计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(4): 1033−1049, 2025

最坏执行时间分析中的时序异常问题

蒋滨泽　朱钇宣　陈香兰　龚小航　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　215004）
（jiangbinze@mail.ustc.edu.cn）

Timing Anomalies Issue in WCET Analysis

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

Abstract　Timing anomalies are counter-intuitive behaviors observed in worst-case execution time (WCET) analysis. A key aspect of these anomalies is that a locally faster execution does not necessarily lead to a reduction in the overall program execution time. Therefore, WCET analysis must examine all possible execution states conservatively to ensure the safety of the analysis results, making the process extremely challenging. On the contrary, if it can be ensured that there are no timing anomalies in the program and platform to be analyzed, the number of states and the time required for WCET analysis can be significantly reduced. Consequently, addressing timing anomalies is a critical challenge in WCET analysis. However, despite more than 20 years of research, the academic community has not reached a unified definition and consensus on the problem of timing anomalies. This article reviews various perspectives from the literature since the concept of timing anomalies was first introduced. We classify these viewpoints based on their definitions and descriptions, and evaluate their respective strengths and weaknesses. Additionally, we investigate the causes of timing anomalies and identify three main factors: scheduling strategies, cache behavior, and component interactions. Furthermore, we explore current research efforts aimed at detecting and eliminating timing anomalies, highlighting the issues and limitations of these approaches. Finally, we suggest that future research on timing anomalies should be integrated with WCET analysis methods to more effectively address these challenges.

Key words　worst case execution time (WCET); processor architecture; timing anomaly; time predictable; real time system

摘　要　时序异常是一种程序在最坏执行时间（worst-case execution time, WCET）分析中反直觉的时间行为，其特征在于，尽管局部执行速度更快，但整体的执行时间可能会增加，因此 WCET 分析必须以保守的态度考虑所有可能的执行情况，以确保分析结果的安全性，这使得 WCET 分析变得非常困难. 相反地，如果能够确保要分析的程序与平台不存在时序异常，将大大减少 WCET 分析需要考虑的状态以及所花费的时间. 因此，时序异常问题是 WCET 分析中的关键挑战. 然而，尽管历经 20 多年的研究，学术界对时序异常问题仍未形成统一的定义和共识. 自时序异常概念提出以来，学术界涌现出了各种文献描述时序异常，

收稿日期：2023-12-14；**修回日期**：2024-08-14
基金项目：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）
This work was supported by the National Natural Science Foundation of China (62102383, 61976200, 62172380) and the Natural Science Foundation of Jiangsu Province (BK20210123).
通信作者：陈香兰（xlanchen@ustc.edu.cn）

对于这些定义,可以根据定义方式和描述内容对它们进行分类,并分析它们各自的优劣.进一步地,如果对导致时序异常的原因进行研究,可以将其归结为调度策略、高速缓存以及组件影响这3个方面.此外,目前也有一些验证与消除时序异常的相关研究工作,这些工作也存在一些问题和不足.WCET分析的未来应当以分析方式为基础进行时序异常相关的研究,以更好地解决时序异常问题.

关键词 最坏执行时间;处理器架构;时序异常;时间可预测;实时系统

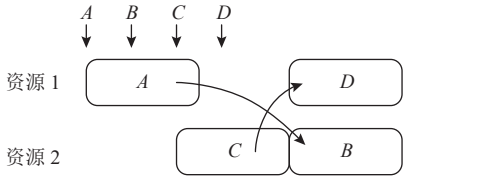
中图法分类号 TP302

DOI: 10.7544/issn1000-1239.202331014 **CSTR:** 32373.14.issn1000-1239.202331014

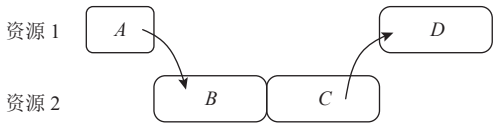
在实时嵌入式系统领域中,最坏执行时间(worst-case execution time, WCET)分析一直是研究的重点.一个程序的WCET是指其代码片段在特定硬件平台上单独执行所需的最长时间[1-2].现有的WCET分析尝试计算出程序WCET的一个安全且紧致的上界,主要有2种办法[3]:一种是基于测量的分析方法,通过在特定的硬件或者模拟器上反复执行需要测量的程序,得到程序的WCET;另一种是静态分析的方法分析程序的代码,并构建出其中的基本块以及控制流图,根据硬件的执行模型来计算出整个程序的执行时间.一个紧致而安全的WCET分析结果,不仅关乎系统资源的利用率,还决定着实时系统中程序时间行为的正确性.然而,在目前绝大部分高性能平台、特别是具备多级缓存架构以及超标量处理器的系统中,WCET分析陷入了两难的境地:要么因为体系结构的不确定性而难以获得一个紧致的分析结果,要么就因为复杂的处理器架构而使得WCET分析时所需要搜索的状态空间爆炸,从而无法在可接受的时间内完成分析.

因此,现有的WCET分析技术[4-10]都尽可能地减少需要搜索的状态空间来优化WCET分析的过程,实现更短的分析时间并得到更紧致的分析结果.传统的WCET分析方法假设如果在分析中取每条指令的最长执行时间,那么从直觉上来说,其对应的程序执行路径的执行时间就是程序的WCET.但是,Lundqvist和Stenström最早在1999年指出,这种直觉是错误的[11].在一个允许动态指令调度的处理器中,如果一条指令在执行时有更长的执行时间,反而可能会使程序总的执行时间减小.

如图1所示,4条指令依次到达2个资源上执行,其中指令A和D使用资源1,指令B和C使用资源2,指令A与B之间、C与D之间存在图中箭头所示的依赖关系.若增加指令A的执行时间(图1(a)),则相较于另一个执行(图1(b)),反而会减少整串指令的执行时间.



(a) 指令A延时更长的执行



(b) 指令A延时更短的执行

Fig. 1 Example diagram of timing anomaly

图1 时序异常示例图

Lundqvist等人[11]将这种反直觉的现象称为时序异常(timing anomaly, TA),这也是时序异常这一概念被首次提出.时序异常的存在意味着在WCET分析中,不能仅仅搜索局部执行时间较长的那些执行,而是要完整地遍历所有可能的执行状态,这需要花费大量的时间与资源.但是如果能够确保系统中不存在时序异常,就能够安全地在WCET分析中抛弃其他状态,大大减少WCET分析所要花费的时间.因此,如何在现有的实时嵌入式系统中避免时序异常、消除时序异常,或是验证某个系统是否存在时序异常就显得至关重要.随着时序异常成为实时领域中的一个热点,越来越多的针对时序异常的定义、探究与证明也被提出[12-16].然而,学术界至今尚未形成一个对时序异常的统一定义与认识.

1 时序异常的定义

1.1 原始时序异常定义

在时序异常的概念被提出之前,Graham[17]最早就于1966年发现了在多核系统中任务的调度存在异常:将一个存在依赖关系的任务集交给n个相同的处理器来执行,如果增加处理器数目或是放松依赖关系亦或是减少单个任务执行时间,都有可能增加

整个任务集的执行时间. 当时的文献将其称为多处理器异常，而非时序异常. 但是从现今的对时序异常的定义来看，这种异常实际上就是时序异常在多核系统下的一种表现.

时序异常这一概念来自于 WCET 分析，早期时序异常的定义是基于程序执行时间的变化，如图 2 所示. Lundqvist 等人[11]最早提出了 2 种对时序异常的定义：考虑一串指令被执行的情况，假设在该串指令执行时，指令序列中的第 1 条指令的执行延时发生了 2 种不同的变化. 在第 1 种情况下，这个执行延时增加了 i 个时钟周期，而在第 2 种情况下执行延时减少了 d 个时钟周期. 如果设 C 为整串指令的执行时间在这 2 种情况下发生的变化，那么时序异常可定义为：

定义 1. 时序异常为下述 2 种情况之一[11].

1）$C > i$ 或者 $C < 0$；

2）$C < -d$ 或者 $C > 0$.

如图 2 所示，无论哪种情况，时序异常都描述了第 1 条指令的执行时间变化，对指令序列的总执行时间产生了过大的影响（图 2(a)(c)）或是相反的影响（图 2(b)(d)）.

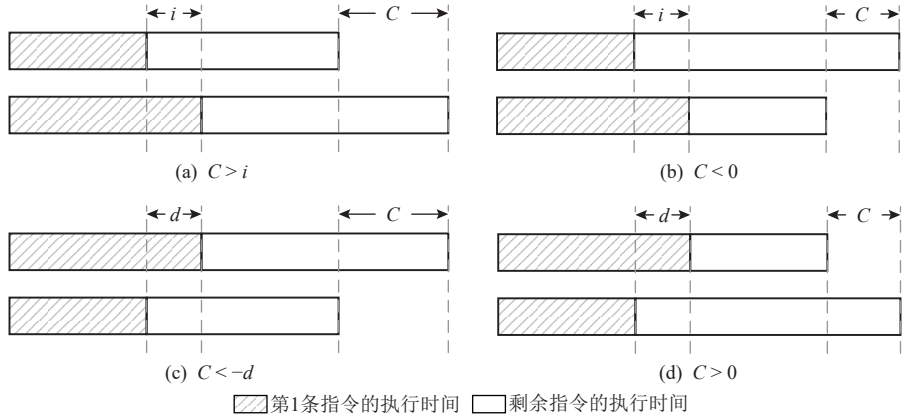


Fig. 2 Behavior of timing anomaly on execution time

图 2 时序异常在执行时间上的表现

定义 1 并不是一个严格的形式化定义，但是它直观地描述出时序异常在执行时间上的表现. 在后续针对时序异常定义的研究中，时序异常往往被定义为 2 类：第 1 类是图 2(a)(c) 所描述的时序异常（一些论文称其为“强影响时序异常（strong impact timing anomaly）”[18] 以及“时序放大（timing amplification）”[19]）；第 2 类是图 2(b)(d) 所描述的时序异常（一些论文称其为“非直接影响（counter-directive impact）”[18] 以及“反直觉时序行为（counterintuitive timing behavior）”[19]）. 为了统一起见，本文称第 1 类为放大型时序异常，第 2 类为反直觉型时序异常.

