基于抽象解释的嵌入式软件模块化 Cache 行为分析框架

喻垚慎[1] 黄志球[1),2),3)] 沈国华[1),2),3)] 王 飞[1)] 崔少轩[1)]

[1)](南京航空航天大学计算机科学与技术学院 南京 211106)

[2)](高安全系统的软件开发与验证技术工业和信息化部重点实验室(南京航空航天大学) 南京 211106)

[3)](软件新技术与产业化协同创新中心 南京 210093)

摘 要 程序最坏执行时间(Worst Case Execution Time,WCET)是嵌入式实时系统时间属性验证的基础,在采用静态分析技术的 WCET 估算中需要分阶段对不同的执行环境约束条件进行分析,并整合所有约束信息、结合程序控制流结构估算全局最坏路径,因此各阶段分析的中间结果对最终的 WCET 估算性能具有较大影响.在现代嵌入式系统中,硬件平台中的 Cache 机制成为对执行时间影响较大的硬件体系结构,对其进行精确的行为分析在 WCET 估算中具有重要的现实意义.采用抽象解释理论对 Cache 行为进行分析已有较为成熟的技术成果和相关工具,但由于静态分析技术具有较难理解和使用的特点,对于技术没有覆盖、工具没有支持的硬件架构,针对这类硬件架构进行相关研究和验证工具开发都具有较大难度和挑战.该文以抽象解释为理论基础,以复用 Cache 分析过程为目标,提出了基于抽象解释的模块化 Cache 行为分析框架,对 Cache 行为分析过程进行了层次划分,提出了易于复用的 Cache 行为分析方法设计,能够针对不同架构的 Cache 机制分析方法进行建模,并以统一的分析框架对分析过程进行复用.案例实验表明,该框架可支持采用抽象解释对使用 LRU 策略的 Cache 行为进行建模分析,并能够得到 Cache 命中情况标记信息以支持后续 WCET 的估算过程.

关键词 嵌入式软件;Cache 行为分析;静态代码分析;模块化分析;抽象解释

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2019.02251

Abstract-Interpretation-Based Framework of Modular Cache Behavior Analysis for Embedded Software

YU Yao-Shen[1)] HUANG Zhi-Qiu[1),2),3)] SHEN Guo-Hua[1),2),3)] WANG Fei[1)] CUI Shao-Xuan[1)]

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

[2)](*Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing* 211106)

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

Abstract The Worst-Case Execution Time (WCET) estimation is the basis of timing validation for embedded real-time system. In WCET estimation research which use static analysis techniques, it will separate into different stages for different execution environments like cache and pipeline to analyze the constraints which will affect the execution time of the program. Then it integrates all these constraints from different analysis stages and combines them with the control flow structure of the program to explore the global worst-case execution path and calculate the worst-case execution

收稿日期:2017-11-06;在线出版日期:2018-07-26.本课题得到国家"八六三"高技术研究发展计划基金项目(2015AA105303)、国家重点研发计划(2016YFB1000802)、国家自然科学基金(61502231)资助.**喻垚慎**,博士研究生,中国计算机学会(CCF)学生会员,主要研究方向为软件工程、形式化方法、静态程序分析和嵌入式软件建模与分析. E-mail: yaoshen.yu@outlook.com. **黄志球**(通信作者),博士,教授,中国计算机学会(CCF)理事,主要研究领域为软件工程、形式化方法和云计算. E-mail: zqhuang@nuaa.edu.cn. **沈国华**,博士,副教授,中国计算机学会(CCF)高级会员,主要研究方向为需求工程、软件安全性分析、语义 Web. **王 飞**,博士研究生,中国计算机学会(CCF)学生会员,主要研究方向为软件工程、需求工程、形式化方法和嵌入式软件建模与分析. **崔少轩**,硕士研究生,中国计算机学会(CCF)学生会员,主要研究方向为软件工程、形式化方法.

time based on the worst-case path. Therefore, the intermediate results of each stage analysis such as cache behavior analysis, have a great influence on the performance of final WCET estimation. In modern embedded system, the cache mechanism in the processor is an execution environment that affects the execution time very prominently. So, it has a practical significance for an accurate and safe cache behavior analysis in WCET estimation process which will be applied in embedded system especially in safety-critical system. Using abstract interpretation to analyze cache behavior is a very mature technology and there are some related tools for industrial WCET estimation like aiT, OTAWA and so on. However, because static analysis is difficult to understand and use, if there exist any hardware architecture which will not supported by existing analysis techniques or verification tools, it is very difficult to study the analysis method and develop related tool to deal with these architectures. In this paper, it takes the abstract interpretation as the theoretical basis and takes the reuse of the cache behavior analysis process as the goal, proposes an abstract-interpretation-based framework for modular cache behavior analysis. This framework divides the cache behavior analysis process hierarchically, formalize the design method of the cache behavior analysis which is easily reusable. The design can be used to model the cache analysis methods for different architectures and reuse the analysis process in a unified framework. In case study, it shows that the framework can support the use of abstract interpretation to model and analyze the cache behavior using LRU replacements strategy, and the analysis result is the cache hit category of memory access which can be used in subsequent WCET estimation.

Keywords embedded software; Cache behavior analysis; static code analysis; modular analysis; abstract interpretation

1 引 言

在软件系统中,程序正确性需求不仅包含程序执行结果的正确性,同样也包含程序是否能够在规定时效内完成其执行过程.对于一般性的软件系统,在需求分析中通常不包括对程序执行时间的要求,然而在安全关键软件中存在很多硬实时性的任务需求,一旦软件未能在规定的时效内完成既定功能,则有可能会出现巨大财产损失和人员伤亡等灾难性后果[1].安全关键软件的部署平台通常采用嵌入式系统,如航空机载系统、车载电子控制系统、医疗设备等.嵌入式系统具有硬件架构多样的特点,对于这类存在硬实时需求的嵌入式软件系统,如何在可执行代码层面结合硬件架构进行程序时间属性的验证,成为安全关键软件工程中必不可少的一项工作.

现有的嵌入式软件时间属性验证方法,都需要知道系统中各项任务的最坏执行时间(Worst-Case Execution Time,WCET)[2-3].目前针对 WCET 的估算方法主要从软件和硬件两个方面入手,通过软件监测或者双回路基准测试的方法都会对代码进行修改,从而改变真实的执行环境,时间属性分析的结果会存在较大偏差.而通过硬件仿真、模拟或直接测量的方法,往往只能对特定输入进行执行时间的检测,其结果没有覆盖性,不能完全体现最坏执行情况.

在具有高安全性要求的机载嵌入式软件领域,适航标准 DO-333 作为对 DO-178C 标准的扩充,提供了多种形式化方法用于在软件开发过程中对机载软件进行验证.在代码验证阶段,该标准建议采用抽象解释等静态分析技术对源代码或可执行目标代码进行静态代码分析,主要分析内容包括对运行时错误的检查和对不可达代码段的检查等[4].抽象解释理论能够证明代码不存在某些错误,同时具备速度快、不执行等特点,对于大型机载软件的错误验证具有良好的性能,并在业界已有广泛的应用[5-12].因此,使用形式化方法对软件最坏执行时间进行估算能够保证其方法具有有效性和安全性.

将静态代码分析技术应用于 WCET 估算,通常作用于两个方面:硬件架构无关的源代码层面流信息分析和硬件相关的可执行代码层面执行模型分析[13].流信息分析是在源代码层面对程序控制流和数据流进行相关流属性分析,找出程序执行路径相

关约束如分支条件、循环体执行次数、不可达路径查找等；而执行模型分析是对硬件架构的程序执行行为进行建模，静态的分析在硬件中该程序执行时的行为，建立程序执行环境约束，对影响程序执行时间的属性进行表达和分析，从而得到程序执行时间估计. 执行模型分析需要对影响程序执行时间的相关硬件架构进行建模，主要包括 CPU 流水线机制、Cache 机制以及总线等.

使用静态分析技术对嵌入式软件 WCET 进行估算是一个较为复杂的系统性工程. 不仅需要在代码层面(源代码或可执行代码)对软件执行相关约束进行分析和表达，还需要在硬件执行层面对硬件架构的执行过程进行建模分析，如对 Cache 行为进行建模分析其在内存访问时的 Cache 命中情况、对 CPU 指令执行行为进行建模分析其指令和数据在执行时的处理流程，最终通过对软硬件添加执行环境约束信息，并以此找出所有程序执行可能情况中的最坏路径. 在 WCET 估算中，某一分析阶段的局部性结果并不能指导全局最坏路径的找出(某一程序段中 Cache 一直不命中造成较多时间消耗并不能决定该段程序一定处于最坏路径)，因此在使用静态分析技术对 WCET 进行估算时，通常采用各阶段分析结果信息叠加的形式(如将 Cache 命中情况标记记录在程序控制流结构中)，将所有分析的中间结果积累起来，最终在最坏路径分析中使用这些结果得到安全、精确的 WCET 估计. 所以，针对不同分析阶段如何得到较为精确的分析结果，是影响最终 WCET 估算性能的重要因素.

现代嵌入式系统的硬件架构中通常都使用了高速缓存(Cache)机制，Cache 是用于弥补 CPU 处理速度远高于内存访问速度的结构，由于内存访问速率远低于 CPU 时钟，因此在程序执行时会出现 CPU 需要等待内存指令或数据的现象，导致 CPU 等待一定的读取时间，从而降低了 CPU 的工作效率. 在 CPU 和内存之间增加一级访问速率较高的 Cache，可以将 CPU 所要使用的信息提前从低速的内存读取至高速的 Cache 中，当 CPU 需要该信息时，可以直接从 Cache 中读取而不必等待内存访问的时间，从而提高 CPU 执行效率. Cache 的使用可以提高程序的平均执行效率，但是对于估算程序最坏执行时间则带来了更多的问题. Cache 对最坏执行时间的影响主要体现在 Cache 中指令和数据的命中情况，不同的 Cache 命中情况(即 Cache 行为)影响了程序在执行时的 CPU 执行时间，而 Cache 中的真实储存信息实则对 CPU 计算时间并无影响，因此在分析程序最坏执行时间的过程中，如何对 Cache 行为进行建模以表达程序执行过程中 Cache 的命中情况，是影响 WCET 分析精度的关键过程之一.

