计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220908
60(2): 294−310, 2023

中断驱动型航天嵌入式软件原子性违反检测方法

于婷婷[1,2] 李 超[1,2] 王博祥[1] 陈 睿[1,2] 江云松[1,2]
[1](北京轩宇信息技术有限公司 北京 100190)
[2](北京控制工程研究所 北京 100190)
(yutingting@sunwiseinfo.com)

Atomicity Violation Detection for Interrupt-Driven Aerospace Embedded Software

Yu Tingting[1,2], Li Chao[1,2], Wang Boxiang[1], Chen Rui[1,2], and Jiang Yunsong[1,2]
[1] (*Beijing Sunwise Information Technology Ltd.*, *Beijing* 100190)
[2] (*Beijing Institute of Control Engineering*, *Beijing* 100190)

Abstract The dependability of embedded software is vital to the success of the aerospace mission. Aerospace embedded software extensively uses interrupt-driven concurrency mechanisms. However, uncertain interleaving execution of interrupts may cause concurrency bugs, which could result in serious safety problems. Researches had showed that atomicity violation was the most prominent interrupt concurrency bug. Current atomicity violation detection methods for interrupt-driven embedded software cannot achieve both high precision and high scalability, and their effectiveness on real embedded software has not been evaluated. To significantly improve the ability of techniques in bug detection, we design an empirical study on 82 atomicity violations in aerospace embedded software and gain their manifestation characteristics of atomic region range, the number of interrupt nesting levels, shared data access interleaving pattern, access patterns and access granularities. On this basis, we present intAtom-S, a precise and efficient static detection technique for atomicity violations. Firstly, to realize accurate analysis of shared data, intAtom-S constructs a fine-grained memory model parameterized by numerical invariants, and introduces rule-based method to eliminate flag variable accesses. Moreover, it can handle array elements as well as shared I/O addresses. Then, a lightweight data flow analysis technique is used to find all access interleavings that match the patterns as potential bug candidates. Finally, intAtom-S adopts a consecutive access pair feasibility analysis and access interleaving feasibility analysis to prune the infeasible paths. intAtom-S gradually eliminates false positives, and obtains the final atomicity violations. Experiments on a benchmark and 8 real-world aerospace embedded software show that intAtom-S reduces the false positive rate by 72% and improves the detection speed by 3 times, compared with the state-of-the-art methods. Furthermore, it can finish analyzing the real-world aerospace embedded software very fast with an average false positive rate of 8.9%, while finding 23 bugs that had been confirmed by developers.

Key words aerospace embedded software; interrupt; concurrency bugs; bug characteristics; atomicity violation; static analysis

摘 要 嵌入式软件的可信性对航天任务的成败至关重要.航天嵌入式软件广泛采用中断驱动的并发机制,不确定的中断抢占可能导致并发缺陷,引发严重的安全问题.研究表明原子性违反是中断并发缺陷中最突出的一类缺陷模式.目前面向中断驱动型嵌入式软件的原子性违反检测方法都无法同时实现高精度和高可扩展性,且其对真实航天嵌入式软件的有效性尚未得到证实.为了有效提升检测该类缺陷的精度

收稿日期: 2022−10−26; **修回日期**: 2022−12−29
基金项目: 国家自然科学基金项目(61802017, 62192730)
This work was supported by the National Natural Science Foundation of China (61802017, 62192730).
通信作者: 陈睿(chenrui@sunwiseinfo.com)

与效率，对 82 个航天嵌入式软件原子性违反进行实证研究，获得该类缺陷在原子区范围、中断嵌套层数、访问交错模式、共享数据访问方式、访问粒度等 5 个方面的表现特征. 并在此基础上，提出一个精确、高效的原子性违反静态检测方法 intAtom-S. 首先，基于一个由数值不变式参数化的细粒度内存访问模型，引入基于规则的方法剔除标志变量访问，并采用抽象解释进行精确的共享数据分析，该分析能将共享数据访问粒度精确至数组元素，并可识别共享的内存映射 I/O 地址. 然后，使用轻量级数据流分析技术匹配符合缺陷访问交错模式的所有并发三次访问序，作为潜在的原子性违反缺陷候选. 最后，采用基于路径条件的约束求解对缺陷候选中的串行访问对和并发三次访问序进行路径可行性分析，逐步消除误报，得到最终的原子性违反结果. 在基准测试集和 8 个真实航天嵌入式软件上的实验表明，与目前最先进的方法相比，intAtom-S 误报率降低了 72%，检测效率提高了 3 倍. 此外，该方法能够快速完成对真实航天嵌入式软件的分析，平均误报率仅为 8.9%，并发现了 23 个已被开发人员确认的缺陷.

关键词 航天嵌入式软件；中断；并发缺陷；缺陷特征；原子性违反；静态分析

中图法分类号 TP311

嵌入式软件的可信性对航天任务成败至关重要，即使一个微小的软件缺陷也可能导致重要任务的失败. 航天嵌入式软件广泛采用中断驱动的并发机制，与硬件频繁地进行交互并实时地响应外界事件[1]. 当一个系统中的大量处理都由中断发起时，我们称这类系统为中断驱动型系统[2].

中断驱动型嵌入式软件使用大量的共享数据来实现主任务和中断，以及不同中断之间的通信和数据交互. 如图 1 所示，航天嵌入式软件通常由主任务和多个中断服务程序构成. 主任务的结构是一个无限循环，等待中断的触发进行相应的操作. 中断由来自软件或硬件（如定时器、外设和 I/O 端口）的中断信号触发，例如由内部事件及“异常”，或由接收字节的串行端口控制器触发. 然后，处理器通过挂起其他当前活动、保存其状态并执行中断处理程序来响应当前事件. 待中断处理完成后，处理器恢复其正常活动. 由于中断可能在任意时刻发生并抢占正在执行的计算，因此，如果对共享数据的访问没有被恰当地同步，则不确定的中断抢占可能会导致数据访问冲突，引起严重的安全问题. 据中国空间技术研究院软件产品保证中心统计[3]，在航天器总装集成测试（assembly, integration & test, AIT）阶段发现的软件质量问题中，约 80% 为中断相关的并发缺陷. 这些缺陷在经过各种严格的软件测试后仍然被遗漏，是航天嵌入式软件开发中最难以消除的缺陷类型之一. 根据文献 [4] 对航天嵌入式软件并发缺陷的研究，原子性违反是中断并发缺陷中最突出的一类缺陷模式. 这种模式指的是程序员对共享数据连续多次访问的原子性执行期望由于中断抢占而遭到破坏，导致非预期的程序行为.

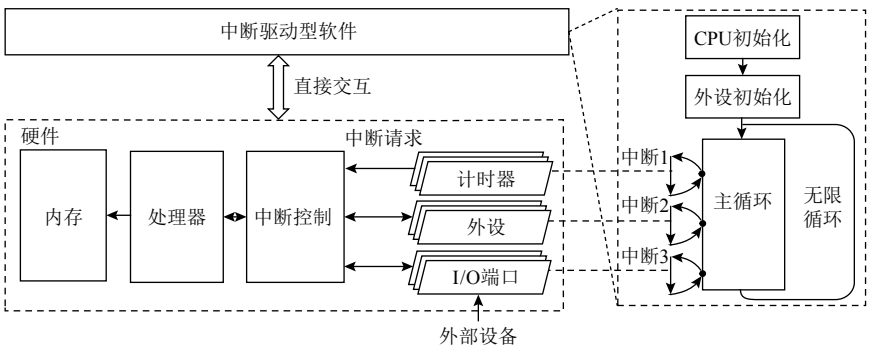

Fig. 1 Interrupt-driven model in aerospace embedded software

图 1 航天嵌入式软件的中断驱动模型

原子性违反检测是国内外研究的热点，已有方法大多面向多线程程序[5-14]，但这些方法不能直接应用于中断驱动型航天嵌入式软件，这是由于中断驱动型航天嵌入式软件与多线程程序在并发模型和编程范式方面存在显著差异. 例如，任务与中断之间或中断与中断之间具有非对称抢占关系，而线程之间可以互相抢占，异步并发事件及其优先级之间的隐式依赖关系，使用于检测竞争的 happens-before 关系

复杂化，这导致适用于2类程序的缺陷检测方法存在差异；中断驱动型航天嵌入式软件一般缺乏底层并发编程框架，经常采用基于自定义标志变量的同步方式，而多线程程序多采用标准并发控制原语，自定义标志变量的存在极易导致检测结果存在大量误报；此外，中断驱动型航天嵌入式软件依赖大量的内存映射I/O地址与外设进行交互，若不能恰当识别这些数据，将使检测结果存在漏报. 这些差异对检测方法的有效性具有决定性作用，直接应用针对多线程程序的检测方法面临检测结果不精确的问题.

目前，仅有少量工作针对中断驱动型程序中的原子性违反[4,15]. 文献[4]采用静态分析的方法，可扩展到工业规模的嵌入式软件，但存在较多误报；文献[15]采用程序模型检验方法，但仅能扩展到百行规模的程序. 一方面，文献[4, 15]都无法同时实现高精度和高可扩展性；另一方面，中断驱动型航天嵌入式软件的硬件依赖性强且多采用裸机编程，其并发缺陷的表现形式具有领域相关特征，且已有方法的有效性都未经过真实航天嵌入式软件的评估.

本文面向真实中断驱动型嵌入式软件中的原子性违反，研究精确且高效的静态检测方法. 首先对2009—2022年中国空间技术研究院第三方评测机构报告的82个原子性违反缺陷进行特征研究，获得该类缺陷在访问交错模式、共享数据访问方式、访问粒度等3方面的表现特征. 在此基础上，提出了一个精确、高效的静态缺陷检测方法intAtom-S. 该方法的核心思想是逐阶段地采用更精确的技术消除不可行路径导致的误报，从而同时实现更少的误报和更好的性能. intAtom-S主要分为3个阶段：1)基于一个由数值不变式参数化的细粒度内存访问模型，引入基于规则的方法剔除标志变量访问，并采用抽象解释进行精确的共享数据分析，以识别程序中的所有可能造成缺陷的共享访问点. 该分析能够排除标志变量访问，将共享数据访问粒度精确至数组元素，并可识别共享内存映射I/O地址. 2)使用轻量级数据流分析技术匹配符合交错模式的所有并发3次访问序作为潜在的原子性违反缺陷候选. 3)采用基于路径条件的约束求解对缺陷候选中的串行访问对和并发3次访问序进行路径可行性分析，逐步消除误报，得到最终的原子性违反结果.

