计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220731
60(5): 1021−1036, 2023

实时计算机系统结构综述

龚小航　蒋滨泽　陈香兰　高银康　李　曦
（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　215123）
（gxh2018@mail.ustc.edu.cn）

Survey of Real-Time Computer System Architecture

Gong Xiaohang, Jiang Binze, Chen Xianglan, Gao Yinkang, and Li Xi
（*School of Computer Science and Technology*, *University of Science & Technology of China*, *Hefei* 230027）
（*Suzhou Institute for Advanced Research*, *University of Science & Technology of China*, *Suzhou*, *Jiangsu* 215123）

Abstract　In time-sensitive embedded systems, tasks need to meet their deadlines, and missing deadlines can significantly affect the quality of service or have catastrophic consequences. Compared with the general system, the research progress of real-time system is slow, even many basic concepts have not reached consensus. Precision timed (PRET) machine and real-time processing unit (RPU) are two existing real-time system solutions. Taking them as examples, we introduce the related concepts of real-time system, expound the problems in the development of real-time system, and compare the similarities and differences. The problems encountered at each level of the real-time system and the existing solutions are analyzed. At the application layer, the user needs an interface for periodic operations; in instruction set architecture (ISA) layer, resources provided by hardware should be fully utilized to provide sufficient semantic abstraction and timing precision to the upper layer. The hardware layer needs to support ISA's time attributes and time semantics, and to improve performance as much as possible while ensuring real-time performance. Research on real-time systems faces many challenges. In the process of designing and researching the existing real-time system, the key problem is that the time semantics of the upper application is difficult to keep consistent with the lower implementation.

Key words　time sensitive system; time attribute; time semantics; real-time machine; real-time system solutions

摘　要　在时间敏感的嵌入式系统中，任务需要满足其最后截止期限，错失任务期限会显著影响服务质量或带来灾难性后果. 与通用系统相比，实时系统研究进展缓慢，甚至很多基本概念都未达成共识. 精确计时 (precision timed, PRET) 机和实时处理单元 (real-time processing unit, RPU) 是 2 套现有的实时系统解决方案，以它们为例介绍实时系统相关的概念，阐述实时系统发展中遇到的问题，比较异同，并分析实时系统各个层次遇到的问题和现有的解决方法. 在应用层，用户需要定时操作的接口；在指令集架构 (instruction set architecture, ISA) 层，应充分利用硬件提供的资源，向上层提供足够的时间语义抽象和计时精度；硬件层需要支持 ISA 的时间属性和时间语义，并在保证实时性的基础上尽可能地提高性能. 对实时系统的研

收稿日期：2022-08-22；**修回日期**：2023-04-04
基金项目：国家重点研发计划项目 (2017YFA0700900, 2017YFA0700903)；国家自然科学基金项目 (62102383, 61976200, 62172380)；江苏省自然科学基金项目 (BK20210123)；中国科学院青年创新促进会项目 (Y2021121)；中国科学技术大学青年创新重点基金项目 (YD2150002005)
This work was supported by the National Key Research & Deveolpment Program of China(2017YFA0700900, 2017YFA0700903), the National Natural Science Foundation of China(62102383, 61976200, 62172380), the Jiangsu Provincial Natural Science Foundation(BK20210123), the Youth Innovation Promotion Association CAS(Y2021121), and the USTC Research Funds of the Double First-Class Initiative (YD2150002005).
通信作者：李曦（llxx@ustc.edu.cn）

究面临许多挑战.在现有的实时系统设计研究的过程中,关键问题在于上层应用的时间语义难以与底层实现保持一致.

关键词 时间敏感系统;时间属性;时间语义;实时机;实时系统解决方案

中图法分类号 TP302

实时嵌入式系统几乎存在于任何环境中,诸如汽车、航空电子设备和电信等领域都可以找到大量的实时嵌入式系统.在实时系统中,系统的正确性不仅取决于其逻辑行为,还取决于执行计算的时间.

任务对时间的敏感程度不同,可区分为软实时(soft real-time, SRT)任务和硬实时(hard real-time, HRT)任务.在软实时系统和硬实时系统中,应用通常实现为与时间约束相关的实时任务的集合,并根据选定的调度算法调度.但是,在硬实时应用(如紧急救援行动)中,违背时间限制或错失最后期限可能会造成不可估量的灾难性损失;而在软实时应用中,错失最后期限只会导致系统提供的服务质量(quality of service, QoS)下降.因此,为了提供正确的行为和良好的QoS,实时嵌入式应用程序的设计必须始终满足其最后期限.

对实时系统的研究与对通用计算机系统的研究几乎同时开始[1],但与通用系统基于图灵完备性迅速发展相反的是,实时系统研究发展缓慢.主流通用计算到目前为止还没有充分认识到与时间可预测性相关的挑战[2].在通用计算快速发展的影响下,实时系统的研究者往往直接采用通用计算机系统各个层次(体系结构、编译器、编程语言等)的技术来完成设计.在通用系统中设计可预测的分析是具有挑战性的,因为时间需求在系统层次结构中向下传递,意味着必须预见系统所有部分的时间属性:处理器和指令集架构、语言和编译器支持、软件设计、运行时系统和调度、通信基础设施等.随着现代处理器性能提高的趋势,为了提高通用计算系统的性能、生产力和提高程序吞吐量或执行效率,引入的体系结构元素,如流水线、无序执行、片上内存系统等,导致了系统执行中很大程度的不确定性,这些因素都让提供时间保证变得更加困难.现代通用处理器引入了大量的体系结构优化和多个软件抽象层,以时序的巨大可变性为代价来提供高性能的系统.如果保证最后期限是一个关键目标,时间的不可预测性就使部署这些系统变得非常麻烦.以此方法设计的系统的时序软硬件紧密相关,无法进行层次化设计,部署前需要进行大量的验证和测试工作,且一点微小的改动或升级都将影响系统的安全性.

学界对实时系统的许多基础概念、定义都没有共识.例如在对实时系统的理解上,Roop[3]认为实时系统即为反应式系统,实时性体现为系统确定的响应能力,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,并提出基于C语言扩展的PRET-C同步语言,及支持PRET-C同步语义的微体系结构ARPRET[4];Lee等人[5]认为实时性体现在操作的定时执行上,提出了精确计时(precision timed, PRET)机并在ISA层面扩展了时间操作指令,以DU(delay until)原语控制过早问题,并采用EE(exception on expired)原语通过异常处理而响应过晚错误.

典型的嵌入式系统结构如图1所示.最上层为实现嵌入式系统功能的应用程序,由实时操作系统(RTOS)等软件层应用提供实时性支持;中间层位于软硬件之间,为软件层提供硬件抽象;硬件层包括嵌入式微处理器、I/O、通用接口、人机交互接口等.

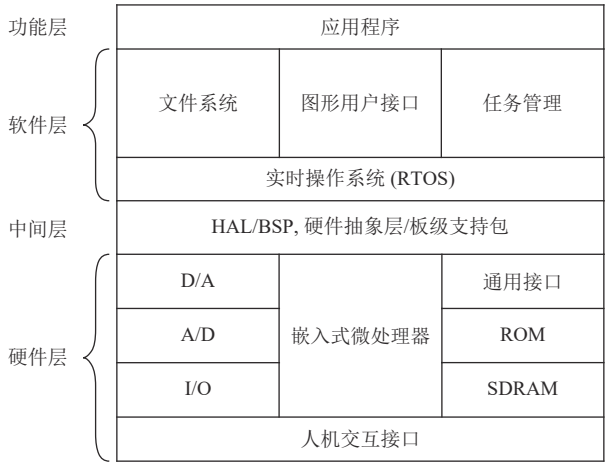


Fig. 1 Architecture of real-time embedded system

图1 实时嵌入式系统架构

实时应用程序提出对时间属性、时间语义的要求.我们可以在软件层、中间层或硬件层支持这些需求.表1列出了在不同层次实现时间语义的对比.

图2展示了实时系统各个方面的研究工作.传统实时系统设计方法存在脆弱性、冗余和平台依赖3个问题(将在第2节讨论).实时领域的研究工作大多针对单一软硬件平台,研究成果难以整合,导致对实

Table 1 Comparison of Time Semantics Supported by Each Layer of Embedded System

表 1 嵌入式系统各层支持时间语义对比

特性	软件层	中间层	硬件层
应用程序移植难度	较低	一般	较高
支持的时间精度	较低，一般比硬件高 2，3 个数量级	较高，与硬件相关	最高，与硬件晶振时钟同数量级
设计灵活度	受中间层提供的接口限制	受硬件结构限制	自由
兼顾性能	有挑战性	较易	最易

时系统的研究在工业界与学术界脱钩，不同研究者也难以复现彼此工作，因此难以深刻认识各自工作的价值. 实时系统的研究相较于其他领域需要更深的积累、更长的周期，需要自行搭建自编程语言到硬件的研究平台.

更有甚者，由于实时系统面临安全关键的问题，而软硬件平台又没有很好的抽象，导致系统即使需要微小的升级，也需要重新开始整个设计周期，导致大量的人力物力浪费.

