10 Timing of time-triggered operation

图 10 时间触发操作指令时序

图 11 则展示了线程切换时序,*head_time* 表示 TT 表中下一个切换发生时间,*curr_stat* 表示 FSM 的当前状态,*thread_control* 表示 FSM 提供给 RPU 的控制信号,*ID* 表示 PC 中的 *ID* 寄存器.图 11 和图 8 所示的线程切换状态机吻合,说明 RPU 的并发管理同样 *cpuClk* 周期精确,时间可预测.

图 9、图 10 和图 11 体现了 RPU 的时序确定性,时序确定有利于建立 RPU 的确定性抽象模型,并据此进行严谨的形式化时间行为分析,满足硬实时安全性设计要求.

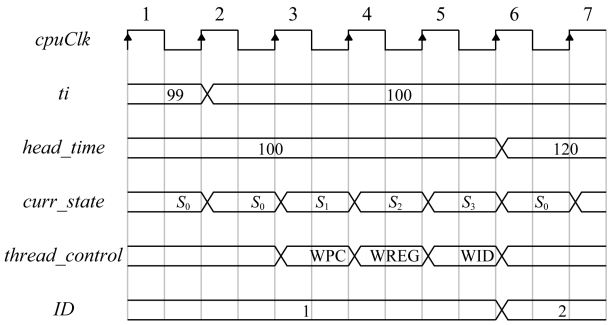


Fig. 11　Timing of thread switching

图 11　线程切换时序

4.4　资源消耗

RPU 基于 CV32E40P 实现.实现之前,本文对 CV32E40P 进行了裁剪,删去了如硬件循环等影响处理器时序的功能.裁剪后 CV32E40P 的资源消耗如表 4 所示.本文关注 FPGA 资源消耗 3 个重要指标:查找表(lookup table, LUT)、FF(flip-flop)和 BRAM(block ram),但是 CV32E40P 未用到 BRAM,因此没有列出 BRAM 数据.表 5、表 6 测得 RPU 线程数为 2,4,8,16,32 时,TT 表项大小为 4,8,16,32,64 时 LUT 和 FF 的消耗.

Table 4　Resource Consumption Table of Modified CV32E40P

表 4　裁剪后 CV32E40P 的资源消耗表

LUTs	FFs
5 320	1 711

Table 5　LUT Consumption Table of RPU

表 5　RPU LUT 消耗表

线程数	TT 表大小				
	4	8	16	32	64
2	6 310	6 353	6 429	6 588	6 923
4	10 410	10 456	10 534	10 623	11 243
8	18 679	18 723	18 803	18 974	19 322
16	32 182	32 227	32 310	32 485	32 859
32	53 180	53 226	53 312	53 491	53 873

Table 6　FF Consumption Table of RPU

表 6　RPU FF 消耗表

线程数	TT 表大小				
	4	8	16	32	64
2	2 929	3 064	3 331	3 868	4 933
4	4 973	5 112	5 387	5 940	7 036
8	9 152	9 295	9 578	10 149	11 276
16	17 514	17 661	17 952	18 539	19 698
32	33 834	33 985	34 284	34 887	36 079

从表 5 和表 6 可以看出,RPU 的实现会增加 LUT 和 FF 资源的使用,FF 的额外消耗多于 LUT 的额外消耗,这是由于 RPU 会更多地使用寄存器导致的.另外,LUT 和 FF 的资源消耗随着线程数大小的增长(表的每一列)和 TT 表大小的增长(表的每一行)线性增长.其中,资源消耗随线程数的增加增长迅速,随 TT 表大小的增加增长缓慢,符合设计时的预期.

5　LET 应用示例

本节首先对 LET 模型和 LET 模型的 3 种实现语义进行简单介绍,进而给出根据其中的交错 LET 实现方案使用 TTI 对 RPU 编程的应用实例.

