博 士 学 位 论 文

实时系统最坏情况执行时间分析技术的研究

导　师：于　戈　教授

邓庆绪　教授

研究生：吕鸣松

东北大學

二 OO 九年十一月

分类号_____密　级_____

U D C _____

学　位　论　文

实时系统最坏情况执行时间分析技术的研究

作 者 姓 名：　吕鸣松

指 导 教 师：　于　戈　教授

　　邓庆绪　教授

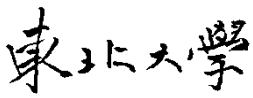
申请学位级别：博　士　　学 科 类 别 ：　工　学

学科专业名称：嵌入式系统及应用

论文提交日期：2009 年 11 月 2 日　论文答辩日期：　2010 年 1 月 3 日

学位授予日期：　　答辩委员会主席：

评　阅　人：



2009 年 11 月

A Dissertation for the Degree of Doctor in Embedded Systems and Applications

Research on Worst-Case Execution Time Analysis Techniques of Real-Time Systems

by Lv Mingsong

Supervisor: Professor Yu Ge

Professor Deng Qingxu

Northeastern University

November 2009

独创性声明

本人声明，所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人已经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：

日　期　：

学位论文版权使用授权书

本学位论文作者和指导教师完全了解东北大学有关保留、使用学位论文的规定：即学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人同意东北大学可以将学位论文的全部或部分内容编入有关数据库进行检索、交流。

作者和导师同意网上交流的时间为作者获得学位后：

半年 □　　一年 □　　一年半 □　　两年 □

学位论文作者签名：　　导师签名：

签字日期：　　签字日期：

实时系统最坏情况执行时间分析技术的研究

摘 要

实时系统时间验证主要包括可调度性分析和最坏情况执行时间（WCET）分析。其中 WCET 分析的目的是计算实时任务在最坏情况下的执行时间，其结果是可调度性分析的重要输入。在硬实时系统中，为保证分析结果的安全性，通常采用静态方法分析程序的 WCET。静态分析主要包括处理器行为分析、程序流分析和 WCET 计算三个子任务。目前在 WCET 计算技术方面，隐式路径枚举技术是主导技术，但是该技术的主要问题是描述复杂程序控制流程信息的能力很有限，进而也限制了处理器行为分析所能采用的技术。在处理器行为分析方面，Cache 分析一直是一个难度较大的任务。目前基于抽象解释技术的分析方法在分析 LRU 替换策略中占有主导地位，但是对于基于 FIFO 等其他替换策略的 Cache 行为的分析尚不成熟。随着处理器体系结构向多核发展，共享 Cache 成为了主要的设计趋势之一。不同核心上的程序在共享 Cache 中相互干涉，给 Cache 分析带来了极大的挑战。此外，随着 WCET 分析技术的不断发展，相关技术在实际系统中的可用性问题也逐渐受到越来越多的关注。

基于上述研究现状，本文针对 WCET 静态分析中的 WCET 计算、单核与多核 Cache 行为分析、实时操作系统 WCET 分析等课题进行了深入研究。主要工作包括如下几点：

（1）提出了一种全新的基于模型检测技术的 WCET 计算方法，设计了从程序到对应自动机模型的转换语义。该方法作为静态 WCET 分析的基础性框架，充分利用了模型检测技术搜索最优解的能力，使得分析结果具有更高的精度。探索了采用不同工作原理的模型检测器用于 WCET 计算的时间可伸缩性和空间可伸缩性，给出了基于模型检测技术的 WCET 计算方法的适用范围。

（2）提出了一种基于剪枝思想的单核系统 Cache 分析方法。在基于模型检测技术的 WCET 计算框架的基础上，研究了利用这一技术对单核系统中采用 FIFO 替换策略的 Cache 进行分析的方法。针对模型检测技术在 WCET 分析中可能出现的状态空间爆炸问题，提出了基于剪枝思想的分析方法。该方法通过削减程序模型中符合特定条件的程序分支，能够在不损失分析精度的前提下，有效提高分析的性能以及可伸缩性。

（3）提出了一种面向多核共享 Cache 的处理器行为分析方法。分析多核系统的共享 Cache，关键在于对程序间的干涉情况进行精确的分析。现有的方法在分析这一问题的过程中只考虑了不同核心上的程序在地址上的冲突情况，导致分析结果的过度估计过高。本文采用模型检测技术对多核共享 Cache 的行为进行建模，模型能够在更细的粒度

上挖掘冲突发生的时间关系，从而大大提高了分析的精确性。

（4）对工业界广泛使用的 μC/OS-II 实时操作系统进行了 WCET 分析。实时操作系统以控制密集型代码为主，其代码结构和时间行为特性与普通应用程序有很大区别。本文对 μC/OS-II 实时操作系统的系统调用和禁止中断区间进行了静态 WCET 分析，详细分析了传统技术在分析实时操作系统代码时的精度以及尚存在的主要问题。同时，实验结果本身就是对 μC/OS-II 实时操作系统实时特性的一个完整的量化描述，它对于使用者了解该系统的时间特性，以及对于系统开发者改善系统的实时性能都具有重要的应用价值。

（5）本文在新加坡国立大学开发的 Chronos 工具的基础上，设计并实现了一个支持多核体系结构的静态 WCET 分析工具，实现了本文所提出的基于模型检测技术的 WCET 计算方法、基于 FIFO 替换算法的单核 Cache 行为分析、面向多核共享 Cache 的处理器行为分析、实时操作系统 WCET 分析等具体技术和方法，并验证了这些方法的正确性和有效性。

总之，本文研究了采用模型检测技术进行静态 WCET 分析的问题，实验表明采用这种技术能够有效的提高静态分析的精确性。同时还探讨了采用静态 WCET 分析方法分析实时操作系统的可行性及主要问题。本文的研究对于静态 WCET 分析以及 WCET 分析可用性等领域的研究具有一定的参考价值。

关键词：实时系统；WCET 分析；静态分析；模型检测；Cache 分析；多核；μC/OS-II

Research on Worst-Case Execution Time Analysis Techniques of Real-Time Systems

Abstract

Schedulability analysis and Worst-Case Execution Time (WCET) analysis are two major tasks of timing validation of real-time systems. The purpose of WCET analysis is to obtain the execution time of a real-time task in the worst-case, and its results serve as the input to schedulability analysis. Static analysis, which includes processor behavior analysis, flow analysis, and WCET calculation, is usually adopted to guarantee safe results in hard real-time systems. Currently Implicit Path Enumeration is the dominating technique in WCET calculation, but the drawback of this technique is limited expressiveness in describing program control flow, which further limits the techniques used to conduct processor behavior analysis. Cache analysis has always been a hard task in processor behavior analysis. Abstract Interpretation based methods have gained predominance in analyzing caches with LRU replacement policy, but analysis methods for other policies, such as FIFO, are still immature. With the trend of multicore processors, shared cache becomes a prevalent Cache architecture. Fine-grained sharing of caches among different cores makes the timing behavior of tasks unpredictable, which brings great challenges to WCET analysis. With the development of WCET analysis techniques, usability issues have also become a major concern.

Given the above research background, we conducted research on WCET calculation, single-core and multi-core Cache analysis, and WCET analysis of real-time operating systems in this thesis. Detailed contributions are as follows:

(1) A new WCET calculation method based on model checking was developed, and the transformation semantics from a program to the corresponding automaton was designed. The ability of searching optimal solutions of model checking is leveraged in WCET calculation for better analysis precision. We used different model checkers in the analysis with a discussion on the time and space scalability issues. Usage guidelines of WCET analysis based on model checking were also given.

(2) A single-core cache analysis method based on the idea of branch cutting was put forward in this thesis. First, model checking was used to model the behavior of FIFO caches;

then a method that implements the idea of branch cutting was designed to tackle the state space explosion problem in WCET analysis based on model checking. By cutting off the program branches that satisfy specific criteria, the method can efficiently improve the analysis performance and scalability without sacrificing analysis precision.

(3) A model checking based method to analyze shared caches in multi-cores was designed in this thesis. The key point of analyzing shared caches is to precisely analyze the inter-core interferences. Existing techniques only take address-related conflicts into account, thus result in large overestimation. We used model checker to model shared cache behaviors, and experiment results show that the analysis precision is greatly improved, since the timing order of cache conflicts are captured in the models.

(4) We also conducted research on WCET analysis of the μC/OS-II real-time operating system (RTOS). RTOS code is control-intensive, which makes them different from application code in terms of timing properties. The system calls and disabled interrupt regions of μC/OS-II are analyzed using static methods, and the analysis precision and existing problems in static WCET analysis of RTOS are analyzed intensively. The results of the analysis can directly serve as a comprehensive and quantitative description of the timing properties of μC/OS-II, which is helpful to RTOS users to understand the real-time performance of the operating system, and can provide useful information for system designers in improving system performance as well.

(5) A static WCET analysis tool that supports multi-core architecture was developed based on Chronos – a WCET analysis prototype developed by National University of Singapore. Techniques on WCET calculation based on model checking, single-core cache analysis with branch cutting, multi-core shared cache analysis, and static analysis of RTOS are implemented in this tool to verify the techniques designed in this thesis.

To summarize, research on static WCET analysis based on model checking is conducted in this thesis, and experiment results show that the techniques can achieve better precision. Experience and problems in static analysis of RTOS are also explored. The work of this thesis could provide reference value to researches on static WCET analysis and usability aspects of WCET analysis.

Keywords: real-time systems; WCET analysis; static analysis; model checking; Cache analysis; multi-core; C/OS-II

目 录

第1章 绪 论

1.1 研究背景及意义

嵌入式系统是指被嵌入到具体的产品中的特殊计算机系统，用于完成数据处理、通信、监控等功能[1]。嵌入式系统应用广泛，大量存在于航空军事、工业控制、汽车电子、网络基础设施、网络安全、无线通信基础设施、数字医疗和个人电子等应用领域。大多数嵌入式系统的共性是：系统功能的执行以及与外部环境的交互必须满足某种时间要求，我们通常称具有这种属性的系统为实时系统。一个实时系统的正确性不仅仅取决于系统逻辑运算的正确性，同时取决于得到运算结果的时间是否满足特定的要求。

根据对时间要求紧迫程度的不同，可以将实时特性划分为硬实时和软实时两种[2]。硬实时系统不允许任何不满足时间特性的情况出现，一旦出现将会导致灾难性的后果。通常关键任务系统（Mission-Critical Systems）都是硬实时系统，例如汽车控制系统、航空控制系统、复杂医疗系统以及武器系统等。与硬实时系统对应的是软实时系统，在软实时系统中，即使时间要求在一定范围内不被满足，系统仍能够正常运行。多媒体系统就是一种典型的软实时系统，例如在视频播放的过程中，少数几帧的延迟可以被容忍，且不会导致系统瘫痪。

由于硬实时系统对于执行时间的要求非常苛刻，因此在设计此类系统时，通常要在系统实际运行之前对系统的时间特性进行完整的验证，以确保系统在运行过程中规定的时间特性不会被破坏。这一工作被称为时间特性验证[3]（Timing Validation）。我们通常用截止期（Deadline）来描述任务的时间特性。所谓截止期是一个规定的时间点，硬实时系统要求所有的任务必须在这个时间点之前执行完毕，否则将会出现灾难性后果。因此，时间特性验证的主要工作就是在系统实际执行之前分析所有的任务是否都满足截止期要求。由于几乎所有的实时系统都是多任务系统，多个任务要分时的共享系统的计算资源，因此需要对任务进行调度。不同的调度算法会直接影响任务完成的时间，所以系统分析者必须确保系统所采用的调度算法能够调度目标任务集，而不出现任何一个任务不满足截止期要求的情况，通常称这种分析为可调度性分析[4]（Schedulability Analysis）。对任务集进行可调度性分析，要求必须事先知道每个任务在最坏情况下的执行时间，我们称之为任务的最坏情况执行时间（WCET）。得到任务 WCET 的过程称为最坏情况执行时间分析[5]（WCET Analysis）。WCET 分析和可调度性分析是实时系统时间特性验证的两个核心任务，是实时系统研究的两大基础。同时，WCET 分析又是可调度性分析的

基础，它为后者提供分析所需要的任务执行时间信息。本文的重点就是研究实时系统 WCET 分析中的关键技术。

由于受程序流程、处理器功能特性以及执行环境等因素的影响，一个任务的执行时间通常是不确定的，且客观上呈现出某种分布[5]。我们把一个任务所有可能的执行时间中的最大值称为这个任务的“最坏情况执行时间（WCET）”，把最小值称为这个任务的“最好情况执行时间（BCET）”。为获取一段程序的 WCET，最直观的方法就是实际执行待分析程序，记录程序的执行时间（称之为“执行时间观测值”）。通过多次执行，可以获得不同执行环境下的多个执行时间观测值，取其最大值作为程序的 WCET，这种分析 WCET 的方法称为动态 WCET 分析。其特点是直观、简单，但是主要问题是多次执行并不能保证覆盖程序执行的最坏情况，也就是说，执行时间观测值的最大值可能小于程序的实际 WCET 值。这在硬实时系统中是不允许的。为此，可以采用某些数学手段，在不实际执行程序的情况下，对程序的可执行代码进行离线分析，通过计算得到程序的 WCET 值（称之为“执行时间估计值”），这种分析方法叫做静态 WCET 分析。由于静态 WCET 分析可以用数学手段证明其正确性，因此可以保证分析得到的执行时间一定不会小于程序的实际 WCET 值。执行时间观测值、执行时间估计值、及程序的实际 WCET 值的关系见图 1.1。

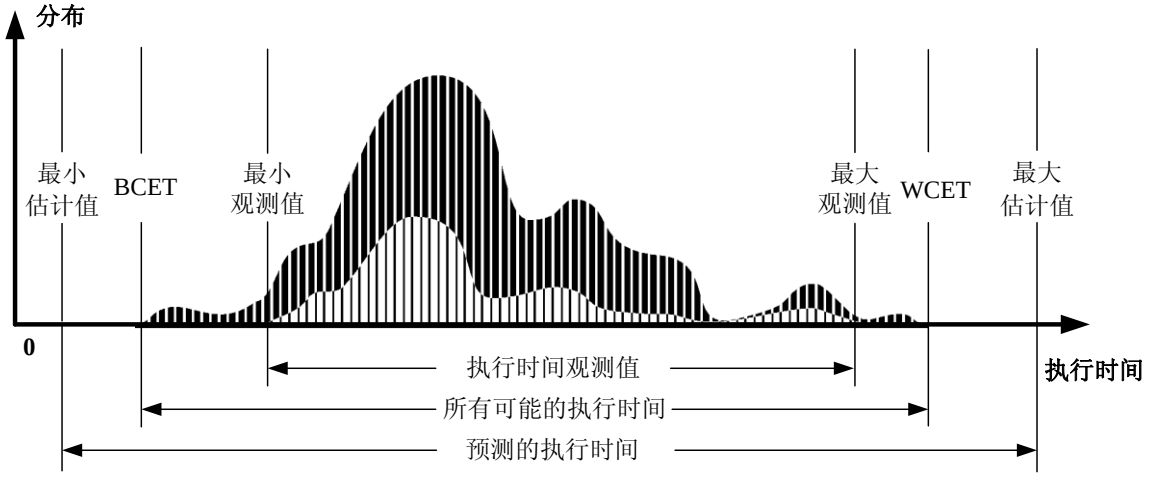


图 1.1 程序执行时间的基本概念[5]

Fig. 1.1 Basic concepts of execution times of programs [5]

如果分析得到的执行时间大于程序的实际 WCET 值，我们称这种分析结果是“安全（Safe）”的。分析结果越接近实际的 WCET 值，我们称其越“精确（Accurate）”。硬实时系统的 WCET 分析要求分析结果必须是安全的，因此通常采用静态分析的方法。在实际系统中，程序的最坏情况执行时间是很多因素综合作用的结果，影响程序 WCET 的主要因素有三类：第一类是程序的流程，包括分支、循环、函数调用等；第二类是处

理器特性，常见的流水线、Cache、分支预测等处理器功能部件将会直接影响任务的执行时间；第三类是执行环境，包括操作系统环境、任务的参数输入等。由于这种复杂性的存在，静态分析程序的 WCET 就成为了一项非常具有挑战性的课题。

静态 WCET 分析的关键技术主要包括三方面主要内容：

（1）处理器行为分析。该分析的主要目的是获得指令或者局部程序在特定处理器上的执行时间。由于应用需求不断提升，日益复杂的系统设计对系统性能提出了越来越高的要求，因此很多用于桌面系统的处理器特性正越来越多的被使用到实时系统中。复杂处理器特性的出现，导致了指令在处理器中的执行时间越来越不确定，这就给 WCET 分析造成了极大的困难。

（2）程序流分析。程序流程也是影响程序 WCET 的直接因素。程序流分析的主要目的是解析程序的流程结构，分析程序的不可行路径，分析程序的循环信息等。只有清楚了程序的流程信息，才可能计算出程序在最坏情况下的执行时间。

（3）WCET 计算。在获得了程序的流程信息，同时能够计算出局部程序在特定处理器上的执行时间的情况下，需要采用某种技术，计算出导致程序执行时间最长的执行路径，并求得这条路径上程序的执行时间。采用何种 WCET 计算技术，将直接影响到程序流分析和处理器行为分析，因此 WCET 计算是三个关键技术中的基础性技术。

WCET 分析是一个应用性较强的研究领域，我们除了要关注相关的理论成果，同时也应该考虑 WCET 分析在实际系统中的可用性问题[6]。传统的研究主要集中在理论层面，对于相关技术的评价也大多是通过分析简单的小程序而获得的。现有的理论和方法应用于实际系统，尤其应用于实时操作系统等特殊程序的实际效果尚待更进一步的实践和探讨。本领域学者在以上方向上已经展开了相关研究工作，并取得了一定的进展，也解决了其中的一些问题。但是在某些关键环节上，依然存在很多具有挑战性的问题等待解决。本文结合现有研究成果和基础，在 WCET 计算、处理器行为分析、实时操作系统 WCET 分析等关键技术上开展研究工作。

1.2 国内外研究现状

在 WCET 计算关键技术方面，目前的研究工作主要集中在基于语法树的 WCET 计算技术和基于隐式路径枚举的 WCET 计算方法，此外还有基于路径的 WCET 计算方法。基于语法树的计算技术首先由 Shaw 和 Park 等学者提出[7-9]，这种技术的基本思想是将待分析程序结构化的表示为一棵语法树，树的叶子节点表示程序的某一部分的执行时间，通过从树的最底层依次向上规约，最终得到整个程序的执行时间。

隐式路径枚举技术由 Li 等学者首先提出[10,11]。之所以称为“隐式”路径枚举，是因

为这种技术并不是显式的枚举出所有的程序路径，然后寻找具有最长执行时间的路径。它的核心思想是“将 WCET 的求解问题，或者说最长执行时间路径搜索问题，转化为求解 CFG 中的每个基本块的最大执行次数的问题”。这一问题可以利用整数线性规划（ILP）进行建模和求解。隐式路径枚举技术在实际应用中表现出了比较好的求解效率，因此被很多静态 WCET 分析工具所采用[12-15]，目前已经成为 WCET 应用领域的一个主流技术。

在基于路径的计算技术方面，Stappert 等学者开展了相关研究[16]。他们为程序建立一个 Scope Graph，它能够以层次化的方法描述程序结构。循环所对应的每一个 scope 都可以被展开成为若干个 virtual scope，最长执行路径就是在这些 virtual scope 中搜索得到的。整个程序的 WCET 的计算类似于基于语法树的计算方法。Chapman 等学者开发了一个名为 SPATS 的系统，该系统可以将程序证明和 WCET 分析通过程序的 CFG 集成在一起[17-19]。他们提出了一种从 CFG 向 WCET 转化的算法，这种算法可以将一个带环的 CFG 图转化为路径表达式，通过分析路径表达式来求解程序的 WCET。

处理器行为分析受制于 WCET 计算方法，因此主要有基于语法树的分析技术和基于隐式路径枚举的分析方法。Lim 等人扩展了基于语法树的 WCET 计算技术，使之能够分析处理器特性，并对 RISC 处理器的流水线和 Cache 部件进行了分析[20-22]。Li 等人提出了一种基于隐式路径枚举 WCET 计算方法的 Cache 分析方法[23,24]。其基本思想是通过分析程序内部潜在的 Cache 冲突，来估计 Cache 访问是否命中。Li 和 Mitra 等人对乱序流水线进行了分析[25,26]。由于指令的乱序执行，单独的某条指令的执行时间将会有很多种可能。他们通过采用数值区间来表示指令的执行时间，在分析中避免了对所有可能的指令执行时间的枚举，因此有很高的分析效率。

Saarland 大学的研究小组提出了采用抽象解释技术[27]分析指令 Cache 的分析技术[28,29]。其研究工作的基本出发点就是力求独立计算出 CFG 中每个基本块的最坏情况执行时间，如果能够得到这个信息，那么就可以采用任何可能的 WCET 计算技术来求解整个程序的 WCET。该研究小组还采用抽象解释技术对流水线进行了分析[30]。其基本思路是首先建立流水线的具体模型，定义抽象流水线状态。抽象流水线状态的运算是通过更新集合中的每个具体状态来实现的。通过计算抽象流水线状态，流水线对执行时间的影响就可以被估算出来。

Yan 和 Zhang 等人提出了面向多核共享 Cache 体系结构的 WCET 分析方法，该方法是对传统的“隐式路径枚举+抽象解释技术”的分析方法的一种改进，通过静态分析多个核心上不同程序在 Cache 中的地址冲突，得到程序在共享 L2 Cache 中的命中情况[31]。在文献[32]中，Zhang 和 Yan 等学者对上述方法进行了改进。改进之一就是采用基于 Cache

冲突图的技术手段替代了抽象解释技术进行 Cache 分析，使得分析结果的精度更高。同时，在对不同核心 Cache 冲突的分析中，作者考虑了潜在冲突指令在执行时间上的先后关系，从而提高了分析结果的精度。

在实时操作系统 WCET 分析方面，Colin 与 Puaut 等人采用 WCET 分析工具 Heptane 对 RTEMS[33]实时内核进行了分析，这是第一个采用静态 WCET 分析技术分析实时操作系统的研究工作[34]。Colin 等人的研究首次给出了采用静态分析技术分析实时操作系统的诸多问题。Carlsson 与 Sandell 等学者对 OSE 实时内核[35]进行了分析[36-39]。Petters 及其研究小组对 L4 实时内核[40]进行了 WCET 分析，旨在探索对实时操作系统进行 WCET 的自动化程度[41]。

WCET 分析是实时系统时间验证的核心任务之一，这一领域目前已有很多的研究成果，且一部分成果已经应用到了实际系统的分析之中。但是仍有许多新的问题需要进一步的研究。目前相关研究尚存在的问题包括以下一些方面：

在 WCET 计算方面，目前隐式路径枚举技术是主导技术。但是该技术的主要问题是描述复杂的程序控制流程信息的能力很有限，这种计算框架同时也限制了处理器行为分析所能采用的技术种类。同时，整数线性规划问题的求解效率取决于线性约束的个数，而随着程序复杂度的提高，生成的线性约束的个数可能呈现大幅度的增加。相比之下，基于路径的 WCET 计算方法在描述能力和分析精度等方面具有其他技术所不能比拟的优势，目前这方面的研究还很不充分，因此基于路径的 WCET 计算方法的优势也没有在 WCET 分析中得到充分的发挥，对于这种技术的特性的探讨也比较匮乏。

在处理器行为分析方面，目前基于抽象解释理论的 Cache 分析已经取得了主导地位，但是相关研究主要集中在对 LRU 这一经典替换算法的分析上。但是在实际的处理器中，出于硬件实现复杂度的考虑，大多采用了 FIFO 或 Psudo-LRU 等替换算法。目前基于抽象解释这一理论框架进行 FIFO 和 Psudo-LRU 替换算法的研究还很不成熟。为使得 WCET 研究在实际系统中能够得到更好的应用，需要提出新的分析手段来处理基于上述替换算法的 Cache 分析技术。

多核处理器的出现，为 WCET 分析提出了新的挑战，其中最关键的就是共享末级 Cache 导致的程序执行时间的高度不可预测。目前面向多核共享 Cache 的 WCET 分析研究还很不成熟，但是正逐渐成为研究热点。现有技术主要通过分析不同核心上程序间的地址冲突来分析 Cache 命中情况，这种方法的分析精度很低。为得到程序在共享 Cache 上的命中情况和时间特性，就必须从更细的粒度上探讨程序间的干涉情况，并把这类信息集成到 WCET 的计算中。目前这方面还没有有效的分析技术出现。

在实时操作系统 WCET 分析方面，目前的相关研究主要是对某个操作系统的小部分

代码进行分析，对于完整系统的分析还不充分。同时，相关研究采用“单值 WCET 分析”分析操作系统代码，因此分析精度较低。采用现有静态 WCET 分析方法分析实时操作系统所存在的主要问题及对可能解决方法的研究还需要进一步深入，这对于探索如何将 WCET 领域研究成果应用于实际系统的分析具有重要意义。

1.3 本文研究内容

本文针对 WCET 分析领域当前主要的热点及该领域研究尚存在的不足，在 WCET 计算方法，单核及多核处理器行为分析以及实时操作系统 WCET 分析等方面展开了研究。本文的研究内容主要有以下几个方面：

（1） 基于模型检测技术的 WCET 计算方法

“WCET 计算”是 WCET 分析的三个子任务之一，采用何种计算方法直接决定了 WCET 分析的整体框架，并影响到处理器行为分析与控制流程分析。基于路径的 WCET 计算技术具有分析精度高的优点，但是对于此类技术的相关研究和特性分析还很不充分。本文提出了一种基于模型检测技术的 WCET 计算方法，给出了从程序的控制流程图向程序自动机转换的语义，并采用基于不同理论的多种模型检测器（SPIN，NuSMV，UPPAAL）对该问题进行建模。通过与基于隐式路径枚举技术的对比，分析了基于模型检测技术的 WCET 计算方法在分析精度和描述能力方面的优势，评价了该方法的可伸缩性。同时横向比较了三种不同模型检测器在时间、空间可伸缩性方面的差异。最终给出了这一技术的适用范围。

（2） 基于剪枝思想的单核系统 Cache 分析

采用模型检测技术进行 WCET 计算，并进而基于这一框架进行处理器行为分析，主要的一个问题就是程序路径数量的爆炸，这一问题严重影响到了基于模型检测技术的 WCET 计算方法的可伸缩性。本文提出了一种基于剪枝思想的 WCET 分析方法，通过计算不同程序分支的执行时间特性，将那些在 WCET 计算中不需要考虑的分支进行削减，从而大大降低 WCET 计算中所需要考虑的程序分支的个数，能够有效提高基于模型检测技术的 WCET 分析的性能及可伸缩性。

（3） 面向多核共享 Cache 的处理器行为分析

计算机处理器体系结构设计正在经历从单核处理器到多核处理器的巨大转变。多核处理器通常采用共享 Cache 的设计，多个核心上的任务对共享 Cache 的细粒度访问造成了系统时间行为的复杂性和不可预测性，这给 WCET 分析带来了巨大挑战。本文提出了一种基于模型检测技术的多核共享 Cache 的分析方法，通过建立精确的处理器行为模型达到提高分析精度的目的。通过与相关工作对比，分析了基于模型检测技术的多核共

享 Cache 的分析方法分析精度高的主要原因，以及该种技术对新体系结构的强大适应性，同时也评价了该技术在分析效率方面的问题。面向多核共享 Cache 的处理器行为分析的研究尚不成熟，这部分的研究探索了一种全新的解决问题的框架，对于加深对多核共享 Cache 时间特性的理解起到了重要作用。

（4） 实时操作系统的 WCET 分析

一个系统的时间可预测性不仅仅取决于应用程序，同时取决于为应用程序提供服务的实时操作系统。另一方面，实时操作系统以控制密集型代码为主，其代码结构和执行特性与普通应用程序有很大区别。因此，对实时操作系统进行静态 WCET 分析，并探讨传统技术在分析实时操作系统代码时的精度等问题就显得尤为重要。本文对工业界广泛使用的 μC/OS-II 实时操作系统进行了静态分析，包括对系统调用以及禁止中断区间的分析，得到了该系统实时时间特性的完整描述。在此基础上，评价了传统技术分析实时操作系统的精度，以及尚存在的问题。这部分研究为今后研究工作的进一步深入开展提供了很多新的方向。

1.4 本文的组织结构

本文共分 8 章，各章具体内容组织如下：

第 1 章为绪论，主要介绍了本文的研究背景及意义，包括实时嵌入式系统研究背景，WCET 分析的主要任务及其意义，分析了国内外研究现状与尚存在的问题，介绍了本文的主要内容，以及篇章结构。

第 2 章主要介绍 WCET 分析的背景知识，全面阐述了主流 WCET 分析技术及其分类，并详细介绍了静态 WCET 分析的三大主要功能模块，以及各模块中所使用的主流技术，同时简要阐述了 WCET 分析理论应用于实际系统存在的问题。

第 3 章研究了静态 WCET 分析中的 WCET 计算技术，提出了基于模型检测技术的 WCET 计算方法，采用 SPIN，NuSMV 和 UPPAAL 等模型检测器对问题进行建模，讨论了该技术的可伸缩性和主要优缺点，并提出了该分析方法的适用范围。

第 4 章在第 3 章工作的基础上，研究了基于剪枝思想和模型检测技术的单核系统 WCET 分析方法，讨论了模型检测在建模不同 Cache 替换算法时的强大描述能力，着重研究了采用程序路径剪枝削减状态空间的技术。

第 5 章研究了多核共享 Cache 分析技术，阐述了多核时代 WCET 分析面临的主要挑战，提出了一种基于模型检测技术的多核共享 Cache 的分析方法，并讨论了该分析方法在分析精度上的优势，以及方法的可伸缩性问题。

第 6 章主要研究了实时操作系统的 WCET 分析，具体工作包括对著名实时操作系统

μC/OS-II 的系统调用和禁止中断区间的 WCET 分析，同时评价了传统静态 WCET 分析方法用于分析实时操作系统的精度，以及尚存在的主要不足。

第 7 章介绍了基于第 3 章至第 6 章的理论和方法所设计的支持多核体系结构的静态 WCET 分析工具，详细介绍了各模块的主要功能。

第 8 章总结全文，并提出了未来研究方向以及可以继续深入研究的内容。

第2章 WCET 分析研究背景

实时系统的核心问题之一是分析实时任务的执行时间特性。由于受程序流程、处理器功能特性、以及执行环境等因素的影响，一个任务的执行时间通常是不确定的，且客观上呈现出某种分布[5]。我们把一个任务所有可能的执行时间中的最大值称为这个任务的“最坏情况执行时间（WCET）”，把最小值称为这个任务的“最好情况执行时间（BCET）”。分析或计算实时任务的 WCET 是实时系统的重要研究课题。从实现的角度来讲，实时任务就是一段计算机程序，因此在后面的讨论中，我们不明确区分任务和程序这两个概念。本章主要介绍影响程序 WCET 以及 WCET 分析复杂性的主要因素；之后分别介绍静态 WCET 分析技术和动态 WCET 分析技术，重点介绍静态 WCET 分析技术，并对上述二者进行比较；最后简单探讨 WCET 分析在实际应用中的可用性问题。