1.2 反直觉型时序异常

根据反直觉型时序异常的定义，指令序列中第 1 条指令执行时间的增加（或减小），会导致总执行时间的减小（或增加），即发生与预期相反的变化. 这一特性使得 WCET 分析不能仅仅搜索那些在短期内有着较长执行时间的状态，必须遍历所有可能的执行，从而使得 WCET 分析需要搜索的路径数量指数级增加. 反过来说，如果能确保某一程序在特定平台上执行时不会发生时序异常，那么 WCET 分析就可以安全地抛弃那些局部执行时间非最长的状态，大大减少分析时间. 因此，反直觉型时序异常得到了大量的研究，并被提出了各式各样的定义.

1.2.1 原始定义的初步形式化

为了寻找导致时序异常的原因，Wenzel 等人[18]将时序异常采用更加形式化的语言来表述：假设执行的程序为一个指令序列 $I_1, I_2, \cdots, I_n$，每一条指令都有着一个在对应的功能单元上执行所需的延时 t_{I_i}，$i \in \{1, 2, \cdots, n\}$. 执行整个指令序列所需要的时间记为 C，指令序列中第 1 条指令的延时变化大小为 Δt，变化后整个指令序列的执行延时记为 C'，指令序列变化后的时间为 $t_{I_1} + \Delta t, t_{I_2}, \cdots, t_{I_n}$，记这两者的时间差为 $\Delta C = C' - C$. 则时序异常定义见定义 2.

定义 2. 一个时序异常定义为下述两者之一成立的情形[18].

1）第 1 条指令的执行延时增加时：

$\Delta t > 0 \rightarrow (\Delta C > \Delta t) \vee (\Delta C < 0)$；

2）第 1 条指令的执行延时减小时：

$\Delta t < 0 \rightarrow (\Delta C < \Delta t) \vee (\Delta C > 0)$.

定义 2 与定义 1 所描述的含义基本相同，其区别

在于，定义 2 细化了对指令执行时间的描述，并将这个执行时间的粒度限制在单条指令在功能单元执行的延时上. 此外，定义 2 在更加具体地描述执行时间的同时，也隐含地给时序异常的定义增加了约束，即指令序列中除了第 1 条指令外，后续的指令执行时间必须保持不变[16]，而定义 1 中的时序异常的情况可能不满足这个约束.

定义 1 和定义 2 还存在着一个相同的问题，即执行时间的来源. 在现代更加复杂的处理器中，无论是一段程序还是单条指令的执行时间，可能由处理器中多个不同功能单元上的延时组成，它们之间还可能存在着重叠、间隔，但是这 2 个定义并没有告知如何将这些细粒度的延时与定义中的指令执行时间相关联. 因此，这 2 个定义仅适合在概念上对时序异常的理解，却不便于在实际分析中应用.

1.2.2 基于微操作的定义

先前所描述的时序异常仅仅从一个理想的角度形式化地描述了时序异常的表现，对于稍加复杂的系统，如何描述指令执行时间的问题就显露了出来. Reineke 等人[12]即以兼容不同系统的硬件特征为目标提出了新的形式化时序异常定义，为了兼容不同系统的硬件特征，该定义采用一个转移系统来抽象描述系统执行的过程. 简单地来说，将一个硬件平台所有可能具备的状态抽象成一个集合，硬件执行程序的过程就是转移系统在不同状态之间转移的过程，并可以根据系统所途经的状态得到一个状态序列. 如图 3 所示，π 和 π'就是转移系统执行相同的指令流时 2 个可能存在的状态序列，其中每个小方块都代表系统的一个状态.

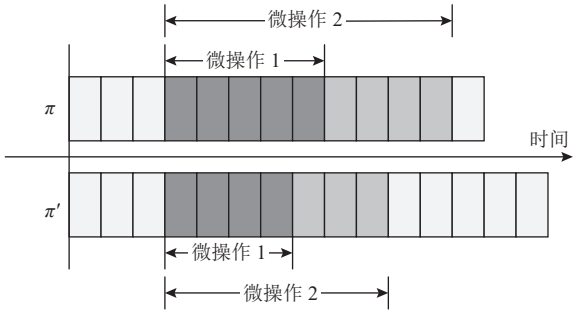


Fig. 3 Example of timing anomaly defined by Reineke

图 3 Reineke 定义的时序异常示例

Reineke 等人[12]还提出指令的局部执行时间应当在系统的微操作方面来进行描述(如取指、执行操作等). 在这个概念下，“局部更坏”可以由微操作表达为：如果在某一执行状态序列中，每个微操作的长度都大于其余所有序列中微操作的长度，则该执行状态序列被称为一个“局部更坏”的序列. 由此，提出的基于微操作的时序异常定义如下：

定义 3. 一个硬件模型存在时序异常，当且仅当存在 1 个程序以及程序中的 1 个指令序列，使得该指令序列下存在一个非局部更坏的执行状态序列，其总执行时间是所有序列中最长的.

如在图 3 中，微操作 1 与微操作 2 在 2 个不同的执行状态序列下消耗了不同的执行时间，状态序列 π 中 2 个微操作都比在序列 π'消耗的执行时间更长，因而根据局部性的定义，序列 π 是一个局部更坏的执行序列. 然而，π'序列最终有着更长的总执行时间，π 作为局部更坏的执行序列反而全局更优，因而这个系统存在着反直觉时序异常.

Reineke 等人[12]提出的基于微操作的反直觉时序异常定义，从局部性角度入手，在一个更细的执行粒度上描述了它在系统中的表现. 然而，该定义仍然存在着一些问题，Binder 等人[14]指出，在系统中同时发生多个局部更坏的执行情况时，这个定义无法识别出实际存在的反直觉型时序异常.

比如在图 4 中，序列 π 的微操作 1 比在序列 π'中的执行时间更长，微操作 2 则相反. 根据微操作下局部最坏执行状态序列的定义，只有当一个序列中的每个微操作的执行时间都比另一个序列中对应的微操作的执行时间更长时，才称之为局部更坏的执行序列. 因而图 4 中的 2 个执行序列互相令对方不为局部更坏的序列，进而不满足基于微操作的时序异常的定义，然而从微操作 1 或微操作 2 各自的角度来看，系统本身是存在时序异常的，基于微操作的定义却无法覆盖到这种情况. 这一问题的本质在于该定义对局部更坏的执行序列要求过于严格，使得其无法处理多个局部更坏的微操作同时发生的场景.

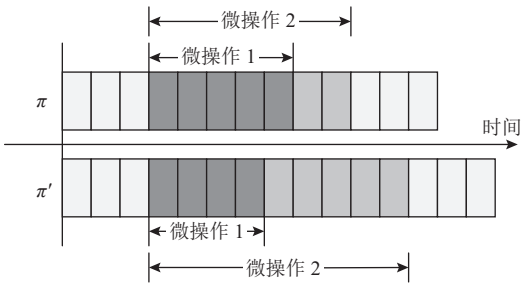


Fig. 4 Drawback of timing anomaly defined by Reineke

图 4 Reineke 定义的时序异常缺陷

1.2.3 基于执行 trace 的定义

Cassez 等人[16]指出，微操作定义的时序异常仍然存在一些不足：它们仅要求系统所执行的指令流

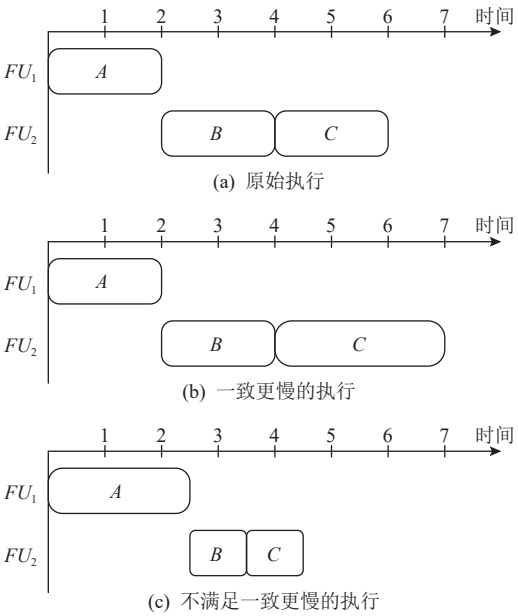
相同，但是一个系统实际执行时，往往还需要从外界输入或输出数据. 因此，在定义时序异常时，还有必要要求系统执行时从外界接受的数据也相同.

此外，Cassez 等人[16]还认为，对系统进行过度的抽象会导致一些形式化模型中存在虚假的时序异常，因此，时序异常应当在具体的而非抽象过后的硬件模型中定义. 为此，他们采用硬件系统执行时的踪迹(trace)作为时序异常的描述方法.

在 Cassez 等人[16]的工作中，为了保证系统执行的指令流严格且唯一，系统执行时的 trace 被定义为在特定程序与数据输入的情况下一个带编号的指令序列. 硬件系统则通过一个带有时间与事件标签的转移系统来描述. 这样一来，系统在各个状态之间的转移所构成的标签序列就是指令流随着时间在硬件系统中执行的过程. 根据这些信息可以计算出每条指令(相对于开始执行时的)在 trace 中的完成时间. 在这个结果下，无时序异常系统被定义如下[16]：

定义 4. 一个系统相对于某一程序和输入的数据是无时序异常的，当且仅当系统中指令的完成时间在不同状态之间满足“单调性”，即系统从某一状态下执行时，每条指令的完成时间都大于等于其余所有状态下的完成时间.

图 5 分别展示了系统在相同程序和数据输入的情况下，从 3 个不同的初始状态开始，执行 trace：[1→*A*, 2→*B*, 3→*C*] 的过程. *A*, *B*, *C* 这 3 条指令在图 5(a)中的完成时间分别为 2, 4, 6, 在图 5(b) 中的完成时间则分别为 2, 4, 7. 如果不考虑图 5(c)，则图 5(b)初始状态下每条指令的完成时间都大于其他所有初始状态(在本例中只有图 5(a) 状态)下每条指令的完成时间，Cassez 等人[16]称这一性质为“一致更慢(consistently as slow)”. 因此根据定义 4，在不考虑图 5(c)状态的情况下，这个系统是无时序异常的. 如果考虑图 5(c)状态，不难发现任一初始状态都不满足“一致更慢”这一性质的要求，因此表明系统中存在时序异常.



注：*FU* 表示功能单元.