5.1　LET 模型介绍

Henzinger 和 Kirsch 等人借鉴 ZET 和 BET 模型各自的优点,提出 LET 模型[7].采用 LET 模型时,任务总是在其激活区间的开始处读数据和结束处写数据,使其可观察的时序行为独立于任务的物理执行.LET 确定了读程序输入到写程序输出之间所需的时间,而不考虑执行程序所需的时间,具有平台可移植性.虽然实际应用时 LET 模型的资源(缓存、时间)利用率较低,但 LET 为控制工程师和软件工程师提供了清晰的时序模型接口.2017 年以来,随着多核平台下可预测性成为关键问题,工业界认为 LET 对于解决多核混合安全关键应用和分布式实时系统的通信确定性问题极具吸引力.

LET 编程模型将任务的逻辑行为“输入-计算-输出”过程相分离,如图 12 所示,定义了严格的输入输出时刻,要求在指定时刻输入,在输出时刻之前计算出结果,并能够将输出结果保持到指定时刻输出,具有时间行为可预测、可组合和平台无关性等重要特征,利于设计时进行系统行为验证.

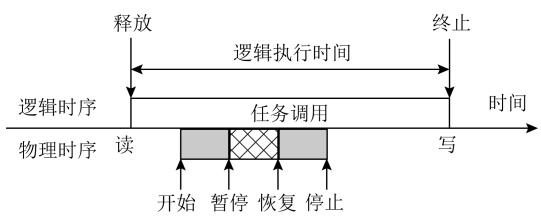


Fig. 12　LET programming model

图 12　LET 编程模型

LET 抽象中输入和输出都是零执行时间的,然而现实中输入和输出操作需要一定的执行时间,

因此,在实践中要尽量满足 LET 模型零执行时间的假设.在考虑系统端到端延迟的情况下,LET 模型主要可以分为 3 种实现语义[36],如图 13 所示.

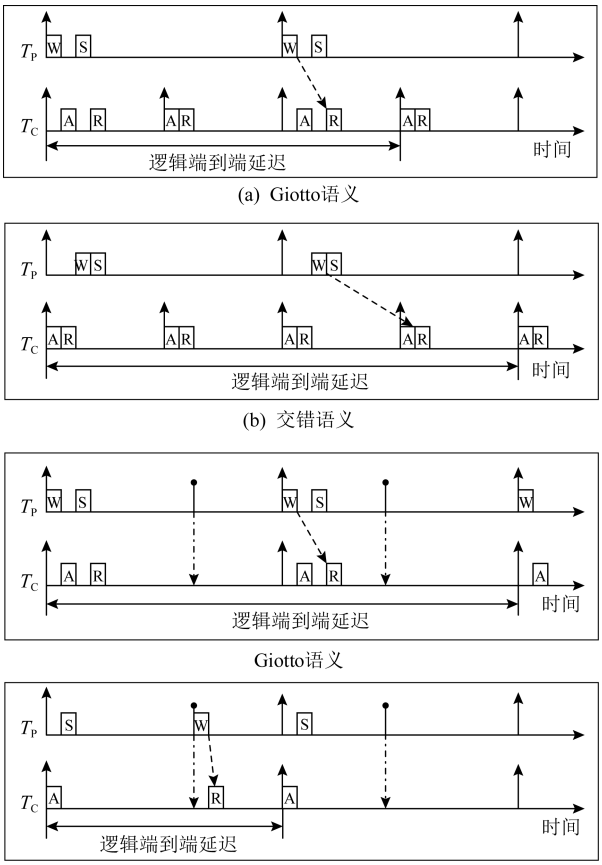


Fig. 13 Example schedule of three semantics of LET implementation

图 13 LET 模型 3 种实现语义

图 13(a)和图 13(b)中 P 任务的周期为 C 任务的 2 倍,图 13(c)中 P 任务和 C 任务同周期.图 13(a)显示的是 LET 的 Giotto 语义,它按照输入输出任务块的因果关系确定执行顺序.图 13(b)显示的是 LET 的交错通信语义,在 LET 通信阶段,每个任务的输入输出操作被合并为一组任务,不同的任务组交错执行.基于任务链的 LET 语义可以在特定情况下优化不可观测事件的通信,降低通信开销,优化任务链的端到端延迟.图 13(c)显示的是 LET 的 Giotto 语义和基于任务链的 LET 语义的对比,其上半部分是按照 Giotto 语义执行的结果,下半部分是按照基于任务链的 LET 语义执行的结果.从图 13 中可以看出,基于任务链的 LET 语义具有更低的逻辑端到端延迟.