2.1 影响程序 WCET 及分析复杂性的主要因素

计算或估计一个程序的 WCET 值，实际上就是寻找程序按照哪条路径在目标处理器上的执行导致了最长的执行时间。因此，程序的 WCET 取决于待分析程序的结构、目标处理器、以及执行环境。

影响 WCET 的程序结构主要是分支、循环以及函数调用。分支是计算机程序中最常见的结构，其意义是根据不同的条件完成不同的功能。一个最简单的分支程序可以只包含两条分支路径。由于两条路径所完成的功能不同，因此程序在两条路径上的执行时间通常是不相同的。对于循环，影响执行时间的一个关键参数就是循环次数；循环次数越大，执行时间越长。为了估计程序的 WCET，通常需要知道循环在最坏情况下的执行次数，也称循环上限。在实际程序中，循环的次数通常是不固定的，它经常依赖于某些变量的值，而这些变量的值又会随程序状态的不同而变化，因此确定循环上限并不简单。此外，几乎所有的计算机程序都会通过调用函数来完成某些功能，对函数的调用同样会影响程序的 WCET。一类典型的程序结构是通过函数指针实现的动态函数调用，这种函数调用的目标函数只有在程序的执行过程中才能够确定，这就给静态 WCET 分析带来了很大的问题。而且，即使是调用目标可确定的静态函数调用，同一函数可能在程序的不同位置被调用，被调用函数的执行时间一般不同。单独分析程序分支、循环、函数调用等可能并不困难，但是实际系统中的程序往往是以复杂的方式大量嵌套使用上述程序结构，从而使得最终的程序非常复杂，这就为 WCET 分析带来了巨大挑战。此外，实际系统中相当数量的程序往往不是严格按照“结构化程序”的要求设计的，非结构化的

程序也会增加 WCET 分析的难度。

目标处理器的特性也是影响程序 WCET 值以及 WCET 分析复杂性的重要因素。例如，同样的一个程序运行在 x86 处理器上和 ARM 处理器上，其执行时间通常不相等。一般来讲，影响程序 WCET 的处理器特性包括指令集、流水线、Cache、分支预测器以及处理器运行频率等。

不同的指令集会影响程序的执行时间。例如，某些简单处理器其指令集中的所有指令都具有固定长度的执行时间（通常以处理器周期为单位）；而复杂处理器的指令集中常包含一些诸如乘法运算等的指令，这些指令的执行时间往往依赖操作数的数值。

为提高处理器的指令吞吐率，目前几乎所有的桌面处理器和嵌入式处理器都采用了流水线的设计。同时为了更高效的利用处理器内部的计算资源，流水线通常设计为乱序执行的，也就是将可执行代码中的机器指令顺序打乱执行（但保证逻辑结果的正确性）。流水线的存在导致程序执行时间的波动和不可预测。例如程序中的循环结构，每次执行到循环体的第一条指令的时候，流水线中的内容通常是不固定的，这样就导致循环的每次执行时间都可能不相同。为得到较为精确的分析结果，就必须对流水线的工作过程进行详细分析，这将导致分析难度的加大。

另一方面，为了解决处理器运算速度和主存访问速度之间的差异，目前几乎所有的处理器都引入了 Cache 来解决上述问题。无论是指令还是数据，在 Cache 中命中与否都将直接影响程序的执行时间。由于 Cache 的组织形式各异，替换策略也不尽相同，这就导致分析程序在最坏情况下的 Cache 命中情况变得非常困难。

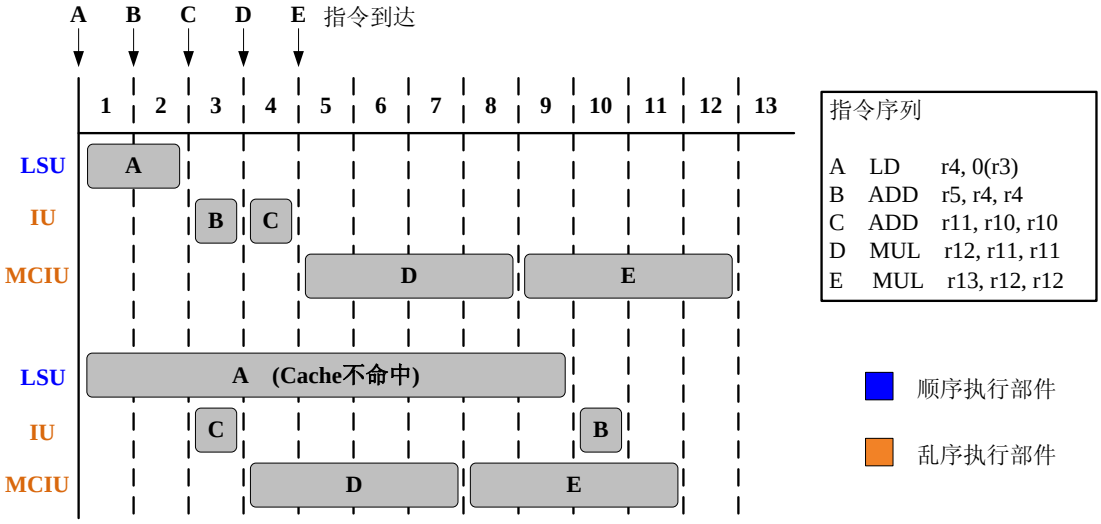


图 2.1 时间异常举例

Fig. 2.1 An example of timing anomaly

多种处理器功能特性综合在一起，还会给 WCET 分析带来更大的问题。Lundqvist

等学者在 1999 年首次提出了“时间异常（Timing Anomaly）”的问题[42]。图 2.1 表示了同一段程序在一个同时配备了乱序流水线和 Cache 的处理器上的两种不同的执行情况。A-E 分别代表指令。在情况一中，指令 A 已经存在于 Cache 中，因此 A 的执行是 Cache 命中，仅用 2 个时钟周期执行完毕。接下来 B 和 C 依次占用 IU 运算单元执行。由于 D 依赖于 C 的运算结果，E 依赖于 D，所以 D 和 E 将在 C 执行完毕后方可依次执行。5 条指令共执行了 12 个时钟周期。而在情况二中，如果指令 A 在 Cache 中不命中，那么它将执行 10 个时钟周期，因此 B 将在第 11 个时钟周期执行。由于流水线是乱序执行，与 A 和 B 没有逻辑依赖关系的 C 可以提前在 IU 运算单元上执行，进而 D 和 E 的执行时间都比情况一中提前了一个时钟周期。这样一来，5 条指令仅用 11 个时钟周期便可执行完毕。直觉上，情况二中指令 A 发生 Cache 不命中，其整体的执行时间应该比情况一的要长。但是指令 A 的 Cache 不命中却间接导致了整体执行时间的缩短。这种现象是与直觉相反的，故称其为“时间异常”。时间异常现象已经在实际的处理器中被捕捉到[43]。由于时间异常的存在，在分析程序 WCET 时就不能简单的认为 Cache 不命中一定导致更长的执行时间，需要对流水线进行更加细致的分析才能发现并处理上述情况，这就给 WCET 带来了很大的困难。

处理器中的分支预测器等功能部件也将影响程序的 WCET 及其分析复杂性[44,45]，这里不一一详述。除程序自身结构、处理器特性等因素外，程序的执行环境对执行时间也会有影响。这里“执行环境”主要是指操作系统状态，例如软件资源的数量、系统当前运行状态等。实时系统中的一类典型任务是周期任务，在周期任务中，任务代码会按照指定的周期被触发执行。由于系统的状态是系统中的所有任务的执行综合作用的结果，因此周期任务的每次执行所处的系统状态也不可能相同，这会造成程序执行时间的变化。

总之，程序结构、处理器特性、以及程序的执行环境等因素都将影响程序的 WCET，同时也为 WCET 分析带来了巨大挑战。在具体分析程序 WCET 的过程中，需要尽可能完整的考虑上述因素所产生的影响，才能够得到更加精确的分析结果。

2.2 主流 WCET 分析技术

目前在研究领域和工业领域主要有两类 WCET 分析技术：静态 WCET 分析技术和动态 WCET 分析技术。所谓静态 WCET 分析，就是在不执行程序代码的情况下，通过数学方法离线分析程序代码本身，进而获得程序 WCET 的估计值。而动态 WCET 分析，是指多次执行待分析程序的代码，在多次执行的观测值中取最大者作为程序的 WCET 值。本节将分别介绍上述两类分析技术，其中重点介绍本文所研究的静态 WCET 分析技术，之后对两种分析技术的特点进行对比。

```
void main()
{
        int b;
        int i = 0, j = 0;
        while (i < 10){
                if (b)
                        j++;
                else
                        j--;
                i++;
        }
}
```

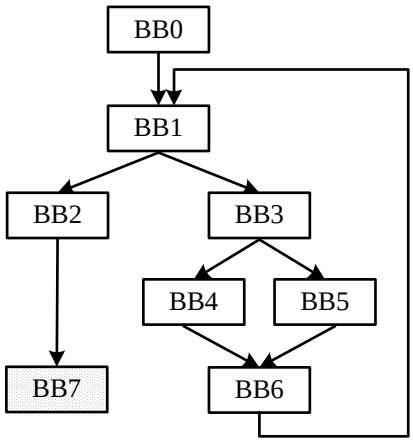


图 2.2 程序的 CFG 举例

Fig. 2.2 An example of the CFG of a program

2.2.1 静态 WCET 分析技术

如上所述，静态 WCET 分析是在不执行程序代码的情况下完成的。通常情况下，不管采用何种具体技术，静态分析的首要工作都是将待分析的程序进行反编译，从可执行代码中重建程序的控制流程图（CFG）。图 2.2 是一个简单的示例程序及其对应的 CFG，为简化表示，编译得到的可执行代码省略。程序的 CFG 本质上是一个“单入口、单出口”的有向图。CFG 的每个节点是一段连续的机器指令，且其中不包含分支跳转类型的指令；CFG 的每条边表示的是程序的跳转信息，如果一个节点有 N 条出边，那说明在这个节点执行完毕后将有 N 个可能的跳转目的地址；图 2.2 中的 BB1、BB3、BB4、BB5 和 BB6 节点及连接它们的边构成了一个循环体。静态 WCET 分析的目标输入实际上就是程序的 CFG。完整的静态 WCET 主要由三个子任务组成：控制流分析、处理器行为分析、以及 WCET 计算。这三个子任务及其关系见图 2.3。我们将在下面的几个小节中对这三部分所要完成的具体功能分别进行详细介绍。由于 WCET 计算所采用的技术手段会影响到控制流分析和处理器行为分析所能够采用的技术，因此我们首先介绍“WCET 计算”子任务。

2.2.1.1 WCET 计算

同样以图 2.3 为例，对于一个程序及其 CFG，我们假定 CFG 中每个节点的执行时间已知，且所有循环的上限已知，那么“WCET 计算”子任务的目的就是找到程序的一条执行路径，满足在这条路径上的所有 CFG 节点的执行时间之和最大，那么这条路径就是程序的 WCET 所对应的路径。有些计算方法可能并不直接寻找这条路径，而是通过其他方法求得程序的 WCET，其本质和搜索 WCET 对应的路径是相同的。目前主要

有三类 WCET 计算技术：基于语法树的技术（Tree-Based Techniques，又称 Timing Schema）、隐式路径枚举技术（Implicit Path Enumeration Technique）、基于路径的技术（Path-Based Techniques）。下面分别进行介绍。

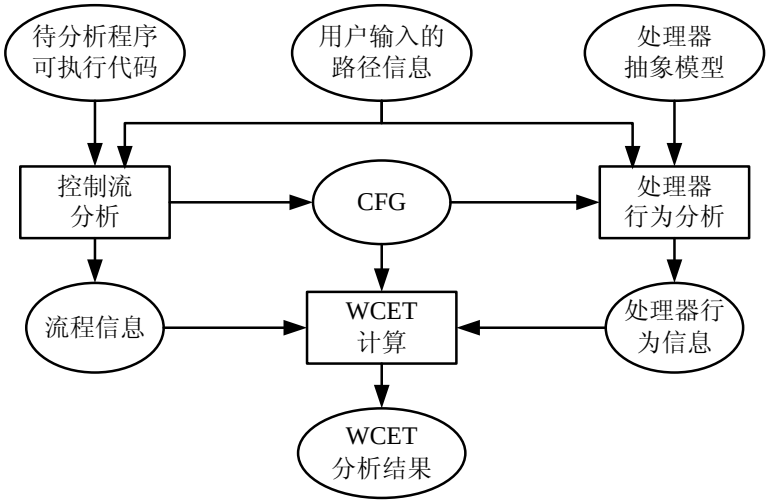


图 2.3 WCET 分析子任务及其关系

Fig. 2.3 The sub-tasks of WCET analysis and their relationship

（1） 基于语法树的技术

基于语法树的技术首先由 Shaw 和 Park 等学者提出[7-9]，图 2.4 表示了采用基于语法树的技术计算程序的 WCET 的基本思路[46]。这种技术要求待分析程序的结构可以被表示为一棵语法树。在这一条件满足的情况下，我们可以从树的最底层依次向上规约，规约法则如图 2.4 所示。当低一层的叶子节点的执行时间已经求出后，就可以通过这些叶子节点来计算它们对应的上一级节点的执行时间。当规约到树根的时候，得到的执行时间就是程序的 WCET。图 2.4 表示了该示例中的语法树的规约过程，以及在这一过程中如何计算程序的 WCET。

基于语法树的技术最突出的优点就是计算速度快，但是其问题就是不能处理非结构化的程序，因为这些程序可能不存在直接对应的树形语法结构。而在现实系统中，非结构化的程序大量存在，这就限制了这种技术的应用。而且，即使程序在高级语言层面是结构化的，但是由于目前大多数的编译器都采用了各种优化技术，编译之后的可执行程序可能已经不再是结构化的，那么也就无法使用基于语法树的技术进行分析。此外，程序的某些控制流信息可能是建立于两个不同子树之间的，这种控制流信息是很难被建模到基于语法树的分析框架中的。

Lim 等学者将这一技术进行了扩展，使之能够支持分析流水线和 Cache 等，并提出了支持 RISC 处理器的分析方法[20-22]。Colin 和 Puaut 等学者在[47-49]中研究了如何将 Cache 分析和分支预测分析纳入到基于语法树的分析框架中，同时基于这一技术开发了一个名

为 Heptane 的静态 WCET 分析工具。

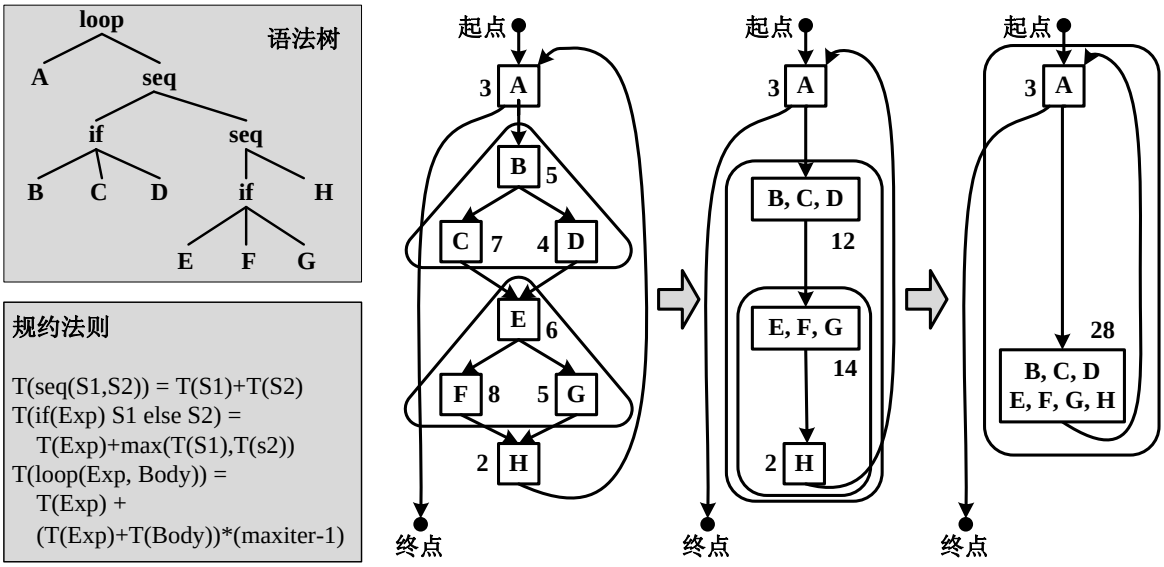


图 2.4 基于语法树的 WCET 计算技术

Fig. 2.4 Tree-based WCET calculation technique

（2） 隐式路径枚举技术

隐式路径枚举技术由 Li 等学者首先提出[10,11]。之所以称为“隐式”路径枚举，是因为这种技术并不是显式的枚举出所有的程序路径，然后寻找具有最长执行时间的路径。它的核心思想是“将 WCET 求解问题，或者说最长执行时间路径搜索问题，转化为求解 CFG 中的每个基本块的最大执行次数的问题”。这一问题可以利用整数线性规划（ILP）进行建模和求解。举例来说，假定一个程序对应的 CFG 有 N 个基本块（B_i），每个基本块 B_i 对应的执行时间为 C_i。令每个基本块的执行次数为 X_i，那么求解程序 WCET 的问题就可以描述为公式(2.1)所示的整数线性规划的目标函数，其中 MAX 表示取最大值。

$$wcet = MAX\sum_{i=1}^{N} C_i \cdot X_i \tag{2.1}$$

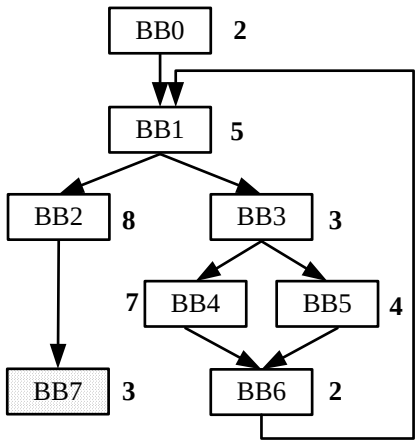
而每个基本块的执行次数并不是可以任意取值的，实际上它们受制于程序结构，也就是体现在 CFG 中的控制流程信息。我们可以针对 X_i 建立一系列的线性约束。

首先，可以建立程序结构约束。例如，我们用 d_{i_j} 表示从 B_i 到 B_j 的有向边的执行次数，那么对于 CFG 中的任何一个基本块，可以知道所有入边的执行次数之和一定等于出边的执行次数之和，如公式(2.2)所示。

$$\forall B_i, X_i = \sum_{\text{所有入边}} d_{*\_i} = \sum_{\text{所有出边}} d_{i\_*} \tag{2.2}$$

其次，还可以建立程序功能约束。例如循环上限就可以建模为程序功能约束。通过在不同节点之间建立值的关系，可以将循环上限的信息添加进来。除了循环上限，我们

还可以把用户输入或工具分析出来的一些不可行路径信息等都通过基本块的执行次数描述为一系列的线性约束，这里不一一举例。图 2.5 表示了一个简单程序所对应的 ILP 问题的描述[46]。通过求解这个 ILP 问题，我们可以得到程序的 WCET 值，以及对应的每个基本块的执行次数。



// 优化目标
WCET = max{2*X_0 + 5*X_1 + … + 3*X_7}

// 起止条件约束
X_0 = 1; X_7 = 1;
// 结构约束
$X_1 = d_{0\_1} + d_{6\_1} = d_{1\_2} + d_{1\_3}$
$X_2 = d_{2\_2} = d_{2\_7}$
……
// 功能约束
X_6 <= lb * X_0　// lb是循环上限

图 2.5 隐式路径枚举技术举例

Fig. 2.5 An example of Implicit Path Enumeration Technique

相比于基于语法树的技术，隐式路径枚举技术不受程序结构的限制，几乎任何的程序结构都可以通过隐式路径枚举技术进行描述和求解。同时，隐式路径枚举技术使用整数线性规划进行数学建模，由于整数线性规划本身具有较好的描述能力，因此基于这一框架可以集成更加复杂的处理器行为分析。Li 和 Malik 等学者已经将对 Cache 的分析描述为整数线性规划的约束，并集成到隐式路径枚举框架中进行 WCET 的求解[23,24]。

但是基于整数线性规划的隐式路径枚举技术也存在一些问题。首先，整数线性规划的描述能力并不是最强的，它仍旧无法描述复杂的程序控制流程信息。同时，整数线性规划问题的求解效率取决于线性约束的个数，而随着程序复杂度的提高，生成的线性约束的个数可能呈现大幅度的增加。但是在实际应用中，隐式路径枚举技术还是表现出了比较好的求解效率，因此被很多静态 WCET 分析工具所采用[12-15]，目前已经成为 WCET 研究和应用领域的一个主流技术。

（3） 基于路径的技术

Stappert 等学者研究了基于路径的 WCET 计算技术[16]。他们为待分析的程序建立一个 Scope Graph，它能够以层次化的方法描述程序结构。循环所对应的每一个 scope 都可以被展开成为若干个 virtual scope，最长执行路径就是在这些 virtual scope 中进行搜索的。如果发现找到的最长路径是不可行路径，那么就放弃结果再次寻找。最后每一个 scope 都可以找到一个最长路径。整个程序的 WCET 的计算类似于基于语法树的计算方法。

Chapman 等学者开发了一个名为 SPATS（SPARK Proof and Timing System）的系统，

该系统可以将程序证明和 WCET 分析通过程序的 CFG 集成在一起[17-19]。他们提出了一种从 CFG 向 WCET 转化的算法，这种算法可以将一个带环的 CFG 图转化为路径表达式，通过分析路径表达式来求解程序的 WCET。由于分析 WCET 是基于路径的，对 Cache 等处理器行为的分析可以很直接的被加入进来。但是当程序可能的路径数量较多的时候，这种方法将遇到比较大的问题。

Metzner 在 2004 年提出了采用模型检测器分析程序 WCET 的办法[50]，其本质也是一种基于路径的分析方法。但是 Metzner 的工作中并没有给出完整的实验数据。本文第三章将着重研究采用模型检测器分析程序 WCET 的技术，并采用多种不同工作原理的模型检测器对 WCET 问题进行建模，比较它们的优缺点并评价其性能。

（4）三种 WCET 计算技术的比较

表 2.1 比较了上述三种 WCET 计算技术的各方面特性。在分析效率上，基于语法树的分析技术具有绝对优势；而隐式路径枚举技术效率较高，但是当线性约束的数量增加时分析效率随之下降；基于路径的分析技术的分析效率受制于程序可能的路径个数，在程序比较复杂的时候分析效率低下。相反的，在分析精确性上，基于语法树的分析技术相对较弱，而基于路径的技术最好。隐式路径枚举技术和基于路径的技术都具有较好的描述流程信息的能力，其中后者更为突出。三种技术中只有基于语法树的技术会受编译器优化的影响。

表 2.1 不同 WCET 计算技术的比较

Table 2.1 Comparison of different WCET calculation techniques

比较项目	基于语法树的技术	隐式路径枚举技术	基于路径的技术
分析效率	最高	较高	低
分析精确性	一般	较好	最好
流程信息描述能力	差	较好	最好
受编译器优化影响	是	否	否

可以看出，在 WCET 计算中，分析效率和分析精度是一对相互矛盾的优化目标。分析效率较高的技术往往分析精度不高。不同的技术实际上是对这两个目标的不同折中。三种技术各有优缺点，因此客观上不存在一个最好的技术。需要采用哪种技术，还要看分析者对分析效率和分析精确性的具体要求。

2.2.1.2 控制流分析

上面的 WCET 计算子任务给出了进行 WCET 分析的基本框架，但是还有其他一些问题需要进一步解决，例如循环上限问题、不可行路径问题。如果程序的循环没有上限，

那么程序将不能停机，也就不存在 WCET，所以求解程序的 WCET 必须知道循环的上限值。此外，根据程序的语义，CFG 中的某些路径实际上是不可能被执行的，这些路径称为不可行路径（Infeasible Path）。如果能够将这些不可行路径分析出来，并考虑到 WCET 的计算中，那么将提高结果的精确性。控制流分析的主要目标就是“确定循环上限”和“分析不可行路径”。

通常情况下，WCET 分析工具要求用户通过“程序注释”等方式人工输入循环上限和不可行路径等信息。但是这种做法的主要问题就是需要大量的用户介入，导致 WCET 分析的自动化程度过低。同时，系统难以保证用户输入的信息是正确的，因此可能导致分析错误。很多学者在自动化程序流分析方面开展了研究。

（1）循环上限的自动分析

Liu 和 Gomez 等学者提出了一种基于符号化执行（Symbolic Execution）的循环上限自动分析技术[50,51]，这项技术可以对高级语言源程序进行分析以获取循环上限信息。首先，待分析的程序会被转化为一种时间函数，该函数以程序的某些属性为参数，返回程序的执行时间。然后，根据部分已知的输入信息，将这个时间函数转化为循环上限函数。基于上述思路，通过进行符号化的分析来获取循环上限信息。

上述基于符号化执行的技术存在的一个问题是，必须多次执行循环，这就造成了运算过于复杂的问题。Healy 等学者提出了一种基于抽象解释（Abstract Interpretation）的技术来估计循环的上限值和下限值[52,53]。其方法主要分为如下几个步骤：首先分析位于循环中的哪些分支结构会影响循环次数；其次，计算可能达到这些分支的次数的上、下限；最后，利用上一步的信息进而计算循环的上、下限。

（2）不可行路径分析

Ermedahl 和 Gustafsson 等人使用符号化执行技术分析不可行路径以及循环上限[54]。其基本原理是为程序定义一系列抽象语义，核心概念是环境的抽象值 σ_i^h，它表示从路径 h 到达程序点 i，在这个程序点上某些变量的抽象值（一般是一组值域表示）。同时定义一组规则来描述如何更新和计算不同程序点的环境抽象值。如果在某个点上的环境抽象值为空，那么说明不存在从 h 到 i 的路径，或者说从 h 到 i 的路径是不可行路径。

Lundqvist 等人使用处理器周期精度的符号化模拟（Symbolic Simulation）技术来分析不可行路径，同时进行 WCET 的计算[55]。在他们的分析框架中，对于控制分支的变量，除了有值域信息外，还额外定义了一种“不可知”类型的变量值。如果在模拟的过程中，控制分支的变量不是“不可知（*unkown*）”，那么模拟过程将按照控制变量指向的方向执行，而另外一个方向则为不可行路径；如果控制分支的变量是“不可知”，那么两条分支路径都必须一一被模拟执行。通过上述形式的模拟执行，得到程序的 WCET。

Ferdinand 等人使用抽象解释技术分析不可行路径[56]，这种技术又被称作“值分析（Value Analysis）”。其主要工作原理是分析每个处理器的寄存器可能的取值范围，在分析一个程序分支的时候，将上述取值范围与分支判断条件进行比较，如果表明程序将走一条确定的分支路径，那么另外一条路径就是不可行路径。

（3）标注技术

即使有上述技术可以自动分析循环上限和不可行路径，但并不能解决所有问题，因此在分析实际程序的时候，仍旧需要分析人员的介入，以人工方式提供程序流信息。但是，不管是自动分析，还是人工输入，必须有形式化的技术手段对这些程序流信息进行描述，才能够将它们集成到整个 WCET 分析框架中使用。

最早进行这方面研究的是 Puschner 等学者，他们提出了一种叫做 MARS-C 的语言，这种语言专门用于程序流信息标注[57]。该语言定义了 scope，markers，loop sequence 等一系列的基本元素，通过这些元素标记程序片段、循环信息等。这种语言主要与基于语法树的分析框架结合使用。

Park 等学者提出了一种名为 IDL（Information Description Language）的脚本语言，这种语言可以被进一步转化为正则表达式[9]。IDL 不仅能够描述类似程序片段的执行次数的信息，同时也能够描述表达式之间的逻辑关系等。

在前面介绍的隐式路径枚举技术中，Li 等人采用整数线性规划约束的手段对控制流信息进行建模，前面提到的程序功能信息就属于这一类。他们已经证明 IDL 中的所有语句都可以被转化成整数线性规划的约束形式。

Gustafsson 等人提出了一种名为 ALF 的控制流分析表示语言[58]。其工作的主要目的是提供一种灵活通用的中间型语言，无论待分析的对象是高级语言级别，还是可执行代码，或者中间语言，都可以将其转化为 ALF 语言的程序。这样，可以使用一个能够处理 ALF 语言的控制流分析器对 ALF 语言的程序进行控制流分析，之后再将分析得到的程序流信息映射回原有的分析框架所支持的描述形式。这部分工作服务于 ALL-TIMES 项目[59]，该项目旨在融合当今主流的 WCET 分析技术，最终构建一个强大的实时系统时间分析工具。

（4）路径信息翻译与编译器支持

如前所述，对于路径信息的标注可能存在于高级程序语言层面，而分析程序 WCET 的工具可能直接分析可执行代码。这样的话，就需要将在高级语言源程序里标注的路径信息正确的转化并对应到可执行代码中，从而能够被 WCET 分析器所使用。由于目前大多数的编译器会对生成的代码进行优化，因此生成的可执行代码的控制流程未必与高级语言源程序的流程一一对应，因此路径信息的翻译并不是一个简单的工作。有时，甚

至需要编译器支持上述翻译。

Pushner 等人提出了一种从源程序向可执行代码进行路径信息映射的技术[60]。这一技术由一个映射函数来完成。这个映射函数会遍历程序的语法树，每当程序向语法树的底层走一步，它都将利用行号和结构嵌套信息等，寻找高级语言语句对应的机器指令。

Engblom 等人提出了一种叫做“co-transformation”的技术来完成路径信息从高级语言源程序到可执行代码的翻译[61]。其工作主要是保证在编译器有代码优化的情况下，翻译仍旧能够正确完成。这种技术要求对编译器进行修改，使得编译器可以将具体的优化操作以 ODL（Optimization Description Language）语言的形式输出，根据这一输入生成对应的 co-transformation 引擎，这一引擎可以保证将高级语言的约束信息准确的翻译到对应的可执行代码中。

2.2.1.3 处理器行为分析

到目前为止，我们已经知道如何分析程序控制流程，给定一个程序的 CFG 如何计算导致 WCET 的最长路径，但是 CFG 中每个基本块的执行时间的计算仍旧是个问题。不严格的讲，“处理器行为分析”的目的就是根据目标处理器的特性，分析程序的 CFG 中的每个基本块在最坏情况下的执行时间。处理器行为分析是静态 WCET 中难度较大的一个子任务。目前大多数的桌面处理器和嵌入式处理器都设计有流水线、Cache、分支预测器等功能部件，大多数的嵌入式处理器还集成了内存控制器等。这些部件的存在都会影响到程序的执行时间；部件功能越复杂，处理器的状态就越多，分析 WCET 就越困难。更为严重的是，目前大多数的处理器以提高平均情况下的运算性能（或吞吐率）为设计目标，这往往使得程序的行为非常难以预测，分析程序的 WCET 就变得异常困难。处理器特性是如何具体影响程序的 WCET 以及影响 WCET 分析的，已经在 2.1 节中进行了介绍。由于处理器行为分析需要最终纳入 WCET 计算框架，所以我们依照前文的不同 WCET 计算框架，分类介绍处理器行为分析领域的相关工作。