在程序 WCET 估算中，对 Cache 机制进行分析，即是对程序执行过程中的所有 Cache 行为进行分析，以确定程序执行时的指令或数据在 Cache 中的命中情况，也是以此估算程序执行过程在 Cache 中所需的时间消耗. 在嵌入式系统中，由于不同硬件平台使用了不同结构的 Cache 机制，导致目前基于静态分析技术的 WCET 分析工具如 OTAWA[14-15]、aiT[12]等只能对其工具所支持的硬件架构进行分析，而对于其他类型的硬件平台，此类工具在使用和二次开发上都具有较大的难度. 开源项目 OTAWA 过于复杂，针对 Cache 行为分析的相关代码难以进行二次开发以适应用户指定的硬件平台；而闭源工具 aiT 则无法进行客户自定义开发，同时软件定制的成本也较为高昂.

因此针对嵌入式系统中 Cache 结构多样、交换策略不一致造成的 Cache 分析过程无法重用的情况，本文从复用和重构的角度出发，提出一种基于抽象解释的模块化 Cache 行为分析框架，以抽象解释理论为分析方法设计基础，对 Cache 行为的执行过程进行抽象建模；对 Cache 架构和分析方法进行关联分析，形式化定义 Cache 行为分析过程，并构建模块化分析框架以支持多种不同架构的 Cache 行为分析，从而达到 Cache 分析过程可重用的目的.

本文第 2 节介绍相关抽象解释理论在硬件执行环境建模方面的已有研究工作；第 3 节简述使用抽象解释对 Cache 行为进行分析的基本思想；第 4 节对模块化 Cache 行为分析框架进行概述设计，总结该框架下影响 Cache 分析结果的两种设计要素(状态描述和状态更新函数)和三种分析方法(Must 分析、May 分析和 Persistence 分析)；第 5 节对基于抽象解释的 Cache 语义进行定义，并以 Must 分析、May 分析和 Persistence 分析为例简述其 Cache 分析方法的设计；第 6 节通过简单案例描述 Cache 行为分析的应用过程；第 7 节对已有工作进行总结，分析该框架合理性及未来研究方向；最后总结全文，并对未来值得关注的研究方向进行初步探讨.

2　相关研究工作

使用抽象解释静态分析对程序执行时间进行估

计最核心的问题就是对硬件执行环境建模.在静态分析 WCET 估计发展的 20 年里,围绕如何对执行环境进行建模一直是研究的热点,各种分析方法的设计也与执行环境建模息息相关.硬件执行环境中,对执行时间影响较大的架构主要是 Cache 和流水线(Pipeline)机制,因此针对执行环境建模方面的研究主要也集中于 Cache 和 Pipeline 的行为分析.

在 Cache 行为建模分析方面已有较多的理论研究,文献[16-17]对 Cache 语义进行了建模,首先提出了使用抽象解释描述 Cache 行为的静态分析方法,主要针对使用 LRU 交换策略的 Cache 机制进行了建模,提出了 Must 分析、May 分析、Persistence 分析三种 Cache 行为分析方法[17-19];文献[20]对数据缓存进行了建模,与指令缓存不同的是,在数据指令中由于存在读写操作,因此对于多地址访问如数组、指针等分析存在较大难度,该文献使用全局数据流分析和数据依赖分析尽可能减小读写指令对 Cache 命中情况的影响;文献[21]提出了改进的组相联指令 Cache 局部分析方法,通过将程序切割成独立的组件以进行局部分析来降低对整个程序分析的计算时间消耗;文献[22]提出了对 Ferdinand 采用的 Persistence 分析的修正方法,对多次循环情况下语义不动点计算的错误进行了修正,完善了 Cache 行为分析方法;文献[23]提出了一种精化的 Persistence 分析方法以修正原有分析可能会低估的问题,通过结合内存访问模式分析和抽象解释提高分析精度,使用计算地址分析期间的访问时间范围来捕获内存访问的动态行为,实现更精确的抽象缓存状态建模.

在 Pipeline 建模方面,文献[24]提出了针对流水线机制的抽象解释建模方法,对 Pipeline 中的具体执行状态抽象为抽象执行状态,给定状态变迁规则,使其能够形式化的描述 Pipeline 行为过程,将处理器具体 Pipeline 结构抽象为抽象的 Pipeline 结构以进行分析;文献[25]系统化的对 Pipeline 语义进行了定义,完整的设计了抽象解释方法对 Pipeline 机制的建模过程,提供了较为完善的 Pipeline 行为分析方法.

在其他硬件架构方面的研究中,文献[26]从硬件架构描述语言的角度提出了从 VHDL 代码转换到控制流表达语言(Control-Flow Representation Language,CRL2)的架构描述方式,扩展了静态分析工具 aiT 支持的硬件描述语言,使其能够适用于更多应用场景;文献[27]对架构指令进行抽象,通过描述复杂指令集架构构造一种指令模拟器,使其能够进行指令反汇编、解码和模拟,进而以指令模拟的形式替代真实硬件执行过程进行 WCET 估计;文献[28]提出了使用模型检验建模系统共享总线的方法对多核处理器架构环境进行建模,在单核内部仍然采用已有静态分析技术,而在共享总线机制中使用模型检验进行建模,对多核处理器进行执行时间估计.

对于硬件架构在分析工具的设计和应用方面的影响,也有许多研究进行了相关讨论.文献[29]从分析流程设计和架构对分析方法的影响等方面进行了探讨,给出了一般性执行时间分析结构,并结合 ColdFire 处理器进行了分析工具设计;文献[30]中分析了硬件架构对 WCET 分析工具设计的相关影响,例如缓存替换策略将影响缓存行为分析的结果,进而影响软件指令和数据在缓存中时间消耗估计,处理器组件中的无序执行和控制推测也会为流水线分析引入干扰,这类影响因素造成了 WCET 工具模块化设计的难度;文献[7]介绍了在项目应用中的 WCET 估计工具设计及使用方法,对分析各阶段架构建模及形式化过程进行了相关描述,并真实应用于航空软件 WCET 估计,与 Airbus 所使用的方法相比在精确度上有较为显著的提升;文献[31]从硬件架构建模的角度对执行时间估计工具 aiT 进行了细节介绍,主要介绍了在 Pipeline 机制中的分析工具设计思想.

3 基于抽象解释的 Cache 行为分析

抽象解释[32]理论由 Cousot 等人于 1977 年提出,其本质是程序静态分析时构造和逼近程序不动点语义的理论.该理论的基本思想是以抽象语义表达程序语义,并通过抽象语义上的近似计算逼近程序语义上的真实计算,使得程序抽象执行的结果能够在一定程度上反映程序真实执行的部分信息[6].如图 1 所示,程序语义表示程序中的具体程序执行对象,如变量、函数等,即表示程序在真实执行中的相关对象和操作等.通过程序抽象,将程序语义中的计算对象(变量)和计算过程(函数)抽象表达为另一个抽象域中的抽象属性和抽象操作,并通过抽象域中的不动点迭代计算近似真实程序执行过程,得到能够反映真实执行的抽象计算结果(语义不动点),通过验证抽象结果的属性满足性以间接证明程序的属性是否得到满足[33].

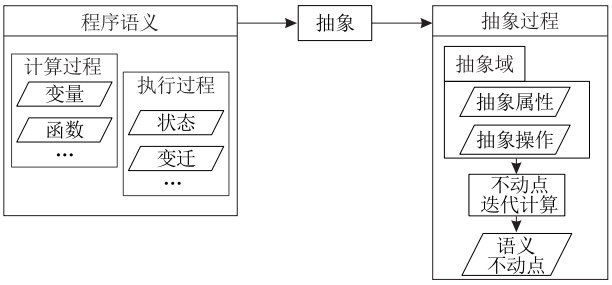

图 1　抽象解释简要工作原理

基于抽象解释的静态代码分析方法可以根据其分析与建模的对象不同，分为两个方面：(1) 面向源代码和程序计算过程的计算语义分析，该分析主要处理对象为程序源代码，抽象的对象为程序的计算过程，通常意义上是对函数的抽象，得到的结果主要为程序变量的可能取值范围，并通过相应的取值范围验证其是否满足关注的安全性质，如程序运行时错误验证等；(2) 面向可执行代码和硬件架构平台的执行过程语义分析，该分析主要处理对象为可执行代码及对应的硬件架构平台，抽象的对象为硬件环境的执行过程，一般以模拟硬件执行的抽象对可执行代码的执行过程进行分析，得到的结果主要为执行过程中的相关属性，以对程序执行过程的约束验证其结果是否满足对其行为约束的安全性质，如程序最坏执行时间等.

基于抽象解释的 Cache 行为分析，属于上述分析方法中的第二种，其基本思想仍然是以抽象语义上的计算表达程序语义中的计算过程，但在 Cache 行为分析中，程序语义并非指具体程序执行过程，而是对其状态和状态更新的描述，即对 Cache 行为的抽象建模和模拟.

如图 2 所示，对 Cache 行为进行抽象表达，则需要对其状态和状态更新分别进行抽象，并且保证其有效性. 基于抽象解释的 Cache 行为分析过程是以抽象的 Cache 状态集合表示实际 Cache 的所有可能状态，以抽象状态更新函数表达实际 Cache 交换过程，并通过抽象状态中的迭代更新以模拟真实程序执行过程中 Cache 的运行情况，最后通过对比当前访问的内存块和抽象的 Cache 可能状态集合来估算在真实程序执行过程中的 Cache 命中情况.