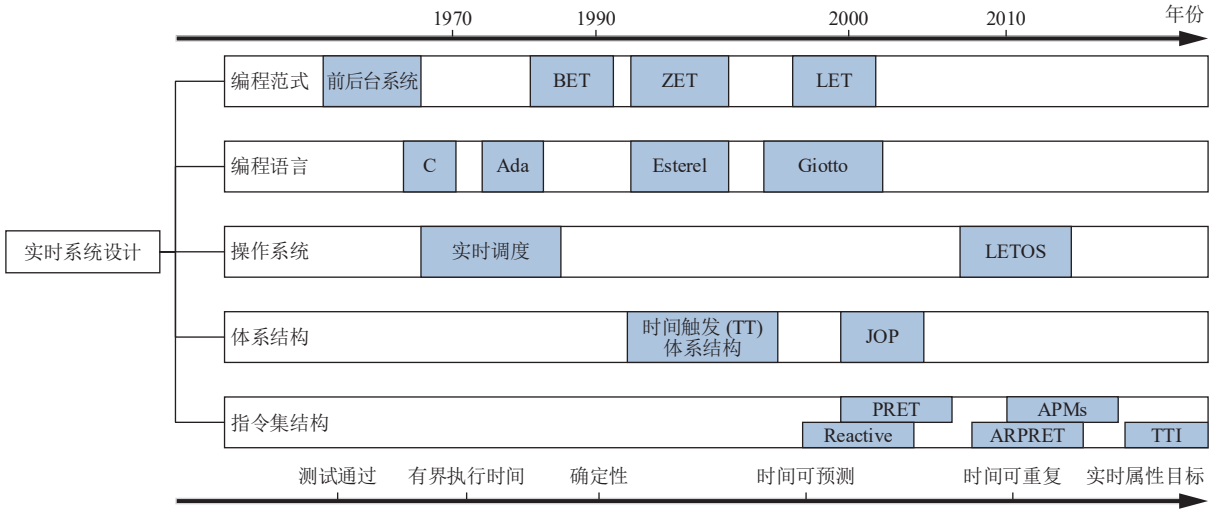


Fig. 2 Research work on various aspects of real-time systems

图 2 实时系统各方面的研究工作

本文介绍了实时系统在系统结构方面发展遇到的主要问题，着重比较了 PRET 机[5] 和我们之前提出的实时处理单元(real-time processing unit, RPU)[6] 及在其基础上扩展的实时计算系统(real-time system, RTS). 本文的主要内容包括 3 方面：

1) 简要介绍了实时系统的相关概念、需求、发展现状以及挑战.

2) 实时系统在应用层、指令集架构(instruction set architecture, ISA) 层、硬件层的需求，遇到的问题以及现有的解决方法，并着重比较了 PRET 和 RPU 这 2 套解决方案.

3) 实时系统的未来发展方向与主要挑战.

1 研究背景与问题描述

本节阐述实时计算的相关概念、发展中遇到的问题并简述现有的解决思路.

1.1 实时计算与时间属性

实时计算系统的设计是为了满足实时的最后期限要求.实时系统对时间敏感，一般应用在特定的应用领域，如航空设备、汽车部件和宇航领域等.

对实时系统的研究几乎与对通用计算系统的研究同时开始. 早在 1947 年，MIT 在 Whirlwind[1] 中引入了当时先进的实时系统概念，用于控制飞行模拟器. 与基于图灵完备性理论和冯·诺依曼体系结构定义而迅速发展的通用计算系统相比，对实时系统的研究却进展缓慢. 目前学术界和工业界对于实时系统最基础的概念、定义都还未有规范统一的定论. 例如“什么是实时？”都还未有一致的共识. 一方面，实时系统属于多学科交叉领域，来自不同领域的研究者很自然地对同样的系统有不同的观察视角. 另一方面，实时系统囊括内容广泛，常有人用其表示某一类具体的实时系统. 因此，在不同的上下文中，实时系统常有不同的含义. 1988 年，Stankovic[7] 指出，很多人认为实时计算仅仅是指对速度要求快一点的计算. 而直到 2018 年，Lee[8] 仍然强调，在实践中，当工程师谈到“实时”时，他们可能会想表达：1)高速计算；2)优先级调度；3)流数据上的实时计算；4)有界执行时间；5)程序的时间语义；6)网络的时间语义.

一般情况下，研究者从系统的逻辑正确性和时

间正确性这2个方面定义实时系统[5,9-10]，实时系统定义为系统的正确性不仅取决于系统产生的结果(逻辑正确性)，而且取决于这些结果产生的时间(时间正确性).

本文认为，实时性体现在计算系统的多个方面，从软件应用层到ISA，再到底层硬件，实时性无处不在. 为了清楚地说明什么是实时性，必须对这一概念进行详细地描述与定义. 因此，本文从软件和硬件2个方面描述当前学术界对于实时性的研究背景.

在软件方面，实时性体现在实时计算程序的时间属性上，Lee[8]总结了多种对实时性的定义：快速计算、优先调度、对流数据的计算、有界执行时间、程序的时间语义等. 本文认为，根据实时计算的应用场景，实时性是一种来自软件应用层的对时间属性的需求，这种需求即为应用程序的时间语义.

硬件上对实时性的概念也来自于应用软件对时间语义的需求. 因为超标量等技术在处理器平均性能上的过度开发，单独一条指令的执行时间变得难以预测，从而硬件难以表达来自程序的时间语义. 鉴于此，硬件上对实时性的描述聚焦于对硬件执行程序时间的可预测性.

值得注意的是，“可预测性”是一个很容易混淆的概念：一方面，它可以代表对一个硬件的时间属性分析(如最坏情况执行时间(worst case execution time, WCET)分析)结果的准确程度(即理论分析得到的执行时间界限与实际执行时间界限的差距). 另一方面，“可预测性”也可以代表着程序实际执行时间的变化程度，相对应地，越小的变化程度对应着越好的可预测性. 这2种概念在学术界均被广泛地采用，如Thiele等人[11]和Kirner等人[12]分别使用了形式化的方法来描述这2种时间可预测性.

1.2 实时计算系统发展面临的问题

随着实时嵌入式应用程序对处理能力的要求更高，多核架构，如对称多处理(SMP)，也被用于实时嵌入式系统领域. 因为多核平台的低成本以及其特性的发展和集成便于满足实时计算系统的需求. 例如，在汽车环境中，“自动紧急断开”和“夜视辅助”等新的安全功能必须读取和融合来自传感器的数据，处理视频流，并在实时约束条件下检测到道路上的障碍物时发出警告. 这是一个典型的场景，对高级特性的需求不断增加，导致对额外计算能力的需求成比例地增加. 此外，系统功能的增加决定了电力消耗、散热和空间占用(如布线)方面的额外成本[13]. 因此，多核处理器是降低上述成本的一种经济有效的解决方案，因为额外的计算需求可以分配到单个处理单元，而不是分散在车辆上的多个处理单元.

然而，在不断引入新技术提高性能的同时，处理器等硬件设备的执行时间逐渐地缺少时间可预测性，抑或是变得难以分析. 这导致现代实时计算系统开发往往需要耗费大量的人力物力在硬件的“验证—纠错—重新设计”流程上. 因为硬件设计上一个微小的变化经常会导致时序的巨大改变，为了保证设计出的硬件系统能够满足实时系统的时间与要求，需要通过对硬件的时序进行严格的建模与分析，但这一过程的代价是巨大的. 为了解决这一困境，提高生产效率，业界也提出了许多解决方案，如：采用有大量冗余性能的硬件、使用传统的简单而易于验证的设计、执行时间分析、实时操作系统等. 本文接下来会简要介绍其中几种解决方案并将它们进行对比，着重介绍现代工业界常用的WCET分析方法，进一步指出导致实时系统设计困难的关键问题.

1.3 现有的解决思路及其优劣

一种最简单的平衡实时计算系统对实时性和性能需求的方式就是采用高性能的硬件，但是这样做的效果并不好，原因很简单：这样做大大增加了生产所需的成本，即使采用高性能硬件，也无法验证程序运行时所需的时间语义是否可以满足，因为传统高性能硬件从未考虑过程序的时间语义.

另一个极端就是使用传统且简单的硬件设计，因为它们的硬件结构足够简单，在其基础上的验证与分析变得非常容易. 然而，这样做只能满足一些小型工业设备的生产设计要求，对于同时需要高性能与时间语义的复杂实时计算系统，过于简单的硬件无法胜任.

执行时间分析是一种相较而言成熟的解决方案，一般来说，实时系统满足其最后期限的能力是通过可调度性分析来验证的[14]. 所有此类可调度性分析技术都有一个基本的假设，即每个任务的WCET的上限是已知的. 然而，在任务WCET上导出安全而严格的边界正变得越来越困难. 在多核架构上尤其如此，因为处理器或核心共享硬件资源，例如缓存内存层次结构、总线、DRAM和I/O外设. 因此，由一个处理单元执行的操作可能导致在任何共享资源级别上不受管制的争用，从而不可预测地延迟在不同核心上运行任务的执行.

