实时系统程序最差情况执行时间(WCET)的分析*)

姬孟洛　齐治昌
(国防科技大学计算机学院　长沙 410073)

摘　要　事先获知系统中程序最差情况的执行时间(Worst-Case Execution Time，WCET)，是设计和验证实时系统调度及可调度性分析的前提，也是确定周期性任务是否满足其性能目标，从而发现系统性能瓶颈的基础。本文概述了程序 WCET 的分析方法，描述了 WCET 分析的定义和组成，重点总结其中的程序流事实分析方法，并指出程序流事实分析存在的问题和 WCET 分析的研究热点。

关键词　程序流事实分析，最差情况执行时间 WCET 分析，实时系统，软件工程

An Overview of Worst Case Execution Time (WCET) Analysis

JI Meng-Luo　QI Zhi-Chang
(Department of Computer Science, National University of Defense Technology, Changsha 410073)

Abstract　The purpose of Worst-Case Execution Time (WCET) analysis is to provide a-priori information about the worst possible execution time of a program or piece of a program before using them in a system. When designing and verifying real-time systems, WCET estimates can be used to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, to check that interrupts have sufficiently short reaction time, to find performance bottlenecks, and so on. In this paper we overview the analysis methods of WCET analysis, describe its components, and summarize the analysi methods of program flow fact analysis in WCET analysis. We point out the problem in program flow fact analysis and the research hot spot in WCET analysis.

Keywords　Worst-case execution time analysis, Real-time system, Software engineering

1　引言

实时系统与其它应用系统的不同之处在于其正确性具有更加严格的标准。实时系统的正确性不仅取决于它所产生的输出，同时取决于输出产生的时间。实时系统的结果只有在规定的时间范围内完成才是有效的。当没有在规定的时间范围内完成时，轻则降低系统的性能(弱实时系统)，重则引起灾难性的后果(强实时系统)。因此，事先获取系统中每个任务最差情况下的执行时间 WCET(有时也需要知道最好情况下的执行时间(Best-Case Execution Time，BCET)，因为 BCET 的分析和应用与 WCET 基本相同，故统称为 WCET。)对实时系统的时序分析具有特别重要的意义。事实上，事先得知系统中任务的 WCET 既是进行调度及可调度性检测的前提，又是划分系统设计中软硬件界限的一个依据，同时还是确定周期性任务是否满足其性能目标，从而发现系统性能瓶颈的基础。

WCET 分析值必须安全和精确(tightness)，前者保证不能低估最差执行时间，后者要求提供可接受的高估值。

获取程序的 WCET 是实时系统的一个重要研究领域，也是最近 10 多年来的一个研究热点[1]。从 1986 年发表第一篇有关 WCET 的文[2]开始，到目前为止，几乎所有比较发达的国家都有研究机构从事这方面的研究，比较著名的有美国 Florida 州立大学、Princeton 大学、奥地利的 Vienna 技术大学、瑞典的 Uppsala 大学、英国的 York 大学以及韩国国立大学等。

WCET 分析包括动态度量、静态分析和混合方法共 3 种方法。动态度量方法就是直接运行程序以测量(measure)程序的执行时间，目前使用的有随机方法、基于遗传算法的进化方法(Evolution Method)、模拟退火方法和统计方法。动态度量方法很难保证所得到结果是安全的，尤其是对现代高性能处理器。

静态分析方法根据程序的流信息，针对运行程序的处理器特性估算出程序的 WCET。因为程序的流信息通常是非常复杂的，而处理器尤其是现代处理器(比如高速缓存和超级流水线)的特性也很复杂，所以静态分析和计算也会变得非常复杂。但静态分析方法能够保证得到的结果是安全的，而且能够不运行程序就获得结果，从而成为 WCET 分析研究的主流[1]。

