面向无线传感器网络应用的自适应调试方法

李 丰 霍 玮 冯晓兵

(中国科学院计算技术研究所系统结构重点实验室 北京 100190)

摘 要 传感网技术是物联网得以实现的重要基础.然而,受到资源有限以及程序行为不确定等因素的影响,无线传感器网络上编程和调试的难度尤甚于普通的分布式程序.文中提出了一种面向无线传感器网络程序的源码级错误诊断方法.该方法采用基于全局量计数器的方法进行程序追踪,然后根据追踪日志重放错误执行轨迹,支持属性违反错误的分析和调试.同时,通过依赖分析确定与属性相关的程序片段,并根据系统资源约束以及用户反馈,自适应调整追踪这些程序片段的代码,以满足系统资源的限制,支持错误定位.文中以 Open64 编译器为基础,实现了一个针对 TinyOS 操作系统中 nesC 程序错误诊断的原型系统.实验数据表明,此方法能够有效地控制确定性重放技术的时空开销,有力地支持了无线传感器网络程序中属性违反类型错误的诊断.

关键词 物联网;无线传感器网络;依赖分析;确定性重放;调试

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01195

An Adaptive Debugging Approach for Wireless Sensor Network Applications

LI Feng HUO Wei FENG Xiao-Bing

(*Key Laboratory of Computer System and Architecture*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)

Abstract Wireless Sensor Networks (WSN) are gaining more attentions with the progress of Internet of Things (IoT). However, due to the constrained resources and non-deterministic behaviors, programming and debugging WSN applications still face challenges. In this paper, we propose an adaptive source-level debugging approach for WSN applications. This approach, based on dependency analysis and instrumentation, retrieves execution traces and feeds them back to the replay system which adopts a global counter method. The scope and granularity of tracing and replay can be adjusted automatically according to the resource constrains and user knowledge. Moreover, a prototype debugging system for nesC applications is implemented on top of Open64 compiler. Experimental results show that this approach not only mitigates memory consumption of deterministic replay, but also improves the efficiency of error diagnosis for WSN applications.

Keywords Internet of Things; wireless sensor network; dependency analysis; deterministic replay; debugging

1 引 言

无线传感器网络能够广泛获取客观物理信息,具有大规模自组织的特性,是物联网得以实现的重要基础.同时,当前物联网应用的发展也对传感网技术提出了更加强烈的需求.然而,无线传感器网络的大规模部署和管理迄今仍面临诸多挑战,应用程序

收稿日期:2011-01-13;最终修改稿收到日期:2011-05-27.本课题得到国家"九七三"重点基础研究发展规划项目基金(2011CB302504)、国家核高基重大专项项目(2009ZX01036-001-002)和国家自然科学基金创新研究群体科学基金(60921002)资助.**李 丰**,女,1985 年生,博士研究生,主要研究方向为程序分析及错误诊断.E-mail: lifeng2005@ict.ac.cn. **霍 玮**,男,1981 年生,助理研究员,主要研究方向为程序分析和错误检测. **冯晓兵**,男,1969 年生,研究员,博士生导师,主要研究领域为先进编译技术及相关工具环境.

中错误的诊断是阻碍其发展的重要因素之一.由于受自组织特性、资源受限以及通信不可靠等因素的影响,无线传感器网络上编程和调试的难度更甚于普通的分布式程序.

分布式系统中,程序调试工作面临的主要难题是如何解决由程序执行的不确定性带来的错误难以复现的问题.确定性重放技术是一种公认的解决方法.该技术通过动态插桩或特殊硬件辅助,在程序执行的过程中,追踪程序的不确定性行为,构成追踪日志.一旦程序失效,或出现执行结果不符合预期的情况,该技术将根据追踪日志重现错误执行轨迹,为错误根源的定位提供支持.确定性重放技术的难点在于如何缩减技术应用时的时空开销.尽管近十年间,相关研究工作收效颇丰,但对无线传感器网络应用而言,受传感器结点上计算、存储以及通信资源的限制,该技术仍无法直接用于无线传感器网络中程序的调试.以 TelosB 传感器结点为例,其上只提供 48 KB的闪存空间(program flash memory),10 KB 的 RAM 和 1 MB 的额外数据空间(external flash),这就极大地限制了插桩的粒度以及追踪日志的规模.但如果为了满足上述约束而仅记录粗粒度、无针对性且不完整的程序执行信息,又不利于减轻用户诊断错误的负担.

为解决上述问题,本文提出一种以依赖分析为基础,采用自适应的源码级追踪和重放技术进行错误诊断的机制,在无线传感器网络部署后辅助对程序行为的监控和错误调试.该机制简称为 ADA(Adaptive Diagnosis Assistant).本文还基于 Open64 开源编译器①,实现了一个面向 TinyOS② 操作系统上程序错误诊断的原型.

给定一个无线传感器网络程序 P,该方法首先为用户提供一套描述机制,供其描述希望在 P 的运行过程中监控的属性,该属性的违反表示产生一种程序错误;而后,依次采用以下步骤进行属性监控和错误诊断:

(1)编译分析.针对程序中的不确定性行为,通过插桩技术,分别生成两套源程序副本:追踪控制程序 *Ptrace* 和重放控制程序 *Preplay*.同时,通过依赖分析[1],提取程序中可能影响所监控属性的片段,记作 S;在此基础上,以自适应规则为指导,根据存储资源的充足程度,自动选择对 S 中不确定性行为的追踪范围和粒度,以控制追踪和重放的时空开销;

(2)日志收集.追踪信息随追踪控制程序 *Ptrace* 的执行陆续生成,构成追踪日志.当出现程序执行状态违反预期属性的情况时,各传感器结点上追踪日志将回传到汇聚结点,作为复现错误执行轨迹的依据.

(3)重放诊断.重放控制程序 *Preplay* 根据追踪日志在 PC 机上复现程序的错误执行轨迹,供调试工具使用.

如果第一轮重放的粒度或范围不足以辅助用户定位到错误根源,ADA 将根据用户反馈,自动调整追踪范围,精化追踪粒度,并进行新一轮的追踪日志收集和错误轨迹重放,直到定位错误根源.本文将上述迭代称为自适应迭代.为确保重放的平台无关性和机器无关性,追踪和重放控制代码的插入均在源码级完成.

本文的贡献在于:

(1)提出一种面向无线传感器网络中程序的自适应追踪策略,有效地控制错误追踪的时空开销.该策略可依据程序特征以及资源可用状况,自动生成用于追踪、重放程序执行轨迹的源代码,并根据用户诊断的反馈自动调整追踪和重放的粒度和范围.

(2)提出一套在无线传感器网络部署后进行错误调试的解决方案,该方案以程序错误的离线重放技术为基础,能够有效地提高错误诊断效率.

本文第 2 节介绍依赖关系分析与确定性重放等相关概念;第 3 节简要介绍本文方法的处理流程;第 4 节具体描述该方法的设计与实现;第 5 节介绍并分析实验结果;第 6 节介绍相关工作;第 7 节总结全文工作并提出展望.

2 背景介绍

2.1 依赖关系分析与程序切片

程序切片[2]是一项程序提取技术,由 Mark Weiser 最先提出.该技术能够从整个程序中提取出可能影响某个变量在程序中某个位置上的取值的程序片段.由程序点 P 和变量 V 构成的二元组$\langle P,V\rangle$称为切片标准,由提取出的程序片段构成的语句集合称为切片.切片程序执行后得到的变量 V 的值应与源程序执行到程序点 P 处时变量 V 的值一致.

依赖关系分析[1]是程序切片的基础.程序语句间的依赖关系是由语句间数据流或控制流形成的一种约束,分别称为数据依赖和控制依赖.此处的数据

① http://www.open64.net/
② http://www.tinyos.net/

依赖专指流依赖(也称为真依赖).语句 S2 数据依赖于 S1,当且仅当,S1 中定义了变量 x,S2 中使用变量 x,并且存在一条从 S1 到 S2 的非空路径,该路径上没有对 x 的定值.如果存在一条从 S1 到 S2 的非空路径,S2 后控制该路径上除 S1 之外的所有结点,但不后控制 S1,则称语句 S2 控制依赖于 S1.一组语句之间的依赖关系可以用有向图来表示,其中,结点代表语句,边代表依赖关系,构成的有向图称为依赖图.

目前主流的程序切片算法可分为两类:基于控制流图的数据流迭代方法[2]和基于系统依赖图(System Dependence Graph,SDG)[3]的图可达算法.SDG 是对程序依赖图(Program Dependence Graph,PDG)的扩展.SDG 在 PDG 基础上加入了对过程间依赖关系的描述,如:输入/输出参数、函数副作用等.相对而言,PDG 上的依赖关系也称为过程内依赖关系.

2.2　确定性重放

串行程序在输入确定的前提下,总是执行确定的路径,获得确定的输出.然而,受线程交替、进程通信等不确定性行为的影响,同一并行程序即便在输入数据确定的前提下,执行行为也可能不同.以复现并行程序的执行轨迹为目的,分析并追踪程序执行过程中的不确定性行为的技术,称为确定性重放技术[4-6].

确定性重放可分为两类:(1)强调值的确定性(value determinism),即在重放过程中,当执行到内存读写操作时,所读写的内存值,与程序实际执行到该位置时读写的值相同;(2)仅强调输出的确定性(output determinism)即仅要求重放能够获得与实际执行相同的输出[4].本文实现的确定性重放策略属于第一类.

3　方法概述

本文处理的对象是用 nesC 语言编写的运行在 TinyOS 操作系统上的程序(下文简称 TinyOS 程序).TinyOS 是由加州大学伯克利分校开发的一个针对无线传感器网络的嵌入式开源操作系统.一个 TinyOS 程序可分为两部分:(1)仅由任务(task)可达的代码,也称同步代码(SC);(2)至少可由一个中断服务函数到达的代码,也称异步代码(AC).异步代码的执行只能通过中断响应实现.

图 1 所示为 ADA 机制的迭代处理流程.按照约定形式添加了属性描述后的 nesC 程序,首先经过扩展后的 nesC 编译器分析和源源变换,生成具有等价语义的 C 程序,其中,对所监控的属性的判定被转化成 C 语言中的函数,并在相应的判定位置插入对该函数的调用.生成的 C 程序经扩展的 Open64 编译器上的依赖分析后进入自适应迭代流程.首次迭代中,根据得到的依赖信息,由 nesC 编译器生成的 C 程序经自适应插桩模块处理,再由源源变换,分别生成用于控制对错误轨迹进行追踪的 C 程序 *Ptrace* 和包含重放控制的 C 程序 *Preplay*.前者经交叉编译(譬如:使用 msp430-gcc 交叉编译器①),产生可在传感器结点上执行的二进制码,并在执行过程中生成作为重放输入的追踪日志.后者则由本地编译器编译,生成可离线执行的二进制码,并在执行过程中读取追踪日志,重现错误轨迹,供用户诊断.自适应插桩模块将根据用户提供的反馈信息(譬如:错误根源所在的范围、定值不符合预期的可疑变量等),指导新一轮迭代,直到发现错误根源.