实时系统缓存管理方案的主要目标是简化任务的WCET估计. WCET估计通常遵循2种主要方法之一：静态分析或经验测量. 在静态分析方法中，分析

工具试图通过分析应用程序二进制代码来提供WCET, 而不直接在硬件上执行它. 一般来说, WCET估计工具只适用于简单的处理器, 这是由于复杂的硬件特性, 如缓存、分支预测器和流水线等, 使得静态分析极其困难或过于悲观[15]. 在后一种方法中, 应用程序二进制代码在硬件平台上执行多次, 并从这些执行中提取具有可调置信值的 WCET 估计. 此外, 在基于度量的方法中, 为了考虑可能由于延迟执行的未观察到的条件, WCET 的结果值通常会进一步增加一个误差范围(观测值的 20%～30%). 因此, 基于度量的方法也可能导致高估 WCET. 缓存的使用使得静态和基于测量的 WCET 估计更加复杂, 因为指令的执行时间可能会根据:

1) 数据/指令在内存层次结构中的位置;

2) 存储器的存取导致任何缓存层的缺失或命中;

3) 正在使用的缓存一致性协议;

4) 缓存控制器上实现的缓存替换策略而变化[16].

虽然已经有人为单核系统提出了多种实时缓存分析框架[15], 但目前的静态分析方法对共享缓存的失效提供了悲观的上界. 此外, 据我们所知, 没有现有的静态分析技术能够解释一致性协议的影响. 总而言之, 硬件设计上的复杂性会导致 WCET 分析结果不紧致, 从而根据 WCET 分析得出的硬件性能需求往往会比实际高得多.

在软件层面, RTOS 提供计时功能, 但 RTOS 提供的计时行为不精确, 且在复杂系统中不可控. 软件抽象时间属性将在计时精度上损失若干个数量级. 硬件的中断行为一般需要纳秒级的计时精度, 而软件实现在用户级可能只能达到毫秒级, 对于部分实时程序, 这一量级远远超出了可接受的范围.

目前实时系统的挑战在于兼顾性能和实时性. 如果愿意放弃性能, 那么精确的计时可以很容易实现. 相反地, 如果注重性能, 那么很难在现有的高性能设计基础上继续保证硬件有良好的时间可预测性. 现代处理器架构使 WCET 分析变得相当困难, 即使是分析简单的程序也需要付出巨大的努力. 从根本上说, 通用处理器的中间层 ISA 和硬件层未能提供足够的抽象, 即计算机体系结构的 ISA 层缺乏对时间语义的描述, 硬件层缺少对时间语义的支持.

1.4 基于 ISA 层引入时间语义的解决方案

相较于在软件层面实现实时性产生的巨大开销和挑战, 以及使用时间可预测、可重复的硬件带来的设计、应用成本花费, 在中间层 ISA 引入时间属性、设计时间语义是有潜力的研究方向.

针对 ISA 层缺乏时间语义的问题, 目前已经有若干解决方案. 通常, 这类工作依赖于系统中子组件之间的强制时间隔离: 1 个子组件的执行时间不应该依赖于其他子组件的行为.这种隔离可以:

1)在单个任务级别实现, 其中子组件是任务代码中的功能或基本块;

2)在核心一级, 其子组成部分是任务;

3)在多核系统级别, 子组件是单独的核心或运行在每个核心上的软件分区.

具体工作将在本文第 3 节阐述.目前现有的解决方案主要有 2007 年始由 Lee 等人[5] 提出精确计时机 PRET, 和我们实验室提出的实时处理单元 RPU 及在其基础上扩展的实时计算系统 RTS.本文着重介绍与比较 PRET 和 RPU 这 2 种解决方案, 它们均在 ISA 层引入时间语义来尝试解决性能和实时性之间产生的冲突.

1.4.1 精确计时机 PRET

来自 UC Berkley 的 Lee 等人提出的精确时序机器 PRET 共包含 3 个版本. 其中, 第 1 个版本仅仅是一个基于 SPARC 的处理器模型[17], 该模型用来在前期论证硬件多线程、scratchpad memory 以及对主存的调度访问在 PRET 中实现的可行性. 第 2 个版本, 又称之为 PTARM[18], 是一个在 FPGA 上实现的基于 ARM 指令集扩展的硬件多线程处理器, 对该处理器的测试表明, 其在应用程序中具有足够的并发性, 实现可重复的时序行为并不需要以过多的性能为代价. 在该版处理器的实现中, Reineke 等人[19] 同时证明了 DRAM 存储的访问延迟的不确定性可以通过按照 DRAM 设备的内部结构划分物理地址空间, 并交错访问分区中的块来解决. 第 3 代 PRET 机器, 又名 FlexPRET[20], 是一个基于开源 RISC-V 指令集的硬件多线程处理器. 相较于第 2 版, FlexPRET 的特别之处在于它的硬件线程数是可变的. FlexPRET 既可以只配置一个硬件线程, 像一个普通的 RISC-V 处理器一样执行, 也可以配置数个硬实时线程来保证任务执行时有确定的时序. 除此之外, FlexPRET 还支持混合优先级任务调度, 通过调度硬实时线程以及软实时线程, FlexPRET 可以最大化利用处理器的资源.

所有 PRET 机器都是细粒度多线程处理器, 其中 PTARM 和 FlexPRET 均使用多线程在流水线中交错执行的方式来实现线程之间的硬件隔离, 从而每个线程的执行时序都独立于其他线程而不会受影响. 为了精确表达与控制任务的时间属性, PRET 机器都在各自的指令集上进行了指令扩展, 在指令集架构

层引入时间语义,实现从软件到硬件的时间语义表达的一致性这样一种自上而下的解决方案.

1.4.2 实时处理单元 RPU

RPU[6]是中国科学技术大学高能效智能计算实验室所提出的一个实时硬件多线程处理器项目.其中第 1 版 RPU(RPU1.0)是一个基于 RISC-V 整形指令集的粗粒度硬件多线程处理器,并额外支持时间语义指令集(time triggered instruction, TTI)扩展.RPU1.0 通过粗粒度多线程切换的方式隐藏处理器运行时的长时延问题(如 DRAM 访存等),从而提高处理器资源利用率,间接地提高性能.除此之外,RPU1.0 针对 LET 应用模型的要求,通过 TTI 中的线程控制指令来严格控制线程的执行时间,并通过定时 I/O 指令来实现精确的输入输出控制.

RPU2.0 是在 RPU1.0 基础之上实现的一个细粒度硬件多线程处理器,与 RPU1.0 不同的是,RPU2.0 为每个硬件线程都配置了一套寄存器、PC、流水段间寄存器等部件,从而实现每个硬件线程之间的隔离以及线程之间的无缝切换.通过这种方式,RPU2.0 可以独立地对每个线程进行时序分析,减少线程之间的干扰.线程调度采用可动态配置的循环调度方式,每个线程可以自由配置执行时长等参数,处理器严格按照调度执行各个线程,从而保证线程执行的时序,在每个线程内,线程还可以使用 TTI 来实现精确定时的 I/O 操作.

2 应用、软件层级的实时性

本节介绍实时应用在通用的传统体系结构上实现的方法技术.

2.1 应用的时间需求

实时计算系统所面向的应用场景相当广泛,从航空航天到汽车控制系统、医用设备,以及高精度工业控制设备方面都有广泛的应用.这些应用都需要实时地与外界物理环境进行交互,这不仅需要实时计算系统的操作逻辑正确,还需要其操作能够满足一定的时间约束,即应用的时间语义.显然,为了尽可能地满足应用的时间需求,需要明确在这些应用场景下有哪些时间语义需要表达.

在响应式系统(reactive systems)[21]中,系统需要在有限的时间内根据外界的输入进行快速计算并输出结果.这类系统在实际生产中的常见应用往往是一些安全攸关(safety critical)的应用,如汽车刹车系统、医疗电子控制系统、核电站控制系统等.这些系统需要保证程序能够在给定的时间内完成执行,否则会带来灾难性的后果.从这点来说,它们的时间需求聚焦于程序的 WCET 是可分析的,且往往要求 WCET 的变化范围小于某一特定的值.

在交互式系统(interactive system)[22]中,系统以一定的频率和外界环境进行交互(通常按照环境的步调持续地与其进行交互),这一交互往往需要和其他组件同步(synchronize).比如在时分复用的实时通信系统中,计算结果的输入和输出需要和通信时对应的时间片同步.在这种情况下,系统的时间需求注重系统输入/输出的时刻需要和外界精确同步,更进一步地来说,这一过程可能还具有周期性.

此外,在现代的工业生产设计中,一种系统可能具有多种不同优先级的任务,即混合关键系统(mixed criticality systems, MCS)[22],在这种系统中,有着不同优先级的任务在同一个硬件平台上共享执行资源.在实时系统中,高优先级任务往往有较强的时间语义,低优先级任务可能有较弱甚至不具备时间语义.在这种情况下,实时系统往往需要进行合理的调度来保证高优先级任务的时间语义被优先保证.

2.2 实时操作系统

实时应用程序在软件层面通用的解决方法是在传统结构上运行实时操作系统(real time operating system, RTOS).传统计算机系统的层次结构中,体系结构层次以“指令”作为最基础的单元,而操作系统通过外部中断的协助对“指令”流进行管理和封装,为上层提供丰富的服务.在实时领域的研究过程中,出现一批实时操作系统,如 μCOS[23], FreeRTOS[24], RTEMS[25]等.它们基于传统的操作系统和实时调度算法,提供具有一定时间语义的任务模型接口.然而,在这种方式下,操作系统更像是编程语言配套的实时库,既要保证实时应用的时间属性,仍然需要用户结合实时应用和体系结构进行设计和调整.这些“实时操作系统”仍未突破传统计算机系统的层次结构,其实际效果也具有一定的局限性.

