Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2013-CJ-003

2013-CJ-003

面向维护的实时软件时间变化敏感点检测

汤恩义，李宣东

计算机学报 2013 年 第 36 卷

面向维护的实时软件时间变化敏感点检测

汤恩义[1),3)] 李宣东[1),2)]

[1)](南京大学软件新技术国家重点实验室 南京 210093)

[2)](南京大学计算机科学与技术系 南京 210093)

[3)](南京大学软件学院 南京 210093)

摘 要 正确的时间属性和行为对于实时软件来说非常关键,然而这却很难得到完全的保障.在实际工业中,实时性错误不仅会在软件的设计开发阶段被引入,在软件的维护阶段,随着软件的演化也同样会引入这种错误.当软件维护人员对系统不够熟悉时,维护阶段引入这些错误的可能性会更大.目前,还没有研究结果可以在软件修改发生之前,通过分析系统的时间关系信息来帮助指导软件维护人员减少或者避免引入时间相关的错误.在这样的背景下,文中提出了一种实际可用的分析途径来解决这一问题,称之为时间变化影响分析.这一解决途径在软件维护人员做出修改之前就可以分析软件内在的时间关系信息,从而帮助维护人员在软件的实际演化前预测可能带来的时间影响.在具体的操作上,可以通过告知软件维护人员程序代码中的某些位置点可能会对整个程序任务的执行时间产生很大的影响,来提醒他们修改这些位置点的代码时应当特别小心.由于这些点对程序任务的执行时间敏感,我们称之为时间变化敏感点.文中通过一种基于扰动的测试实现来检测程序中的时间变化敏感点,通过在程序中插桩不同的代码时延,达到对执行时间的扰动作用,对这些扰动作用下的程序执行时间数据进行统计,得到程序中各个位置点对时间影响的定量度量.对于实际使用的大规模程序,该方法通过集成静态程序分析技术提高吞吐量和减小漏报率.文中对分析途径做了原型实现,并在 Mälardalen WCET 基准用例集和开源项目 Free Lossless Audio Codec(FLAC)上做了实例评估.评估结果展示了该方法的可行性和有效性.

关键词 软件维护;实时软件;时间变化影响分析;基于扰动的测试;时间变化敏感点

中图法分类号 TP311 **DOI 号** 10.3724/SP.J.1016.2013.02455

Time-Leverage Points Detection for Software Maintenance

TANG En-Yi[1),3)] LI Xuan-Dong[1),2)]

[1)](*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093)

[2)](*Department of Computer Science and Technology, Nanjing University, Nanjing* 210093)

[3)](*Software Institute, Nanjing University, Nanjing* 210093)

Abstract Correct time behavior is an important aspect for time sensitive software. However, it is difficult to get right. Real-time failures can be introduced not just during the development but also maintenance. So maintainers without time information of the software tend to have more chances to introduce unintended time behaviors. Few practical works have been done on collecting the time-related information before the software evolves, which we call time change impact analysis. In this paper, we propose a novel, practical framework to get high-level information about the impact of time before new code is added to the time-sensitive project. Our main insight is that by reminding maintainers some places (or points) in the source code, which largely affect the execution time, maintainers can be more cautious when updating such places. Because these points have a leverage effect that multiplies the task time, we call them time-leverage points. Our

收稿日期:2012-01-18;最终修改稿收到日期:2012-11-23.本课题得到国家自然科学基金(91118007)、国家"八六三"高技术研究发展计划项目基金(2012AA011205)及计算机软件新技术国家重点实验室开放课题项目基金(ZZKT2013B09)资助.汤恩义,男,1982 年生,博士,助理研究员,主要研究方向为软件工程、程序分析与软件测试.E-mail:eytang@seg.nju.edu.cn.李宣东,男,1963 年生,博士,教授,博士生导师,主要研究领域为软件工程、形式化方法、软件建模与分析、软件测试与验证.

framework integrates a disturbing based testing technique to find these points for maintainers. The technique disturbs the evaluation time by instrumenting different codes in software. When summarizing the time changes, we can measure the time impact of points. For a real project, we also integrate several static program analysis techniques to reduce the false negatives and improve the scalability of our framework. We implement our method as a prototype and evaluate the method on Mälardalen WCET Benchmark suite and Free Lossless Audio Codec (FLAC) project. The result shows that our framework successfully detected the time-leverage points in software.

Keywords software maintenance; real-time software; time impact analysis; disturbing based testing; time-leverage point

1 引 言

随着实时系统的广泛应用,时间属性的正确性在软件中的重要地位日益凸显.特别是对于使用在诸如交通工具、医疗设备等关键领域的系统软件,时间属性的错误将会导致严重的后果.然而,软件功能的复杂性导致了其时间属性的正确性很难得到完全的保障.不仅如此,我们发现与时间相关的软件错误除了软件的设计和开发阶段以外,在软件的维护阶段也同样会引入.对于如何在软件的开发阶段避免时间相关的错误,现有的研究已经有所积累,但对于如何帮助软件维护者在软件维护阶段避免时间相关错误的引入,目前还未见相关文献.本文的研究就是针对这一问题而来的.

软件修改影响分析(Software Change Impact Analysis)是一种帮助软件维护人员预测软件演化过程中可能带来潜在后果的重要方法[1-4].已有的软件修改影响分析技术关注于分析程序代码在功能修改上的依赖关系而忽视了其时间变化属性.在本文中,我们提出一种新的软件修改影响分析——时间变化影响分析.这一分析在软件修改发生前,就可用于预测软件演化带来的时间变化影响,从而帮助软件维护人员避免在实际软件的修改过程中引入错误的时间变化行为.

我们所提出的时间变化影响分析基于如下的观察事实:在程序中的各个不同位置点的修改会对程序任务执行时间的变化产生不同的影响,其中有一部分位置点即使因修改发生微小的局部时延,也会造成整体程序任务的执行时间发生很大的改变.由于这样的位置点对程序修改后的执行时间很敏感,我们称这样的点为时间变化敏感点.举例来说,如图 1 所示,任务 A、B 为一个中断驱动的工业控制系统中两个不同的中断处理,用来管理不同的设备,由于任务 B 管理着更加重要的设备,其优先级高于 A.由于在任务 A 的末尾会处理一段比较关键的设备通信,该系统的开发者为了保证其不被打断而预留了较大的时间冗余,其效果如图 1(a)所示.可是在软件的维护过程中,由于要加入新的功能,任务 A 发生了修改而加入了一小段代码片段.虽然这段被加入的代码片段的执行时间很短,但由于涉及到任务 A 中的一个特殊的位置点(即我们所称的时间变化敏感点),任务 A 的执行时间被显著地增长.从而如图 1(b)所示造成 A 末尾的设备通信会被打断而使整个系统出现问题.本文希望给出一种时间变化敏感点的检测方法,能够在软件修改发生之前就帮助指导软件维护者在维护过程中避免引入这样的实时性错误.软件维护人员需要非常谨慎地在时间变化敏感点附近做修改操作来减少时间错误的引入,当确有必要修改这些时间变化敏感点时,本文方法

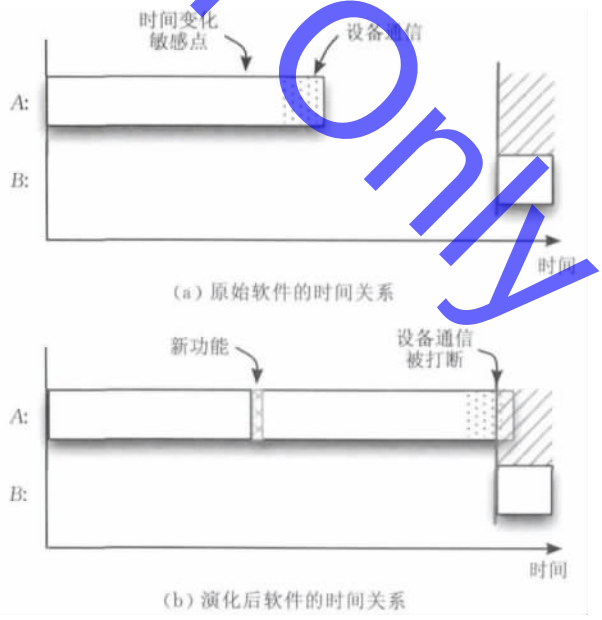

图 1 一个工业软件在维护阶段由于修改了敏感点引起时间错误的例子

也可以提供造成这些时间变化敏感现象发生的测试用例. 维护人员可以利用我们分析所筛选的测试用例做进一步的回归测试，或者将我们所分析到的这些点的详细性质信息输入设计模型做进一步的验证，从而使软件的时间正确性在软件演化过程中仍然得到保持.