Fig. 5 Example of timing anomaly defined by Cassez[16]

图 5 Cassez 定义的时序异常示例[16]

Cassez 等人[16]提出的定义重点在于，存在时序异常的执行不仅需要有相同的指令序列，还要有相同的数据输入，否则不构成发生时序异常的条件. 因为如果输入数据不同，可能导致指令的实际执行情况发生变化. 以 ARM 指令集架构为例，若图 5 中 *A*, *B*, *C* 指令分别是以下 3 条指令：*A*: MUL R_0, R_0, R_1; *B*: MULNZ R_2, R_2, R_1; *C*: MULNZ R_2, R_2, R_1. 其中 *B* 和 *C* 是条件执行指令，只有当 R_0 不为 0 的时候才会进行运算. 因此，若输入数据使 R_0 不为 0 时，系统的执行过程为图 5(a)，当输入数据使 R_0 为 0 时，系统的执行过程为图 5(c). 用不同的定义对这 2 个执行进行判断，就产生了截然相反的结果：定义 3 仅考虑到了 2 个执行的指令流相同，因此认为这 2 个执行是一个时序异常；而定义 4 认为因为 2 次执行有着不同的输入数据，因此不满足构成时序异常的条件.

总得来说，Cassez 等人[16]使用指令的完成时间作为判断局部更坏性质的依据，相较于 Reineke 等人[12]的方法粒度更粗，但是它避免了传统定义中如何将单条指令的执行时间映射到硬件这一问题，具有一定的意义.

1.2.4 基于因果关系的定义

随着对时序异常的关注不断加深，越来越多的针对反直觉型时序异常的定义被提出[12,15-16,20]. 然而，大部分定义都只描述了时序异常在抽象硬件模型上的表现，无法描述时序异常复杂的硬件架构(特别是超标量处理器的架构)下的表现[14]. 因此，Binder 等人[14]从指令运行时在各个部件之间因果关系的角度，提出了一个适用于具体架构下精确的时序异常定义.

基于因果关系的时序异常的定义将硬件视为各个功能部件组成的集合，程序在硬件系统上的一次执行由发生在功能部件上由指令、部件、时刻构成的事件构成，描述了在该时刻下某条指令获取或释放了一个部件. 根据不同事件之间的依赖关系(如资源竞争、指令流顺序、数据依赖等)，可以构成一个以事件为节点、依赖关系为有向边的图，Binder 等人[14]

称其为事件时间依赖图(event time-dependence graph, ETDG). 从事件时间依赖图中修剪去无关的边, 即可得到程序在硬件上执行的因果关系图(causality graph, CG)[14]. 根据因果关系图, 可以精确地确定每一个事件会影响后续哪些事件的执行, 确定其影响范围.

基于执行事件之间的因果关系, Binder 等人[14]将时序异常定义如下:

定义 5. 一个时序异常是指, 在 α 执行中存在一个资源, 它的执行时间比在 β 中更短, 且在其因果事件范围内, β 执行中存在一个发生时间比 α 更长的事件.

定义 5 可以精确地界定时序异常的范围, 并根据因果关系推导出导致时序异常的指令或部件. 一个例子就是它能够排除一些虚假的时序异常. 如图 6 所示, α 与 β 是指令流 A 到指令流 E 在某一硬件平台上的 2 个不同的执行, 根据 Reineke 等人[12]提出的时序异常定义(定义 3), α 执行构成了一个时序异常. 然而, 从因果关系的角度来看, α 执行之所以有着更长的总执行时间, 是因为指令 E 在 IF 段消耗了更长的执行时间. 根据因果关系图, 指令 A 在 FU_1 段执行事件的因果范围仅仅局限在指令 A 提交事件的范围内(图 6 中的浅灰色部分), 这与直觉上的因果关系是一致的. 因而从因果关系的角度来看, 图 6 的 2 个执行无法构成 1 个时序异常.

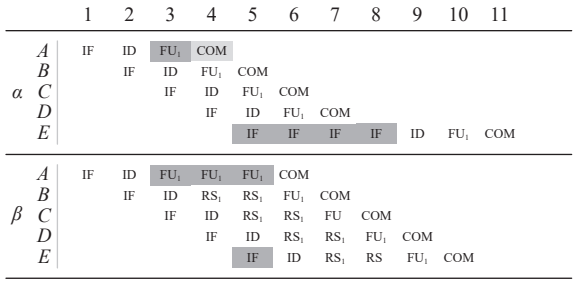


Fig. 6 Example of fake timing anomaly[14]

图 6 虚假的时序异常示例[14]

基于因果关系的定义没有提到局部执行时间与全局执行时间的概念, 相对应地, 它采用一个事件所花费的时间以及事件之间的时间, 来刻画一次执行时间变化对整个执行的影响. 这种细粒度的描述方式使得该定义有着许多优点, 包括:

1)指出其他定义中存在的虚假时序异常实例;

2)发现先前定义所遗漏的时序异常实例;

3)处理事件组合、交错等情况下的时序异常.

此外, Binder 等人[14]还首次提出了在时间变化组合下定义时序异常的问题, 这是先前定义所从未涉及的.

1.3 放大型时序异常

放大型时序异常指在一次局部执行时间变化的情况下, 全局执行时间的变化比局部执行时间的变化要更大, 导致了一种“放大”局部变化的效果. 显然, 放大型时序异常不会影响 WCET 分析时的决策, 因为一次局部时间更长的执行仍然有着更长的全局执行时间, WCET 分析时只需继续沿着该分支进行搜索即可. 也正因如此, 放大型时序异常一直是 WCET 分析中的冷门研究, 鲜有对它的严格定义.

近几年来, 在针对时间可组合性的研究中, Jan 等人[19]指出, 要实现时间可组合的 WCET 分析, 处理器必须要消除放大型时序异常. 时间可组合性是指在存在多个组件的系统中, 系统整体的 WCET 可以通过组合各组件的 WCET 来得到, 比如在一个包含高速缓存的处理器中, 可以单独地分析流水线和高速缓存 WCET 的上界, 然后结合调度策略将它们各自的结果组合到一起, 得到整个系统的 WCET 的上界. 然而, 如果存在放大型时序异常, 组合后的 WCET 可能不再安全: 一次缓存命中到缓存缺失的变化所产生的延时, 可能对组合后总 WCET 的变化有着更大的影响, 从而影响计算结果的安全性.

为了描述放大型时序异常, Jan 等人[19]使用一个标准流水线模型模拟处理器执行时的过程. 如图 7 所示.

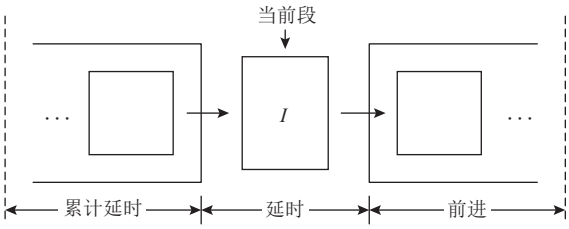


Fig. 7 Canonical pipeline model[19]

图 7 标准流水线模型[19]

在标准流水线模型中, 程序在流水线中执行的过程就是指令在各个流水段之间进行状态转移的过程, 一条指令的状态被定义为如下的六元组[19]:

$$\langle type, \boldsymbol{vl}, stage, latency, delay, progress\rangle,$$

其中 *type* 描述了指令的类型, ***vl*** 是包含着该指令在流水线中每个段延时的向量, *latency* 是指令在当前流水段 *stage* 剩余的执行时间, 即图 7 中的延时. *delay* 描述了指令在先前各个流水段中因为流水线停顿而导致的累计延时(对应与图 7 中的累计延时). *progress* 是一个决定流水线是否停顿的布尔值.

在标准流水线模型中, 放大型时序异常即为指

令在某一流水段执行延时的增长，导致了整个指令序列执行时间更大的增长[19]（类似执行时间减少的情况）. 相较于反直觉型时序异常的定义，放大型时序异常的定义仍然具有很多不完善的地方，如整个指令序列的执行时间在模型中的计算方式、模型的初始状态与转移又该如何形式化地表达，均缺少严格定义. 现有的放大型时序异常定义还局限在提供概念理解以及满足可组合性分析的工作[21]上.

1.4 有界时序异常

前述介绍的各类时序异常定义均为对系统的定性描述，即系统中要么存在时序异常，要么不存在. 本节介绍有关在时序异常方面进行定量分析的工作，即有界时序异常.

1.4.1 多米诺效应

在介绍有界时序异常之前，需要先说明时序异常的多米诺效应. 多米诺效应在最早关于时序异常的文献中就有被提出. Lundqvist 等人[11]和 Wilhelm 等人[22]指出，在存在时序异常的体系结构中，如果时序异常位于某一循环内，可能会使程序 WCET 的变化受到循环次数的影响，从而无法有界地被确定，这一现象即被称为多米诺效应.

图 8 展示了一个因静态预测取指未命中所导致的反直觉型时序异常，其中分支来自指令 A 和指令 C 之间的循环. 嵌套在循环中的时序异常进一步导致了多米诺效应：随着每一次循环的执行，时序异常反复地出现，使得指令 A 命中缓存下的执行时间与未命中缓存下的执行时间的差值随着循环次数线性增加. 在无法确定循环次数的情况下，这 2 种执行的时间差可能达到任意大小.

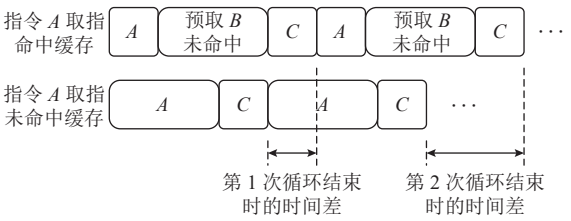


Fig. 8 Illustration of Domino effect

图 8 多米诺效应示意图

遗憾的是，目前许多硬件系统都存在着多米诺效应[15,22]，Schneider[23]曾证明 MPC755 处理器的流水线存在多米诺效应，此外，Berg[24]证明了常见的几种缓存替换策略，如 round-robin、FIFO（first in first out）、伪 LRU（least recently used）都受多米诺效应的影响，只有 LRU 替换策略例外.

多米诺效应实际上描述了时序异常“无界”的性质，即一些时序异常对程序 WCET 的影响可能是任意大小的. 那么相对地，是否在某些场景下，时序异常仅能对程序的 WCET 产生有限的影响？这就是有界时序异常.