本文实现了一个面向C程序的中断原子性违反静态检测工具intAtom-S，并在基准测试集Racebench 2.1[16]和8个真实航天嵌入式软件上进行了实验评估. 结果表明，与当前的研究工作相比，intAtom-S的误报率降低了72%，同时检测速度提高了3倍. 此外，intAtom-S可以非常快地完成对真实航天嵌入式软件的分析，发现了23个经开发人员确认的错误，且平均误报率仅为8.9%. 本文的部分内容已发表于文献[17]和[18]，相比已发表内容，intAtom-S基于更精确、更细粒度的共享数据识别结果，可以处理数组元素访问、标志变量访问导致的误报，显著提高了缺陷检测的精度.

本文的主要贡献包括3个方面：

1)对来自真实航天嵌入式软件的82个并发缺陷进行了全面、深入的实证研究，揭示了航天嵌入式软件原子性违反的表现形式特征.

2)提出了一个精确、高效的中断驱动型程序原子性违反静态检测方法intAtom-S.

3)在基准测试集和真实航天嵌入式软件上进行实验评估，验证了本文方法的有效性.

1 原子性违反特征研究

本节首先介绍原子性违反模式并给出真实案例；然后，对本文缺陷特征的研究方法进行阐述；最后，给出特征研究结果及对应的方法.

1.1 原子性违反缺陷模式

原子性违反缺陷模式指的是程序员对共享数据的连续多次访问具有原子性期望,而其他并发流的抢占执行打破了这种期望，导致非预期的程序行为. 对于中断驱动型程序，该类缺陷主要发生在对单个共享变量的并发访问中. 由于程序员通常以顺序思维进行程序设计，往往会假设对同一变量的顺序访问是原子执行的，容易忽视中断抢占及其干扰，但程序在实际执行时并不能确保这种原子性[4,19].

图2所示为一个典型的原子性违反案例，该案例来自某航天控制软件. 主任务循环执行*mode_manage*函数并周期性地更新变量*v_error_max*的值，使其表示*fabsf*(*set_loop-tyro_L*)的最大值. 若中断在S_1之后、S_3之前发生抢占，则S_2在这2次访问之间执行，导致*tyro_L*的值将在S_2中被改写. 若此时改写之后的*tyro_L*不再使$fabsf(set_loop-tyro_L)$的值满足S_1的条件语句，就会造成*v_error_max*并不是真实的最大值. 变量*v_error_max*用于重要的航天器控制任务，此缺陷将会引发严重的后果.

1.2 缺陷特征研究方法

中国空间技术研究院负责研制我国卫星、飞船和深空探测器等各类航天器，包括中国空间站、天问

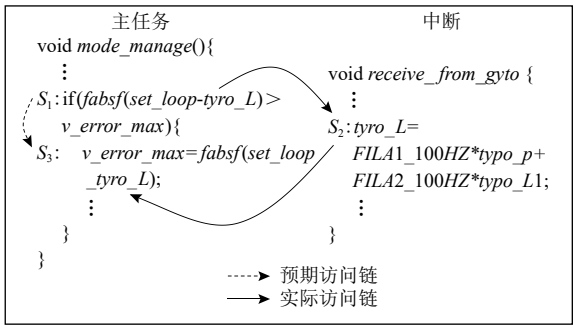

Fig. 2 An example of typical atomicity violation

图 2 一个典型的原子性违反案例

一号火星探测器、嫦娥五号月球探测器和北斗导航卫星等. 本文以 2009—2022 年中国空间技术研究院第三方评测机构软件问题库作为缺陷的数据来源. 该问题库包含超过 28 000 个软件缺陷, 这些缺陷在经过各种开发方测试后依旧遗留, 代表了航天嵌入式软件研制过程中最具挑战性的可信问题.

在案例分析过程中, 本文首先通过“中断”“原子性”“冲突”“竞争”等关键字进行检索和挖掘, 得到了 424 个相关缺陷案例. 然后, 为了筛选出具有完整的、可供进一步分析的原子性违反缺陷案例, 逐一地对 424 个案例进行核查. 核查的标准为: 1)案例包含详细的问题报告单、原始代码和修复后版本的代码; 2)对源代码进行分析和理解后, 确认该缺陷案例属于原子性违反. 最终, 获得了 82 个信息完整的缺陷案例. 这个选择过程保证了最终研究的缺陷案例集具有代表性.

这些缺陷来源于各种不同型号的航天嵌入式软件, 覆盖了近十年我国研制和发射的各类空间飞行器中的关键嵌入式软件, 包括航天器控制软件、航天器综合电子软件、航天器中央数据单元软件以及各类遥测遥控软件等; 此外, 这些软件的硬件运行环境涉及各种不同类型的处理器, 如 80C32, TSC695F, BM3803 等, 这保证了实证研究的公平性和多样性.

进一步, 本文对所有收集的案例逐个进行研究, 总结了造成原子性违反的共享数据访问交错模式, 并对发生缺陷的共享数据的访问方式和访问粒度进行了分析和统计.

1.3 缺陷表现特征

本节对实证研究获得的共享数据访问交错模式、访问方式和访问粒度等原子性违反缺陷表现特征进行了说明.

1)共享数据访问交错模式. 本文借用可序列化[20]的思想, 总结形成了 4 种造成原子性违反的访问交错模式, 如表 1 所示, 这些模式刻画了中断驱动型程序中的原子性违反. 每个访问交错 (e_p, e_r, e_c) 都可以通过一次执行中对同一个共享数据的 3 次访问描述. 其中, e_p和e_c为来自主任务或低优先级中断isr_l的本地串行访问对, 而e_r为来自另一个高优先级中断isr_h的远程交错访问.

Table 1 Access Interleaving Pattern of Atomicity Violation

表 1 原子性违反的访问交错模式

序号	模式	描述
1	$\begin{cases} e_p: R(x) \\ e_r: W(x) \\ e_c: R(x) \end{cases}$	2个连续的读访问被中断的写访问打断. 2个对同一内存的连续读访问预期读到相同的值, 而交错的写访问打破了该预期.
2	$\begin{cases} e_p: R(x) \\ e_r: W(x) \\ e_c: W(x) \end{cases}$	连接在读访问之后的本地写访问被中断的写访问打断. 写操作依赖于本地读操作读到的值（例如, 本地写访问依赖于本地读访问相关的分支条件）, 而交错的写访问修改了该值.
3	$\begin{cases} e_p: W(x) \\ e_r: R(x) \\ e_c: W(x) \end{cases}$	2个连续的写访问被中断的读访问打断. 本应不对外可见的中间结果被交错的读访问获取并使用.
4	$\begin{cases} e_p: W(x) \\ e_r: W(x) \\ e_c: R(x) \end{cases}$	连接在写访问之后的本地读访问被中断的写访问打断. 读访问预期读到本地的赋值, 而交错的写访问修改了该值.

2)原子区范围. 发生缺陷的本地串行访问对在源代码上的距离范围分布如图 3 所示, 最小范围为 1 行(例如 2 次访问在同一行代码中, 如 a++), 最大范围为 31 行, 平均范围为 5 行. 其中, 24 个访问对位于同一行, 17 个访问对相邻(范围为 2 行). 此外, 有 8 个访问对位于循环结构中, 6 个访问对作为函数参数被访问. 特别的是, 所有的访问对都在同一个函数中, 或作为同一主调函数内不同被调函数的参数.

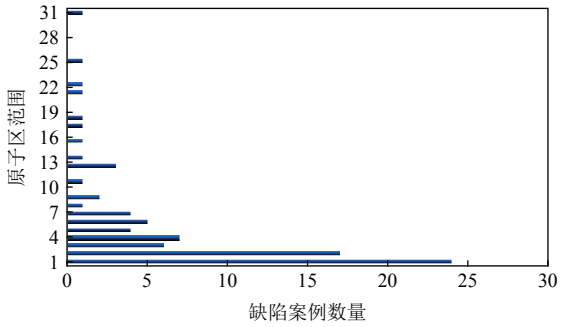

Fig. 3 Distribution of atomic region range

图 3 原子区范围分布

3)中断嵌套层数. 在中断驱动型程序中, 并发交错的数量会随着中断的数量呈指数级增长, 此外中断嵌套的存在将加剧此增长速度. 然而, 本文研究的 82 个缺陷案例中, 触发这些缺陷所涉及的中断数量都不超过 2 个. 其中, 69 个缺陷发生在主任务和中断之间; 9 个发生在中断与中断之间; 4 个发生在主程

序和嵌套的中断之间.

4)共享数据访问方式. 如表 2 所示, 在发生缺陷的共享数据中, 有 60% 是通过全局变量名进行直接访问的. 在全局变量访问方式中, 程序直接将变量所在内存单元的值取出并进行运算; 有 32% 通过指针解引用进行间接的访问. 8% 的缺陷发生于访问内存映射 I/O(memory mapped I/O, MMIO)时. 在中断驱动型嵌入式软件中, MMIO 是一种典型的数据类型, 广泛用于表示硬件数据资源, 例如 0x40008000 代表某个外设寄存器的内存映射地址, 程序可以通过 *((unsigned char*)0x40008000)对该寄存器进行读写.

Table 2 Access Modes and Access Granularities on Shared Data

表 2 共享数据访问方式与访问粒度

访问方式	整体	元素	成员	位	合计
全局变量名	44	9(4*)	7	12	72(4*)
指针	6(2*)	0	0	1	7(2*)
MMIO	1	1(1*)	0	1	3(1*)
合计	51(2*)	10(5*)	7	14	82(7*)

注: (*) 表示共享数据访问的偏移量不是常量, 例如$BRCST_buf[i]$.

5)共享数据访问粒度. 即使是对同一块内存单元, 程序也可能通过不同的访问粒度进行访问[18]. 如表 2 所示, 在研究的案例中, 嵌入式软件对共享内存的访问存在各种不同粒度. 46% 的共享内存为整体访问, 即通过变量名对变量指代的整个内存区域进行访问; 26% 的共享内存为通过数组下标对元素进行访问; 17% 的共享内存为通过结构体对成员进行访问; 11% 的共享内存需要精确到位访问. 此外, 在所有的内存访问中, 9% 的共享内存访问带有非常量的偏移量, 例如数组元素$BRCST_buf[i]$, 其偏移量需要通过计算变量i得到, 使得这些内存地址难以静态确定.

缺陷表现特征研究表明, 航天嵌入式软件中, 程序员具有原子性意图的串行访问在源代码上具有较近的距离, 可以通过限定原子区范围对原子区进行推断, 从而降低计算开销并减少误报. 对原子性违反的检测可以通过成对(主任务和中断, 或中断与中断)的方式进行, 从而以损失较低精度的代价实现对绝大部分缺陷的高效检测. 对共享数据的访问通常涉及访问方式和粒度的复杂组合, 这增加了共享数据分析的难度. 若共享数据分析只关注数组对象而不能区分数组元素, 将会导致大量误报; 若不能识别 MMIO 类型的共享数据, 则会导致漏报. 因此, 为了提高缺陷检测的精度, 检测工具需要精确的共享数据分析.