从具体的实现途径来看，我们采用一种基于扰动的测试方法来检测程序中的时间变化敏感点. 通过在程序中插桩不同量级的增量时延代码，可以对程序任务执行时间的变化起到一个扰动的效果. 而对于某一个程序点的执行时间扰动做统计综合，就可以定量地度量出这个点在程序任务发生变化时对执行时间的影响. 依据这一度量是否发生统计异常，或者是否超出用户指定的阈值，可以判定这一点是否为时间变化敏感点. 我们的时间变化敏感点检测方法假设软件维护前后的运行环境不变. 这一假设在多数情况下是合理的. 该方法也可以利用软件进入维护阶段之前已经积累的测试用例，因此，从整体上说，该方法并不需要很大的开销. 另外，这一分析还可以得到产生异常时间变化敏感点的测试用例作为副产品，从而供软件维护人员在修改程序之后进行回归测试. 由于这一方法是测试方法，从理论上说不存在误报.

在实际应用中，当所分析的软件规模较大时，我们集成了多项静态程序分析技术来提高检测方法的运行效率. 这些技术通过静态扫描的方法，快速地发现程序中相对于路径的高频热点，并以此来作为时间敏感疑似点. 而后，基于扰动的测试方法仅需要对这些疑似点进行处理，而不需要针对程序里的所有位置点进行测试，从而提高了本文方法的应用范围. 另一方面，由于这些静态程序分析技术不依赖于具体的软件测试用例，对减小本文方法的整体漏报率也能起到一定的作用.

本文第 2 节详细的描述本文的分析途径及其具体方法和技术；在第 3 节里，给出一个本文方法的原型工具实现，并在多个实例上进行实验评估；第 4 节对已有的相关工作进行比较，最后在第 5 节，对本文的工作做出总结，并给出一些将来研究会涉及到的主题.

2　时间变化敏感点检测途径

本节提出时间变化影响分析途径的目标在于寻找程序中对时间变化敏感的位置，这些位置局部微小的时间变化会导致程序中整个任务时间发生很大变化. 首先引入相关定义.

2.1　时间变化敏感点与相关度量

从软件维护的角度，我们的工作基于如下假设：程序中的修改可以被分解成多个的单点修改. 这样的单点修改仅在软件的某一位置做独立的插入操作，我们的度量和评估都是基于单点修改而活动的. 在这里，首先定义程序位置点，描述程序修改的具体位置.

定义 1(程序位置点).　给定程序$\mathcal{P}$，其任一条可执行语句的结束位置是$\mathcal{P}$的一个程序位置点.

从实时系统的角度，我们仅关注在程序位置点插入语句段所导致程序任务完成时间的变化，并且假设所插入的语句段对程序的控制流没有任何影响.

定义 2(增量时延修改).　给定程序$\mathcal{P}$，给定$\mathcal{P}$的程序位置点 p_1，若在 p_1 处插入一段语句且该段语句对$\mathcal{P}$控制流不产生任何影响，则称程序$\mathcal{P}$在 p_1 处发生了增量时延修改，被插入语句段的执行时间称为增量时延. 更进一步地，当修改 c 既不直接涉及到软件的控制流程跳转条件(包括分支条件、循环条件以及其它类型的跳转条件等)，并且程序中的控制流程跳转条件也不依赖于修改 c 所改变的数据流时，我们称修改 c 为独立于软件控制流程条件的时延修改，或简称时延修改.

本文主要关注软件维护时增量时延修改所产生的增量时延对程序中给定任务的完成时间所造成的影响，为了度量和评估这些影响需要引入以下定义.

定义 3(扰动点和观察点).　给定程序$\mathcal{P}$，且在程序位置点 p_1 发生了增量时延修改；给定$\mathcal{P}$中的一个程序任务 R，且其在程序位置点 p_2 结束. 若我们关注该增量时延修改对程序任务 R 的完成时间所产生的影响，则称 p_1 为扰动点，p_2 为观察点.

定义 4(时间杠杆率).　给定程序$\mathcal{P}$和其中发生的一个增量时延修改，设 p_1 和 p_2 分别为相应的扰动点和观察点，增量时延为 t. 设对给定的一次程序执行，发生该增量时延修改前，程序到达 p_2 的执行时间为 T；发生该增量时延修改后，程序到达 p_2 的执行时间为 T'. 则相对于该程序执行和该增量时延修改，p_1 的时间杠杆率 $r(p_1,p_2,t)$ 定义为

$$r(p_1,p_2,t)=(T'-T)/t \tag{1}$$

对给定程序中的某个增量时延修改和某次程序执行，时间杠杆率是该增量时延修改对给定程序任务完成时间产生影响的一种度量. 若该度量值超过

了某个阈值(从而破坏了给定程序应该满足的时间约束条件),我们就可以断定时间变化敏感点的存在.

定义 5(时间变化敏感点). 给定程序$\mathcal{P}$和其中发生的一个增量时延修改(令增量时延为 t),设 p_1 和 p_2分别为相应的扰动点和观察点. 若存在$\mathcal{P}$的一次执行,使得 p_1 的时间杠杆率 $r(p_1,p_2,t)$大于给定的阈值,则称 p_1 为相对于 p_2的时间变化敏感点.

在实际应用中,用于判断时间变化敏感点存在的阈值是根据给定程序应该满足的时间约束条件设定的. 时间变化敏感点的定义基于具体的程序执行,因此直觉上测试是检测时间变化敏感点最直接的途径.

2.2 时间变化敏感点检测过程

基于上述定义,我们提出一种基于扰动测试方法的时间变化敏感点检测途径,它通过在程序中的各个位置点插入时延进行时间变化扰动,并进一步在程序中各个任务结束的位置点或者整个程序结束的位置点获取程序执行时间的变化信息,以判断时间变化敏感点的存在. 图 2 给出了该途径的整体流程,它共分为 5 步.

第 1 步是位置点标定预处理,在这一步中,我们针对系统的原始程序标定其扰动点和观察点. 并将这些标定的位置点作为时间敏感疑似点提供给后续步骤做进一步测试;第 2 步确定检测过程的测试用例,当待检测程序经过功能测试而保留了测试用例时,我们直接使用这些保留的测试用例;另外,我们会生成一些补充的测试用例;在以上两步准备工作的基础上,第 3 步到第 5 步完成时间变化敏感点的检测. 第 3 步对程序的扰动点插桩具体的时延代码,同时对相应的观察点插桩时间行为数据收集代码,原始程序被转换成经过插桩的程序;第 4 步使用第 2 步产生的测试用例来驱动执行经过插桩的程序,记录程序执行的时间行为数据;第 5 步分析上一步输出的时间行为数据,并依据它判断并输出时间变化敏感点. 下文的算法 1~5 列举了每一个检测步骤的过程细节,而检测方法中更进一步的理论探讨与结论参见文后附录.