1.4.2 k-有界的时序异常

根据多米诺效应的定义，时序异常对系统的影响程度直接取决于执行时间的变化程度. Gebhard[15]从程序的执行时间出发，提出了一个时序异常的形式化定义：

在基于执行时间的定义中，程序在硬件上的一次执行被视为（硬件状态，指令编号）二元组到正实数域上的映射，它将程序中的每条指令按照编号映射到在硬件状态下的相对执行时间. 将每条指令的相对执行时间累加，就得到了指令从初始硬件状态开始的绝对执行时间. 不难发现，指令的相对执行时间描述了局部执行特征，指令的绝对执行时间则对应于全局下的表现. 因此，基于执行时间的时序异常被描述为[15]定义 6.

定义 6. 当系统中存在一个硬件状态 θ 和 2 条编号不同的指令 i, n, $i<n$, 使得指令 i 在状态 θ 下的相对执行时间小于其余任意硬件状态下的执行时间，且指令 n 在状态 θ 下的绝对执行时间大于其余任意硬件状态下的执行时间，则状态 θ 是时序异常的.

如图 9 所示，状态 θ 执行是时序异常的，因为相较于状态 η 执行，指令 i 的相对执行时间更短，但是状态 θ 在后续有总的更长的执行时间（即指令 n 的绝对执行时间）. 如果要求状态 θ 中，指令 n 的绝对执行时间至多比在状态 η 中（或者说其余任意执行）大 k，则状态 θ 就构成了一个 k-有界的时序异常状态. 从这个角度来说，k-有界时序异常本质上是反直觉型时序异常的一个分类.

k-有界时序异常的意义在于，如果某一类时序异常满足 k-有界时序异常的定义，则在 WCET 分析中，可以通过往 WCET 分析结果上增加常量 k 的方式，安全地抛弃一些有着大量状态的执行分支，从而减

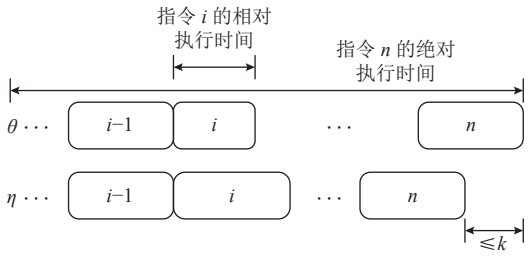


Fig. 9 Example of k-bounded timing anomaly

图 9 k-有界的时序异常示例

少 WCET 分析所需的计算量. 如果 k-有界时序异常中的常数 k 可以以某种方式得到足够小的结果，则分析出的 WCET 结果也相对地更加紧致. 这为解决 WCET 分析中的时序异常问题提供了一个新的研究方向.

尽管如此，目前针对 k-有界时序异常的工作仍然较少. 如前所述，相当一部分体系结构中已被证明存在多米诺效应，因而不适用于这种分析方法. 此外，由于 k-有界时序异常的定义仍然停留在抽象执行模型的层次，难以直接应用于具体的体系结构与分析工具，这导致如何发现与证明 k-有界时序异常的相关工作也没有得到有效开展.

1.5 并行时序异常

Lundqvist 等人[11]从指令在 WCET 分析中线性组合的角度提出了时序异常. 大部分对时序异常的定义也都集中在指令流的线性分析中. 然而，对硬件平台的分析也是 WCET 分析中重要的一环. 由于硬件系统的复杂性，硬件平台的分析往往通过分而治之的方式，先分析出每个子组件的时间行为，然后再将它们组合到一起，得到整个系统的时间行为. Kirner 等人[20,25]提出，在分解组合和组件之间的时间行为时，也可能会发生时序异常，即并行时序异常.

并行时序异常和指令流分析中时序异常的主要区别在于，指令流分析中时序异常描述的对象是指令以及程序的执行时间，这两者之间往往存在着(不严格的)线性关系；并行时序异常针对的是硬件系统中组件与整体的执行时间，显然，组件之间是并行执行的，时间行为的组合方式也取决它们之间的依赖关系. 并行时序异常的存在与否直接影响在 WCET 分析中各个组件的时间行为是否可组合.

类似于定义 1，并行时序异常也被分为并形反直觉型时序异常和并形放大型时序异常(Kirner 等人[25]称其为并行反转(parallel inversion)异常与并行放大(parallel amplification)异常). 只不过判断对象由指令和程序变成了组件和整体. 并行反直觉型时序异常定义如下[25]：

定义 7. 设系统可以被分解为 2 个组件 A 与 P，若存在某一程序，使得其在 A 上 2 个不同初始状态 a 与 a'下执行时间的大小关系，与在 A 与 P 上 2 个不同的初始状态(a, p)与(a', p)下执行时间的大小关系相反，则称系统中存在并行反直觉型时序异常.

并行放大型时序异常的定义可由定义 7 稍加修改得到，除此之外，Pedersen 等人[26]还给出了一个在半马尔可夫过程下并行时序异常的形式化定义，此处均不再赘述.

相较于面向指令流的时序异常，并行时序异常的影响更容易规避，Kirner 等人[25]已证明，当系统中仅存在特定的时序异常时，通过调整组件之间组合时间行为的方式，可以规避时序异常的影响，得到一个安全的 WCET 分析结果. 因此，针对并行时序异常的研究有望成为满足可组合性，以及实现 WCET 分析分解的关键.

1.6 时序异常定义小结

本节介绍了自时序异常被提出以来，各类时序异常的定义. 本节将前述介绍的定义汇总如表 1 所示. 在对时序异常现象的抽象方式上面，虽然各个定义对执行时的抽象方式不同，但是本质上都是在对比执行时间，只不过所取的粒度不同. 如在局部执行时间变化的粒度方面，原始定义[11,18]和 Gebhard[15]的定义都采用单条指令的执行时长作为局部执行时间变化的单位，这使得无法较好地将定义和实际执行关联起来，即因为流水线等技术的实现，指令之间的执行时间是互相重叠的，而并非像指令集架构层所

Table 1 Comparison of Different Definitions of Timing Anomalies

表 1 不同时序异常定义的对比

时序异常属性	原始时序异常定义[11,18]	Reineke 等人[12]的时序异常定义	Cassez 等人[16]的时序异常定义	Binder 等人[14]的时序异常定义	Gebhard[15]的时序异常定义	并行时序异常定义[25-26]
抽象对象	指令执行时间	指令微操作的执行时间	指令执行 trace	指令微操作间的因果关系	指令执行时间	组件执行时间
能否表示反直觉型时序异常	是	是	是	是	是	是
能否表示放大型时序异常	是	否	否	否	否	是
局部执行时间变化的粒度	指令	微操作	指令	微操作	指令	(子)组件
全局执行时间变化的粒度	程序(指令片段)	程序(指令片段)	指令	微操作	程序(指令片段)	复合组件
是否要求输入数据一致	否	否	是	否	否	是(取决于抽象程度)
是否适用于多个局部执行时间同时发生的情况	不适用	有误判(图 4)	适用	适用	不适用	不适用
时序异常组合分析	不适用	不适用	不适用	适用	不适用	不适用

假设的那样，只有一条指令执行完之后下一条指令才会开始执行. 值得注意的是，虽然 Cassez 等人[16] 的定义采取的粒度也为单条指令，但是在实际描述中采用的是每条指令结束执行的时刻，而非指令的执行时长，这使得他们的定义能够较好地与具体执行过程相联系.

在全局执行时间变化的粒度方面，大部分定义采用整个程序执行时间为粒度，这在直觉上是正确的，因为时序异常最终影响的就是程序的 WCET 分析结果. Cassez 等人[16] 与 Binder 等人[14] 的定义则将全局性与局部性的粒度保持一致，其中 Cassez 等人[16] 的定义并没有直接描述程序整体的执行时间，而是通过“始终更慢”这一性质来间接地描述全局执行时间上的变化. Binder 等人[14] 则将程序整体执行时间的变化视为程序中最后一条指令微操作执行时间（相较于局部事件）的变化，从而实现在局部粒度下描述全局执行时间[27]. 然而，这样做也存在着一些问题，不直接使用程序执行时间作为全局执行时间变化的粒度，无法保证模型定义的执行时间变化与程序实际执行时间的变化是一致的. 如 Binder 等人[14] 基于因果关系的定义，在实验时检测到了其他定义所没有发现的时序异常（如文献 [14] 中图 6），这本质上就是因为使用微操作表述的全局执行时间与程序的执行时间存在偏差，从而检测出与其他定义不一致的时序异常. 并行时序异常着眼的对象为系统组件，因而局部执行时间和全局执行时间的粒度都是以组件为单位.

此外，在不同的时序异常定义之间，只有 Cassez 等人[16] 的定义要求时序异常必须在外界输入保持一致的情况下才能成立. Binder 等人[14] 的定义仅在分析中提到了这一点，没有在时序异常的定义中加上对应的约束. 并行时序异常由于仅考虑时间相关的动态计算机状态（timing-relevant dynamic computer state, TRDCS）[25]，因而是否满足数据一致性取决于与数据相关的状态是否被包含在 TRDCS 内. 其余定义均没有要求输入数据的一致性，所以在面对诸如条件执行等情况时，有可能错误地检出时序异常.

在复杂的执行情况下（如有多个局部执行时间同时发生变化，或者数个时序异常重叠），Reineke 等人[12] 的定义就显得过于严格，因此无法识别在复杂情况下的时序异常. Cassez 等人[16] 的定义虽然能够识别出时序异常，但是无法区别它们. 基于因果关系的定义在这一方面最为完善，它不仅可以检出这些复杂情况下的时序异常，还能正确地区分不同类型的时序异常，并界定它们的影响范围.

从时序异常的表达能力来看，所有时序异常定义都描述了反直觉型时序异常的概念，却少有定义涉及放大型时序异常. 实际上，如果允许稍加修改，表 1 中大部分定义都能够描述放大型时序异常. 这也反映出在 WCET 分析中，反直觉型时序异常的影响最为重要，也受到了最多的关注. 此外，表 1 中没有涉及 k-有界时序异常的相关内容，这是因为本文认为 k-有界时序异常本质上是对时序异常的一种分类，而非一种新的形式化定义，故此处没有提及.

2 导致时序异常的原因

各种时序异常的存在给体系结构的 WCET 分析带来了巨大的困难，为了消除时序异常，就必须确定导致时序异常的原因. 尽管学界尚未有一个统一的时序异常定义，目前已经确定了数种特定情况下导致时序异常的原因.