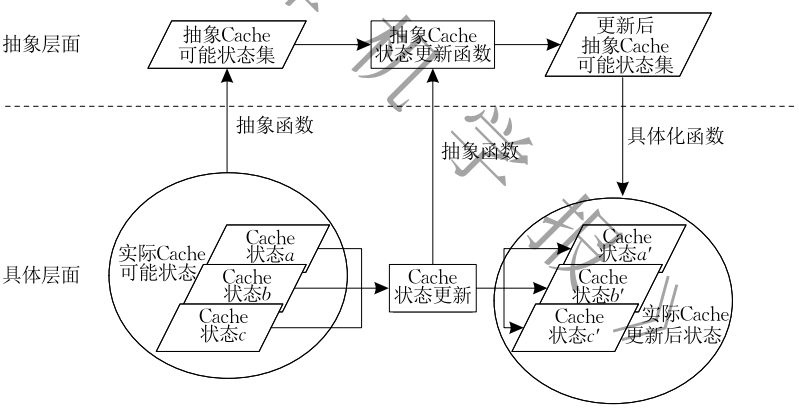

图 2　基于抽象解释的 Cache 分析原理

使用抽象解释对 Cache 行为进行分析主要包括两个方面的抽象：(1) 对 Cache 状态表示进行抽象，对 Cache 状态的抽象建模主要依赖于 Cache 的基本结构，其中包括容量、字块长和地址映射策略三个基本要素，因此对 Cache 状态的抽象即是对以上三种要素的抽象表示；(2) 对 Cache 状态更新进行抽象，对状态更新的抽象则是关注于交换策略的抽象表达，在 CPU 访问内存内容时，若内存块在 Cache 内，则直接访问 Cache 块内容，若当前访问的内存块不在 Cache 内，则需要将内存块交换至 Cache 内并对 Cache 状态进行更新，因此对状态更新的抽象则是对相应交换策略的抽象表达.

由于抽象解释语义对其表达行为具有理论上的有效性保证，因此采用抽象解释理论的 Cache 行为分析结果也能保证其有效性. 本文将在第 5 节具体阐述如何使用抽象 Cache 语义进行 Cache 行为分析设计.

4　模块化 Cache 行为分析框架

对采用静态程序分析的 WCET 估算过程进行分层分解、模块划分已有一定的研究和应用，OTAWA 项目[14-15]对 WCET 估算中的各项功能独立的模块进行了分层划分，并在流程上对 WCET 的信息处理过程进行了多层抽象，确保层与层之间的信息独立性，并且提供了较好的算法替换接口. 然而该项目限

制了用户对硬件建模的修改能力，对于项目不支持的硬件架构难以重新建模，同时也无法继续复用其提供的完整 WCET 估算流程，面对新硬件和新的硬件建模方法其适配性较弱.

在对单核嵌入式软件进行 WCET 估算时，需要有单独的处理流程对 Cache 行为进行分析[31]；而在多核的 WCET 分析工作中，对于独立处理器内部也需要单独进行其 Cache 行为分析[28]，如图 3 所示. 上述两种 WCET 估算流程中对于独立的 Cache 行为分析都采用了基于抽象解释的分析方法，其不同点在于，上述两种 WCET 估算流程处理的硬件架构不同，因此针对各相同阶段的分析方法（如 Cache 分析）都需要开发适用于不同硬件架构的相关分析工具. 采用抽象解释理论的分析方法具有不执行、安全性强的优点，同时也具有理论性强、难以实用、针对性强难以复用的缺点. 因此，针对硬件架构多变的嵌入式系统环境，如何快速开发或复用采用静态分析技术的 WCET 估算工具具有一定的难度和挑战.

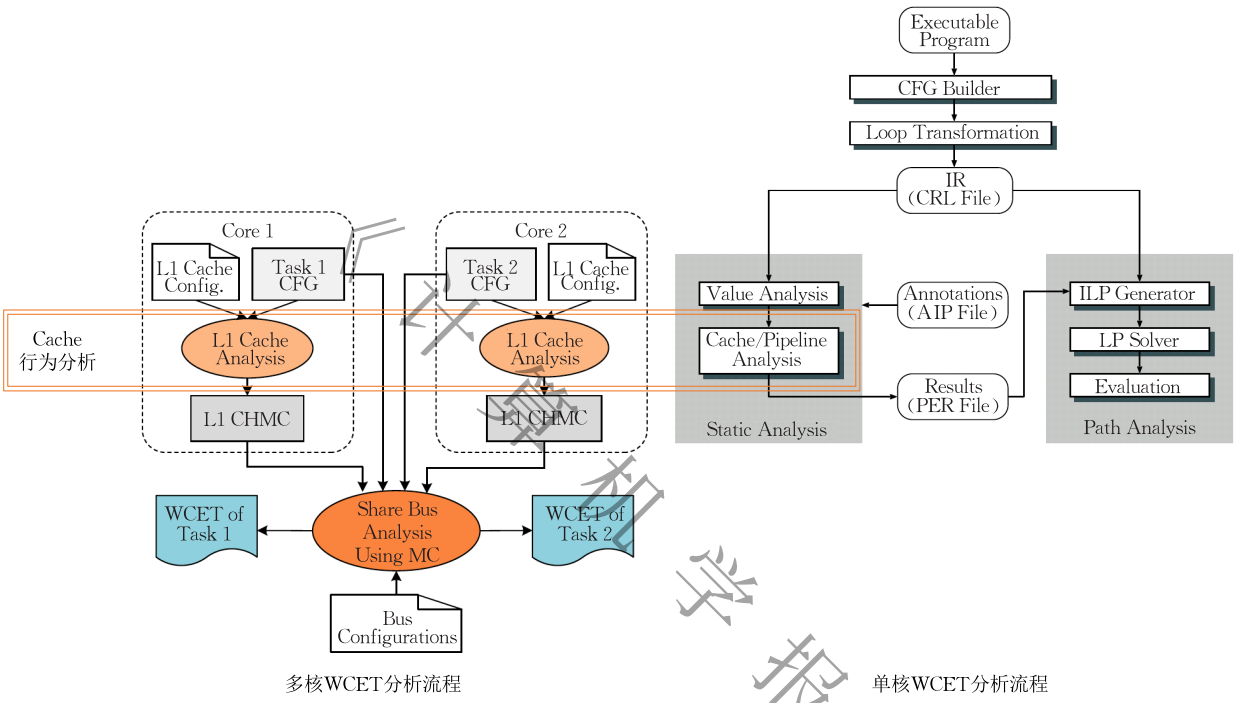

图 3 WCET 估算中的 Cache 行为分析层次

本文从应用层面出发，针对 Cache 结构多样、交换策略不一致的架构，为重用基于抽象解释的 Cache 行为分析过程，提出了模块化 Cache 行为分析框架，对 Cache 行为分析中的共性内容进行归纳，并对 Cache 分析设计和分析过程进行一般性总结. 使基于抽象解释的 Cache 行为分析具有更广泛的适用性，便于用户根据其定制化的硬件平台架构重新设计 Cache 分析流程，减少工具验证开发难度，提高验证工作效率.

4.1 Cache 行为分析框架

本文在此小节提出了对 Cache 行为进行建模分析的一般性框架，用于描述如何通过模块化的方法对 Cache 行为分析方法进行设计，及如何应用该分析方法对真实执行过程中的 Cache 命中情况进行估算.

如图 4 所示，在分析方法的设计方面，需要考虑三方面的问题：Cache 状态描述、Update 函数设计和有效性证明. Cache 状态需要通过 Cache 的结构配置进行分析，根据相应的参数，设计分析过程中如何对状态进行表达，即 Cache 配置决定 Cache 分析过程中状态的表达形式及具体参数. Update 函数则依据不同的交换策略进行设计，用于执行分析过程中的状态更新，即交换策略决定 Cache 分析过程中状态的转换过程. 由于对 WCET 进行估算需要符合一定的安全性，因此在对分析过程进行设计时，还需考虑到分析方法的有效性证明，即对 Cache 状态的描述和 Update 函数的设计都要符合安全的原则，在任何情况下，估算出的 WCET 不能低于真实的执行时间. 在理论方面，本文提出的方法主要基于抽象解释理论，因此上述三个设计要点都依托于抽象解释理论，即通过抽象解释进行 Cache 状态的描述、Update 函数的设计和分析有效性的证明.

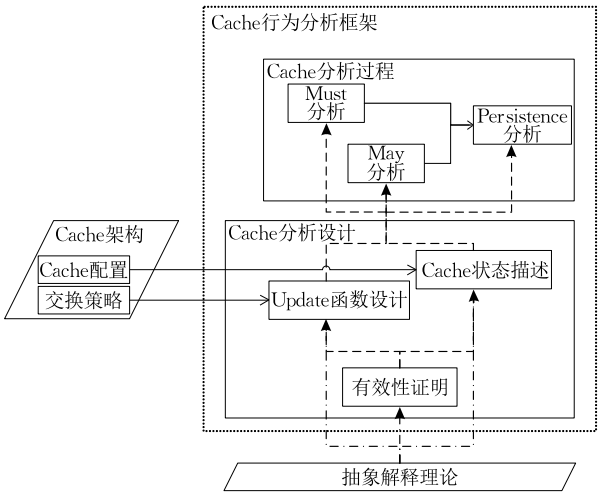

图 4　模块化 Cache 行为分析框架

Cache 行为分析过程是对分析方法的使用，根据目标不同，首先设计不同的 Cache 分析方法，采用不同的分析方法对 Cache 行为进行分析，标注出达到某一程序结点时的 Cache 块状态，并根据所有结点的 Cache 块状态，计算程序执行过程在 Cache 这一硬件上的时间消耗.

在分析过程中，通常会采用三种分析方法对 Cache 行为进行分析：Must 分析[17-19]、May 分析[17-19]、Persistence 分析[17,19,34]. Must 分析用于找出当程序执行到某一程序结点时一直命中的 Cache 块；May 分析用于找出当程序执行到某一程序结点时一直不命中的 Cache 块. 通过 Must 分析和 May 分析，可以将到达某一程序结点时的 Cache 块状态分为三类：一直命中（always hit，ah）、一直不命中（always miss，am）和无法判定（not classified，nc）. 为进一步精确分析结果，则需要缩小其中无法判定（nc 标注）的 Cache 块，确定其命中状态. Persistence 分析则是建立在 Must 分析和 May 分析之上，进一步精确分析结果的分析方法.