2 intAtom-S

本文基于原子性违反特征研究的结果, 提出 intAtom-S. 图 4 所示为 intAtom-S 的框架. 该框架主要包括 3 个阶段: 基于抽象解释的共享分析(见 2.1 节)、访问交错模式匹配(见 2.2 节)、不可行路径裁剪(见 2.3 节).

首先, intAtom-S 精确地分析程序中的共享数据. intAtom-S 构建了一个数值不变式参数化的细粒度内存访问模型; 采用区间抽象域和同余抽象域的约化积(reduced product of interval and congruence, RPIC)[21]进行基于抽象解释的数值分析, 以获得采用内存访问模型表示的共享数据地址中的数值不变式, 从而达到精确识别的目的. 在文献 [18] 的基础上, intAtom-S

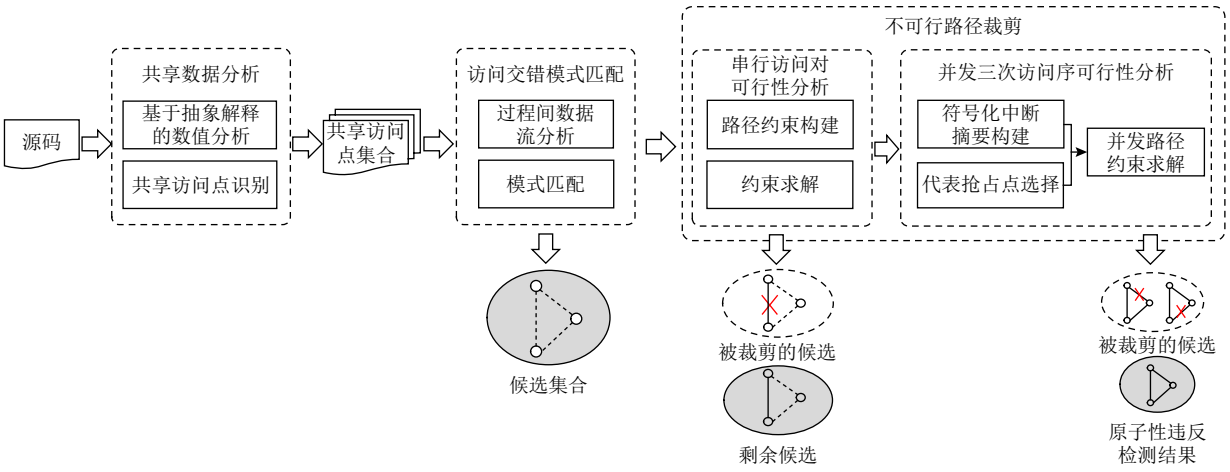

Fig. 4 intAtom-S framework

图 4 intAtom-S 框架

引入基于规则的方法剔除标志变量，仅识别那些会导致缺陷的共享数据，这对进一步精化缺陷检测结果至关重要.

其次，intAtom-S 使用轻量级数据流分析技术匹配符合 4 种交错模式的所有并发 3 次访问序作为潜在的原子性违反候选. intAtom-S 使用过程间分析并采用“工作单元”的概念[22]识别isr_l中的串行访问对. 在此基础上，结合来自isr_h的访问进行模式匹配以识别出属于访问交错模式（表 1）的并发 3 次访问序. 这些 3 次访问序是输入到后续步骤的原子性违反候选.

最后，对缺陷候选采用基于路径条件的约束求解. 通过对串行访问对和并发 3 次访问序进行路径可行性分析，逐步消除没有违反程序员预期和实际不可能发生的 2 类误报，得到最终的原子性违反结果：1）依次检查isr_l中 2 个串行访问之间的路径可行性，如果不可行，表明程序员对该串行访问没有一致性预期，并发 3 次访问序不会造成原子性违反，因此将该类候选排除. 2）intAtom-S 通过一个模块化、路径敏感的分析，消除在实际执行中由于约束条件无法满足而不可能真实发生的并发 3 次访问序. 为了缩减交错空间和提高分析效率，本文引入了 2 个策略：1）intAtom-S 为每个中断构建一个符号化的摘要，帮助进行模块化的路径裁剪，以消除不可行的交错. 2）intAtom-S 采用代表性抢占点策略，在不损失精度的情况下缩减交错空间.

相比文献 [17], intAtom-S 基于更精确、更细粒度且剔除标志变量访问后的共享数据识别结果，可以处理数组元素访问、标志变量访问导致的误报，将显著提高精度.

2.1 基于抽象解释的共享分析

根据缺陷特征，intAtom-S 提出一种精确的共享数据静态分析方法. 该方法主要包括：

1）参数化的内存访问模型.为了应对航天嵌入式软件中共享数据访问通常涉及的访问方式和粒度的复杂组合，本文通过参数化的内存访问模型对不同粒度的共享数据访问进行统一刻画.

2）基于抽象解释的数值分析.使用 RPIC 对每一个并发流进行基于抽象解释的数值分析，计算内存访问模型中偏移量的数值不变式.

3）共享访问点识别.采用基于规则的方法剔除标志变量访问，进而通过对不同并发流（主任务、各个中断）之间的数据访问点的数值抽象域进行“与操作”，识别出会导致缺陷的共享数据访问点.

2.1.1 参数化的内存访问模型

本节建立了一个可对标量类型、聚合类型以及 MMIO 类型的共享数据进行统一表示的参数化的内存访问模型来描述被访问的各种抽象内存区域. 这一模型表示为一个五元组（*Base*, *Offset*, *Size*, *Mode*, *ConstAccess*）.

1）*Base* 是抽象内存区域的起始地址（一般用变量名标识，若数据为 MMIO 则以 ADDR_ZERO 标识）.

2）*Offset* 是抽象内存区域的偏移量，为处理在同一程序点访问多个不同元素或成员的情况（例如循环访问数组变量），可将 *Offset* 定义为一个集合.

3）*Size* 是每一块抽象内存区域的大小.

4）*Mode* 是对抽象内存区域的访问方式，包括读和写.

5）*ConstAccess* 可判断该抽象内存区域是否写入常量或者 *Mode* 为读，若是，则置为 true；否则，置为 false.

表 3 为图 5 案例中标量类型、聚合类型、MMIO 这 3 种类型数据在该模型中的表示（针对每个数据类型，以一个具有代表性的访问为例，给出其表现形式）. 例如，在访问点⑤和⑥处，对 *upload_data*[0]、*upload_data*[1]的写访问可表示为 (*upload_data*, {0, 4}, 4, write, false), 描述了对地址为 *upload_data*+0、大小为 4B 以及地址为 *upload_data*+4、大小为 4B 的 2 个内存单元的写访问.

本文通过指针分析确定内存访问模型中的 *Base* 值，根据发生访问的程序语句可得到 *Size* 值和 *Mode* 类型. 因此，如何精确地描述抽象内存区域将取决于 *Offset* 值集的精确性.

2.1.2 基于抽象解释的数值分析

本节采用 RPIC 抽象域对每一个并发流进行独

Table 3 Representation of Different Data Types

表 3 不同数据类型表现形式

数据类型	数据名称	*Base*	*Offset*	*Size*	*Mode*	*ConstAccess*
标量类型	*func_run*	*func_run*	{0}	1	read	true
聚合类型	*upload_data*[*i*]	*upload_data*	{0,4}	4	write	false
MMIO	*(unsigned int*) *addr)*	ADDR_ZERO	{0xff00}	4	read	true

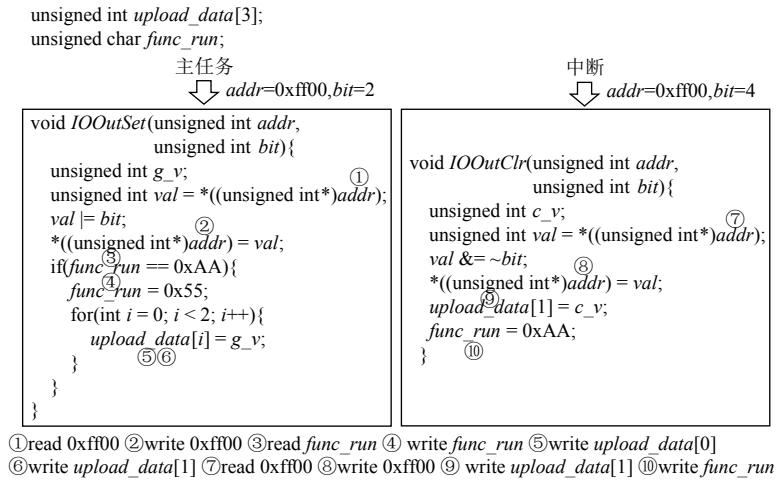

Fig. 5 Example for shared data access

图 5 共享数据访问示例

立的基于抽象解释的数值分析，计算内存访问模型中 *Offset* 的数值不变式. 除此之外，本节还扩展了现有的抽象解释迭代算法，以实现对中断干扰的上近似，保证分析无漏报.

2.1.2.1 RPIC 抽象域.

本文使用数值抽象域来刻画内存访问模型中的 *Offset* 值集. 在缺陷特征研究中发现偏移量的使用会出现在 3 种情况中：1)访问数组元素；2)访问结构体或联合体的成员变量；3)通过增加偏移量访问 MMIO. 因此在抽象解释过程中需要选择适合这 3 种情况的抽象域. RPIC 抽象域是区间抽象域和同余抽象域的约化积(reduced product)[21]，它可以描述连续区域和步长信息，步长信息用于分析通过间接寻址来实现访问操作，如结构体成员访问和指针解引用等. 在 RPIC 中，区间抽象域提供变量值的上界和下界，而同余抽象域计算值之间的步长. 因此抽象元素可以表示为 $[a, b]\cap(cZ+d)$. 例如，图 5 示例中的访问点⑤，⑥处，对应的偏移量被近似表示为 $[0, 4]\cap(4Z)$，它表示一个下界为 0、上界为 4、步长为 4 的离散整数集，即 *Offset* 值集为 $\{4Z|Z\in[0, 1]\}$.

2.1.2.2 中断干扰的上近似.