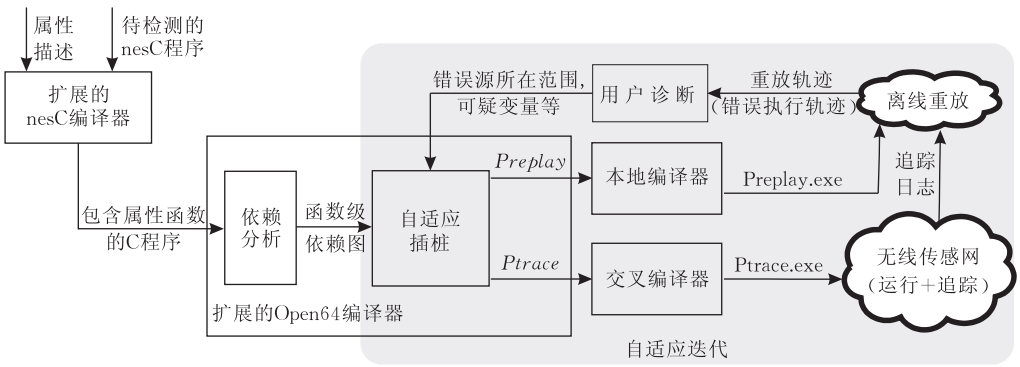


图 1　ADA 框架概貌

本文后续章节将以一个用 nesC 语言编写的 TinyOS 简单应用程序为例,介绍 ADA 的工作原理.该程序使用 TinyOS-2.x 自带的汇聚树协议

① http://mspgcc.sourceforge.net/

(CTP),控制远程传感器结点周期性地向基站发送数据.所发送的数据是连续的(0～127).图 2 为该收集程序的部分代码片段(暂时忽略字母 N 标注的语句).程序中,传感器结点在消息发送前,首先检查 sendBusy 标志(第 17 行),只有当 sendBusy 为 FALSE(发送缓冲区空闲)时,待发送的消息才能进入发送缓冲区,并将 sendBusy 设为 TRUE(7～10 行),缓冲区中的消息成功发送后,sendBusy 标志复位(第 14 行).

```
1. module EasyCollectionC {…}
2. implementation {
3. message_t packet;
4. bool sendBusy=FALSE;
   //初始化用于描述属性的辅助变量
N  uint8_t old_data=0;
N  uint8_t counter=0;
   …
5. void sendMessage() {
     //连续数据 0～127
6.   msg->data=(data++)/128;
7.   if (call Send.send(&packet,
       sizeof(EasyCollectionMsg)) !=SUCCESS)
8      call Leds.led0On( );
9    else
       //待发送的消息进入发送缓冲区
       //缓冲区标志置位
10.    sendBusy=TRUE;
   }
11. event void Send.sendDone
     (message_t* m, error_t err) {
12.  if (err !=SUCCESS)
13.    call Leds.led0On( );
     //发送方属性:成功发送的数据总是连续的
N    /*@assert=(old_data+1==msg->data)@*/
     //消息发送成功,缓冲区标志复位
14.  sendBusy=FALSE;
N    old_data=msg->data;
   }
15. event void Timer.fired () {
     //计时器到时(50ms),触发消息发送
16.  call Leds.led2Toggle();
     //消息发送前检查缓冲区是否空闲
17.  if (!sendBusy)
18.    sendMessage();
     …
   }
19. event void RadioControl.startDone(…) {
     …
     //为基站结点添加计时器 AssertTimer
N    call AssertTimer.startPeriodic(60);
   }
20. event message_t* Receive.receive
              (…, void* payload, …) {
     …
     //统计基站接收到的消息数目
N    counter++;
   }
     //基站属性函数
N  /*@assert func*@/
N  event void AssertTimer.fired ( ) {
     //基站属性:指定时间内接收到
     //来自所有结点的正确消息
N    if (counter<10)
N      fprintf ("EasyCollection Err\n");
N    else
N      counter=0;
N  }
```

图 2 带属性描述的 nesC 程序片段

我们向图 2 所示程序注入错误,注销消息发送之前对 sendBusy 标志位的检查(第 17 行).注销上述检查语句后,每当计数器 Timer 到时,新生成的消息就会直接进入发送缓冲区,取代缓冲区中尚未成功发送的消息,从而导致消息丢失.当然,如果消息发送频率太低,缓冲区在大多数时间内都是空闲的,那么即使不检查 sendBusy 标志,程序的行为也可能符合预期.为此,我们提高消息发送的频率,指定传感器结点每隔 50 ms 向基站发送一条消息.

为捕获注入的错误对程序行为的影响,首先需要向程序中插入不变属性,监控程序行为.对于发送方结点而言,成功发送的数据应当是连续的.而基站结点应当能在指定时间内(譬如,每隔 60 ms)收到来自所有消息发送结点(假定共有 10 个消息发送结点)发出的消息.图 2 中字母 N 标注的语句代表用户添加的属性描述语句.属性描述语句分两类:(1)属性监控语句;(2)辅助描述语句.其中,由特殊字符串"/*@"和"@*/"标注的语句或函数定义统称属性监控语句,如:发送方属性由语句 assert=(old_data+1==msg->data)监控,基站属性的监控则由函数 AssertTimer.fired 实现.属性监控语句经扩展的 nesC 编译器源源转换后,将生成作为依赖分析模块的初始输入的属性函数(F_{Inv}).为维护属性监控语句中新出现的变量,如:old_data、counter,用户还需额外添加用于初始化或更新这些变量的辅助描述语句.

添加属性描述后的程序经 nesC 编译器源源变换后,将生成具有等价语义的 C 程序,作为后续分析的输入.

4 ADA 的设计与实现

ADA 的基本思路是利用确定性重放技术辅助程序员诊断 TinyOS 程序错误.需要解决的关键问题是:

(1)如何保证重放的确定性,包括确定重放的起点和需要追踪不确定性行为;

(2)如何降低错误追踪和重放的代价以满足资

源的限制.

对于问题 1,由于无线传感器网上运行的 TinyOS 程序往往是无穷迭代程序,并通过中断服务函数异步地完成系统功能.这就意味着:

(1)需要追踪外部输入和中断响应等不确定性行为,以支持确定性的重放;

(2)由于对无穷迭代循环的追踪日志可能会无限增长,所以重放完整的无穷迭代并不合理,需要选择一个合理的重放起点,即检查点.

对于问题(2),为使基于确定性重放的离线调试技术能够沿用在资源极端受限的无线传感器网络程序的错误诊断中,本文采用以依赖关系分析为基础的自适应迭代方法,降低错误追踪和重放的代价.同时,通过剔除与监控的属性无关的代码,减少用户在诊断错误时需要关注的程序轨迹的规模,减轻用户负担.用户既可以从所监控的属性的最近一次被违反的位置出发,根据引用-定值关系以及关键位置上的程序状态,从重放轨迹中找寻错误定值的来源,也可以将重放轨迹作为调试器的输入,进而使用设置断点等调试方法定位错误根源.

4.1　确定性重放

4.1.1　不确定性行为追踪

为了确定性地重放 TinyOS 程序的错误轨迹,必须对与所监控的属性相关的不确定性程序行为进行追踪,追踪对象包括:(1)非确定性输入,(2)中断响应位置.同时,为了尽可能地控制追踪开销,我们仅对与所监控的属性相关的不确定性程序行为进行追踪.本节具体讨论确定性重放方法(对本文重放策略的证明详见附录 2 的确定性重放证明),分析与属性相关的程序行为的方法详见第 4.2 节.

由于本文采用的是多结点独立追踪、并行重放的方法,故而从其它结点接收到的信息对于接收方结点而言也属于非确定性输入.非确定性输入包括由传感器收集到的数据以及结点间通过无线通信收发的数据.因为上述数据的收发均由 TinyOS 提供的系统库函数完成,我们仅需追踪消息收发函数中 message_t 类型的参数,即可支持对非确定输入的重放.

在此基础上,下文仅考虑程序中的不确定性行为是中断的确定性重放方法.当前知名的确定性重放工具在处理中断时大多采用追踪中断响应副作用的策略,在重放时用记录的副作用替代实际应该响应的中断.然而该处理策略不足以支持错误诊断,特别是对通过中断响应为主要功能实现的 TinyOS 程序而言,中断处理函数的内部执行行为很可能是错误传播路径的重要组成部分,不重放这些行为就不能有效地进行诊断.针对上述问题,本文提出了一种对中断响应位置的追踪/重放策略,以确保对 TinyOS 程序的确定性重放.

TinyOS 程序可分为同步代码和异步代码两部分.其中,异步代码只能通过中断响应被执行.中断响应函数对其外部的同步代码和异步代码的影响包含两个方面:(1)修改外部代码中可能使用的全局变量,(2)向任务队列中添加新任务.如果任务队列是先入先出(FIFO)队列,则一旦确定了中断响应的位置和顺序,即可保证重放时的任务队列与追踪程序执行过程中的任务队列一致;否则,也可以通过追踪任务派生函数复现任务队列.

给定中断处理函数 $Intr_i$、$Intr_j$ 和函数 F(F 也可以是中断处理函数),将 $Intr_i$、$Intr_j$ 和 F 三者之间可能共享的全局量集合记作 G.对任意 $f,g\in G$ 以及 F 的执行语句序列 $S=\{s_1,s_2,\cdots,s_{n-1},s_n\}$,如果 S 满足:s_1 是对 f 的读/写操作,s_n 是对 g 的读/写操作,$s_2,\cdots,s_{n-1}$ 均没有对 G 中变量的读/写操作,则在开区间(s_1, s_n)中任意位置分别执行一次 $Intr_i$ 和 $Intr_j$,且 $Intr_i$ 和 $Intr_j$ 的前后顺序固定,在 s_n 执行前 G 中变量的值都是相同的.基于以上观察,只要能够记录序列 S 的开始和终止位置以及该区间内中断的响应顺序,就能重放函数 F 中的语句序列 S 及 s_n 处对 g 的读/写操作的值.

基于上述分析,我们采用记录全局量操作数目的方法追踪中断响应位置.该方案维护一个全局量操作计数器 C,C 在每个函数(包括中断处理函数)的每个执行实例的入口处被保护并重置,出口处恢复;每执行一个全局变量读/写操作,C 的值加 1.这样,我们可以根据 C 的值,将任意函数(包含中断处理函数)的任意执行实例中除全局变量读/写操作之外的部分划分成若干个互不重叠的执行区段,其中每个执行区段都不包含显式的全局变量操作(通过响应中断或函数调用修改/使用的全局变量除外),如图 3 所示.由函数名(F)、该函数的某次执行实例(E)以及全局量计数器的值(C)构成的三元组可以唯一确定一个区段.对中断响应顺序的追踪是通过在追踪日志中顺序记录各执行区段中被响应的中断对应的处理函数来实现的;重放时,当执行到对应区段时可依次调用所记录的中断处理函数,复现程序的执行.该方案可以处理带中断嵌套的程序.