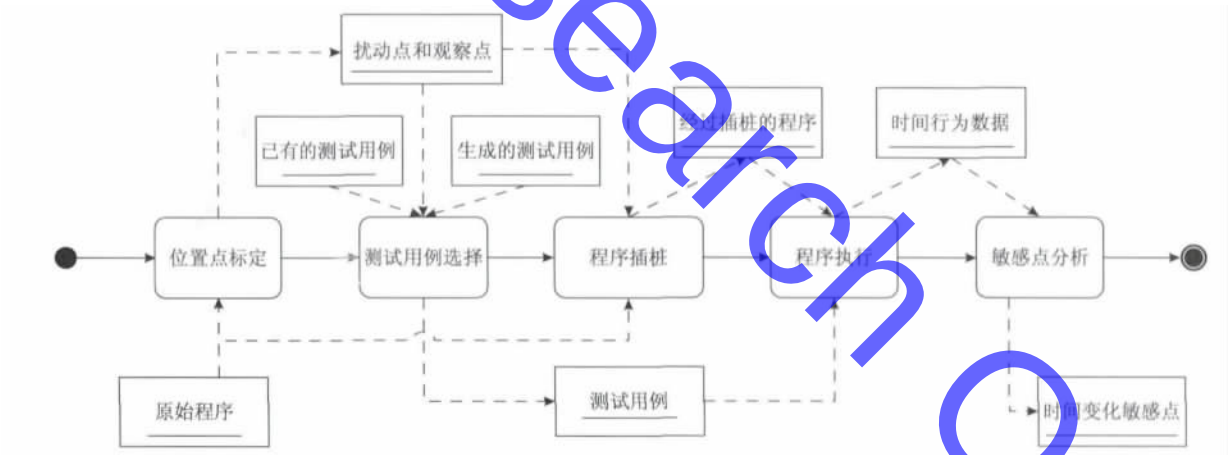

图 2 时间变化敏感点检测流程

算法 1 描述了位置点的标定方法,它扫描待测程序$\mathcal{P}$来生成扰动点集合 P_d和观察点集合 P_o. 我们给待测程序中每一个位置点都规定了一个唯一的编号 id,该编号标注当前位置点所在源文件及行列号等信息,整个检测过程都通过这个编号来识别位置点. 扰动点的生成集成了一个的静态分析预处理函数 $static()$,该函数通过静态方法预测程序中各个位置点被执行到的频繁程度,并将较为频繁的执行点输出. 在 2.3 节里,我们将详细描述该静态分析预处理函数的实现细节. 另外,当 $static()$函数输出的扰动点数目未能达到我们根据运行平台的承受能力而估计的最大扰动数目 N_d 时,我们还使用随机方法来补充一批扰动点,当不与 $static()$函数的输出重复时,随机函数 $random()$按定义 1 的要求随机地输出程序$\mathcal{P}$中执行语句的结束点来作为扰动点. 最终我们将静态分析预处理函数与随机函数产生的这两部分位置点均作为扰动点提供给后续步骤做进一步测试. 算法 1 直接以程序任务的结束位置点作为观察点,它包括线程结束点和消息处理函数的结束点,以便在后续步骤中观察程序任务执行结束时间的变化. 这些生成的扰动点和观察点将用于在第 3 步程序插桩中标识插桩位置.

算法 1. 位置点的标定.

输入:原始程序$\mathcal{P}$

输出:扰动点集合 P_d,观察点集合 P_o

$P_d \leftarrow static(\mathcal{P})$, $P_o \leftarrow \varnothing$

FOR $\forall p \in \mathcal{P}$ DO
  IF p 为任务结束点 THEN $P_o \leftarrow P_o \cup \{p\}$
END FOR
WHILE $number(P_d) < N_d$ DO
  $P_d \leftarrow P_d \cup random(p \in \mathcal{P})$
END WHILE

算法 2 用于选择测试用例，从而生成测试用例集合 C. 如果待测程序已经经过了功能测试，算法 2 直接使用功能测试遗留的测试用例 $fun\_case()$. 当测试用例数量未达到目标数量 N_c 时，随机用例生成方法 $random\_case()$ 帮助生成测试用例，该方法首先生成随机数，然后依据程序的输入类型来约减它们，以满足程序的输入要求（例如，当程序需要输入 0 到 99 之间的整数类型时，它将生成的随机数取整并 mod 100）. 处理后的数值将被用作程序的输入. 在程序的输入类型要求过于复杂时，检测过程会要求用户手工提供测试用例 $manual\_case()$. 测试用例选择程序会帮助我们过滤掉不能满足测试要求的用例，即执行路径不经过扰动点或对应观察点的测试用例. 算法 2 的第 5 行开始描述测试用例的选择过程. 首先我们在所有的扰动点和观察点上插桩记录语句 $w()$ 以标记程序的执行轨迹，得到 $\mathcal{P}'$；$w()$ 的实现向文件输出当前的位置点编号 id 以及该位置点被标定的类型——是扰动点还是观察点. 当程序执行路径经过当前位置点时，相应的程序执行会调用该位置点的记录语句 $w()$ 而输出该处的位置点 id. 由于编号 id 标注了当前位置点的各种信息，我们能够很容易地定位出是哪一个位置. $eval\_\mathcal{P}'()$ 会驱动测试用例逐一执行程序 $\mathcal{P}'$ 以记录各用例的执行路径所经过的位置点，最后分析这些执行路径的输出以选择出同时覆盖扰动点和其观察点的测试用例，未能覆盖任何扰动点或者未能覆盖任何观察点的用例将会被直接过滤. 这些经过选择的测试用例将在第 4 步中作为程序输入来驱动检测过程的执行.

算法 2.　测试用例选择.

输入：原始程序 $\mathcal{P}$，扰动点集合 P_d，观察点集合 P_o

输出：测试用例集 C

$C \leftarrow fun\_case()$，$\mathcal{P}' \leftarrow \forall p \in P_d \cup P_o$，$\mathcal{P}[p \leftarrow w()]$
WHILE $number(C) < N_c$ DO
  $C \leftarrow random\_case() \vee manual\_case()$
END WHILE
FOR $\forall case \in C$ DO
  IF $\forall p \in eval\_\mathcal{P}'(case)$，$p \notin P_d \vee p \notin P_o$ THEN
    $C \leftarrow C - \{case\}$
  END IF
END FOR

算法 3 将时延代码 d 插桩到程序的扰动点，并将时间记录函数 o 插桩到程序的观察点，以形成一组插桩后的程序 $\mathcal{P}_t$；作为对比，算法 3 还生成了仅在观察点插桩时间记录函数 o 的程序版本 $\mathcal{P}_o$. 本文在 Linux 平台上针对 C/C++ 程序实现了检测算法，其中时延代码 d 是按照定义 2 的要求所预定义的一组与待测程序控制流无关的代码块 $delay\_fun()$. 它包括不同量级的 Linux 系统时延调用 $sleep()$ 和 $mdelay()$，普通的循环代码块以及普通的执行代码块. 本文的实现通过调用 Linux 系统时间记录接口 $gettimeofday()$ 来编写时间记录函数 o，它的精度可以达到微秒量级. 另外，我们把时延代码 d 与时间记录函数 o 的自定义接口都开放给了用户，以方便用户在使用中扩展这些插桩代码. 这样，经过插桩的程序经过第 4 步的执行，就可以记录到当前插桩的时延代码对观察点所记录的执行时间的影响. 对比程序 $\mathcal{P}_o$ 在第 4 步的程序执行过程中记录原始程序在未经扰动下程序任务的执行时间，时延代码 d 本身的时延值 t 将用于后续步骤时间敏感率的计算. 在算法 3 的输出元素 $\mathcal{P}_t(p_1, p_2)$ 和 $\mathcal{P}_o(p_2)$ 中，括号中的 (p_1, p_2) 与 (p_2) 是标签，用来标定当前经过插桩程序的插桩点.

算法 3.　程序插桩.