本文扩展了现有的抽象解释迭代算法，以实现对中断干扰的上近似分析. 对于中断驱动型程序，若在程序分析时不考虑中断并发对程序状态的影响，将导致主任务中的部分执行路径被误认为不可行，从而导致漏报. 如图 6 所示，若不考虑中断执行，仅考虑主任务执行，由于 $(Flag = \text{false})\wedge(Flag == \text{true})$ 为互斥的约束，S_1 将不会被执行，则 *KZData* 不会被访问. 而事实上在中断的作用下，*Flag* 会被赋值为 true，即 S_1 是可以被执行的，因此主任务中对 *KZData* 的读

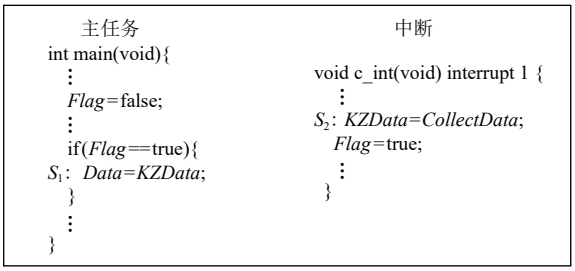

Fig. 6 A case of interrupt side effect

图 6 中断副作用示例

访问也是一次共享访问.为了保证分析结果的可靠性，本文采用了一种简单的处理策略，即在迭代过程中认为所有分支路径都可行，以收集到所有的数据访问点，实现对中断干扰的上近似分析.

2.1.3 共享访问点识别

共享访问点识别的目的是识别被不同并发流(主任务、各个中断)访问的共享数据. 特别是 intAtom-S 引入基于规则的方法将不会导致缺陷的标志变量的访问剔除，仅返回那些会导致缺陷的共享访问点集合. 在中断驱动型嵌入式软件中，自定义标志变量总是遵循的规则为：标志变量为 char 类型，且在整个程序执行过程中仅被赋值为常量. 而这些标志变量的并发访问并不会导致缺陷. 因此，本节首先识别出仅被写入常量的 char 类型共享数据. intAtom-S 通过对不同并发流之间的数据访问点的 *ConstAccess* 进行合取操作，识别出标志变量. 例如，图 5 示例中的访问点③，④和⑩，其共享数据分别被表示为 $(func\_run, [0, 0]\cap\{1Z\}, 0, \text{read}, \text{true})$，$(func\_run, [0, 0]\cap\{1Z\}, 0, \text{write}, \text{true})$ $(func\_run, [0, 0]\cap\{1Z\}, 0, \text{write}, \text{true})$. ③，④和⑩具有相同的 *Base*，对它们的 *constAccess* 字段进行合取操作，即 $\text{true}\wedge\text{true}\wedge\text{true}$ 得到 true，且

func_run 是一个 char 类型数据，因此其是一个标志变量. 然后，intAtom-S 通过对不同并发流之间的数据访问点的数值抽象域进行与操作，识别出共享数据访问. 例如，图 5 示例中的访问点⑨，其共享数据可被表示为 (*upload_data*, [4, 4]∩{1Z}, 4, write, true). 访问点⑤和⑥处的共享数据为 (*upload_data*, [0, 4]∩{4Z}, 4, write, false). ⑤和⑥与⑨具有相同的*Base*，对[4, 4]∩{1Z}和[0, 4]∩{4Z}进行与操作得到[4, 4]∩{1Z}，因此*upload_data*[1]是主任务(⑥)与中断(⑨)之间的共享数据. 在此过程中，会分析并发流的优先级以确定它们能否并发执行. 比如，相同优先级的中断之间不能发生抢占，相应的数据访问并不会构成共享. 最后，将两个或多个并发流之间的数据访问交集标记为共享数据.

intAtom-S 最终返回一个包含所有去掉标志变量访问之后的共享访问点和访问点位置信息的集合，访问点位置信息包括发生访问的函数、程序入口、语句行号、列号、源文件等字段.

2.2 访问交错模式匹配

表 1 中访问交错模式的 3 次访问序 (e_p, e_r, e_c) 由对同一共享数据的 3 个访问事件组成，其中e_p和e_c来自isr_l(或主任务)，e_r来自isr_h. 每一个访问事件 e 由一个四元组$e=(T, L, V, A)$表示，其中 T 为任务或中断的标识，L 是事件发生的程序点，V 是被访问的共享数据，A 是访问类型(读或写).

算法 1 描述了基于过程间数据流分析框架[23]的访问交错模式匹配方法. 本文将对同一个共享变量 v 的 2 次串行访问建模为一个数据流问题.如果对于一个变量的 2 次访问e_p和e_c之间有一条路径，且在该路径上 v 没有其它访问，则构成一个串行访问对. *dataFlowAnalysis* 对该数据流问题进行求解，以查找出所有的串行访问对. 此外，根据 Vaziri 等人[22]的工作，程序员的原子意图与程序结构相关，可以在特定的“工作单元”边界内对原子性进行推断. 受此启发，结合实证研究结果，本文根据共享变量所在位置，将原子性的边界限制在同一函数或主调函数内，并过滤掉那些超出边界的访问对.然后，对于isr_h中的每个共享内存访问，intAtom-S 检查它和isr_l(或主任务)中的串行访问对是否可以组合为表 1 中的访问交错模式(行⑪~⑯). 如果可以，intAtom-S 将该 3 次访问序 (e_p, e_r, e_c)添加至原子性违反候选集.

算法 1.访问交错模式匹配算法.

输入：源程序 P;

输出：原子性违反候选集合 $C=\{(e_p, e_r, e_c)\}$.

① $C \leftarrow \varnothing$;
② $B \leftarrow \varnothing$; /* B 为串行访问对集合 (e_p, e_c)*/
③ for each T in P do /*T为中断或主任务*/
④ $N = SharingAnalysis()$; /*N为共享访问点集合*/
⑤ $B\cup = dataFlowAnalysis(T, N)$;
⑥ end for
⑦ $patternSet \leftarrow \{(R, W, R), (W, W, R), (R, W, W), (W, R, W)\}$;
⑧ for each (e_p, e_c) in B do
⑨ for each T in $P$and the priority of T is higher than $e_p.T$ do
⑩ for each shared access e_r in Tdo
⑪ if $e_r.V == e_p.V$ and $(e_p.A, e_r.A, e_c.A) \in patternSet$ then
⑫ $C\cup =(e_p, e_r, e_c)$;
⑬ end if
⑭ end for
⑮ end for
⑯ end for

2.3 不可行路径裁剪

本阶段利用路径约束求解进行串行访问，对可行性分析和并发 3 次访问序进行可行性分析，逐步消除误报，从而得到最终的原子性违反结果.

2.3.1 串行访问对可行性分析

在这个阶段，本文通过识别路径不可行的串行访问对 (e_p, e_c) 消除一类误报. 如果某串行访问对不可行，说明程序员对它们并没有进行原子性假设，此类原子性违反候选将被过滤掉. 如图 7 所示，对共享数据*YCData*共有 3 次访问，S_1、S_2与S_3显然满足(*W*, *W*, *R*)模式. 在该案例中，中断Timer0_ISR通过S_2周期性地更新*YCData*，并将标志变量*fgAh*置为true. 只有当中断将*YCData*数据更新后，主任务才会通过S_3读取*YCData*的值，该程序有意设计语句if(*fgAh* ==

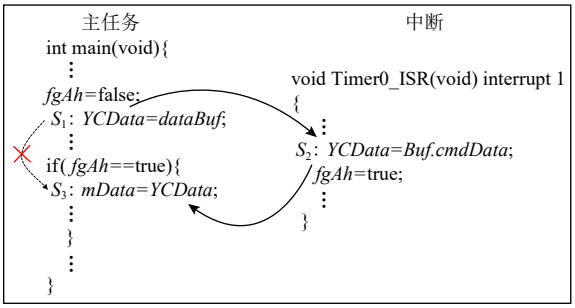

Fig. 7 A case of infeasible consecutive access pair

图 7 不可行串行访问对示例

true)以判断中断是否已经发生. 虽然S_1和S_3是2个连续的串行访问, 但它们具有互斥的约束($fgAh$ = false)∧($fgAh$ == true), 因此, 这2个访问组成的串行访问对是不可行的, 表明程序员对它们没有进行原子性假设. 所以, 有必要排除这些不可行的连续访问对.

为了提高分析效率, 本文建立依赖图[23]对程序进行稀疏的表示. 并在此基础上构建串行访问对的路径约束, 然后利用SMT求解器进行约束求解. 若路径中存在循环, 为了防止过高的开销, 对循环展开2次后跳出.

对于一个给定的串行访问对(e_p, e_c), $P1$和$P2$分别是由入口到e_p和e_c的路径. 路径条件$PC(e_p, e_c)$由$P1$与$P2$的有效路径条件[24-25]合取得到. 其中, 有效路径条件通过对该路径相关的数据依赖和控制依赖合取得出. 因此, 给定一条从入口出发的路径P, v_i为P的路径约束所依赖的变量, P的路径条件计算公式为

$$PC(P)=\bigwedge_{i=1,\cdots,n} CD(v_i)\wedge\bigwedge_{i=2,\cdots,n}(v_{i-1}=v_i)\wedge\bigwedge_{i=2,\cdots,n} DD(v_{i-1},v_i).$$

一个路径条件由3个部分组成:

1) $CD(v_i)$为控制依赖, 表示到达v_i的条件约束;

2) $v_{i-1}=v_i$描述v_i中存储的值流向v_{i-1};

3) $DD(v_{i-1}, v_i)$表示值从v_i流向v_{i-1}是可行的.

2.3.2 并发3次访问序可行性分析

在串行访问对(e_p, e_c)可行的前提下, 分析3次访问序(e_p, e_r, e_c)并发路径的可行性, 进一步消除误报. 由于中断程序的副作用, 仍然存在一些原子性违反候选在实际的并发执行中永远不会发生, 即不存在一条可行的并发路径包含该3次访问序.如图8所示, 当中断在主任务中的S_1后被抢占, 则S_2是不可能发生的. 这是由于如果在该抢占点的状态需满足($onoffnum>0$)∧($fgExecute$ == 1), 则从ONOFFManage入口到S_2的路径条件($fgExecute$ == 0)将不能被满足. 因此, 给定一个原子性违反候选(e_p, e_r, e_c), 若要检查其

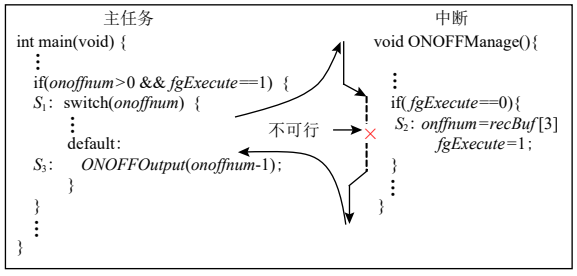

Fig. 8 A case of infeasible concurrent triple access interleaving

图8 不可行并发3次访问序示例

可行性, 需要对e_p与e_c之间的每个抢占点p分别检查其子路径($p\rightarrow entry_{isr}\rightarrow\cdots\rightarrow e_r$)与($exit_{isr}\rightarrow p\rightarrow\cdots\rightarrow e_c$)的可行性. 由于并发路径的数量为指数级, 这一步骤非常耗时.