5.2 RPU 对 LET 的支持示例

本节通过示例 RPU 对逻辑执行时间模型的支持,说明 RPU 的应用能力,提供时间可预测的实时应用开发平台.

本节使用的示例说明在自动驾驶中,假设有:

1) 采集任务,负责通过传感器采集数据,并进行数据预处理,周期为 2 ms;

2) 分析任务,对预处理之后的数据进行预定算法的分析和处理,得到响应数据,周期为 1 ms,WCET 为 0.25 ms;

3)执行任务,将响应数据分发到执行器进行外部环境的控制,周期为 2 ms,WCET 为 0.5 ms.

3 个任务的数据依赖图如图 14 所示:

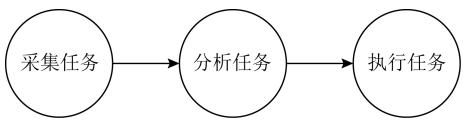


Fig. 14 Data dependency graph of sample tasks

图 14 示例任务的数据依赖图

本文按照交错 LET 的语义对示例任务集进行编程,采用共享内存进行通信.图 15 显示了该任务集的任务分解和建模时序,图 14 中的数据依赖体现为图 15 中的虚线箭头.其中,每个 LET 任务被分解为 I,C,O 三个子任务, O 和 I 两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,显示出图 15 所示时序.

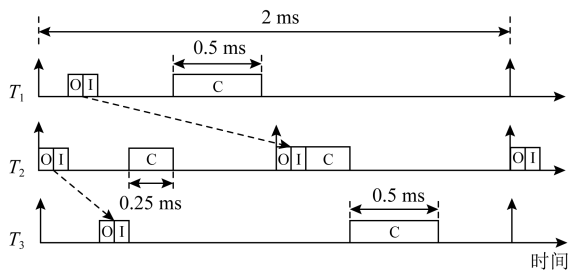


Fig. 15 Interleaved LET implementation

图 15 交错 LET 语义实现方案

图 15 中 OI 子任务组在 RPU 上会执行 0.05 ms.图 16 是使用 TTI 在拥有 4 线程的 RPU 上实现图 15 所示时序的伪代码.TT 表在时间触发地执行线程切换时会弹出表头项,对于图 15 中以 2 ms 为周期的运行系统,需要一个额外任务负责向调度表添加每个周期的调度表项.图 16 的代码中,使用 0 号线程管理调度表;使用 1 号线程执行各个任务的 OI 子任务组;使用 2 号线程执行各个任务的 C 子任务;