输入：原始程序 $\mathcal{P}$，扰动点集合 P_d，观察点集合 P_o

输出：经过插桩的程序集 $\mathbb{P}$

$\mathbb{P} \leftarrow \varnothing$
FOR $\forall p_1 \in P_d, \forall p_2 \in P_o, \forall d \in 2^{delay\_fun()}$ DO
  $t \leftarrow eval(d)$
  $\mathcal{P}_t(p_1, p_2) \leftarrow \mathcal{P}[p_1 \leftarrow d(), p_2 \leftarrow o()]$
  $\mathcal{P}_o(p_2) \leftarrow \mathcal{P}[p_2 \leftarrow o()]$
  $\mathbb{P} \leftarrow \mathbb{P} \cup \{(\mathcal{P}_t(p_1, p_2), \mathcal{P}_o(p_2), t)\}$
END FOR

算法 4 使用第 2 步生成的测试用例来驱动执行第 3 步插桩后的程序 $\mathcal{P}_t$ 和 $\mathcal{P}_o$. 在执行过程中，$\mathcal{P}_t$ 调用了在观察点插桩的时间记录函数来记录程序经过时延后到达观察点的执行时间 T'；与此同时，对比版本程序 $\mathcal{P}_o$ 的执行，输出原始程序直接到达观察点的执行时间 T. 有了这些数据，算法 4 依照定义 4 中的式(1)来计算时间杠杆率 r. 这些对于每一次执行而产生的时间杠杆率细致地描述了在这次执行中当前扰动点对所观察的程序任务执行时间的大小，算法 4 将它们放入集合 R 中输出.

算法 4.　程序执行.

输入：经过插桩的程序集 $\mathbb{P}$，测试用例集 C

输出：时间杠杆率的集合 R

$R \leftarrow \varnothing$
FOR $\forall(\mathcal{P}_t(p_1,p_2),\mathcal{P}_o(p_2),t)\in\mathbb{P}$ DO
 FOR $\forall case\in C$ DO
  $T'(p_1,p_2,t,case)\leftarrow eval\_\mathcal{P}_t(p_1,p_2)(case)$
  $T(p_2,case)\leftarrow eval\_\mathcal{P}_o(p_2)(case)$
  $r(p_1,p_2,t,case)\leftarrow T'(p_1,p_2,t,case)-T(p_2,case)/t$
  $R\leftarrow\{r(p_1,p_2,t,case)\}\cup R$
 END FOR
END FOR

算法 5 分析第 4 步输出的时间杠杆率数据，将其抽象成针对位置点的高层信息，并以此为根据输出时间变化敏感点. 这一步的主要目标在于将每一次执行所得到的时间杠杆率数据总结成各个扰动点的属性信息，然后依据这些属性信息比较各个扰动点对程序任务执行时间的影响. 由于我们的分析关注于当前扰动点在各次执行中对程序任务的执行时间可能造成的最坏影响，故而最终针对扰动的时间杠杆率取的是不同测试用例下时间杠杆率的最大值 max. 而最终判定时间变化敏感点的依据是不同时延下的平均时间杠杆率. 当这一时间杠杆率的计算值大于用户根据原始程序的实时冗余而给定的阈值 R_l 时，当前扰动点被判定为时间变化敏感点，并最终将该点和该点的平均时间杠杆率输出给用户.

算法 5.　敏感点分析.

输入：各次执行的时间杠杆率的集合 R，扰动点集合 P_d，观察点集合 P_o

输出：时间变化敏感点集 P_s

$P_s\leftarrow\varnothing$,
$r(p_1,p_2,t)\leftarrow\max(\{\forall case,r(p_1,p_2,t,case)\in R\})$
FOR $\forall p_1\in P_d,\forall p_2\in P_o$ DO
 $r(p_1,p_2)\leftarrow average(\{\forall t,r(p_1,p_2,t)\})$
 IF $r(p_1,p_2)>R_l$ THEN
  $P_s\leftarrow P_s\cup\{(p_1,r(p_1,p_2))\}$
 END IF
END FOR

2.3　静态分析预处理过程

在检测流程的第 1 步，标定程序位置点时，扰动点的标定依赖于一个静态分析预处理过程 $static$，它通过静态扫描程序中会频繁执行到的位置来标定扰动点. 我们称这些可能会被频繁执行到的位置为时间敏感疑似点. 它们是程序轨迹可能会停留较长时间的程序位置，或者是程序路径会反复经过的位置.

静态分析预处理过程集成了如图 3 所示的程序分析技术. 首先我们使用可达性分析滤除与当前的程序任务无关的代码模块；而后的递归检测和循环检测可以得到被这些结构包含的所有代码块，这些代码块是相对于路径的高频热点常常聚集的地方；最后，上限估计会粗略地去除明显还在实时软件容忍能力范围内的位置点. 经过这些步骤，我们可以静态地提取时间敏感疑似点，以供后续的测试步骤进一步确认.

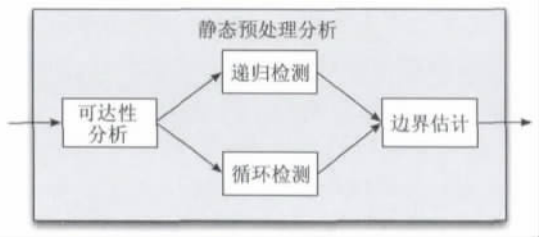

图 3　静态预处理分析的技术架构

虽然在编译优化、软件错误检测以及最坏状况执行时间分析等应用中也使用了类似的静态分析技术，但在处理时间敏感疑似点检测时这些技术需要作一些改动. 另外，我们对算法的精度和效率做了必要的折衷和选择.

可达性分析是一项用于死代码消减的软件错误检测技术[5]. 由于相对于给定的程序任务来说，不可达的源代码模块中一定不存在时间变化敏感点，故我们的静态预处理分析通过可达性分析来提取程序任务和其源代码模块的可达性信息，从而达到为进一步地分析过滤掉不可达模块的目的. 值得注意的是，和传统的可达性分析不同，用于时间变化影响分析的算法并不会消减掉软件中的无用功能代码，因为这样的代码仍然会占用时间，甚至成为时间变化敏感点. 在我们分析中所要消减的不可达代码主要是指相对于给定的程序任务，没有控制路径经过的代码. 算法 6 描述了本文检测方法所采用的可达性分析方法，它采用一种基于控制流图 g_f（Control Flow Graph，CFG）的标记-清除算法（类似于程序设计语言的垃圾收集器）来完成这一分析. 该控制流图的节点集合为 N_f、有向边集合为 E_f 以及入口节点为 n_s. 分析器初始将所有代码块标为“不可达”，而后从当前任务控制流图的初始节点开始，依次将可达节点标为“可达”，并加入我们所设定的可达语句集 S_a. 在理想状况下，所有剩下的节点都可以作为不可达节点而被过滤掉. 在实际情况下，当存在难以确定的代码块时，算法将保守地保留这些代码块而由进一步的分析来确认.

算法 6.　可达性分析.

输入：控制流图 $g_f(N_f,E_f,n_s\in N_f)\in G_f$

输出：可达语句集 S_a

$S_a\leftarrow\{n_s\}$

FOR $\forall i,j\in N_f$ DO
  IF $i\in S_a\wedge(i,j)\in E_f$ THEN
   $S_a\leftarrow S_a\cup\{j\}$
  END IF
END FOR

程序中的递归和循环结构块由于占用的程序执行时间较长，是相对于路径的高频热点常常出现之处. 在图 3 中，这两种代码块的检测是我们静态预处理分析的核心. 这里我们采用一种基于函数调用图的贪心算法来检测程序中的递归块，算法 7 描述了递归检测的执行过程. 在一个调用图 g_c 中，每一个节点代表一个函数或过程，而每一条从节点 i 到节点 j 的有向边(记为(i,j))表示一个从过程 i 到过程 j 的调用. 算法 7 描述了递归检测过程，其中 G_c 是所有调用图的集合，而 N_c 和 E_c 则分别表示算法所接受的调用图 g_c 的节点集和有向边集. 当我们把当前程序的调用关系图输入给算法 7 所描述的递归检测算法后，算法经过迭代标记，快速地输出处于递归块中的函数过程.

算法 7.　递归检测.