如算法2所示, intAtom-S采用了一种组合方法以降低不可行并发3次访问序裁剪的开销.

1)模块化地构建中断摘要. intAtom-S首先通过*constructSummary*对每个中断构建符号化摘要, 从而避免冗余路径条件计算(行①~③). 摘要使用符号约束描述了e_r的路径条件与$exit_{isr}$的状态.

2)选择具有代表性的抢占点. 对于每一个原子性违反候选, intAtom-S通过函数*selectPreemptPoint*选择具有代表性的抢占点, 以缩减交错空间(行④~⑤).

3)利用约束求解验证可行性. 对于每一个选择出的代表性抢占点p, 分别检查其子路径($p\rightarrow entry_{isr}\rightarrow\cdots\rightarrow e_r$)与($exit_{isr}\rightarrow p\rightarrow\cdots\rightarrow e_c$)的可行性.

intAtom-S通过函数*obtainState*(行⑧)获得代表性抢占点处的全局程序状态, 并将此信息作为上下文传递给中断. 然后, 函数*isFeasible*提取摘要中的约束条件, 并利用约束求解器来检查以抢占点p为上下文时, 路径$entry_{isr}$到e_r的可行性(行⑨). 如果这条路径不可行, 则意味该3次访问序为误报, 需要被裁剪掉; 否则, intAtom-S将继续分析以$exit_{isr}$为上下文时, 抢占点p和e_c之间路径的可行性(行⑪). *constructPC*使用与2.3.1节相同的方法构建路径条件. 如果($p\rightarrow entry_{isr}\rightarrow\cdots\rightarrow e_r$)与($exit_{isr}\rightarrow p\rightarrow\cdots\rightarrow e_c$)这2条路径都可行, intAtom-S将向用户报告一个原子性违反. 由于intAtom-S区分了每个抢占点的上下文并检查每条路径的可行性, 因此是上下文敏感和路径敏感的.

intAtom-S还在全局程序状态中对中断屏蔽的使用进行了建模. 本文将中断屏蔽寄存器作为全局变量, 并在每个中断的入口检查是否被屏蔽. 因此, 本文的方法可以处理通用中断屏蔽.

算法2.并发3次访问序可行性分析算法.

输入: 原子性违反候选集合C={(e_p, e_r, e_c)}、源程序P;

输出: 原子性违反.

① for each isr in P do

② $Summ_{isr}\leftarrow constructSummary(C)$;

③ end for

④ for each (e_p, e_r, e_c) in C do

⑤ $PSet\leftarrow selectPreemptPoints(e_p, e_c)$;

⑥ $i=e_r.T$;

⑦ for each p from $PSet$ do
⑧ $state = obtainState(p)$;
⑨ if $isFeasible(state, Summ_{isr})$ then
⑩ $pc = constructPC(p, e_c)$;
⑪ if $isFeasible(Summ_{isr}, pc)$ then
⑫ $reportWarning(e_p, e_r, e_c)$;
⑬ end if
⑭ end if
⑮ end for
⑯ end for

接下来，将详细介绍中断摘要构建与代表性抢占点选取的细节.

1）中断摘要构建. 当检查并发 3 次访问序可行性时，需要获取 2 个中断信息：

①了解 $entry_{isr}$到e_r的路径条件，用于检查路径e_p到e_r的可行性；

②了解经过e_r的路径在中断出口处全局变量的状态，以确定是否存在从中断退出回到e_c的可行路径.

因此，本文构建包含这 2 个中断信息的中断摘要，当检查路径可行性时，在不同的抢占点处复用这些摘要.

给定一个中断 isr 和一组变量集合 $\Pi=\{e_0, e_1, \cdots\}$，该集合代表 isr 中所有可能的e_r，isr 的摘要可由集合$Summ_{isr}$表示，可以表示为三元组集合$Summ_{isr} = \{(e_0, pc_{e_0}, exitState_{e_0}), (e_1, pc_{e_1}, exitState_{e_1}), \cdots\}$，其中$pc_{e_i}$代表从$entry_{isr}$到$e_i$的路径条件，$exitState_{e_i}$代表 isr 出口处的全局状态.

本文单独分析每个中断来构建它的摘要，可在并发 3 次访问序可行性检查时直接复用. 对于 isr 中的访问事件e_r，存在 2 种情况可以重用中断摘要. 一是检查涉及e_r的多个原子性违反候选的可行性；二是对同一个候选的多个抢占点进行检查.

2）代表性抢占点选取. 并发 3 次访问序(e_p, e_r, e_c)的可行性取决于 2 条子路径（$p \to entry_{isr} \to \cdots \to e_r$）与（$exit_{isr} \to p \to \cdots \to e_c$）的可行性. 然而，只有当程序语句直接修改了 2 条并发子路径的约束条件，或者修改了约束条件所依赖的变量时才会影响到并发 3 次访问序的可行性. 也就是说，某些程序语句的副作用可能对这 2 条子路径可行性检查的结果没有影响. 因此，可以选择性地检查部分抢占点对应的并发路径，而无需检查所有的路径，这些抢占点就被称为代表性抢占点.

从e_p到e_c路径上的所有程序语句可用集合$S = \{S_1, S_2, \cdots, S_n\}$表示，其中$S_i$是$S_{i+1}$的直接前序语句. 用$P = \{p_1, p_2, \cdots, p_n\}$表示抢占点集合，其中 p_i位于S_i与S_{i+1}之间，称S_i为抢占点 p_i对应的程序语句. 本文代表性抢占点定义为：给定抢占点 p_i与 p_{i+1}，如果 p_i满足 2 个条件之一，那么其为一个代表性抢占点. 1）S_{i+1}为一个赋值语句，且S_{i+1}没有改变任何路径约束 pc_{e_r}中包含的变量的取值；2）S_{i+1}包含一个条件表达式，该条件表达式的控制变量与$exitState_{e_r}$中包含的变量不重合.

因此，通过按顺序遍历 P 以选取所有e_p到e_c之间的代表性抢占点. 对于每一个抢占点 p_i，如果其未满足上述条件，那么 p_{i+1}将被选择为代表性抢占点；否则，p_{i+1}可以被当前选定的代表性抢占点代表.

图 9 显示一个选择代表性抢占点的例子. (e_p, e_r, e_c)为一个原子性违反候选. 主任务与中断中的语句记为S_i，抢占点记为 p_i. S_2对变量 $fgCurrentB$ 进行了写操作. 由于路径 $(p \to entry_{isr} \to \cdots \to e_r)$ 的路径条件 pc_{e_r}与变量 $fgCurrentB$ 无关，且S_2不是分支语句，因此 p_2可以被 p_1代表；同样 p_3可以被 p_2代表. 而S_4对全局变量 $fgAh$ 进行了写操作，$fgAh$ 在路径条件 pc_{e_r}中，因此 p_4不能被 p_3代表. 本文选择 p_4为新的代表性抢占点. S_5对变量 $fgCharge$ 进行写操作，然后 $fgCharge$ 用于S_6中条件表达式中. 然而 $fgCharge$ 不包含于$exitState_{e_r}$中，因此 p_5与 p_6可以被 p_4代表. 对于S_7，$fgPower$ 是条件表达式的控制变量，且包含在$exitState_{e_r}$中，因此 p_7不能被 p_4代表. 最后，选择 3 个代表性抢占点 p_1，p_4与 p_7.

3 实验评估与应用

本文在一台配备 Intel(R) Xeon(R) E5-2 620 CPU、32GB RAM 和 Ubuntu 20.04 操作系统的计算机上进行了实验. 为了评估 intAtom-S，考虑 4 个研究问题：

问题 1：intAtom-S 在检测中断驱动型程序的原子性违反方面有效性如何？

问题 2：intAtom-S 在检测中断驱动型程序的原子性违反方面的效率如何？

问题 3：intAtom-S 分析共享数据的有效性如何？

问题 4：intAtom-S 路径裁剪的有效性和效率如何？

3.1 实验设置

为了评估 intAtom-S 的缺陷检测能力，本文将其与 Rchecker[15] 进行实验对比，Rchecker 是与 intAtom-S 最相近且最新提出的方法，它们针对的都是中断驱动型程序中的原子性违反. 然而 Rchecker 不是开源工具，且只在基准测试集 Racebench2.1 上进行过实验，因此本文同样在 Racebench2.1 上对 intAtom-S 进行实