（1）基于语法树 WCET 计算框架的处理器行为分析

Lim 等人扩展了基于语法树的 WCET 计算框架，使之能够分析处理器特性，并对 RISC 处理器的流水线和 Cache 部件进行了分析[20-22]。他们首先将描述每个局部程序块（语法树的子树）执行时间上限的变量扩展为一个叫做 WCTA（Worst Case Timing Abstraction）的数据结构，这一数据结构包含一个集合的元素，其中每个元素对应描述局部程序块中一条路径的执行时间信息和流水线、Cache 的状态信息。当两个相邻的局部程序块需要合并时（语法树向根部规约），局部程序块之间由于流水线和 Cache 造成的影响通过 WCTA 记录的状态信息进行传递和计算。尽管通过扩展，可以使得对处理

器行为的分析被集成到基于语法树的 WCET 计算框架中，但是这一框架自身的局限性（参考 2.2.1.1 小节）并没有得到本质的改善，故此技术一直没有得到广泛应用。

（2）基于隐式路径枚举 WCET 计算框架的处理器行为分析

Li 等人提出了一种基于隐式路径枚举 WCET 计算框架的 Cache 分析方法[23,24]。其基本思想是通过分析程序内部潜在的 Cache 冲突，来估计 Cache 访问是否命中。对于直接相联的 Cache，采用“Cache 冲突图（Cache Conflict Graph）”来描述潜在冲突信息，每一个 Cache 块对应一个 Cache 冲突图。Cache 冲突图的节点表示映射到该 Cache 块的主存内容，边表示程序中存在的路径信息。可以定义描述 Cache 命中与否的一组变量，然后从每个 Cache 冲突图中导出一系列的线性约束，并将其集成到基于隐式路径枚举 WCET 计算框架中，计算出考虑了 Cache 影响的程序 WCET 值。如果是组相联的 Cache，则采用“Cache 状态转换图（Cache State Transition Graph）”替代 Cache 冲突图进行分析。类似的思路还被用来解决数据 Cache 的分析。采用上述方法分析 Cache，优点是分析精度很高；但随之而来的问题就是以此方法生成的线性约束的数量非常大，这将进而导致分析时间的大幅增加。在分析复杂程序的时候该问题非常突出。

Li 和 Mitra 等人对乱序流水线进行了分析[25,26]。由于指令的乱序执行，单独的某条指令的执行时间将会有很多种可能。他们通过采用数值区间来表示指令的执行时间，在分析中避免了对所有可能的指令执行时间的枚举，因此可以有很高的分析效率。对于流水线的分析同样可以用线性约束的方式给予表达，所以这种流水线分析方法可以很容易的被集成到基于隐式路径枚举的 WCET 计算框架中。而且，即使 Cache 分析部分采用了基于抽象解释的技术，该流水线分析方法仍旧可以与之完美集成。

（3）基于抽象解释的处理器行为分析

Saarland 大学的研究小组提出了采用抽象解释技术[27]分析指令 Cache 的分析技术[28,29]。其研究工作的基本出发点就是力求独立计算出 CFG 中每个基本块的最坏情况执行时间，如果能够得到这个信息，那么就可以采用任何可能的 WCET 计算框架来求解整个程序的 WCET。对于程序中的某个基本块，可能有多条路径经过它，那么由于每条路径到达该基本块时系统的状态一般都不相同，所以可能导致这个基本块有多个可能的执行上下文。抽象解释的目的就是将这些不同的上下文综合成一个抽象状态，以此来计算当前基本块的执行时间。以 Cache 分析为例，首先定义在每个程序点上的“抽象 Cache 状态（Abstract Cache State）”，这里的程序点主要是指 CFG 中每个基本块的起始点。同时定义 update 和 join 等操作进行抽象 Cache 状态的更新和连接运算。在定义了这些基本元素和运算法则之后，根据程序的流程，遍历整个程序，计算出每个程序点的抽象 Cache 状态。由于程序中可能存在循环结构，因此对于同一个程序点，需要迭代计算多

次才能够得到确定的抽象状态，这一过程的收敛性由抽象解释理论来保证。利用得到的抽象 Cache 状态，可以将程序的每一个访存操作归类为如下四种中的一种：Always Hit，Always Miss，Persistent，Not Classified。利用这一分类结果就可以计算出考虑了 Cache 行为的基本块的 WCET 值。

该研究小组还采用抽象解释技术对流水线进行了分析[30]。其基本思路是首先建立流水线的具体模型，每一条指令的执行体现为对流水线具体状态的修改。流水线具体状态有利于准确的把握流水线的具体行为。在此基础上，定义抽象流水线状态，类似的，抽象状态就是一组具体状态的集合。抽象流水线状态的运算是通过更新集合中的每个具体状态来实现的。通过计算抽象流水线状态，流水线对执行时间的影响就可以被估算出来。

基于抽象解释技术的处理器行为分析的最大优势就是，它采用抽象状态来描述流水线、Cache 等部件的状态信息，大大缩减了需要分析的状态空间，使得分析效率非常高。但是抽象状态带来的问题就是分析精度的损失，不过在实际应用中，采用该技术能达到很高的分析精度。该研究组研发的 aiT 工具[12]就是采用了这一技术，并在实际应用中取得了很好的效果。由于采用抽象解释技术的分析方法能够计算出 CFG 中每个基本块的执行时间，实现了处理器行为分析和 WCET 计算的分离，所以原则上可以使用任何一种 WCET 计算框架来求解整个程序的 WCET。但是该技术最大的的问题就是，每当有一种新的处理器部件出现且需要分析的时候，要为该部件建立特定的抽象模型，并证明基于该抽象模型分析出来的 WCET 是安全的。这一过程往往非常复杂，需要大量的人力和时间才能够完成。

2.2.2 动态 WCET 分析技术

上面介绍了静态 WCET 分析的三个子任务，以及各子任务中所涉及的主流技术。可以看出，虽然静态 WCET 分析能够保证得到的 WCET 估计值是安全的，但是分析的复杂度普遍很高，在目标处理器特性比较复杂的时候这个问题尤为突出。在某类系统（如软实时系统）中，可能并不要求分析得到的 WCET 值一定是安全的，只要接近实际的 WCET 就可以。在这种情况下，就可以采用一些虽然结果不安全，但是分析复杂度很低的技术方法进行分析。动态 WCET 分析技术就是基于这样一个考虑而出现的。所谓“动态 WCET 分析”，又称“基于测量（Measurement-Based）的 WCET 分析”，其基本方法是在目标处理器上实际执行待分析的程序，采用软件或硬件测量的办法来估计程序的 WCET。动态 WCET 分析通常要解决以下几个问题：（1）测量工具；（2）测量整个程序还是局部程序；（3）测试向量的生成等。下面给予简单介绍。

通常情况下，测量程序的执行时间有软件方法和硬件方法两种。软件测试的方法比

较简单，但是会对测试结果造成额外的误差。通常软件测试的方法就是把记录时间的代码插入到待分析的程序中，在执行过程中输出程序的启动和终止时间。但是由于将额外的代码嵌入了待分析程序，导致待分析程序的执行上下文会发生变化，这样分析得到的结果是不精确的。为此，我们可以采用专用硬件测量程序的执行时间。例如可以采用逻辑分析仪、示波器等设备监视系统总线，从而得知程序的起始和终止状态。采用专用硬件测量具有非常高的精度，但是问题是专用硬件普遍比较昂贵。

不管采用软件方法还是硬件方法测量，都需要分析者考虑的一个问题是“测量整个程序还是测量局部程序”。测量整个程序的优点是过程简单，因为仅需要在程序启动和结束的时候记录时间即可；但是测量整个程序往往精度不高，这是因为完整程序的状态空间很大，在测量中很难完全覆盖。而为了提高测量的精度，可以将程序划分成若干块，分别测量每一块的执行时间，再按照某种计算方法求得整个程序的 WCET。图 2.6 表示的是相关工作中的动态 WCET 分析框架[62]，它采用的就是测量局部程序的技术路线。用户对精度和分析复杂性有着不同的要求，可以通过调整程序划分的粒度来实现。

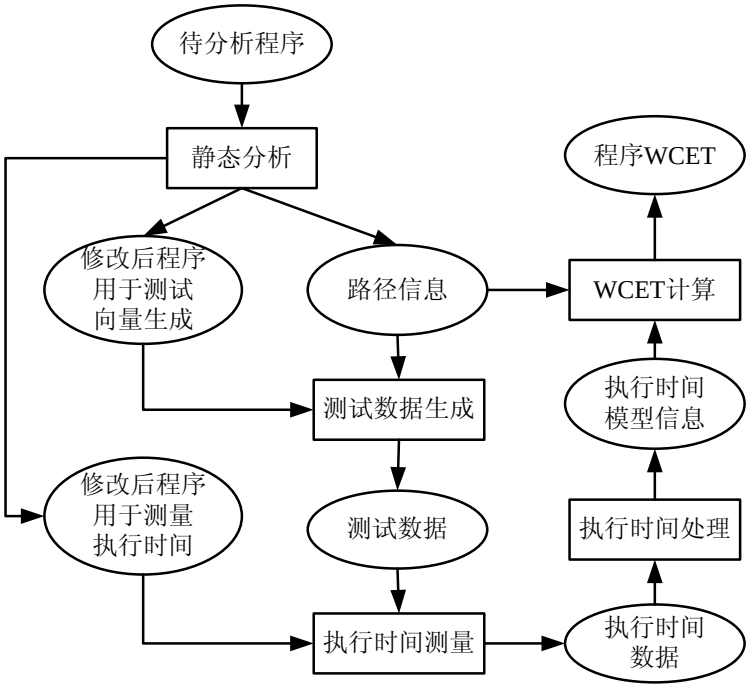


图 2.6 一种动态 WCET 分析框架

Fig. 2.6 A dynamic WCET analysis framework

通常情况下，一个程序或一个程序块不可能仅测量一次，往往需要按照不同的状态输入测量多次，以达到覆盖或接近最坏执行情况的目的。而如何高效的获得测试向量则是一个关键问题。文献[63]提出了一种采用模型检测技术生成测试向量的办法。在测试向量生成以后，可以采用软件的办法根据测试向量引导程序执行，采用硬件的办法实际测量程序的执行时间。

虽然动态 WCET 分析不能够保证估计出来的 WCET 是安全的，但是由于其具有分析复杂度低的特点，在实际应用中仍旧被广泛采用。除了上面的相关工作，SymTA/P[64]和 RapiTime[65]也都采用了类似的动态 WCET 分析方法。

2.2.3 静态分析技术与动态分析技术的比较

以上分别介绍了 WCET 分析领域的两大类分析方法——静态分析和动态分析。这两类分析方法各有优缺点，这里对其进行一个系统的比较。表 2.2 列出了所比较的主要方面，以及两类分析方法对应的特点。

表 2.2 静态 WCET 分析与动态 WCET 分析的对比

Table 2.2 Comparison of static WCET anslysis and dynamic WCET analyis

比较内容	静态 WCET 分析	动态 WCET 分析
分析结果的安全性	安全	不安全
分析复杂度	高	低
对新体系结构的适应性	差	好
分析精确性	主要取决于分析技术	主要取决于测试向量
用户帮助提高分析质量	提供额外控制流信息	提供更好的测试向量

静态 WCET 分析和动态 WCET 分析的本质区别就是分析结果的安全性。静态分析由于有数学理论的保证，其结果是安全的，也就是说，WCET 的估计值一定大于程序实际的 WCET 值。而动态分析是基于测量的，难以保证多次测量一定能够覆盖最坏情况，所以动态分析的结果是不安全的，测量得到的最大执行时间可能小于程序实际的 WCET 值。因此，静态 WCET 分析通常用于对执行时间有非常苛刻要求的硬实时系统，而动态 WCET 分析通常用于对时间要求相对宽松的软实时系统，或者极其复杂的系统。动态分析除了用于估计程序的 WCET，同时还可以粗略的给出程序执行时间的分布情况，该信息对于了解程序的执行时间特性也是很有意义的。

相比于动态分析技术，静态分析的分析复杂度普遍很高。根本原因是因为静态分析需要对目标处理器建立抽象分析模型。而且，当有新的体系结构出现的时候，动态分析技术仅需要把待分析的程序在新的处理器上重新测量即可；而静态分析需要根据新的体系结构的特点，重新建立处理器抽象模型，以及相应的分析方法。因此，后者对新体系结构的适应性相对较差。

静态分析的精确性主要取决于所采用的分析技术。通常情况下，分析技术对程序的分析越细致，结果越精确，但是分析的复杂度也随之提高。动态分析的精确性主要取决于测试向量的质量，好的测试向量能够更加接近程序的最坏执行情况。不管是静态分析

还是动态分析，用户都可以介入分析过程以提高分析质量。对于静态分析，用户主要是提供额外的控制流信息，帮助分析器去除不可行路径或者设置更加精确的循环上限；对于动态分析，用户可以手工添加更好的测试向量。

2.3 WCET 分析的可用性问题

WCET 分析的可用性问题由 Gustafsson 在 2008 年首次明确提出[6]，但是这一问题早已被相关领域研究者和工业界所关注。目前大多数的 WCET 研究工作都是在理论层面开展的，通常研究所采用的模型以及所面向的体系结构都是经过一系列的假设简化后的。这些理论在用于实际系统进行程序分析的时候是否还能保证足够的分析精度和分析效率，是一个非常具有实际意义的问题。一项技术或一类技术流派的可用性的好坏主要可以从三个方面加以衡量：

（1） 对硬件体系结构的适应性

一个程序的 WCET 值与运行该程序的实际硬件有着直接的关系。在学术研究中，大多通过一些假设简化硬件体系结构，但是在实际系统中，随着应用需求的提高，硬件体系结构已经高度复杂，曾经主要使用于桌面处理器或服务器处理器的超标量乱序流水线、多级 Cache、分支预测器等功能部件目前已经广泛存在于嵌入式芯片中。对于在众多假设基础上建立起来的分析理论或分析方法，用以分析实际系统中的复杂硬件体系结构时很有可能出现分析效率低下或分析精度低下的问题。此外，嵌入式系统大多采用协同设计的设计方法，在系统的早期设计阶段，软件与硬件的设计是并行进行的。也就是说，在设计初期需要分析系统时间特性的时候，可能目标硬件尚未设计完毕。在这种情况下，就只能采用静态分析方法，而无法采用动态分析方法——这实际上限制了不同类别分析技术的使用范围。

（2） 分析目标

分析目标通常是指要分析的程序 WCET 是否一定是安全的。对于硬实时系统，通常要求分析结果必须是安全的，否则将发生灾难性后果，因此在此类系统中，通常只能采用静态分析方法。有些系统的结构可能非常复杂，使用静态分析方法在分析效率上很难满足要求。如果待分析的系统是软实时系统，也就是说不要求分析结果绝对安全，这种情况下就可以采用牺牲了结果安全性，但分析效率相对较高的动态分析方法。此外，用单一值描述程序的 WCET 还是用一个参数化的方法描述程序的 WCET 也是分析目标范畴的问题。像类似实时操作系统这样的特殊程序，它的运行通常可以区分为不同的模式，且不同模式下的执行时间差异很大。在这种情况下，如果分析技术只能给出单一的 WCET 值，那么结果的可用性就很差。

（3） 对待分析程序的适应性

待分析程序本身的特性也直接反映分析技术的可用性。影响分析精度和分析效率的程序因素主要包括：待分析程序的大小，是否需要分析全部程序还是只分析局部，实际系统采用何种操作系统，待分析代码是人工编写还是自动生成，分析者是否可以获得待分析程序的源代码，以及编译过程对程序执行时间的影响等。例如有些分析技术，在对小程序的测试中表现出比较好的分析效率，但是当程序规模增加的时候，分析时间也显著增加。有的分析技术只能分析完整程序，但是有时也可能有分析局部程序的需求，本文第 6 章对实时操作系统禁止中断区间的分析就是局部程序分析。此外，有的分析技术只能分析源代码，而能够分析可执行代码的技术或工具显然具有更好的可用性。而且，如果待分析源程序是由代码生成器自动生成，那么代码的可读性将非常差，这也将影响分析的难度。由于程序是 WCET 分析的直接工作对象，因此程序的各方面特性都会直接影响分析技术的可用性。

总的来说，检验 WCET 分析技术可用性就是要把相关理论、技术和工具放到实际的应用系统中，检验其是否能够满足用户所要求的分析精度和分析效率。实时系统广泛存在于航空航天、武器装备、核能应用、汽车电子等重要工业领域，因此研究 WCET 分析可用性的目的并不是简单的给出现有技术的一个可用性的指标，而是要进一步推动分析技术的更新，使得相关研究能够同步于实时系统复杂度的提高。

2.4 小结

本章主要介绍了 WCET 分析研究领域的相关工作基础，主要包括动态 WCET 分析和静态 WCET 分析，分别介绍了各类分析方法需要完成的功能，所采用的主要技术手段，以及在分析精度和分析效率上的特点。其中着重介绍了本文研究所隶属的静态 WCET 分析。最后简要阐述了 WCET 分析在实际系统中的可用性问题。

第3章　基于模型检测技术的 WCET 计算方法

第 2 章介绍了 WCET 分析的几个子任务，其中比较重要的一个就是“WCET 计算”，因为采用何种计算方法，直接决定了 WCET 分析的整体框架，而处理器行为分析和控制流程分析都将受制于所选择的技术手段。目前主流的 WCET 计算方法有“基于语法树的技术、隐式路径枚举技术和基于路径的技术”三大类，这三类技术在分析能力、分析精确性和分析效率上各有优劣。在实际应用中，往往根据需求的不同选用不同的技术。在现有的研究中，使用的最多的是隐式路径枚举技术，其次是基于语法树的技术。但是这两种技术对于复杂的程序语义的描述能力有限，这将导致分析精确性的降低。同时，随着处理器技术的发展，目前实时系统所采用的处理器越来越复杂，而上述两种技术由于描述复杂处理器行为的能力较差，将进一步造成分析精确性的下降。因此在对分析精确性有很高要求的应用中，需要采用基于路径的分析技术。

虽然有相关研究涉及到了基于路径的分析技术，但是并没有形成系统的技术路线。同时对于具体的基于路径的分析技术的分析性能、主要特点、适用范围等都缺乏充分的研究和探讨。随着计算机性能的飞速提升，传统用于硬件逻辑验证的模型检测技术正逐渐受到软件验证领域的关注，同时很多学者将模型检测器作为一种优化工具求解问题的最优解，其应用最为广泛的就是实时嵌入式系统。本章主要研究基于模型检测技术的 WCET 计算方法，给出了采用该技术对 WCET 计算问题进行建模的基本框架。同时，本章选取了三个目前比较流行的模型检测工具用于 WCET 计算，通过详细的实验给出了三种工具在分析时间和内存使用量方面的性能结果及其对比，并探讨了 WCET 计算中影响模型检测技术分析效率的主要因素。在此基础上，我们进一步分析评价了基于模型检测技术的 WCET 计算方法的特点与适用范围。本章的研究工作对于探索、分析和评价基于路径的 WCET 计算技术具有重要意义。

3.1　相关工作

由于本章的工作采用了模型检测技术进行 WCET 计算，因此在相关工作中简要介绍模型检测技术的一些基本知识与背景；之后对目前研究领域中采用模型检测技术进行 WCET 分析的相关工作进行介绍。

3.1.1 模型检测技术

随着信息与通信技术的发展，以及用户需求的提升，目前信息通信系统的复杂性迅

速提高，这就给复杂系统的设计带来了很大的挑战。对于任何系统的设计，最基本的要求就是所设计的软硬件系统能够满足用户提出的功能性和非功能性指标。系统设计的错误通常会带来严重的经济损失或灾难性后果。例如，在 90 年代，Intel 公司的 Pentium II 处理器曾经由于浮点除法运算的错误，导致了 4.75 亿美元的损失用来为用户更换掉出错的处理器[66]。而 1996 年 Ariane-5 火箭发射过程中，由于一个 64 位浮点数向 16 位整点数的转换问题而导致了火箭在发射 36 秒之后爆炸[67]。因此，保证系统设计的正确性具有重大意义，对于具有高安全性要求的实时嵌入式系统就更加重要[68]。

通常，对系统正确性的分析叫做系统验证（System Verification）。所谓验证，就是通过各种技术手段，分析所设计的系统是否符合事先提出的功能性及非功能性需求描述（specification）。由于一个信息处理系统通常由软件和硬件两部分组成，系统验证也通常被分为软件验证和硬件验证。传统的软件验证方法主要是代码评审（Peer Review）和软件测试（Software Testing）[69-71]。代码评审主要是采用人力对已经编写的代码的高级语言源程序进行纠错，平均能够达到 60%的纠错率，但是人工方式很难用于分析并行程序等。软件测试指的是实际运行所设计的程序，通过设计不同的数据输入，使得测试过程能够尽可能的覆盖所有的程序状态。硬件验证主要有仿真（Emulation）和模拟（Simulation）两种方式[72,73]。仿真验证是采用 FPGA 等可重构器件来模仿实际硬件的执行，通过给仿真硬件提供不同的激励来验证硬件工作的正确性。模拟验证是使用硬件描述语言设计出功能与硬件相同的软件模型，通过验证软件模型的正确性来验证对应硬件设计的正确性。

上述传统验证方法有两大主要问题。第一是，所有基于模拟和类似模拟的验证手段，都无法真正保证所有的系统状态都被覆盖，也就是说，通过验证的系统仍旧可能存在潜在的错误。第二个问题是验证效率的问题，通常验证工作在系统设计中占据将近 50%的时间和成本开销。随着系统复杂性的提高，这两个问题将更加突出。为此，研究领域和工业领域开始尝试采用形式化方法对系统进行验证[74]。形式化验证的核心内容就是采用数学手段对系统进行精确的建模和描述，并通过形式化验证工具对上述模型进行验证。模型检测技术首先由 Clarke 等人[75]和 Queille 等人[76]分别独立提出，目前在研究和工业领域得到了广泛关注。模型检测技术是一种自动化的验证技术，给定一个系统的有限状态模型以及形式化的属性描述，模型检测器通过探索所有可能的系统状态来验证所要求的属性是否得到满足[77]。

我们借助图 3.1 简要介绍使用模型检测技术验证系统属性的主要步骤[77]。第一步是系统建模。建模就是采用数学手段为系统建立模型，模拟系统行为。通常一个系统被建模成为一个有限状态自动机，建模语言可以采用 C，Java，VHDL 等。在获得了系统模

型之后，需要对系统需求进行形式化描述。我们通常采用属性描述语言来完成这一功能。目前在模型检测技术中，最常见的是基于时间逻辑（temporal logic）的描述语言[78]，采用这种语言可以描述系统的功能正确性（functional correctness）、特定状态可达性（reachability）、安全性（safety）、存活性（liveness）、以及实时特性（real-time properties）等。在有了“系统模型”和“属性描述”后，我们可以把这两种信息输入给模型检测器，模型检测器将按照“系统模型”的描述模拟系统的执行，并分析所有可能的系统状态，最终判定所设计的系统是否符合“属性描述”。如果符合，模型检测器会给出“属性满足”的返回结果；如果不符合，那么模型检测器将报告错误，同时给出造成错误的反例。通过模拟执行所给出的反例，就很容易确定错误所发生的位置。

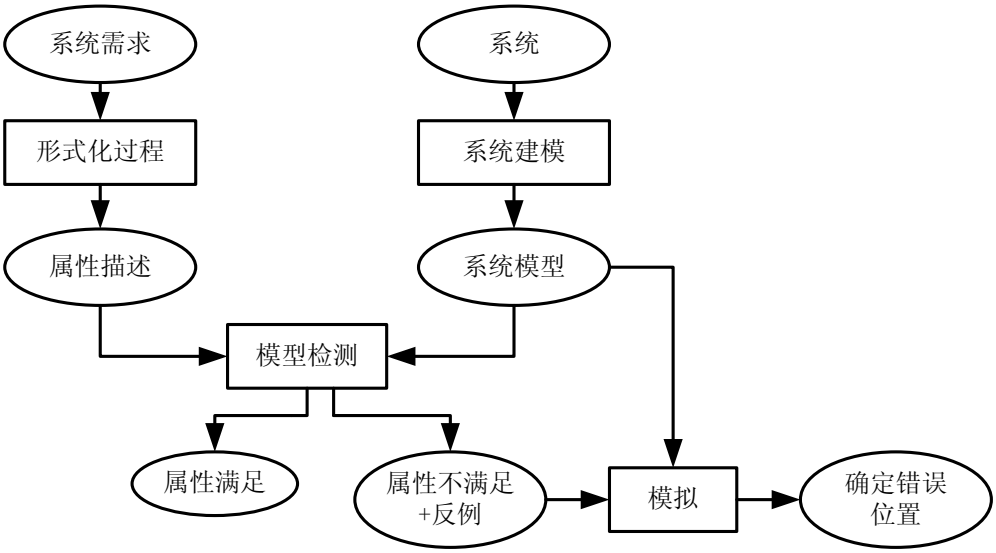


图 3.1 系统描述与基于模型检测技术的验证

Fig. 3.1 System specification and verification via model checking

模型检测技术的优点主要有以下几点[77]。首先它是一种通用的验证方法，可以用来验证很多种类的应用。其次，模型检测支持验证系统的局部特性；再次，模型检测技术以数学理论为基础，保证覆盖所有的系统状态；最后，模型检测器在属性不满足的情况下都会给出反例，这对于调试系统有很大的帮助。但是模型检测技术也存在一些弱点[77]。模型检测技术通常适用于控制密集型应用的验证，而不适用于数据密集型应用的验证，同时它要求被验证的问题应该是可确定的（decidable）。模型检测技术最大的问题就是状态空间爆炸问题，如果系统状态空间爆炸，那么模型检测器将由于没有足够的内存而无法完成验证。同时采用模型检测技术验证系统，同样要求验证者具有必要的技术基础，以保证设计出正确且合理的系统模型。

目前，模型检测技术在软件验证[79-83]和硬件验证[84-86]领域已经取得了一些很有意义的结果。近年来的一个显著的趋势，是采用模型检测技术求解实时系统中的优化问题

[87-90]。由于模型检测器保证覆盖系统的所有状态，因此在需要得到系统的最优解的情况下，模型检测技术就能发挥其作用。同时，模型检测技术具有强大的描述能力，能够描述非常复杂的系统行为，这也是它在优化领域得到应用的主要原因。本章以及第 4 章、第 5 章的工作，就是利用模型检测技术解决“求解程序 WCET”这一优化问题。

3.1.2 基于模型检测技术的 WCET 分析

对于模型检测技术是否适合于 WCET 分析的讨论，首先由 Wilhelm 在文献[91]中提出。作者对三类主要的 WCET 分析手段进行了对比分析，包括“基于抽象解释和整数线性规划技术的分析方法”、“基于模型检测技术的分析方法”、以及“单纯采用整数线性规划技术的分析方法”。Wilhelm 指出：“采用抽象解释对处理器行为进行分析，并采用整数线性规划进行 WCET 计算”技术具有最好的分析效率，同时具备不错的分析精度；而基于模型检测技术和单纯采用整数线性规划技术的分析方法则存在可伸缩性问题，不适用于大型程序[91]。但是作者并没有给出任何实验数据以分析在实际应用中三种技术的性能差异。

针对 Wilhelm 的观点，Metzner 提出了模型检测技术能够应用于 WCET 分析，并具有更好的分析精度[49]。作者提出了一种采用模型检测技术进行 WCET 分析的基本框架，并给出了采用模型检测器模拟执行的方法分析 Cache 的思路。但是作者并没有开展充分的实验，因此也就没有对模型检测技术在 WCET 分析中所遭遇的可伸缩性问题进行透彻的分析。

Ouimet 等人采用 UPPAAL 模型检测器对抽象时间状态机语言（Time Abstracted State Machine Language）的程序进行了 WCET 和 BCET 的分析[92]。TASM 语言是一种系统级的描述语言，它可以将对系统的功能性描述和非功能性描述集成在一种语言中。作者开发了一个从 TASM 语言源程序向 UPPAAL 模型的转换器，将任何 TASM 程序转化为对应的 UPPAAL 模型，并通过迭代有界存在性（Iterative Bounded Liveness）来描述程序的 WCET。由于 TASM 语言的特殊性，以及实验中涉及的 TASM 程序都相对简单，所以采用 UPPAAL 能够取得较好的分析效率。但是这种实验结果显然不能够反映采用模型检测技术分析常见高级语言程序（如 C/C++，Java 等）的分析效率与精度问题。

从上述相关工作可以看出，目前研究领域正在探索采用模型检测技术进行 WCET 分析的可能。众所周知，模型检测技术有潜在的可伸缩性问题，且这一问题的严重程度往往和待解决的具体问题有紧密的联系。因此，探究模型检测技术用于 WCET 分析的实际效率，并具体分析影响分析效率的因素就尤为重要。现有工作普遍缺乏充分的实验数据来验证模型检测技术的分析效率，因而也无法有效的分析可能影响分析效率的主要

因素。此外，不同的模型检测器往往具有不同的工作原理，因此在分析效率上也不尽相同。对于这种差异，目前也缺少必要的分析。针对以上问题，本章工作旨在提出一种基于模型检测技术的 WCET 计算框架，并采用不同模型检测器进行建模，通过具体的实验数据分析模型检测技术在 WCET 分析中的性能和可扩展性。

3.2　基本概念与定义

在第 2 章中，我们简要介绍了程序的控制流程图（CFG）的概念，以及静态分析中 WCET 计算子任务的主要功能。本节将说明本章所分析的程序应满足的一些假设，同时对程序的 CFG 进行形式化的定义。这些定义将在后面小节中被使用。

3.2.1 假设与问题描述

由于分析技术或分析工具的限制，目前的 WCET 分析技术尚不能处理常见高级语言（例如 C、Java 语言等）所支持的所有程序特性。这里，我们通过对待分析的程序进行假设，来明确本章所能够分析的程序种类。我们采用大多数该领域研究工作普遍采用的假设集，具体如下：

假设 1：待分析程序是已经编译好的可执行文件，且对所有库函数的调用都采用静态链接（statically linked）；

假设 2：循环仅以 for 或 while 的形式实现，且循环上限已知；

假设 3：不允许递归程序；

假设 4：抽象处理器模型完美，即处理器行为分析能够给出安全结果。

下面对上述假设进行简要的解释。假设 1 要求待分析程序是采用静态链接方式编译好的可执行程序，主要是因为如果采用动态链接库的话，程序中调用到的库函数与用户程序本身存在于不同的文件。由于目前条件所限，我们仅能分析静态链接的程序，故作此假设。假设 2 限制了循环结构的实现形态。实际上从高级语言的层面，可以采用很多方式（包括采用 goto 语句）来实现循环，而编译成机器指令之后的形态应该是一致的。这里假设循环以 for 或 while 的形式实现，是为了讨论方便。同时，循环上限的分析是“程序流程分析”子任务所需要完成的工作，由于本章主要研究“WCET 计算”问题，因此假设循环上限已知。假设 3 禁止递归程序的实现，是因为目前我们所研究的技术尚不支持递归程序。假设 4 的含义是所采用的处理器行为分析技术是安全的，也就是说它所给出的程序基本块的执行时间的估计值不会小于实际的 WCET，这是因为“处理器行为分析”不是本章所要讨论的内容。

在上述假设之下，我们将给出“WCET 计算”的一个基本描述。“WCET 计算”的

实质是一个最优化求解问题：给定程序对应的 CFG、CFG 中每个基本块的 WCET、以及用户输入或程序分析得到的程序流程信息，找到程序执行时间的最大值以及导致这一最大值的路径（称之为“最坏情况路径”）。