使用 3 号线程执行软实时任务,即当硬实时任务执行完,RPU 处于空闲状态时,会切换到 3 号线程执行.结合 4.2 节展现的时序可知,图 16 所示的代码保证了图 15 的交错 LET 语义.

```
① thread0:  # init andadd entry to TT table
② init:
③     addi a1,x0,50
④     settg a1  # set time granularity to 50 μs (0.05 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,a0,1  # base time+1×0.05 ms
⑫     addtk a0,a1  # task entry for OI subtask of task1
⑬     addi a0,a0,1  # base time+2×0.05 ms
⑭     addtk a0,a1  # task entry for OI subtask of task3
⑮     addi a0,a0,1  # base time+3×0.05 ms
⑯     addtk a0,a2  # task entry for first C subtask of task2
⑰     addi a0,a0,5  # base time+8×0.05 ms
⑱     addtk a0,a2  # task entry for C subtask of task1
⑲     addi a0,a0,12 # base time+20×0.05 ms
⑳     addtk a0,a1  # task entry for second OI subtask of task2
㉑     addi a0,a0,1  # base time+21×0.05 ms
㉒     addtk a0,a2  # task entry for second C subtask of task2
㉓     addi a0,a0,5  # base time+26×0.05 ms
㉔     addtk a0,a2  # task entry for C subtask of task3
㉕     addi a0,a0,12 # base time+38×0.05 ms
㉖     addtk a0,x0  # task entry for thread0
㉗     tkend
㉘     addtk 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,x0,2  # init base time 2×0.05 ms
③ t2oi1:  # first OI subtask of task2
④     ...
⑤     ttoat a4,a3,a0  # ex time,base time
⑥     ttiat a2,a1,a0
⑦     ...
⑧     tkend
⑨ t1oi:  # OI subtask of task1
⑩     ...
⑪     addi a0,a0,1  # ex time,base time+1×0.05 ms
⑫     ttoat a4,a3,a0
⑬     ttiat a2,a1,a0
⑭     ...
⑮     tkend
⑯ t3oi:  # OI subtask of task3
⑰     ...
⑱     addi a0,a0,1  # ex time,base time+2×0.05 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,a0,18  # ex time,base time+20×0.05 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,willrun at base time+3×0.05 ms
③     ...
④     tkend
⑤ t1c:  # C subtask of task1,willrun at base time+8×0.05 ms
⑥     ...
⑦     tkend
⑧ t2c2: #2nd C subtask of task2,will run at base time+21×0.05 ms
⑨     ...
⑩     tkend
⑪ t3c: # C subtask of task3,willrun at base time+26×0.05 ms
⑫     ...
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 LET implementation

图 16　LET 实现示例

6　总　　结

由于现代计算机体系结构不显式地提供时间语义支持,实时嵌入式系统设计只能依赖定时器的时钟中断进行时序控制,导致程序时间行为难以预测,且验证困难,平台依赖性高.

本文尝试重新定义支持实时计算的处理器体系结构和指令集,使程序员可以准确描述任务、动作和操作的确定性时间行为语义,以满足实时系统设计时域控制与值域控制相分离,时序行为可预测、可组合,且平台无关等时间关键性要求.

实时处理器 RPU 基于 RTM 模型和 TTI 指令集,实验表明其具备良好的实际应用前景.未来,我们将进一步完善所提出的实时处理器模型,并研究据此进行实时系统时间安全属性形式化推理和检查的方法.同时,我们也将进一步丰富 RPU 提供的实时服务,更好地支持实时编程模型.

参 考 文 献

[1] Sun Beilei, Li Xi, Wan Bo, et al. Definitions of predictability for cyber physical systems [J]. Journal of Systems Architecture, 2016, 63: 48-60

[2] Chen Xianglan, Li Xi, Wang Chao, et al. Research on real-time machine model and instruction set with time semantics [J]. Computer Engineering and Science, 2021, 43(4): 571-578 (in Chinese)
(陈香兰, 李曦, 汪超, 等. 实时机模型及时间语义指令集研究[J]. 计算机工程与科学, 2021, 43(4): 571-578)

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

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

[5] 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

[6] 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 (RTAS). Piscataway, NJ: IEEE, 2014: 101-110

[7] 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

[8] Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment [J]. Journal of the ACM, 1973, 20(1): 46-61

[9] Becker M, Dasari D, Mubeen S, et al. Analyzing end-to-end delays inautomotive systems at various levels of timing information [J]. ACM SIGBED Review, 2018, 14(4): 8-13

[10] Schoeberl M. Time-predictable computerarchitecture [J]. EURASIP Journal on Embedded Systems, 2009, 2009: 1-17

[11] Stankovic J A. Misconceptions about real-time computing: A serious problem for next-generation systems [J]. Computer, 1988, 21(10): 10-19

[12] Edwards S A, Lee E A. The case for the precision timed (PRET) machine [C] //Proc of the 44th Annual Design Automation Conf. New York: ACM, 2007: 264-265

[13] Karshmer A, Nehmer J. Operating Systems of the 90s and Beyond [M]. Berlin: Springer, 1991

[14] Wan Bo, Li Xi, Zhang Bo, et al. DCW: A reactive and predictable programming framework for LET-Based distributed real-time systems [J]. ACM Transactions on Design Automation of Electronic Systems, 2019, 24(3): 1-35