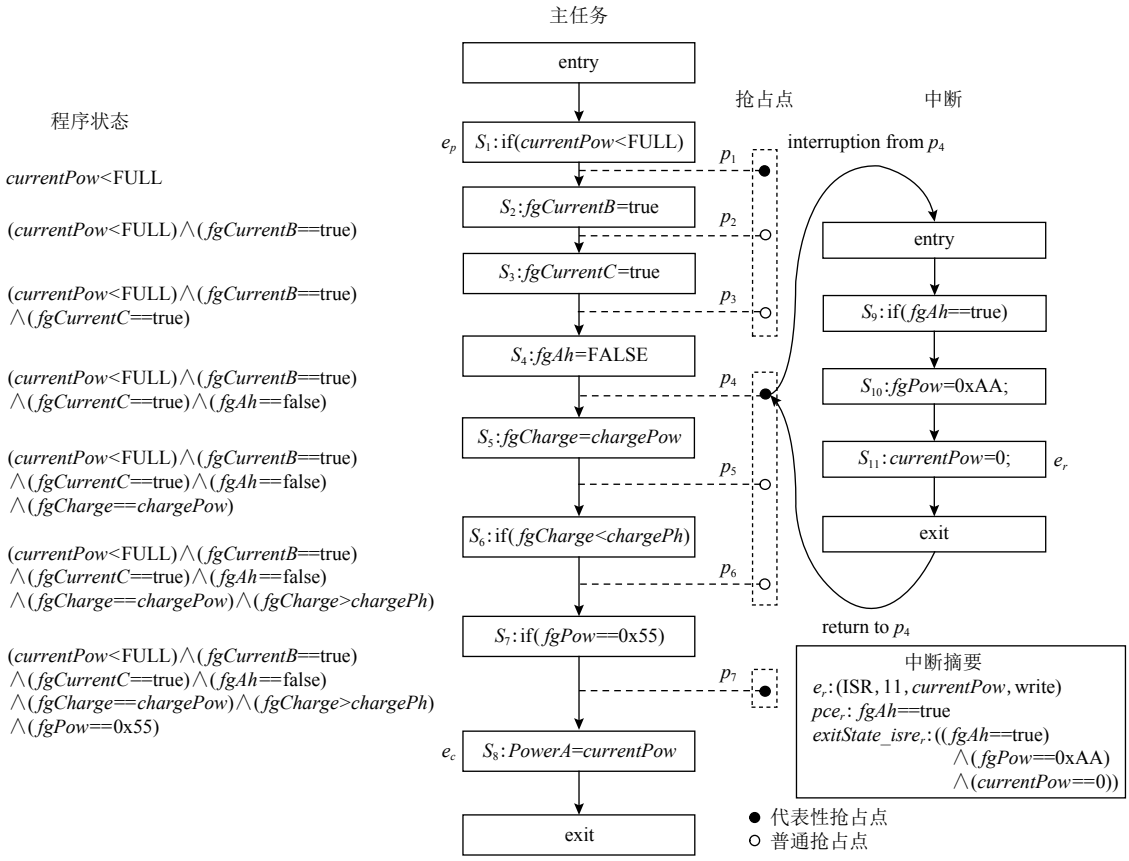

Fig. 9 An example of representative preemption points selection

图 9 代表性抢占点选择示例

验. 为了进一步评估本文方法的实用性，还在 8 个真实的中断驱动型航天嵌入式软件上进行了实验. 目前为止，intAtom-S 是唯一一个在工业嵌入式软件上进行评估的并发缺陷检测工具. 此外，本文还将 intAtom-S 与 static-TSA[26] 进行实验对比，以评估其分析共享数据的有效性. 实验中使用了 2 个数据集.

1)数据 1 是基准测试集. 本文使用开源基准测试集 Racebench 2.1 来评估 intAtom-S 是否可以在各种场景中检测原子性违反. Racebench2.1 最初被用于作为评估 NASAC 2019 并发缺陷检测工具竞赛的参赛工具. 后来，它也被用于 Rchecker[15] 的实验分析. Racebench 2.1 由 31 个中断驱动型案例和 54 个手动注入的原子性违反构成. 这些案例都是基于真实航天嵌入式软件的特点设计的，能够反映现实世界中断并发程序的语法和语义特点.这些案例涵盖原子性违反的各种表现形式，如访问交错模式、共享数据类型、访问方式、控制流场景等. Racebench 2.1 考虑了发生缺陷的不同场景，有助于对工具进行全面的评估.

2)数据 2 是真实中断驱动型航天嵌入式软件. 为了评估 intAtom-S 是否能够有效检测真实中断驱动型软件中的原子性违反，从实证研究的缺陷案例中选取了 8 个具有代表性的软件，如表 4 所示，所选案例的规模从 970 行到 5 099 行不等，是中断驱动型航天嵌入式软件最典型的代码规模. 首先，标记了这些软件的 CPU 类型、程序规模和中断个数，这是反映真实嵌入式软件多样性的 3 个重要属性. 然后，以涵盖这 3 个属性的多种组合为目标，选取具有代表性的案例. 尽管我们希望选用尽可能多的案例进行实验，然而评估真实的航天嵌入式软件需要消耗大量的时间和资源. 主要原因为：①嵌入式软件的编译平台多种多样，不同的编译器(如 MCS-51, MSP430)采用不

Table 4 Analyzed Real-world Aerospace Software Projects

表 4 被分析的真实航天软件项目

序号	项目	描述	代码行数
1	module1	某电源供应器控制软件	2 275
2	module2	某推进控制软件	970
3	module3	某热控软件	1 701
4	module4	某遥测和指挥软件	2 380
5	module5	某姿态控制软件	5 099
6	module6	某电源控制软件	3 767
7	module7	某 CPU 软件	1 544
8	module8	某推进线路盒软件	4 015

同的语言扩展，如数据类型、关键字等. 由于原型工具只支持 ANSI C，因此需要对源代码进行大量修改才能将原始代码转换为等价的 ANSI C 代码. ②原子性违反的确认需要应用程序（系统）相关的特定知识. 所有原始报告中的警报都需要开发人员单独分析确认. 一般来说，每个项目需要一个多月的时间才能完成确认. 因此，本文最终选择了 8 个具有代表性的案例作为实验对象.

3.2 检测有效性（针对问题 1）

1）基准测试集检测结果.

在基准测试集 Racebench 2.1 上的实验结果如表 5 所示. intAtom-S 可以在基准测试集中检测出所有的原子性违反，并显著减少误报. intAtom-S 在 31 个基准测试集案例中只有 6 个误报，相比 Rchecker 误报率降低了 72%.

我们发现，Rchecker 误报的主要原因是因为忽视了对程序员的原子性意图的推断. 因此，没有原子性期望的 3 次访问序也被报告为缺陷. 而 intAtom-S 则有效地解决了这一问题.进一步分析了 intAtom-S 的 6 个误报，误报原因都是因为在循环中使用了简化路径条件计算. 由于有一些在循环中被访问的共享数据，是被循环计数变量相关的约束所保护的.

Table 5 Results of Benchmark

表 5 基准测试集检测结果

序号	代码行数	中断个数	共享数据个数	原子性违反个数	Rchecker				intAtom-S			
					#WN	#TP	#FP	时间/s	#WN	#TP	#FP	时间/s
1	65	2	3	1	2	1	1	0.3	2	1	1	0.203
2	46	2	2	1	2	1	1	0.47	2	1	1	0.093
3	74	2	4	1	2	1	1	0.51	1	1	0	0.484
4	69	2	9	1	3	1	2	0.31	1	1	0	0.375
5	47	1	2	1	3	1	2	5.2	1	1	0	0.125
6	54	1	3	1	2	1	1	0.31	1	1	0	0.266
7	51	1	2	1	11	1	10	0.37	1	1	0	0.328
8	53	1	2	1	2	1	1	0.32	2	1	1	0.375
9	48	1	3	3	2	2	0	0.34	3	3	0	0.156
10	54	1	2	1	1	1	0	0.32	2	1	1	0.125
11	44	1	4	1	1	1	0	0.28	1	1	0	0.078
12	35	1	2	1	1	1	0	0.31	1	1	0	0.031
13	67	3	4	1	2	1	1	0.45	2	1	1	0.203
14	60	3	4	1	2	1	1	0.36	2	1	1	0.156
15	41	1	2	1	1	1	0	0.31	1	1	0	0.094
16	34	1	1	3	3	3	0	0.36	3	3	0	0.063
17	42	1	2	5	5	5	0	1.81	5	5	0	0.109
18	65	2	2	3	3	3	0	0.36	3	3	0	0.109
19	66	1	8	1	2	1	1	0.37	1	1	0	0.531
20	55	2	3	1	3	1	2	0.32	1	1	0	0.172
21	81	1	6	3	3	3	0	0.35	3	3	0	0.578
22	67	1	4	4	6	4	2	0.34	4	4	0	0.219
23	40	1	1	2	2	2	0	0.3	2	2	0	0.063
24	65	1	3	1					1	1	0	0.141
25	39	1	2	1	1	1	0	0.31	1	1	0	0.078
26	44	2	1	2	1	1	0	0.29	2	2	0	0.062
27	49	3	1	5	2	2	0	0.32	5	5	0	0.078
28	54	3	2	1	1	1	0	0.41	1	1	0	0.109
29	95	1	3	1	1	1	0	1.4	1	1	0	0.359

续表 5

序号	代码行数	中断个数	共享数据个数	原子性违反个数	Rchecker				intAtom-S			
					#WN	#TP	#FP	时间/s	#WN	#TP	#FP	时间/s
30	57	3	2	1	2	1	1	0.35	1	1	0	0.109
31	92	1	7	3	3	3	0	0.31	3	3	0	0.21
合计	1 753	48	96	54	75	48	27	17.76	60	54	6	6.082
准确率						88.9%				100%		
误报率							36%				10%	
平均时间/s								0.592				0.196

注：#WN 表示报告的缺陷数量；#TP 表示正报数；#FP 表示误报数.

为了避免循环展开，intAtom-S 简单地忽略了该路径约束. 因此可能会错误地将条件访问视为对同一变量的多次访问. 可以通过对循环使用路径进行敏感分析以消除这些误报. 然而，这将不可避免地增加开销. 因为当循环计数非常庞大的时候，会产生巨额开销.

2）真实中断驱动型航天嵌入式软件检测结果.

表 6 给出了在真实航天嵌入式软件上的检测结果.每个阶段的结果分别由 3 部分组成，“违反个数”表示进行当前阶段的原子性违反候选个数；“减少比例”指与上一阶段相比，原子性违反减少的比例；“时间占比”表示当前阶段的时间开销占总时间的百分比. 每一个报告的原子性违反都经过开发人员检查与确认，最终确定是否为一个真实缺陷. 从表 6 可以看出，intAtom-S 在 8 个真实软件中检测出 45 个原子性违反，平均误报率为 8.9%. 实验结果显示，intAtom-S 可以有效地检测真实工业嵌入式软件中的原子性违反，且具有非常低的误报率.

Table 6 Detection Results of Real-World Interrupt-Driven Aerospace Embedded Software

表 6 真实中断驱动型航天嵌入式软件检测结果

项目	代码行数	中断个数	共享数据分析		模式匹配		路径裁剪第 1 阶段			路径裁剪第 2 阶段			#FP	总时间/s
			时间占比/%	数据个数	时间占比/%	违反个数	时间占比/%	违反个数	减少比例/%	时间占比/%	违反个数	减少比例/%		
module1	2 275	3	8.7	72	23.5	289	22.3	123	57.4	45.4	10	96.5	1	114.7
module2	970	3	1.8	52	4.9	376	1.9	194	48.4	91.4	6	98.4	1	136.7
module3	1 710	2	21.3	81	37.7	201	27.7	125	37.8	13.3	6	97.0	0	30.0
module4	2 380	2	33.5	222	56.4	334	4.9	179	41.5	5.2	2	99.4	0	185.7
module5	5 099	5	5.9	16	87.8	15	12.5	6	60.0	0.2	2	86.7	0	58.9
module6	3 767	2	32.9	94	36.6	183	21.7	101	44.8	8.7	8	95.6	2	169.9
module7	1 544	2	23.0	26	54.4	88	12.7	62	29.5	9.9	1	98.4	0	28.1
module8	4 015	2	5.5	83	36.1	418	7.9	311	25.6	50.5	10	97.6	0	304.7
总计						1904		1 101	42.2		45	97.6	4	
平均值			15.9		37.8		11.1			35.2				

注：#FP 表示误报数.