3.2.2 一个简单的示例程序

图 3.2 是一个简单的示例程序，在本章的讨论中，我们采用这一简单程序为例，来具体解释抽象定义，以及各种模型。示例程序的源程序采用 C 语言编写。其主要程序结构是一个 while 循环，i 是循环计数器，这里可以看出循环上限为 10。循环体主要由一个 if-then-else 分支构成。图 3.1 同时给出了该程序所对应的 CFG。这个 CFG 实际上是将高级语言源程序首先编译为可执行代码，然后从可执行代码提取出来的。由于示例程序结构简单，且编译过程没有使用任何优化选项，所以提取出来的 CFG 基本保持了高级语言所表现出来的程序结构。

```
void main()
{
        int b;
        int i = 0, j = 0;
        while (i < 10){
                if (b)
                        j++;
                else
                        j--;
                i++;
        }
}
```

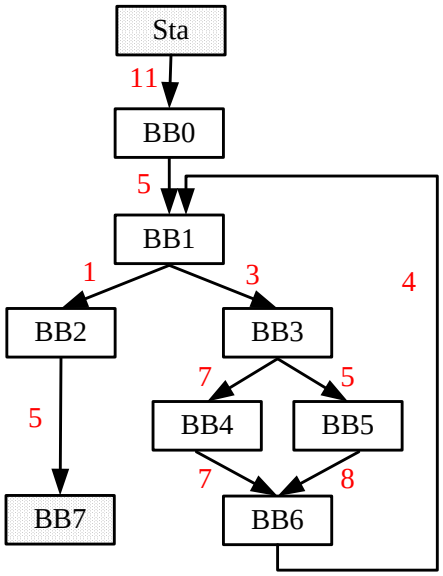


图 3.2 示例程序及其 CFG

Fig. 3.2 A motivating example and the corresponding CFG

CFG 图中的每个节点叫做一个“基本块”，它是由一系列的顺序指令组成；CFG 图的边表示程序的执行流程。这里需要特别注意的是，基本块的执行时间并不是按照传统的方式标记在每个基本块上，而是标记在边上。我们以基本块 BB1 为例解释这一区别。BB1 能够执行有两种可能的情况：一种是执行完 BB0 执行 BB1，另外一种是执行完 BB6 执行 BB1。两条不同的执行路径将为 BB1 的执行制造不同的处理器状态，换句话说，BB0 执行后的 Cache 和流水线中的内容通常情况下是和 BB6 不同的，这就造成了 BB1 执行时间的不同。在图 3.2 的例子中，如果在 BB0 之后执行 BB1，那么后者的

执行时间为 5；如果在 BB6 执行 BB1，那么后者的执行时间为 4。如果以二者的最大值 5 来代表 BB1 的执行时间，那么结果一定是安全的；但是如果区分不同路径来表示，WCET 的计算结果将更加精确。进行 WCET 计算的前提是进行程序 CFG 的提取，在本章的实验中，这一工作由 Chronos[13]工具来完成。而 Chronos 工具采用了上述方法，因此在本章的讨论中，基本块的时间根据路径不同而不同，且标记在对应基本块的入边上。如果把原 CFG 图的节点都变成边，边都变成节点（可能需要增加空初始/终止节点），那么就能得到一个在点上标记有执行时间的 CFG 图。因此，上述区别并不影响 WCET 计算问题的本质。基于这种执行时间的标记方法，需要给整个程序增加一个空的初始节点（执行时间为 0），使得这个初始节点到 CFG 实际初始节点的边上标记有实际初始节点的执行时间。

3.2.3 程序控制流程图的形式化定义

这里我们对上面介绍的程序 CFG 进行形式化的定义。特别的，对程序中的循环结构进行专门的定义。

定义 3.1：程序的 CFG 是一个五元组 $CFG=(B,Sta,Ter,T_B,L)$，其中各元素定义为：

- B：一个程序 CFG 中所有基本块的集合，其中第 i 个基本块表示为 BB_i；
- Sta：程序 CFG 的唯一起始节点，$Sta \in B$；
- Ter：程序 CFG 的唯一终止节点，$Ter \in B$；
- T_B：程序 CFG 的所有边的集合，$T_B \subseteq B \times B$，我们采用 t_{i_j} 表示从基本块 BB_i 到基本块 BB_j 的边；
- L：程序 CFG 中所有循环的集合，其中任何一个循环 $Loop_i$ 的定义见下文。

基于上述 CFG 的基本定义，我们进一步定义一些算符：$cost(t_{i_j})$返回边 t_{i_j} 的值，也就是从基本块 BB_i 到达 BB_j 的情况下，BB_j 的执行时间；$src(t_{i_j})$和 $dst(t_{i_j})$分别返回边 t_{i_j} 的起始节点和终止节点，也就是 BB_i 和 BB_j。在此基础上，可以形式化定义循环结构。

定义 3.2：程序循环体是一个八元组，$Loop=(BODY,EDGE,Head,Tail,Tbj,BL,BE,lpb)$，其中各元素的含义如下：

- $BODY$：一个循环体所包含的所有基本块的集合，$BODY \subseteq B$；
- $EDGE$：循环体所包含的所有边的集合，$EDGE \subseteq T_B$；
- $Head$：循环体的头节点，也就是从入口进入循环后第一个被执行的基本块，$Head \in BODY$；
- $Tail$：循环的尾节点，当循环的尾节点执行完毕后，循环将跳转到头节点执行，$Tail \in BODY$；

- *Tbj*：从循环尾节点到循环头节点的返回边，$Tbj \in EDGE$；
- *BL*：循环的所有入口节点的集合，入口节点执行完毕后将执行循环体，但是入口节点不是循环体的一部分，$BL \subseteq B$ 且 $BL \not\subset BODY$；
- *BE*：循环的所有出口节点的集合，循环体跳出后执行的第一个基本块为循环的出口节点，出口节点不是循环体的一部分，$BE \subseteq B$ 且 $BE \not\subset BODY$；
- *lpb*：循环上限值，$lpb \in \mathrm{N}$。

以图 3.1 的程序中的循环为例，循环体基本块集合 *BODY={BB1, BB3, BB4, BB5, BB6}*，循环体的头节点 $Head=BB_1$，循环体的尾节点 $Tail=BB_6$，循环体的返回边 $Tbj=t_{6\_1}$，循环体的入口节点集合 $BL=\{BB_0\}$，循环体的出口节点集合 $BE=\{BB_2\}$，循环上限 *lpb=10*。

3.3　基于模型检测技术的 WCET 计算技术

本节首先介绍采用模型检测技术求解 WCET 的基本思路，其次采用三种不同的模型检测器（SPIN，NuSMV，UPPAAL）分别对该问题进行建模。

3.3.1 采用模型检测技术求解 WCET 的基本思路

采用模型检测技术求解 WCET 的基本思路是：让模型检测器模拟待分析程序的执行，模型检测器会自动覆盖所有的执行可能，在本章的讨论中，就是程序的所有执行路径；然后从中选择执行时间最长的路径，并报告最长执行时间值。为使模型检测器能够模拟程序的执行，首先要做的就是将程序的执行过程建模为一个有限状态自动机，这个有限状态自动机就是大多数模型检测器所能够接受的输入。一个程序能够满足 3.2.1 小节的假设就能保证程序停机，并且程序的状态是有限的，因此这样的程序都可以转化为对应的有限状态自动机。这个有限状态自动机主要从程序的 CFG 转化而来，通过添加必要的控制流程的语义使得自动机能够完整的模拟程序的运行。这种创建自动机的思路和相关工作[49]中的做法是类似的。我们称这个自动机为基本块自动机（Basic Block Automaton，简称 BBA），下面将具体定义 BBA。

定义 3.3：一个 BBA 可以定义为一个五元组 $BBA=(s_0,s_T,S,V,T)$，其中：

- *S*：自动机所有状态的集合，这里 $S = B$ (CFG 中的基本块集合)；
- s_0：自动机的初始状态，$s_0 \in S$，且 $s_0 = Sta$；
- s_T：自动机的终止状态，$s_T \in S$，且 $s_T = Ter$；
- *V*：控制变量的集合，包括循环次数控制变量，分支控制变量，程序的执行时间等；
- *T*：自动所有状态转换的集合，这里 $T \in S \times S$，且 $T = T_B$。

以上是根据程序的 CFG 直接转化过来的 BBA，我们还要为程序中的循环、分支等结构定义对应的语义，以使得自动机能够正常运行。为记录程序的执行时间，我们引入一个变量 $wcet \in V$，用以记录程序的执行时间。在自动机启动的时候，该值初始化为 0，每当程序执行了一个基本块，该基本块的执行时间都会被累加到 $wcet$ 中，以模拟时间的推进。

我们为每个分支结构定义一个分支控制变量 $cond_i$，分支的两个路径分别对应于 $cond_i$ 为真或为假两种情况。本章不对 $cond_i$ 具体赋值，也就是说，在我们定义的 BBA 中分支结构被建模成不可确定的：当遇到一个分支的时候，分支的两条路径都有可能被执行。而模型检测器负责探索所有的可能。这是一种通常的做法。实际上，如果 WCET 控制流分析子任务能够得到一些关于分支走向和不可行路径的分析结果，我们可以将这些结果通过设置分支控制变量值建模到 BBA 中。由于控制流程分析子任务不是本章的研究重点，因此这里不作详细讨论。

除了分支，还需要对循环进行详细的建模。图 3.3 表示的是从一个 CFG 的循环结构向 BBA 的循环结构的转换。对于任何一个循环 $Loop_i$，我们为其定义变量 $lpb_i \in V$ 表示循环上限，同时定义变量 $lpc_i \in V$ 来记录当前时间循环已经执行的次数。每次进入循环，lpc_i 变量都将被置 0，当 BBA 执行从循环尾节点到循环头节点的迁移的时候，lpc_i 变量都会加 1，表明一次循环体执行完毕。在本章的讨论中，假定循环的判断条件在头节点中执行，因此在头节点向循环体的下一个节点的迁移边上需要添加 $lpc_i < lpb_i$ 条件，该条件的语义是“只要循环次数不超过循环上限，就可能再次执行循环体”。

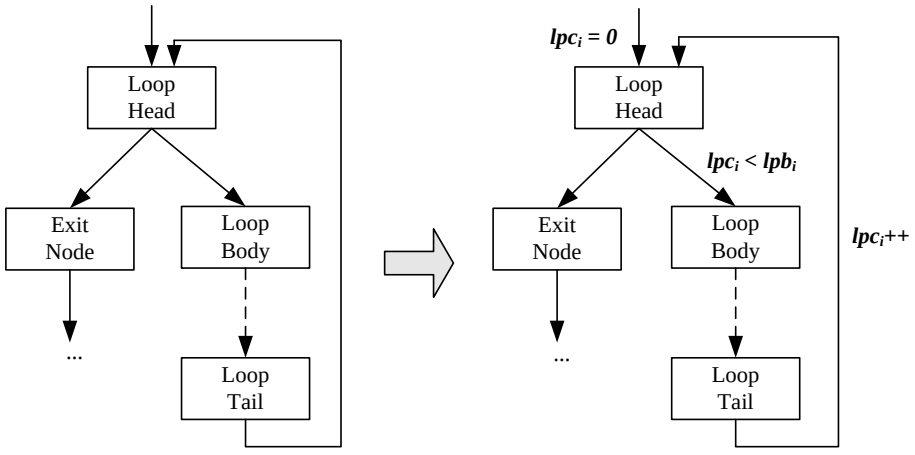


图 3.3 循环结构的语义转换

Fig. 3.3 Semantic transformation of loops

将以上内容组合起来，我们可以形式化的定义 BBA 的状态迁移语义。

定义 3.4：对于 BBA 的任何一个状态迁移 $t_{i\_j} = (BB_i, BB_j) \in T$，迁移条件和对应的

行为定义如下：

$$guard(t_{i\_j}) = \begin{cases} cond_k \in BOOLEAN & if\ \forall m, BB_i \notin Head_m \\ & \wedge \exists BB_n, \exists t_{i\_n} \in T_B \ \wedge \ guard(t_{i\_n}) = \neg cond_k \\ \neg cond_k & if\ \forall m, BB_i \notin Head_m \\ & \wedge \exists BB_n, \exists t_{i\_n} \in T_B \ \wedge \ guard(t_{i\_n}) = cond_k \\ (lpc_k < lpb_k) & if\ \exists k,\ Loop_k \in L \ \wedge \ BB_i = Head_k \\ & \wedge BB_j \in BODY_k \ \wedge \ BB_j \neq Head_k \\ TRUE & else \end{cases}$$

$$action_{loop}(t_{i\_j}) = \begin{cases} \{ lpc_k := 0 \} & if\ \exists k, Loop_k \in L \\ & \wedge BB_j \in BL_k \ \wedge \ t_{i\_j} \neq Tbj_k \\ \{ lpc_k ++ \} & if\ \exists k, Loop_k \in L \\ & \wedge t_{i\_j} = Tbj_k \\ \Phi & else \end{cases}$$

$$action_{time}(t_{i\_j}) = \{ wcet := wcet + cost(t_{i\_j}) \}$$

$$action(t_{i\_j}) = action_{loop}(t_{i\_j}) \cup action_{time}(t_{i\_j})$$

以上定义了一个完整的 BBA，这个自动机可以被大多数的模型检测器所接受，将自动机转换成模型检测器的模型描述，就可以让模型检测器模拟程序执行。通常情况下，模型检测器自身不能够直接寻找程序所有可能的执行时间的最大值，它们通常接受类似“程序的 WCET 小于 100 吗？”这样的问题，并回答“是”或“否”。因此我们定义一个线性时间逻辑（Linear Time Logic，简称 LTL）的断言 $[]\phi N$，这里 $\phi N = (wcet \le N)$。这个 LTL 的断言所表达的含义是“对于从起始节点出发的所有可能的执行路径，$(wcet \le N)$ 永远成立”。将这个属性交给模型检测器去检测，它会给出这个断言的真值。假定程序实际的 WCET 值为 100，那么当 N=100 时，$[]\phi N$ 断言显然成立；而当 N=99 时，$[]\phi N$ 断言显然不成立，因为程序存在一条路径其执行时间大于 99。利用这一性质，我们就可以确定程序的 WCET 值是不是某个给定的值（N）。只要在一个值域范围内进行二分搜索，就可以找到这个值。二分搜索的算法如图 3.4 所示。

与二分搜索有关的另外一个问题就是搜索上限和下限的确定。假定搜索次数为 n，搜索的上限和下限分别为 UB 和 LB，那么 $n = \log_2(UB - LB)$，因此搜索区间的确定将影响搜索的次数。一个程序在分析之前是很难预知其执行时间的，假设一个很小的 WCET 下限和一个很大的 WCET 上限将导致搜索范围过大，搜索次数过多。寻找下限的一个可行的办法就是模拟执行程序。前文已述，模拟执行一个程序可以得到它的部分情况下的执行时间，称其为 WCET 观测值。WCET 观测值一定小于程序实际的 WCET，所以

这个值可以作为下限值 *LB*。在没有更多关于 WCET 估计值信息的情况下，我们可以采用步进的办法设定一个正确的上限值。我们可以测试（LB×2）是否大于 WCET，测试的办法就是用模型检测器检测 $[]\phi(LB\times 2)$ 和 $[]\phi(LB\times 2-1)$ 两个属性，如果都返回“满足”，那么（LB×2）就是一个正确的上限值；如果两个属性都“不满足”，那么我们可以调整下限为（LB×2）。这样迭代下去，就可以找到一对正确的下限和上限值。

算法3.1　二分法搜索程序的WCET值

输入：某个模型检测器所能接受的模型描述，以及值域[LB, UB]
输出：程序的WCET值

```
设定搜索的上限和下线, lower_bound = LB, upper_bound = UB;
while (lower_bound < upper_bound - 1)
    middle = (lower_bound + upper_bound) / 2;
    检测断言 []φ(middle) ;
    if (断言 []φ(middle) 成立)
        upper_bound = middle;
    else
        lower_bound = middle;
end while
return upper_bound;
```

图 3.4 二分搜索程序 WCET 的算法
Fig. 3.4 Finding WCET with binary search

采用上述方法得到的上限值一定是下限值的 2 倍，因此循环次数实际上就取决于找到的下限值的大小。如果这个值比较大，那么搜索次数将会很多。实际上我们还可以采用其他的办法，设置更加接近实际 WCET 的上限值。通常情况下，采用模型检测技术分析 WCET，往往是为了获得更加精确的结果。而实际上，现有的一些 WCET 静态分析技术可以在很短的时间内计算出一个 WCET 估计值，只是这个估计值没有模型检测技术分析出来的精确。因此我们可以先采用其他技术分析得到一个 WCET 估计值，这个值能够保证大于程序实际的 WCET。于是我们把模拟执行的结果作为下限，把采用其他技术分析得到的 WCET 估计值作为上限，这样的一个搜索范围一定是更加精确的。

上面完整的介绍了采用模型检测技术计算 WCET 的基本思路，完成了从 CFG 到 BBA 的转换，定义了 BBA 的语义，同时给出了基于二分搜索的模型检测优化步骤。下面各小节将基于上述思路，分别采用不同的模型检测器对 WCET 计算问题进行建模。

3.3.2 采用 SPIN 模型检测器建模

SPIN[93,94]是一个显式状态的在线模型检测器。所谓“显式状态”指的是模型对应的

状态空间都是直接存储的，每个状态对应一个数据结构，不存在多个状态规约表示的情况。所谓“在线”指的是，SPIN 在运行的过程中，随着状态空间的探索，实时的创建状态空间。SPIN 模型检测器目前仅支持线性时间逻辑（LTL）描述的断言。

```
// 变量定义
int wcet, lpc1;

// 程序对应的BBA自动机
proctype BBA()
{
Sta: atomic {
        wcet = wcet + 11;  goto S0;
}
S0: atomic {
        lpc1 = 0;  wcet = wcet + 5;  goto S1;
    }
S1: atomic {
        if
        ::(lpc1 <= 9) -> wcet = wcet + 4;  goto S3;
        ::(lpc1 >= 10) -> wcet = wcet + 1;  goto S2;
        fi;
    }
// S2~S7状态略去
}

// 初始化
init
{       atomic{
                wcet = 11;
                lpc1 = 0;
                run CFG();
}}

// LTL断言描述
#define BOUND 1000
#define p (wcet ≤ BOUND)

never{
T0_init:
        if
        ::(!((p))) -> goto accept_all;
        ::else -> goto T0_init;
        fi;
accept_all:
        skip
}
```

图 3.5 示例程序对应的 SPIN 模型

Fig. 3.5 The SPIN model of the demo program

图 3.5 为图 3.2 中的示例程序转化而来的 SPIN 模型。这里首先定义了一些必要的变量。变量 *wcet* 用来记录程序的执行时间。所有的 *lpc**形式的变量表示的是循环计数器。由于循环上限值事先已知，因此已经直接写入 SPIN 模型的代码。其中 proctype BBA() 定义了一个 SPIN 进程，该进程的行为就是根据 BBA 定义的语义模拟程序的执行。其中由行号“Si:”标记的是 BBA 的状态，在 atomic{}里面的所有语句执行了当前状态下所规定的程序语义，同时更新执行时间值 *wcet*，然后根据程序语义跳转到下一个目标状态。init()进程是 SPIN 要求的不可缺少的进程，主要负责初始化所有定义的变量。

SPIN 支持类似 C 语言语法的断言描述。图 3.5 里的 never{}定义就实现了一个 LTL

的断言，这里[]ϕN 断言被定义为 p，也就是($wcet \le BOUND$)。在模型检测器进行状态空间探索的过程中，每执行一步，都要执行 never{}定义的程序。如果发现断言[]ϕN 不满足，那么检测过程将终止，返回 error 1，同时给出造成断言不满足的反例；如果在所有的可能都检测完毕且断言[]ϕN 没有被证明是错的，那么返回 error 0，表示该断言成立。

3.3.3 采用 NuSMV 模型检测器建模

```
// 变量定义
MODULE main
VAR
state:{Sta, SS0, SS1, ...};
wcet:0..5000;
lpc1:0..50;
// 初始化
ASSIGN
init(state):=Sta;  init(wcet):=0;  init(lpc1):=0;
// 程序对应的BBA自动机
TRANS
(
  (    (state = Sta) -> (
       next(lpc1) = lpc1 &
       next(wcet) = wcet + 11 &
       next(state) = SS0
        )
  ) &
  (    (state = SS0) -> (
       next(lpc1) = lpc1 &
       next(wcet) = wcet + 5 &
       next(state) = SS1
        )
  ) &
  (    (state = SS1) -> (
       next(lpc1) = lpc1 &
       (lpc1 <= 9) -> (
           next(wcet) = wcet + 4 &
           next(state) = SS3 ) &
       (lpc1 >= 10) -> (
           next(wcet) = wcet + 1 &
           next(state) = SS2 )
        )
  ) &
-- SS2 - SS7 omitted due to limited space
)
// CTL断言描述
SPEC AG (wcet <= BOUND)
```

图 3.6 示例程序对应的 NuSMV 模型

Fig. 3.6 The NuSMV model of the demo program

NuSMV[95,96]是由 CMU 大学所设计的著名的 SMV 模型检测器的改进版，同时增加了很多新的功能。NuSMV 有着与 SPIN 不同的工作原理，它的主要特点是采用了二叉决策图（Binary Decision Diagram，简称 BDD）来符号化的表示状态空间。其本质是对状态空间描述中的重复部分的化简，并用 BDD 数据结构进行表示。需要特别注意的是，

BDD 是对空间描述的化简，它并不能够削减状态空间。由于采用了 BDD 技术，NuSMV 在检测的过程中所使用的存储空间要明显小于 SPIN，这在后面的实验数据中可以看出。NuSMV 支持多种建模风格，而本章采用的是基于命题表达式（propositional formulas）的有限状态自动机的建模方法，因为这与 BBA 的语义是直接对应的。

图 3.6 是我们采用模型生成器自动生成的示例程序的 NuSMV 模型。VAR 部分定义了需要用到的变量，其中程序状态 *state* 是一个枚举类型。NuSMV 与 SPIN 的本质不同是，前者要求必须为每一个变量指定值域范围，否则 NuSMV 无法完成检测。ASSIGN 部分相当于 SPIN 模型中的 init()进程，主要负责初始化变量。TRANS 部分定义了 BBA 的结构和迁移语义。对于 BBA 的每个状态，(state = SSi) -> next(…)语句表示“如果 BBA 的当前状态是 SSi，那么所有变量的值应按照 next(…)指定的规则修改”。在自动机的每一次执行中，所有的状态都将被分析，只有符合要求的状态才能够执行对应的操作。

NuSMV 同时支持线性时间逻辑（LTL）和计算树逻辑（Computational Tree Logic，简称 CTL）两种断言描述方法。在我们的实验中，采用基于 CTL 的断言描述。这里，AG (wcet <= BOUND)所表示的含义是：“对于所有从初始状态出发的路径上的任何一点，(wcet <= BOUND)永远满足。令 NuSMV 模型检测器读入上述模型，就可以验证出断言的正确性。类似的，对于断言不满足的情况，NuSMV 会给出所有反例中的一个。

3.3.4 采用 UPPAAL 模型检测器建模

UPPAAL 是瑞典 Uppsala 大学与丹麦 Aalborg 大学联合开发的模型检测器[97,98]，它是基于时间自动机[99]理论工作的。时间自动机是有限状态自动机的一种扩展，它允许在自动机中定义用来描述时间的时钟变量。UPPAAL 允许用户在模型中定义多个时钟变量，这些时钟以同步的方式向前推进。UPPAAL 所采用的属性描述语言是 CTL 的一个子集。在 UPPAAL 中，一个完整系统通常由多个时间自动机组成，多个自动机通过 UPPAAL 定义的通信语义进行同步。由于 UPPAAL 自身的基础理论就已经具备了对时间建模的能力，因此就非常适用于实时系统的验证。同时，UPPAAL 也可以用作验证非时间系统（untimed systems），它的建模能力比大多数的模型检测器要强大。

图 3.7 是示例程序对应的 UPPAAL 模型。UPPAAL 提供给用户一个 GUI 以方便设计。图 3.7 就是图形化显示的 UPPAAL 模型，而实际的文件是以 XML 文档存储的。我们可以从这个图中看到自动机的结构与程序的 CFG 是一一对应的。整个模型定义了两个时钟，时钟 c 用来描述程序在每个基本块上的执行时间，时钟 gc 用来记录已经流逝的时间。整个模型有一个标志变量 flag 用以标记系统的启动与停止，它在系统启动的时候被赋值为 1，在系统进入结束状态的时候被赋值为 0。

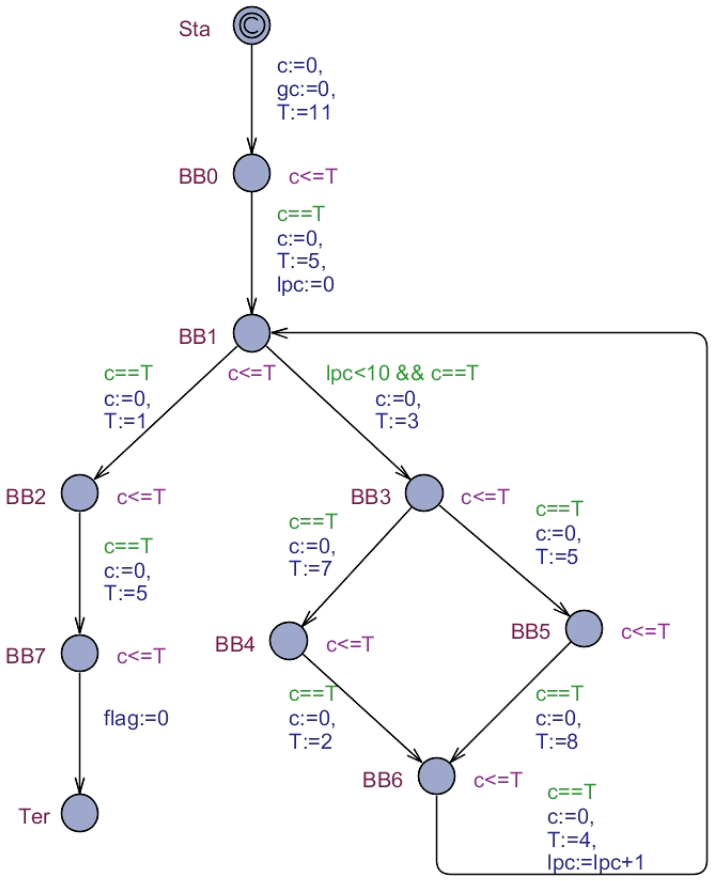


图 3.7 示例程序对应的 UPPAAL 模型

Fig. 3.7 The UPPAAL model of the demo program

在该时间自动机的每个节点（location）上都有一个表达式(c<=T)，这叫做节点的不变量（invariance），它所表达的含义是当时钟 c 推进到大于 T 的时候，表达式将不成立，那么状态机必须离开当前状态。实际上这里的 T 就是每个基本块的执行时间，我们用这种不变量的方式来描述基本块的执行时间信息。在每条边上的绿色标记是迁移条件（guard），只有当迁移条件满足的时候，迁移才能够完成。这里除了控制循环次数的条件以外，主要还有(c==T)这个条件，该条件配合节点上的不变量，保证程序在某个基本块上停留了最长允许的时间 T 后，会立即完成状态迁移。边上的其它表达式是变量值更新，它描述了当一次状态迁移发生的时候，模型定义的何种变量值将发生何种变化。在我们的模型中，将局部时钟 c 置 0，目的是使其可以为下一个基本块记录执行时间；此外，由于我们的 CFG 中基本块的执行时间放在了入边上，所以还要对变量 T 进行赋值，以描述对应基本块的执行时间。

UPPAAL 支持蕴含表达式形式的属性描述（这种描述实质上就是一种 CTL 的描述）。我们让模型检测器验证属性“flag==1 --> gc<=BOUND”，该属性的含义是：在自动机所有执行路径上的任何一点，(gc<=BOUND)都成立。因此当 BOUND 的值大于等于程序实际的 WCET 时，验证结果为“真”；当 BOUND 的值小于程序的实际 WCET 时，

验证结果为“假”。通过这种方式我们就可以确定程序的 WCET 是多少。

3.4　基于隐式路径枚举技术的 WCET 计算技术

隐式路径枚举技术在很多 WCET 分析中被广泛采用。在这项技术中，通常将 WCET 最大值的求解问题描述为一个整数线性规划的问题。在第 2 章中，我们已经简要介绍了隐式路径枚举技术的基本原理，这里我们将根据 3.2.3 小节的形式化定义，详细介绍在给定程序 CFG 和每个基本块执行时间的情况下，如何生成对应的整数线性规划问题。

这里，我们定义 CFG 中的边 t_{i_j} 的执行次数为 x_{i_j}，那么整个程序的执行时间实际上就是所有边的执行时间乘以其执行次数的总和；让这个总和取得最大值，也就可以求出程序的 WCET。因此，求解 WCET 的整数线性规划问题的目标表达式为：

$$wcet = MAX \sum_{t_{i\_j} \in T_B} cost(t_{i\_j}) \times x_{i\_j} \tag{3.1}$$

该整数线性规划问题的约束条件主要来自程序的控制流程信息。这里，主要的控制流程信息有三类。第一类信息描述的是程序的 CFG 的结构信息。我们知道，每个基本块的执行次数并不是可以任意取值的，实际上它们受制于程序结构。第一类约束所表达的意思是，对于每个程序基本块，所有入边执行次数的总和应该等于所有出边执行次数的总和。这里，我们引入中间变量 b_i 表示基本块 BB_i 的执行次数，那么将得到如下一组线性约束：

$$\forall BB_i, b_i = \sum_{dst(t_{k\_i})=BB_i} x_{k\_i} = \sum_{src(t_{i\_j})=BB_i} x_{i\_j} \tag{3.2}$$

第二类控制流程信息规定了程序只能进入一次和退出一次，这是通过将 Sta 节点和 Ter 节点的执行次数设置为 1 来实现的。对应的线性约束为：

$$b_{Sta} = 1, \quad b_{ter} = 1$$

第三类控制流程信息描述的是程序的循环上限。例如一个程序的循环上限为 10，这说明程序每进入该循环一次，循环体将最多被执行 10 次。循环入口节点到循环体头节点的执行次数总和能够表示进入循环的次数；而循环的尾节点的执行次数可以用来代表循环体的执行次数。本章所处理的循环假定都具备图 3.2 中循环的形态，也就是循环的跳出判断是在循环的首节点中完成的。因此，循环首节点将比循环体多执行一次。为表示方便，我们采用循环尾节点的执行次数代表循环体的执行次数。因此关于循环上限的约束条件可以表示为：

$$\forall Loop_i, \mathrm{b_{Tail}} \le lpb_i \cdot \sum_{BB_j \in BL_i} t_{j\_head_i} \tag{3.3}$$

图 3.8 是图 3.2 的示例程序所对应的整数线性规划问题的具体描述，这里采用了整数线性规划求解器 lp_solve 所接受的格式。将此问题描述文件作为 lp_solve 的输入，就可以求得程序的 WCET。

```
Max:

11 dSta_0 + 5 d0_1 + 1 d1_2 + 3 d1_3 + 5 d2_7
+ 7 d3_4 + 5 d3_5 + 7 d4_6 + 8 d5_6 + 4 d6_1;

\ === 程序结构约束 ===
dSta_0 = 1;
b0 - d0_1 = 0;
b0 - dSta_0 = 0;
b1 - d1_2 - d1_3 = 0;
b1 - d0_1 - d6_1 = 0;
b2 - d2_7 = 0;
b2 - d1_2 = 0;
b3 - d3_4 - d3_5 = 0;
b3 - d1_3 = 0;
b4 - d4_6 = 0;
b4 - d3_4 = 0;
b5 - d5_6 = 0;
b5 - d3_5 = 0;
b6 - d6_1 = 0;
b6 - d4_6 - d5_6 = 0;
b7 - d2_7 = 0;

b0 = 1;
b7 = 1;
b6 - 10 d0_1 <=  0;    // 循环上限约束

// 整数变量定义，部分省略
int b0, b1, ...
```