Reineke 等人[12] 在一篇关于时序异常定义与分类的文章中，将时序异常分为了（至少）3 类，分别是调度型时序异常、高速缓存型时序异常以及超标量型时序异常. 巧合的是，从导致时序异常的原因来看，它们也基本满足这 3 个分类（对于超标量型时序异常，本文认为其原因来自于处理器组件）. 本节即从调度、高速缓存以及处理器组件的角度，分析导致时序异常的原因.

2.1 调度型时序异常的原因

在 Lundqvist 等人[11] 最早提出时序异常的文章中，给出的时序异常范例就是因调度策略产生的时序异常，Reineke 等人[12] 将该种时序异常称为“调度型时序异常”，即在 2 个构成时序异常的执行之间，它们执行时间的变化是由于指令在资源的分配顺序上发生变化所导致的. 如图 1 所示，在资源 2 上指令 B 和 C 的执行顺序颠倒，因此图 1 所示的时序异常即为调度型时序异常.

调度型时序异常的发生主要来自于调度策略. 现今绝大部分处理器在满足数据依赖关系的情况下，都使用贪婪的策略为指令分配现有的执行资源. 特别是对一众超标量处理器，指令无需等待前序指令执行结束即可开始执行. 时序异常的发现者 Lundqvist 等人[11] 提出，正是指令在处理器中的乱序执行才导致了时序异常的出现，如果一个处理器仅包含顺序执行的资源（in-order resources），则不会产生时序异常.

Lundqvist 等人[11]将顺序执行资源定义为一类只能按照程序顺序分配给指令的资源,如处理器中的寄存器堆就一定是顺序执行的资源,否则程序执行的结果将会不正确. 与之相对的乱序执行资源则可以按照任意顺序执行已发射的指令,如通常的计算单元、乘除法器、存储控制器等处理器中的功能组件. 他们提出,如果处理器中的所有资源都是顺序执行的,那么前一条指令的执行延时一定会使后续指令对该资源的使用产生影响,在最理想的情况下(即后续没有指令需要使用前一条指令的资源),2 个执行的时间差为 0,不会出现反直觉型时序异常.

但是这一结论缺乏严格的证明,在 Lundqvist 等人[28]后续的文献中,将导致时序异常的原因的描述修改为:如果一个处理器包含乱序执行的资源,则动态调度指令执行将会产生时序异常. 这一修改表示,仅包含顺序执行的资源不再是处理器无时序异常的充分条件. 此外,Gebhard[15]以 LEON2 处理器架构为例,说明了即使在仅包含顺序执行资源的处理器中,也有可能发生时序异常. 他们提出的反例是一个因处理器中高速缓存命中与未命中情况下的执行导致的时序异常,其本质上可被分类为因高速缓存所导致的时序异常,对于该反例的详细说明见 2.2 节.

本文认为,Lundqvist 等人[11]提出导致调度型时序异常的原因之所以遭到否定,并非是因为其在论述过程中出现了错误,而是缺少对调度型时序异常的严格描述. 调度型时序异常应当是指因指令在资源上的分配顺序变化而产生的时序异常. 换一种方式来说,如果一个处理器仅包含顺序执行的资源,可能会产生普遍的反直觉型时序异常,但不会产生调度型时序异常. 这一点也在 Gebhard[15]对 LEON2 处理器的分析中得到了验证.

因此,产生调度型时序异常的原因之一在于处理器中存在乱序执行的资源. 但是该命题的逆否命题,即调度型时序异常不会出现在仅包含顺序执行资源的处理器中,尚未得到严格的形式化证明. 虽然 Lundqvist 等人[11]给出了关于该命题的一小段证明,但是证明不严格. 从上述讨论也不难看出,对该命题的证明需要从对调度型时序异常的形式化建模开始进行严格的论述,否则证明过程中的不严谨会很容易将其他原因导致的时序异常囊括在内,使得到的结论不可靠.

2.2 高速缓存中的时序异常

高速缓存的不确定性也是导致时序异常的重要原因之一. 特别地,本文称所有因高速缓存的访存时间发生变化而导致的时序异常为高速缓存导致的时序异常. 在高速缓存中,一次命中可能会引发缓存内容的替换,进而导致后续的访问数次未命中,产生了额外的执行时间,从而构成了一次时序异常.

由于高速缓存的容量总是有限的,因此无法保证某一缓存状态始终是其余缓存状态中数据的超集,进而不能确保某一缓存状态在未来的访问下的响应时间一定比其他状态更好. 本文认为,这一问题的本质在于信息差,缓存替换策略的工作实际上就是决策将哪些数据留存在缓存内,以实现最高的命中率和最短的访问时间,但是这一决策是在线的(on-line),换而言之,缓存的每次决策仅能参考当前以及先前的访问序列,而无法预知未来处理器将会发出什么访存请求. 这使得决策的结果相较于整个访存序列来说不一定是最优的. 显然,如果能够预先知道所有的访存请求,那么一定能通过某种方式构造出一个访存效率最优的缓存替换序列,但是对于需要立刻处理每次访存请求的缓存来说,其最多只能构造出在现在以及先前的访存序列下最优的替换策略. 能否知晓整个访存序列就是其中的信息差,使得缓存的替换策略在决策时无法做出对整个序列来说最理想的选择.

一个例子就是 Gebhard[15]在 LEON2 处理器中发现的时序异常,LEON2 处理器的高速缓存采用了不具备多米诺效应的 LRU 替换策略,即使如此,LEON2 的缓存仍然存在时序异常. 图 10 展示了这个问题.

在 LEON2 处理器中,每个缓存行可以容纳 8 个字,缓存行中的每个字都有一个有效位来标志字的有效性. LEON2 缓存对缓存未命中的处理方式和一般的缓存存在少许区别:不同于在缺失时读取整个缓存行的替换策略,LEON2 仅会从对应缓存行中开始缺失的那个字开始读取,直到缓存行的末尾(即总线不使用 wrap 方式工作,不一定读取一个完整的缓存行),且填充缓存行时不检查每个字的有效位. 在图 10 中,初始取指令块 A 在访存命中(浅灰色)的情况下,不能保证指令块 B 与指令块 C 也一定在缓存中,而程序先执行了指令块 C,从而发起了一次对指令块 C 的取指,接着程序又跳转到指令块 B,由于在取指令块 C 时,指令块 B 没有被一同读入,因此在执行指令块 B 时,又需要发起一次对指令块 B 的取指请求,因为指令块 C 紧接着指令块 B 且它们在同一个缓存行中,所以指令块 C 随着指令块 B 被重新读入了一遍,从而导致相较于初始取指令块 A 未命中的情况下,指令块 C 传输了更多的数据,花费了更长

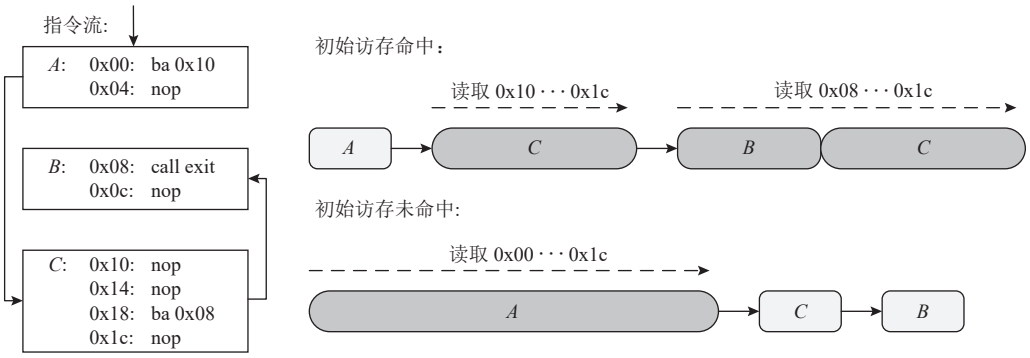


Fig. 10 Timing anomaly caused by cache in LEON2 processor[15]

图 10 LEON2 处理器中因高速缓存导致的时序异常[15]

的执行时间，进而构成了一次高速缓存导致的时序异常[15].

上述例子即是替换策略因信息差而导致的一个时序异常，替换策略并不知道在第 1 次访存过后，程序跳转到了缓存行的后半段开始执行，接着又返回了前面的代码段，这其中所产生的指令访问需求在第 1 次取指时完全是未知的，正是这种局部与整体的信息差，导致了缓存型时序异常.

2.3 处理器组件中的时序异常

导致时序异常的原因可能来自于处理器中各种结构复杂的组件. 现代处理器为了提高指令吞吐率与平均执行性能，往往采取各种激进的方式来充分利用处理器中的各个功能组件. 各个功能组件内部以及组件之间，都有可能因 1 次执行时间变化而产生时序异常.

组件内部产生时序异常的典型例子就是 2.2 节所述的因高速缓存所导致的时序异常，高速缓存本身作为处理器中的一个组件，自然属于因组件内部导致的时序异常. 同样类似的还有分支预测组件、指令预取组件等. 事实上，组件内部时序异常的原因一般都可以归类到 2.1 节和 2.2 节所述的原因中，本文在此罗列如下：

1）调度策略导致的时序异常. 如动态调度流水线、乱序执行流水线、存取控制器.

2）信息差导致的时序异常. 如高速缓存、指令预取、分支预测.

在调度策略导致的时序异常中，这些组件往往因为追求性能，允许指令或请求不按照程序序列（program order）来执行，产生了大量可能的执行顺序，这些不同的执行顺序构成了时序异常. 而在因信息差导致的异常中，这些组件均有着大量的状态空间，且都存在着局部执行和整体执行之间的信息差，从而使得组件执行时无法根据已有的执行情况做出（全局）最优的决策，更加详细的原因已在 2.2 节中进行了分析，此处不再赘述.

另一种即为组件之间互相影响而产生的时序异常. 比如在经典 5 段流水线处理器中，存储结构采用哈佛结构设计，指令存取和数据存取可以各自发出访存请求，它们之间对总线的竞争以及对缓存内容的影响，都有可能产生时序异常. 一个例子就是一次指令取指未命中导致的缓存替换，可能会使得后续的数据存取能够连续命中，减少了总的访存时间，这样一种情况便被称为因组件之间互相影响导致的时序异常.