在时间精度方面,实时操作系统基于内核服务提供时间延迟和超时.如 FreeRTOS 用 32 位寄存器保存定时器中断的计数值,在常见的嵌入式设备(例如 STM32F429 开发板)上时钟精度约为 50μs,比硬件运行的纳秒级别的时钟周期高约 3 个数量级.另一方面,RTOS 按预定的调度策略调度任务执行.RTOS 通常支持优先级抢占或时间片轮转(round-robin, RR)这 2 种基本任务调度机制,由于在软件层面切换任务带来的大量开销,RTOS 提供的快速任务抢占耗时一般

达到毫秒级(相较于通用系统的非抢占内核,最坏情况为秒级)[26].

在提供的服务方面,典型的 RTOS 采用微内核(micro kernel)体系结构[27],其内核只提供最基本的系统服务(如任务调度和管理),其他服务位于内核外,需要根据应用需求单独裁剪配置;且由于状态切换带来的大量开销,对时间敏感的嵌入式系统可能不区分用户态和内核态,应用程序之间的隔离性较弱.

2.3 最坏情况执行时间(WCET)分析

WCET 分析始终是一项富有挑战性的工作[15].因为程序流分析是一个不可判定问题,需要描述清楚,对一个在某个系统上运行的程序进行 WCET 分析,需要对这个系统的每个细节,包括流水线、存储系统以及执行结构进行建模.现有的先进 WCET 工具有 AbsInt 的 aiT 工具,该工具曾被用在空客 A380 的安全攸关的软件系统的分析中[17]. 除此之外,还有 Otawa 以及 Chronos 等工具.

此外,说明 WCET 工具的有效性也是件困难的工作,Wägemann 等人[28] 提出 GenE 工具来评估 WCET 分析器的有效性,GenE 可以生成一系列已知流图的程序测试用例,从而通过对比 WCET 分析工具在其上的分析结果与已知流图的分析结果可以判断 WCET 分析结果的有效性. 令人遗憾的是,GenE 发现了 aiT 工具在 ARM Cortex-M4 处理器的建模下分析出的 WCET 并不正确[17]: aiT 分析出的 WCET 结果小于程序的实际执行时间. 这说明针对复杂的系统设计,建模必须格外小心,否则 WCET 结果难以和实际的系统设计保持一致. 更进一步地,正如 Wägemann 等人[28] 提到的:“最精确的执行时间模型就是处理器设计本身”,因为建模的过程不可避免地会发生错误.

WCET 分析的经验表明单单依靠分析工具来保证实时性并不可靠. 针对这种困境,目前有一种观点认为应当通过在程序执行的更低层次引入时间语义来提供程序执行时间的抽象表述,从而简化 WCET 分析,保证应用的实时性.

2.4 编译器保证的代码段执行时间上界

传统编译器只聚焦于提高代码的平均性能,且无需知晓程序执行的硬件架构是怎么样的. 在实时程序的设计中,这 2 点都发生了很大的改变. 编译器层面变为着重于对程序 WCET 的优化[29],且这一过程是架构相关的,即编译器至少需要底层硬件执行指令时的时序相关的信息才能进行有效的 WCET 分析.

Broman 等人[30] 在 PRET 的设计过程中为这些问题提出了一种解决方案. 针对编译器过于依赖底层硬件实现的问题,Broman 等人[30] 认为,编译器对于底层硬件的信息需求是必不可少的,但是可以通过参数化描述底层硬件架构的方式来增强编译出的目标代码的可移植性. 因此,可以使用体系结构描述语言(architecture description language, ADL),比如 EXPRESSION[31],来参数化描述硬件架构,从而为编译器增加对硬件的时间行为的描述,减少对硬件架构的依赖,增强目标代码的可移植性.

编译器为了保证代码执行时间的上界,除了需要硬件架构的时序信息之外,还需要对程序本身进行 WCET 分析,PRET 小组使用 MTFD 指令来对程序块的执行时间进行约束[18],并将这一约束信息传递给 PRET 编译器. 编译器通过对程序中被 MTFD 约束的代码块进行 WCET 分析,来判断约束是否被满足.

PRET 小组在设计的过程中还提到了编译器的一个额外任务,即程序内存空间分配的工作. 在 PRET 机器中,实时程序的代码和数据被放在一块名为便签存储器(scratchpad memory, SPM)[32] 的部件中执行,SPM 可以快速地响应数据请求且其响应时间是可预测的,因为实时程序不能忍受访问如 DDR 主存时的抖动巨大的响应时间. 但 SPM 必须显式管理、手动分配,这就需要编译器知晓 SPM 在内存中的位置,并主动将实时程序的代码与数据分配到 SPM 中.

总而言之,实时程序的编译器相较于传统的编译器(如 C 编译器),承担了许多额外的工作来保证代码段执行时的时间属性,包括 WCET 分析、硬件时序分析、内存空间管理等,对支持实时程序的编译器的开发仍将会是一项富有挑战性的工作.

2.5 本节小结

传统的实时系统设计方法基于通用计算系统,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.

很久以来,程序的“正确”执行从未考虑时间. 计算的所有抽象层次都未曾提供时间属性. 指令集体系结构不能保证时间属性;没有通用的语言有时间语义的表示;实时操作系统由于并发等因素不能保证时间语义属性;现有的网络架构也没有时间语义属性[5]. 这导致实时系统设计方法始终无法摆脱 3 个问题:

1)脆弱性. 表现为在实时系统中即使是一个微小的改变或“改进”也可能会导致严重的时序紊乱,需要重新验证实时性是否得到保证,甚至重新设计.

2)冗余. 体现在实时系统的设计往往需要预留

多余的时间或计算资源以保证系统的正确性.

3)平台依赖. 体现在实时系统设计依赖其使用的软硬件平台、系统时序与平台绑定.

传统的实时系统设计方法往往通过投入大量的人力物力来解决这些问题. 然而，当今社会的实时需求日益复杂，对敏捷设计的要求也在逐步凸显，实时系统急需新的高可用、高易用的设计方法.

3 ISA 级的系统实时性

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，指令一条接一条地执行，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的时间约束.

由于传统的计算机体系结构模型缺乏时间语义，早期的实时系统设计者不得不把具有时间语义的程序运行在不具有时间语义的体系结构上. 莱斯定理[33]指出，程序所有非平凡的语义属性都是不确定的. 这意味着在基于缺乏时间语义的体系结构模型设计的处理器上，难以确定性地运行时间语义程序. 只有在体系结构模型中加入时间语义，结合具有时间语义的指令集设计实时处理器，才能保证时间关键应用的运行时确定性(PRET 研究组将这个特性称为时间可重复性).

3.1 将时间作为关键因素编程

虽然一些实时系统建模语言(Timed C 等)中包含了时间概念，但实现语言，如 C 语言、通用 RiscV 汇编语言等缺乏时态语义.因此，程序的执行时间受其自身特性和软硬件执行环境的影响，分析复杂、脆弱易变.时序仅仅是特定硬件平台上软件实现的一种附带效果.硬实时程序测试、验证和认证时需要仔细考虑软硬件交互的细微之处，但任何微小的软硬件设计变化都可能导致不可预测的时序变化，成本高昂.实时系统的建模和实现既没有可移植性，也无法保证在实际系统中执行的正确性.在系统的设计阶段，设计者对所设计系统的时序行为进行了仔细定义和分析，但现有的实现技术基本上忽略了这些工作，迫使设计者通过测试对所实现系统的时序行为进行重新确认.

时序指令仅仅建立一种指定时序约束的方法，并不强制执行时间行为，它们仅仅用于对软件中的时序变量进行监测、检查和交互. PRET 和 RPU 的目标都是在 ISA 中引入时序语义，但不过度约束 ISA 的时态属性. 时序指令只需要满足其时序属性完全实现，不限制其他指令的性能优化. 基于这些时序指令，程序员可以分析和控制程序的时态属性，独立于体系结构. 同时，时序指令自身并不保证程序的执行时间. 静态分析为了可以得到紧致的 WCET 界，需要下层体系结构提供可预测的执行时间.

目前，实时 ISA 的研究工作主要有 2 个方向：

1) 基于传统的 ISA，以辅助 WCET 分析为目标设计时间可预测的体系结构.

2) 定义带有时间语义的 ISA，为上层提供精确的时间语义，并为目标设计时间可预测的体系结构.

3.2 基于传统 ISA 的工作

传统 ISA 缺乏时间语义，Kopetz 最早提出采用时间触发(time-triggered, TT)范式代替事件触发(event-triggered, ET)范式进行体系结构的设计，并采用时间触发体系结构(time-triggered architecture, TTA)以解决分布式实时系统中多节点间缺乏时间语义的问题[34-36]. 随后，越来越多的学者认识到处理器缺乏时间语义的问题，并在传统 ISA 上改进设计时间可预测的体系结构.

Table 2 Comparison of ET System and TT System

表 2 ET 系统和 TT 系统的比较

性质	ET 系统	TT 系统
可预测性	很难满足时间约束	可满足时间约束
可测试性	尽量进行覆盖测试	可进行构造测试
资源利用率	负载低时较优	极端情况较优
可扩展性	易改造，无法保证时序	重新设计，保证时序

Pont[37] 认为了解硬件的每一个细节，就可以基于现代处理器构建满足所有实时要求的软件，这对硬件、系统设计者提出了较高的要求. Pont 研究团队提出了时间可预测的 PH Core[38]，它使用硬件多线程来处理由中断引起的延迟，并且有一个硬件调度器以更好地支持 TT 模式.