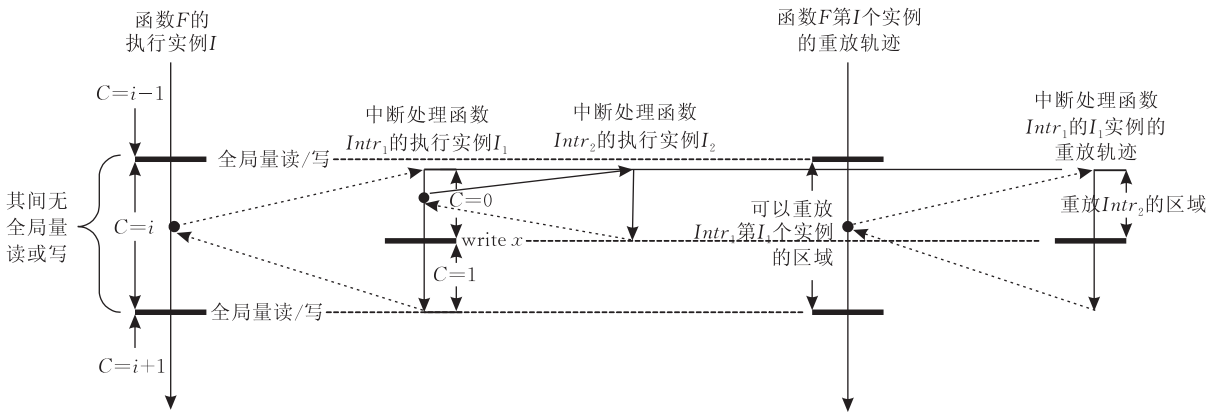


图 3 全局量计数器方案示例

4.1.2 检查点设置

受到传感器结点上存储和通信资源的限制，能追踪并重放的往往只是错误发生之前的一部分执行轨迹. 为保证机制的正常运转，ADA 在追踪不确定性行为的同时，也在程序中关键位置插入检查点.

检查点既是重放的候选起始位置，也是辅助用户诊断错误根源的依据之一. 因此，所选择的检查点应当能够代表程序中一个相对独立且完整的部分的初始或终结状态，譬如，最外层循环的迭代入口、函数入口、函数出口等. 最外层循环的迭代入口检查点记录从该位置到属性违反点之间可能与属性相关且向前暴露的变量. 函数入口检查点的状态，包含了本函数所使用的且可能影响所监控的属性的参数和全局量的值；出口检查点的状态，包含了本函数所修改的且可能影响所监控的属性的参数和全局量的值. 用户通过分析上述状态，可以初步判定函数的执行行为是否符合预期，中断响应是否对当前函数造成了非预期影响等. 此外，检查点的选择也使重放的起始点和范围更加灵活，进而更好地辅助用户调试和对错误根源的诊断.

检查点的设置还应结合传感器结点上存储资源的限制，设置依据将在 4.2 节介绍.

4.1.3 代码示例

图 2 所示的程序经 nesC 编译器源源变换后，生成的 C 程序与 nesC 程序中命名的映射关系如下：nesC 程序中模块 M 中的函数 F、变量 X、事件或命令 E 分别对应 C 程序中 M$F、M$X 和 M$E，模块 M 中接口 I 的事件或命令 E 被映射成 MIE，依此类推. 譬如，C 函数 EasyCollectionC$AssertTimer$fired 对应 EasyCollectionC 模块的 AssertTimer.fired 事件.

C 程序经依赖分析和自适应插桩模块的依次处理，分别生成包含追踪和重放控制代码的 C 程序，程序片段分别如图 4、图 5 所示. 图中加粗的语句为插桩库函数的调用语句（对插桩库函数的介绍详见附录 1 中插桩库介绍）. 图中加粗语句是对完成追踪或重放功能的插桩库函数的调用. 图中的虚框代表追踪和重放过程中的不安全因素，处理方案留待 4.3 节讨论.

```
void EasyCollectionC$AssertTimer$fired ()
{
  WriteInOut(…); //函数入口/出口追踪
  ResetMemCnt( ); //计数器复位
  _load_cnt=TOS_NODE_ID;
  IncMemCnt( );  //计数器递增
   if (_load_cnt==0) {
 T1  _load_cnt0= EasyCollectionC$counter;
 T2  IncMemCnt( );
 T3  if (_load_cnt0<=10) {
       …
    } else {
      EasyCollectionC$counter=0;
      IncMemCnt( );
    }
  }
  WriteInOut(…); //函数入口/出口追踪
  return;
}

void VirtualizeTimerC$0$Timer$fired
  (uint8_t arg_0x410f4030) {
 T4  WriteInOut(…);
 T5  ResetMemCnt( );
    //检查点追踪(候选重放起始点)
    WriteStartPoint(…, TOS_NODE_ID);
    WriteStartPoint(…, EasyCollectionC$counter);
   …
}
```

图 4 插桩后生成的追踪控制程序

4.2 依赖分析与自适应迭代

追踪和重放对存储和通信的压力来自两个方面：(1) 向源程序中插入用于追踪的代码，可能造成可执行码的规模超过传感器结点的代码存储上限；(2) 追踪过程中记录检查点状态和不确定性行为的

```
void EasyCollectionC$AssertTimer$fired()
{
  CheckFunc(…); //函数实例检查
  ResetMemCnt( );//计数器复位
  _load_cnt=TOS_NODE_ID;
  CheckIntr( );    // 中断响应检查
  if(_load_cnt==0) {
R1  _load_cnt0=EasyCollectionC$counter;
R2  CheckIntr( );
R3  if(_load_cnt0<=10) {
      …
    } else {
      EasyCollectionC$counter=0;
      CheckIntr ( );
    }
  }
  CheckFunc(…); //函数实例检查
  return;
}

void VirtualizeTimerC$0$Timer$fired
     (uint8_t arg_0x410f4030) {
  WriteInOut(…);
  ResetMemCnt( );
  // 检查点载入(候选重放起始点)
  ReadBuffer(…, TOS_NODE_ID);
  ReadBuffer(…, EasyCollectionC$counter);
  …
}
```

图 5　插桩后生成的重放控制程序

日志可能远大于传感器结点上的 RAM 空间，需要频繁地将 RAM 中的数据写入额外存储空间，并最终传向基站. 此外，插桩代码的执行以及频繁的日志数据写入、传输都可能提高程序的执行时间和系统的功耗. 为解决上述问题，本文借助依赖关系分析，剔除对与所监控的属性之间不存在直接或传递依赖关系的程序行为的追踪，并采用自适应的插桩策略限制代码规模，以缓解由此引发的存储和通信压力.

4.2.1　函数级依赖图

为实现对需要追踪的信息的自适应选择，本节首先引入一种新的依赖图表示方法：函数级依赖图. 如前所述，程序语句之间的依赖关系可以分为数据依赖和控制依赖两类，且程序内部的依赖关系可以表示成以程序语句为结点，依赖关系为边的依赖图的形式. 如果依赖图上存在一条从结点 u 到结点 v 的依赖边，则结点 v 直接依赖于节点 u；如果结点 u 又直接依赖于结点 w，则结点 v 传递依赖于结点 w. 传递依赖关系反映了程序中某个子部分的输入与输出之间的关联. 如果将系统依赖图上的过程内数据和控制依赖边，用过程间结点(函数调用点处的实参结点、函数入口出口处的形参结点以及函数对外暴露的副作用结点)之间的传递依赖边代替，则程序的系统依赖图可以塌缩成以函数为结点的函数级依赖图. 函数级依赖图仅刻画各函数对外暴露的依赖关系，其所包含的信息既指明了检查点需要追踪的内容，又对纷繁复杂的语句级依赖进行了整合与压缩，为后续分析提供了便利.

以图 2 所示的程序为例，翻译生成的 C 程序，经依赖分析生成的函数级依赖图如图 6 所示. 图中，

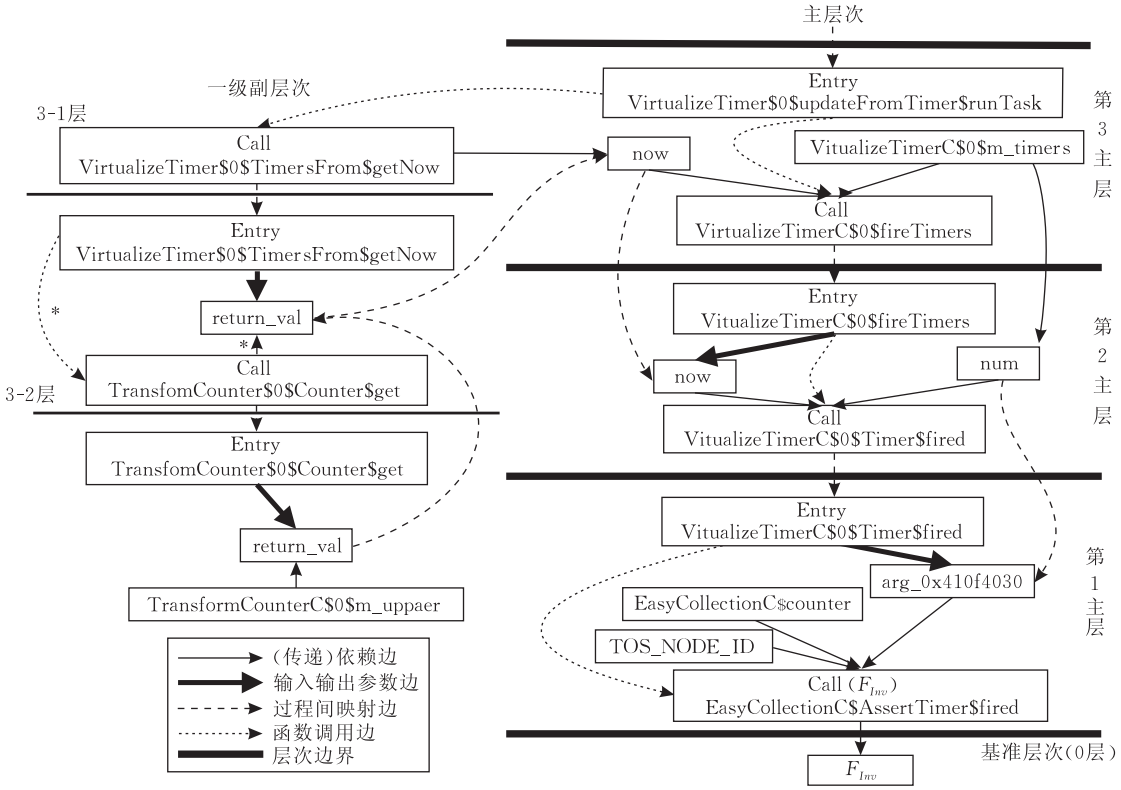


图 6　函数级依赖图(部分)

F_{Inv}代表所监控的属性对应的函数，下文简称属性函数；实线有向边代表传递依赖；虚线有向边代表函数过程间映射关系，包括：从函数调用点到被调用函数入口的映射关系，形参和实参之间的映射关系；加粗的实线有向边代表函数的形式参数，包括输入形参、输出形参和返回值；虚点有向边则代表函数调用.