混合方法就是既包括静态分析也包括动态度量的方法。该方法或者首先对程序进行分析，根据分析结果进行测试，或者先度量，然后在度量的基础上静态计算程序的 WCET 值。

本文研究 WCET 静态分析，并在下文简写为 WCET 分析。

2　WCET 分析的定义和组成

2.1　WCET 分析的定义

公认的 WCET 分析的定义是由 P. Puschner 和 A. Burns 给出的[1]：

WCET 分析是指计算给定应用程序代码片断执行时间的上限，这里代码片断执行时间定义为执行代码片断所花费

*)国家自然科学基金(No. 60303013)资助项目。**姬孟洛**　博士生，研究方向：实时系统分析、面向对象设计；**齐治昌**　教授，研究方向：软件工程、计算机教育。

的处理器时间。

从上述定义能够看出：①WCET 的分析结果是实际 WCET 值的上界，这里不要求完全精确；②分析得到的时间是运行程序占有的处理器时，这里不考虑上下文切换引起的时间，也没有考虑整个应用程序甚至系统的 WCET，考虑的只是不间断运行一个程序所占用的处理器时间；③这个时间显然和程序代码片断以及处理器特征有关。

2.2 WCET 分析的组成

一般认为 WCET 分析包括 3 个组成部分[3]：①程序流事实分析；②执行时间模型的建立；③基于前两项信息的 WCET 计算。

为了得到精确的 WCET 分析值，需要获得程序流事实信息。流事实信息通常包括循环的最大迭代次数、递归调用的最大深度、不可行路径(infeasible path)和其它的程序流约束信息，其中不可行路径是指对任意输入数据都不可能执行的程序路径。在程序流信息中，对 WCET 分析影响较大的是循环的迭代次数和不可行路径。

要获取程序的 WCET，还需要对程序的目标代码进行分析以获得实际的时间，即建立执行时间模型，此分析也称为低层分析。对于不带 cache、没有流水线的传统 CISC 指令，其执行时间是固定的，此时低层分析就非常简单：一个代码段的执行时间就是其所对应的每条指令执行时间的累加。但对于带有高速缓存和流水线以及分支预测机制等特征的现代处理器，代码段执行时间的估算就复杂得多。高速缓存对整个程序段甚至是整个系统的执行时间都有影响，因此它是全局性的，而流水线对相邻的几条指令的执行时间有影响，因此它是局部性的。

流水线分析的典型方法是利用保留表[4]，这也是处理器体系流水线分析的常用方法。两条指令的执行时间就是它们对应的两个保留表连接之后，从第一条指令第一个阶(stage)到最后一条指令的最后一个阶之间的周期(cycle)数。

典型的高速缓存的分析方法是把每条指令分为不同的访问类型，根据不同的访问类型计算缓存引起的 WCET。Mueller[5]把指令的访问类型分为 4 类：always miss、always hit、first miss 和 first hit，分别表示总是丢失、总是命中、第一次丢失和第一次命中。

计算的目的在于：在给定程序流事实和执行时间模型的情况下，为程序计算 WCET 估值。计算方法可以分为 3 大类[3]：基于路径的(path-based)、基于树的(tree-based)和隐含路径枚举的(IPET，Implicit Path Enumeration Technique)方法。

在基于树的计算[6]中，使用为每种类型的复合程序语句定义的规则确定语句的 WCET，然后通过自底向上遍历程序的语法分析树产生整个程序的 WCET 估值。此方法称为 Time schema。Time schema 方法概念简单，计算容易，但难以考虑语句间的依赖性。

在基于路径的计算[3~5,7]中，通过计算程序中不同路径的时间，然后查找具有最长执行时间的路径，产生 WCET 估值。此方法的特性是，可能的执行路径是明确表示或确定的。在单个循环中基于路径的方法是很自然的，但当跨越循环嵌套层次时流事实信息表示会比较困难。

IPET 是由 Li 等[8]提出的，首先应用于指令执行时间不变的简单处理器时间模型中，后又将此方法应用到流水线和高速缓存的计算。

IPET 计算使用代数'和'或者'或'逻辑约束表示程序流和执行时间。为程序中的每个基本块和程序流边赋一个时间变量(t_{entity})和一个计数变量(x_{entity})，t_{entity} 表示该块或边的执行时间，x_{entity} 表示该块或边的执行次数。在满足程序的结构和可能流的约束情况下，通过最大化下列目标函数(cost function)找到 WCET：

$$\sum_{i\in entity} x_i * t_i$$

IPET 约束系统既可以用整数线性规划技术解决(ILP)，也可以用约束解决技术 CSP 解决。由于有可用的有效 ILP 解决器，ILP 成为最流行的计算方法。约束解决技术允许表达更复杂的约束，但一般会花费更长的时间。