图 3.8 示例程序的整数线性规划模型

Fig. 3.8 The integer linear programming model of the demo program

由于隐式路径枚举技术在很多 WCET 分析工具中被采用，因此，在后面的实验中，我们将拿隐式路径枚举技术与本章所采用的模型检测技术进行对比，分析二者在性能等方面的差异。

3.5　实验与结果分析

对于上面提出的基于模型检测技术的 WCET 计算方法，本节采用具体实验验证其性能以及可扩展性等特性。我们首先介绍实验环境，包括分析工具的设计和测试程序集的选取；其次对实验结果进行详细分析；最后，通过实验结果和其他局部实验，我们对基于模型检测技术的 WCET 计算方法的优劣进行客观评价。

3.5.1 实验环境

图 3.9 是本章进行 WCET 分析的完整工作流程。首先 C 语言的源程序被编译为对应硬件平台上的汇编程序，本文的指令集为 PISA。之后 Chronos 工具会对编译后的汇编

程序进行反汇编操作，生成该程序对应的 CFG。同时，Chronos 的处理器行为分析模块会将每个基本块的执行时间运算出来。由于 Chronos 的 Cache 分析和分支预测分析的约束不是直接影响到基本块的执行时间的，而本章所关注的要点是路径分析，因此这两个分析功能在实验中被临时关闭。此外，用户可以通过工具的界面输入循环上限等功能性约束。程序 CFG、基本块执行时间、以及循环上限都生成以后，就可以对程序进行路径分析，寻找执行时间最长的路径及对应的 WCET 值。Chronos 原有的功能是生成整数线性规划的模型，并交给求解器 lp_solve 求解 WCET 值。本章由于采用模型检测技术进行 WCET 计算，因此我们开发了一个模型生成器，该生成器能够根据输入的程序流程信息和基本块的执行时间等分别生成 SPIN、NuSMV、UPPAAL 模型。这些模型分别输入给对应的模型检测器，再结合图 3.4 介绍的二分搜索，就可以得到程序的 WCET 值。

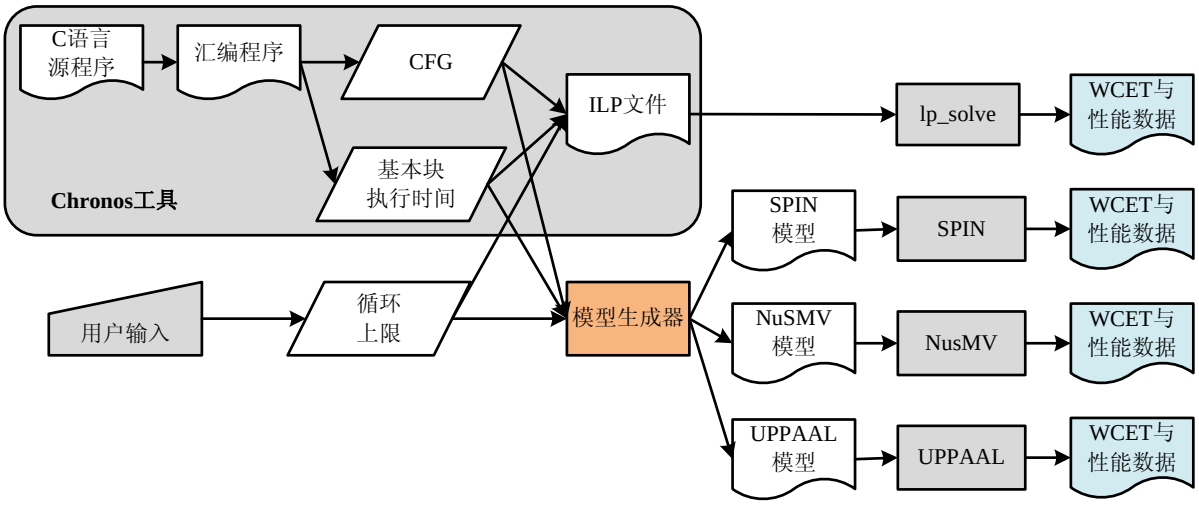


图 3.9 WCET 分析工作流程

Fig. 3.9 The work flow of WCET analysis

表 3.1 列举了本章实验所采用的 10 个测试程序。对于每个程序，我们分别给出了其功能描述、程序长度（C 语言源代码行数）、循环个数、以及最大的循环嵌套层数等信息。上述程序选自瑞典 MRTC 研究组给出的 WCET 测试程序集[100]，它是目前 WCET 研究领域比较权威的测试集。这些程序作为某些子功能，广泛应用于实时嵌入式系统中，因此具有较强的代表性。

3.5.2 实验结果与分析

由于循环是能够影响模型检测 WCET 计算方法的主要因素，因此对于所有程序中的所有循环，我们都忽略程序的实际语义，手工设置循环上限，目的就是探索循环结构对该计算方法的影响程度。对于每一个程序，我们设计了 4 种情况，其中每种情况中所有的循环上限分别被设置为 2、4、8、16。

表 3.1 测试程序说明

Table 3.1 Description of the benchmark program

程序名称	程序描述	程序长度	循环个数	嵌套层数
select	从一个浮点数矩阵中选择大小第 N 位的数	114	4	3
sqrt	求平方根程序	77	1	1
statemate	STARC 工具自动生成的代码	1,276	1	1
edn	FIR 过滤器程序	285	12	3
insertsort	插入排序程序	92	2	2
janne_complex	普通嵌套循环程序	64	2	2
fir	另一个 FIR 过滤器程序	276	2	2
expint	指数函数的计算程序	157	3	2
fdct	快速离散余弦变换函数计算程序	239	2	1
matmult	矩阵乘法运算程序	163	7	3

所有程序运行于 IBM Blade Server 集群服务器，其中每个刀片配置有 2 个 4 核心的 Xeon 1.6GHz 处理器以及 8GB 内存。由于本章实验所采用的 ILP 求解器和模型检测器都仅能在一个核心上运行，因此该平台的多核优势没有被利用，不过这并不对本章实验结果的讨论产生影响。

由于整数线性规划和模型检测技术都能够得到最优解，因此它们计算出来的 WCET 值应该是完全一致的。对于整数线性规划，分析时间都在 1 秒以内，内存使用都在 4MB 以内。对于模型检测技术，我们分别列举它们验证 $[]\phi(WCET)$ 属性以及 $[]\phi(WCET-1)$ 属性所需要的时间和内存。我们采用一个简单实用的时间与内存测试工具 memtime。该工具测试时间的精度是 0.01 秒，测试内存使用量的精度为 1KB。由于模型检测器存在潜在的状态空间爆炸的问题，因此我们为执行时间和内存使用量都设置了上限：如果执行时间超过 36,000 秒，我们就认为分析是“时间不可行”的；如果内存使用超过了 8GB，我们认为分析是“内存不可行”的。下面，我们将对实验得到的数据进行详细分析。

在所有的实验中，对于模型检测技术，无论是分析时间还是内存使用量都无法与基于整数线性规划的技术相比，尤其当程序中循环次数增加（也就是循环上限增加）的情况下，性能差距尤为突出。其主要原因在于，从 CFG 中寻找执行时间最长路径的问题，基本上等价于网络流问题[24]，整数线性规划技术已经被验证对于这类特殊问题具有非常高效的计算性能。而对于模型检测技术而言，无论采取哪种具体的模型检测器，其根本原理都是穷举所有的程序路径，这是造成模型检测技术分析时间慢以及内存使用量大的根本原因。例如，对于一个上限为 100 的循环，如果循环体有一个分支，那么该循环所

有可能的路径将有 2^{100} 个。

但是，单纯比较两种技术在 WCET 计算上的表现是不合理的。在采用整数线性规划技术的 WCET 分析中，对 Cache、流水线等部件的分析会增加变量以及约束的个数。实践证明仅对组相联 Cache 进行分析，可能导致线性约束数量的增加[23]。在这种情况下，基于整数线性规划技术的分析效率将显著降低。而对于模型检测技术，在 WCET 计算阶段已经将程序的所有可能路径进行了枚举，所以即使在模型中对 Cache、流水线等部件进行建模，可能的路径数量也不会改变，状态空间也不会显著增加。因此，如果考虑处理器行为分析这一子任务，那么两种技术的性能差距远远不是本章实验数据所表现的那样。即使模型检测技术仍旧可能在性能上不及整数线性规划技术，但是前者的优点在于丰富的建模能力，因此可以对程序的行为进行更加细致的描述，导致分析结果更加精确。下面，我们分别对三种模型检测器的性能特点进行详细的分析。

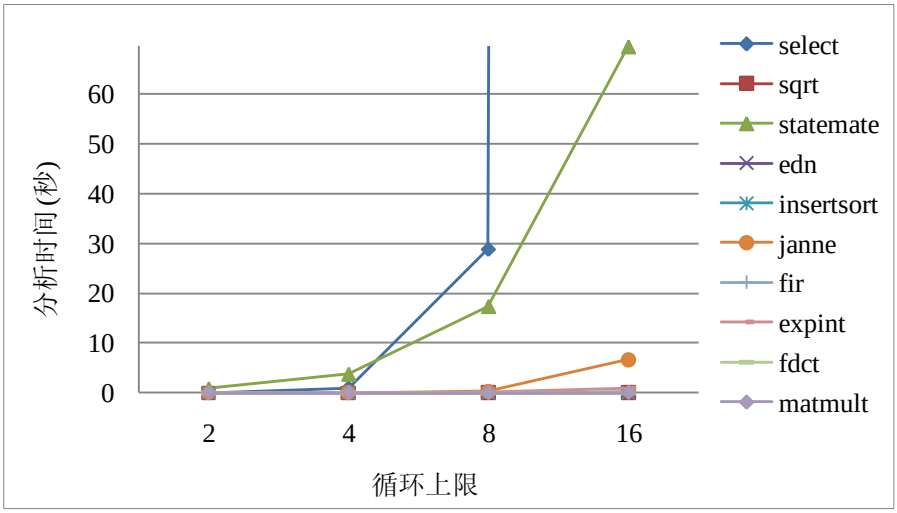


图 3.10 SPIN 分析时间

Fig. 3.10 The execution time of SPIN models

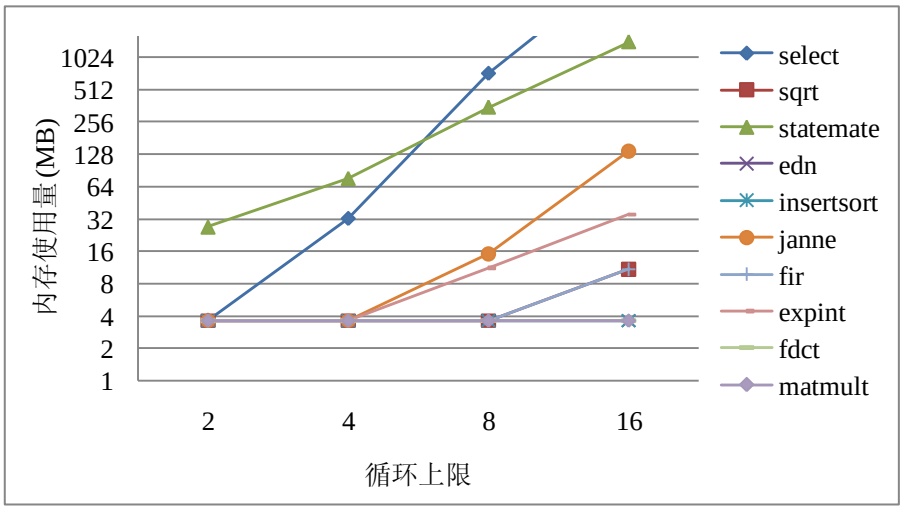


图 3.11 SPIN 内存使用量

Fig. 3.11 The memory usage of SPIN models

图 3.10 与图 3.11 分别表示了 10 个测试程序在循环上限分别为 2、4、8、16 时的分析时间和内存使用量。需要特别注意的是，这两个图表里的数据取自模型检测器验证 $[]\phi(WCET)$ 属性时的实验——这是因为模型检测器在验证 $[]\phi(WCET)$ 属性时需要检查整个状态空间。图 3.12 至图 3.15 的数据选取是类似的，后面不再进行重复说明。SPIN 模型检测器在分析时间上的结果比较好，对于大多数测试程序，都能在 1 秒以内完成分析。分析时间出现爆炸情况的三个程序是 select、statemate 和 janne。通过对这三个程序源代码进行分析，我们发现这三个程序的共性是“循环内部具有大量分支结构”。Select 有三层循环嵌套，且每层循环体内都出现分支结构；statemate 是工具自动生成的代码，其程序主体就是一个循环，而循环体内部有 20 条左右的分支路径。我们知道当循环体内部可能的路径多于 1 条，那么程序路径的个数与循环上限呈指数关系。由于底数就是循环体内部的可能路径数，因此这个数值越大，程序路径个数随循环上限的变化就越快。相反的，fdct、insertsort 和 matmult 三个程序由于循环体内都是单路径程序，因此它们的分析时间没有随循环上限的变化呈现指数变化趋势。

SPIN 的内存使用量是最大的，这跟 SPIN 的设计原理是有直接关系的。前文已述，SPIN 是一个“显式状态”的模型检测器，它没有对状态空间进行任何的符号化描述，因此问题的状态空间有多大，对应要使用的内存就有多大。这和我们观察到的现象也是对应的。除了 fdct、insertsort、matmult 等程序，其他程序都出现了程度不同的空间爆炸的情况。这一现象除了从内存使用量上来观察，也可以从 SPIN 模型检测器所报告的状态空间总数得以体现。在所有的测试程序中，只有 select 在循环上限为 16 的情况下，在使用完系统 8GB 内存的情况下未能分析出结果。

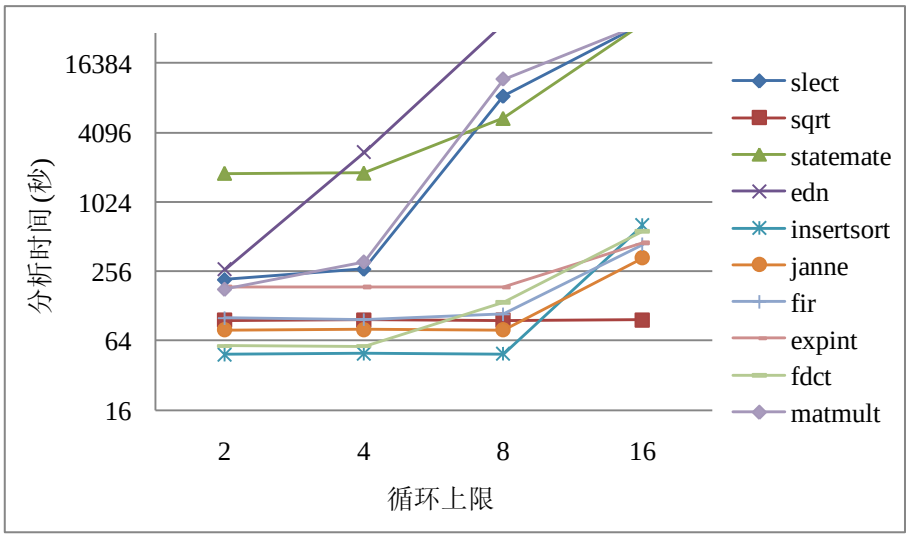


图 3.12 NuSMV 分析时间

Fig. 3.12 The execution time of NuSMV models

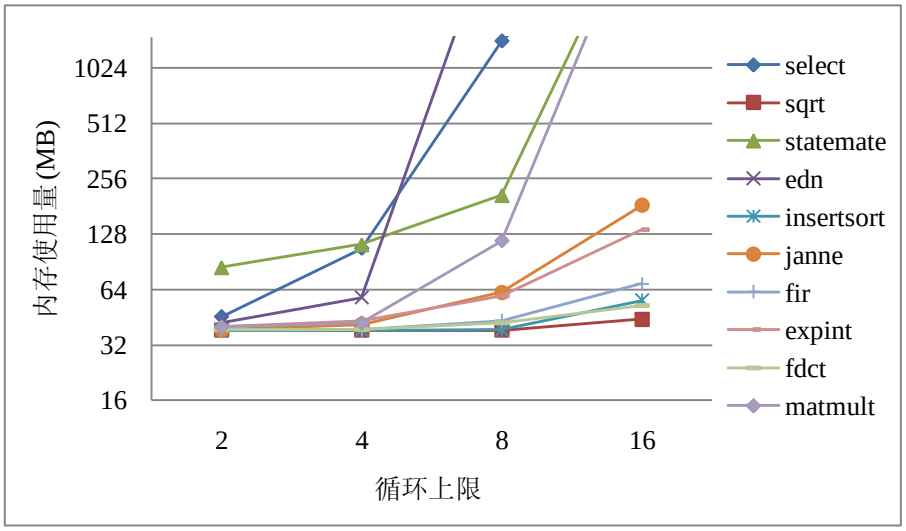


图 3.13 NuSMV 内存使用量

Fig. 3.13 The memory usage of NuSMV models

图 3.12 和图 3.13 反映的是 NuSMV 模型检测器在分析测试程序时的分析时间情况和内存使用情况。可以看出，NuSMV 的分析时间较 SPIN 有显著的增加，且爆炸情况也在更多的测试程序中出现。除了部分测试程序本身结构复杂以外，另一个主要原因就是 NuSMV 采用二进制的形式来表示模型中的任何变量，那么变量的值越大，二进制数的位数就越多，状态空间就越大。这一问题也出现在相关工作中[88-90]并导致了执行时间的显著增加。在本章的实验中，select、edn、matmult 程序在循环上限为 16 的时候，WCET 值已经普遍超过了 100,000，造成这些程序的分析不能在规定的时间内完成。而由于采用了二叉决策树（BDD）这一数据结构来符号化的表示状态空间，NuSMV 的内存使用量要普遍小于 SPIN 的内存使用量。在所有程序的分析过程中，没有任何一个程序的分析因为内存使用量超过上限而停止，且包括超时实验在内的所有实验，其内存使用量都没有超过 2GB。

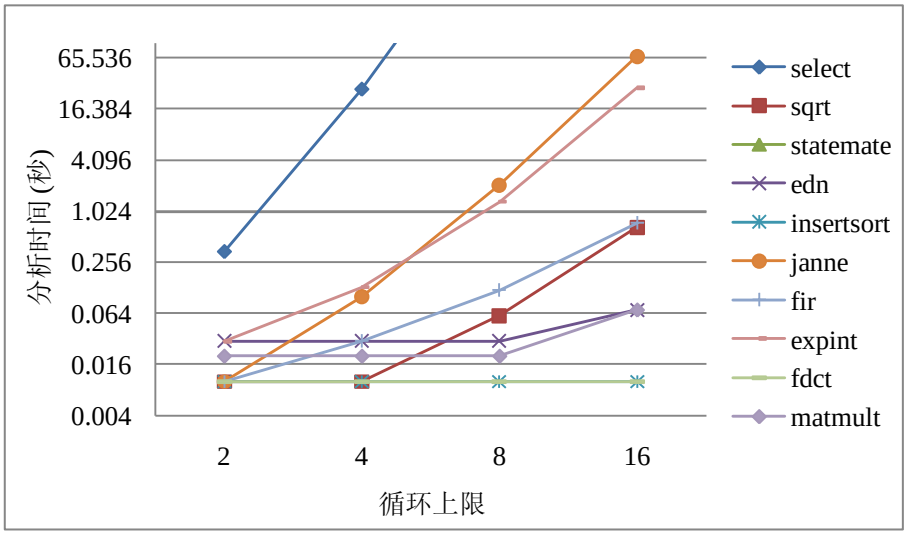


图 3.14 UPPAAL 分析时间

Fig. 3.14 The execution time of UPPAAL models

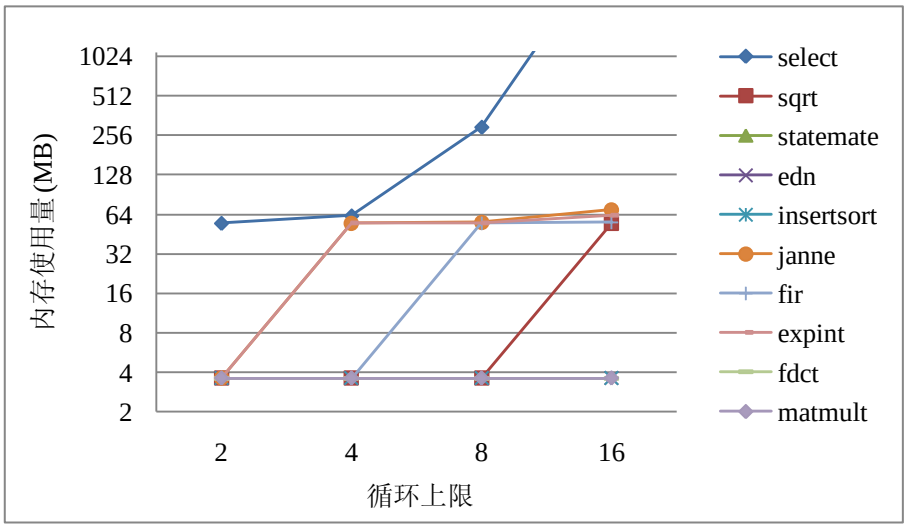


图 3.15 UPPAAL 内存使用量

Fig. 3.15 The memory usage of UPPAAL models

图 3.14 和图 3.15 反映的是 UPPAAL 模型检测器在分析测试程序中所需要的时间和内存。UPPAAL 未能成功分析出 statemate 程序，同时在 select 程序循环上限为 16 的时候也未能在规定时间内得到分析结果。除此之外，UPPAAL 在分析时间上的表现也可以接受，大多数程序都在 1 分钟以内即可得到分析结果。随着循环上限的提升，大多数程序的分析时间也开始呈现爆炸趋势。UPPAAL 的内存使用量普遍不大，除个别实验使用到接近 300MB 内存外，大多数程序的分析都使用了不超过 70MB 的内存。

综上所述，影响模型检测器分析性能（包括时间和内存）的主要因素来自复杂的循环结构。如果一个程序含有嵌套深度较大的循环，且循环体内部具有大量的分支，那么模型检测器将可能产生状态空间爆炸，导致分析时间和内存使用的指数增加。但是在对相对简单的程序进行分析的时候，模型检测器无论是分析时间，还是内存使用，都在合理的范围内。本章分别使用了三种不同的模型检测器，它们各基于不同的理论工作。实验数据表明：SPIN 在分析时间上的性能表现最好，但是“显式状态表示”的设计导致它更容易出现内存使用爆炸的问题；UPPAAL 在分析时间和内存使用上取得了最好的折中，对于大多数程序，UPPAAL 的时间爆炸和空间爆炸的程度都相对较轻；NuSMV 在时间和空间性能上都无法与 SPIN 和 UPPAAL 相比，因此最不适用于 WCET 分析。下一小节，我们将脱离具体工具，对于基于模型检测技术的 WCET 计算方法进行评价。

3.5.3 对基于模型检测技术的 WCET 计算方法的评价

虽然同隐式路径枚举技术相比，基于模型检测技术的 WCET 计算方法在分析性能和内存使用上都更差，但是模型检测技术最大的优势就是具有很强的描述能力，以及由此导致的分析精确性。模型检测器通常以有限状态自动机为建模方式，而这种建模方式

能够很自如的描述绝大多数的程序语义以及处理器行为，从而使得分析模型更加接近实际系统，得到的分析结果也就更加精确。

```
int i, j, k, m;

for (i = 0; i < 10; i++)
{
   k++;
   for (j = 0; j < i; j++)
         m++;
}
```

(a)

```
int i, j, k, m;
if (i <= 0){          // branch 1
   k++; m = 0; } // left
else{
   k--; m = 1; } // right
if (m == 1)          // branch 2
   k++;      // left
else
   k--;       // right
```

(b)

图 3.16 蕴含关系程序语义示例

Fig. 3.16 The demo of program semantics using implication relations

一类典型的例子就是采用蕴含关系描述的程序语义。例如，图 3.16(a)所示的程序有两层循环，其中内层循环次数依赖于外层循环的当前值。这种关系实际上是蕴含关系：“如果外层循环控制变量 *i* 为 *x*，那么内层循环本次执行的循环上限为 *x*”。这种蕴含关系的程序语义本身是难以描述为线性约束的。如果使用模型检测器，只要增加相应的控制变量，并在必要的程序位置修改控制变量的值即可很容易的实现对蕴含约束关系的建模。在该例子中，模型检测器能够精确的让内层循环执行 45 次。而由于采用整数线性规划难以描述这种关系，往往是假定内外层循环的最大次数都是 9，因此得到的最大内层循环次数为 90 次。类似的典型例子还有如图 3.16(b)所示的控制流程，如果分支 1 执行左边路径，那么分支 2 一定执行右边路径。这种逻辑关系也很难用线性约束表示。以上仅是两个最简单的例子。在实际程序中，程序的不同部分之间可能存在更加复杂的逻辑关系，这种情况下，模型检测技术的建模能力和分析精确性就得到了充分的体现。

```
int main()
{
   int i, j, b, k;
   for (i = 0; i < 3; i++){
        for (j = 0; j < 3; j++){
             if (b)  k++;
             else    return 0;
        }
   }
   return 1;
}
```

图 3.17 程序示例

Fig. 3.17 An exemplary program

此外，隐式路径枚举技术在描述程序的其他功能性约束方面也存在问题，甚至可能导致分析结果错误。我们以图 3.17 的例子来给予说明。图 3.17 的基本结构是两层循环，

当在最内层循环发现某个条件满足的时候，程序将直接跳出循环并返回。这一程序反编译后恢复出来的 CFG 如图 3.18 所示。根据程序语义可知，内外层循环的上限都是 3，也就是说 BB3、BB5、BB10 执行 3 次，而 BB4、BB6、BB7、BB9 将执行 9 次。这个程序的流程结构的一个特点是：它不是结构化的程序，循环的退出路径个数大于 1，具体体现在 t_{8_12} 这条边上。如果采用隐式路径枚举技术对这个 CFG 进行描述，那么两层循环将导致两个循环约束条件，分别是“$b10-3\cdot b0\le 0$”和“$b9-3\cdot b3\le 0$”。根据这一约束方法，采用隐式路径枚举技术计算出来的 BB3 的执行次数为 4 次，比实际程序语义的循环上限多 1 次。产生这一问题的根本原因是：“实际上应该约束所有属于循环体的基本块的执行次数小于循环入口的执行次数，但是隐式路径枚举技术中用循环尾节点执行次数代替整个循环体的执行次数，如果循环不符合结构化程序的要求，那么这种约束方法将导致其他循环体节点执行次数分析错误”。如果采用正确的方法，需要找到属于循环体的所有基本块，对其执行次数都进行约束。但是，如果循环以复杂方式嵌套，那么为每个基本块都找到执行次数约束将非常困难。而如果采用模型检测技术，以本程序的外层循环为例，循环的控制条件是添加在从 BB1 到 BB3 的状态迁移上的，这就保证了程序对应的 BBA 能够准确无误的按照程序语义执行。对比上述两种技术可以看出，在面对非结构化的程序时，传统的隐式路径枚举技术的分析结果不如模型检测技术的分析结果准确。

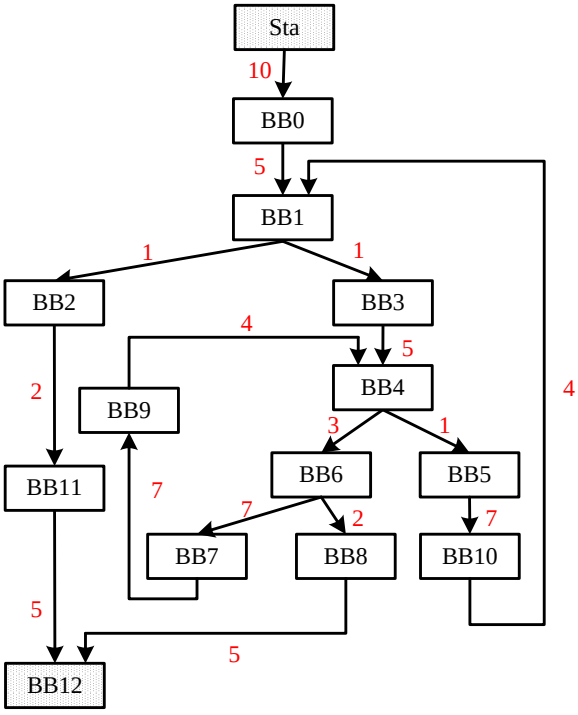


图 3.18　示例程序对应的 CFG

Fig. 3.18 The CFG of the exemplary program

除了在程序语义方面，模型检测技术描述硬件行为的能力也要远远强于其他技术手段。当出现新的硬件体系结构时，采用模型检测技术可以在很短的时间内将新体系结构的行为加以描述和建模；而传统的基于隐式路径枚举计算框架的处理器行为分析技术，则不得不修改或提出新的理论，以使其能够分析新的体系结构特性。在后面两章，我们将采用模型检测技术对单核独立 Cache 和多核共享 Cache 建模并进行 WCET 分析，目的就是利用模型检测技术强大的描述能力提高分析精确性。

从前面的讨论以及相关工作可以看出，不同的 WCET 计算技术的差别的本质就是分析复杂度与分析精确性的不同程度的折中。基于语法树的技术具有最高的分析性能，但是它所能够提供的程序语义和处理器行为的描述能力则是最弱的，因此难以提供较高的分析精确性。隐式路径枚举技术较基于语法树的技术分析复杂度更高一些，但是描述能力更强，分析精度更好。而基于路径的技术，例如本文采用的基于模型检测技术的 WCET 计算方法，具有最高的分析精确性，但所付出的代价就是分析性能。因此，很难有一种技术能够在分析性能和分析精确性两方面同时达到最好。具体采用哪种技术，应该根据应用需求、程序特点、以及体系结构特性等进行判断。基于本章的实验和分析，我们粗略的给出适合使用模型检测技术进行 WCET 分析的一些范围：

（1）结构较为简单的程序适合采用模型检测技术分析；结构复杂，尤其是循环嵌套层数较深的程序不适合使用模型检测技术分析。如果采用该技术，应结合层次化分析或剪枝等手段以降低状态空间；

（2）系统的关键程序段，或程序的关键片段可以采用模型检测技术分析 WCET，以获取更高的分析精度；

（3）对分析精度要求很高的系统适合采用模型检测技术；

（4）分析全新的或特殊的硬件体系结构，适合采用模型检测技术。

3.6　小结

本章主要研究了基于模型检测技术的 WCET 计算方法。实验采用 SPIN、NuSMV 和 UPPAAL 三种主流模型检测器对 10 个测试程序进行了分析，通过实验数据验证了模型检测技术用于 WCET 计算的可行性，分析了该技术在可伸缩性方面的主要问题，并给出了适用范围。