输入：调用图 $g_c(N_c,E_c)\in G_c\cdots$

输出：递归函数集 F_r

$E'_c\leftarrow E_c$
FOR $\forall i,j,k\in N_c$ DO
  IF $(i,k)\in E'_c\wedge(k,j)\in E'_c$ THEN
   $E'_c\leftarrow E'_c\cup\{(i,j)\}$
  END IF
END FOR
$F_r\leftarrow\{i\mid i\in N_c\wedge(i,i)\in E'_c\}$

在本文的静态预处理分析中，我们采用了 Sreedhar-Gao-Lee[6] 所提出的基于支配节点的循环检测算法. 它相对于间隔分析与结构分析等其它循环检测技术[7-8]的优势在于该算法可以高效地迭代处理不可规约循环. 算法 8 描述了循环检测算法的过程，它使用了基于支配节点的循环检测技术. 当从控制流图的入口节点到达节点 n 的每一条路径都必经过节点 d 时，我们称节点 d 支配节点 n，记为 $d\ dom\ n$. 假如控制流图的某一个深度优先生成树的某一条回边的目标节点支配其源节点，则此边在当前控制流图的任意深度优先生成树中都为回边. 并且由这条回边可以确定一个自然循环块，这一循环块包括节点 d 以及所有存在不通过节点 d 到达节点 n 的路径的节点. 这样，循环检测问题就等价于支配节点的检测. 在算法 8 中，我们首先生成支配关系 dom，它将控制流图 g_f 中的每一个节点映射到所有支配它的节点集合. 而后依据生成的 dom 关系构建循环块.

算法 8.　循环检测.

输入：控制流图 $g_f(N_f,E_f,n_s\in N_f)\in G_f$

输出：循环块集合 L_s

$dom\leftarrow\{n_s\rightarrow\{n_s\}\}$, $L_s\leftarrow\varnothing$
FOR $\forall n\in N_f-\{n_s\}$ DO
  $dom\leftarrow\{n\rightarrow N_f\}\cup dom$
END FOR
REPEAT
$C\leftarrow\varnothing$
  FOR $\forall n\in N_f-\{n_s\}$ DO
    IF$\{n\rightarrow\{n\}\cup\bigcap_{(n',n)\in E_f}dom(n')\}\notin dom$ THEN
     $dom(n)\leftarrow\{n\}\cup\bigcap_{(n',n)\in E_f}dom(n')$
     $C\leftarrow C\cup\{n\}$
    END IF
  END FOR
UNTIL $C\leftarrow\varnothing$
FOR $\forall n_1\in N_f-\{n_s\}$ DO
  IF $\exists n_1,n_2,\cdots,n_k\in N_f$,
     $(n_i,n_{i+1})\in E_f\wedge n_k\in dom(n_1)$ THEN
    $S\leftarrow\{n_1,n_k\}$
    FOR $\forall m_1\in N_f-\{n_1,n_k\}$ DO
     IF $\exists m_1,m_2,\cdots,m_k\in N_f$,
       $(m_i,m_{i+1})\in E_f\wedge m_k=n_1\wedge$
       $m_k\notin\{m_1,m_2,\cdots,m_k\}$ THEN
      $S\leftarrow S\cup\{m_1\}$
     END IF
    END FOR
  END IF
  $L_s\leftarrow L_s\cup\{S\}$
END FOR

算法 9 描述了本文使用的基于模板的边界估计方法，它是一项在最坏状况执行时间分析中提出的技术[9-10]，用于检测程序中循环块的迭代次数上限. 在算法 9 中，当循环块匹配

for $(i=i_1;\ i<i_n;\ i+=k)$

或 $i=i_1$; while $(i<i_n)$ $\{\cdots\ i+=k;\}$

或 $i=i_1-k$; do $\{\cdots\ i+=k;\}$ while $(i<i_n)$

等模板时，求解函数 $solve$ 会解得循环块边界 $(i_n-i_1)/k$，在其它情况下，算法 9 将其边界值标为 $\perp$. 这些边界值被标为 $\perp$ 的部分和边界值较高的循环块将被直接输出给 2.2 节所描述的测试算法测试，以判定其中是否存在时间变化敏感点.

算法 9.　边界估计.

输入：循环块集合 L_s

输出：边界信息 $bound$

```
bound ← ∅
FOR ∀b∈L_s DO
  IF pattern(b) ≠ unknown THEN
    bound(b) ← solve(b)
  ELSE
    bound(b) ← ⊥
  END IF
END FOR
```

3 实例研究与评估

这一节给出时间变化影响分析在 Mälardalen WCET 学术基准用例集和工业界的实际开源项目 Free Lossless Audio Codec(FLAC)上的实例研究. 首先介绍实验设计与支撑环境;然后分别给出两个案例上的实验结果与分析评估.

3.1 实验设计与支撑环境

由于静态分析库 ROSE[11] 拥有较丰富的程序转换的机制,且可用于 Fortran、C 和 C++ 等多种语言的静态分析,因此我们时间变化影响分析的原型内核基于 ROSE 实现. 此内核负责预处理和动态测试插桩,它是以对程序的抽象语法树(Abstract Syntax Tree,AST)使用 visitor 设计模式来遍历实现的. 而外围的测试驱动、动态调度、结果分析等则是使用 Ruby 脚本来实现.

整个实验评估过程运行在一台 Dell Optiplex 755 工作站上,其 CPU 型号为 Intel Core 2 Duo E4500 2.2GHz,拥有 2GB 内存容量,操作系统为带 2.6.24-29-generic 版本 Linux 内核的 Ubuntu 8.04,编译器为 gcc 4.2.4, Ruby 解释器版本为 Ruby 1.8.6 patchlevel 111.

本节以 2.1 节所定义的时间杠杆率 $r(p_1,p_2,t)$ 作为评估依据,并使用统计中的频数分布对位置点进行成组. 以对数分布的形式,我们构建了[0,1.5]、(1.5,5]、(15,50]、(15,50]、(50,150]、(150,500]、(500,1500]和(1500,+∞)这 8 个数值段集合 R,每个集合代表一个组. 当程序中某个位置点的时间杠杆率 $r\in R$ 时,我们将这个位置点的计数统计在对应的分组上. 作为示例,在我们的评估中以 5 和 50 作为阈值,在实际时间杠杆点检测中,阈值取决于待分析项目的需求以及所分析任务的时间冗余情况. 当待分析任务在原始系统中有较大的实时性冗余时,我们可以将阈值取高,以过滤出更少的时间变化敏感点,反之我们应当将阈值取低,从而将更多的时间变化敏感点汇报给用户,以提醒软件维护人员注意. 在我们的实验中,当时间杠杆率到达 50 以上时,依据经验我们就将这样的点判为时间变化敏感点. 而当时间杠杆率小于或等于 5 时,由于时间影响确实很小,我们相信这样的点是安全点. 而当时间杠杆率在 5 和 50 之间时,我们的方法不能确定这些点是否安全,对这样的位置点的分析需要一步的判断或者手工确认.

3.2 WCET 基准用例集的评估

首先我们使用了 Mälardalen WCET 基准作为评估用例集. 这一用例集本身是设计用于评估最坏状况执行时间分析的,故而在我们看来,时间变化影响分析在这些用例上也具有实际意义. 在本组实验中,评估用例集自身解决了实验所需的测试用例,因而不需要考虑测试用例的覆盖问题. 为了提高所求解度量的辨识度,我们将 5 组不同量级的时延代码作为插桩代码来对待测任务做实时性扰动,经过测算这 5 组代码的时延值分别为 6.74E-6s、2.69E-5s、4.15E-4s、8.11E-3s 以及 1.31E-1s,这样就基本上涵盖了从微秒到秒级的时延状况. 这些代码时延方式包括了普通语句时延、循环时延、API 等待时延等等,以保证结果在统计上更为客观.

在这里各组 WCET 用例的代码规模都不大,我们对程序中所有位置点直接做扰动而得到较为全面的结果. 表 1 给出了这些基准用例集的评估结果. 我们首先按对数分布进行分组,而后再将各个位置点按平均时间杠杆率 $r\in R$ 统计得到表 1 对应的频数分布,对应位置点的计数被统计到各个分组内. 从表 1 结果来看,Mälardalen WCET 基准用例集中的用例一共可以分为 3 类:不具有时间变化敏感点的用例,例如 nsichneu 和 bs,对于这一类用例软件维护人员不需要太多担心因为软件演化上的因位置的不小心而引入时间错误隐患;具有明显时间变化敏感点的用例,例如 adpcm 和 edn,软件维护人员在对这些用例的时间变化敏感点发生修改时需要特别注意;难以明确判断的用例,例如 inssort 和 select,这类用例由于存在中等敏感的位置点,因而需要对相关的疑似点做手工校对. 由于篇幅的限制,在这里我们仅仅举例对各不同类型用例的位置点分析进行说明,更详细的追踪数据可从地址:http://seg.nju.edu.cn/~eytang/timeleverage/trackingdata.tar.gz 下载.