需要说明的是，WCET 分析的 3 个组成部分是相互依赖、相互影响的。举例来说，执行时间模型的不同层次对流事实的要求是不同的。

3 程序流事实分析

程序流事实用来确定程序可能的执行路径，也就是程序的动态行为。流事实分析的结果是诸如哪些函数得到调用、循环的迭代次数、不同的 if 语句之间的依赖关系等信息。因为此问题一般是计算处理，所以通常都进行简化和近似的分析。但此分析必须遵从安全的路径信息，即要考虑到所有的可行路径。

流事实信息分析包括手工标注和自动分析两种。手工标注是用户通过与 WCET 工具的交互，给源程序代码添加标注(annotation)，或者在程序代码外面给出需要的信息。比如，Kirner 等[9]通过用额外的语法来扩充 C 语言以定义循环的界限和路径信息，从而允许流信息进入源程序代码。Börjesson[10]使用 C 代码中的 # pragmas 添加标注。

手工标注对程序员来说是个很大的负担，因为有时即使是一个很小的程序想标注出来也是困难的，同时手工标注容易出错。因此，最好能够通过自动流分析获取需要的信息。

自动流信息分析通过分析程序代码来获取流信息，自动流分析通常获得循环的执行界限和循环中的不可行路径。下面介绍几种程序流事实分析方法。

3.1 直接计算

Healy 等[11]分析编译后的目标代码，他针对依赖于归纳变量的循环使用特定的算法，自动获取循环的界限。其分析包括 4 步：(1)标识循环中影响循环执行次数的条件分支(称为迭代分支)。(2)对每一个迭代分支，计算什么时候每个被标识分支能够影响改变自身的结果，其计算公式为：

$$N_i=\left\lfloor \frac{limit_i-(initial_i+before_i)+adjust_i}{before_i+after_i} \right\rfloor+2$$

其中，$limit$ 是对循环变量 i 进行比较的值；$initial$ 是循环变量进入循环体之前的值；$before$ 是循环测试分支之前改变的值；$after$ 是循环测试分支之后改变的值；$adjust$ 的值是 0 或 1，用来补偿运算符之间的差异(比如≤和<)。(3)确定这些分支能够到达的循环迭代的范围。(4)计算循环迭代的最小和最大次数。

对于非正交(non-rectangular)多重循环嵌套(内层循环次数依赖于外层循环变量)，当一个循环的跨度 $strides=before+after=1$ 时，其执行次数为：

$$I=\sum_{i=a}^{b}1=\begin{cases} b-a+1 & \text{if } a\leqslant b \\ 0 & \text{otherwise} \end{cases}$$

由此公式计算循环的执行次数。比如下面是 LU 分解程

序的循环嵌套结构：

for ($j=1;j\leqslant 100;j++$)

for ($i=j;i\leqslant 100;i++$)

for ($k=1;k<j;k++$) A;

此多重循环的执行次数为：

$$N=\sum_{j=1}^{100}\sum_{i=j}^{100}\sum_{k=1}^{i-1}1$$
$$=\sum_{j=1}^{100}\sum_{i=j}^{100}(j-1)$$
$$=\sum_{j=1}^{100}(\sum_{i=1}^{100}(j-1)-\sum_{i=1}^{j-1}(j-1))$$
$$=\sum_{j=1}^{100}(102j-j^2-101)=166650$$

当包括多重循环嵌套的跨度为非单位变化时，Healy[11]给出了相应的计算公式，该计算比较复杂。

与 Healy 方法类似，Holsti 等[12]在目标代码层使用 Presburger Arithmetic 计算基于计数循环的循环界限，Presburger Arithmetic 是指整数算术的子集。

同样，对于非正交多重循环，Colin 等[13]使用一组数学表达式[*Maxiter*,*counter*]对每一个循环进行标注，其中 *Maxiter* 是循环的最大数目，counter 是循环计数值。标注之后，使用外部工具 Maple 对数学表达式进行符号计值，以计算内层循环的数目。