由于 Cache 机制的设计目的是为了保持部分内存块存储在速度更快的储存介质中，方便 CPU 更快的使用，而保存的这部分内存块是 CPU 最近使用过的指令或数据，并且近期再次被使用的可能性更高. 因此考虑循环或嵌套的情况，在此类结构下，相同的指令或数据可能在一段时间内被 CPU 反复使用，有可能存在某些内存块在第一次使用时不命中，而在之后循环的每次执行中都一直留存在 Cache 内. 这类内存块在循环或嵌套结构下，最多只有一次 Cache 不命中而其他执行都命中，所以分析出此类 Cache 块的命中情况，能够进一步提升 Cache 行为的分析结果，Persistence 分析就是基于这种思想而提出的. 在 Persistence 分析中，根据对 Must 分析和 May 分析无法判定的情况，在循环或嵌套结构中对标注 nc 的 Cache 块进行分析，找出其是否满足第一次使用未命中，之后使用一直命中的情况.

模块化 Cache 行为分析框架，以抽象解释为理论基础，以统一化 Cache 架构描述为平台信息，通过设计 Cache 行为分析方法并应用，以期完成对多种不同架构 Cache 行为的模块化分析，标注程序执行不同结点下的 Cache 块命中状态，计算程序执行时的 Cache 时间消耗，优化 WCET 估算结果. 该框架下如何对分析方法进行设计本文将在第 5 节进行简要介绍，并在第 6 节给出如何使用 Cache 分析过程的应用.

4.2　Cache 行为分析框架形式化描述

上一小节对本文所提的 Cache 行为分析框架进行了简要描述，介绍了该框架中两种设计要素：Cache 分析设计和 Cache 分析过程，以及两种要素中需要考虑的各种问题. 本小节将对 Cache 行为分析框架进行形式化描述，给出该框架的公式定义.

在 Cache 行为分析框架中，分析方法设计主要以理论定义的形式对 Cache 行为分析的方法和过程进行描述，分析过程则是对分析方法的实现及应用.

4.2.1　Cache 分析方法定义

采用抽象解释理论对 Cache 行为进行分析，即是对真实 Cache 状态及其状态变迁进行抽象表达，以抽象的可能状态集合反映 Cache 行为的真实可能状态. 因此对于某一 Cache 行为分析方法，需要从 Cache 结构配置和交换策略两个角度对其进行设计，对真实 Cache 配置和交换策略进行抽象，在此过程中不仅需要对 Cache 状态进行抽象描述，还需要设计抽象状态更新函数对交换策略的状态更新过程进行表达.

一个 Cache 分析方法可定义为如下形式：

$$CA^M=(C_S,C_R,set(m),AC_S,AC_U,AC_J),$$

其中，C_S 为 Cache 结构配置，即 Cache 容量、块长、地址映射方式；C_R 为 Cache 交换策略描述；$set(m)$ 为内存块映射函数，即当前访问的内存块将被映射到 Cache 的哪一组中（以组相联 Cache 结构为例，直接映射或全相联 Cache 可以看作是组相联 Cache 的特殊形式）；AC_S 为抽象 Cache 状态集合（Abstract Cache State），即将可能储存的内存块以集合形式表示为该 Cache 块内容；AC_U 为抽象 Cache 更新函数，

定义抽象 Cache 状态集合下 Cache 块进行更新的操作函数；AC_J 为抽象 Cache 状态合并函数，当程序控制结构存在分支合并时，需要使用状态合并函数对当前两种 Cache 抽象状态进行合并.

一个 Cache 分析方法，可以看作是由 Cache 结构 C_S 和交换策略 C_R 决定的一系列抽象 Cache 状态定义. 对于不同的 Cache 分析方法，当 Cache 结构和交换策略相同时，其主要区别在于为了找出不同标记的 Cache 命中情况，需要根据分析内容重新定义抽象状态更新函数 AC_U 和抽象状态合并函数 AC_J，本文将在第 5 节分别以 Must 分析和 May 分析为例介绍 LRU 策略下的 Cache 行为分析方法设计.

4.2.2 Cache 分析过程定义

根据上一小节对分析方法的定义，当存在一个内存访问序列时，可采用前文定义的分析方法，对当前 Cache 状态进行行为分析，具体 Cache 分析过程定义如下：

$$CA=(S_{MA},initCS,listCS^M,CA^M),$$

其中，S_{MA} 为内存访问序列，即根据程序执行过程的控制流结构分析得到的相应内存块访问顺序；$initCS$ 为初始 Cache 状态，可根据用户需求进行定义；$listCS^M$ 为采用 Cache 分析方法 CA^M 得到的 Cache 行为历史列表，即对应某一内存块访问时，当前 Cache 抽象状态集合情况记录.

在给定内存访问序列 S_{MA} 和初始 Cache 状态 $initCS$ 的情况下，采用分析方法 CA^M 对 Cache 的执行过程进行抽象分析，得到各结点内存访问时的 Cache 状态集合记录 $listCS^M$. 若采用不同分析方法如 Must 分析或 May 分析，将得到不同的 Cache 状态集合记录 $listCS^{\text{Must}}$ 和 $listCS^{\text{May}}$，最后对比当前的内存块访问和当前 Cache 状态集合，即可得到 Cache 命中情况标记.

本文将在第 6 节中介绍分析过程应用的详细步骤，并以案例分析的形式阐述 Cache 分析的应用过程.

4.2.3 Cache 命中情况标记方法定义

对 Cache 命中情况标记，可给出定义：

$$MAC^M=(S_{MA},listCS^M,AC^M_{\text{label}}),$$

其中，S_{MA} 为内存访问序列；$listCS^M$ 为根据分析方法 CA^M 得到的 Cache 行为历史列表；AC^M_{label} 为对比当前内存访问和当前 Cache 状态集合得到的访问情况标记(Access Category)，通常可标记为 always hit、always miss 或 not classified.

命中情况标记操作简单，一般仅作当前内存块访问和当前 Cache 状态集合的对比，本文将在第 6.2 小节的最后部分简要介绍该过程.

在基于静态分析技术的 Cache 行为分析中，通常采用一种分析方法并不能得到全部 Cache 命中的标记信息，如采用 Must 分析一般用于标注 always hit，而采用 May 分析一般用于标注 always miss. 因此为得到全面的 Cache 标注信息，可能需要采用多种分析方法结合的形式；同时，在某些交换策略中，由于其结构的特殊设计，可能造成部分分析方法失效，如 Pseudo-Round-Robin 策略会使得 May 分析失效. 所以对于不同的 Cache 机制，应遵循其机制的特殊性，有针对性的采用不同分析方法进行 Cache 行为分析，本文不对此进行展开描述.

5 分析方法设计

为进行 Cache 行为分析，在分析方法设计中首先需要对 Cache 状态及交换策略进行建模和表达，本文采用基于抽象解释理论的方法对 Cache 状态和状态更新进行建模. 根据 4.2.1 小节中的定义，该方法从语义的角度对实际 Cache 状态和状态更新进行抽象表达，并通过抽象状态的迭代更新模拟真实执行过程中的状态变化，最终依据抽象状态验证达到某程序点时所关注的内存块是否在 Cache 内. 该建模方法采用过近似的方式对真实状态进行抽象，因此经过抽象状态验证的属性在真实的 Cache 行为中一定能够得到保持.

5.1 Cache 架构描述

由于在不同硬件平台中 Cache 的设计和描述方式较为统一，并且其结构简单、行为单一的特性，使得 Cache 行为的模块化分析具有一定的可行性. 为进行 Cache 行为分析，首先需要对各种类型的 Cache 架构进行统一化描述，其架构描述中主要包括结构配置和交换策略两方面的内容.

5.1.1 Cache 结构配置

Cache 主要通过三个重要参数描述其结构[19]：容量、块长和地址映射. 容量 $capacity$ 表示一个 Cache 总的可存储空间大小；块长 $line\ size$ 表示在内存和 Cache 进行交换时的连续字节数，每个 Cache 最多可有 $n=capacity/line\ size$ 个字块；地址映射表示一个内存块在 Cache 中的特定映射地址，通过地址映射可以将 Cache 块和内存块建立联系，

若 Cache 块与 CPU 将要访问的内存地址建立了对应关系，则 CPU 可以直接访问 Cache 块中的内存内容，即 Cache 命中，若 CPU 将要访问的内存地址未找到对应的 Cache 块，则 CPU 访问时需要从内存中重新读取，即 Cache 丢失.

Cache 结构中容量和块长两种参数仅仅只是限定了与内存交换过程中的最大可使用容量和单次交换的字块大小，对 Cache 状态描述起决定性作用的是地址映射. 现有 Cache 结构中存在三种地址映射方式：直接映射、全相联映射、组相联映射. 直接映射就是每个内存块只与一个 Cache 块相对应，映射结果为每个 Cache 块将对应多个内存块；全相联映射则可以将内存中每一个字块映射到 Cache 的任何块中；组相联映射是直接映射和全相联映射的折中，也是目前应用最多的地址映射方式，它将 Cache 分为多个组，内存块按组放入 Cache 中.

对 Cache 结构配置进行建模来描述 Cache 状态即是对地址映射关系的建模与表达，本文以 A 路组相联 Cache 为例介绍如何对 Cache 状态进行形式化描述（全相联映射和直接映射可以表达为 $A=n$ 和 $A=1$ 的特殊情况）. 对于 A 路组相联映射的 Cache，可将其看作为多个全相联 Cache 组成的集合序列 $F=\langle f_1,\cdots,f_{n/A}\rangle$，其中每个集合 f_i 又是字块的一个集合序列 $L=\langle l_1,\cdots,l_A\rangle$. 而对于内存则可描述为内存块的集合 $M=\{m_1,\cdots,m_s\}$，其中 s 表示内存块的个数[18].