表 1　WCET 基准用例集各位置点时间杠杆率的频数分布

基准用例集	R 的频数/占比							
	[0,1.5]	(1.5,5]	(5,15]	(15,50]	(50,150]	(150,500]	(500,1500]	(1500,+∞)
adpcm	42/14.38%	164/56.16%	61/20.89%	16/5.48%	0/0%	0/0%	0/0%	9/3.08%
bs	16/76.19%	5/23.81%	0/0%	0/0%	0/0%	0/0%	0/0%	0/0%
bsort	17/58.62%	0/0%	0/0%	0/0%	6/20.69%	0/0%	0/0%	6/20.69%
cnt	33/80.49%	0/0%	2/4.88%	0/0%	6/14.63%	0/0%	0/0%	0/0%
compress	122/76.25%	0/0%	1/0.62%	30/18.75%	7/4.38%	0/0%	0/0%	0/0%
cover	595/99.50%	0/0%	1/0.17%	1/0.17%	1/0.17%	0/0%	0/0%	0/0%
crc	0/0%	1/12.50%	0/0%	0/0%	0/0%	4/50.00%	2/25.00%	1/12.50%
duff	11/40.74%	1/3.70%	14/51.85%	0/0%	1/3.70%	0/0%	0/0%	0/0%
edn	61/53.51%	1/0.88%	0/0%	23/20.18%	17/14.91%	3/2.63%	8/7.02%	1/0.88%
expint	40/88.89%	0/0%	0/0%	1/2.22%	4/8.89%	0/0%	0/0%	0/0%
fac	6/60.00%	0/0%	2/20.00%	2/20.00%	0/0%	0/0%	0/0%	0/0%
fdct	28/22.22%	0/0%	98/77.78%	0/0%	0/0%	0/0%	0/0%	0/0%
fft1	2/8.70%	1/4.35%	12/52.17%	8/34.78%	0/0%	0/0%	0/0%	0/0%
fibcall	16/84.21%	0/0%	0/0%	3/15.79%	0/0%	0/0%	0/0%	0/0%
fir	23/69.70%	0/0%	9/27.27%	0/0%	0/0%	1/3.03%	0/0%	0/0%
inssort	19/73.08%	0/0%	3/11.54%	4/15.38%	0/0%	0/0%	0/0%	0/0%
jcomplex	10/55.56%	1/5.56%	7/38.89%	0/0%	0/0%	0/0%	0/0%	0/0%
jfdctint	31/28.97%	0/0%	74/69.16%	0/0%	2/1.87%	0/0%	0/0%	0/0%
lcdnum	38/86.36%	1/2.27%	5/11.36%	0/0%	0/0%	0/0%	0/0%	0/0%
lms	4/5.48%	1/1.37%	0/0%	6/8.22%	23/31.51%	25/34.25%	5/6.85%	9/12.33%
loop3	168/60.87%	62/22.46%	42/15.22%	4/1.45%	0/0%	0/0%	0/0%	0/0%
ludcmp	27/46.55%	0/0%	18/31.03%	13/22.41%	0/0%	0/0%	0/0%	0/0%
matmult	13/54.17%	3/12.50%	0/0%	2/8.33%	0/0%	2/8.33%	3/12.50%	1/4.17%
minmax	23/88.46%	3/11.54%	0/0%	0/0%	0/0%	0/0%	0/0%	0/0%
minver	48/42.11%	33/28.95%	31/27.19%	2/1.75%	0/0%	0/0%	0/0%	0/0%
ns	11/73.33%	1/6.67%	0/0%	1/6.67%	1/6.67%	0/0%	1/6.67%	0/0%
nsichneu	835/48.60%	883/51.40%	0/0%	0/0%	0/0%	0/0%	0/0%	0/0%
qsort-exam	23/32.39%	31/43.66%	16/22.54%	1/1.41%	0/0%	0/0%	0/0%	0/0%
qurt	36/53.73%	21/31.34%	5/7.46%	5/7.46%	0/0%	0/0%	0/0%	0/0%
select	35/54.69%	3/4.69%	11/17.19%	15/23.44%	0/0%	0/0%	0/0%	0/0%
sqrt	17/68.00%	2/8.00%	5/20.00%	1/4.00%	0/0%	0/0%	0/0%	0/0%
statemate	597/80.24%	147/19.76%	0/0%	0/0%	0/0%	0/0%	0/0%	0/0%

nsichneu 是一段由 Petri 模型生成的代码，它模拟了一个具有大量分支语句以及少量简单循环的模型行为. 经分析，此程序中所有位置点的时间杠杆率均小于或等于 5，这意味着此程序中各个位置都相对安全，用户不需要担心因为修改位置的不同而在软件演化过程中引入时间错误.

adpcm 是自适应差值脉冲编码调制[12]的一个结构化实现. 在这个程序中，我们的原型共发现了 9 处时间杠杆率在 1500 倍以上的位置点，其中 5 处在函数 *my_fabs* 里，另外 4 处在函数 *my_sin* 的第 2 和第 3 个循环里. 软件维护者在涉及到这些位置的修改时需要特别注意.

inssort 是数组插入排序的一个具体实现，其中有一个两层的嵌套循环. 就我们工具分析，这一程序没有明显的时间变化敏感点，但有 7 个疑似点需要稍加注意. 当我们追踪这个程序的结果时，发现其中 4 个位于内层循环的位置点的时间杠杆率在 15 到 50 之间，而 3 个位于外层循环的位置点的时间杠杆率在 5 到 15 之间，维护人员需要对这些位置点稍加注意.

3.3　FLAC 开源项目上的评估

自由音频无损编码(Free Lossless Audio Codec，FLAC)项目是一个多媒体编解码的开源软件项目. 当多媒体播放器调用这一模块进行解码时，时间属性的破坏会引起声音的延迟和畸变. 我们的评估基于 flac 1.2.1 版，这是我们在进行评估实验时所能得到的最新版本. 我们没有得到该项目发布前的测试用例集，故而手动收集了一个测试用例集来作为输入. 该用例集分别有通俗歌曲、古典音乐、中国民乐、白噪声、电影配声(撞击声、汽车马达声)、自录人声、自然音效以及低采样音效一共有 9 组测试用例，基本可以涵盖主要的音频类型.

我们评估了项目中最主要的两个组件，分别是用于提供编码和解码所需的核心 API 的 libFLAC 库(在表 2 中以 lib 标识)以及调用 libFLAC 库进行实际编码解码的控制台驱动程序(在表 2 中以 bin 标识). 由于这两个组件都具有一定的规模，在分析

中我们采用静态预处理分析预先得到了疑似点，并且，为了减少系统的误报，我们根据评估环境的承受能力，仍然给扰动测试模块随机添加了一定数量的原始位置点以进行分析.

表 2 FLAC 开源项目各位置点时间杠杆率的频数分布

任务	组件名称	R 的频数							
		[0,1.5]	(1.5,5]	(5,15]	(15,50]	(50,150]	(150,500]	(500,1500]	(1500,+∞)
编码	bin	2190	69	12	1	5	24	0	2
	lib	2557	16	8	3	0	0	0	0
解码	bin	2306	63	8	1	49	0	0	1
	lib	2047	182	90	87	0	0	0	0

我们分别对 FLAC 项目中这两个组件的编码和解码任务做了评估，结果如表 2 所示. 对于扰动测试输入的每一个位置点，我们仍然使用了同前面 WCET 实验相同的 5 组不同量级的时延代码来做插桩扰动. 由于结果并不是对所有位置点而得，相应的百分比信息就变得没有意义，故在表 2 中去除.