导致组件之间发生时序异常的原因，以及产生这些时序异常的条件是时序异常研究中的关键. 遗憾的是，由于处理器结构的多样性和复杂性，目前暂没有这方面的结论. 不过对简化的处理器模型来说有一个结论. 如 Hahn 等人[29] 提出，在经典的 5 段流水线处理器中，如果加之以适当的约束，比如禁止指令预取、分支预测等超标量技术，并且限制访存必须按照指令序发起，则构成的处理器是无时序异常的. Hahn 等人[29] 将这个要求称之为“单调性”要求，意为程序在处理器中执行时，每条指令的执行不能受到它后面指令执行的影响. 对于这一性质具体的分析可见 3.4 节. 这一要求本质上就是约束了处理器组件之间的干扰，尽量确保每个组件的执行只和尽量少的指令有关，从而避免时序异常的发生. Gruin 等人[30] 在构建 MINOTAuR 处理器的过程中也验证了这一点结论.

总而言之，随着现代处理器日益增加的复杂性和多样性，时序异常的成因和处理器本身的结构息息相关，想要准确地确定时序异常的成因，往往需要对处理器进行完整、细致的分析，但是进行分析代价往往是不可接受的. 本节讨论了时序异常的各类成因，但是对于处理器组件之间的时序异常，仍有很多

工作需要开展，才能更准确地定位导致时序异常的原因.

3 验证与消除时序异常

为了避免时序异常对 WCET 分析的影响，针对时序异常研究的另一个重点在于，如何采用已有的定义来对时序异常进行检测以验证某一体系结构是否存在时序异常[31-32]，或者从导致时序异常的原因入手消除时序异常.

3.1 反直觉型时序异常的自动验证

Eisinger 等人[33]提出了一种通用的时序异常自动验证方法，可以检测某一体系结构中是否存在反直觉型时序异常. 在他们给出的方案中，处理器被建模成一个有限状态机，并例化了 2 个模型用于比对验证，分别称之为 CPU_f(下标 f 表示 full-featured，全功能处理器)与 CPU_d(下标 d 表示 disabled，禁用部分功能后的处理器). 其中，CPU_d 是禁用了所有可能导致时序异常功能的处理器模型(比如，高速缓存模块可能导致时序异常，则在 CPU_d 中，高速缓存会被禁用，每次访存都会产生等价于缓存未命中的执行时间). CPU_f 则是一个有原处理器完整功能的模型. 简而言之，CPU_d 是一个被用来模拟处理器在最坏情况下执行的模型，CPU_f 则是用来反映处理器运行时的真实状态. 显然，在一般情况下，CPU_f 所需的运行时间应当比 CPU_d 要短，运行速度更快，因此如果 CPU_f 运行得比 CPU_d 还要慢，则意味着发生了时序异常.

在验证时，2 个例化的处理器模型从相同的初始状态开始，并在相同的指令流输入下开始运行. 如果 CPU_f 在某一时刻比 CPU_d 执行得要更快(即 CPU_f 相较于 CPU_d 完成了更多的指令)，那么 CPU_d 应当在接下来的执行中始终慢于 CPU_f，否则即为检测到了一次时序异常. 这个条件以断言的形式在模型中表示，模型检查其通过检查状态机在所有合法转移下能够到达的状态是否满足断言，来判断处理器在运行程序时是否存在反直觉型时序异常.

文献 [33−34] 使用了有界模型检查 BMC 工具对一个采用了 Tomasulo 方案的超标量处理器进行了时序异常验证，并成功找到了该处理器模型在一段程序下表现出的时序异常，同时给出了处理器在的这个时序异常下周期精确的执行记录. 然而，这个方法也存在着一些缺陷. 由于该方案没有说明处理器模型是如何建立的，且在文献 [33] 中也仅提到了针对处理器的 VHDL 部分设计代码进行自动修改，因此尚不清楚处理器被抽象成为一个有限自动机的过程. 本文认为，这一过程目前仍需要验证人员的手动参与，因为在实现 CPU_d 模型的过程中，需要人为地禁用那些可能会导致时序异常的部件，由此带来的处理器功能上的缺失往往也需要进行手动补全.

另一个问题在于文献 [33] 在抽象过程中略去了过多的信息，导致模型与实际的硬件不一致. 文献 [16] 提到在抽象处理器模型的过程中，处理器的输入数据也作为抽象的一部分被略去，这使得 CPU_d 与 CPU_f 这 2 个模型在执行时可能有着不同的输入，因而可能出现因不同输入数据而产生的反直觉型时序异常，从文献 [16] 对时序异常的定义来看，这种情况不能构成一次时序异常.

3.2 放大型时序异常的验证

文献 [19, 27, 35] 针对放大型时序异常展开了一系列验证工作，专为在流水线处理器中验证放大型时序异常的工作设计了一个标准流水线模型如图 7 所示. 这个模型着重描述处理器的时间行为，主要有 4 点：

1)模型不处理具体的每条指令，只对指令的类型进行区分，如读取/写入、算术/逻辑类型等.

2)模型不进行具体的计算过程(如算术运算的计算过程)，只有涉及指令在流水线中流动的逻辑、高速缓存的交互以及对总线的访问才会被建模.

3)高速缓存的内容不包括在模型之中，模型只考虑高速缓存命中或时不命中对流水线的影响.

4)仅考虑 2 条连续的指令之间产生的具体影响，流水线中其他指令之间可能造成的影响则直接考虑最坏情况.

对于模型具体的建模与参数在 1.3 节中进行了详细描述，此处不再赘述.

标准流水线模型并不直接模拟流水线运行的全过程，而是每次检查 2 条指令在流水线中的运行情况. 其中一条指令作为下游指令被放在流水线中的任意一段，另一条指令则在上游发射. 以此方式，可以模拟验证 2 条指令在流水线中、间隔任意距离的执行情况. 通过在初始化阶段将流水线初始化为不同的状态，模型检查可以借此遍历所有类型指令之间以及所有可能延迟下的执行能够到达的状态. 在模型检查中，一次运行结束的标志为下游指令流出流水线，在一次运行时，机器检查指令状态中的 *delay* 字段，如果其为正值，则表示上游指令影响了下游指令的执行，因此可能发生了一次放大型时序异常.

值得一提的是，*delay* 值为正仅仅是一个判断是

否发生了放大型时序异常必要但不充分的条件. Binder 等人[19,27,35]使用的模型检查方法仅能检出那些可能发生放大型时序异常的实例，而对于每个检出的实例，还需要人工判断它是否真的是一次放大型时序异常.

文献 [19, 27, 35] 使用 UCLID5 来进行有界模型检查，尽管存在些许不足(存在误报)，这个方案仍然成功地应用在了 PRET[36]，Patmos[37]，SIC[29]，K1[38] 等处理器的验证工作中. 相较于 Eisinger 等人[33]的工作，标准流水线模型有着更加详细具体的建模方案，对于一个任意处理器架构来说，标准流水线模型提供了一个较为统一的模型表达方式，使得验证无需单独对每个处理器架构进行适配，减少了验证所需的建模工作量. 而在模型检查方面，文献 [19, 27, 35] 的方案没有让模型直接执行指令，而是分析指令之间的组合方式，相较于直接执行指令减少了分析所涉及的状态空间大小和模型运行所需时间，提高了分析性能.

3.3 乱序处理器的验证

Binder 等人[13]对数种时序异常的定义进行了分析，并得出结论：现有的时序异常定义均不够精确，从而无法系统地检测乱序执行处理器中的时序异常. 因此，他们以因果关系的时序异常定义为基础，开展了一系列对乱序处理器进行验证的工作.

在 Gebhard[15]提出的时序异常定义中，没有提到硬件状态的定义，显然，这是缺乏对乱序处理器的建模所导致的，因此其定义难以被应用在各式处理器的异常检测中. 类似的还有 Wenzel 等人[18]提出的时序异常定义，其定义没有具体地描述指令执行时间是如何从处理器硬件或模型中得到的，因此，Wenzel 等人[18]的定义仅能应用在简单的顺序流水线中(指令在简单流水线中的执行延时是显而易见的)，对于复杂的乱序指令处理器，如何从处理器中抽象出指令的执行延时就成为了最大的问题. 为此，他们提出了基于因果关系的时序异常定义.

但是，即使一些定义描述了如何抽象出硬件状态，它们在检测乱序处理器的时序异常时仍然存在缺陷. 在 Kirner 等人[25]的定义中，硬件状态由一个个组件所构成，这虽然可以将一个具体的处理器抽象为定义中的模型，但是没有要求在何种程度上进行抽象，即哪些硬件结构应当被视为整个组件进行处理. 事实上，一些时序异常只会出现在特定粒度所抽象出的模型之中[13]. 因此，必须指定从具体硬件结构到抽象组件之间的映射方式. Binder 等人[14]提出的基于因果关系的时序异常定义，面向处理器中的验证工作并进行了适配. 在组件抽象方面，基于因果关系的定义允许任意非空的功能单元构成一个组件，由此可以参数化地构建硬件状态. 在指令执行的局部性方面，他们的定义使用每条指令在组件上的执行时间来表示，因而可以很好地和乱序处理器的结构相匹配.

在验证方面，Binder 等人[14]使用 TLA+ 语言来对一个使用了 Tomasulo 算法的不定配置处理器进行建模. TLA+语言可以构建一个由状态组成的转移系统，系统从初始状态开始，由转移关系定义其每次转移之后可能达到的状态——对应于一个周期精确的流水线执行指令的过程. TLC 模型检查器因此能够遍历所有可能的执行 trace，并将它们两两比较来自动检测其中可能存在的反直觉型时序异常. 该验证方法能够回答下列问题：

1)某一超标量处理器配置是否存在反直觉型时序异常.

2)某一特定的程序或指令流是否存在反直觉型时序异常.

3)在限制模型初始状态给定程序的情况下，是否还会发生反直觉型时序异常.

相较于标准流水线模型的验证工作，基于因果关系的时序异常验证针对于反直觉型时序异常. Binder 等人[14]的定义在乱序处理器中时序异常的验证精确程度高、可行性好、验证方便，目前能较好地部署到实际验证中.

3.4 消除反直觉型时序异常

在第 2 节中，本文分析了导致时序异常的数种原因，并将它们大致分类为调度策略导致的时序异常、高速缓存导致的时序异常以及处理器组件导致的时序异常. 其中，前两者可以简单地通过禁用或替换一些功能和组件来避免时序异常的发生，比如，要求处理器中的所有资源都顺序执行来避免调度型时序异常，使用便笺式存储器(scratchpad memory)替代高速缓存来避免高速缓存中的时序异常. 而来自处理器内部组件之间的互相影响则极为复杂，几乎每个不同的处理器内部的组件都大相径庭，使得存在于处理器内部的时序异常往往难以被定位与解决.