根据编译器产生的基本块中赋值对控制变量的作用(包括谓词和迭代范围)以及分支谓词与分支谓词之间的关系，文[14]自动确定循环中的不可行路径、循环中路径的执行范围和执行次数。

3.2 抽象解释

抽象解释[15]是一种形式化的程序分析方法。Gustafsson 和 Ermedahl[16]使用抽象解释对源代码进行自动分析以获取程序流信息。他们把程序中改变变量值的位置称为控制点 i，把该点所有变量 a 的可能取值组合称为环境 σ，即

$$\sigma_i^h=\{a_1\mapsto v_1,\cdots,a_m\mapsto v_m\}$$

这里 h 是指循环中的第 h 次迭代，v 是变量 a 可能的取值或取值范围。

对条件语句，把环境按照条件分为两个环境：

$$S[\![\text{if}(C)S_1\ \text{else}\ S_2]\!]\sigma=\{\sigma_2,\sigma_4\}$$

其中，$\sigma_1=C[C]\sigma$，$\sigma_2=S[S_1]\sigma_1$，$\sigma_3=C[\neg C]\sigma$，$\sigma_4=S[S_2]\sigma_3$。

$C[\cdot]$是条件语义函数，$S[\cdot]$是条件语句函数。语义函数根据依赖的规则，把一个环境修改为另一个环境。比如，$\sigma(x)=0..10$，令 $\sigma_1=C[x<4]\sigma$，则 $\sigma(x)=0..3$；令 $\sigma_3=C[\neg(x<4)]\sigma$，则 $\sigma(x)=4..10$；令 $\sigma_1=S[x=x+2]\sigma$，则 $\sigma(x)=2..12$。

根据操作语义展开循环计算循环的迭代次数：

$$S[\![\text{while}(C)S]\!]\sigma=S[\![\text{if}(C)\{S;\text{while}\ (C)\ S\}]\!]\sigma$$

当存在一个变量，该变量在一个环境下其值为空，则说明该环境对应的路径为不可行路径。

因为环境的个数会随着分支和循环呈指数级增长，为了减少环境的个数，该方法在循环尾部(出口之前或者回到循环头节点之前)合并环境。

Gustafsson 和 Ermedahl 的方法虽然能够根据程序语义获取循环的最大和最小执行次数，标识不可行路径，甚至判断无法执行的代码(dead code)等其他信息，但该分析方法无法确定分析终止。

Lisper[17]使用文[18]的抽象解释导出程序流图节点的执行计数以及路径的线性约束。其针对的是整数型变量，其抽象状态表示的是空间中的多面体(Polyhedral)，空间的坐标对应于不同的程序变量的值。也即，抽象状态是用表示的多面体 P，多面体等价于系统的线性不等式。

令$\sqcup$和$\sqcap$表示多面体的并和交操作，$proj(P,x_i)$表示包含 P 的删除有关 x_i 所有信息的最小多面体，则赋值节点 $x_i:=e$ 的抽象迁移函数[15]有 3 种情况：

(1)e 为非线性：$f(P)=proj(P,x_i)$；

(2)$e=\vec{l}*\vec{x}$且 $l_i\neq 0$：

$f(P)=P[x_i\leftarrow(x_i-\sum_{j\neq i}l_jx_j-b)/l_i]$，这里 $P[x_i\leftarrow t]$是由 t 替换 x_i 产生的多面体；

(3)$e=\vec{l}\cdot\vec{x}$且 $l_i=0$：

$f(P)=proj(P,x_i)\wedge(x_i=e)$

对条件表达式节点 c，其对应于 true 和 false 分支的迁移函数 f_{true} 和 f_{false} 为：

(1)c 为非线性：

$f_{true}(P)=f_{flase}(P)=P$；

(2)$c\equiv\vec{l}*\vec{x}=b$：

如果 $P\sqsubseteq(\vec{l}*\vec{x}=b)$，则 $f_{true}(P)=P$ 且 $f_{false}(P)$无定义；否则 $f_{true}(P)=P\sqcap(\vec{l}*\vec{x}=b)$且 $f_{false}(P)=P$；