同上一个实验一样，我们仍然以分组后的频数分布来查找程序中的时间变化敏感点. 从表 2 的数据来看，较明显的时间变化敏感点都落在了控制台驱动程序里，这说明相应 libFLAC 库的设计在时间属性方面做了比较充分的考虑，其中每一个我们所测试到的位置点，都满足时间杠杆率 $r \leq 50$. 而坐落在控制台驱动程序里的 3 个时间杠杆率大于 1500 倍的位置点经过追踪，分别在 encode.c 的 2147 行和 2148 行以及 decode.c 的 1067 行. 这 3 个点都落在上限很高的循环体内，因而也被我们的静态预处理分析直接捕获到.

3.4 时间变化敏感点检测的度量误差评估

以上两个实验都在测试用例中只考虑了输入数据，对于时间变化敏感点检测的度量存在的误差，我们通过计算程序各次执行的平均变异系数来进行评估. 经过计算，Mälardalen WCET 基准用例集的平均变异系数为 0.281，而 FLAC 项目的平均变异系数为 1.284，作为变异系数的这两个数值是和时间杠杆率处于同一数量级，而我们在评估中判断时间杠杆点的依据数值在 50 以上，远远高于变异系数的数量级. 这表明评估中相应的误差没有对检测结果产生影响.

4 相关工作

经过查找相关文献调研，我们之前并未发现与围绕软件系统演化所带来的时间影响方面有直接相关工作，但是与本文间接相关的研究工作是大量存在的. 在这里我们分析和讨论其中的主要方面，主要包括：(1) 软件修改影响分析(Impact Analysis)；(2) 最坏状况执行时间(WCET)分析；(3) 调度相关的敏感度分析(Sensitive Analysis)以及 (4) 频繁执行点(Hot Spots)的定位研究.

软件修改影响分析. 为了预测软件修改可能带来的后果，大量软件修改影响分析相关的研究工作已经开展[1,13]. 其中既有通过基于软件的生命周期或者软件源代码的语义而采用的静态分析技术来获得影响数据的方法[3-4,14]，又有通过动态执行软件来收集相关的影响信息的途径[2,15-16]. 尽管这中间有很多研究都使用了基于插桩的技术，但这些研究都没有涉及到实时软件在执行时间上的需求，而是集中在软件的功能需求上. 本文运用了基于扰动的方法为这个方面工作增添一个新的子类——时间变化影响分析.

最坏状况执行时间分析. 实时软件开发和验证过程中的一个必要步骤，是通过对最坏状况执行时间的判定来保障其实时性. 对于最坏状况执行时间分析的研究，已经有十多年的历史了[17-19]. 根据 Bate[20] 和 Seshia[21] 的分类，最坏状况执行时间分析的研究主要有两个独立的方向：基于度量的最坏状况执行时间分析技术和基于静态分析的最坏状况执行时间分析技术. 近年来的一些静态最坏状况执行时间上的研究已经开始考虑实时系统硬件微结构的影响，从而提高分析精度[22-23]. 不同于我们工作的是，这些分析工作所注重的是系统本身最坏状况的执行时间，本文的研究所关注的是程序修改和软件演化对时间产生的影响，能在软件演化发生前就开始分析，从而得到可以指导软件维护人员减少或避免时间错误的相关信息.

敏感度分析. 敏感度分析是统计领域里的常用方法，它主要用于分析统计结果变化的原因是由统计模型怎样的输入变化所引起的[24]. 这样的技术同样被用于分析实时系统[25-26] 甚至是分布式实时系统[27] 的系统参数将怎样影响其时间约束. 和我们的

工作不同，实时系统的敏感度分析主要聚焦于系统模型，并以系统的可调度性问题为其目标. 这些技术关注于怎样获得引起实时系统不可调度的最有可能的参数[28]. 而我们的工作主要聚焦于代码层面，而不是设计模型层面，主要目的是在软件演化中帮助维护人员避免问题的发生.

频繁执行点的定位. 程序频繁执行点的概念是随着软件优化技术的发展所提出来的，通过程序剖析技术(Program Profiling)和程序分析技术所检测出的频繁执行点，可以帮助软件优化工具调动更多的资源在频繁执行点做局部优化以提高整体软件的优化效果，即减少程序的平均执行时间[29-33]. 与我们在时间变化敏感点检测中关注的程序任务执行路径上多次经过的位置点不同，程序频繁执行点是程序大量执行后统计出的高频热点. 程序频繁执行点不一定是时间变化敏感点，同样时间变化敏感点也不一定是程序频繁执行点.

5　结束语

时间属性的正确性在实时软件中非常重要，而时间相关的软件错误不仅在软件的设计开发阶段会被引入，软件维护和演化阶段也会产生时间约束破坏等问题. 本文的主要贡献可以归纳为如下几个方面：(1) 针对软件维护阶段引入的实时性错误问题，本文提出了时间变化影响分析的概念. 它可以在维护工程师修改软件之前做出分析，从而帮助维护人员减少或者避免时间变化错误的发生；(2) 本文开发了一种基于扰动的测试技术，来实现相关的时间变化影响分析，即实现了时间变化敏感点的检测. 并在此基础上，采用了多种静态程序分析技术来优化相关的待检测点的发现过程，以提高分析整体的运行效率；(3) 本文对所提出的分析方法给出了原型软件工具的实现，并在 Mälardalen WCET 基准用例集和开源项目 Free Lossless Audio Codec (FLAC)上做了实例评估. 评估结果显示了我们分析方法及优化的有效性.

在本文工作的基础上，未来我们将尝试把测试用例扩展成包括处理器调度、中断序列、多级缓存等因素影响的形式，以得到更为精确的处理. 另外，我们也会考虑进一步提高静态分析预处理效率的检测方法.

参　考　文　献

[1] Arnold R S. Software Change Impact Analysis. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996

[2] Law J, Rothermel G. Whole program path-based dynamic impact analysis//Proceedings of the 25th International Conference on Software Engineering(ICSE'03). Washington, DC, USA: IEEE Computer Society, 2003: 308-318

[3] Zimmermann T, Zeller A, Weissgerber P, Diehl S. Mining version histories to guide software changes. IEEE Transactions on Software Engineering, 2005, 31(6): 429-445

[4] Ren X, Ryder B G, Stoerzer M, Tip M. Chianti: A change impact analysis tool for java programs//Proceedings of the 27th International Conference on Software Engineering (ICSE'05). St. Louis, MO, USA, 2005: 664

[5] Janota M, Grigore R, Moskal M. Reachability analysis for annotated code//Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems (SAVCBS'07). New York, NY, USA, 2007: 23-30

[6] Sreedhar V C, Gao G R, Lee Y. Identifying loops using DJ graphs. ACM Transactions on Programming Languages and Systems (TOPLAS), 1996, 18(6): 649-658

[7] Muchnick S S. Advanced Compiler Design and Implementation. San Francisco, CA: Morgan Kaufmann Publishers, 1997

[8] Ramalingam G. On loops, dominators, and dominance frontiers. ACM Transactions on Programming Languages and Systems (TOPLAS), 2002, 24(5): 455-490

[9] de Michiel M, Bonenfant A, Casse H, Sainrat P. Static loop bound analysis of C programs based on flow analysis and abstract interpretation//Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08). Kaohsiung, Taiwan, China, 2008: 161-166

[10] Zhou L, Chen K. Recursive symbolic bound analysis in loop structure//Proceedings of the 2010 the 2nd International Conference on Computer and Automation Engineering (ICCAE). Singapore, 2010: 76-80

[11] Quinlan D. ROSE: Compiler support for object-oriented frameworks//Proceedings of the Conference on Parallel Compilers (CPC2000). Aussois, France, 2000: 215-226

[12] ITU-T Recommendation G. 722, 7 kHz audio-coding within 64 kbit/s, Std., November, 1988

[13] Breech B, Tegtmeyer M, Pollock L. A comparison of online and dynamic impact analysis algorithms//Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005). Manchester, UK, 2005: 143-152