第4章　基于剪枝思想的单核指令 Cache 分析

在上一章的研究中我们提出了一种基于模型检测技术的 WCET 计算框架，并阐述了该技术在描述能力方面的优势，以及由此带来的分析精度的提高。模型检测技术强大的行为描述能力不仅仅体现在提高分析精度这一个方面，这种能力使得该项技术对于新体系结构具有非常强的适应性。以处理器行为分析中很重要的 Cache 分析为例，目前在研究领域占有统治地位的是基于抽象解释的分析方法，主要用于对 LRU 替换策略进行分析。而实际的嵌入式处理器中大多采用 FIFO 和 PLRU 等替换策略[43]，这些替换策略的工作原理与 LRU 差别很大，因此无法使用面向 LRU 的分析方法。为此，研究或工程人员必须针对新的替换策略设计全新的基于抽象解释的分析方法，而在研究和实践中，这一过程被证明为是难度巨大的。通常当一种全新的体系结构出现以后，需要很长的时间建立面向新体系结构的分析技术，这在某些关键的实际应用中是不能满足要求的。在这种情况下，借助模型检测技术强大的行为描述能力，系统分析者可以在很短的时间内迅速完成对新体系结构的建模，使得系统的时间验证工作可以在最短的时间内得以开展。本章的主要研究内容，就是在第 3 章提出的基于模型检测技术的 WCET 计算框架的基础上，继续使用该技术对单核系统的 Cache 行为进行分析。

在上一章的实验中我们能够发现，模型检测技术的主要问题在于：当循环内部存在分支的情况下，随着循环嵌套深度和循环上限的加大，程序的路径数量将会产生爆炸，进而导致程序模型的状态空间爆炸。如果在此模型基础之上进一步对其他处理器行为（例如 Cache 行为）进行建模，将使得状态空间爆炸更加严重，这就有可能限制模型检测技术在 WCET 分析中的应用。模型检测器自身通常设计有状态空间削减技术，但是由于模型检测器是一种通用的分析工具，针对不同的应用，其自身的状态空间削减技术有时并不能提供良好的削减效果。为此，必须针对应用本身设计新的分析方法与模型检测技术配合，来解决状态空间爆炸的问题。本章拟继续采用模型检测技术对单核系统中基于 FIFO 替换策略的 Cache 行为进行分析，并提出一种基于剪枝思想的分析方法，使之与模型检测技术配合，在保证了高分析精度的前提下，能够有效提高分析的可伸缩性。该方法对于扩大模型检测技术在 WCET 分析中的应用将起到重要作用。

4.1　相关工作

LRU 被证明为是各种 Cache 替换算法中可预测性最好的一种[101]，目前几乎所有的考虑 Cache 的 WCET 分析的研究中都采用了这一替换算法。但是由于 LRU 替换算法的

执行逻辑比较复杂，通常一个 LRU 的 Cache 管理器需要使用大量的硬件逻辑进行实现，因此在实际的处理器中，往往采用 FIFO、PLRU 等替换算法取代 LRU，例如在著名的摩托罗拉 ColdFire 系列处理器以及 ARM 处理器中广为采用的就是 FIFO 替换算法。

图 4.1 FIFO Cache 组示例

Fig. 4.1 An example of a FIFO cache set

FIFO 替换算法是本章所要研究的主要对象。图 4.1 表示的是一个 4 路组相联的 Cache 组。顾名思义，FIFO 策略对 Cache 的更新采用先进先出的顺序。如果某个访存在 Cache 中不命中，那么将按照箭头的方向，从 Cache 组的最左边将新的访存插入，同时将最右边的 Cache 块替换出本组；如果访存在 Cache 中是命中，那么对应的 Cache 块在 Cache 组中的位置不发生任何改变，也就是说这个 Cache 组的状态不发生任何改变，这是 FIFO 与 LRU 最大的不同。

FIFO 替换策略的可预测性显然没有 LRU 替换策略好。例如一个访存序列，第一个访存元素可能在 Cache 中是命中的，假定其命中在某个 Cache 组中最后面的位置。如果下一个访存元素在 Cache 中是不命中，那么它将把刚才命中的那个访存元素替换出本组。而在 LRU 策略中，最近访问的元素总是排列在 Cache 组的最前面。换句话说，访存元素在 Cache 组中的位置与其年龄是完全对应的。相比之下，FIFO 策略仅当 Cache 访问为不命中的时候，其行为才与 LRU 相似；而在部分 Cache 访问为命中的时候，预测该元素在 Cache 组中可能存在的时间将变得非常困难。FIFO 替换策略的这种特殊性，导致为其建立基于抽象解释的分析方法非常困难，而学术界关于基于 FIFO 替换策略的 WCET 分析方面的研究也很不成熟。正是基于这一背景，本章利用模型检测技术的优势，提出基于 FIFO 替换算法的单核系统 Cache 行为的分析方法；并通过剪枝技术来提高模型检测技术在分析中的可扩展性。

4.2　剪枝的基本思想和关键问题

第 3 章中，采用模型检测技术进行 WCET 计算的研究表明，影响模型检测分析效率的决定性因素之一就是程序中的循环结构。当一个循环体中存在多个程序分支，且循环次数较高的情况下，程序可能的路径数量就会爆炸，进而导致程序模型状态空间爆炸。例如，一个循环内部有两条可能的路径，如果循环上限为 1000 次，那么程序可能的路径就有 2^{1000} 个，状态空间将非常巨大。如果同时对处理器行为（如 Cache 行为等）进行建模，那么状态空间爆炸的问题将进一步恶化。因此需要采用有效的手段削减状态空间。

模型检测技术所面临的主要问题是，模型检测器并不知晓用户模型的语义，也就是并不知道分析目的是验证程序的最大执行时间，因此它将考虑所有可能的路径，以确切验证给定的属性是否安全。实际上，如果可以证明在任何状态下，程序的两个分支中某一个的执行时间一定小于另外一个，那么显然执行时间较小的那个分支完全没有必要考虑到 WCET 计算中。因此一种可能的办法就是将这样的分支从循环中剪掉，程序的路径数量就会大大减少，这样一来模型检测的状态空间爆炸问题将得到有效解决。剪枝的根本目的就是在不丧失分析精度的前提下，降低基于模型检测技术的分析的复杂度。这就是我们拟采用剪枝技术配合模型检测技术进行 WCET 分析的根本动机。

在实际分析中，正确的对程序进行剪枝并不是一项简单的工作。首先应该明确一点，就是本章所提到的这种剪枝，是剪掉那些在 WCET 计算中不需要考虑的路径，但是这并不意味着这些程序路径在程序的实际运行过程中不被得到执行。在这样一个前提下，对待分析的程序进行正确有效的剪枝，就必须解决两个问题：一是正确的识别可能被剪的分支，以及在什么条件下如何剪枝；二是要在分析被剪分支的过程中保证分析结果的安全性。下面我们就这两个问题给予进一步的阐述。

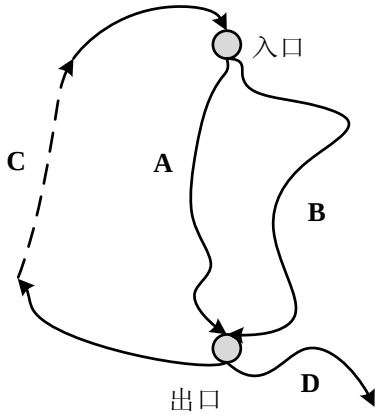


图 4.2 剪枝安全性问题示意图

Fig. 4.2 A demonstrative situation of safety problems in branch cutting

首先探讨剪枝安全性的问题。如图 4.2 所示，假定程序有两条分支 A 和 B，如果不考虑 A 和 B 的执行上下文，那么分析出来的 A 的执行时间是 t_1，B 的执行时间是 t_2，且 $t_2 > t_1$。直观的感觉是应该剪掉分支 A。但是不管哪个分支的执行，都可能导致原来 Cache 中的一些数据被替换出去。如果被替换出去的内容在未来的某个时刻还会被再次访问到，那么就会造成这部分执行时间的增加。例如在图 4.2 中，分支 A 的执行将会导致 C 段有 n_1 条指令从 Cache 中被替换，而 B 的执行只会替换出 C 段的 n_2 条指令。如果满足(n_1 - n_2)条指令 Cache 不命中的额外延时比(t_2 – t_1)还要大，并且 C 段和分支 A 与 B 同属于循环体，那么从整个循环体的角度来讲，程序走 A 路径将会导致更长的执行时间。这个例子

说明，对程序分支进行安全的剪枝，不能仅仅考虑分支本身，同时也需要考虑不同分支的执行对程序其他部分的影响。Cache 的组织形式、替换策略很多，不管采用何种 Cache，为安全的计算各分支的执行时间，就必须安全的计算分支入口和出口的处理器状态，以及这种状态对程序其他部分的影响。同时需要特别注意的是，当程序中存在嵌套循环的时候，安全的考察各分支对程序其他部分的影响将会变得更加困难。

在剪枝的过程中仅仅考虑安全性还是不够的。例如一个循环内部有两条分支 A 和 B，我们必须分析分支 A 和 B 的执行时间的上限及下限，如果一条路径的上限比另外一条路径的下限还要小，我们就可以安全的剪掉前者。因此能够精确的计算程序分支执行时间的上限和下限，是确保剪枝是否有效的关键因素。最简单的计算上限和下限的方法就是假定所有的 Cache 访问全不命中或全命中。举例说明，假设分支 A 有 10 条指令，分支 B 有 20 条指令；Cache 只有一级，命中需要 5 个时钟周期的访问时间，不命中访存则需要耗费 20 个时钟周期；同时假设每条指令的执行时间为 3 个时钟周期。可以计算，A 分支执行时间的区间为[80, 230]，B 分支执行时间的区间为[160, 460]。在这种情况下，由于没有任何一个分支的执行时间上限小于另外一个分支的执行时间下限，因此无法进行剪枝。而实际程序中，可能 A 和 B 的执行都不会是 Cache 全命中或 Cache 都不命中，如果能分析得到这类信息，就可以提高计算分支执行时间区间的精度，从而扩大剪枝的可能。这就要求必须精确的计算各分支中指令的命中情况。

但是即使能够比较精确计算各分支中指令的命中情况，也不能保证剪枝一定能够成功。例如两个分支 A 和 B，A 的执行会从 Cache 中替换出 B 的指令，而 B 的执行同样会从 Cache 中替换出 A 的指令。在这种情况下，如果第一次执行了 A，那么下一次执行 B 的时间更长；反过来，执行完 B 后，下一次循环执行 A 路径时间更长——这说明可能存在程序在两条路径之间跳跃的情况，而这种情况下，确定的剪掉某个分支显然是不可能的。实际上，循环分支只有在循环上限特别大的情况下，才会导致路径个数很大。因此，即使不能完全剪掉某个分支，但如果能分析得到循环在执行完 N 个周期后就会稳定到某个分支上，那么我们就可以把循环体转换为两个先后执行的部分，第一个部分是一个循环次数为 N，且循环体带有分支的循环；假定原循环上限为 LPB，第二个部分就是一个循环次数为(LPB - N)，但循环体为单路径（N 个周期后的稳定路径）的循环。显然第二个部分的循环将不会造成状态空间爆炸的问题，而第一个部分的循环由于上限从 LPB 减少到 N，结果是状态空间得到了削减。

以上就是本章所研究的剪枝技术的基本思想，目的是通过剪枝达到削减状态空间，提高基于模型检测技术的 WCET 分析的性能与可伸缩性。后续小节我们将分别针对本节提出的剪枝中的关键问题进行具体的阐述。需要特别指出的是：对于 LRU 替换算法

而言，基于抽象解释的 Cache 分析的结果就可以作为剪枝的依据。但是考虑到基于抽象解释的技术已经能够在分析单核系统 LRU 策略时提供足够精确的分析结果，因此采用模型检测技术对 LRU 策略进行分析以提高分析精度的意义并不大。本章主要基于 FIFO 替换策略展开讨论。

4.3　基本假设与定义

对程序进行剪枝，最重要的问题就是保证分析的安全性。这一问题已经在上一小节结合图 4.2 进行了粗略的说明。本节首先给出本章讨论所基于的假设，并对将使用到的概念进行定义；其次给出一类特殊但是常见的分支结构，提出针对这种特殊分支结构的安全的分支削减方法；之后探讨了面向普通分支结构的分支削减方法。

本章讨论问题的前提假设如下：

假设 1：程序启动时的 Cache 内容为空；

假设 2：所有的 Cache 块的大小都为 8B，即一条 PISA 指令的长度；

假设 3：假定流水线是完美的，即每条指令的执行时间固定为 1 个时钟周期；

假设 4：假定所有循环上限已知，且上限值很大。

对于假设 1，如果程序启动时的 Cache 状态不为空，那么程序的一些指令第一次执行就可能在 Cache 中出现命中的情况，这种情况非常难以分析。假设程序启动时 Cache 状态为空可以简化这一问题。目前大多数的处理器都提供了刷新 Cache 的功能，可以在程序启动之前先刷新 Cache，因此这一假设在实际系统中也是合理的。对于假设 2，如果一个 Cache 块的大小可以包含多条指令，那么可能导致某些基本块最开始的指令在第一次执行的时候就是命中，我们假设 Cache 块大小等于指令长度以简化这一情形。假设 3 中假设流水线是完美的主要是将研究的重心放在 Cache 的行为分析上。由于在循环上限很大的情况下分支削减才有意义，因此我们在假设 4 中假定所讨论的循环的上限值都足够大。

接下来对于后面小节将使用到的概念进行定义：

LOOP：待分析程序中所有循环的集合；

$loop_i$：第 i 个循环，任何一个循环都是由若干个基本块组成；

A, B, C：用大写字母表示程序的基本块；

<ABC>：表示基本块 A、B、C 顺序依次执行对应的路径；

a, b, c：用小写字母表示访存序列中的一个元素，其实质是一条指令；

RD_i：第 i 条指令的复用距离，定义为访存序列中，第 i 条指令两次连续出现的最小间隔；

T_{hit}：Cache 命中时的访问时间；

T_{miss}：Cache 不命中时的访问时间；

Asso：Cache 的相联度；

$T_{H.i}$：第 i 条路径指令全命中时对应的执行时间；

$T_{M.i}$：第 i 条路径指令全不命中时对应的执行时间。

4.4　特殊分支结构的分析

```
...
If (cond){
        S1;     // some instructions
        S2;     // some instructions
        S3;     // some instructions
}
...
```

图 4.3 一种分支结构的示例

Fig. 4.3 A demo of a certain kind of branches

通过对测试程序源代码的分析，我们发现很多程序中有图 4.3 所示的分支结构。其语义所表达的是当条件 cond 满足的时候，就执行一段程序（S1;S2;S3;）；而如果条件 cond 不满足则不执行任何指令。图 4.4 表示的是图 4.3 程序对应的 CFG。其中 A 节点判断条件是否满足，B 节点为条件满足时执行的基本块。如果条件满足，执行完 B 后，会继续执行分支结构后续基本块 C；如果条件不满足，在 A 执行完之后程序将直接跳转到 C 执行。下面我们将证明，对于程序中所有的特殊循环，类似 AC 这种边都可以被安全的删除。这里我们称 AC 边为“旁路边（bypass edge）”。

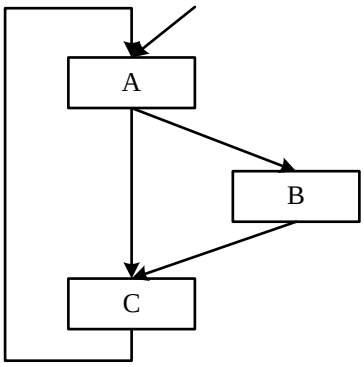


图 4.4 特殊分支结构对应的 CFG

Fig. 4.4 The CFG of a special branch structure

以图 4.4 为例，假定基本块 A、B、C 映射到第 i 个 Cache 组中的访存序列的长度分别为 L_{Ai}，L_{Bi}，L_{Ci}。如果程序不执行 B 分支，那么依次执行<AC>访存序列；否则，依次执行<ABC>访存序列。如果旁路边可以删除，就意味着无论循环多少次，程序一定是执行 B 分支才能达到最大执行时间。下面我们首先证明如果程序只有单级循环，旁路边

可以安全剪除。进而扩展到在多级循环中，旁路边都可以安全剪除。

定理 1：在只有单级循环的程序中，图 4.4 中的旁路边 AC 可以安全剪除。

证明：证明的主要思路是分析在多次循环的过程中，执行基本块 B 对整个程序执行时间的影响。显然，我们可以仅针对一个 Cache 组进行讨论，其他 Cache 组的结果是类似的。根据映射到第 i 个 Cache 组中指令个数的不同，分三种情况进行讨论。

（1）$L_{Ai} + L_{Ci} > Asso$

在这种情况下，不管程序执行<AC>还是<ABC>，所有指令的复用距离都一定大于 Asso。根据 FIFO 的工作原理，循环体中的每条指令在任何一次循环的执行中都不命中。

（2）$L_{Ai} + L_{Bi} + L_{Ci} \leq Asso$

在这种情况下，无论循环执行哪条分支，所有指令除第一轮的首次访问失效之外，在其他各次循环中的执行都为 Cache 命中。因此，循环执行指令数量更多的<ABC>路径导致最长的执行时间。

（3）$L_{Ai} + L_{Ci} \leq Asso$，且 $L_{Ai} + L_{Ci} + L_{Bi} > Asso$

这种情况的含义是，如果程序执行<AC>路径，那么对于以后的各次循环，所有指令都为命中；如果程序执行<ABC>路径，由于 $L_{Ai} + L_{Ci} + L_{Bi} > Asso$，可知这条路径上所有的指令的复用距离都大于 Asso，所以在各次循环的执行中所有指令皆为不命中。又由于 $L_{ABCi} \geq L_{ACi}$，因此循环的任何一次执行中，都是执行<ABC>路径时间最长。

综上所述，在只有单级循环的程序中，旁路边 AC 是可以安全剪除的。

证毕。

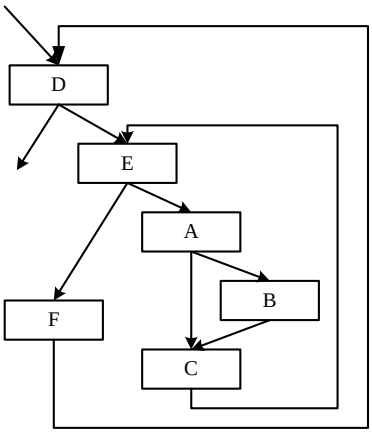


图 4.5 两层嵌套循环举例

Fig. 4.5 An example of 2-level nested loops

对于如图 4.5 中的两层嵌套循环的情况，关键是要证明剪掉内层循环的旁路边不影响 WCET 计算的安全性。两级循环与单层循环最大的不同在于：单级循环中，循环的初始状态只可能是空状态。而在两级循环或多级循环中，内层循环可能被多次进入，且

两次进入内层循环之间可能执行外层循环的指令，导致内层循环的某些指令被替换出 Cache，因而内层循环的初始状态可能不止一种。故而在两级循环嵌套的情况下，需要考虑内层循环所有可能的初始状态，以及这种初始状态对内层循环体指令命中情况的影响。证明的目标就是在考虑了这种影响的情况下，确定剪掉内层循环的旁路边依然是安全的。

定理 2：在带有两级嵌套循环的程序中，如果内层循环为图 4.5 所示结构，那么旁路边 AC 可以安全剪除。

证明：首次进入内层循环的时候，初始状态为空。图 4.5 表示的是一种通常的情况。基本块的执行路径是<D(EABC)*EF>，其中*表示括号内的部分循环执行多次。因此，当第一次循环执行完毕后，之后各次循环的初始状态，都是上一次循环执行完毕的结束状态加上<EFD>执行的结果。需要特别注意的是，循环体内部的 E 的执行应该看作是循环体的一部分。

首先，如果内层循环体满足“$L_{Ei} + L_{Ai} + L_{Ci} > Asso$”或“$L_{Ei} + L_{Ai} + L_{Ci} + L_{Bi} > Asso$”，依照前面的分析，可知每次进入内层循环的时候所有指令都为不命中，这就相当于每次进入内层循环时 Cache 状态为空。因此我们仅需讨论内层循环体满足“$L_{Ei} + L_{Ai} + L_{Bi} + L_{Ci} \le Asso$”的情况。

程序从内层循环退出后，需要执行路径<EFD>才能够进入下一次循环。由于已知条件“$L_{Ei} + L_{Ai} + L_{Bi} + L_{Ci} \le Asso$”，因此路径<EFD>中 E 的执行一定是命中。根据 FIFO 的工作原理，不影响 Cache 状态。所以，可能影响内层循环初始状态的基本块只有 F 和 D。这里我们把<FD>简称为 G。下面将具体探讨 G 的指令执行对内层循环的影响。

显然，如果 $L_{Gi} \ge Asso$，那么 G 的执行将会把所有内层循环在第 i 组中的内容清空，内层循环每次进入时的初始状态都为空。因此，我们只需要讨论 $L_{Gi} < Asso$ 的情况。已知 $0 \le L_{Gi} < Asso$，令 $x = Asso - (L_{Ei} + L_{Ai} + L_{Bi} + L_{Ci})$，对“$0 \le L_{Gi} \le x$”和“$x < L_{Gi} < Asso$”两种情况分别讨论。

（1）若 $0 \le L_{Gi} \le x$，可知 $L_{Ei} + L_{Ai} + L_{Bi} + L_{Ci} + L_{Gi} \le Asso$，所以内层循环的内容都会保留在第 i 个 Cache 组中，因此第二次循环之后（包括第二次）每次循环的执行指令在 Cache 中都为命中，执行<ABC>路径时间更长；

（2）若 $x < L_{Gi} < Asso$，可知 $L_{Ei} + L_{Ai} + L_{Bi} + L_{Ci} + L_{Gi} > Asso$，当再次进入内层循环体的时候，循环体所有指令的执行都是不命中，因此依旧是执行<ABC>路径时间更长。

如果外层循环存在分支，也就是说，在第二次进入内层循环的时候，外层循环可能先执行了一次其他分支，那么实际上等价于增加了 G 的长度，这并不影响到上面的证明过程的正确性。

综上所述，在两级嵌套循环程序中，内层循环的旁路边可以安全剪除。

证毕。

推论 1：在带有多级循环的程序中，任何一层循环的旁路边都可以安全剪除。

推论 1 可以通过迭代使用定理 2 推出，这里略去证明细节。

4.5　普通分支结构的分析

相比于上述特殊分支结构，普通的分支结构分析难度很大。我们将在本节分别对含有普通分支的单层循环和多层循环分别进行讨论。

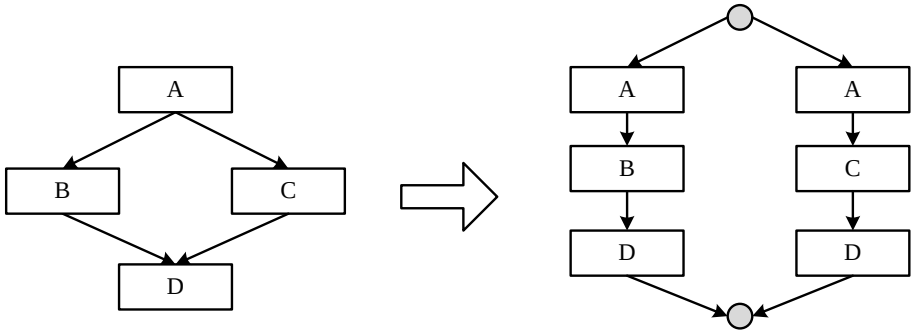


图 4.6 平铺操作示例

Fig. 4.6 An example of the flatten operation

在讨论开始，首先介绍对循环体的“平铺（flatten）”操作。在带有普通分支结构的程序中，分支 A 和 B 的执行都可能对后续的访存序列 C 造成影响，不加约束的剪掉某个分支将会引发潜在的不安全性。为解决这一问题，我们对循环体内部进行“平铺”操作，如图 4.6 所示。平铺的作用就是将那些同时被两个或多个分支执行的基本块分别复制到每个分支中，使得平铺后的循环体中没有任何一个基本块被多个分支共享。那么在分析的时候，在一个循环体内部，就不存在上述不安全性问题。如果一个循环体包含两个依次执行的分支，那么平铺后循环体将有 4 条路径。在本节后面的讨论中，都假定已经对循环体进行了“平铺”处理。

4.5.1 单层循环的剪枝

对于 4.3.1 小节中介绍的特殊分支结构，我们可以证明将旁路边剪掉是安全的，但是对于普通的分支结构，情况要复杂得多。我们在 4.2 节中已经指出，如果程序的两个分支在执行时间上的差异不悬殊，执行某个分支可能导致另外一个分支的内容被替换出 Cache，因而可能出现循环在两个分支间交替执行才能达到最长执行时间的情况。所以并非对所有的普通分支都能进行剪枝操作，因此需要确定一组条件，在满足条件的情况下才能安全的对分支进行剪枝。

4.5.1.1　剪枝判断条件

假定两个分支分别为 B 和 C，最简单的办法就是假定分支中所有的指令全为命中计算分支执行时间的下限，同时假定分支中所有的指令全为不命中计算执行时间的上限，如果 B 分支的上限比 C 分支的下限还要小，那么就可以安全的剪掉 B 分支；反之亦然。我们称这种方法为“基线方法”。基线方法显然是安全的，但是很不精确，除非两个分支的指令个数、执行时间相差非常悬殊，否则很难通过这个判断。为此，必须更加精确的判断分支结构在执行过程中的命中/不命中的情况，并把这一信息考虑到每个分支上下限的计算中，从而得到更加精确的执行时间区间。下面我们将提出一种更精确的面向单层循环的执行时间区间分析方法，该方法首先分析各分支中每条指令的命中类型，之后根据命中类型计算分支执行时间的上限和下限。

假定程序的两个分支为 A 和 B，循环的每次执行可能选择两者中的任何一个。如果 A 和 B 都有指令映射到同一个 Cache 组，那么我们称在这个 Cache 组上 A 和 B 有交集；反之如果某个 Cache 组中仅有 A 或 B 中的一个分支所包含的指令，那么称在这个 Cache 组上 A 和 B 无交集。我们需要分别对这两种情况进行讨论。不妨假设 A 分支独立使用 Cache 组 CS1，B 分支独立使用 CS2，A 和 B 分支共同使用 CS3。

（1）对没有交集的 Cache 组的分析

A 分支独立使用 CS1 表明 CS1 的行为将不对 B 分支的执行时间造成影响。假定 A 分支落在 CS1 中的指令个数为 n_1，如果 n_1 > Asso，那么所有这 n_1 条指令在执行过程中都是不命中，所以将这些都标记为一定不命中（AM）；如果 $n_1 \le$ Asso，说明 CS1 能够容纳 A 分支映射到其中的所有指令，那么这些指令除了在循环第一次执行时为首次执行失效，其他各次执行都是命中，因此所有 n_1 条指令都被标记为首次失效（FM）。

（2）对有交集的 Cache 组的分析

根据上面的假设，A 分支和 B 分支同时使用 CS3。由于循环体是进行过平铺操作的，A 和 B 映射到 CS3 中的指令可能相同，也可能不同。假定 A 映射到 CS3 中的指令个数为 n_1，B 映射到 CS3 中的指令个数为 n_2，A 映射和 B 映射的并集中的指令个数为 n_3，可以分如下几种情况讨论。

- n_1 > Asso，且 n_2 > Asso。也就是说，无论执行 A 还是 B，映射到 CS3 中的指令的个数都大于 Cache 组的长度，因此所有的访问都是不命中。这种情况下，映射到 CS3 中的所有指令都设置为 AM 类型；
- $n_3 \le$ Asso。在这种情况下，不管是执行 A 还是执行 B，也不管执行次序如何，只要被执行一次，所有的指令都会被载入到 CS3 中，从而造成后面的循环中指

令都是命中。因此这种情况下，CS3 中的指令类型都被设置为 FM；

- 对于 n_1 > Asso 且 $n_2 \leq$ Asso，或者 n_2 > Asso 且 $n_1 \leq$ Asso 的情况，无法确定 CS3 中指令的命中情况，因此这些指令设置为不可确定（NC）类型；
- $n_1 \leq$ Asso，且 $n_2 \leq$ Asso，但是 n_3 > Asso 的情况与上面的类似，因此所有指令都设置为 NC。

在完成了上述指令类型分类之后，我们计算每条指令的执行时间区间。显然，所有 AM 类型指令的上限和下限都为 T_{miss}，所有 FM 或 NC 指令的下限都为 T_{hit}，上限都为 T_{miss}。累积一条路径上所有指令的执行时间区间，就可以得到路径的执行时间区间。我们称这一方法为“改进方法”。显然，“改进方法”相对于“基线方法”要精确一些，但是精确性主要来源于更加精确的 AM 分析。但是在实际程序中，如果循环体的长度不超过 Cache 组的个数，那么 AM 出现的情况就非常少，而大多数的指令将最终被归类为 FM，区间计算的结果就会退化到“基线方法”。

4.5.1.2　基于程序变形的剪枝

但是，并非所有的程序客观上都能完成剪枝，我们举一个简单的例子进行说明。假设 A 分支有 20 条指令，B 分支有 2 条指令，且 A 与 B 分支的指令类型都为 FM。指令命中的访问时间为 2，不命中的访问时间为 30。那么可以知道，程序在大部分的循环中都要选择 A 分支执行。但是假定一开始就执行 A 分支，考察中间的某一次循环执行。如果执行 B 分支，那么 B 分支的指令皆为不命中，执行时间为 60；如果继续执行 A 分支，那么 A 分支的指令为命中，执行时间为 40。显然，中途执行一次 B 分支比一直执行 A 分支执行时间更长。在这种情况下，导致最长执行时间的行为可能是在 A 和 B 两个分支间的某种交替，而不是单纯执行某一个分支。为处理这种情况，我们将试图发现程序执行的一些规律，通过对循环体进行某种变形，使得剪枝变为可能。

通过对部分测试程序的分析，我们发现程序中有相当数量的循环体并不是很大，即使包含多个分支，整个循环体也能够被完全放置到 Cache 中。在这种情况下，一旦整个循环体都被载入 Cache，那么循环体之后的各次执行中指令都将是命中。这就存在通过程序变形后进行剪枝的可能。具体分析过程如下。

首先需要计算循环体的大小，可以通过累积循环体中所有基本块占用的地址空间的方式获得。如果满足循环体的大小小于 Cache 容量，就可以断定映射到任何一个 Cache 组中的指令个数一定不会大于 Asso。循环体内部各指令的执行一定不会造成其他指令出现不命中的情况。对于每个分支，第一次载入 Cache 时由于指令皆不命中，因此执行时间较长；但是一旦载入 Cache，所有指令在 Cache 中都是命中的。在这种情况下，剪枝

的主要思路是：对于程序的各个分支，计算它们在第一次载入和其后各次执行分别对应的执行时间，并对此依从大到小的顺序进行排序。如果循环规定执行 N 次，那么就选择排列在前 N 位的执行情况，以使循环体达到最大的执行时间。通过这种选择过程，会发现程序执行的某些特定规律，依据这种规律就可以对循环体先进行一些变化，之后进行剪枝。