(3)$c\equiv\vec{l}*\vec{x}\leqslant b$：

$f_{true}(P)=P\sqcap(\vec{l}*\vec{x}\leqslant b)$且

$f_{false}=P\sqcap(\vec{l}*\vec{x}>b)$。

对于控制流图[19]中具有多个入口的节点，其入口的迁移函数为：

$$f(P_1,P_2)=P_1\sqcup P_2$$

根据上述迁移函数，并使用加宽操作[15]，能够建立每个节点的迭代方程，求解方程，能够得到每个节点的多面体。根据多面体中变量定义的元素个数，确定节点的循环迭代次数。同样，根据程序路径[19]对应的控制流图，确定任意两个节点的约束。

Corti[20]也使用抽象解释分析循环中线性路径，即从循环头[19]节点开始到辅助节点 continue 或 break 终止(且非处于同一循环迭代次数的)的路径，以便发现循环中的不可行路径。

3.3 一般抽象

Schuele[21]使用带时间标记的有限自动机为系统建模，通过寻找初始状态与终止状态之间最短路径来确定汇编程序的 WCET。为了减少计算复杂度，使用程序切片技术删除与控制流中与控制转移无关的指令，从而加快 WCET 分析[22]。

3.4 mode 和 Scenario

在一定的输入变量范围内，程序往往会执行某种固定的轨迹，这种固定的轨迹称为程序的模式(mode)。Chapman 等人[23,24]通过人工标注的形式，在程序中插入标记，把程序分成不同的模式。针对指定的模式，按照程序的语法树结构(Time schema)，计算其 WCET 值。

同样针对 Time schema，文[25]研究了情节(Scenario)。一个情节是输入数据的特定类型定义的程序行为。情节是模式的特例。自动发现情节步骤包括 5 步：

(1)确定可能影响程序执行时间的参数。检查那些在程序开始读取数据，在程序剩余部分保持不变的程序参数。这些参数只有一个地方改变参数的值；

(2)计算这些参数对程序 WCET 的最大影响。这些参数 v 的影响用影响系数 IC(influence coefficient)表示。$IC(v)$表示的是参数 v 的不同值所能引起的程序 WCET 的最大变化。

通过后序遍历程序的抽象语法树计算每个语句 s 的 IC

(v)：

$$IC_s(v)=contribution(s)+\max_{x\in successors(s)}(IC_x(v))$$

语句 s 的 $IC(v)$包括两部分：一个是语句 s 的贡献，一个是语句 s 后续语句的 IC。因为是后序遍历程序，所以后续语句的 IC 总是已知的。

如果 v 出现在 if 语句的条件表达式中，则 if 语句的贡献是两个分支语句的 WCET 之差。如果是循环语句，则该语句的贡献是循环体的第一个语句乘以循环的迭代次数。

(3)根据这些参数及其影响，按照情节对应用程序进行划分；

(4)产生针对每个情节的源码，利用 Time schema 估算其 WCET；

(5)使用下面公式计算整个程序的 WCET：

$$WCET(app)=\max_{S\in Scenarios}(WCET(S))$$

结束语 WCET 分析经过 10 多年的发展，不仅在学术界具有很大的影响，而且逐渐应用于工业界。

程序流事实分析的主要方法是直接计算和抽象解释。直接计算方法虽然简单、直接，但不完备。Gustafsson 和 Ermedahl[16] 的抽象解释方法虽然能够极大限度地获得流事实信息，但无法确定分析的终止。Lisper[17] 的抽象解释方法利用抽象状态中变量包含的元素个数来确定循环迭代次数和基本块之间的约束，但没有给出确定变量元素的方法。

程序模式的 WCET 分析能够提高 WCET 分析的精度，但其自动分析缺少更通用的理论和方法。

目前 WCET 分析的研究热点包括：新处理器特性分析、提高 WCET 分析精度以及 WCET 分析和其他研究领域结合。

参 考 文 献

1 Puschner P, Burns A. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. Real-Tim Systems, 2000, 18(2-3): 115～128

2 Klingerman E, Stoyenko A D. Real-Time Euclid: A Language for Reliable Real-Time Systems. IEEE Transactions on Software Engineering, 1986, 12(9):941～989