[15] Pnueli A. The temporal logic of programs [C] //Proc of the 18th Annual Symp on Foundations of Computer Science (SFCS 1977). Piscataway, NJ: IEEE, 1977: 46-57

[16] Henzinger T A, Manna Z, Pnueli A. Timed transition systems [C] //Proc of Workshop/School/Symp of the REX Project (Research and Education in Concurrent Systems). Berlin: Springer, 1991: 226-251

[17] Alur R, Dill D L. A theory of timed automata [J]. Theoretical Computer Science, 1994, 126(2): 183-235

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

[19] Schoeberl M. JOP: A Java optimized processor [C] //Proc of the Move to Meaningful Internet Systems 2003: OTM 2003 Workshops. Berlin: Springer, 2003: 346-359

[20] Schoeberl M, Schleuniger P, Puffitsch W, et al. Towards a time-predictable dual-issue microprocessor: The Patmos approach [C] //Proc of Bringing Theory to Practice: Predictability and Performance in Embedded Systems. Grenoble, France: INRIA, 2001: 11-21

[21] Hahn S, Reineke J. Design andanalysis of SIC: A provably timing-predictable pipelined processor core [C] //Proc of 2018 IEEE Real-Time Systems Symp (RTSS). Los Alamitos, CA: IEEE Computer Society, 2018: 469-481

[22] Roop P S. Predictablereactive processors for next generation computing: A proposal [J/OL]. School of Engineering Report, 2008, 662. [2021-05-19]. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.6620&rep=rep1&type=pdf

[23] Berry G, Gonthier G. The Esterel synchronous programming language: Design, semantics, implementation [J]. Science of Computer Programming, 1992, 19(2): 87-152

[24] Salcic Z, Dong Hui, Roop P S, et al. HiDRA—A reactive multiprocessor architecture for heterogeneous embedded systems [J]. Microprocessors and Microsystems, 2006, 30(2): 72-85

[25] Salcic Z, Dong Hui, Roop P, et al. REMIC: Design of a reactive embedded microprocessor core [C] //Proc of the 2005 Asia and South Pacific Design Automation Conf. New York: ACM, 2005: 977-981

[26] Yuan S, Andalam S, Yoong L H, et al. STARPro—A new multithreaded direct execution platform for Esterel [J]. Electronic Notes in Theoretical Computer Science, 2009, 238(1): 37-55

[27] Andalam S, Roop P, Girault A, et al. PRET-C: A new language for programming precision timed architectures [D]. Grenoble, France: INRIA, 2009

[28] Liu I, Reineke J, Broman D, et al. A PRET microarchitecture implementation with repeatable timing and competitive performance [C] //Proc of 2012 IEEE 30th Int Conf on Computer Design (ICCD). Piscataway, NJ: IEEE, 2012: 87-93

[29] Wan Bo, Li Xi, Luo Haizhao, et al. Work-in-Progress: TTI: A timing ISA for LET model in safety-critical systems [C] //Proc of 2017 IEEE Real-Time Systems Symp (RTSS). Piscataway, NJ: IEEE, 2017: 363-365

[30] 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

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

[32] Furia C A, Mandrioli D, Morzenti A, et al. Modeling time in computing: A taxonomy and a comparative survey [J]. ACM Computing Surveys, 2010, 42(2): 1-59

[33] Waterman A S. Design of the RISC-V instruction setarchitecture [D]. Berkeley, CA: UC Berkeley, 2016

[34] Institut für Integrierte Systeme, ETH Zurich. PULP platform [EB/OL]. [2021-02-18]. https://www.pulp-platform.org.

[35] Patterson D A, Hennessy J L. Computer Organization and Design: The Hardware/Software Interface [M]. 3rd ed. Amsterdam: Morgan Kaufmann, 2005

[36] Biondi A, Di Natale M. Achieving predictable multicore execution of automotive applications using the let paradigm [C] //Proc of 2018 IEEE Real-Time and Embedded Technology and Applications Symp (RTAS). Piscataway, NJ: IEEE, 2018: 240-250

Chao Wang, born in 1999. Master candidate. His main research interest is computer architecture.

汪　超,1999 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 is system software.

陈香兰,1977 年生.博士.主要研究方向为系统软件.

Zhang Bo, born in 1994. PhD candidate. His main research interest is computer architecture.

章　博,1994 年生.博士研究生.主要研究方向为计算机系统结构.

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

李　曦,1963 年生.教授.主要研究方向为计算机系统结构.

Wang Chao, born in 1985. Associate professor. His main research interest is computer architecture.

王　超,1985 年生.副教授.主要研究方向为计算机系统结构.

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

周学海,1966 年生.教授.主要研究方向为计算机系统结构.