Schoeberl[39] 希望使用 Java 进行实时系统的开发，并为 Java 环境设计了一套硬件体系结构——JOP (Java optimized processor)，从架构的角度增加系统的可预测性. JOP 架构中更简单、更准确的 WCET 分析比平均情况性能更重要. JOP 是 Java 虚拟机在硬件上的实现，它采用了 3 级流水线，没有使用分支预测. JOP 主要应用于需要 Java 环境的嵌入式实时系统，其微指令长度一定且执行时间相同.

我们认为基于传统的 ISA 设计可预测的体系结

构没有明确在ISA层抽象出时间触发、时间控制的概念，复杂度高、泛用性较低.

3.3 重定义ISA的相关工作及时间语义

ISA层在实时系统中对上层提供时间接口、抽象时间属性和时间语义(对时间相关操作的表达能力)，对下层硬件提出需要满足的抽象要求.

Roop[3]对支持同步语义的反应式处理器进行研究，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；以Esterel[40]为代表的同步编程语言基于同步假设，通过信号管理，如轮询(polling)、发射(emission)和抢占(preemption)等同步程序，保证并发行为的确定性，易于程序验证；HiDRA[41]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[42]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[43]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等人[4]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. ARPRET分为2部分，GPP实现通用计算，PFU负责调度和保证功能的时间属性正确.

据本文调查所知，目前仅PRET项目和RPU完整地抽象了时间语义，向上层提供了ISA级的时间接口，扩展了时间语义指令集.

3.3.1 PRET基于delay的延时语义

PRET扩展的时间语义主要约束了程序段执行时间长度. DU(delay until)指令保证了程序段的执行时间下界，mtfd结构在编译阶段静态检查代码段是否满足设定的执行时间上界，并采用EE(exception on expired)指令通过异常处理响应过晚错误. PRET基于代码块进行时序控制，我们认为PRET在功能上，逻辑与时序控制分离；在时序上，“过早”与“过晚”分离.

此方案添加时序语义，可以约束指令序列执行时间的上下界，然而其缺少对定时语义的支持，无法直接表达操作发生在某个时刻的语义. 正如Zimmer等人[20]阐述的那样，如果系统中存在中断，PRET基于delay+load/store的结构实现定时I/O就存在实现与语义不一致的问题，操作缺少原子性.

Wan等人[44]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，但该想法处于较早期阶段.

3.3.2 时间触发指令集TTI的语义

TTI指令集所描述的时间语义主要包括2个方面：1)对程序执行时间长度的约束；2)对程序执行操作时刻的约束. 在对程序执行时间长度的约束方面，TTI并没有特别定义针对某个程序设置执行时长的指令，对程序执行时长的控制体现在TTI的多线程指令方面. TTI中有一个线程调度表，该表负责为每个线程分配执行的时间片大小. 通过对多线程的调度，可以控制每个线程在处理器中运行的时间长度.

值得注意的是，相较于PRET, TTI对程序执行时间的控制方式是不一样的. 在执行超时的情况下，PRET机器通过超时触发定时中断的方式实现对程序执行时间长度的控制. 在执行时间短于规定时间的情况下，PRET机器通过DU指令来延长程序的执行时间. 而TTI在这2种情况下都使用线程切换的方式来实现，在程序超时后，时间语义指令将异常状态代码写入特定的控制与状态寄存器(control and status register, CSR)中，并切换至其他线程，原有线程的程序被中止执行. 而在一个线程被分配到的执行时间片内，即使程序已经完成了执行，处理器仍只会运行该线程而不会跳转至其他线程，直到该线程被分配的时间片用完.

针对同步式实时控制系统的需求，TTI还支持针对I/O的定时语义，TTI可以将程序的I/O与执行时刻绑定，在处理器运行的离散时间中指定某一时刻执行操作.

3.3.3 时间指令集扩展及对比

为实现ISA定义的时间语义抽象，需要向通用指令集中扩展时间操作指令. PRET系列机和TTI的时间扩展指令集对比如表3所示.

PRET和TTI时间扩展指令集对于过早的操作都采取等待的方式，对超时错误抛出异常. 不同的扩展时间指令集对实时性的影响差异体现在指令集可描述的时间语义上，如3.3.1节和3.3.2节所述. 在时钟精度方面，PRET的时钟精度由微处理器运行的周期长度决定，而TTI基于时间触发范式TT[34]和时间触发自动机TTA[34-36]设计，完整地抽象了时间触发、时间控制的概念，时钟可以由外部接入(例如网络时间)，支持的时间精度与微处理器运行的时钟周期同数量级[6]. TTI针对RISC结构的微处理器使用的load/store结构，定义了定时I/O指令ttiao/ttoat，原子性地保证了I/O操作的时间点(time point)[26]，符合LET编程模型[45]的规范.

RPU实现TTI指令集，基于LET编程模型.目前在Xilnx FPGA开发板(xc7a100t-csg324-1)上实现了

Table 3 Comparison of PRET and TTI on the Extended Time Instruction Set

表 3 PRET 与 TTI 在扩展的时间指令集的对比

类型	PRET	解释	TTI	解释
时间管理指令	set time *%r*, *offset*	*%r* = 当前时间+*offset*	getti *r*1	*r*1 = 当前时间
			setti *r*1	当前时间 = *r*1
			settg *r*1	标准时钟粒度 = *r*1
			getts *rd*	*rd* = 上次时间触发操作的时间戳
			（TTI 2.0） settsg *r*1	时间片长度 = *r*1 指定的 CPU cycle 数
			（TTI 2.0） gettsg *rd*	*rd* = 当前时间片长度
时间触发操作指令	delay until *%r*	延迟后续指令，直至 *%r* 时刻开始执行	delay *r*1	延迟后续指令直至 *r*1 指定时刻开始执行
			ttiat *rd*，*r*2，*r*1	if（*ti*==*r*1） then *rd*←[*r*2]；else wait.
			ttoat *r*3，*r*2，*r*1	if（*ti*==*r*1） then [*r*2]←*r*3；else wait.
异常管理指令	expection on expire *%r*，*id*	当前时间等于 [*%r*] + 1，则异常		
	deactivate exception *id*	禁止 *id* 异常		
静态检查执行时间上界约束指令	mtfd *%r*	保证该指令执行时，当前时间≤*%r*	mtfd *r*1	保证该指令执行时，当前时间≤*r*1
任务并发管理指令			tkend	当前线程的任务执行完毕
			addtk *r*1,*r*2	添加一个调度表项，使得处理器在 *ti*=*r*1 时，切换到 *r2* 指定的线程执行

智能小车跟车系统[26]和 ABS 防抱死应用系统[46]的验证，结果符合预期.

3.4 本节小结

实时系统在中间层 ISA 实现实时性，相较于软件层实现实时性，利用了微处理器一般达到纳秒级的运行时钟，为上层直接提供时间语义指令.

目前指令集架构层面引入时间语义指令的工作主要有 PRET 研究组的系列工作[20,47-48]和 TTI 指令集. PRET 研究组提出了指令集架构在时序控制方面需要具备的能力[47]，基于这一想法，文献 [20,48] 分别基于 ARM 指令集和 RISC-V 指令集扩展，加入了控制代码段执行时间的指令，并依据扩展后的指令集构建了可以执行时间语义指令的处理器 PTARM 和 FlexPRET; 文献 [6] 在 PRET 研究组工作的基础上提出了时间和操作相结合的定时指令，定义了 TTI 指令集，并实现了 RPU. 文献 [6, 20, 48] 所述的工作，重点在于时间语义指令集的定义，以及处理器硬件架构的实现. 文献 [47] 虽然给出了指令集架构在时序控制方面需要具有的语义，但是没有说明从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.

现有的时间语义指令集工作对指令集可以表达的时间语义没有给出明确说明，同时缺乏相应的时间语义程序设计范式，导致上层控制模型或编程模型映射至底层时间语义程序时依旧缺乏相关理论指导.

4 硬件层级的实时性支持

4.1 实时系统对硬件性能的需求

现今实时计算系统面临的挑战在于同时兼顾系统的性能以及实时性，因此，对性能的需求也是必不可少的. 一个单周期处理器显然有很优秀的实时性，但是它的性能远远满足不了现代应用的需求.

一个常见的误区在于高性能等于更好的实时性，导致许多工业系统设计错误地估计了实时系统的性能需求，Stankovic[7]也曾在 1988 年提出过，高性能的硬件有助于对系统实时性的提升，但是却不能保证系统的实时性，这和实时计算系统的设计初衷是相违背的. 若使用软件来取代通常由硬件提供的功能或与其他硬件设备交互的应用程序需要精确计时的输入/输出（I/O），其所需的计时精度在纳秒级，这是处理器时钟周期的粒度. 目前，尽管对于许多应用程序来说毫秒级的计时精度已经足够了，但对于某些特定的应用程序来说还不够精确.