Hahn 等人[29]从流水线运行时的性质入手，提出处理器在满足“单调性”的条件下，不会存在反直觉型时序异常. 他们基于此设计了一个“严格顺序核心”(strict in-order core, SIC)处理器，并通过形式化方法严格地证明了 SIC 中不存在反直觉型时序异常.

图 11(a)展示了不满足单调性而产生反直觉型时序异常的情况, c 表示流水线执行时达到的某一状态, 该状态有 2 种可能的执行分支: 一种为局部时间更长的执行; 另一种为局部时间更短的执行(在图中分别被表示为 c_w 与 c_b, Hahn 等人[29]分别称其为“进展更少”和“进展更多”的执行). 然而, 在经过数个周期的执行过后, c'_w 反而比 c'_b 有着更多的执行进展, 因而构成了一次反直觉型时序异常. Hahn 等人[29]总结该情况, 并认为: 反直觉型时序异常发生的条件, 就是硬件执行之间的序关系相较于执行进展的序关系是非单调的($\sqsubseteq$). 因此, 如果满足硬件执行时的单调性, 就能够消除时序异常.

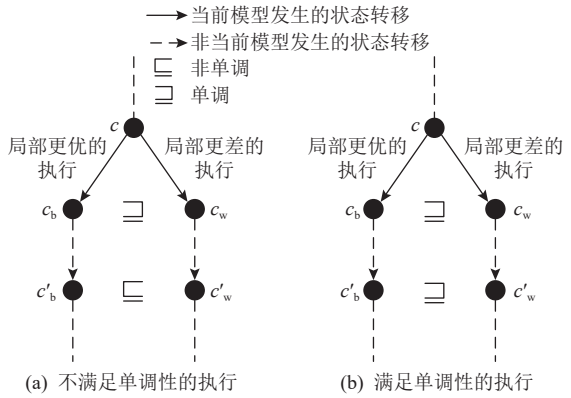


Fig. 11 Monotonicity in SIC pipeline[29]

图 11 SIC 流水线中的单调性[29]

为了满足硬件执行时的单调性, SIC 采用了数种设计策略. 比如 SIC 中没有先前提及的分支预测、指令预取等技术. 除此之外, Hahn 等人[39]认为访存总线上的访存顺序发生重排也是导致处理器中发生时序异常的重要原因(在本文中, 这应属于调度策略导致的时序异常), 因此为了保证所有访存严格按照指令的先后顺序完成, SIC 会主动停顿流水线, 使得在流水线中存在一条访存指令时, 后续指令不能进行任何访存操作(包括取指), 这也是 SIC 被命名为严格顺序核心处理器的原因[29].

上述种种方案使得 SIC 处理器满足了一个特殊的性质: 在流水线中的任意一条指令的执行, 不会受在程序顺序上后续指令执行的影响. 这一性质是至关重要的, 因为这意味着可以对流水线中最后一个流水段的指令进行独立分析, 判断其是否可能产生时序异常, 在保证最后一个流水段不会产生时序异常的情况下, 就可以利用该结果对流水线中倒数第 2 段指令的执行状态再进行分析, 以此类推, 最终可以证明, 整个处理器在执行时不会产生时序异常. 事实上, 这就是 SIC 进行形式化证明的过程.

为了证明 SIC 的单调性以及不存在时序异常等性质, Hahn 等人[29]为 SIC 手动建立了一个形式化的执行模型, 这个模型能够周期精确地模拟指令在每个流水段的流动.

相较于普通的 5 段流水处理器, SIC 仅比它们要慢 6%~7%, 但是借助于其无时序异常的性质, 在 SIC 进行 WCET 分析的速度要比普通处理器快数个数量级[29], 这也充分说明了时序异常对于 WCET 分析的重要性.

SIC 的不是唯一一个成功使用限制处理器严格顺序执行方案消除时序异常的案例. 在 SIC 之后, Gruin 等人[30]采用了类似的方法构造了一个在允许超标量执行的情况下, 不存在时序异常的处理器 MINOTAuR. 其缺点是, 目前所有不存在时序异常的处理器, 都需要为其构建形式化模型, 并进行严格的证明, 相较于先前自动验证时序异常的工作, 这些方法仍然需要大量的人工作业, 效率较低.

3.5 时间语义对 WCET 分析的影响

程序的时间行为难以分析的一个关键问题就在于, 现有的指令集架构(instruction set architecture, ISA)往往仅包含了程序的逻辑语义, ISA 只关注程序的逻辑功能是否正确而忽视了程序的时间行为[40]. 这也使得现代处理器在设计时过度地注重程序的平均性能而非单条指令的执行时间, 进而难以分析基本块、程序的执行时间. 因此, 如果能够在 ISA 层次以及硬件层次引入时间语义, 可能有助于解决时序异常的问题, 降低 WCET 分析的难度.

加拿大伯克利分校的文献 [36, 41−42] 将时间语义引入指令集以及硬件中, 构建了 PRET 处理器模型以及 FlexPRET 处理器. FlexPRET 是一个硬件多线程处理器, 使用时间语义指令 du(delay until)以及指令 ee(exception on expire)来限制代码段的执行时间. 处理器中的硬件线程互相隔离, 使得在 WCET 分析时无需考虑线程之间的互相干扰. 这也使得单个线程所拥有的硬件资源相对独立, 一定程度上减少了因资源竞争导致的时序异常.

中国科学技术大学也提出了能够精确控制程序执行时间行为的时间语义指令集(time triggered instruction set, TTI)[40], 并实现了一个基于该指令集的处理器 RPU(real-time processing unit)[43]. RPU 也是一个硬件多线程处理器, 但区别于 FlexPRET, 线程的执行时间由 TTI 中的 addtk 与 tkend 指令控制. TTI 还为程序的 I/O 操作赋予了时间语义, 允许用户直接控制程序

的时间行为，比如在某一特定时刻控制处理器进行读取或者写入操作. 虽然使用时间语义指令集控制代码段的执行时间仍然需要 WCET 分析的支持，但是时间语义的引入让程序的时间行为更加确定，进而可能减少时序异常的发生，在组合代码段的执行时间时，也可以借助于时间语义提供的信息，便于计算整个程序的 WCET.

3.6 时序异常的验证与消除小结

本节介绍了时序异常验证与消除方面的各类工作. 表 2 总结了这些方法. 在检查与验证时序异常方面，学界普遍使用先建模再验证的方式展开工作. 不同学者采用了不同的工具与方法对构建的模型进行验证，其中文献 [33−34] 采用模型模拟处理器执行的方式验证时序异常，计算量大. 后来的学者普遍采用模型检查的方式来验证. 在模型构建方面，Binder 等人[14] 为反直觉型时序异常验证所构建的模型以及文献 [33−34] 的方案都需要手动为每个不同的处理器建模，工作量较大. 而对于放大型时序异常，由于标准流水线模型的存在，对处理器的适配性较强，建模验证工作量较小. 从验证结果来看，Binder 等人[14] 的工作对于反直觉型时序异常的验证工作精确程度高、可行性好、验证方便，目前能较好地部署到实际验证工作中.

Table 2 Comparison of Methods to Verify and Eliminate Timing Anomalies

表 2 验证与消除时序异常方法对比

方法提出者	类型	验证/消除方式	验证/消除所用工具	优缺点
Eisinger 等人[33]		模型模拟执行	BMC	建模困难、验证计算量大
Binder 等人[35]	放大型异常	模型检查	UCLID 5	流水线模型适配性强
Binder 等人[14]	反直觉型异常	模型检查	TLA+	精确、可行性高、功能多
Hahn 等人[29]		人工证明		结论严格但不具备普遍性
Liu 等人[36]		时间语义隔离		能够减少但不能完全消除时序异常

在消除时序异常方面，目前仅有 Hahn 等人[29] 给出了消除时序异常的具体方法，并配有严格的形式化证明，但是这个工作仍然不具备普遍性，因为对于每个不同的处理器，都需要经过人工证明才能确保时序异常不存在. 文献 [36, 41−42] 提出的时间语义隔离的方式，虽然可以直接控制处理器的时间行为，但是执行时间方面仍然依赖于传统的 WCET 分析，不能完全摆脱时序异常的影响.

总得来说，目前针对时序异常验证与消除的工作仍有很多问题存在，现有的方法要么结果不尽如人意，要么不具备通用性与普遍性，仅能在少数几种处理器上展开. 因此，未来针对时序异常验证与消除工作的最大的挑战在于如何将这两者结合到一起，实现对一般处理器通用的时序异常验证，或是形成一种处理器设计规范，确保所设计的硬件结构不会发生时序异常.

4 总结与展望

时序异常是 WCET 分析技术中的一个关键问题，虽然已有大量针对时序异常的研究，但是鲜有 WCET 分析工具以及硬件平台考虑了时序异常对 WCET 分析的影响. 本文总结了目前针对时序异常的工作，并从概念定义、导致时序异常的原因、验证与消除时序异常等方面进行了分析，认为导致现有的工作主要存在 3 个问题：

1)时序异常定义脱离了 WCET 分析本身. 现有的定义往往以硬件平台的形式化建模为基础，构造出一组满足时序异常的执行实例. 然而，在 WCET 分析中，对基本块执行时间的计算并不需要直接搜索程序执行时的实例，这一过程与时序异常的定义相偏离，导致现有的时序异常定义往往很难直接应用于 WCET 分析工具，仅停留在理论中. 目前，只有 SIC 消除了时序异常并实验验证了对 WCET 分析带来的加速作用，但是并未透露所采用的 WCET 分析工具以及将时序异常应用到其中的方法，且所使用的时序异常定义仅局限于 SIC 中因高速缓存所导致的异常，不具备通用性.

2)时序异常的原因描述不完全. 时序异常可能在不同的硬件设计、不同的体系结构层次、不同的程序执行粒度下发生. 目前提出的原因只能描述某一类时序异常，并且在尚未确定体系结构的情况下，无法确定系统中是否存在其他因素导致的异常.