图中从函数 VirtualizeTimerC0TimerFrom$getNow 的返回值结点 return_val 到函数调用结点 Call TransformCounter0Counter$get 的实线有向边上的“*”符号，代表经内联优化后的函数级依赖边.虽然，该程序的函数调用图上，从 VirtualizeTimerC0TimerFrom$getNow 到 TransformCounter0Counter$get 的函数调用链的实际长度为 4，但由于调用链上的每个函数都只包含一条函数调用语句，且该函数的返回值就是这个函数调用语句的返回值；即：调用链上除 TransformCounter0Counter$get 之外的其它 3 个函数都相当于包装函数，故在构建函数级依赖图时可不予考虑.

实现过程中，待监控的属性，经过 nesC 编译器的源源翻译后，转化成 C 语言函数定义的形式(F_{Inv}).依赖分析模块首先为同步代码构造系统依赖图，并在其上对由 F_{Inv} 的调用位置以及其中使用的全局变量构成的二元组为切片标准对同步代码切片，切片结果表示成系统依赖图的子图的形式；然后，将上述子图塌缩成函数级依赖图的形式，其中，各个函数的输入、输出形参、对外暴露的变量以及返回值，就是在该位置设置检查点时需要追踪的内容.

由于静态无法确定中断的响应位置，故而只能保守地认为中断可能在程序中的任意非临界区域内被响应.实现时，如果某中断服务函数的副作用与选定的追踪区域内可能访问到的全局变量之间存在交集，则该中断服务函数的行为也必须包含在追踪范围内.

静态依赖分析无法确定的另一个问题是通信匹配.目前，依赖分析模块只能根据消息长度和数据类型判断通信匹配关系.此外，虽然本文采用的是各传感器结点独立记录追踪日志的方法，但由于追踪日志中包含了可能影响所监控的属性的外部输入，故而用户诊断时还可以通过分析消息头，进一步判断通信操作之间的匹配关系，以确定消息的来源.

4.2.2 依赖属性

在函数级依赖图的基础上，我们引入两个新的属性：依赖的层次和密度，作为依赖分析时选择检查点以及限制追踪、重放的范围和粒度的依据.

设属性监控函数结点 F_{Inv} 为基准层次(0 层)，若将函数级依赖图上任意结点 v 所在的依赖层次，定义为从该结点到 F_{Inv} 结点之间的所有路径上函数调用边的数目的最小值，可得图 6 中部分结点所在的依赖层次如下：结点 Entry VirtualizeTimerC0fireTimers所在的依赖层次为 2，结点 Entry VirtualizeTimerC0updateFromTimer$runTask 以及结点 Entry VirtualizeTimer0TimersFrom$getNow 所在的依赖层次均为 3.

上述依赖层次定义的缺陷在于：无法区分程序的函数调用图上来自父亲结点和兄弟结点的依赖关系.以图 6 中带边框的 VirtualizeTimerC0fireTimers 函数调用结点为例，该函数的外部依赖关系可分为两类：

(1) 父子依赖. 调用者函数 VirtualizeTimerC0updateFromTimer$runTask 中的局部变量以及所传递的全局变量决定了VirtualizeTimerC0fireTimers是否被调用；

(2) 兄弟依赖. 输入参数 now 的值取决于 VirtualizeTimerC0fireTimers 在调用图上的兄弟函数 VirtualizeTimer0TimersFrom$getNow 的返回值.

当存储资源受限时，除了限定追踪范围外，还必须对不同类型的依赖在追踪粒度上进行取舍，譬如，优先追踪父子依赖，并由用户依据检查点状态和重放轨迹，逐步缩小错误根源所在的范围，并逐步精化对此范围内兄弟依赖的追踪，直至发现错误根源.

为此，我们对依赖层次的定义进行修订，用主层次表示父子依赖关系，用副层次表示兄弟依赖，称为一级副层次，一级副层次又可分为若干二级副层次，依此类推.层次间的分界线将作为候选的追踪检查点和重放起始点.

根据修订后的依赖层次定义，图 6 中的加粗水平线代表依赖主层次之间的分界线，除 0 层外，每两条分界线之间代表一个主层次.主层次的编号为该主层次与基准层次之间包含的主层次的数目加 1.根据新定义，结点 Entry VirtualizeTimerC0fireTimers 位于编号为 2 的主层次上，简称第 2 主层；结点 Entry VirtualizeTimerC0updateFromTimer$runTask 位于编号为 3 的主层次上，简称第 3 主层；结点Entry VirtualizeTimer0TimersFrom$getNow 则位于第 3 主层的一级副层的第 1 层上，记作 3-1 层，代表该结点与属性函数之间既有父子依赖关系，

又有兄弟依赖关系.同理,结点 Entry Transform-CounterC0Counter$get 在第 3 主层一级副层的第 2 层上,记作 3-2 层.

每个依赖层次的依赖密度定义为各级副层次中函数结点(图 6 中以 Entry 为前缀的结点)的总和,总和越大,密度越高.依赖密度是也是选择检查点的依据之一.

4.2.3　自适应迭代算法

自适应插桩模块将根据依赖属性和系统资源指导追踪和重放代码的生成.受到存储和通信资源的限制,每次追踪的范围和精度都是有限的.因此,在定位到错误根源之前,可能需要对自适应插桩、追踪以及重放进行多次迭代.本文将这个迭代过程称为自适应迭代.

自适应迭代规则分两类:插桩选择规则和插桩调整规则.

其中,插桩选择规则指导插桩位置的选择顺序,包括 3 条子规则:

(1) 主层次优先规则(Main-level-first Rule,MR);

(2) 高密度优先规则(High-density-first Rule,HR);

(3) 低层次优先规则(Low-level-first Rule,LR).

插桩调整规则根据用户反馈调整插桩位置,包括两条子规则:

(1) 粒度调整规则(Granularity-adjust Rule,GR);

(2) 范围调整规则(Scope-adjust Rule,SR).

为描述自适应迭代算法,本节定义了插桩状态和 4 个新函数:

插桩状态是一个二元组〈*instr_level*,*instr_type*〉,其中,*instr_level* 代表插桩的层次(如:主层次、一级副层次、1-2 层等),当基准层次发生变化后,函数依赖图上的依赖层次划分及各层的编号也将根据定义重新计算;*instr_type* 代表插桩的类型,包括:仅对非确定性行为插桩(NOND)、仅插入检查点(CKPT),或两者兼备(BOTH).〈null, null〉代表空状态.

函数 *Next_instr_level*: (*State*, *Rule*)→*State* 在程序存储空间尚有剩余的前提下,根据当前插桩状态〈*level*, *type*〉(可以是空状态)和优先级最高的自适应插桩规则(MR、HR 和 LR),选择下一个插桩备选状态〈*new_level*, *type*〉.其中,MR 规则将选择 *level* 的下一级副层次作为 *new_level*,HR 规则将选择与 *level* 同级且密度仅次于 *level* 的层作为 *new_level*,LR 规则选择与 *level* 同级且编号仅高于 *level* 的层作为 *new_level*.如果 *new_level* 不存在或不唯一,则返回空状态.

函数 *Next_instr_type*: *State*→*State* 在程序存储空间不足以支持当前插桩状态〈*level*,*type*〉的前提下,调整插桩类型,生成新的备选插桩状态〈*level*, *new_type*〉.如果 *type* 的类型为 BOTH,则选择 CKPT 作为 *new_type*;如果 *type* 的类型为 CKPT,则将新的插桩状态设为空状态.

函数 *Instr_pre*: (*Resource*,*State*)→(*Resource*, *Instr_state*)评估 *State* 对应的备选插桩状态是否能够在当前剩余的程序存储空间 *Resource* 内顺利完成以及完成后程序存储空间的余量.输出参数 *Instr_state* 代表程序存储空间对插桩状态的支持情况:支持(SUCC)、不支持(FAIL).输出参数 *Resource* 代表完成上述插桩状态后,程序存储空间的余量.初始状态下,插桩代码规模上界即传感器节点上程序存储空间的大小减去原始程序与插桩库共同编译生成的代码规模.由于插桩插入的语句都是对插桩库函数的调用语句,因此,根据函数参数的数目可以估算出该语句对代码规模的影响(根据 msp430 手册,每条 call 指令和 mov 指令均占 4 个字节).

函数 *Instr_exec*: (*P*,*State*[ ])→(*T*,*R*)根据程序 *P* 和插桩状态数组 *State*[ ],生成追踪程序 *T* 和重放程序 *R*.

自适应迭代算法 Adptive_Iter 如算法 1 所示.算法 1 首先以属性函数为基准层次,计算 FPDG 的依赖层次和依赖密度,然后根据用户反馈(首轮迭代用户反馈为空),开始新一轮自适应插桩,插桩输出的追踪和重放程序将生成可供用户分析、调试的程序执行轨迹,辅助用户定位错误根源(第 1～7 行).

算法 1.　自适应迭代算法 Adptive_Iter.

输入:属性函数 *Inv*
　　函数级依赖图 *FPDG*
　　插桩空间上限 *Resource*
　　规则优先级组合 *R_set*

输出:用户诊断反馈 *F*

1. 以 *Inv* 为基准层次,计算 *FPDG* 的依赖层次和依赖密度
2. 读取用户反馈 *F*
3. {
4. 〈*T*,*R*〉=call *Instr_Iter*(*FPDG*,*R_set*,*Resource*,*F*);
5. 执行 *T*,生成追踪日志 Log

```
6.    执行 R,用户根据 R 所复现的执行轨迹分析错误根源,更新 F
7.  } while (F!=null)
//自适应插桩子函数
//输入:函数级依赖图 FPDG,规则优先级组合 set,插桩空间上限 r 和用户反馈 f
//输出:追踪插桩程序 T,重放插桩程序 R
SubRoutine 〈T,R〉Instr_Iter (FPDG, set, r, f) {
    //初始化规则 rule(set 中优先级最高的规则),当前状态 cur_state
    rule=First_Rule(set);
    cur_state=〈null, null〉;
    //初始化动态数组 state[ ](预设规模 n)
    state[ ]=new array[n]; i=0;
8.  if (f 提示精化对 new_level 层的插桩)
9.      cur_state.instr_level=new_level;
10. if (f 提示新的基准层次)
11.     更新 FPDG 的依赖层次和依赖密度
12. while (r !=null && rule !=null) {
13.     cur_state=Next_instr_level(cur_state,rule);
14.     while (cur_state !=〈null,null〉) {
15.       〈tmp_r,instr_state〉=Instr_pre(r,cur_state);
16.       if (instr_state==SUCC) {
17.         state[i++]=cur_state;
18.         r=tmp_r;
          } else { //读取 set 中优先级仅次于 rule 的规则
19.         rule=Next_rule(R_set,rule);
          }
20.       if (rule !=null)
21.         cur_state=Next_instr_level(cur_state,rule);
          else
22.         cur_state=Next_instr_type(cur_state);
        }
23.     if (rule !=null)
24.         rule=Next_rule(R_set,rule);
    }
25. 〈T,R〉=Instr_exec(S,state[ ]);
26. return 〈T,R〉;
  }
```