4.2 实现时间属性的硬件支持与计时精度

PRET[5]和 RPU[6]都基于经典 5 段流水线，在 EX 段执行扩展的时间指令，采用寄存器寄存当前计时时间. 在 PRET 系列最新的 FlexPRET 实现中，系统上电时刻计时时钟寄存器归 0，随后每个 CPU 运行时钟上升沿到来时，计时时钟值加 1. 由于最终运行频

率为 100 MHz, FlexPRET 认为其计时精度为 10 ns. 我们认为, 在 PRET 中计时时钟和运行时钟只是同一个晶振源的不同视角, 且冯·诺依曼结构并未指明硬件实现必须要使用时钟同步的各个部件, 因此 PRET 未对计时时钟和时间语义有足够的抽象.

实现 TTI[6,22] 的 RPU 提出了实时机(real-time machine, RTM)的概念, 在冯·诺依曼机 5 大组成部件的基础上显式引入计时时钟, 直接提供给上层针对程序的时间行为予以控制约束的接口, 为 RTM 赋以时间语义. RPU 基于 RTM 模型, 引入计时时钟(称为标准时钟), 明确了时间触发、时间控制的概念.

标准时钟和 CPU 运行时钟之间显然有 2 种关系: 同步与异步. 在实际应用中, 如果需要从外部接入标准时钟(例如网络时间), 则运行时钟和标准时钟之间是异步关系. 在之前的研究中, 我们说明了在异步情况下 TTI 的时间触发指令抖动小于一个 CPU cycle[6].

4.3 硬件多线程流水线结构

实时系统标准的传统方法是在单线程处理器上进行软件调度, 而不需要为每个任务使用整个处理器或核心, 并承受相关的效率损失. 调度算法选择在不同时间段执行某个任务, 因为只能有一个任务在执行. 使用传统方法已经越来越不能满足实时系统对性能日渐增长的需求, 在单核系统上引入硬件多线程可以在保证实时性的基础上以较小的开销提升处理器的吞吐量, 减少 stall 或是等待线程代码段执行时间下界的空闲周期.

我们的调查得出的一个主要结论是, 一定程度的硬件支持对于在实时系统中为并发执行任务提供可靠的隔离保证是必要的. 即使可以只用软件实现, 但与硬件解决方案相比, 软件的解决方案通常更麻烦、更难以认证且增加更多的开销.

粗粒度多线程(coarse-grain multi-threading, CGMT)描述了一种硬件多线程结构, CGMT 在同一时间只能执行 1 个线程, 但又可以使用硬件切换为执行另一个线程, 无需软件干预. 为此, 它必须在处理器中使用硬件存储多个线程的状态, 该状态称为硬件上下文(context). 切换 2 个存储在硬件上下文中的线程, 包括清空正在运行的线程状态并将另一个线程状态加载到流水线中. 与操作系统使用的软件上下文切换相比, 这通常需要少量的周期. 相应地, 软件上下文切换只能用于隐藏长延迟(例如磁盘和其他 I/O 延迟), 但粗粒度多线程处理器可以隐藏更短时间的延迟, 例如高速缓存未命中带来的延迟.

细粒度多线程(fine-grain multi-threading, FGMT)相较于粗粒度多线程, 将若干个 CPU 时钟组织成时间片, 在每个时间片按线程调度表切换线程. 线程之间交错执行, 实现了“时间触发”而不是“事件触发”切换线程[6,17,20]. PRET 实现细粒度多线程, 采用线程交错的流水线来避免同一线程中的数据依赖, 在 PTARM[18] 和 FlexPRET[20] 实现中, 每个 CPU 时钟上升沿选择一个线程取指. PTARM 中, 多线程以严格轮询的方式调度, 流水线的每一段交错地执行若干个线程. 而 2015 年提出的 FlexPRET 改进型结构, 使用细粒度多线程和灵活的调度与定时指令来实现每个任务在基于硬件的隔离和有效的处理器利用率之间进行权衡, 多个线程不必严格轮转, 每个周期可以由设定好的线程调度表选择线程取指进入流水线. 每个线程中程序相邻的 2 条指令取指之间的时间间隔(以 CPU cycle 计量)称之为一个 thread cycle. FlexPRET 同时调度 HRT 任务和 SRT 任务时, 对 HRT 线程采用时分复用(time division multiplexing, TDM)的调度, 消除线程间干扰.

RPU1.0 实现粗粒度多线程, RPU2.0 版本实现了细粒度多线程. RPU1.0 采用和预设最大线程数相等套数的段间寄存器、PC 寄存器、Regfile, 增加了一定的硬件消耗, 确保了切换线程的时序. 多线程由专用的时间触发(TT)线程控制器管理, TT 线程控制器根据 TT 表控制切换线程, TT 表的抽象结构为队列. 在 RPU1.0 版本中, TT 表记录了若干线程编号-标准时间数据对, 每当队头指定的时刻(在 EX 段判断)到来时, 花费 4 个 CPU cycle 切换至指定线程[6]. TT 表由 0 号线程维护, 通过 addtk 指令不断向 TT 表中添加表项, 0 号线程需要经过特定设计以满足调度需要. 在 RPU2.0 版本中, 多线程按时间片轮转调度, TT 表记录了各个线程编号及其分配到的时间片个数. 相较于 RPU1.0 版本, RPU2.0 损失了一定的灵活性, 但在保证可预测性的同时降低了 0 号线程的设计难度.

4.4 存储系统的实时性

存储层次结构设计影响功能、可预测性和性能, 根据应用程序领域的不同, 需要做出不同的权衡.

目前对实时存储系统的研究主要有 2 个方向:

1)在通用的传统 Cache-Bus-Memory 结构上改进以令存储系统具有时间可预测性和时间可重复性.

2)使用新的存储组织结构.

在通用结构上改进的主要挑战来自于 Cache 访问时的不确定性、总线/存储器访问调度策略等, 以及扩展至多核系统时保证时间可预测的缓存一致性;

使用新的存储组织结构能保证较好的实时性，但现有的实时应用和硬件都需要重新设计.

4.4.1 传统 Cache-Bus-Memory 结构

CPU Cache 对程序员来说是透明的，并且其依赖于内存访问的时间和空间局部性来减少应用程序的平均执行时间. 因此，CPU Cache 使用一组启发式方法来保存在不久的将来更有可能被访问的数据，并替换旧的、未引用的条目. 在较高的层次上，Cache 的启发式行为意味着在整个任务执行过程中对同一位置的内存访问可能会导致 Cache 命中，也可能不会命中，这取决于系统的历史记录. 事实上，任务本身的执行模式、同一核上的不同任务的执行模式，甚至是不同核上的一组任务的执行模式都会影响给定内存访问模式的 Cache 命中率. 这意味着复杂功能的系统历史可以直接影响任务从内存层次检索数据所需的时间，这也解释了为什么 Cache 是不可预测性的主要来源之一. 此外，实时应用程序的内存访问行为是由其功能驱动的. 例如，简单的 HRT 控制循环不需要高的内存带宽，而视频处理应用程序需要内存带宽和延迟. Cache Line 分配中的冲突称为空间争用；由多个核同时访问同一个级别的 Cache，称为时间域的争用.

解决基于 Cache 的传统存储结构可预测性的 2 种最常用的方法都是在 Cache 上强制执行更确定的行为.

第 1 种实现时间隔离的方法是 Cache 分区，它将 Cache 划分为若干分区，并将特定的分区分配给任务或核心.基于组相联 Cache 的结构，这些方法可以进一步分类为：

1）基于 index 的缓存分区，每个 index 指示的 Cache Line 被划分一个分区(水平切片)；

2）基于 way 的缓存分区，每一个可用的 Cache Way 都被专门划分为一个分区(垂直切片).

第 2 种常用技术是 Cache 锁定. Cache 锁定依赖于许多嵌入式平台中存在的硬件特性. 具体来说，可以将给定的 Cache Line 或方式标记为锁定，从而防止其内容在连续执行解锁操作之前被替换[36,49]. 应用程序可以通过操作系统(OS)内存分配器使用 Cache 分区和锁定.

单核情况下只有一对指令和数据 Cache，时序较为确定，可分析性强. 在多核情况下，保持各个核心的 Cache 一致是保证逻辑正确性的必然要求. 在通用计算机系统中，应用较为广泛的 Cache 一致性协议是 MSI 协议，它定义了 Cache 的 3 个状态：Modified(已修改)、Shared(共享)、Invalid(无效)[50]，可以保持多核情况下各个 Cache 数据的一致性. 但分析表明，仲裁延迟随核数的增加呈线性增长，而一致性延迟随核数的增加呈二次增长，这强调了考虑缓存一致性对延迟范围影响的重要性[51].

University of Waterloo 的研究人员在 2017 年基于 MSI 协议设计了可预测的 PMSI Cache 一致性协议，通过引入一组暂态来实现 MSI 上提出的不变量，同时仅对缓存控制器进行最小的架构更改. 在维持可预测性的同时，PMSI 协议在 4 核系统上比竞争总线的调度方法实现 4 倍的性能提升[51]. 随后不久，又提出了新的缓存通信机制 CARP[52]，CARP 在 SPLASH2 基准测试中比 PMSI 协议性能提高 4%. University of Waterloo 的研究人员在可预测一致性协议的基础上实现了实例 MapleBoard[53]，并支持 MSI, MESI, PMSI[51], PMESI[54], CARP[52] 一致性协议.