假定循环体内部有两个分支 A 和 B，且 A 和 B 中出现的指令的集合分别为 S_A 和 S_B。由于处理循环分支之前我们已经进行了平铺操作，因此可能有部分指令同时出现在 A 分支和 B 分支中，将这一集合记作 $S_{A\cap B}$。假设 A 的指令条数比 B 多，那么首次载入的执行情况下，分支 A 的执行时间是最长的。因此第一次执行选择分支 A。在后面的循环中，如果继续执行分支 A，那么所有 S_A 中指令都是命中，用 T_1 代表其执行时间；如果转而执行分支 B，那么所有 $S_{A\cap B}$ 中的指令都为命中，执行时间为 T_2，所有(S_B-S_A)中的指令都为不命中，记其执行时间为 T_3。如果 $T_2 + T_3 > T_1$，那么第二次循环应执行分支 B。从第三次循环开始，两个分支的指令都已经载入 Cache，因此任何一条分支的执行都是 Cache 命中，那么在以后的循环次数中，就应该选择在都命中情况下执行时间最长的分支执行，在这个例子中就是分支 A。

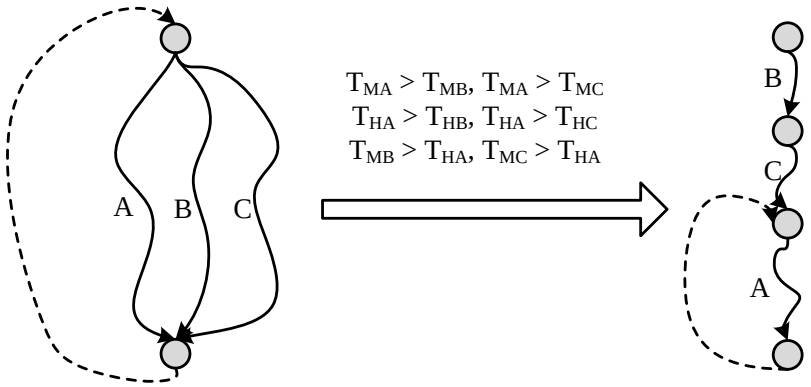


图 4.7 单层循环中的程序变形与剪枝

Fig. 4.7 Program transformation and branch cutting in single-nested loops

如果循环体分支数量大于 2，那么情况与上面的例子类似。基于假设 3，所有指令的执行时间为 1，因此第一次执行时间最长的分支应该是指令数量最多的分支。同时，所有的分支在载入 Cache 之后的执行中，这个分支的执行时间也是最长的。假定 A 是所有分支中指令数量最多的分支，则将其他各分支的载入执行时间与 $T_{H.A}$ 进行比较，如果前者较大，说明载入该分支的执行时间更长，那么对应的分支就执行一次。令所有这样的分支的集合为 S。如图 4.7 所示，我们可以把循环体转化为一个新的结构，这个结构由 2 个连续的部分组成，其中前一部分是集合 S 中各分支依次执行 1 次；第二部分是一

个只有分支 A 的循环体，如果原来的循环上限为 N，新的循环体的循环上限则为(N - |S|)。通过这种循环体变形的方式，实现了剪掉循环体中多余分支的结果。算法 4.1 就是基于上述程序变形思路的剪枝算法。

算法4.1　基于程序变形的单层循环剪枝算法

输入：循环体对应的CFG——CFG_{old}，循环上限为N
输出：变形及剪枝后的局部程序CFG——CFG_{new}

对循环体进行“平铺操作”，得到的所有分支放入集合S
将可展开分支集合S_E置为空
所有分支中所有基本块的访问特性记作MISS
找到全命中情况下执行时间最长的路径<*max*>，及其执行时间T_H
$S = S - \{<max>\}$;　$path = <max>$
while (S不为空)
　for (集合S中所有路径<*i*>)
　　for (路径<*i*>上的所有基本块*BBj*)
　　if (*BBj*出现在*path*中)
　　*BBj*的访问特性设置为HIT
　　endfor
　　依各基本块访问特性，计算路径<*i*>的执行时间，记做T_i
　　if ($T_i < T_H$)
　　剪掉路径<*i*>
　endfor
　$path = max\{T_i \mid for\ all\ <i>\ in\ S\}$
　$S = S - \{path\}$;　$S_E = S_E + \{path\}$
endwhile
对于S_E中的所有分支，依次组合成顺序执行的CFG，记作*C1*
构造以<*max*>分支为循环体的循环CFG，记作*C2*
*C2*循环上限 = $N - |S_E|$
将C1和C2顺序组合，构成CFG_{new}
return CFG_{new}

图 4.8 基于程序变形的单层循环剪枝算法
Fig. 4.8 Program transformation based branch cutting for single-nested loops

如果循环体的大小大于 Cache 的容量，就意味着循环体的一个部分可以将另外一个部分的内容替换出 Cache，具体可能表现为两个分支之间的相互替换。在这种情况下，一旦某条路径被替换出去，那么再次执行的时候将会出现大量的不命中，就可能出现循环体在多个分支之间往复跳转的执行情况。可能客观上程序在不同路径间的往复就没有显著规律，因此通过程序变形再进行剪枝的办法将难以奏效。对于这种情况下如何探索循环体的执行规律，并进行对应的程序转换和剪枝，将作为本文的未来工作。下面讨论在存在多层循环的情况下，如何进行合理的程序变形及剪枝。

4.5.2 多层循环的剪枝

在分析普通分支结构的时候，多层循环所带来的主要问题是：外层循环的执行使得内层循环存在多个初始状态。例如在某次进入内层循环时，可能部分指令由于上次执行而被载入 Cache，同时在外层循环的执行过程中没有被替换出去，那么这些指令在本次进入内层循环的时候将出现命中的情况。由于 FIFO 算法的不可预测性，这种现象有可能导致程序分支中复杂的命中/不命中情况出现，进而可能导致内层循环在多个分支之间往复执行，其规律非常难以把握。

因此，我们仅处理一种特殊的情况，即一个嵌套的多层循环可以完整放入 Cache 的情况。在这种情况下，无论是外层循环还是内层循环，一旦被载入 Cache，之后的各次执行在 Cache 中都将是命中的情况。由于整个嵌套循环都能够放入 Cache 中，那么每个内层循环也都可以完整的放入 Cache。循环体除了在第一次执行中被载入 Cache 外，其他各次执行的初始状态都是所有指令完全命中。这种程序就存在剪枝的可能。

显然，对于这种特殊情况，我们可以遵循从内层循环向外层循环依次分析的思路。对于内层循环而言，处理方法与单层循环是类似的。首先采用基线方法或扩展方法对内层循环进行剪枝；如果这两种剪枝方法无效，则采用算法 4.1 对程序进行变形后再剪枝。但是基于程序变形的剪枝方法，在多层循环分析的时候会有一些差别。

如果待分析的循环是内层循环，那么这个内层循环可能被多次进入。以图 4.6 为例，只有在第一次进入内层循环的时候，才会出现 $T_{MB} > T_{HA}$ 以及 $T_{MC} > T_{HA}$ 的情况。对于后面的若干次内层循环的执行，由于类似 B 和 C 这样的分支都已经载入了 Cache，所以对应的内层循环仅需执行 A 分支即可。因此在分析多层循环的时候，需要对这一情况进行区分和处理。我们所采用的处理办法是在内层循环进入的时候区分是第几次进入，如果是第一次，就按照图 4.6 转换后的结构执行，执行完毕后对 B 进行标记；在以后的内层循环的执行中，根据标记的情况，跳过 B 的执行，并同时对应调整分支 A 所在循环的执行次数。这一区分过程在模型检测器中是很容易建模的，这里不再详述细节。

在多层循环中，一种可能的情况是，内层循环存在于外层循环的某一个分支中，如图 4.9 所示。在这种程序中，如果分析外层循环，就涉及到分支 A 和分支 B 是否被剪掉的问题，而这一问题的实质是如何计算分支<CDEF>的执行时间。我们知道内层循环首次进入时的执行时间最长，其它各次的执行时间相同且最短。因此，把首次进入内层循环时分支<CDEF>的执行时间作为这一分支执行时间的上限；对应的内层循环其他各次执行时间，也就是分支<CDEF>所有指令为一定命中情况下的执行时间设置为下限。分支 A、分支 B 和分支<CDEF>就可以按照前面介绍的算法进行剪枝。

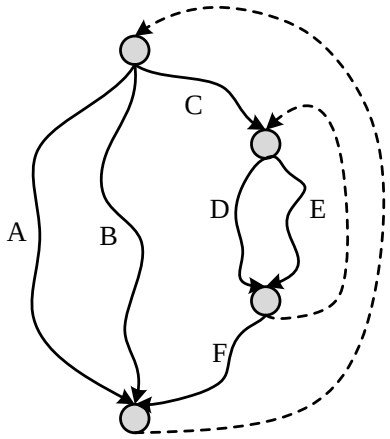


图 4.9 两层嵌套循环示例

Fig. 4.9 An example of two-level nested loop

如果外层循环体积较大而不能完整的放入 Cache，那么就意味着循环体的一次执行可能导致自身的指令被替换出 Cache。那么每次进入内层循环时，入口处的 Cache 状态是难以确定的。甚至即使内层循环能够完整放入 Cache，由于初始状态的不可知，导致分析内层循环在何种情况下才能全部载入 Cache 非常困难，因此进行剪枝的可能也随之变小。在这种情况下，我们只能利用模型检测工具来检测所有可能的执行情况。

综合前面的讨论，我们可以首先将待分析程序中所有的“旁路边”剪掉；其次，对符合要求的循环或嵌套循环进行程序转换以及剪枝。对于无法通过程序变形而进行剪枝的循环体，其原程序的结构完全保留。根据剪枝后的程序流程图生成对应的程序模型，交模型检测器进行验证，求解 WCET。

4.6　实验结果与分析

表 4.1 测试程序

Table 4.1 Benchmark programs

程序名称	程序描述	指令条数	循环上限
fir	FIR 过滤器程序	145	16
minver	浮点数矩阵翻转算法	803	16
janne	快速离散余弦变换函数	74	16
expint	求指数运算程序	227	8
crc	CRC 校验算法	462	8
bsort	冒泡排序算法	137	8

在本章实验中，我们设置了一个拥有 64 个组的 4 路组相联 L1 Cache。Cache 块大小为 8B，等于 PISA 指令集的一条指令长度。因此该 L1 Cache 最大可容纳 256 条指令。我们采用模型检测技术对基于 FIFO 替换算法的单核系统独立指令 Cache 进行了建模。

模型的基本框架与第 3 章是类似的。主要不同在于在对每个基本块进行建模的时候，加入了访问 Cache 的动作。模型内部对指令 Cache 进行了建模，基本块中的各指令对 Cache 的访问和修改体现为对模型中 Cache 数据结构的操作。表 4.1 是本章实验所采用的 6 个测试程序。测试程序的选择与第 3 章类似，但是去掉了那些不需要进行剪枝优化的简单程序。由于本章实验的主要目的是验证剪枝算法的有效性，因此每个程序的循环上限为固定值，列于表 4.1 中。同时我们统计了原程序的循环个数和指令条数。

表 4.2 实验结果

Table 4.2 Experiment results

程序	WCET	剪枝	搜索次数	时间(秒)	内存(MB)
fir	14,492	N	20	1,132.82	99.408
		Y	20	1.05	3.512
minver	231,672,358	N	27	7,165.58	4,130.816
		Y	27	3,836.22	2,327.132
janne	13,454	N	16	145,032.23	99.677
		Y	16	1.18	3.512
expint	4,328	N	22	16,478.52	279.100
		Y	22	2.55	55.104
crc	10,432	N	22	11,170.60	280.052
		Y	22	4,309.19	130.016
bsort	9,180	N	15	2,518.04	71.832
		Y	15	0.58	3.512

表 4.2 分别列举了 6 个测试程序在不采用剪枝算法和采用剪枝算法两种情况下进行 WCET 分析的实验结果。我们分别列举了每个测试程序分析得到的 WCET 估计值，二分搜索的次数，以及时间和内存的消耗，这里分别予以解释。首先对于 WCET 的分析值，不管采用剪枝算法与否，得到的结果都是相同的。一方面，这一结果验证了本章剪枝算法的正确性，即没有出现过低估计（Under-estimation）的情况；另一方面，该结果说明本章的剪枝算法并没有牺牲分析精度来换取分析性能。此外，为保证对比的公平性，对于每个程序，无论采用哪种分析方法，二分搜索都设置了相同的上限和下限值，因此二分搜素的次数也是相同的。

从表 4.2 的数据来看，剪枝算法在提高分析性能方面的效果是十分显著的。fir，janne，expint 和 bsort 四个程序在分析性能上的提升最为明显，相比于不采用剪枝算法，分析时间缩短到了近 1/1000，同时内存消耗上也有显著的降低。对于 fir 测试程序，某个循环

体内部是由 4 个分支前后连接组成的，这就意味着循环体有 16 条可能的路径。分析发现该程序大部分的分支都是由 4.4 小节所讨论的“旁路边”所组成，因此剪除旁路边的操作有效的降低了该程序的状态空间。同时剪除旁路边之后，循环体内部剩余一个两分支的结构，通过 4.5.1 小节的基于程序变形的剪枝办法，我们可以将其中的某一个分支展开到循环体外，从而导致该循环体为单路径程序。以上两点原因使得对 fir 程序的分析在性能上有了显著提高。Janne 程序主要是由一个两层嵌套的循环组成，其中内层循环体有三个分支构成 8 条可能的路径，我们通过剪除其中的一条旁路边，同时使用程序变形的办法将内层循环体构造成了单路径的程序，因而分析性能得到了提高。bsort 程序和 expint 程序性能得到提升的原因主要来自旁路边的剪除。

相比之下， minver 和 crc 程序的分析性能的提升不如前面 4 个程序显著，但是分析时间和内存消耗平均都下降到了未剪枝之前的 1/3。在多层循环剪枝的讨论中，我们指出：如果内层循环体可以展开，那么对于展开的部分需要进行判断，如果是第一次进入循环体，那么执行展开部分，否则不执行展开部分。这等价于是在内层循环的上一层添加了一个分支结构，因此在某种程度上削弱了剪枝的效果。如果程序循环层次比较深，结构复杂，那么剪枝得到的收益就会下降。minver 和 crc 两个程序就具备类似的特征。尤其是 minver 程序，由于某些嵌套循环的指令个数已经超过了我们设置的可存放 256 条指令的 L1 Cache，因此无法对其实施剪枝操作，故其提升幅度是最小的。

综合上面的实验结果，我们可以得出如下一些结论。整体来看，对大多数程序的分析都可以通过剪枝技术获得分析性能的提升，同时这种性能提升不会对分析精度造成任何影响，这是本章工作最重要的意义所在。特别地，我们发现类似“旁路边”这样的程序结构在实际程序中是广泛存在的，而模型检测器本身并不能识别这种特性，因此无法对旁路边造成的分支数量的增加进行有效的削减。在采用剪枝技术之后，通过削减旁路边可以使这些程序的分析性能得到显著提高。本章还讨论了基于程序变形的剪枝技术，这种技术被证明也是有效的，但是其问题是，在分析复杂嵌套程序的情况下，剪枝的收益将有所下降。因此，需要继续深入研究面向多层嵌套循环的剪枝办法，解决剪枝收益下降这一问题。

同时，本章处理多层循环的一个假设就是，整个嵌套的循环结构都能够放入指令 Cache。这实际上从某个方面限制了可分析的程序的体积，导致剪枝技术无法用于大型程序的分析。但是通常情况下，即使外层循环不能完全放入 Cache，但是体积较小的内层循环是有可能完全放入 Cache 的，这就使得我们可以对内层循环进行某种形式的剪枝。但是解决这一问题的主要挑战来自 Cache 替换算法。不同的替换算法的可预测性也不尽相同，可预测性较差的算法可能造成内层循环入口的 Cache 上下文非常难以预测，为得

到较高精度的分析结果，就不得不让模型检测器来考虑所有的可能，这就降低了剪枝的几率。未来我们将针对这一问题进行深入研究，扩展本章所讨论的剪枝算法。同时，我们将探索分析性能和分析精度之间的更好的平衡点，例如对程序进行某种抽象，用分析精度上的少许牺牲换取分析性能上的显著提高。同时，将更加深入的研究其他 WCET 分析技术与模型检测技术的可能的结合，从而提出一些兼具分析精度和分析性能的混合分析框架。

4.7　小结

本章主要讨论了采用模型检测技术分析单核系统中的 Cache 行为的方法，为了提高分析性能，本章提出了一种基于剪枝思想的分析方法，削减 WCET 计算过程中不必考虑的程序分支。实验结果表明本章所提出的剪枝技术是安全的，分析所消耗的时间和空间显著降低，分析的可伸缩性得到了有效提高。

第5章 多核共享指令 Cache 分析

实时嵌入式系统广泛存在于航空军事、工业控制、汽车电子、网络基础设施、网络安全、无线通信基础设施、数字医疗和个人电子等应用领域[102,103]。近年来，上述应用领域正面临着系统功能不断提升的局面，由此带来的结果是实时系统的复杂度迅速提升，进而对系统性能提出了越来越高的要求。随着半导体技术向深亚微米技术的发展，依靠提高时钟频率和挖掘更高的指令级并行性来提升处理器的运算性能的技术手段所能获得的性能收益越来越小，同时功耗问题也成为了芯片设计的一个主要障碍。计算机处理器设计正在经历从单核处理器到多核处理器的巨大转变[104]。由于多核处理器在性能上具有非常好的规模可伸缩性，因此得到了实时嵌入式领域的广泛关注。目前，主流的多核系统通常将最后一级 Cache 设计为共享，这一设计已经被验证具有“灵活利用有限 Cache 容量[105]，有利于在 Cache 级别上实现高效数据共享[106]，减少由于一致性保证所导致的缓存抖动现象[107]”等优势特性。但是在带有共享 Cache 的体系结构中，多个核心上的任务对共享 Cache 的细粒度访问造成了系统时间行为的复杂性和不可预测性，进而给实时系统的时间特性的分析（尤其是程序 WCET 分析）带来了巨大挑战。

静态 WCET 分析中的一个重要任务就是处理器行为分析，而 Cache 分析是处理器行为分析中最重要的一个部分。Cache 分析的目的是得到程序的指令和数据在 Cache 中的命中情况。目前主要有两大类 Cache 分析技术：一类是基于 Cache 状态冲突图的分析技术[23]，一类是基于抽象解释的分析技术[28]。这两类技术的本质都是解析一个程序自身的指令和数据在访问 Cache 时可能产生的地址冲突，并依据这一信息分析所有指令和数据在 Cache 中的命中情况。传统技术在基于独享 Cache 的单核处理器的分析上取得了较好的分析结果，但是不适用于多核共享 Cache 的分析。这是因为在多核共享 Cache 体系结构中，影响程序的 Cache 命中情况的因素，不仅仅包括程序内部指令和数据间的冲突，同时也包括在其它核心上正在运行的任务并行访问共享 Cache 所产生的干涉。由于这种程序间的干涉非常不可预测，因此分析任何一个核心上程序的 Cache 命中情况都变得异常困难。目前已经有相关工作[31,32]采用改造传统手段的方法进行多核共享 Cache 的分析，但是分析结果比较悲观。基于这一问题，本章主要研究基于模型检测技术的多核共享 Cache 行为的分析，利用模型检测技术的强大建模能力，对系统进行精确建模，进而获得更加精确的分析结果。

在本章后续小节中，我们首先阐述多核共享 Cache 体系结构给实时系统分析带来的新问题，并介绍多核共享 Cache 分析的相关工作；其次，介绍本章工作所涉及的多核共

享 Cache 的模型及假设；在此基础上，我们提出了基于模型检测技术的面向多核共享 Cache 体系结构的 WCET 分析方法，并通过实验分析了本章所提出的分析方法的分析精度和分析性能，同时对该分析方法的可伸缩性进行了讨论。

5.1 相关工作

在本章的相关工作中，我们首先介绍硬件体系结构向多核转变的必然性，并阐述多核共享 Cache 这一设计给实时系统分析带来的新问题。之后介绍在面向多核共享 Cache 的 WCET 分析领域的相关工作及存在的主要问题。

5.1.1 多核共享 Cache 给实时系统分析带来的新问题

传统的微处理器工业主要通过提高时钟频率和挖掘更高的指令级并行性来提升处理器的运算性能。但是在当前半导体技术下，这种发展趋势面临很多突出的问题[108,109]：时钟频率的提升使得芯片功耗大幅增加；而通过设计更复杂的处理核心提高指令级并行，进而获取性能提升也越来越困难。因此芯片体系结构正在经历从单核处理器向多核处理器的转变。所谓多核处理器，是指在同一芯片内部集成多个处理器核心，这些核心通过片内总线或片上网络方式进行连接，片内设计有独占或共享的存储资源。与单核处理器相比，多核处理器可在单位功耗下提供更高的运算性能；并且随着芯片集成度的提高，这种性能提升具有更好的规模可伸缩性，因此多核处理器成为芯片发展的必然趋势。

多核处理器在实时嵌入式系统中也有着巨大的应用潜力。实时嵌入式系统具有一些突出的特点：强实时性，低功耗，高并行性，高可靠性以及复杂多变的应用需求。上述综合需求使得多核处理器非常适合于实时嵌入式系统，主要表现在以下几个方面。首先，功能日益复杂的实时嵌入式应用要求系统在单位功耗上提供更大的运算性能，而这恰是多核体系结构的最大优势。其次，实时嵌入式应用本身具有很高的任务级和数据级并行性，例如多通道数据采集和处理、移动多媒体计算，这种任务级和数据级的高并行性需求非常适合多核体系结构的性能特点[110]。同时，在一个芯片上集成多个处理核心，有利于建立更加低成本、低功耗的冗余系统以增加系统的可靠性。而基于 SMP 和共享 Cache 的处理器结构具有很好的灵活性，可以根据不同应用需求建立对称或非对称的计算系统。目前， TI[111]，ARM[112]， Freescale[113]，PicoChip，RENESAS 等嵌入式厂商都推出了嵌入式多核处理器。

多核处理器的设计原则与传统的单核处理器有着本质的不同，这就要求设计者必须尽可能挖掘应用程序的任务级并行性和数据级并行性[114,115]，以有效利用多个处理核心所带来的运算潜能。通过任务级和数据级并行化，应用程序被划分为很多并行线程，这

使得工作负载更均衡的被分配到多个处理核心上执行，进而提高了系统的整体平均性能。多核体系结构在显著提高芯片运算性能的同时，也导致了处理速度与访存速度之间的差异进一步加大[116]。多核处理器通常在片内集成 Cache 以减少系统主存高延时和低带宽造成的性能瓶颈。多数多核处理器采用共享二级 Cache 的结构设计，其主要优点是：灵活利用有限的 Cache 容量[105]，利于在 Cache 级别上实现高效数据共享[106]，减少由于一致性保证所导致的缓存抖动现象[107]。但是另一方面，多核处理器中共享 Cache 的设计也加重了系统行为的不可预测性。相关研究表明[117]，多个核心上并行执行的任务，由于共享二级 Cache 产生的访问冲突，导致二级 Cache 上不命中次数显著增加，且增加幅度变化大，难以预测。而任务本身的访存特性决定了它们特有的 Cache 访问行为，这正是造成不同冲突结果的主要因素。难以预测的任务间冲突导致不可预测的 Cache 访问行为，进而导致系统的时间行为高度不可预测。这种不可预测性的一个直接而且严重的结果就是使得实时系统的时间特性分析变得非常困难。

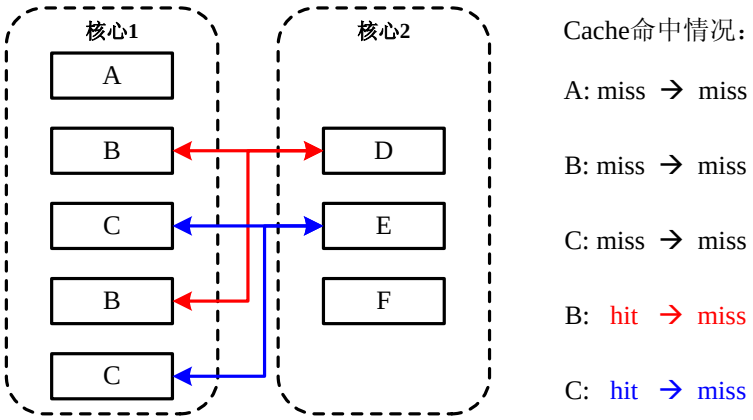


图 5.1 多核共享 Cache 干涉示例

Fig. 5.1 An example of conflicts in shared caches of multicores

如前所述，多核系统正在以不可逆转的趋势越来越多的应用于实时系统，而共享 Cache 也已经成为了多核 Cache 体系结构的主流技术。在这种背景下，进行面向多核共享 Cache 的 WCET 分析的研究就变得非常重要，同时具有很大的挑战性。图 5.1 是多核共享 Cache 中干涉影响 WCET 分析的例子。图中 A-F 表示的是指令，从上到下是指令的执行次序。对于核心 1 上的程序而言，如果不考虑另一个核心上程序的干涉，那么第 4 条指令 B 和第 5 条指令 C 的执行在 Cache 中应该是命中。如果考虑另外核心上程序执行的影响，那么情况完全不同。假定核心 2 的指令 D 和核心 1 的指令 B 在地址上会冲突（红线），核心 2 的指令 E 和核心 1 的指令 C 在地址上冲突（蓝线）。如果核心 1 上的程序在执行完第 2 条指令 B 后，核心 2 上的指令 D 得到了执行，那么 D 将把 B 从 Cache 中替换出去，当核心 1 执行第 4 条指令 B 的时候，将产生“不命中”的结果。C 的执行

情况类似。在面向多核共享 Cache 的 WCET 研究中，Cache 分析的主要目的就是精确高效的分析出程序之间的这种干涉情况，并将其考虑到 WCET 的计算中。

5.1.2 面向多核共享 Cache 的 WCET 分析相关工作

目前这一领域的研究工作开展的还很不充分，仅有美国南伊利诺伊大学的 Zhang 和 Yan 对这一问题进行了研究。在文献[31]中，作者提出了一种对传统的“隐式路径枚举+抽象解释技术”的分析方法的改进，用以分析共享 L2 Cache 的双核系统。传统分析框架中的 Cache 分析可以得到一个程序的每条指令在单核处理器上各级 Cache 中命中与否的结果，根据这一结果就可以计算出整个程序在最坏情况下的执行时间。作者在这篇文献中提出的基本思想就是：首先用传统分析方法得到程序在 L1 Cache 和 L2 Cache 上的命中与否的结果；之后分析在另一个处理器核心上执行的程序对待分析程序的干涉情况。程序在没有受到干涉时的一些 Cache 命中，在考虑了干涉效果之后将变为 Cache 不命中。通过这种分析得到考虑多核干涉情况下的程序 WCET。这个分析方法的一个隐含的假设是：“如果两条指令能够映射到一个 Cache 块上，那么就会发生冲突”，这正是该分析方法的最大缺陷。这是因为，程序的执行是需要时间的，即使两个程序中的某两条指令在地址上会发生冲突，但是在实际执行的过程中这两条指令在时间上是完全可能错开的，在这种情况下还认为这两条指令会相互干涉，就会导致分析过于悲观。另一方面，文献[31]所提出的方法仅能适用于“直接映射（Direct-Mapped）”的 Cache，无法分析组相联的共享 Cache。

在文献[32]中，Zhang 和 Yan 等学者基于文献[31]中尚存在的问题进行了改进。改进之一就是采用基于 Cache 冲突图的技术手段替代了抽象解释技术进行 Cache 分析，并把对共享 L2 Cache 的分析纳入基于 Cache 冲突图的技术框架中，使得分析精度更高。同时，在对不同核心 Cache 冲突的分析中，作者考虑了潜在冲突指令在执行时间上的先后关系，基于这一信息可以将一些地址上冲突但实际执行中一定不会冲突的指令分析出来，从而提高了分析结果的精度。文献[32]的分析方法的问题主要在于，无论是分析框架采用的“Cache 冲突图”技术，还是对指令的执行时间关系的分析，其复杂度都非常高。同时，改进后的技术仍旧不能够分析组相联 Cache，这也是其一大缺陷。

5.2 多核共享 Cache 体系结构模型与假设

本节将介绍基于共享末级 Cache 的多核体系结构，以及各主要部分的硬件行为及其参数。图 5.2 是本章所采用的体系结构。处理器部分包括了若干个处理核心以及各级 Cache。处理器通过片外总线与系统主存以及各种外设相连。目前在实时嵌入式系统中

常见的核心数量为 2、4 或 8 个。在本章的讨论中，为描述方便，我们假定采用 2 核心的多核处理器，相关的方法可以很容易扩展到 N 个核心的处理器。目前的处理器大多采用多级 Cache 的设计，通常前面的各级 Cache 为各核心独享，而最后一级 Cache 为所有核心共享。本章所研究的 Cache 具有两级，其中 L1 Cache 为各核心独享，且指令 Cache 与数据 Cache 分离；L2 Cache 为各核心共享，指令 Cache 和数据 Cache 不分离。目前这是最常见的一种多核 Cache 组织结构。下面的讨论将基于这一 Cache 体系结构开展。

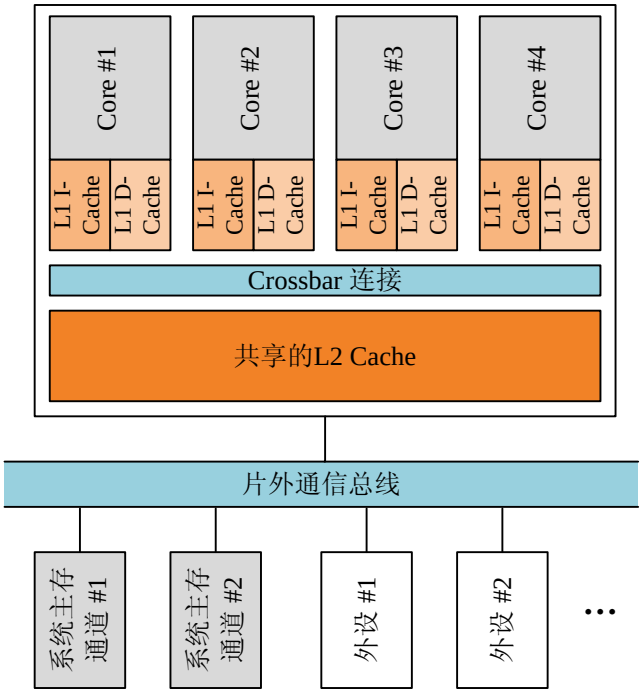


图 5.2　多核共享 L2 Cache 体系结构

Fig. 5.2 The multi-core architecture featured by shared L2 Cache

此外，我们需要对本章所分析的 Cache 体系结构的具体行为进行假设，假设的主要原则是尽量符合主流多核系统的设计。但是为方便本章的问题讨论，我们去除或简化一些部件的具体行为。本章所涉及的体系结构假设包括：

假设 1：数据 Cache 是完美的，也就是说，数据 Cache 在任何级别上对指令 Cache 都不会产生任何影响；

假设 2：所有各级 Cache 均为组相联 Cache，由于直接相联是组相联的一种特殊形式，因此所有结论也适用于直接相联 Cache；

假设 3：Cache 块大小是任意的；

假设 4：不限制所采用的替换策略，但是本章以 LRU 策略为例阐述问题；

假设 5：仅当对 L1 Cache 访问是“不命中”时才访问 L2 Cache，对两级 Cache 不能够并行访问；

假设 6：当任何一级 Cache 访问出现“不命中”时，都要从下一层将数据载入；

假设 7：除假设 5 的访问行为以及 Cache 搜索以外，不定义其他的 Cache 行为；

假设 8：所有核心的 L1 Cache 与共享的 L2 Cache 之间采用 Crossbar 的连接方式，所有核心访问共享的 L2 Cache 具有完全相同的访问时延；