自适应插桩由子函数 *Instr_Iter* 完成. 该函数首先分析用户反馈,精化本轮插桩所关注的层次范围(粒度调整),或根据用户反馈的新基准层次重新计算 *FPDG* 上的依赖层次和依赖密度(范围调整)(第 8～11 行). 如果当前的插桩空间仍有剩余(第 12 行),则根据优先级最高的规则,决定下一个备选插桩状态(第 13 行). 如果该状态并非空状态,则评估当前的插桩空间余量是否足以支持该插桩状态(第 14～15 行),如果支持,则将该备选状态存入数组 *state*[],并更新插桩空间的余量(第 16～18 行),否则,读取优先级仅次于当前规则的下一条规则(第 19 行),并根据该规则生成新的插桩状态,并再度进行评估(第 20～21 行),如果已无可用规则,则调整当前插桩状态的插桩类型,并再度进行评估(第 22 行),直到当前剩余的插桩空间能够支持调整后的插桩状态,或调整后的状态为空状态. 而后,按照优先级递减顺序继续分析下一条规则,如果下一条规则不存在,则预估分析结束(第 23～24 行),*Instr_Iter* 根据存放在 *state*[]中的插桩操作,生成追踪程序 *T* 和重放程序 *R*(第 25～26 行),本轮插桩结束.

自适应插桩规则依据用户的调试经验以及对所诊断的程序的熟悉程度,可以有两种组合方法.

规则组合 I 以追踪粒度的逐步精化为主导,在代码存储空间受限的前提下,优先强调追踪程序的整体行为(比如:关键位置上的变量状态). 构成该组合的子规则的优先级按 MR→HR→LR 的次序递减. 依据上述规则优先顺序,自适应迭代算法首先对主层次上非确定性行为以及检查点插桩;然后根据各层的密度,优先对密度最高(即依赖关系最复杂,人工分析负担最大)的主层中的所有一级副层次上的非确定性行为以及检查点插桩;如果出现两个主层次的密度相当,且没有用户反馈的情况,则优先处理层次编号较低(最靠近基准层次)的主层次中的所有一级副层次;依此类推,直到完成对函数依赖图上各级副层次的处理. 规则组合 I 适合于经验较丰富,或者对所诊断的程序较为熟悉的用户. 粒度调整规则(GR)常与规则组合 I 联用. 当用户判断错误根源可能在某个层次内,但本次迭代的追踪和重放粒度不足以证明这个判断时,自适应插桩模块将针对用户所反馈的范围以及有嫌疑的变量,生成粒度更细的追踪代码.

规则组合 II 以追踪范围的逐步扩大为主导,在代码存储空间受限的前提下,优先强调追踪靠近基准层次的程序行为细节;子规则的优先级别按 LR→MR→HR 的次序递减. 依据上述规则优先顺序,自适应迭代算法首先对层次编号最低的主层次以及其中的各级副层次上的非确定性行为插桩,并在该主层次的边界上插入检查点;如果完成对所有主层次的处理后,代码存储资源仍有富余,则设置更多的检查点,以提高重放范围选择的灵活性. 检查点的设置遵循先主层后副层,先高密度后低密度的顺

序.规则组合 II 更适合经验不足或者对程序不甚熟悉的调试用户,用户可根据追踪日志,从属性违反位置出发,逆向查找错误的传播路径,定位错误根源.范围调整规则(SR)常与规则组合 II 联用.当用户确定错误根源不在当前所重放的若干层次中,自适应插桩模块将上一轮迭代的追踪边界以及其上与属性相关的变量(也可以由用户指定有嫌疑的变量)封装成新的基准层次,重新生成插桩代码.

考虑到结点功耗以及轨迹日志记录的时空开销,ADA 的默认插桩范围仅包括主层次和第一副层次,用户可以自主选择是否遵循该范围限制.此外,为提高原型系统的灵活性和易用性,我们还为用户提供了追踪范围和粒度的选择接口.

4.2.4　迭代与错误定位

自适应迭代将对程序行为进行多次追踪和重放,这就引入了一个新问题.即便程序中只有单个错误根源,不同次追踪迭代也可能通过不同执行路径到达属性被违反的位置.当出现上述情况时,错误根源只可能位于不同执行路径的公共部分.如图 7 所示,用户诊断时可以忽略非公共部分中函数执行实例的重放轨迹.对于存在多个错误根源的情况,不同次迭代追踪并重放的可能是由不同错误根源引发的不同错误执行路径.遇到上述情况时,自适应迭代将同时维护用于追踪不同错误根源的插桩代码,但每次迭代生成的追踪日志是否由同一个错误根源引发需由用户判别.

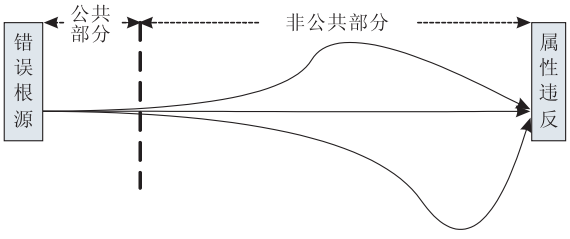


图 7　迭代追踪路径

4.3　安全性与代价

在源码级通过插桩的方式对中断和检查点进行追踪和重放可能会导致不安全因素.

图 4、图 5 所示的追踪和重放代码采用全局量操作计数方案追踪并重放中断的响应位置.其中,语句 T1～T3、R1～R3 均由 if (EasyCollectionC$counter==0)经插桩转化而来.如果,在追踪程序的执行过程中,中断响应发生在语句 T1 和 T2 之间,且该中断(记作 *I*)修改了全局量 EasyCollectionC$counter,追踪日志将记载当前函数本次实例在全局量操作计数器为 1 的区间内响应了中断 *I*.但由于 T1 在中断服务函数执行之前已经将 EasyCollectionC$counter 的值存入寄存器_load_cnt,因此,T3 使用的依然是未经中断服务函数修改的旧值.当重放程序执行到追踪日志记录的函数执行实例时,根据追踪日志,可以选择在图 5 中虚框内任意语句的前后响应中断 *I*.如果,在以上区域内随机选择的中断响应位置恰好在语句 R1 之前,那么当重放程序执行到语句 R3 时,该语句使用的就将是经中断服务函数更新后的EasyCollectionC$counter的值,由此引起的重放执行轨迹与追踪执行轨迹之间的偏差,可能导致重放轨迹与追踪的行为不一致,甚至造成重放失败.

为解决上述问题,生成追踪代码时,必须对类似图 4 中 T1 至 T3 的区间加锁(AtomicStart 和 AtomicEnd),屏蔽上述区间内的中断响应,但这势必将增加插桩代码的规模以及追踪代码的执行时间.为降低时空代价,我们对中断分析进行优化.

以图 8 所示的 C 程序片段为例,假定:所监控的属性为(a==b),且运行时,中断 I1、I2 均在全局量 b 的定值之后被响应.函数 foo 中语句 S1 和 S2 都可能影响属性(a==b)的真值.追踪代码生成模块必须在这两条语句之后插入带锁保护的全局量操作计数.但由于中断服务函数 Intr_2 仅修改全局量 b 的值,故仅 I2 的响应位置与 S2 之间的先后顺序,就可以决定 Intr_2 对 b 的定值是否可以到达 P.因此,不需要对语句 S1 进行全局量计数插桩.

```
int a,b;      //全局变量
  …
void foo ( )
{
S1  a=…          //修改全局变量 a
  …
S2  b=…          //修改全局变量 b(old_b)
  //响应中断 I1,中断处理函数 Intr_1
  //响应中断 I2,中断处理函数 Intr_2
  …   //期间不修改 a、b 的定值
P //所监控的属性:(a==b)
}
…
void Intr_1 ( )
{
  …              //既不使用也不修改 a,b
}
…
void Intr_2 ( )
{
S3  b=…          //仅修改 b(new_b)
        …
}
```

图 8　中断示例

基于以上事实,我们对中断分析进行优化.首先,按照所完成的功能,将 TinyOS 程序中的中断服务函

数分为3类:(1)仅向任务队列里添加新任务(更新存放任务队列的全局数组 SchedulerBasicP$m_next);(2)使用与所监控属性相关的全局变量;(3)修改除任务队列外与所监控属性相关的全局变量.对于第(1)类中断服务函数,4.1.1节已经进行了单独处理.第(2)类中断服务函数并不对程序中除其本身以外的任何部分带来副作用,追踪时也可以忽略.事实上,追踪和重放模块需要关注的仅为第(3)类中断服务函数.这类函数与同步代码之间的相互依赖可以通过依赖分析获得,分析结果可以精确到它们各自与同步代码中的哪些函数的哪些语句中所使用或者修改的哪些变量有交集.在对全局量操作进行插桩计数和锁保护时,仅考虑出现在上述交集区域中的操作.譬如,图2中的 AssertTimer.fired 函数与该程序中所有8个中断服务函数之间依赖分析的交集均为空,因此,图4、图5中对该函数内部全局量操作计数的插桩均可被优化.类似的问题也会出现在对检查点的追踪与重放中.加锁的方式以及上述优化同样适用于此类区域.

此外,对于用 nesC 语言编写的 TinyOS 程序,经由 nesC 编译器翻译成 C 程序后,程序中的临界区的首尾分别由_nesc_atomic_start 和_nesc_atomic_end 标识.对上述区域的识别可进一步筛选需要追踪的中断服务函数.

5 实验结果分析

本文选择用 nesC 语言编写的 TinyOS 程序作为实验程序,并选用由 TelosB 传感器结点搭建的传感网作为实验程序的运行环境.每个 TelosB 传感器结点拥有 48KB 的程序存储空间,10KB 的 RAM 空间,和 1MB 的外部存储空间.测试用例以及所监控的属性如表1所示.

表 1 测试用例描述

测试用例名称	测试用例描述	属性描述	规模/Byte
BlinkC	随 TinyOS 2.x 发布	控制3个 LED 的闪烁顺序	2650
TestSerialCO2	监视室内 CO2 数据	基站必须在指定时间内收到数据	18670
EasyCollectionC	使用 TinyOS 2.x 自带的 CTP 协议实现的连续数据的发送和收集程序	消息的发送和接收必须是连续的	24002

我们基于 Open64 编译器,在精确指针分析[7]的基础上计算依赖信息,构建函数依赖图.基于以上信息,建立本文提出的自适应诊断系统原型,并对上述测试用例,(1)评估确定性重放的开销,包括依赖分析和自适应插桩方法对空间开销以及整体的追踪时间开销的控制效果;(2)分析自适应插桩方法对错误根源定位的支持效果.