3 Engblom J. Processor Pipelines and Static Worst-Case Execution Time Analysis:[PhD thesis]. Uppsala, Sweden: Acta Universitatis Upsaliensis, 2002

4 Healy C A, Arnold R D, Mueller F, et al. Bounding Pipeline and Instruction Cache Performance. IEEE Transactions on Computers, 1999,48(1): 53～70

5 Arnold R D, Mueller F, Whalley D, et al. Bounding Worst-Case Instruction Cache Performance. In: Proc. 15th Real-Time Systems Symposium (RTSS), Brookline, Massachusetts, Dec. 1994. 172～181

6 Shaw A C. Reasoning about time in higher level language software. IEEE Transactions on Software Engineering, 1989,15(7): 875～889

7 Stappert F, Ermedahl A, Engblom J. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems, Atlanta, Georgia, USA, 2001

8 Li Y T S, Malik S. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In: Proc. 32nd ACM/IEEE Design Automation Conference, Jun. 1995

9 Kirner R. The Programming Language wcetC: [Technique report]. Vienna, Austria: Technische Universitat Wien, 2002

10 Borjesson H. Incorporating worst-case execution time in a commercial compilers:[PhD thesis]. Uppsala, Sweden: Uppsala University, 1995

11 Healy C A, Sjödin M, Rustagi V, et al. Supporting timing analysis by automatic bounding of loop iterations. Real-Time Systems, May 2000. 121～148

12 Holsti N, Långbacka T, Saarinen S. Worst-Case Execution-Time Analysis for Digital Signal Processors. In: Proceedings of the X European Signal Processing Conference, Sept. 2000

13 Colin A, Puaut I. Worst case execution time analysis for a processor with branch prediction. Real-Time Systems, 2000,18(2): 249～274

14 Healy C A, Whalley D B. Automatic Detection and Exploitation of Branch Constraints for Timing Analysis. IEEE Transactions on Software Engineering, August 2000. 763～781

15 Cousot P, Cousot R. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM Symposium on Principles of Programming Languages, 1977. 238～252

16 Ermedahl A, Gustafsson J. Deriving Annotations for Tight Calculation of Execution Time. In: Lengauer C, Griebl M, Gorlatch S, eds. 3th Int Euro-Par Conference. Passau Germany: 1997. 26～29

17 Lisper B. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In: 3rd Intl Workshop on Worst-Case Exexution Time (WCET) Analysis, Porto, 2003

18 Cousot P, Halbwachs N. Automatic discovery of linear restraints among variables of a program. In: Proc of the 5th ACM Symposium on Principles of Programming Languages, 1978

19 Aho AV, Sethi R, Ullman JD. Compilers, Principles, Techniques, and Tools. New York: Addison-Wesley, 1997

20 Corti M, Gross T. Approximation of the Worst-Case Execution Time Using Structural Analysis. In: 4th ACM International Conference on Embedded Software, Pisa, Sep. 2004

21 Schüle T, Schneider K. Exact runtime analysis using automata-based symbolic simulation. Formal Methods and Models for Codesign, Mont Saint-Michel, France, IEEE Computer Society, 2003

22 Schüle T, Schnerder K. Abstraction of assembler programs for symbolic worst case execution time analysis. In: Proc of the 41th Annual Conference on Design Automation, San Diego, CA, USA, 2004

23 Chapman R, Burns A, Wellings A. Integrated program proof and worst-case timing analysis of spark ada. In: Proc. ACM Workshop on Language, Compiler and Tool Support for Real-time Systems, Jun 1994

24 Bernat G, Burns A. An approach to symbolic worst-case execution time analysis. In: 25th IFAC Workshop on Real-Time Programming. Palma (Spain), 2000

25 Gheorghita S V, Stuijk S, Basten T, et al. Automatic Scenario Detection for Improved WCET Estimation. In: 42th ACM/IEEE Design Automation Conference (DAC), Anaheim, California, USA, 2005