定义内存取址函数 $adr:M\to\mathbb{N}_0$，该函数用于计算每个内存块的内存地址. 定义 *set* 函数 $set:M\to F$ 计算内存块将会储存于 Cache 中的哪一组：

$$set(m)=f_i;\ \text{where } i=adr(m)\%(n/A)+1.$$

以元素 I 表示在某一 Cache 字块中未放入任何内存块，即 Cache 块为空的状态，扩展内存块集合的表达能力，此时有 $M'=M\cup\{I\}$ 表示所有可能的内存块集合.

为表达 Cache 中各组的状态，定义 Cache 状态 *cache state* 来描述整体 Cache 中的所有组的情况；对于各组中的 Cache 块状态，定义组状态 *set state* 来描述在同一组中 Cache 块的状态.

此时可将组状态 *set state* 描述为以下形式 $s:L\to M$，其中

$$\forall l_a,l_b\in L:s(l_a)=s(l_b)\Rightarrow s(l_a)=s(l_b)=I\vee l_a=l_b.$$

上式表示在同一组 Cache 中，若存在两个 Cache 块对应的内存块相同的情况，要么 Cache 块为空或者两个 Cache 块相同，即任何一个内存块只可对应到 Cache 中某组的其中一个位置.

对于 Cache 状态，定义其描述函数 $c:F\to S$，其中

$$\forall f_y\in F:\forall l_x\in L:c(f_y)(l_x)\neq I\Rightarrow set(c(f_y)(l_x))=f_y.$$

上式表示任何一个内存块永远只能对应到由其地址决定的 Cache 组中.

5.1.2　Cache 交换策略

通过统一化的 Cache 架构描述，可以对其真实执行状态进行完整的表达，该架构仅描述了 Cache 的真实模型，而用于描述 Cache 行为的抽象分析模型应当依托于该真实模型进行设计和分析.

在 Cache 中交换策略决定了 Cache 块如何与内存块进行交换，即通过交换策略完成 Cache 状态的更新. 不同的交换策略决定了 Cache 状态更新函数的设计，因此针对模块化 Cache 行为分析，交换策略将指导不同情况下的状态更新函数设计与使用.

交换策略的目的是使得硬件平台的整体平均性能最好，即在大多数情况下，程序执行的平均时间最优，由此带来的问题是最坏执行情况的判别和检测较为困难. 如何对最坏执行情况下的 Cache 命中进行判断，必须严格考虑交换策略对 Cache 状态的影响. 常用嵌入式系统中的 Cache 交换策略主要有以下三种：LRU、ColdFire 平台中的 Pseudo-Round-Robin、PowerPC 平台中的 Pseudo-LRU[30]. 针对不同交换策略需要对分析过程设计不同的状态更新函数，本文将在第 5.2 节、5.3 节以 LRU 策略的 Must 分析和 May 分析中为例进行详细介绍.

5.2　LRU 策略下 Must 分析

在 5.1.1 子小节中，本文介绍了如何对实际 Cache 状态进行建模描述，通过形式化的语义描述，对于统一描述的 Cache 结构，为进行相应的行为分析，还需要针对不同的分析目的设计不同的分析方法. 本节将以 Must 分析为例，详细介绍如何设计分析过程.

为分析程序达到某一结点时，CPU 所需访问的内存块是否一定存在于 Cache 中，可采用 Must 分析的方式对 Cache 状态更新进行建模与表达. Must 分析以一定存在于 Cache 中的内存块集合作为分析目标，通过结合不同的交换策略设计实际状态和抽象状态下的转换函数，以进行状态更新. 最终以一定存在于 Cache 中的可能状态集合进行结果分析，验证达到程序某一结点时，所需访问的内存块是否存

在于该可能状态的集合中.

以 LRU 交换策略为例,该策略将最近一段时间未有访问的 Cache 块内容交换出 Cache,并以 Cache 块年龄信息标注该块的访问历史.本节将详细介绍如何对采用 LRU 交换策略的 Cache 行为进行建模分析.

针对实际 Cache 状态,前文在 5.1.1 子小节中已有描述,通过多个全相联 Cache 组成的集合序列 $F=\langle f_1,\cdots,f_{n/A}\rangle$ 表示 A 路组相联 Cache,以集合元素 f_i 表示字块的集合序列 $L=\langle l_1,\cdots,l_A\rangle$. 内存则表达为内存块的集合 $M=\{m_1,\cdots,m_s\}$,其中以 s 表示所有内存块的个数. Must 分析以针对 Cache 中每一组 Cache 块的状态更新为单元.

为表示每一次存在内存块访问时的该组 Cache 块更新函数,可定义组状态更新函数 $\mathcal{U}_S:S\times M\to S$ 如下:

$$\mathcal{U}_S(s,m)=\begin{cases}[l_1\mapsto m,l_i\mapsto s(l_{i-1})\mid i=2,\cdots,h,\\ \quad l_i\mapsto s(l_i)\mid i=h+1,\cdots,A], & \text{如果}\exists l_h:s(l_h)=m.\\ [l_1\mapsto m,l_i\mapsto s(l_{i-1})\mid i=2,\cdots,A], & \text{其他}\end{cases}$$

该函数表示,当存在一个内存块访问时,首先判断该内存块是否在 Cache 内,若存在,先找出该 Cache 块年龄 h,更新时将年龄小于 h 的块所有年龄加 1,年龄大于 h 的块保持不动,同时将该访问的内存块年龄置为 1;若内存块不在 Cache 内,则将 Cache 中所有块年龄加 1,该访问内存块年龄置为 1.

定义 Cache 状态更新函数 $\mathcal{U}_C:C\times M\to C$ 为

$$\mathcal{U}_C(c,m)=c[set(m)\mapsto\mathcal{U}_S(c(set(m)),m)].$$

通过两种状态更新函数,即可对实际 Cache 的组状态和 Cache 状态进行状态转换,完成整个 Cache 的更新操作,但由于分析过程需要对所有可能的 Cache 状态进行分析,因此单纯使用实际 Cache 状态无法表达出状态的可能性,在此需要引入抽象的概念,以抽象语义表达所有可能状态,即在 Cache 行为分析框架中对 AC_S 的展开定义.

定义抽象组状态 *abstract set state* $\hat{s}:L\to 2^M$ 将一组 Cache 块映射到可能的内存块集合,其中:

$$\forall l_a,l_b\in L:\forall m\in M:m\in(\hat{s}(l_a)\bigcap\hat{s}(l_b))\Rightarrow l_a=l_b.$$

上式表示某一个内存块仅可存在于一个抽象的 Cache 组中.

定义抽象 Cache 状态 *abstract cache state* $\hat{c}:F\to\hat{S}$ 表示抽象的 Cache 组集合,其中:

$$\forall f_y\in F:\forall l_x\in L:\forall m\in M:m\in\hat{c}(f_y)(l_x)\Rightarrow set(m)=f_y.$$

由于使用 Cache 的可能状态集合来表示内存访问时的 Cache 块储存情况,因此当存在一个内存块 m 被访问时,若该内存块已经存在于 Cache 内,使用 Must 分析的抽象更新步骤为:(1) 当前访问内存块年龄置为 1;(2) 将访问时年龄小于 $h-1$ 的内存块状态集合依次后移一位,即其年龄依次加 1;(3) 从年龄为 h 的内存块状态集合 $\hat{s}(l_h)$ 中移除内存块 m,与原来年龄为 $h-1$ 的 Cache 块状态集合 $\hat{s}(l_{h-1})$ 合并,然后置于年龄为 h 的 Cache 块状态集合中;(4) 将当前内存块年龄 h 之后的状态集合保持不动.

对 Cache 行为分析框架中以 Must 分析方法的 AC_U 展开定义,即定义抽象组状态更新函数 $\hat{\mathcal{U}}_{\hat{S}}^{\text{must}}$,表示当存在一次内存访问时,组内可能的 Cache 块集合抽象更新情况,函数定义如下:

$$\hat{\mathcal{U}}_{\hat{S}}^{\text{must}}(\hat{s},m)=\begin{cases}[l_1\mapsto\{m\},l_i\mapsto\hat{s}(l_{i-1})\mid i=2,\cdots,h-1,\\ \quad l_h\mapsto\hat{s}(l_{h-1})\bigcup(\hat{s}(l_h)-\{m\}),\\ \quad l_i\mapsto\hat{s}(l_i)\mid i=h+1,\cdots,A], & \text{如果}\exists l_h:m\in\hat{s}(l_h)\\ [l_1\mapsto\{m\},l_i\mapsto\hat{s}(l_{i-1})\mid i=2,\cdots,A], & \text{其他}\end{cases}.$$

可定义抽象 Cache 状态更新函数 $\hat{\mathcal{U}}_{\hat{C}}^{\text{must}}$ 如下:

$$\hat{\mathcal{U}}_{\hat{C}}^{\text{must}}(\hat{c},m)=\hat{c}[set(m)\mapsto\hat{\mathcal{U}}_{\hat{S}}^{\text{must}}(\hat{c}(set(m)),m)].$$

对抽象的 Cache 组状态更新举例如表 1 所示.

表 1 Must 分析更新函数举例

	l_1	l_2	l_3	l_4
$\hat{s}$	$\{m_a\}$	$\{\}$	$\{m_b,m_c\}$	$\{m_d\}$
$\hat{\mathcal{U}}_{\hat{S}}^{\text{must}}(\hat{s},m_c)$	$\{m_c\}$	$\{m_a\}$	$\{m_b\}$	$\{m_d\}$

对于一个 4 路组相联 Cache 考虑某一组内的 Cache 块更新情况,当前状态如 $\hat{s}$ 所示. 此时,若存在一个内存块访问 m_c,根据上述组状态更新函数对状态 $\hat{s}$ 进行更新,如 $\hat{\mathcal{U}}_{\hat{S}}^{\text{must}}(\hat{s},m_c)$ 所示.