5.1 依赖分析与空间优化

为了评价依赖分析对本文重放方案的空间优化效果,本节选取检查点设置的2种极限情况:(1)仅设置单一的检查点;(2)将所有可能影响所监控属性的函数入口状态设为检查点.

对于第1种情况,选择最外层循环的迭代入口位置为检查点.表2展示了依赖分析对追踪程序规模的优化效果.其中,Orig+Instr_lib 代表将测试程序与插桩库一起编译生成的可执行码的规模;Baseline 代表不经依赖分析得到的追踪程序规模;Dep 代表基于依赖分析得到的追踪程序规模;Dep_Opt 代表通过依赖分析以及包装函数内联优化得到的追踪程序的规模.上述追踪插桩均仅选择最外层循环的迭代入口位置为检查点.可执行码均由 msp430-gcc 交叉编译器(版本号:3.2.3)编译生成,编译选项:"-mmcu= msp430x1611 -Os -O -mdisable-hwmul -Wall -Wshadow",生成的代码存放在传感器结点的 ROM 空间.

由表2可知,对于 EasyCollectionC 和 TestSerial 这两个程序,如果不借助依赖分析,其生成的追踪程序规模将远超过 TelosB 结点上的程序存储空间的上限.而基于依赖分析的追踪插桩可以将追踪程序规模限定在 48KB 以内,对追踪程序规模的平均优化效果为无优化时的 57.7%.对包装函数的内联优化进一步缩减了插桩规模,可用于设置更多的检查点,提高重放的灵活程度和错误诊断的效率.

表 2 依赖分析对追踪程序规模的优化效果

测试用例	Orig+Instr_lib/Byte	Baseline/Byte	Dep/Byte	Dep_Opt/Byte
BlinkC	10304	18760	11944	11712
EasyCollectionC	25550	73214	42502	39954
TestSerialCO2	30646	80058	47378	44794

表3说明了依赖分析对追踪程序规模优化的主要原因:对追踪范围的优化.如果不利用依赖分析,则需要追踪程序中的所有函数.表3的 Func# 代表

测试用例中函数的数量；Dp_Func＃表示经过依赖分析后，确定需要追踪的函数数量；Dp％是Dp_Func＃和Func＃的百分比，表示经依赖分析后的追踪范围在程序中所占的比例（依赖分析输出的是函数级依赖图，因此，该比例也表示为函数数目之间的百分比）；Opt_Func＃代表经过依赖分析和包装函数内联优化后需要追踪的函数数量；Opt％为Opt_Func＃和Func＃的百分比. 由表3可知，经过依赖分析和包装函数内联优化后，需要追踪的函数数量约为全程序中函数数量的1/4.

表3　依赖分析对追踪范围的优化效果

测试用例	Func＃	Dp_Func＃	Dp％	Opt_Func＃	Opt％
BlinkC	294	45	15.31	31	10.54
EasyCollectionC	1447	614	42.43	401	27.71
TestSerialCO2	1499	612	40.83	396	26.42

依赖分析对追踪程序规模的优化为检查点的选择提供了空间. 表4考虑第(2)种极限情况，选择所有可能影响属性的函数入口为检查点，以检验依赖分析、包装函数优化以及中断分析优化对追踪程序规模的控制效果. 其中，Dep_All列表示基于依赖分析并选择所有函数入口作为检查点得到的追踪程序规模；Dep_All_Opt列在Dep_All的基础上增加了对包装函数的内联优化；Intr_Opt指在Dep_All_Opt的基础上增加对中断分析的优化（详见4.3节）.

表4　三种优化对追踪程序规模的影响

测试用例	原始规模/Byte	Dep_All/Byte	Dep_All_Opt/Byte	Intr_Opt/Byte
BlinkC	2650	13372	11936	11456
EasyCollectionC	18670	66182	54138	47282
TestSerialCO2	24002	70930	58866	52138

数据表明，对测试用例EasyCollectionC，仅借助依赖分析和包装函数内联优化，并不足以将追踪代码规模控制在48 KB的阈值以内，需要通过对中断分析的优化解决代码存储空间问题. 该优化方案可在依赖分析和包装函数内联的基础上，进一步降低追踪程序规模，最大减少12.7％，平均减少9.4％. 表5罗列了上述3种优化对插桩语句数的控制效果. 其中，Total＃表示插桩语句总数，InOut＃代表追踪函数实例的插桩语句的数目，Ckpt＃代表追踪检查点的插桩语句的数目，Cnt＃则是与全局量操作计数器相关的插桩语句的数目. 由表4可知，随以上3种优化的依次进行，插桩语句数量不断减少. 遗憾的是，表4中的测试用例TestSerialCO2即便使用了上述3种优化，其追踪控制程序规模仍然高于阈值. 因此，必须引入自适应迭代规则，根据资源情况，自动选择检查点.

表5　插桩语句数目分析

	Total＃	InOut＃	Ckpt＃	Cnt＃
Dep_All	4352	1373	1273	1706
Dep_All_Opt	3290	948	836	1506
Intr_Opt	1982	946	835	201

本节分析了检查点设置的两种极限情况. 实验表明，无论怎样设置检查点，依赖分析和包装函数优化都能有效地控制追踪代码的规模，部分满足传感器结点上的资源限制. 同时，对中断的优化能进一步降低追踪代码规模，即使在设置所有可能检查点的情况下，也能令EasyCollectionC的追踪控制代码符合空间要求. 但对于TestSerialCO2，即便使用了上述3种优化，追踪代码规模依旧不能满足空间限制. 可见，这种盲目的检查点选择策略并不可取. 为此，我们需要自适应的检查点选择和插桩策略.

5.2　自适应插桩与诊断效率

表6记录了自适应插桩模块根据规则组合I、II分别为EasyCollectionC和TestSerialCO2生成的追踪代码规模. 规则组合I默认只处理主层次以及其上的一级副层次，生成的追踪程序规模都低于阈值；规则组合II优先处理编号最低的主层次及其内部的各级副层次. 对于测试用例TestSerialCO2，与所监控属性相关的程序片段可以划分成6个主层次，但自适应插桩模块完成对前5个主层次的追踪插桩后，剩余的代码空间已不支持对任何完整结构（如：检查点）的追踪. 表中，Adp_size和trace_func＃分别表示生成的追踪代码的规模以及追踪的函数数目. Main_level＃和trace_level＃分别表示程序中与所监控属性相关的部分被划分成的主层次的数量以及规则组合II选择追踪的主层次数量.

表6　自适应插桩效果分析

测试用例	规则组合I效果		规则组合II效果			main_level＃
	Adp_size /Byte	trace_func＃	Adp_size/Byte	trace_func＃	trace_level＃	
EasyCollectionC	33134	88	44014	197	6	6
TestSerialCO2	39530	93	47966	176	5	6

检查点的设置能够有效地提高错误诊断的效率.以 EasyCollectionC 为例,消息发送结点从程序开始执行到发送方属性被违反,记录的轨迹日志将近 28KB(不包含检查点).如果没有设置检查点,则用户调试时只能从程序入口位置开始重放,但他们需要关注的实际上只有从触发属性违反的错误根源语句执行实例到属性被违反位置之前的追踪日志.表 7 记录了 EasyCollectionC 程序执行过程中 5 次从错误根源语句到属性违反位置之间的轨迹日志规模,其平均大小仅为以程序入口点为起始的日志的 8.36%.一旦设置了检查点,用户就可以灵活地选择需要重放的程序区段,从而避免不必要的重放和诊断开销.

表 7 日志规模【错误根源→属性违反】

日志规模/Byte					百分比/%
1	2	3	4	5	
2282	2346	2732	2330	2332	8.36

实验表明,自适应规则能够依据可用资源,选择适当的检查点数量和插入位置,生成规模符合空间要求的追踪控制程序.检查点的设置使得用户在基于重放对错误进行诊断过程中,既能自主地选择轨迹片段,又可以避免对执行轨迹中位于错误根源之前的部分的重放,降低诊断的时间开销.

5.3 追踪时间开销

本节分析了追踪对程序执行时间的影响程度以及关键影响因素.追踪程序的额外执行时间开销源于 3 个部分:(1)插桩库函数的执行;(2)轨迹数据从数据缓冲区写入外部存储空间;(3)向基站传输轨迹记录.

本文分别测量了 3 个测试用例由追踪带来的额外开销.其中,对于 BlinkC,测量对象是从程序开始执行到断言违反位置之间的追踪开销;对于 TestSerialCO2,测量对象是每个抽样周期内的追踪开销;而对于测试用例 EasyCollectionC,我们测量了一个发送方结点从开始发送消息,到消息发送成功之间的轨迹追踪对执行时间的影响.具体测量方法如下:首先,在测量的开始位置插入一条将传感器结点第 3 针脚的电压置为高电压的指令,在测量的结束位置插入一条将传感器结点第 3 针脚的电压置为低电压的指令;然后使用 NI LabVIEW 软件观测高低电压变化的时间间隔,即可获得从测量开始位置到结束位置的执行时间.测量过程中,默认将 RAM 中 1KB 的空间作为存储轨迹数据的缓冲区(以下简称日志缓冲区),仅当记录的轨迹数据规模到达日志缓冲区上限时,将数据写入外部存储空间,并清空日志缓冲区.

表 8 在仅设置单一检查点的前提下,测量 3 个测试对象的额外追踪开销.其中,trace_size 代表轨迹日志规模;orig_exe_time 和 tracing_exe_time 分别表示插入追踪代码前后的执行时间;overhead% 代表追踪引入的额外时间开销占原始程序执行时间的比例.由于 TestSerialCO2 的每两次抽样之间有较长的空闲周期,这些空闲周期掩盖了追踪的额外开销,故而可以忽略轨迹追踪对该程序执行时间的影响.而对于测试用例 BlinkC 和 EasyCollectionC,追踪带来的额外执行开销分别仅占原始执行时间的 1.01%和 13.32%.此外,由于针对这两个程序测量得到的追踪日志规模均未超过日志缓冲区的规模上限,故表 8 所测量的数据并不包含将轨迹数据写入外部存储空间的开销.