在总线层次，Bus 一端承接了多个 Cache，接收多个核心的访存请求；另一端连通多个存储器或外设(也可以看作特定地址的存储器). 需要解决的问题是设计可预测的总线架构和仲裁算法. 目前通用的总线结构解决方案是使用 2 级调度器，将每个核心的访存请求按可调度算法排成队列，在外界看来仅有一个被核内时间可预测调度器选择的访存请求，而多核间的请求由高一级的调度器选择一个核心的请求并给予总线控制权. 目前通用的可预测的调度方法为 TDMA 的轮询型调度，它保证了每个请求的最长响应时间和最长结束时间. 作为实例，Barcelona Supercomputing 的研究人员提出的可预测性架构[55]就使用了 2 级调度结构，以 ICBA(intra-core bus arbiters)调度核内请求，以 XCBA(inter-core bus arbiters)调度核间请求，并给出了可预测的调度算法和仲裁方法.

存储器方面，实时嵌入式系统一般采用动态 RAM(DRAM)，以应对现代设计中不断增加的数据量和代码大小. 然而，到目前为止，内存控制器的设计主要集中在提高平均性能上[19]. 因此，内存访问的延迟是不可预测的，这使得硬实时嵌入式系统所需的 WCET 分析变得复杂. Hassan 等人[56]提出了一种可预测的内存请求调度方法，该方法能在大量的请求中保持局部性，最小化最坏情况的延迟，同时保证了平均延时没有明显增长. 这种方法引入了一种新颖的紧凑的时分多路复用调度器和一种新的框架，用于构造多核混合关键系统的最优片外 DRAM 存储器控制器调度. 这些调度计划在引导阶段加载到内存控制器.

RPU 扩展而成的实时系统 RTS 采用传统经典的 Cache-Bus-Memory 结构，利用 TDMA 的轮询调度保证总线和 Cache 访问的可预测性[57]，相较于 PRET，能更简单地移植基于 Cache 存储系统编写的程序.

4.4.2 便签存储器

便签存储器（scratchpad memory，SPM）被认为是一种比 Cache 更可预测的选择. SPM 是一种特殊的静态 RAM 存储器，放置在靠近处理器的地方，类似于 L1-Cache. SPM 的地址空间被映射到处理器预定义的内存地址上. 与 Cache 不同，SPM 必须显式管理. 即某个数据在使用之前，其所在内存块必须在软件中从主存中移动并复制到 SPM 中. 因此，SPM 是高度可预测的，因为它们只有一个 CPU 运行时钟周期的访问延迟，而 Cache 有 2 个不同的 Cache 命中延迟和未命中延迟. 然而，SPM 显式程序管理的需求意味着基于 Cache 的系统编写的程序不能轻松地移植到基于 SPM 的系统上执行，而且在可用的商业系统中，对便签存储器的支持是有限的[58].

PRET 机采用 SPM 较为简单地处理存储系统的可预测性问题. 以 FlexPRET 为例，FlexPRET 有单独的指令和数据存储器，使用软件管理的 SPM 而不是 Cache，并且程序只允许访问 SPM 允许的区域. 因此，所有有效的内存访问总是成功的，并且单周期返回结果. FlexPRET 每个周期只能发生一次存储器操作；在加载或存储操作期间，不会获取新指令，从而使用 SPM 降低了处理器的总吞吐量. 在常见的通用设计中，通常使用指令和数据 Cache 而不是 SPM 来构建存储系统，但对于具有类似或更高复杂性的处理器来说，PRET 认为这是合理的设计决策. 其理由是，如果代码和数据之间只共享一个 SPM，那么硬件线程的内存操作只能发生在一个预定的周期内，这个周期可能与内存阶段不一致，它需要更多的控制逻辑和数据存储. 例如，每个硬件线程将需要为一个内存操作存储地址和数据，直到下一个预定周期，然后阻止读取. FlexPRET 可选地允许对指令内存而不是数据内存进行内存操作.

4.5 本节小结

在硬件层实现实时性的支持，设计自由度最大，但必须考虑从成熟的通用系统迁移实时应用的成本. 实时领域的研究工作大多针对单一软硬件平台，为特定应用环境开发专用的成套软硬件是可行的做法，但研究成果难以整合，导致在实时系统研究方面工业界与学术界脱钩，不同研究者也难以复现彼此工作，因此难以深刻认识各自工作的价值. 实时系统的研究相较于其他领域需要更深的积累、更长的周期，需要自行搭建自编程语言到硬件的研究平台. 更有甚者，由于实时系统面临安全关键问题，而软硬件平台又没有很好的抽象，导致系统即使需要微小的升级，也需要重新开始整个设计周期，导致大量的人力物力浪费.

5 总　　结

本文针对实时计算系统的发展情况，从计算机系统结构的各个层次分别对实时计算系统的需求、问题以及解决方案进行了论述. 在对现有的实时系统设计研究的过程中，本文注意到实时系统设计问题的关键在于上层应用的时间语义难以与底层实现保持一致，为此，需要在 ISA 层以及硬件设计时引入时间语义. 针对在 ISA 层引入时间语义的方法，本文着重描述了现有的 2 种解决方案，分别是来自 UC 伯克利大学设计的 PRET 项目以及本文实验室所设计的 RPU 项目，并从 ISA 层的时间语义方面分别介绍了 2 种解决方案的异同. 在硬件层面上，本文针对现有硬件设计总结了在实现时间可预测性方面的问题与挑战，从多线程和存储系统的实时性方面入手，分别对比了传统设计和实时系统的设计，论述了如何在硬件层面取得更好的时间可预测性.

并行性和实时性是目前安全关键实时系统的核心要求和特征，传统计算平台对此支持不足，导致系统设计、实现和验证成本高昂，必须从 3 个基础设施入手加以解决.

1）多核与多线程实时系统

向多核架构的迁移对硬实时系统的发展提出了极大的挑战. 这是因为共享硬件资源（如内存、互连、I/O 等）的存在会在核心之间造成不必要的干扰. 反过来，这种干扰使得我们很难得出安全且准确的任务执行时间的最差情况界限，而这是保证时间限制所必需的.

一个重要的问题是如何处理多核情况下的 CPU 运行时钟和时间语义需要的计时时钟的同步性. 一方面，研究表明分析多核系统上并行程序的存储器一致性需要显式的全局时钟[59]. 对于没有显式全局时钟的片上多核处理器系统，Lamport[60] 证明了在分布式系统中，如果各个进程使用不同的分布时钟，则使用一定的算法可以把不同的时钟换算成一个近似的全局时钟，并且保证这个全局时钟的误差足够小. 另一方面，为了保证时间语义并利于验证，每个核心上

的计时时钟也需要同步,一种简单的实现是为每个核心接入同一个外部晶振作为计时时钟.

2)时间可预测 ISA

实时系统的 ISA 应该提供给上层应用足够的时间抽象. 对于未来的时间增强 ISA, 我们认为其应该:

①提供时间管理接口,从用户需求出发抽象出秒、毫秒、纳秒等计时单位,并设置时间粒度.

②满足用户对于定时执行的需求. 时间需求可以抽象为绝对点时间、相对点时间、绝对段时间、相对段时间, ISA 应该具有能满足这些需求的语义.

③提供程序隔离能力,为新一代的混合关键系统提供执行环境.

④能够提高处理器的利用率,改善长期以来实时系统因预留资源导致计算能力大量浪费的现状.

3)时间可预测存储系统

向多核方向发展需要重新设计可预测的存储系统. 基于传统 Cache 的存储系统和基于 SPM 的存储系统存在较大的差异性, SPM 可预测性更好,但需要显式管理,目前流行的基于 Cache 的存储系统编写的程序都不能很方便地在使用 SPM 的系统中运行. 在基于经典的 Cache-Bus-Memory 结构的实时系统方面,很多研究者都提出了不同的解决方案. 在 Cache 层次,多核之间需要时间可预测的一致性通信算法,目前这方面的工作主要有 University of Waterloo 的研究人员提出的可预测 MSI 方法 PMSI, 以及 PMESI, CARP 等方法;而多核 Cache 内部同一处理器的访问需要具备时间可预测性,目前流行的解决方法是采用 TDMA 的调度方式消除时间不确定性. 未来这方面的工作主要集中在提高一致性方法的性能并减少消耗上. 总线结构和调度方法上,需要着重考虑实时性和性能之间的权衡. 严格轮询的调度具有很好的可预测性,但会严重损失性能. TDMA 的调度方法是目前性能和实时性之间综合考虑的选择. 未来的实时系统使用怎样的总线结构和调度方法仍然有一定的挑战. 存储器控制器和总线面临的情况类似. 与总线系统相比,存储器控制器仅需要调度一侧的请求. 未来对实时系统中存储系统的创新更有可能是整体结构的重新设计,在保证实时性的基础上能以较小的开销实现、迁移基于经典存储系统编写的程序.

作者贡献声明: 龚小航、蒋滨泽共同调研并撰写论文;陈香兰、高银康、李曦提供论文指导和修改意见.