在使用抽象状态进行分析时,由于并非是程序的真实执行情况,因此还需考虑对两种抽象状态进行合并操作以表达程序的分支结点合并情况. 在 LRU 策略的 Must 分析中,对于任何可能状态的合并集合,应保留在两个结点中都出现的 Cache 块的最大年龄作为合并后该 Cache 块的可能年龄,对 Cache 行为分析框架中以 Must 分析方法的 AC_J 展开定义,即定义组状态的 Join 函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}$ 如下:

$$\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}(\hat{s}_1,\hat{s}_2)=\hat{s},\text{当}$$

$$\hat{s}(l_x)=\{m\mid\exists l_a,l_b\text{有}\ m\in\hat{s}_1(l_a),m\in\hat{s}_2(l_b)\text{且}$$

$x=\max(a,b)\}$.

对整体 Cache 状态其 Join 函数 $\hat{\mathcal{J}}_{\hat{C}}^{\text{must}}$ 定义如下：

$$\hat{\mathcal{J}}_{\hat{C}}^{\text{must}}(\hat{c}_1,\hat{c}_2)=$$
$$[f_i\mapsto\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}(\hat{c}_1(f_i),\hat{c}_2(f_i))\mid\forall i,1\leq i\leq n/A].$$

对两个抽象的 Cache 组状态进行合并操作，其 Join 函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}$ 举例如表 2 所示.

表 2　Must 分析状态合并函数举例

	l_1	l_2	l_3	l_4
$\hat{s}_1$	$\{m_a\}$	$\{m_b\}$	$\{m_c\}$	$\{m_d\}$
$\hat{s}_2$	$\{m_c\}$	$\{m_e\}$	$\{m_a\}$	$\{m_d\}$
$\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}(\hat{s}_1,\hat{s}_2)$	$\{\}$	$\{\}$	$\{m_a,m_c\}$	$\{m_d\}$

在一个 4 路组相联 Cache 中，若某一程序点存在两种 Cache 状态分支需要合并，根据状态合并操作的 Join 函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}(\hat{s}_1,\hat{s}_2)$，将两个抽象 Cache 状态进行合并如表 2 所示.

Must 分析过程可根据其状态更新函数 $\hat{\mathcal{U}}_{\hat{S}}^{\text{must}}$ 和状态合并函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{must}}(\hat{s}_1,\hat{s}_2)$ 完成，该分析结果将给出每个程序结点的可能 Cache 状态集合. 当程序执行到某一结点时，对比当前将要访问的内存块 m 是否存在于 Must 分析给出的可能状态集合 $\hat{s}$ 中，若存在，则对于内存块 m 的访问将标记为 always hit；若不存在，则对内存块 m 的访问将不做任何标记.

5.3　LRU 策略下 May 分析

通过 Must 分析，可以得到内存块访问是否具有 always hit 的标记，而为了找出所有 always miss 的内存块访问，则需要通过 May 分析.

与 Must 分析相似，在 May 分析中以同样的方式表示 Cache 状态，通过多个全相联 Cache 组成的集合序列 $F=\langle f_1,\cdots,f_{n/A}\rangle$ 表示 A 路组相联 Cache，以集合元素 f_i 表示字块的集合序列 $L=\langle l_1,\cdots,l_A\rangle$. 内存表达为内存块的集合 $M=\{m_1,\cdots,m_s\}$，其中以 s 表示所有内存块的个数. May 分析也以针对 Cache 中每一组 Cache 块的状态更新为单元.

通过两种状态更新函数 $\mathcal{U}_S$ 和 $\mathcal{U}_C$，对实际 Cache 的组状态和 Cache 状态进行状态转换，完成整个 Cache 的更新操作. 与 Must 分析的区别在于，使用 May 分析中，抽象 Cache 状态集合的更新函数需要做出相应调整，在 Must 分析中记录的 Cache 状态集合表示的是一定存在于 Cache 中的内存块，而 May 分析中保存的 Cache 状态集合表示的是可能存在于 Cache 中的内存块. 在 May 分析的抽象状态更新函数设计中，当存在一个内存块 m 被访问时，若该内存块已经存在于 Cache 内，使用 May 分析的抽象更新步骤为：(1) 当前访问内存块年龄置为 1；(2) 将访问时年龄小于 h 的内存块状态集合依次后移一位，即其年龄依次加 1；(3) 从原来年龄为 h 的内存块状态集合 $\hat{s}(l_h)$ 中移除内存块 m，与原来年龄为 $h+1$ 的 Cache 块状态集合 $\hat{s}(l_{h+1})$ 合并，然后置于年龄为 $h+1$ 的 Cache 块状态集合中；(4) 将当前内存块年龄 h 之后的状态集合保持不动.

对 Cache 行为分析框架中以 May 分析方法的 AC_U 展开定义，即定义抽象组状态更新函数 $\hat{\mathcal{U}}_{\hat{S}}^{\text{may}}$，表示当存在一次内存访问时，组内可能的 Cache 块集合抽象更新情况，函数定义如下：

$$\hat{\mathcal{U}}_{\hat{S}}^{\text{may}}(\hat{s},m)=$$
$$\begin{cases}[l_1\mapsto\{m\},l_i\mapsto\hat{s}(l_{i-1})\mid i=2,\cdots,h,\\ l_{h+1}\mapsto\hat{s}(l_{h+1})\bigcup(\hat{s}(l_h)-\{m\}),\\ l_i\mapsto\hat{s}(l_i)\mid i=h+2,\cdots,A], & \text{如果}\exists l_h:m\in\hat{s}(l_h)\\ [l_1\mapsto\{m\},l_i\mapsto\hat{s}(l_{i-1})\mid i=2,\cdots,A], & \text{其他}\end{cases}.$$

可定义抽象 Cache 状态更新函数 $\hat{\mathcal{U}}_{\hat{C}}^{\text{may}}$ 如下：

$$\hat{\mathcal{U}}_{\hat{C}}^{\text{may}}(\hat{c},m)=\hat{c}[set(m)\mapsto\hat{\mathcal{U}}_{\hat{S}}^{\text{may}}(\hat{c}(set(m)),m)].$$

对抽象的 Cache 组状态更新举例如表 3 所示.

表 3　May 分析更新函数举例

	l_1	l_2	l_3	l_4
$\hat{s}$	$\{m_a\}$	$\{m_b,m_c\}$	$\{\}$	$\{m_d\}$
$\hat{\mathcal{U}}_{\hat{S}}^{\text{may}}(\hat{s},m_c)$	$\{m_c\}$	$\{m_a\}$	$\{m_b\}$	$\{m_d\}$

对于一个 4 路组相联 Cache 考虑某一组内的 Cache 块更新情况，当前状态如 $\hat{s}$ 所示. 此时，若存在一个内存块访问 m_c，根据上述组状态更新函数对状态 $\hat{s}$ 进行更新，如 $\hat{\mathcal{U}}_{\hat{S}}^{\text{may}}(\hat{s},m_c)$ 所示.

与 Must 分析类似，在 May 分析中也需考虑对两种抽象状态进行合并操作以表达程序的分支结点合并情况. 区别于 Must 分析，在 May 分析中，对于任何可能状态的合并集合，首先应保留在两个结点中都出现的 Cache 块的最小年龄作为合并后该 Cache 块的可能年龄；其次由于 May 分析表示的是所有可能状态集合，因此对于存在于某一结点而另一结点中未出现的内存块状态也应合并到等于该内存块年龄的抽象 Cache 状态集合中. 对 Cache 行为分析框架中以 May 分析方法的 AC_J 展开定义，即定义组状态的 Join 函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}$ 如下：

$\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}(\hat{s}_1,\hat{s}_2)=\hat{s}$，当

$\hat{s}(l_x)=$

$\{m\mid\exists l_a,l_b$ 有 $m\in\hat{s}_1(l_a),m\in\hat{s}_2(l_b)$ 且 $x=\min(a,b)\}\cup$

$\{m\mid m\in\hat{s}_1(l_a)$ 且 $\nexists l_a$ 满足 $m\in\hat{s}_2(l_a)\}\cup$

$\{m\mid m\in\hat{s}_2(l_a)$ 且 $\nexists l_a$ 满足 $m\in\hat{s}_1(l_a)\}$.

对于 Cache 状态其 Join 函数 $\hat{\mathcal{J}}_{\hat{C}}^{\text{may}}$ 定义如下：

$\hat{\mathcal{J}}_{\hat{C}}^{\text{may}}(\hat{c}_1,\hat{c}_2)=$

$[f_i\mapsto\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}(\hat{c}_1(f_i),\hat{c}_2(f_i))\mid\forall i,1\leq i\leq n/A]$.

对两个抽象的 Cache 组状态进行合并操作，其 Join 函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}$ 举例如表 4 所示.

表 4 May 分析状态合并函数举例

	l_1	l_2	l_3	l_4
$\hat{s}_1$	$\{m_a\}$	$\{m_b\}$	$\{m_c\}$	$\{m_d\}$
$\hat{s}_2$	$\{m_c\}$	$\{m_e,m_f\}$	$\{m_a\}$	$\{m_d\}$
$\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}(\hat{s}_1,\hat{s}_2)$	$\{m_a,m_c\}$	$\{m_b,m_e,m_f\}$	$\{\}$	$\{m_d\}$

在一个 4 路组相联 Cache 中，若某一程序点存在两种 Cache 状态分支需要合并，根据状态合并操作的 Join 函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}(\hat{s}_1,\hat{s}_2)$，将两个抽象 Cache 状态进行合并如表 4 所示.

根据状态更新函数 $\hat{\mathcal{U}}_{\hat{S}}^{\text{may}}$ 和状态合并函数 $\hat{\mathcal{J}}_{\hat{S}}^{\text{may}}(\hat{s}_1,\hat{s}_2)$ 即可完成 May 分析过程. 当程序执行到某一结点时，对比当前将要访问的内存块 m 是否存在于 May 分析给出的可能状态集合 $\hat{s}$ 中，若不存在，则对于内存块 m 的访问将标记为 always miss；若存在，则对内存块 m 的访问将不做任何标记.