表 8 追踪的时间开销-1

测试用例	trace_size/Byte	orig_exec_time/μs	tracing_exe_time/μs	overhead/%
BlinkC	56	6930	7000	1.01
TestSerialCO2	1496			
EasyCollectionC	838	14790	16760	13.32

表 9 在自适应插桩的前提下,重新统计了测试用例 EasyCollectionC 从开始发送消息,到消息发送成功之间的追踪开销,以测量将轨迹数据从日志缓冲区写入外部存储空间对执行时间的影响,实验中将日志缓冲区(即表 9 中 trace_buffer 列)分别设为 1KB 和 3KB.由表 9 可知,当日志缓冲区大小为 1KB,将发生两次对外部存储空间的写操作,而当日志缓冲区大小为 3KB 时,足以存放 2404 字节的轨迹数据,故不必将其写入外部存储空间.由表 9 可知,将数据写入外部存储空间的频率是影响追踪程序执行时间的关键,频率越高,开销越大;而提高预设的日志缓冲区的规模是减少追踪开销的方法之一.但另一方面,提高日志缓冲区的规模,势必会降低存储静态数据和运行栈的空间,甚至可能出现运行栈与日志缓冲区或静态数据区相重叠的问题.为预防上述问题,可以在链接时调整 RAM 的布局和

重定位的目标地址[8]，将日志缓冲区和静态数据区依次放置在 RAM 的最高端，追踪过程中，一旦检测到运行栈向下溢出或即将溢出的情况，则提示用户调整日志缓冲区的大小.

表 9　追踪的时间开销-2

测试用例	trace_size/Byte	trace_buffer/Byte	orig_exe_time/μs	tracing_exe_time/μs	overhead/%
EasyColectionC	2404	3096	14790	17661	19.41
		1024		63147	326.96

本节实验数据表明，ADA 机制通过依赖分析和自适应迭代技术，不仅能够根据系统资源的可用情况调整追踪的空间需求，使基于确定性重放的调试方法能够沿用于无线传感器网络程序的错误诊断中，还有效地提高了错误诊断的效率.

6　相关工作

面向无线传感器网络应用的错误诊断技术可以分为 3 类：基于模拟器的调试技术、交互式调试技术以及通过运行时监控辅助的错误诊断技术.

基于模拟器的调试平台以Emstar[9]、TOSSIM[10]、S^2DB[11]等为代表，此类方法的优势在于可以在系统部署之前对大规模网络环境下程序执行情况进行模拟，但缺点是无法模拟复杂多变的物理环境，故而并不能完全替代实际设备上的调试.

交互式调试技术则通过远程控制，使传感器结点进入调试状态，而后，采用由用户设置断点、观察点或进行单步调试的方法发掘错误源. 此类技术以 Clairvoyant[12] 和 Marionette[13] 为代表. 前者是一个面向无线、嵌入式网络的源代码级调试环境，提供诸如 step、break、watch、traceback 等常用调试命令；后者实现了一组交互式开发、调试工具，提供丰富的接口，允许用户使用 PC 上的编程、调试环境，如 Java GUI、. NET，访问无线嵌入式设备上运行的程序的函数调用、变量读写等操作. 此类技术的优势是贴近传统调试，适用于用户对错误类型以及需要关注的位置有明确认识的情况，但缺陷在于，当用户对错误类型及相关位置毫无头绪时，此类调试方法缺乏效率，且由于并行程序执行过程和结果的不确定性，实际运行时发生的错误可能在调试过程中不再重现.

通过运行时监控辅助错误诊断的方法能够提供实际设备上与错误相关的信息，但缺点在于：需要由用户设定监控的位置和记录的日志内容，且无法确保记录的信息与错误相关或足以帮助用户定位到错误根源. 其中，Sympathy[14] 是由传感器结点定时向数据汇聚结点主动发送选定的执行信息，当错误出现时，根据基站收集到的执行信息，使用基于决策树的方法诊断错误源. PAD[15] 是一个轻量级的无线传感器网络诊断工具，采用分组标记算法收集并维护一个概率推论模型，推演导致程序不正常行为的原因. DT[16] 设计了一种 traceSQL 语言来描述调试行为，这些称作“tracepoint”的调试行为可以在程序运行时热插入或热删除. PD2[17] 关注导致性能不佳的原因，比如重要数据丢失、数据流延迟等. Interface contracts for TinyOS[18] 通过为函数调用插入前置、后置断言的方法实现静态检测. PDA[19] 则为用户提供了一种形式化地描述结点状态的简单语言，并采用主动抓包(packet sniffing)方法收集信息，判断用户书写的断言是否得到满足. T-check[20] 使用基于 TOSSIM 模拟器的静态空间搜索方法检测程序错误，但由于 TOSSIM 模拟器不提供对中断的模拟，该工具无法检测以下几类错误：时序错误、并发错误以及底层设备中存在的错误.

确定性重放技术是并行程序调试的基础. 在过去 20 年间，如何以尽可能小的时空开销来记录并重现并行程序的执行轨迹，一直是研究人员和工业界关注的热点，一些国内外知名的公司也分别设计了各自的确定性重放框架，如：微软基于 Nirvana 模拟器以及 iDNA 轨迹读写工具开发的确定性重放框架[5]、英特尔以二进制插装库 PIN 为基础设计并实现的重放工具 Pinplay[6]. 迄今为止，关于确定性重放领域的研究大致可分为两类：纯软件插桩方法和借助硬件辅助的方法. 前者具备跨机器和跨平台重放的优势，但时空开销高昂；后者强调通过少量的额外硬件支持，可以显著地降低追踪和重放的时空开销，但不便于推广. 受到有限的存储和通信资源以及传感器结点对硬件设计及功耗的限制，上述方法均不适用于无线传感器网络程序的调试.

7　结　论

无线传感器网络是物联网得以实现的重要基

础.然而,无线传感器网络的大规模部署和管理迄今仍面临诸多挑战,其中重要的阻碍因素是无线传感器网络上的程序错误难以诊断.针对上述问题,本文提出一种以依赖分析为基础,采用自适应的源码级追踪和重放技术辅助无线传感器网络程序错误诊断的机制 ADA,并在 open64 编译器上实现了一个面向 TinyOS 程序的原型.实验数据表明,该机制能够有效地减轻程序员的错误诊断负担,提高无线传感器网络中错误诊断的效率.本文的后续工作将以自动化调试为重点,进一步辅助程序员高效定位错误根源.

参 考 文 献

[1] Kennedy K, Allen J R. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2001

[2] Weiser M. Program slicing//Proceedings of the 5th International Conference on Software Engineering (ICSE'81). San Diego, California, USA, 1981: 439-449

[3] Hotwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs//Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI'88). Atalanta, Georgia, 1988: 35-46

[4] Altekar G, Stoica I. ODR: Output-deterministic replay for multicore debugging//Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09). Bigsky, Montana, USA, 2009: 193-206

[5] Bhansali S, Chen W-K, Jong S D et al. Framework for instruction-level tracing and analysis of program executions//Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06). Ottawa, Ontario, Canada, 2006: 154-163

[6] Patil H, Pereira C, Stallcup M et al. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs//Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10). Toronto, Ontario, Canada, 2010: 2-11

[7] Yu H, Xue J, Huo W et al. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code//Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10). Toronto, Ontario, Canada, 2010: 218-229

[8] Gu Xiao-Ming, Huo Wei, Gui Jian et al. A new approach to detect the overlap between runtime stack and static data sections. Computer Engineering and Applications, 2006, 42(20): 4(in Chinese)
(谷晓铭,霍玮,桂剑等. 一种检测运行栈与静态数据区重叠的新方法. 计算机工程与应用, 2006, 42(20): 4)

[9] Girod L, Ramanathan N, Elson J et al. Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. ACM Transactions on Sensor Networks (TOSN), 2007, 3(3)

[10] Levis P, Lee N, Welsh M et al. TOSSIM: Accurate and scalable simulation of entire TinyOS applications//Proceedings of the 1st International Conference on Embedded Networked Sensor Systems. Los Angeles, California, 2003: 126-137

[11] Wen Y, Wolski R, Gurun S. S2DB: A novel simulation-based debugger for sensor network applications//Proceedings of the 6th ACM & IEEE International Conference on Embedded software (EMSOFT'06). Seoul, Korea, 2006: 102-111

[12] Seawright A, Brewer F. Clairvoyant: A synthesis system for production-based specification//Micheli G D, Ernst R, Wolf W. Readings in Hardware/Software Co-Design. Norwell, MA, USA: Kluwer Academic Publishers, 2001: 375-388

[13] Whitehouse K, Tolle G, Taneja J et al. Marionette: Using RPC for interactive development and debugging of wireless embedded networks//Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN'06). Nashville, Tennessee, USA, 2006: 416-423

[14] Ramanathan N, Chang K, Kapur R et al. Sympathy for the sensor network debugger//Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (Sensys'05). San Diego, California, USA, 2005: 255-267

[15] Liu K, Li M, Liu Y et al. Passive diagnosis for wireless sensor networks//Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (Sensys'08). Raleigh, NC, USA, 2008: 113-126

[16] Cao Q, Abdelzaher T, Stankovic J et al. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks//Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (Sensys'08). Raleigh, NC, USA, 2008: 85-98

[17] Chen Z, Shin K G. Post-deployment performance debugging in wireless sensor networks//Proceedings of the 2009 30th IEEE Real-Time Systems Symposium (RTSS'09). Washington D. C., USA, 2009: 313-322

[18] Archer W, Levis P, Regehr J. Interface contracts for TinyOS//Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN'07). Cambridge, Massachusetts, USA, 2007: 158-165

[19] Romer K, Ma J. PDA: Passive distributed assertions for sensor networks//Proceedings of the 2009 International Conference on Information Processing in Sensor Networks (IPSN'09). San Francisco, CA, USA, 2009: 337-348

[20] Li P, Regehr J. T-check: Bug finding for sensor networks//Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'10). Stockholm, Sweden, 2010: 174-185

附录 1.　插桩库函数.

附表 1 介绍了本文实现并使用的所有插桩库函数以及它们的插桩位置和功能.其中,第 5 列里带“√”标记的函数为追踪过程中可能使用到的插桩库函数,而第 6 列里带相同标记的函数则为重放过程中可能使用到的插桩库函数.

附表 1　插桩库函数

函数类型	函数名称	插桩位置	功能	追踪	重放
函数实例追踪函数	WriteInOut	可能影响所监控属性的函数的入口/出口	记录当前执行的函数(包括中断服务函数)实例	√	
检查点追踪函数 1	WriteStartPoint	可能影响所监控属性的函数的入口/出口	追踪当前函数(包括中断服务函数)当前执行实例中与监控属性相关且向前暴露的变量	√	
检查点追踪函数 2	WriteBuffer			√	
计数器复位函数	ResetMemCnt	可能影响所监控属性的函数入口	全局量操作计数复位	√	√
计数器递增函数	IncMemCnt	可能影响所监控属性的全局量使用/修改位置	全局量操作计数加 1	√	
函数实例判定函数	CheckFunc	可能影响所监控属性的函数的入口/出口	确定当前执行的是该函数的第几个实例		√
检查点载入函数	ReadBuffer	可能影响所监控属性的函数的入口	读取追踪日志中的检查点状态		√
中断响应判定函数	CheckIntr	可能影响所监控属性的全局量使用/修改位置	判断当前位置是否允许响应中断,以及应当响应哪个中断		√
临界区起始函数	AtomicStart	需要屏蔽中断响应的区域的起始位置	屏蔽该区域内部对中断的响应	√	
临界区结束函数	AtomicEnd			√	

附录 2.　确定性重放证明.