参 考 文 献

[1] Wikipedia. Whirlwind I[EB/OL]. [2022-08-18]. https://en.wikipedia.org-/wiki/Whirlwind_I

[2] Mitra T, Teich J, Thiele L. Time-critical systems design: A survey[J]. IEEE Design & Test, 2018, 35(2): 8−26

[3] Roop P. Predictable reactive processors for next generation computing: A proposal[EB/OL]. [2022-05-30]. https://www.researchgate.net/publication/254356051

[4] Andalam S, Roop P, Girault A, et al. PRET-C: A new language for programming precision timed architectures[J/OL]. Inria, 2009[2022-08-18].https://hal.inria.fr/inria-00391621

[5] 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

[6] Wang Chao, Chen Xianglan, Zhang Bo, et al. A real-time processor model with timing semantic[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)

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

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

[9] Laplante P A, Ovaska S J. Real-Time Systems Design and Analysis: Tools for the Practitioner[M]. 4th ed. Piscataway, NJ: IEEE, 2011

[10] Cheng A M K. Real-Time Systems: Scheduling, Analysis, and Verification[M]. Piscataway, NJ: IEEE, 2011

[11] Thiele L, Wilhelm R. Design for timing predictability[J]. Real-Time Systems, 2004, 28(2): 157−177

[12] Kirner R, Puschner P. Time-predictable computing[C]//Proc of the Software Technologies for Embedded and Ubiquitous Systems. Berlin: Springer, 2010: 23−34

[13] Cullmann C, Ferdinand C, Gebhard G, et al. Predictability considerations in the design of multi-core embedded systems[C]//Proc of Embedded Real Time Software and Systems. Toulouse, France: Association Aéronautique et Astronautique de France, 2010: 36−42

[14] Davis R I, Burns A. A survey of hard real-time scheduling for multiprocessor systems[J]. ACM Computing Surveys, 2011, 43(4): 1−44

[15] Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem —Overview of methods and survey of tools[J]. ACM Transactions on Embedded Computing Systems, 2008, 7(3): 1−53

[16] Zhuravlev S, Saez J C, Blagodurov S, et al. Survey of scheduling techniques for addressing shared resources in multicore processors[J]. ACM Computing Surveys, 2012, 45(1): 1−28

[17] Lee E, Reineke J, Zimmer M. Abstract PRET machines[C]//Proc of

the 2017 IEEE Real-Time Systems Symp(RTSS). Piscataway, NJ: IEEE, 2017: 1−11

[18] Liu I. Precision timed machines[D]. Berkeley, CA: University of California, Berkeley, 2012

[19] Reineke J, Liu I, Patel H D, et al. PRET DRAM controller: Bank privatization for predictability and temporal isolation[C]//Proc of the 9th IEEE/ACM/IFIP Int Conf on Hardware/Software Codesign and System Synthesis (CODES+ISSS). New York: ACM, 2011: 99−108

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

[21] Halbwachs N. Synchronous Programming of Reactive Systems[M]. Berlin: Springer, 1998: 1−16

[22] 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)

[23] Cloudflare, Inc. μC/OS-II[EB/OL]. [2022-08-18]. https://www.osrtos.com/rtos/uc-os-ii

[24] Amazon Web Services. FreeRTOS[EB/OL]. [2022-08-18]. https://www.freertos.org

[25] Network Solutions, LLC. RTEMS[EB/OL]. [2022-08-18]. https://www.rtems.org

[26] Wang Chao. Research on time-predictable computer architecture[D]. Hefei: University of Science and Technology of China, 2022 (in Chinese)
(汪超. 时间可预测计算机体系结构研究[D]. 合肥: 中国科学技术大学, 2022)

[27] Li Xi, Chen Xianglan, Wang Chao, et al. Design Method of Real-Time Embedded System[M]. Beijing: Tsinghua University Press, 2022(in Chinese)
(李曦, 陈香兰, 王超, 等. 实时嵌入式系统设计方法[M]. 北京: 清华大学出版社, 2022)

[28] Wägemann P, Distler T, Eichler C, et al. Benchmark generation for timing analysis[C]//Proc of the 2017 IEEE Real-Time and Embedded Technology and Applications Symp(RTAS). Piscataway, NJ: IEEE, 2017: 319−330

[29] Falk H, Lokuciejewski P. A compiler framework for the reduction of worst-case execution times[J]. Real-Time Systems, 2010, 46: 251−300

[30] Broman D, Zimmer M, Kim Y, et al. Precision timed infrastructure: Design challenges[C/OL]//Proc of the 2013 Electronic System Level Synthesis Conf(ESLsyn). Piscataway, NJ: IEEE, 2013 [2022-08-18].https://ieeexplore.ieee.org/abstract/document/6573221

[31] Dutt N D. Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs[J]. ACM Transactions on Design Automation of Electronic Systems, 2004, 11(3): 626−658

[32] Banakar R, Steinke S, Lee B S, et al. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems[C/OL]//Proc of the 10th Int Symp on Hardware/Software Codesign(CODES 2002). Piscataway, NJ: IEEE, 2002 [2022-08-19].https://ieeexplore.ieee.org/abstract/document/1003604

[33] Rice H G. Classes of recursively enumerable sets and their decision problems[J]. Transactions of the American Mathematical Society, 1953, 74(2): 358−366

[34] Kopetz H. Event-triggered versus time-triggered real-time systems[G]//the Operating Systems of the 90s & Beyond, International Workshop. Berlin: Springer, 1991: 86−101

[35] Kopetz H, Bauer G. The time-triggered architecture[J]. Proceedings of the IEEE, 2003, 91(1): 112−126

[36] Aparicio L C, Segarra J, Rodríguez C, et al. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems[J]. Journal of Systems Architecture, 2011, 57(7): 695−706

[37] Pont M J. Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers[M]. Boston: Addison Wesley, 2001

[38] Hughes Z. Design and evaluation of a predictable embedded processor for use in timetriggered applications[D]. Leicester, the UK: University of Leicester, 2010

[39] Schoeberl M. A Java processor architecture for embedded real-time system[J]. Journal of Systems Architecture, 2008, 54(1): 265−286

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

[41] 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

[42] 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. Piscataway, NJ: IEEE, 2005: 977−981

[43] 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

[44] 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 the 2017 IEEE Real-Time Systems Symp(RTSS). Piscataway, NJ: IEEE, 2017: 363−365

[45] Henzinger T, Horowitz B, Kirsch C. Giotto: A time-triggered language for embedded programming[J]. Proceedings of the IEEE, 2003, 91(1): 84−99

[46] Lawes J. Car brakes, New Zealand: A Guide to Upgrading, Repair and Maintenance[M]. Marlborough, New Zealand: Crowood Press, 2014

[47] 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

[48] 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

[49] Sarkar A, Mueller F, Ramaprasad H. Static task partitioning for locked caches in multicore real-time systems[J]. ACM Transactions on Embedded Computing Systems, 2015, 14(1): 4: 1−4: 30

[50] Patterson D A, Hennessy J L. Computer Organization and Design MIPS Edition: The Hardware/Software Interface[M]. 5th ed. San Francisco, CA: Morgan Kaufmann, 2013

[51] Hassan M, Kaushik A M, Patel H. Predictable cache coherence for multi-core real-time systems[C]//Proc of the 2017 IEEE Real-Time and Embedded Technology and Applications Symp(RTAS). Piscataway, NJ: IEEE, 2017: 235−246

[52] Kaushik A M, Tegegn P, Wu Zhuanhao, et al. CARP: A data communication mechanism for multi-core mixed-criticality systems[C]//Proc of the 2019 IEEE Real-Time Systems Symp(RTSS). Piscataway, NJ: IEEE, 2019: 419−432

[53] Wu Zhuanhao, Kaushik A M, Tegegn P, et al. A hardware platform for exploring predictable cache coherence protocols for real-time multicores[C]//Proc of the 27th IEEE Real-Time and Embedded Technology and Applications Symp(RTAS). Piscataway, NJ: IEEE, 2021: 92−104

[54] Kaushik A M, Hassan M, Patel H. Designing predictable cache coherence protocols for multi-core real-time systems[J]. IEEE Transactions on Computers, 2020, 70(12): 2098−2111

[55] Paollieri M, Quiñones E, Cazorla F J, et al. Hardware support for WCET analysis of hard real-time multicore systems[C]//Proc of the 36th Annual Int Symp on Computer Architecture. New York: ACM, 2009, 57−58

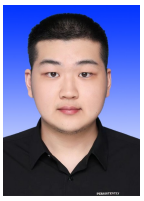
[56] Hassan M, Patel H, Pellizzoni R. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems[C]//Proc of the 21st IEEE Real-Time and Embedded Technology and Applications Symp. Piscataway, NJ: IEEE, 2015: 307−316

[57] Rosen J, Andrei A, Eles P, et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip[C]//Proc of the 28th IEEE Int Real-Time Systems Symp(RTSS). Piscataway, NJ: IEEE, 2007: 49−60

[58] Gracioli G, Alhammad A, Mancuso R, et al. A survey on cache management mechanisms for real-time embedded systems[J]. ACM Computing Surveys, 2015, 48(2): 1−36

[59] Wang Pengyu, Chen Yunji, Shen Haihua, et al. Memory consistency verification of chip multi-processor[J]. Journal of Software, 2010, 21(4): 863−874 (in Chinese)
(王朋宇, 陈云霁, 沈海华, 等. 片上多核处理器存储一致性验证[J]. 软件学报, 2010, 21(4): 863−874)

[60] Lamport L. Time, clocks, and the ordering of events in a distributed system[J]. Communications of the ACM, 1978, 21(7): 558−565



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架构.

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架构.

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.

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架构.

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架构.