6 分析过程应用

通过 Cache 行为分析方法设计，可以对 Cache 行为进行建模，用模型的方式对 Cache 的可能状态进行表达. 为对 Cache 行为进行分析，需要对其分析过程进行设计及应用，本节将根据 4.2.2 小节和 4.2.3 小节的定义，介绍如何开展 Cache 分析过程以对内存访问中的 Cache 命中情况进行分析.

6.1 分析过程简述

在 Cache 行为分析中，主要采用三种分析方法：Must 分析、May 分析和 Persistence 分析，通过三种不同的分析方式，可以为内存访问状态标记不同的 Cache 命中情况：ah、am、nc. 对于 Must 分析和 May 分析并不存在先后顺序，而 Persistence 分析则通常用于在 Must 和 May 分析结束之后的第二轮精确结果分析.

使用 Must 分析能够将所有一定存在于 Cache 中的可能内存块放入可能状态集合，此时 Must 状态集合中所包含的内存块在当前内存访问时一定存在于 Cache 中，即若存在一次内存块访问，且该内存块存在于 Must 集合中，则该次内存访问一定命中.

使用 May 分析能够将所有可能存在于 Cache 中的可能内存块放入可能状态集合，May 状态集合中包含在当前内存访问时所有可能存在于 Cache 中的内存块. 反之，若存在一次内存访问，且该内存块不在 May 集合中，即该内存块不可能在 Cache 中，则该次内存访问一定不命中.

通过 Must 分析和 May 分析可以得到大部分顺序执行中的程序结点内存访问情况，而在循环或者嵌套结构中，Cache 状态可能会陷入局部一直命中，使用 Must 分析和 May 分析则无法分析出局部一直命中的结点，影响分析精度. 因此 Persistence 分析作用在 Must 分析和 May 分析之后，用于进一步分析循环或者嵌套结构中的程序结点内存访问情况.

Persistence 分析将首次循环和其他循环进行分别处理，首次循环仍然按照 Must 分析和 May 分析进行处理，在对第二次及其他循环处理时，将其展开为多次顺序执行的形式，依次对多个循环展开进行状态更新，在 Cache 状态收敛或满足终止条件后，对循环体中位于同一程序结点的 Cache 状态进行合并，得到 Persistence 状态集合. 根据 Persistence 状态集合，可以进一步精确循环或嵌套结构中的内存访问 Cache 命中情况.

Must 分析和 May 分析的 Cache 行为分析过程如下：

(1) 将分析方法中的状态更新函数 Update 和状态合并函数 Join 根据其数学表达转换为算法程序；

(2) 根据提供的程序结点内存访问序列，按照内存访问顺序依次更新对应 Cache 组中的 Cache 可能状态集合；

(3) 当程序结点存在合并时，使用状态合并函数对多种可能的 Cache 状态集合进行合并；

(4) 得到不同程序结点时的 Cache 状态可能集合；

(5) 对比当前访问的内存块是否在 Cache 状态

可能集合内，标记其 Cache 命中情况.

6.2　案例分析

给定一个 4 路组相联 LRU 交换策略的 Cache，以组为单位进行其中某一组 Cache 块的 Must 分析和 May 分析，以 $..x..$ 表示内存块 x 当前被访问，程序中的内存访问顺序 S_{MA} 如图 5 所示.

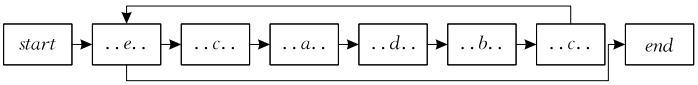

图 5　案例程序内存块访问顺序图

分别给定在 *start* 结点的初始 Must 分析和 May 分析抽象状态 *initCS* 如表 5 所示.

表 5　May 分析和 Must 分析初始抽象状态

	l_1	l_2	l_3	l_4
may	$\{b,e\}$	$\{d,z\}$	$\{\}$	$\{\}$
must	$\{\}$	$\{\}$	$\{b,d\}$	$\{e,z\}$

通过 Matlab 模拟实现相关分析过程的状态更新和合并算法，分别对案例中访问内存块的各个程序点进行 Must 分析和 May 分析，可得到各结点分析结果列表 *listCS*，如图 6 所示，根据各结点状态进行进一步比对，即可得到该结点的访问命中情况.

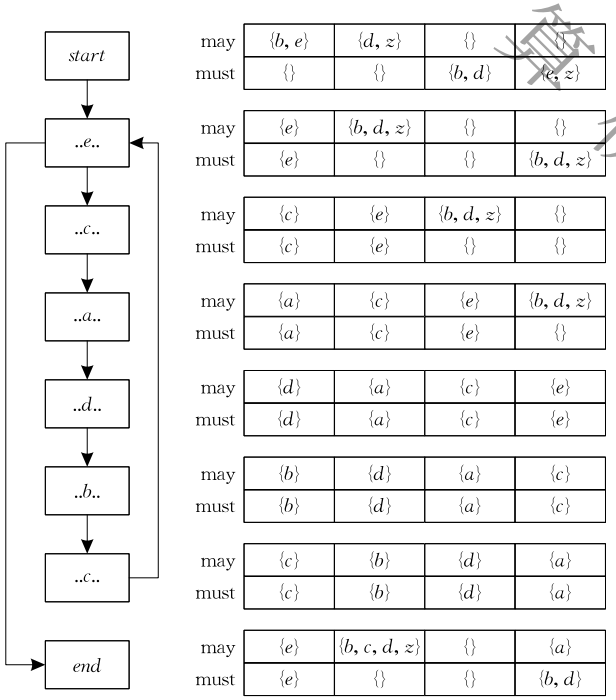

图 6　案例中各结点 May 分析和 Must 分析状态

为标记当前程序结点的访问命中情况 AC^{M}_{label}，根据框架定义中 Cache 命中标记方法 MAC^{M}，需对比当前结点访问的内存块和前一结点访问更新后的 May 分析及 Must 分析结果，若当前访问内存块在上一结点的 Must 集合中，则表示该结点访问 ah；若当前访问内存块不在上一结点的 May 集合中，则表示该结点访问 am；对于不在 Must 集合而在 May 集合中的访问结点则标记为 nc. 案例中各结点 Cache 命中情况标记如表 6 所示.

表 6　每个程序结点 Cache 命中情况标记

..e..	..c..	..a..	..d..	..b..	..c..
ah	am	am	nc	am	ah

本小节以案例形式说明了 Cache 行为分析的基本流程，以 LRU 策略的 Cache 为例，主要介绍了 Must 分析和 May 分析两种分析方法的设计和使用，并得到了有效结果. 在 WCET 估算过程中，可通过赋予 ah 和 am 标记的内存块不同的 CPU 等待时间惩罚，用于计算在 Cache 交换过程中，该段程序产生的 Cache 与内存交换的时间消耗，进而精确整个 WCET 估算结果.

7　总结与展望

嵌入式系统存在硬件平台多样的特点，为相关软件系统的执行时间属性分析增加了难度与挑战，一些成熟的时间属性静态分析方法可能无法直接应用或难以重用到所有需要进行时间分析的平台中. 本文从易于复用和重构的角度出发，提出了一种基于抽象解释的模块化 Cache 行为分析框架，首先以抽象解释理论为分析方法设计基础，对 Cache 行为的执行过程进行抽象建模；然后对 Cache 架构和分析方法进行关联分析，形式化定义了 Cache 行为分析过程；最后，构建模块化分析框架以支持多种不同架构的 Cache 行为分析，从而达到 Cache 分析过程可重用的目的.

本文以适用性为目标，重点关注了如何对 Cache 行为分析过程进行模块化设计，确定 Cache 配置、分析方法和分析过程间的关联关系，提出了各层次间的阶段划分结构，构建了独立于 Cache 机制的 Cache 行为分析框架. 本文所提框架是以较粗粒度来描述 Cache 行为分析的整体过程以及各阶段功能和层次划分，并不针对特定 Cache 机制展开具体详细分析方法和过程设计，因此在针对确定 Cache 机制进行相关分析和工具实现时，需对框架中的分

析方法和分析过程进行具体实例化，如针对 ColdFire MCF 5307 平台进行 Cache 分析，其使用的 Pseudo-Round-Robin 交换策略采用了一个额外的全局计数器来确定内存访问不命中时组内交换块的地址，因此需要针对该架构实例化出具体的分析方法（额外的抽象全局计数器状态描述）和对应的分析流程（该架构下 May 分析失效）.

目前学术界对 WCET 估算方面主要存在两种研究思路[35]：(1) 延续传统的 WCET 计算方法并进行改进扩展，主要针对处理器架构的执行过程进行建模，一类研究力求改进分析方法，从分析精度的角度出发对 WCET 进行估算，如针对分析工具的进一步优化[36]，针对硬件架构如 Cache 行为进行改进的 Persistence 分析[37]，本文即是从该角度出发，提出一种可复用的 Cache 行为分析框架；(2) 另一类研究从处理器硬件架构的建模出发，精细化分析方法颗粒度，对硬件执行环境进行深入建模，如针对 Pipeline 机制的抽象解释建模[24-25]；设计可进行执行时间预测的硬件架构，并在此基础上进行各类执行时间分析，降低分析难度，如针对 Patmos 架构进行的 WCET 分析技术[38].

针对 WCET 估算的研究需要考虑较多因素，本文仅从 Cache 行为建模的角度，对内存访问中的时间问题进行研究，同时也存在一些尚未完成的研究亟需探索：(1) 区别于指令 Cache，数据 Cache 可能存在一些 IO 操作，对于如何将数据 Cache 和指令 Cache 在行为建模上的不同完善到该框架设计仍需探索；(2) 静态程序分析技术中对于循环和嵌套结构需要从控制流结构进行分析，而用于处理循环和嵌套结构的 Persistence 分析需要单独针对其控制流结构进行特殊处理，如何将控制流结构整合进本框架也需要深入研究；(3) 在现代架构的 CPU 中 Pipeline 行为与 Cache 行为紧密相关，流水线处理和 Cache 机制交互更为紧密，对于如何整合该框架下的 Cache 分析与 Pipeline 行为分析的一体化建模也是本文后续研究的重点.