[14] Acharya M, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems//Proceedings of the 33rd International Conference on Software Engineering(ICSE'11). New York, NY, USA, 2011: 746-755

[15] Orso A, Apiwattanapong T, Harrold M J. Leveraging field data for impact analysis and regression testing. ACM SIGSOFT Software Engineering Notes, 2003, 28(5): 128-137

[16] Law J, Rothermel G. Incremental dynamic impact analysis

for evolving software systems//Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003). Denver, USA, 2003: 430-441

[17] Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem—Overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS), 2008, 7(3): 36:1-36:53

[18] Lim S, Bae Y H, Jang G T, et al. An accurate worst case timing analysis for RISC processors. IEEE Transactions on Software Engineering, 1995, 21(7): 593-604

[19] Ermedahl A, Stappert F, Engblom J. Clustered worst-case execution-time calculation. IEEE Transactions on Computers, 2005, 54(9): 1104-1122

[20] Bate I, Khan U. WCET analysis of modern processors using multi-criteria optimisation. Empirical Software Engineering, 2011, 16(1): 5-28

[21] Seshia S A, Rakhlin A. Game-theoretic timing analysis// Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2008). San Jose, USA, 2008: 575-582

[22] Grund D, Reineke J, Gebhard G. Branch target buffers: WCET analysis framework and timing predictability. Journal of Systems Architecture, 2011, 57(6): 625-637

[23] Hardy D, Puaut I. WCET analysis of instruction cache hierarchies. Journal of Systems Architecture, 2011, 57(7): 677-694

[24] Saltelli A, Ratto M, Andres T, et al. Global Sensitivity Analysis. The Primer. Chichester, UK: John Wiley & Sons, Ltd, 2008

[25] Vestal S. Fixed-priority sensitivity analysis for linear compute time models. IEEE Transactions on Software Engineering, 1994, 20(4): 308-317

[26] Bini E, Natale M D, Buttazzo G. Sensitivity analysis for fixed-priority real-time systems//Proceedings of the 18th Euromicro Conference on Real-Time Systems. Dresden, Germany, 2006: 13-22

[27] Racu R, Jersak M, Ernst R. Applying sensitivity analysis in real-time distributed systems//Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 2005). San Francisco, USA, 2005: 160-169

[28] Zhang F, Burns A, Baruah S. Sensitivity analysis for EDF scheduled arbitrary deadline real-time systems//Proceedings of the 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). Macau, China, 2010: 61-70

[29] Merten M C, Trick A R, George C N, et al. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization//Proceedings of the 26th International Symposium on Computer Architecture. Atlanta, USA, 1999: 136-148

[30] Buse R, Weimer W. The road not Taken: Estimating path execution frequency statically//Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE 2009). Vancouver, Canada, 2009: 144-154

[31] Gatlin K S. Profile-Guided optimization with microsoft visual C++ 2005. Microsoft Corporation[Online]. March, 2004. Available: http://msdn.microsoft.com/en-us/library/aa289170(v=vs.71).aspx

[32] Merten M C, Trick A R, Nystrom E M, et al. A hardware mechanism for dynamic extraction and relayout of program hot spots//Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00). Vancouver, British Columbia, Canada, 2000: 59-70

[33] Yang C, Lee C. HotSpot cache: Joint temporal and spatial locality exploitation for I-cache energy reduction//Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED'04). New York, NY, USA, 2004: 114-119

附 录. 时间变化敏感点检测方法的理论探讨.

定理 1. 当程序任务的执行时间仅和指令执行相关时，设 c_1、c_2 为任意两个在相同的程序位置点 p_1 所做的时延修改，则它们对于同一观察点 p_2 的时间杠杆率相等.

证明. 当程序任务的执行时间仅和指令执行相关时，可以把进程的执行时间写成 $t=\tau n$ 的形式，其中 τ 是每一个时钟周期所占的时间，n 是这一个进程所需要的时钟周期数. 设当前任务 A 将所修改的代码执行了 k 次，由于 c_1、c_2 是在相同位置作修改，且它们都是定义 2 所定义的时延修改，故任务 A 将它们都执行了 k 次. 从而有

$$r(p_1,p_2,t_1)-r(p_1,p_2,t_2)=\frac{T_1'-T}{t_1}-\frac{T_2'-T}{t_2}$$

$$=\frac{\tau(n_A+kn_{c1})-\tau n_A}{\tau n_{c1}}-\frac{\tau(n_A+kn_{c2})-\tau n_A}{\tau n_{c2}}=0.$$

因此 $r(p_1,p_2,t_1)=r(p_1,p_2,t_2)$ 成立. 证毕.

由定理 1，当时延修改的时间杠杆率仅和程序位置点(即扰动点和观察点)相关时，我们可以将它写成 $r(p_1,p_2)$.

推论 2. 独立于软件控制流程条件的任何时延修改，其对应的时间杠杆率一定为非负.

证明. 由定义 2 可知，对于独立于软件控制流程条件的时延修改来说，代码的删除操作不会造成程序任务执行时间的增加，从而有

$$t_c<0 \Leftrightarrow \text{当前时延修改删除了代码} \Leftrightarrow T'\leq T.$$

将这一结论代入定义 4 中的式(1)，我们可以很容易地得到结论：$r(p_1,p_2,t)\geq 0$. 证毕.

在编译原理中，一个基本块(Basic Block)代表一个在程序中具有单一入口和单一出口的指令代码序列.

定理 3. 令 B 为一个基本块，p_1、p_2 为基本块 B 中的任意两个程序语句中间点，p 为程序中任意观察点. 当程序任务的执行时间仅和指令执行相关时，有

$$\forall p_1,p_2\in B\Rightarrow r(p_1,p)=r(p_2,p).$$

证明. 当前条件下的时间杠杆率仅和程序位置点相关，和具体的修改无关；用程序位置点 p_1、p_2 的相同时延修改来计算时间杠杆率. 由于修改是相同的，根据定理 1 的证明过程，可知 $t_1=t_2$，即 $n_1=n_2$. 又因为点 p_1、p_2 在同一个基本块里，因而任意一条经过 p_1 的路径必然经过经过 p_2，反之亦然. 从而有 $k_1=k_2$. 因此，

$$r(p_1,p)-r(p_2,p)=r(p_1,p,t_1)-r(p_2,p,t_2)$$
$$=\frac{T_1'-T}{t_1}-\frac{T_2'-T}{t_2}$$
$$=\frac{\tau(n+k_1n_1)-\tau n}{\tau n_1}-\frac{\tau(n_A+k_2n_2)-\tau n}{\tau n_2}=0.$$

从而如下结论成立：

$$\forall p_1,p_2\in B\Rightarrow r(p_1,p)=r(p_2,p).$$

证毕.

TANG En-Yi, born in 1982, Ph.D., assistant professor. His research interests include software engineering, program analysis and software testing.

LI Xuan-Dong, born in 1963, Ph. D., professor, Ph. D. supervisor. His research interests include software engineering, formal method, software modeling and analysis, software testing and verification.

Background

With the technology development, software becomes large and complex, which makes building and maintaining reliable software be a great challenge and a hot research area. Real time software requires not only the functional correctness of each module, but also the satisfaction of the strict time constraints on each response in the software, which attracts many researchers to work on the problem of ensuring time constrains in software. Existing work such as worst case evaluation time (WCET) analysis focuses on helping developers to build the correct real-time programs. However, time faults can also be introduced during software maintenance. Our goal is to help maintainers avoid introducing such faults.

This work is supported by the National Natural Science Foundation of China (No. 91118007), the National High Technology Research and Development Program (863 Program) of China (No. 2012AA011205) and the Open Foundation of State Key Laboratory for Novel Software Technology in Nanjing University (No. ZZKT2013B09). In this paper, we utilize the information from the functional testing of the software, and propose an approach to detect the points which largely affect the execution time of the real-time software. We implement our approach as a prototype and evaluate it on Mälardalen WCET Benchmark suite and Free Lossless Audio Codec (FLAC) project. The result shows that our approach successfully detected such points in software. In future, we will continue our work on improving our approach precisely and efficiently.