进一步分析这些检测结果，将其分为 3 类：有害原子性违反、良性原子性违反和误报. 其中缺陷是否为良性也需要经过领域专家的检查与确认.intAtom-S 检测到 23 个有害的原子性违反都已经得到开发人员的确认. 检测到的 18 个良性原子性违反可分为 2 类. 1）虽然e_p和e_r是 2 个连续的本地访问，但是开发人员对它们没有原子性的期望. 例如，开发人员预计e_p之后会出现中断，并使用e_c进行容错. 2）原子性违反对结果没有影响. 例如，一个并发 3 次访问序(e_p, e_r, e_c)匹配模式(W, W, R)，如果e_p和e_r将相同的值存储到共享数据中，e_c总是读取期望值.值得注意的是，本文基于文献 [17] 中的 intAtom 进行扩展，intAtom 共检测到 3 类良性原子性违反，除去这 2 类，还有一类是由开发人员有意设计的.例如，在模式(R, W, W)中，e_p对标志变量的读访问 处于循环条件语句中，等待特定的中断抢占并执行e_r以改变此标志变量的值使

条件满足，从而执行循环体中的代码段；执行完毕后通过e_c重置此标志变量的值. 然而，本文在共享数据识别阶段去除了标志变量访问，所以检测阶段不会再报告此类良性原子性违反，这表示 intAtom-S 的检测结果更加精确.

intAtom-S 结果中存在的 4 个误报都是由于在循环中使用了简化的路径条件计算所导致. 相比 intAtom，intAtom-S 的检测结果更加精确，如表 7 所示. intAtom 在这 8 个真实程序的检测结果中存在 11 个误报，其中有 6 个是由于当循环计数变量用作数组下标时，intAtom 不能区分这些数组元素所导致的；还有 1 个是由于标志变量访问造成的. 而本文方法采用精确的共享数据分析，能够识别标志变量访问，且区分数组元素，因此可以避免这些误报.

Table 7 Comparison of Effectiveness and Efficiency Between intAtom-S and intAtom

表 7 intAtom-S 与 intAtom 有效性及效率对比

项目	intAtom		intAtom-S		
	时间/s	#FP	时间/s	额外开销	#FP
module1	104.7	2	114.7	9.6%	1
module2	134.3	1	136.7	1.8%	1
module3	23.6	3	30.0	27.1%	0
module4	123.5	1	185.7	50.4%	0
module5	55.4	0	58.9	6.3%	0
module6	113.9	3	169.9	49.2%	2
module7	21.6	0	28.1	30.0%	0
module8	287.9	1	304.7	5.8%	0
合计		11			4

注：#FP 表示误报数.

3.3 检测效率（针对问题 2）

表 5 显示了 intAtom-S 和 Rchecker 的在基准测试集 Racebench 2.1 上的运行时间. 为了公平地分析时间性能，本文只比较了 2 个工具都能完成检测的案例上所花费的时间. 例如，由于 Rchecker 无法分析案例 24. 因此，本文只关注其余 30 个案例的实验数据. 从总体上看，相比 Rchecker，intAtom-S 在大多数案例（26/30）上花费更短的检测时间，intAtom-S 的平均检测时间为 0.196 s，而 Rchecker 的平均检测时间为 0.592 s.

由表 5 可知，在所有的案例中，intAtom-S 的最长检测时间为 0.578 s. 而 Rchecker 在某些案例上的检测时间则都多于 1 s（如案例 5、17、29），甚至最长检测时间达到了 5.2 s. 检查这些案例的代码，发现它们都具有庞大而复杂的交错空间，这导致 Rchecker 的分析时间呈指数级增长. intAtom-S 使用函数摘要和代表性抢占点，可以有效缓解交错空间爆炸，从而达到更高的检测效率.

表 7 显示了 intAtom-S 与 intAtom 在真实中断驱动型程序上的检测时间对比. 真实的中断驱动型程序规模相对较大，包含数千行代码. intAtom-S 可以在 305 s 内分析 5 000 行的真实航天嵌入式软件，虽然比 intAtom 时间开销略高（额外开销介于 1.8%~50.4%），却实现了更高的检测精度，能够满足实际工业需求.

3.4 共享数据分析效果（针对问题 3）

由于基准测试集中都是代码规模较小的案例，对于不同数据类型的代表性不强，不具有统计学意义. 本文在 8 个真实的航天嵌入式软件上进行了 intAtom-S 与 static-TSA 的共享数据分析对比实验.在这 8 个软件中，存在 3 种共享数据类型（标量类型、聚合类型、I/O 地址）. 值得注意的是，static-TSA 利用调用图和指向分析来识别共享数据访问，然而与 intAtom-S 相比，static-TSA 既不能处理 I/O 地址访问，也不能区分数组中的各个数组元素，这导致其实验结果存在大量误报与漏报.

表 8 显示了 intAtom-S 与 static-TSA 共享数据分析的结果. 本文手动分析每个检测到的共享数据并进行最终确认. 由于 intAtom-S 以原子性违反检测为目标，进行共享数据分析时剔除了标志变量访问，而 static-TSA 并不进行此操作，为了保证对比实验的公平性，本文将 static-TSA 结果中的标志变量访问手动剔除,进而对两者进行对比. 对于这 8 个程序，intAtom-S 检测所有共享数据都没有误报；而 static-TSA 具有 191 个误报. 这是由于 static-TSA 不能区分数组索引，导致对不同数组元素的访问被标识为同一数据的访

Table 8 Results of Shared Data Analysis

表 8 共享数据分析结果

项目	static-TSA		intAtom-S	
	共享数据个数	#FP	共享数据个数	#FP
module1	43	0	72	0
module2	54	2	52	0
module3	117	36	81	0
module4	232	10	222	0
module5	16	0	16	0
module6	118	24	94	0
module7	26	0	26	0
module8	202	119	83	0
合计	808	191	646	0

注：#FP 表示误报数.

问. 此外, static-TSA 遗漏了 29 个共享的 I/O 地址访问; 而 intAtom-S 不仅可以区分数组索引, 且能有效检测共享的 I/O 地址访问, 因此能够精确检测航天嵌入式软件中各种类型的共享数据.

3.5 路径裁剪效果(针对问题 4)

为了更好了解 intAtom-S 路径裁剪的有效性和效率, 本文关注通过模式匹配检测到的原子性违反候选的数量、经过路径裁剪后剩余的候选数量, 以及它们的时间开销, 分析结果如表 6 和表 9 所示.

实验结果表明, 路径裁剪具有非常显著的贡献. 在模式匹配阶段, intAtom-S 检测每个项目中潜在的原子性违反候选, 平均每个项目有 238 个候选. 在路径裁剪阶段, intAtom-S 首先通过串行访问对可行性分析裁剪了 42.2% 的候选, 平均每个项目剩余 100 个候选. 然后 intAtom-S 通过并发 3 次访问序可行性分析裁剪了 97.6% 的候选. 每个阶段的时间开销如表 6 所示. 在模式匹配和路径裁剪阶段, intAtom-S 花费的时间分别占总时间的 37.8% 和 46.3%.

为了进一步评估所提出的符号化中断摘要和代表性抢占点选择方法的有效性, 本文实现了 2 个额外的 intAtom-S 变体: Naive 和 Naive+RPPS(使用代表性抢占点选择). Naive 可以被看作是一种基线方法, 它在没有中断摘要的情况下, 检查并发 3 次访问序的可行性. Naive+RPPS 同样不使用中断摘要, 但仅在每个代表性抢占点处进行路径约束求解. 本文在 8 个真实航天嵌入式软件上进行了实验, 比较了 intAtom-S, Naive 和 Naive+RPPS 进行并发 3 次访问序可行性分析时的时间开销. 如表 9 所示, Naive+RPPS 的效率是 Naive 的 1.2~3.6 倍, 平均 1.8 倍; intAtom-S 的效率是 Naive 的 2.6~8.5 倍, 平均 3.0 倍. 结果表明, 符号化摘要和代表性抢占点对提高方法可扩展性是至关重要的.

Table 9 Time Cost of Naive+RPPS and intAtom-S Compared with That of Naive

表 9 Naive+RPPS 和 intAtom-S 与 Naive 的时间开销对比

项目	Naive	Naive+RPPS		intAtom-S	
	时间/s	时间/s	效率比	时间/s	效率比
module1	193.8	54.3	3.6×	52.1	3.7×
module2	331.1	232.0	1.4×	125.0	2.6×
module3	10.9	5.6	1.9×	4.0	2.7×
module4	81.9	68.4	1.2×	9.6	8.5×
module5	0.3	0.1	3.0×	0.1	3.0×
module6	56.6	16.3	3.5×	14.8	3.8×
module7	9.0	3.9	2.3×	2.8	3.2×
module8	397.6	220.9	1.8×	153.8	2.6×
合计	1 081.2	601.5	1.8×	362.2	3.0×

注: 以 Naive 为基线, Naive+RPPS 或 intAtom-S 的效率比表示 Naive 的时间分别和它们的时间的比值.

4 相关工作

近年来, 学者们提出大量检测多线程程序原子性违反的方法[5-10,20,27-29]. Lu 等人[20] 基于同一原子区中共享数据的访问交错定义了可串行化. 在此基础上, 他们提出了访问交错不变量, 并根据该不变量检测原子性违反. 在他们后续的工作中进一步提出 CTrigger[27], 通过轨迹分析以识别不可串行交错, 并测试低概率交错以暴露原子性违反. Lucia 等人[28] 提出 Atom-Aid, 使用硬件签名来检测原子性违反, 动态地调整块边界, 在不需要任何程序注释的情况下规避缺陷. Chew 等人[29] 提出 Kivati, 通过结合静态和动态分析以检测和避免原子性违反. Wang 等人[10] 提出了一种基于预测的检测原子性违反的方法, 通过监视执行交错以记录执行序列, 并根据访问交错候选序列预测潜在的缺陷. Razavi 等人[8] 提出一种基于人工智能的预测原子性缺陷的方法. Sorrentino 等人[9] 提出一种算法挖掘一组导致原子性违反的执行序列, 并使程序按这种序列执行以暴露缺陷. 然而, 中断模型无论在调度、同步还是抢占关系上都与线程模型不同. 而且, 大多数方法所依赖的动态分析很难直接应用于中断驱动型程序中.