本文进行确定性重放的目的是辅助诊断程序错误,因此,本文在设计确定性重放策略时,并不苛求重放轨迹与程序的实际错误执行轨迹完全一致,仅需确保从错误源所在的语句 s 到属性断言 Inv 违反位置之间的重放轨迹与实际错误执行时相同.因此,对本文的重放策略的确定性证明可以表述为引理 1 的形式.

引理 1.　假定 TinyOS 程序 P 的某次执行过程中,错误语句 s 导致了属性断言 Inv 被违反;基于本文的确定性重放策略,必定能够复现一条由错误语句 s 导致属性断言 Inv 被违反的执行路径.

由于错误语句 s 总是通过定值-引用链将影响传播到属性断言 Inv 的违反位置,如果程序 P 从 s 到 Inv 的执行轨迹上没有响应任何中断,而追踪日志又记录了完整的任务调度顺序和外部输入,则重放轨迹必定与实际的错误执行轨迹保持一致.同理,如果从 s 到 Inv 的执行轨迹上响应了中断,但这些中断所对应中断处理函数并不影响从 s 到 Inv 的定值-引用链,则仅记录任务调度顺序和外部输入,也可以确保重放轨迹与实际的错误执行轨迹一致.因此,对引理 1 的证明可以等价成对引理 2 的证明.

引理 2.　假定 TinyOS 程序 P 的某次执行过程中,错误语句 s 导致了属性断言 Inv 被违反;基于本文的中断重放策略,总能复现由错误语句 s 导致属性断言 Inv 被违反的所有定值-引用关系.

证明.

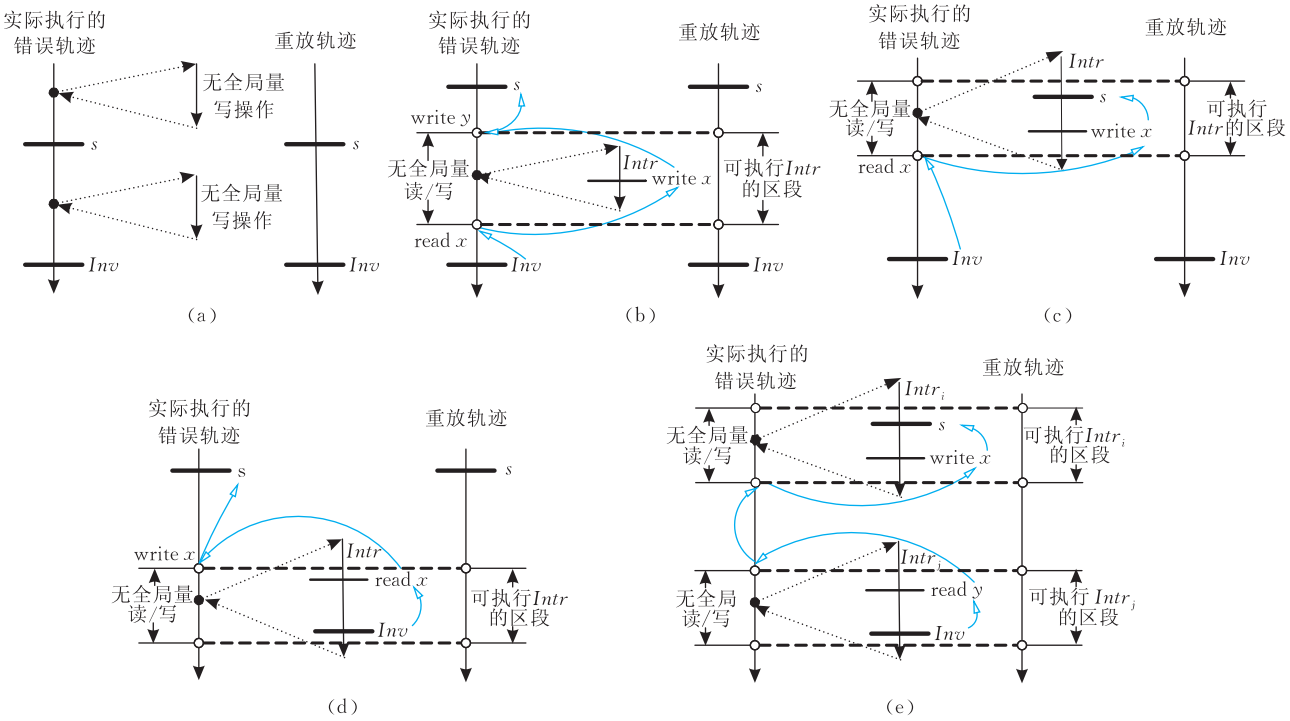
首先,假定 s 和 Inv 都不在中断处理函数中.基于上述假设,可能出现两种情况.(1)程序 P 执行过程中,从 s 到 Inv 的执行轨迹上没有响应中断,或者其间所响应的中断对应的中断处理函数(记作 $Intr$)中没有对全局变量的使用/修改,如附图 1(a)所示;(2)程序 P 执行过程中,从 s 到 Inv 的执行轨迹上有中断响应,并且其对应的中断处理函数可能影响 s 到 Inv 之间的定值-引用链(譬如,Inv 传递依赖于全局变量 x,全局变量 y 的定值传递依赖于 s,中断处理函数 $Intr$ 更新了 x,该更新又直接依赖于 y 的定值),如附图 1(b)所示.针对情况(1),重放 s 到 Inv 的执行轨迹时,无论是否执行 $Intr$ 以及在何处执行,均不会改变从 s 到 Inv 的定值-引用链;而对于情况(2),本文在追踪过程中,将根据 y 的定值位置和 x 的引用位置,把执行轨迹划分成至少 3 个区段;重放时,将 $Intr$ 的执行范围限定在(b)中 y 的定值操作与 x 的引用操作之间的区段内,以保持原始执行过程中的定值-引用链($s \rightarrow^*$ 写 $y \rightarrow$ 写 $x \rightarrow$ 读 $x \rightarrow^* Inv$,其中 $\rightarrow$ 代表直接依赖,$\rightarrow^*$ 代表传递依赖).

其次,假定 s 和 Inv 中至少有一方在中断处理函数中.如果 s 在中断处理函数(记作 $Intr$)中,且 $Intr$ 影响 s 到 Inv 的定值-引用链,则 $Intr$ 中必然存在对某个全局变量 x 的定值,该定值传递依赖于 s,且在 $Intr$ 返回后被读取,并最终导致了 Inv 违反,如图 1(c)所示.对于上述情况,本文在追踪过程中,将根据(c)中所示的 x 引用位置,把执行轨迹划分成至少两个区段,重放时,将 $Intr$ 的执行范围限制在该位置的前一个区段内,以保持原始执行过程中的定值-引用链($s \rightarrow^*$ 写 $x \rightarrow$ 读 $x \rightarrow^* Inv$).如果 Inv 在中断处理函数中,则必然存在某个全局变量 x,其定值依赖于 s,$Intr$ 中读取该定值,并通过传递依赖最终导致 Inv 违反,如附图 1(d)所示.对于上述情况,本文在追踪过程中,将根据(d)所示的 x 定值位置,把执行轨迹划分成至少两个区段,重放时,将 $Intr$ 的执行范围限制在该位置的后一个区段内,以保持原始执行过程中的定值-引用链($s \rightarrow^*$ 写 $x \rightarrow$ 读 $x \rightarrow^* Inv$).

最后,假定 s 和 Inv 分别隶属于两个不同的中断处理函数执行实例 $Intr_i$ 和 $Intr_j$(可以是同一中断处理函数的不同

实例)中,则必然存在全局变量 x、y,x 在 $Intr_i$ 中被定值,且传递依赖于 s,y 在 $Intr_j$ 中被引用,并最终引发 Inv 违反,而 y 的引用传递依赖于 x 的定值,或者 x 和 y 是同一个变量,如附图 1(e)所示. 如果追踪过程中,上述两个中断处理函数实例之间的执行轨迹上没有全局量操作(即 x 和 y 是同一个变量),则重放时,只需顺序执行 $Intr_i$ 和 $Intr_j$,即可维持原始执行过程中的定值-引用链($s\rightarrow^{*}$写 $x\rightarrow$读 $y\rightarrow^{*}Inv$);如果其间存在一组全局量操作(记作 $S_1,S_2,\cdots,S_n$,这些全局量操作不会更新 x,否则,图(e)所示的 x 定值会被注销,无法到达 y 的引用位置),则执行轨迹将被划分成若干区段,重放时,将 $Intr_i$ 的执行限定在 S_1 的前一个区段内,而将 $Intr_j$ 的执行限定在 S_n 的后一个区段内,即可维持原始执行过程中的定值-引用链($s\rightarrow^{*}$写 $x\rightarrow^{*}$读 $y\rightarrow^{*}Inv$).

综上,基于本文的确定性重放策略,总能复现错误语句 s 导致属性断言 Inv 被违反的所有定值-引用关系. 证毕.



附图 1 确定性重放证明示例

LI Feng, born in 1985, Ph. D. candidate. Her research interests include program analysis and error diagnosis.

HUO Wei, born in 1981, assistant professor. His research interests include program analysis and error detection.

FENG Xiao-Bing, born in 1969, professor, Ph. D. supervisor. His research interests include advance compiling technology and related tools.

Background

Methods for error diagnosis and debugging in Wireless Sensor Network can be classified into three categories: simulation, interactive debugging and run-time logging. Simulation offers considerable flexibility but often takes significantly more time than direct execution. However, simulated cases may not be sufficient extensive to catch errors that may happen during the real operation. Interactive debugging allows programmers to interact with sensor nodes by sending commands, but the step-by-step execution can be quite slow and tedious, with no guarantee that the anticipated error will surface in the debugging mode. In other circumstances, especially when the number of motes to be debugged simultaneously is large, it seems much more convenient to have execution traces ready when an error is detected. Run-time logging has gained increased attention recently. The critical questions encountered when adopting this approach include what kind

of errors should be monitored and how to analyze the logged information to find the error cause.

In this paper, we present an adaptive tracing and replay method based on dependence information to assist WSN programming and debugging. The prototype system is implemented on top of Open64 compiler which provides the WHIRL representation and precise SSA form for the consequently analysis, e. g. program slicing and instrumentation. In addition, the whirl2c module of Open64 compiler allows source to source translation. Our experiments show that this approach has made it possible to instrument several test programs on WSN under the stringent program memory constraint and find injected errors. Although our current experiments are performed on TinyOS-based applications, the proposed methodology and tool can be extended to a broader range of embedded systems.

This research is supported by National Basic Research Program (973 Program) of China (No. 2011CB302504), Chinese National Science and Technology Major Project (2009ZX01036-001-002), and Foundation for Innovative Research Groups of the National Natural Science Foundation of China (No. 60921002). These projects aim at improving software reliability, including the detection of software defects, program verification and debugging based on compiling technology. The work of this paper is to reduce the space and time usage of debugging in Wireless Sensor Networks, and improve the efficiency of fault localization.