3)模型与硬件平台缺少一致性以及适用性. 模型检查是目前检测时序异常常用的手段，这一过程通常都在经过抽象的硬件模型中展开，但是建立的模型与硬件本身之间往往存在不一致性，导致硬件设计中存在的时序异常可能在模型中不再存在，影响检测结果的可靠性. 另一个问题在于时序异常定义所依赖的模型与硬件模型不一致，硬件体系结构本身无法或者难以抽象为适用于时序异常定义的模型，这些问题使得时序异常验证的安全性得不到保障.

时序异常的概念来自于 WCET 分析，然而，目前

鲜有工具在 WCET 计算过程中考虑了时序异常的影响. 尽管目前这些分析结果的安全性有其他理论保障, 但是 SIC 的例子说明, 如果能够将针对时序异常的分析纳入 WCET 分析工具之中, 可以大大减少对不存在时序异常的体系结构的分析时间(相较于传统处理器, SIC 在 WCET 上的分析速度要快约 200 倍[29]). 因此, 时序异常对于优化 WCET 分析、减少 WCET 的分析时间具有重要的价值. 本文认为, 未来对时序异常的工作应当重点基于 WCET 分析方式的时序异常定义, 将时序异常的概念结合到 WCET 分析之中, 构造适用于 WCET 分析方式的硬件执行模型, 以此来检测/验证时序异常.WCET 分析也应主动考虑时序异常的影响, 如时序异常在现有的 WCET 分析方法中的表现形式, 以及是否会影响 WCET 分析结果的安全性等. 此外, 还可以尝试在指令及架构层以及硬件平台引入时间语义, 构造时间语义指令集(如 TTI[43]), 来辅助时序异常相关的工作或是简化 WCET 分析的计算过程.

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

参 考 文 献

[1] Rochange C, Sainrat P. A Context-Parameterized Model for Static Analysis of Execution Times[M]. Berlin: Springer, 2009: 222−241

[2] Puschner P, Burns A. Guest editorial: A review of worst-case execution-time analysis[J]. Real-Time Systems, 2000, 18(2): 115−128

[3] 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

[4] Li Y T, Malik S, Wolfe A. Efficient microarchitecture modeling and path analysis for real-time software[C]//Proc of the 16th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 1995: 298−307

[5] Healy C, Whalley D, Harmon M. Integrating the timing analysis of pipelining and instruction caching[C]//Proc of the 16th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 1995: 288−297

[6] Lim S S, Bae Y H, Jang G T, et al. An accurate worst case timing analysis technique for RISC processors[C]//Proc of the 15th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 1994: 97−108

[7] Lim S S, Han J H, Kim J, et al. A worst case timing analysis technique for multiple-issue machines[C]//Proc of the 19th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 1998: 334−345

[8] Lundqvist T, Stenström P. An integrated path and timing analysis method based on cycle-level symbolic execution[J]. Real-Time Systems, 1999, 17(2): 183−207

[9] Greger O. Worst-case execution time analysis for modern hardware architectures[C]//Proc of the 2nd ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems. New York: ACM, 1997: 47−55

[10] Theiling H, Ferdinand C. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis [C]//Proc of the 19th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 1998: 144−153

[11] Lundqvist T, Stenström P. Timing anomalies in dynamically scheduled microprocessors[C]//Proc of the 20th IEEE Real-Time Systems Symp. Piscataway, NJ: IEEE, 1999: 12−21

[12] Reineke J, Wachter B, Thesing S, et al. A definition and classification of timing anomalies[C/OL]//Proc of the 6th Int Workshop on Worst-Case Execution Time Analysis. 2006[2023-05-19]. http://drops.dagstuhl.de/opus/volltexte/2006/671

[13] Binder B, Asavoae M, Hedia B B, et al. Is this still normal? Putting definitions of timing anomalies to the test[C]//Proc of the 27th Int Conf on Embedded and Real-Time Computing Systems and Applications (RTCSA). Piscataway, NJ: IEEE, 2021: 139−148

[14] Binder B, Asavoae M, Brandner F, et al. The role of causality in a formal definition of timing anomalies[C]//Proc of the 28th Int Conf on Embedded and Real-Time Computing Systems and Applications (RTCSA). Piscataway, NJ: IEEE, 2022: 91−102

[15] Gebhard G. Timing anomalies reloaded[C/OL]//Proc of the 10th Int Workshop on Worst-Case Execution Time Analysis (WCET 2010). 2010[2023-12-01]. http://drops.dagstuhl.de/opus/volltexte/2010/2820

[16] Cassez F, Hansen R R, Olesen M C. What is a timing anomaly? [C/OL]//Proc of the 12th Int Workshop on Worst-Case Execution Time Analysis (WCET'2012). 2012[2023-05-23]. http://drops.dagstuhl.de/opus/volltexte/2012/3552

[17] Graham R L. Bounds for certain multiprocessing anomalies[J]. Bell System Technical Journal, 1966, 45(9): 1563−1581

[18] Wenzel I, Kirner R, Puschner P, et al. Principles of timing anomalies in superscalar processors[C]//Proc of the 5th Int Conf on Quality Software (QSIC'05). Piscataway, NJ: IEEE, 2005: 295−303

[19] Jan M, Asavoae M, Schoeberl M, et al. Formal semantics of predictable pipelines: A comparative study[C]//Proc of the 25th Asia and South Pacific Design Automation Conf (ASP-DAC). Piscataway, NJ: IEEE, 2020: 103−108

[20] Kirner R, Kadlec A, Puschner P. Worst-case execution time analysis for processors showing timing anomalies[R]. Wien, Austria: Institut für Technische Informatik, 2009

[21] Hahn S, Reineke J, Wilhelm R. Towards compositionality in execution time analysis: Definition and challenges[J]. ACM SIGBED Review, 2015, 12(1): 28−36

[22] Wilhelm R, Grund D, Reineke J, et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009, 28(7): 966−978

[23] Schneider J. Combined Schedulability and WCET Analysis for Real-time Operating Systems[M]. Maastricht, Germany: Shaker, 2003

[24] Berg C. PLRU cache domino effects[C/OL]//Proc of the 6th Int Workshop on Worst-Case Execution Time Analysis (WCET'06). 2006[2023-05-23]. http://drops.dagstuhl.de/opus/volltexte/2006/672

[25] Kirner R, Kadlec A, Puschner P. Precise worst-case execution time

analysis for processors with timing anomalies[C]//Proc of the 21st Euromicro Conf on Real-Time Systems. Piscataway, NJ: IEEE, 2009: 119−128

[26] Pedersen M R, Bacci G, Larsen K G. A faster-than relation for semi-Markov decision processes[J]. Electronic Proceedings in Theoretical Computer Science, 2020, 312: 29−42

[27] Binder B, Asavoae M, Brandner F, et al. Formal modeling and verification for amplification timing anomalies in the superscalar TriCore architecture[J]. International Journal on Software Tools for Technology Transfer, 2022, 24(3): 415−440

[28] Lundqvist T. A WCET analysis method for pipelined microprocessors with cache memories[D]. Västra Götaland, Sweden: Chalmers University of Technology, 2002

[29] Hahn S, Reineke J. Design and analysis of SIC: A provably timing-predictable pipelined processor core[C]//Proc of the 39th IEEE Real-Time Systems Symp (RTSS). Piscataway, NJ: IEEE, 2018: 469−481

[30] Gruin A, Carle T, Rochange C, et al. MINOTAuR: A timing predictable RISC-V core featuring speculative execution[J]. IEEE Transactions on Computers, 2023, 72(1): 183−195

[31] Asavoae M, Hedia B B, Jan M. Formal executable models for automatic detection of timing anomalies[C/OL]//Proc of the 18th Int Workshop on Worst-Case Execution Time Analysis (WCET'2018). 2018[2023-05-29]. http://drops.dagstuhl.de/opus/volltexte/2018/9748

[32] Thesing S. Safe and precise WCET determination by abstract interpretation of pipeline models[D]. Saarland, Germany: Saarland University, 2004

[33] Eisinger J, Polian I, Becker B, et al. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis[C]// Proc of the 9th IEEE Design and Diagnostics of Electronic Circuits and Systems. Piscataway, NJ: IEEE, 2006: 13−18

[34] Biere A, Heule M, Maaren H. Handbook of Satisfiability[M]. Amsterdam: IOS Press, 2009

[35] Binder B, Asavoae M, Brandner F, et al. Scalable detection of amplification timing anomalies for the superscalar tricore architecture[C]//Proc of the 25th Formal Methods for Industrial Critical Systems. Berlin: Springer, 2020: 151−169

[36] Liu I, Reineke J, Lee E A. A PRET architecture supporting concurrent programs with composable timing properties[C]//Proc of the 44th Asilomar Conf on Signals, Systems and Computers. Piscataway, NJ: IEEE, 2010: 2111−2115

[37] Schoeberl M, Abbaspour S, Akesson B, et al. T-CREST: Time predictable multi-core architecture for embedded systems[J]. Journal of Systems Architecture, 2015, 61(9): 449−471

[38] Dinechin B, Amstel D, Poulhies M, et al. Time-critical computing on a single-chip massively parallel processor[C/OL]//Proc of the 21st Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2014[2023-10-23]. http://ieeexplore.ieee.org/document/6800311

[39] Hahn S, Reineke J, Wilhelm R. Toward Compact Abstractions for Processor Pipelines[M]. Berlin: Springer, 2015

[40] Chen Xianglan, Li Xi, Wang Chao, et al. Reasearch 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)

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

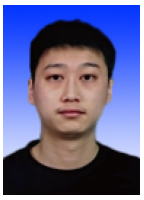
[42] Zimmer M, Broman D, Shaver C, et al. FlexPRET: A processor platform for mixed-criticality systems[C]//Proc of the 19th IEEE Real-Time and Embedded Technology and Applications Symp (RTAS). Piscataway, NJ: IEEE, 2014: 101−110

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



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体系结构.



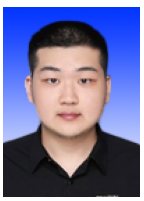
Zhu Yixuan, born in 1998. Master candidate. His main research interest includes computer architecture.

朱钇宣, 1998 年生. 硕士研究生. 主要研究方向为计算机体系结构.



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

陈香兰, 1977 年生. 博士. 主要研究方向为系统软件.



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

龚小航, 1999 年生. 硕士研究生. 主要研究方向为计算机体系结构.



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

高银康, 1998 年生. 博士. 主要研究方向为计算机体系结构.



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

李　曦, 1963 年生. 教授. 主要研究方向为计算机体系结构.