目前有一些方法用于检测中断驱动型程序中的数据竞争[19,30-33]. Wang 等人[19] 提出 SDRacer, 将静态分析与符号执行相结合, 通过为中断驱动型嵌入式软件生成输入数据来自动检测数据竞争. Wu 等人[32-33] 自动化地将中断程序顺序化为一个非确定的顺序程序, 分别采用限界模型检测框架和抽象解释的方法来分析程序中的数据竞争. Chopra 等人[30] 为中断程序定义了数据竞争与 happens-before 的自然概念, 他们还提出不相交块的概念来定义同步以及高效的“sync-CFG”, 从而展开静态分析. 然而中断程序经常使用定制的同步机制, 例如用户定义的基于标志变量的同步. 很多数据竞争是专门设计出来进行数据共享的, 这些数据竞争都是良性的, 若以数据竞争为检测目标, 会存在大量误报. 相比之下, 本文的方法侧重于最可能有害的原子性违反.

据我们所知, 目前只有文献 [4, 15, 17] 可用于检测中断驱动型程序中的原子性违反. 文献 [4] 采用基于抽象解释的方法, 可以有效地识别程序中潜在的原子性违反. 然而, 由于其分析对路径不敏感, 存在

大量误报. 文献 [15] 提出基于限界模型检测的检测方法，可以得到比较准确的结果，但可扩展性很差. 文献 [17] 采用了阶段性的方法，可以同时实现更高的精度和分析. 而本文在文献 [17] 的基础上增加了精确的共享数据分析，使结果进一步精确，能够有效检测工业规模的中断驱动型嵌入式软件.

5 总　　结

本文对航天嵌入式软件中原子性违反缺陷的表现形式进行了系统的实证研究，获得了原子区范围、中断嵌套层数、共享数据访问交错模式、访问方式、访问粒度等 5 方面的表现特征. 基于这些特征，提出了一个精确、高效的中断驱动型程序原子性违反静态检测方法 intAtom-S. 该方法首先进行精确的共享数据分析，将共享数据访问粒度精确至数组元素，并可识别标志变量访问与共享的内存映射 I/O 地址. 然后，使用轻量级数据流分析技术匹配符合缺陷访问交错模式的所有并发 3 次访问序作为潜在的原子性违反缺陷候选. 最后，采用基于路径条件的约束求解对缺陷候选中的串行访问对和并发 3 次访问序进行路径可行性分析，逐步消除误报，得到最终的原子性违反结果. 本文分别在基准测试集与真实航天嵌入式软件中对 intAtom-S 进行了评估. 结果表明，intAtom-S 显著降低了误报率，提高了检测效率，并可满足真实航天嵌入式软件的检测需求.

进一步的研究工作包括：

1）中断驱动型软件的设计中往往隐含了一些时序约束，而这种约束并不体现在代码中，对于以源代码为唯一分析对象的工具，该类误报很难消除. 未来可考虑对由用户创建的系统时序模型进行精确检测；

2）使用访问交错模式虽然很好地刻画了原子性违反，但其中仍涵盖了一些并不是缺陷的情况，未来可以进一步精化该模式.

作者贡献声明：于婷婷和李超为共同第一作者，提出了算法框架和实验方案，完成实证研究并撰写论文；王博祥负责完成工具开发和实验；陈睿提出总体研究思路并撰写论文；江云松提出指导意见并修改论文.

参　考　文　献

[1] Sung C, Kusano M, Wang C. Modular verification of interrupt-driven software[C]//Proc of the 32nd IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2017: 206−216

[2] Regehr J. Random testing of interrupt-driven software[C] //Proc of the 5th ACM Int Conf on Embedded Software. New York: ACM , 2005: 290−298.

[3] China Academy of Space Technology Software Product Assurance Center. Analysis Report on Software Quality Problems of China Academy of Space Technology[R]. 2015(in Chinese)
(中国空间技术研究院软件产品保证中心. 中国空间技术研究院软件质量问题分析报告[R]. 2015)

[4] Chen Rui, Yang Mengfei, Guo Xiangying. Interrupt data race detection based on shared variable access order pattern[J]. Journal of Software, 2016, 27(3): 547−561 (in Chinese)
(陈睿, 杨孟飞, 郭向英. 基于变量访问序模式的中断数据竞争检测方法[J]. 软件学报, 2016, 27(3): 547−561)

[5] Flanagan C, Freund S N, Yi J. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs[C] //Proc of the ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2008: 293–303

[6] Park S, Harrold M J, Vuduc R. Griffin: Grouping suspicious memory-access patterns to improve understanding of concurrency bugs[C]//Proc of the Int Symp on Software Testing and Analysis. New York: ACM, 2013: 134–144

[7] Park S, Vuduc R, Harrold M J. unicorn: A unified approach for localizing non-deadlock concurrency bugs[C] //Proc of the Software Testing, Verification and Reliability. Hoboken: Wiley, 2015: 167–190

[8] Razavi N, Farzan A, McIlraith S A. Generating effective tests for concurrent programs via AI automated planning techniques[J]. International Journal on Software Tools for Technology Transfer, 2014, 16(1): 49−65

[9] Sorrentino F, Farzan A, Madhusudan P. PENELOPE: Weaving threads to expose atomicity violations[C] //Proc of the 18th ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2010: 37–46

[10] Wang Chao, Limaye R, Ganai M, et al. Trace-based symbolic analysis for atomicity violations[C] //Proc of the Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2010: 328–342

[11] Park S. Fault comprehension for concurrent programs[C] //Proc of the 35th Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2013: 1444–1446

[12] Park S, Vuduc R W, Harrold M J. Falcon: Fault localization in concurrent programs[C] //Proc of the 32nd ACM/IEEE Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2010: 245–254

[13] Sadowski C, Freund S N, Flanagan C. SingleTrack: A dynamic determinism checker for multithreaded programs[C] //Proc of the European Symp on Programming. Berlin: Springer. 2009: 394–409

[14] Wang Liqiang, Stoller S D. Accurate and efficient runtime detection of atomicity errors in concurrent programs[C] //Proc of the 11th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2006: 137–146

[15] Feng Haining, Yin Liangze, Lin Wenfeng, et al. Rchecker: A cbmc-based data race detector for interrupt-driven programs[C] //Proc of the IEEE 20th Int Conf on Software Quality, Reliability and Security Companion (QRS-C). Piscataway, NJ: IEEE, 2020: 465–471

[16] Racebench website. 2019.https://github.com/chenruibuaa/racebench.

[17] Li Chao, Chen Rui, Wang Boxiang, et al. Precise and efficient atomicity violation detection for interrupt-driven programs via staged path pruning[C] //Proc of the 31st ACM SIGSOFT Int Symp on Software Testing and Analysis, New York: ACM, 2022: 506–518.

[18] Wang Boxiang, Chen Rui, Li Chao, et al. SpecChecker-ISA: A data sharing analyzer for interrupt-driven embedded software[C] //Proc of the 31st ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2022: 801−804

[19] Wang Yu, Wang Linzhang, Yu Tingting, et al. Automatic detection and validation of race conditions in interrupt-driven embedded software[C] //Proc of the 26th ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2017: 113–124

[20] Lu Shan, Tucek J, Qin Feng, et al. AVIO: Detecting atomicity violations via access interleaving invariants[C] //Proc of the Int Conf on Architectural Support for Programming Languages & Operating Systems. New York: ACM, 2006: 37–48

[21] Codish M, Mulkers A, Bruynooghe M, et al. Improving abstract interpretations by combining domains[J]. ACM Transactions on Programming Languages and Systems, 1995, 17(1): 28−44

[22] Vaziri M, Tip F, Dolby J. Associating synchronization constraints with data in an object-oriented language[J]. ACM Sigplan Notices, 2006, 41(1): 334−345

[23] DG website. 2021.https://github.com/mchalupa/dg

[24] Shi Qingkai, Xiao Xiao, Wu Rongxin, et al. Pinpoint: Fast and precise sparse value flow analysis for million lines of code[C] //Proc of the 39th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2018: 693–706

[25] Snelting G, Robschink T, Krinke J. Efficient path conditions in dependence graphs for software safety analysis[C] //Proc of the ACM Transactions on Software Engineering and Methodology (TOSEM). New York: ACM, 2006: 410–457

[26] Huang J. Scalable thread sharing analysis[C] //Proc of the 38th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2016: 1097–1108

[27] Lu Shan, Park S, Zhou Yuanyuan. Finding atomicity-violation bugs through unserializable interleaving testing[C] //Proc of the IEEE Transactions on Software Engineering. Piscataway, NJ: IEEE, 2011: 844–860

[28] Lucia B, Devietti J, Strauss K, et al. Atom-aid: Detecting and surviving atomicity violations[C] //Proc of the 2008 Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2008: 277–288

[29] Chew L, Lie D. Kivati: Fast detection and prevention of atomicity violations[C] //Proc of the 5th European Conf on Computer Systems. New York: ACM, 2010: 307–320

[30] Chopra N, Pai R, D 'Souza D. Data races and static analysis for interrupt-driven kernels[C] //Proc of the European Symp on Programming. New York: Springer, 2019: 697–723

[31] Yu Tingting, Srisa-an W, Rothermel G. 2012. SimTester: A controllable and observable testing framework for embedded systems[C] //Proc of the 8th ACM SIGPLAN/SIGOPS Conf on Virtual Execution Environments. New York: ACM, 2012: 51–62

[32] Wu Xueguang, Wen Yanjun, Chen Liqian, et al. Data race detection for interrupt-driven programs via bounded model checking[C] //Proc of the IEEE Seventh Int Conf on Software Security and Reliability Companion. Piscataway, NJ: IEEE, 2013: 204–210

[33] Wu Xueguang, Chen Liqian, Miné A, et al. Static analysis of run-time errors in interrupt-driven programs via sequentialization[J]. ACM Transactions on Embedded Computing Systems, 2016, 15(4): 1−26

Yu Tingting, born in 1992. PhD. Her main research interests include dependable embedded software, program analysis and verification.

于婷婷, 1992 年生. 博士. 主要研究方向为可信嵌入式软件、程序分析与验证.

Li Chao, born in 1992. PhD candidate. His main research interests include dependable embedded software, program analysis and verification.

李　超, 1992 年生.博士研究生.主要研究方向为可信嵌入式软件、程序分析与验证.

Wang Boxiang, born in 1994. Master. His main research interest includes program analysis and verification.

王博祥, 1994 年生. 硕士. 主要研究方向为程序分析与验证.

Chen Rui, born in 1984. PhD, professor. Member of CCF. His main research interests include dependable embedded software, concurrency bug detection and formal verification.

陈　睿, 1984 年生. 博士, 研究员. CCF 会员. 主要研究方向为可信嵌入式软件、并发缺陷检测和形式化验证.

Jiang Yunsong, born in 1978. Professor. His main research interest includes spacecraft software engineering.

江云松, 1978 年生. 研究员. 主要研究方向为航天器软件工程.