致 谢 在本文撰写和修改过程中，审稿专家和编辑老师提出了许多宝贵意见，在此表示衷心感谢！

参 考 文 献

[1] Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem—overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 2008, 7(3): 36

[2] Lü Ming-Song, Guan Nan, Wang Yi. Survey of Cache analysis for worst-case execution time estimation. Journal of Software, 2014, 25(2): 179-199(in Chinese)
(吕鸣松，关楠，王义. 面向 WCET 估计的 Cache 分析研究综述. 软件学报，2014，25(2)：179-199)

[3] Ji Meng-Luo, Li Jun, Wang Xin, Qi Zhi-Chang. An automatic WCET analysis tool based on abstract interpretation. Computer Engineering, 2006, 32(14): 54-56(in Chinese)
(姬孟洛，李军，王馨，齐治昌. 一种基于抽象解释的 WCET 自动分析工具. 计算机工程，2006，32(14)：54-56)

[4] Cofer D, Miller S. DO-333 certification case studies// Proceedings of the NASA Formal Methods: 6th International Symposium. Houston, USA, 2014: 1-15

[5] Huang Zhi-Qiu, Xu Bing-Feng, Kan Shuang-Long, et al. Survey on embedded software safety analysis standards, methods and tools for airborne system. Journal of Software, 2014, 25(2): 200-218(in Chinese)
(黄志球，徐丙凤，阚双龙等. 嵌入式机载软件安全性分析标准、方法及工具研究综述. 软件学报，2014，25(2)：200-218)

[6] Li Meng-Jun, Li Zhou-Jun, Chen Huo-Wang. Program verification techniques based on the abstract interpretation theory. Journal of Software, 2008, 19(1): 17-26(in Chinese)
(李梦君，李舟军，陈火旺. 基于抽象解释理论的程序验证技术. 软件学报，2008，19(1)：17-26)

[7] Thesing S, Souyris J, Heckmann R, et al. An abstract interpretation-based timing validation of hard real-time avionics software//Proceedings of the International Conference on Dependable Systems and Networks. San Francisco, USA, 2003: 625-632

[8] Silva e R A B, Arai N N, Burgareli L A, et al. Formal verification with frama-C: A case study in the space software domain. IEEE Transactions on Reliability, 2016, 65(3): 1163-1179

[9] Cousot P, Cousot R, Feret J, et al. Varieties of static analyzers: A comparison with astrée//Proceedings of the 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. Shanghai, China, 2007: 3-17

[10] Cousot P, Cousot R, Feret J, et al. The astrée analyzer// Sagic M ed. Programming Languages and Systems. Berlin, Germany: Springer-Verlag, 2005: 21-30

[11] Mauborgne L. Astrée: Verification of absence of run-time error//Jacquart R ed. Building the Information Society. Boston, Germany: Springer, 2004: 385-392

[12] Ferdinand C, Heckmann R. aiT: Worst case execution time prediction by static program analysis//Jacquart R ed. Building the Information Society. Boston, Germany: Springer, 2004: 377-383

[13] Bonenfant A, Cassé H, Michiel de M, et al. FFX: A portable WCET annotation language//Proceedings of the 20th International Conference on Real-Time and Network Systems. New York, USA, 2012: 91-100

[14] Ballabriga C, Cassé H, Rochange C, Sainrat P. OTAWA: an open toolbox for adaptive WCET analysis//Proceedings of the Software Technologies for Embedded and Ubiquitous Systems, Waidhofen/Ybbs, Austria, 2010: 35-46

[15] Cassé H and Sainrat P. OTAWA, a framework for experimenting WCET computations//Proceedings of the European Congress on Embedded Real-Time Software. Toulouse, France, 2005: 25-27

[16] Alt M, Ferdinand C, Martin F, Wilhelm R. Cache behavior prediction by abstract interpretation in static analysis//Proceedings of the International Static Analysis Symposium, Aachen, Germany, 1996: 52-66

[17] Ferdinand C. Cache Behavior Prediction for Real Time Systems [Ph. D. dissertation]. Saarland University, Saarbrücken, Germany, 1997

[18] Ferdinand C, Martin F, Wilhelm R, Alt M. Cache behavior prediction by abstract interpretation. Science of Computer Programming, 1999, 35(2-3): 163-189

[19] Ferdinand C, Wilhelm R. Efficient and precise cache behavior prediction for real-time systems. Real-Time Systems, 1999, 17(2): 131-181

[20] Kim S-K, Min S L, Ha R. Efficient worst case timing analysis of data caching//Proceedings of the IEEE Real-Time Technology and Applications Symposium. Boston, USA, 1996: 230-240

[21] Ballabriga C, Casse H, Sainrat P. An improved approach for set-associative instruction cache partial analysis//Proceedings of the ACM Symposium on Applied Computing. New York, USA, 2008: 360-367

[22] Cullmann C. Cache persistence analysis: A novel approach theory and practice//Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems. New York, USA, 2011: 121-130

[23] Huynh B K, Ju L, Roychoudhury A. Scope-aware data cache analysis for WCET estimation//Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium. Chicago, USA, 2011: 203-212

[24] Langenbach M, Thesing S, Heckmann R. Pipeline modeling for timing analysis//Proceedings of the International Static Analysis Symposium. London, UK, 2002: 294-309

[25] Thesing S. Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Modes [Ph. D. dissertation]. Saarland University, Saarbrücken, 2005

[26] Schlickling M, Pister M. A framework for static analysis of VHDL code//Proceedings of the International Workshop on Worst-Case Execution Time (WCET) Analysis. Palma de Mallorca, Spain, 2007: 1-6

[27] Ratsiambahotra T, Cassé H, Sainrat P. A versatile generator of instruction set simulators and disassemblers//Proceedings of the International Symposium on Performance Evaluation of Computer Telecommunication Systems. Istanbul, Turkey, 2009: 65-72

[28] Lv M, Yi W, Guan N, Yu G. Combining abstract interpretation with model checking for timing analysis of multicore software//Proceedings of the IEEE Real-Time Systems Symposium. San Diego, USA, 2010: 339-349

[29] Ferdinand C, Heckmann R, Langenbach M, et al. Reliable and precise WCET determination for a real-life processor//Proceedings of the International Conference/Workshop on Embedded Software. Tahoe City, USA, 2001: 469-485

[30] Heckmann R, Langenbach M, Thesing S, Wilhelm R. The influence of processor architecture on the design and the results of WCET tools. Proceedings of the IEEE, 2003, 91(7): 1038-1054

[31] Ferdinand C, Heckmann R, Wilhelm R. Analyzing the worst-case execution time by abstract interpretation of executable code//Proceedings of the Automotive Software Workshop. San Diego, USA, 2004: 1-14

[32] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints//Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages. Los Angeles, USA, 1977: 238-252

[33] Cousot P, Cousot R. Basic concepts of abstract interpretation//Jacquart R ed. Building the Information Society. Boston, Germany: Springer, 2004: 359-366

[34] Cullmann C. Cache Persistence Analysis for Embedded Real-Time Systems [Ph. D. dissertation]. Saarland University, Saarbrücken, Germany, 2013

[35] Yang Zhi-Bin, Zhao Yong-Wang, Huang Zhi-Qiu, et al. Time-predictable multi-threaded code generation with synchronous languages. Journal of Software, 2016, 27(3): 611-632(in Chinese)
(杨志斌，赵永望，黄志球等. 同步语言的时间可预测多线程代码生成方法. 软件学报，2016, 27(3): 611-632)

[36] Maiza C, Raymond P, Parent-Vigouroux C, et al. The W-SEPT project: Towards semantic-aware WCET estimation //Proceedings of the International Workshop on Worst-Case Execution Time Analysis. Dagstuhl, Germany, 2017: 1-13

[37] Zhang Z, Koutsoukos X D. Improving the precision of abstract interpretation based cache persistence analysis//Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems. Portland, USA, 2015: 1-10

[38] Brandner F, Naji A. Worst-case execution time analysis of predicated architectures//Proceedings of the International Workshop on Worst-Case Execution Time Analysis. Dagstuhl, Germany, 2017: 1-13

YU Yao-Shen, Ph. D. candidate. His research interests include software engineering, formal method, static program analysis, modeling and analysis of embedded systems.

HUANG Zhi-Qiu, Ph. D., professor. His research interests include software engineering, formal methods and cloud computing.

SHEN Guo-Hua, Ph. D., associate professor. His research interests include requirements engineering, software safety analysis, semantic web.

WANG Fei, Ph. D. candidate. His research interests include software engineering, formal methods, requirement engineering, modeling and analysis of embedded system.

CUI Shao-Xuan, M. S. candidate. Her research interests include software engineering, formal methods.

Background

Worst-case execution time (WCET) estimation is a basis for time validation in embedded system, especially in hard real-time systems. In aerospace, nuclear and transportation, some new standards require software developers provide the WCET verification and schedulability validation, which lead the WCET estimation need to be more precisely and formally for safety reason. In industrial applications, WCET of software are most directly measured from testing based on large number of test cases or monitoring by inject some system clock into the code. Most researches which based on static analysis consider the WCET work as a complete workflow include code analysis, hardware architecture modeling and path analysis. Their tools and analysis flow are precise enough but monopolistic for fixed code or employed systems. These methods can't be reused if executable code or employed hardware has been changed. This paper proposes a methodology for modular cache behavior analysis based on abstract interpretation, try to decompose WCET workflow level by level for the analysis process reuse. Based on the proposed framework, it can reuse the common elements and process in different cache behavior analysis, but also help developers easily design a fit analysis process for their specially systems. This paper only focused on cache which is a simply mechanism in processor, so it is just a little but necessary work for whole processor modeling.

This work is supported by the National High Technology Research and Development Program (863 Program) of China (2015AA105303), which is about the research on the theory and application of the formal method on the safety-critical domain, and the National Natural Science Foundation of China (61502231). This paper gives a methodology for modular cache analysis based on a formal method called abstract interpretation, and it can help for the application of static analysis in WCET estimation, also it will give more opportunities for industry to accept the use of formal method which is hard to understand or use.