假设 9：同一时刻允许多个核心访问共享的 L2 Cache 不同组；如果在同一时钟周期，多个核心同时发起对 L2 Cache 的访问，那么多个核心访问 L2 Cache 的次序是不确定的；

假设 10：所有核心对主存的访问是带有缓存机制的，也就是说，多个核心同时访问主存不会在总线上产生冲突而导致访问时间增加；

假设 11：流水线是完美的，即每条指令的执行时间为 1 个时间单位。

我们对上述假设进行简要的解释。假设 1 主要是排除数据 Cache 的影响。在实际系统中，指令 Cache 和数据 Cache 的行为特性具有很大的差异，对指令 Cache 的分析方法通常难以直接应用到数据 Cache。由于本章的主要研究对象是指令 Cache，所以我们假设数据 Cache 不对指令 Cache 造成影响。假设 2、假设 3 和假设 4 是对 Cache 组织形式的约定。目前几乎所有的处理器都采用了组相联的组织形式，由于直接相联是组相联的一种特例，因此本章讨论的方法同样适用于直接相联 Cache。通常在桌面处理器中采用 LRU、Psudo-LRU 替换算法，在嵌入式处理器中采用 LRU 或 FIFO 替换算法。由于本章所采用的分析工具是模型检测器，那么对于任何一种替换策略的建模都非常容易，因此我们仅以 LRU 为代表进行讨论。假设 5、假设 6 和假设 7 是对不同级别 Cache 之间行为的约束。多级 Cache 有三类：第一类是 Inclusive Cache，即第 N 级 Cache 中的内容一定存在于第 N+1 级 Cache；第二类是 Exclusive Cache，即相邻两级 Cache 的内容一定没有交集；第三类是 Non-Inclusive Cache，这类 Cache 不约束相邻两级 Cache 之间的包含关系。本章主要研究的就是 Non-Inclusive Cache，因此存在上述三个假设。假设 8、假设 9 和假设 10 约定了各级 Cache 以及 Cache 与主存之间的时延特性，以及多个核心在共享 Cache 上发生冲突时的访问顺序问题。假设 11 简化了流水线对 Cache 行为的影响。后面的设计和讨论，将完全基于上述假设。

5.3 基于模型检测技术的多核共享 Cache 行为分析

本节主要介绍本章工作所采用的整体分析框架，以及采用 UPPAAL 模型检测器建模的具体方法。

5.3.1 分析框架

本节首先对本章的整体分析框架进行介绍。后面的讨论，我们都假定针对两个核心的多核处理器，但是其原理和方法完全适用于多于两个核心的多核处理器。

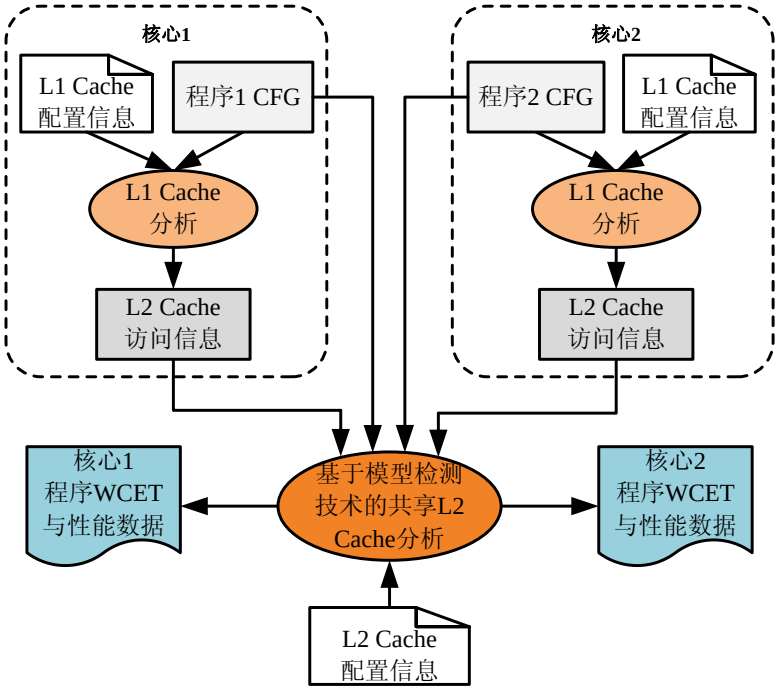


图 5.3 Cache 分析框架

Fig. 5.3 The cache analysis framework

图 5.3 是本章分析框架的示意图。整个分析过程的输入就是分别位于核心 1 和核心 2 上的两个程序。由于本章仅讨论指令 Cache，所以我们把一个程序的执行看作是一个指令序列。这个指令序列首先输入给“L1 Cache 分析器”，该分析器的功能是按照基于抽象解释的 Cache 分析技术，分析得到程序在 L1 Cache 中的命中情况。根据本章对多级 Cache 的假设，指令在 L1 Cache 中的命中情况为不命中时才可能产生对 L2 Cache 的访问。因此根据 L1 Cache 分析的结果，我们将原始的指令序列转化为对 L2 Cache 的访问信息。这一信息结合程序的 CFG，再加上 L2 Cache 的配置信息将作为本章的主要分析模块“基于模型检测技术的共享 L2 Cache 分析”的输入。分析结果是两个程序的 WCET，同时我们还将记录分析的性能数据，以便对该方法进行评价。

本框架中的“L1 Cache 分析器”采用了 Hardy 等人提出的多级 Cache 分析方法[118]。之所以采用现有方法，而没有使用模型检测技术分析 L1 Cache 主要是出于分析效率的考虑。如果采用模型检测技术对 L1 和 L2 两级 Cache 进行分析，那么所产生的模型的状态空间将会非常大。同时本章工作也难以使用第 4 章中所设计的剪枝技术，主要原因是在多核环境中，程序间的干涉情况非常复杂，而在这种复杂且不可预测的状况下难以安全的对程序进行剪枝。因此本章采用现有的基于抽象解释的静态 Cache 分析方法进行 L1 Cache 的分析。而在基于共享 Cache 的多核体系结构中，对末级共享 Cache 的分析才是真正的难点所在。本章采用模型检测技术对共享 L2 Cache 进行分析的根本目的就是提高这部分的分析精度。

Yan 和 Zhang 等学者在[31]中提出的分析技术精度是比较低的。造成这一结果的主要原因来自于两大方面：（1）在分析指令在共享 L2 Cache 中的冲突时，仅仅考虑了指令地址，而没有考虑指令执行的时间次序。举个简单的例子：核心 1 上的指令 $S_{1,i}$ 与核心 2 上的指令 $S_{2,j}$ 根据其地址映射到同一个 Cache 块上，假定 Cache 是直接相联的，那么这两条指令在共享 L2 Cache 中就存在潜在的冲突。如果通过其他某些手段，我们可以知道指令 $S_{2,j}$ 执行的时候指令 $S_{1,i}$ 一定已经执行完毕，那么这两条指令就一定不会在共享 L2 Cache 中发生冲突。如果能够获得这类分析的中间结果，那么将使得 WCET 分析的精度得到提高。而文献[31]正是由于没有考虑这种信息而导致分析精度太差。（2）采用基于抽象解释等技术分析指令序列在 L1 Cache 中的命中情况时，有一种可能的情况是“不可确定（Not Classified）”，也就是说一条指令在 L1 Cache 中不能确定结果一定是命中或一定是不命中。在进行 L2 Cache 分析的时候，要考虑最坏情况，也就是把这种不可确定的情况当做是 L1 Cache 不命中。这样一来将产生一次 L2 Cache 访问。如果在实际中这条指令的访问在 L1 Cache 上是命中的结果，那么过度估计就出现了。

正是针对上述相关工作的主要问题，本章提出采用模型检测技术分析基于共享 L2 Cache 的程序 WCET。由于在模型中我们可以对每条指令的执行时间进行详细准确的建模，因此指令执行的时间次序就很容易的被发掘出来，并直接导致分析结果更加精确。下一小节，我们将详细介绍采用 UPPAAL 模型检测器对面向多核共享 Cache 的 WCET 分析进行建模的技术。

5.3.2 L1 独立 Cache 分析

上一小节所介绍的分析框架中存在一个 L1 Cache 行为的分析模块，这一模块的主要功能是分析程序指令在 L1 Cache 中的命中情况，而只有指令在 L1 中不命中的情况下，才会访问共享的 L2 Cache。由于 L1 Cache 分析并不是本章的重点，所以我们在这个部分采用了 Ferdinand 等人所设计的基于抽象解释的 Cache 分析方法[28]，该方法已经被验证为单核独立 Cache 分析中一种高效且精度较高的分析手段。该方法的根本思想是，静态模拟程序的执行，在这一过程中，分析由程序执行所造成的 Cache 内容的变化情况，并从中提取命中信息。在程序的实际执行过程中，每执行一条指令，Cache 内容都会发生相应的变化，因此针对 Cache 的具体内容进行分析的复杂度过高。Ferdinand 等人采用抽象技术，将具体的 Cache 状态转化为对应的抽象 Cache 状态（Abstract Cache State），并建立一组对应于抽象 Cache 状态的动作，其语义是反映程序执行对抽象状态的改变。抽象 Cache 状态和对应的动作语义构成了 Cache 在抽象空间的完整行为的描述。程序的执行依照规定的动作改变抽象 Cache 状态，当抽象 Cache 状态收敛到固定点的时候分析

结束，此时可以根据指令在状态中的存在情况来确定命中与否这一结果。

基于抽象 Cache 状态的分析有三种，分别是 MUST 分析，MAY 分析，和 PERSISTENCE 分析。MUST 分析规定在抽象状态中，维护指令在对应的 Cache 具体状态中的最大年龄，用来确定哪些指令在 Cache 中一定是命中的。MAY 分析定义在抽象状态中，维护指令在对应的 Cache 具体状态中的最小年龄，其意义是分析哪些指令是可能存在于 Cache 中的，进而判断哪些指令一定不在 Cache 中。单纯采用 MUST 和 MAY 分析，在分析循环结构的时候往往得不到好的结果，这是因为通常情况下，循环中的很多指令在第一次访问的时候是不命中，而以后的各次访问都是命中，这种特性是无法通过简单的用 MUST 和 MAY 分析发现的。因此引入 PERSISTENCE 分析，主要目的是识别那些一旦被载入 Cache 将一定不会被替换出去的指令。

我们的研究发现，Ferdinand 等人所设计的 PERSISTENCE 分析是错误的，也就是说，其 PERSISTENCE 分析可能出现分析结果不安全的情况。下面我们首先简要介绍 PERSISTENCE 分析的语义，其次以一个具体例子阐述不安全结果形成的原因。

$$U_s^{pers}(s,m)=\begin{cases} if \exists h \in \{1,...,A\} : m \in s(l_h) \\ l_1 \mapsto \{m\} \\ l_i \mapsto s(l_{i-1}) | i = 2...h-1 \\ l_h \mapsto s(l_{h-1}) \cup (s(l_h) - \{m\}) \\ l_i \mapsto s(l_i) | i = h+1...A, \\ l_T \mapsto s(l_T) \\ otherwise \\ l_1 \mapsto \{m\} \\ l_i \mapsto s(l_{i-1}) | i = 2...A, \\ l_T \mapsto (s(l_T) - \{m\}) \cup s(l_A) \end{cases} \tag{5.1}$$

公式 5.1 描述的是在 PERSISTENCE 分析中，Cache 抽象状态与具体状态的对应关系。可以看出，PERSISTENCE 分析在维护指令在 Cache 中的年龄上类似于 MUST 分析，它维护的是指令在 Cache 中的最大年龄。公式 5.1 是 PERSISTENCE 分析中抽象 Cache 状态更新的语义。其中 l_i 表示一个 Cache 组中的第 i 行，$s(l_i)$表示该行上的抽象状态，A 表示的是 Cache 的相联度。以 LRU 替换算法为例，在一个 A 路组相联的 Cache 中，位置 1 表示最年轻的位置，位置 A 表示最年老的位置。公式 5.1 完整定义了当一个访存 m 发生的时候，它将如何改变 Cache 的抽象状态。

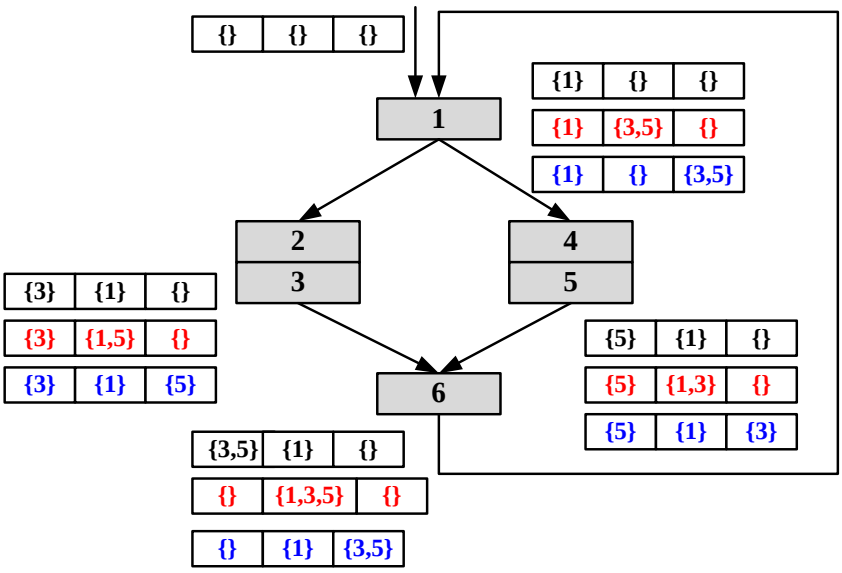


图 5.4 PERSISTENCE 分析错误举例

Fig. 5.4 An example of the error in the PERSISTENCE analsyis

图 5.4 表示的是对一段程序进行 PERSISTENCE 分析的结果，其中 1~6 表示指令的地址，而示例中的 Cache 是 2-路组相联，有 2 个 Cache Set。黑色表示分析过程中的 Cache 抽象状态，红色表示采用公式 5.1 的更新操作，分析终止时的 Cache 抽象状态。可以看到，由于指令 1、指令 3 和指令 5 都存在于结果抽象状态中，因此 PERSISTENCE 分析的结果是，这三条指令在首次载入 Cache 以后的各次访问中都将是命中。但是很显然，如果程序多次循环分别走左边和右边的路径各一次，那么指令 3 和指令 5 不可能除第一次外都是命中。这一错误的出现，主要原因是原有的 PERSISTENCE 分析在抽象 Cache 状态中维护了错误的年龄信息。具体错误出现在更新操作 $l_h \mapsto s(l_{h-1}) \cup (s(l_h) - \{m\})$ 这一行。为维护正确的年龄信息，我们将公式 5.1 修改为公式 5.2 的形式，由此更新操作得到的分析结果如图 5.4 中蓝色的部分所示，可见指令 3 和指令 5 的年龄被正确的维护，因此排除了原分析中存在的错误。

$$U_s^{pers}(s,m) = \begin{cases} if \exists h \in \{1,...,A\} : m \in s(l_h) \\ l_1 \mapsto \{m\} \\ l_i \mapsto s(l_{i-1}) | i = 2...A \\ l_T \mapsto s(l_T) \cup s(l_A) \\ otherwise \\ l_1 \mapsto \{m\} \\ l_i \mapsto s(l_{i-1}) | i = 2...A, \\ l_T \mapsto (s(l_T) - \{m\}) \cup s(l_A) \end{cases} \tag{5.2}$$

5.3.3 采用 UPPAAL 模型检测器的建模方法

第 3 章的实验结果表明，在 WCET 分析中，UPPAAL 模型检测器具有较好的“空间-时间”折中。相比之下，SPIN 模型检测器较容易出现空间爆炸问题，而 NuSMV 模型检测器容易出现时间爆炸问题。更重要的一点是，对共享 L2 Cache 进行分析，需要捕捉多个核心同时访问共享 L2 Cache 的问题，并且进行正确的操作，为此要维护一个全局时钟，以同步各核心上程序的 Cache 访问行为。如果使用 SPIN 或 NuSMV 模型检测器，那么需要人为的为时间建模。并且为捕捉任意时刻可能并行发生的多个访存行为，就必须依照时间驱动的办法进行建模，也就是让时钟以处理器周期为基本单位推进，整个模型在每个时钟周期上检查所有核心上的程序在当前时刻的行为。这样设计模型将大大影响验证效率。而 UPPAAL 模型检测器的理论基础是时间自动机，等于内置了对时间的建模，因此无需额外的建模时间。同时，UPPAAL 本身对基于时间自动机的模型设计了很多削减状态空间的办法。因此，UPPAAL 非常适用于分析带有时间同步需求的多核共享 L2 Cache 的行为。这是本章采用 UPPAAL 的主要原因。

下面的讨论中我们以图 5.5 为例进行说明。图 5.5 是我们人工设计的一个例子程序的 CFG，该程序主体为一个循环，循环体是一个分支结构。该程序共有 7 个基本块，9 条指令，其中基本块 BB1 和 BB6 各包含两条指令。每条指令上标记有“AM”，“AH”，“FM”，“NC”等信息，表示的是 L1 Cache 分析的结果，分别代表“一定不命中”、“一定命中”、“首次不命中”以及“不能确定”。为了方便说明问题，图 5.5 中的 L1 Cache 分析结果并不是从实际的程序分析中得到的，而是我们人工设置的。图 5.5 所表示的就是模型检测器的输入信息，下面我们将以此示例程序为例，介绍如何建立对应的 UPPAAL 模型。

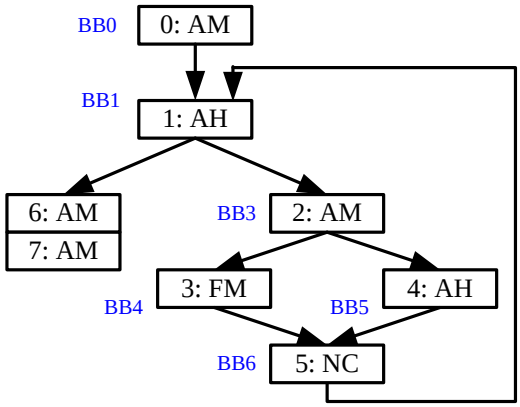


图 5.5 一个示例程序

Fig. 5.5 A demo program

UPPAAL 模型共分为三个部分：第一个部分为程序 CFG 对应的模型，第二个部分

为程序访问 L1 Cache 的行为模型，第三个部分为程序访问 L2 Cache 的行为模型。

程序 CFG 对应的模型与第 4 章中的模型是类似的，但是不同在于，第 4 章中每个基本块对应于一个 UPPAAL 节点，而在本章的模型中，每一条指令对应一组节点。这是因为在共享 Cache 中，程序间的冲突可能发生在某个程序的某两条指令之间，所以对 Cache 行为的建模应该以指令为基本单位。除了上述区别，程序的分支结构，循环结构，以及循环变量的控制等信息都在模型中得以保留。下面我们先根据指令的访存特性分别介绍各种指令的建模方法，之后给出示例程序所对应的程序自动机。

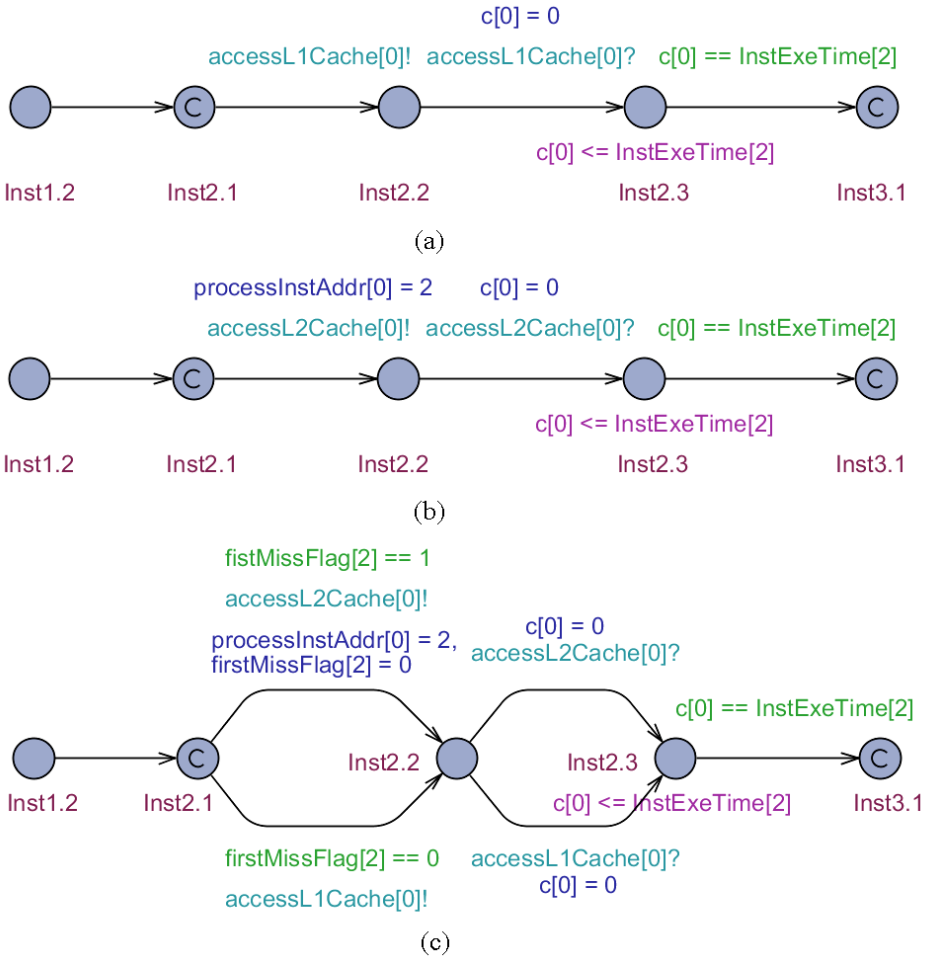


图 5.6 对指令的建模

Fig. 5.6 Modeling instructions

对于每条指令，在 UPPAAL 中的建模方法如图 5.6 所示。由于指令访问 L2 Cache 的行为有多种，因此需要根据不同情况对指令进行建模。图 5.6(a)表示的是指令在 L1 Cache 上的访问情况为 AH，在这种情况下，指令仅需访问 L1 Cache，而无需访问 L2 Cache。其中 Inst2.1 和 Inst2.2 两个节点用来建模一条指令，指令地址为 2。Inst2.1 节点的类型为“committed”，其含义是：在这个节点上时间不会推进，且一旦到达该节点，必须立刻执行某个状态转移离开该节点。这一设置对应的物理意义是“一旦到达该节点，指令 2

将马上得到运行”。在 Inst2.1 向 Inst2.2 迁移的边上，以及 Inst2.2 向 Inst3.1 迁移的边上，我们建立了一个通道 accessL1Cache[i]，其中 i 为处理器编号，程序自动机通过这个通道与 L1 Cache 行为自动机进行通信并达到同步的效果。其中 accessL1Cache[0]!表示当自动机离开 Inst2.1 节点的时候，立即通过 accessL1Cache[0]通道向 0 号核心对应的 L1 Cache 行为自动机发送消息。从 Inst2.2 到 Inst2.3 的边上有一个 accessL1Cache[0]?标记，这个标记表示从 accessL1Cache[0]通道中接收一个消息。当 L1 Cache 行为自动机接收到消息后，就会根据规定的 L1 Cache 行为进行相应的操作，完成后通过 accessL1Cache[0]通道向程序自动机发送消息，程序自动机成功接收到这个消息后才可以向下一个状态迁移。Inst2.3 节点和 Inst2.3 到 Inst3.1 的边对指令的执行时间进行了建模。本章研究的重点是 L2 Cache，所以在 5.2 小节中的假设 11 中，我们假定流水线是完美的，也就是每条指令的执行时间为 1 个时间单位，在这里就是所有指令的 InstExeTime 都为 1。在需要对指令执行时间进行精确建模的时候，仅需将 InstExeTime[i]数组对应的值设置为指令的实际执行时间即可。可见图 5.6 的建模方法是具有通用性的。

图 5.6(b)表示的是程序在 L1 Cache 上的访问为 AM 或 NC 的情况，在这两种情况下，都将产生对 L2 Cache 的访问。对 L2 Cache 的访问行为与图 5.6(a)的情况类似，区别主要是通过 accessL2Cache[0]与 L2 Cache 行为自动机进行同步。同时在 Inst2.1 向 Inst2.2 迁移的边上增加一个 processInstAddr[0] = 2 的动作，这一动作的含义是将程序的地址“2”赋给 processInstAddr[0]变量（这里 0 代表的是核心编号），当 L2 Cache 行为自动机进行 L2 Cache 操作的时候需要用这个地址来判断当前指令所在的 Cache 组，并将这个地址更新到 L2 Cache 中。

图 5.6(c)表示的是一条指令在 L1 Cache 中的访问为 FM 的情况，也就是该指令在 L1 Cache 中第一次访问为不命中，而后面的各次访问都是命中。在这种情况下，需要对访问的次数进行区分。对于所有的 L1 Cache 访问类型为 FM 的指令，我们建立了一个数组 firstMissFlag[i]来记录这一信息，其中 i 表示的是指令的编号。初始情况下，类型为 FM 的指令对应的 firstMissFlag[i]的值被设置为 1。以图 5.6(c)为例，当程序自动机第一次执行指令 2 的时候，将执行从 Inst2.1 到 Inst2.2 的上边那条边，执行过程中将产生对 L2 Cache 的访问，同时在执行完毕后，将 firstMissFlag[2]变量的值设置为 0，表示第一次已经执行完毕，其余各次都不会对 L2 Cache 产生访问。当下一次程序自动机执行到 Inst2.1 的时候，由于 FM 标志变量已经被设置为 0，因此执行从 Inst2.1 到 Inst2.2 的下边那条边，其行为是仅访问 L1 Cache。

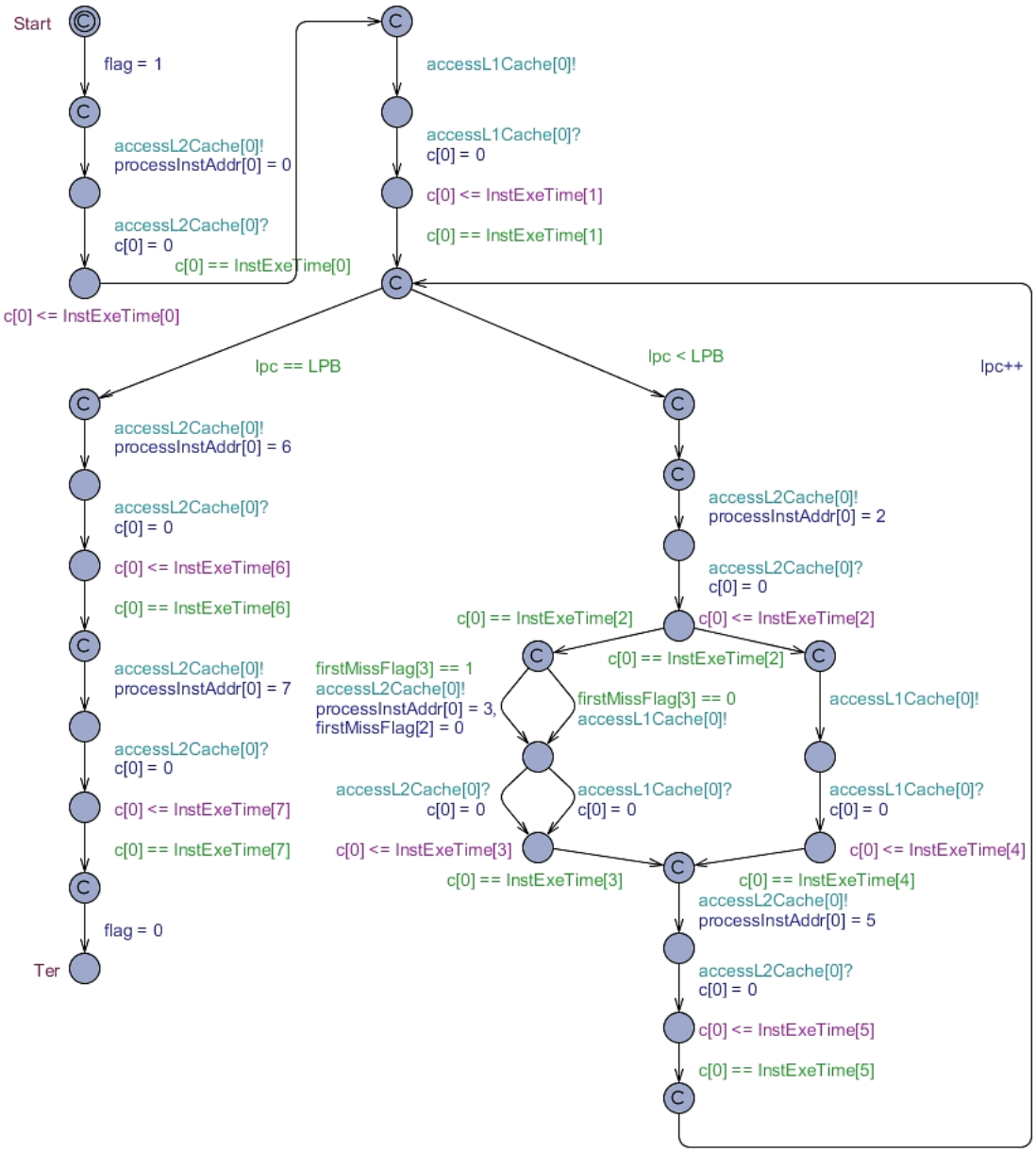


图 5.7 示例程序对应的程序自动机模型

Fig. 5.7 The program automaton for the demo program

有了上述对指令建模的方法，我们就可以把一个程序的程序自动机建立起来。图 5.7 表示的是图 5.5 的示例程序所对应的自动机。其中当进入循环体和从循环尾返回的时候，额外添加两个“committed”类型的节点，其主要目的是把对指令的执行和对循环次数控制变量的操作分离。如果是分析两个核心的多核处理器，那么需要分别为运行于两个核心上的程序进行建模。程序自动机的建模完成后，下面将介绍 UPPAAL 模型中的其它组件。

L1 Cache 行为自动机的工作原理如图 5.8 所示，它主要是用来建模程序在访问 L1 Cache 时的时间消耗，以及指令的执行时间。当该自动机从 accessL1Cache[processID]通

道收到消息后，将从初始节点转移到执行节点，同时将时钟的值置 0，用以在执行节点上记录时间。执行节点上的不变量 c[processID] <= L1AccessTime 以及从执行节点到初始节点的边上的守护条件 c[process] == L1AccessTime 用来实现让自动机在执行节点上严格停留 L1AccessTime 时间长度的语义。其中 L1AccessTime 表示的是指令在 L1 Cache 命中的情况下的 Cache 访问时间。在从执行节点到初始节点的边上还要完成一个动作，即通过 accesL1Cache[processID]通道向程序自动机发送消息，表明 L1 Cache 的访问已经完成，相应的时间也已经流逝。由于 L1 Cache 是各个核心独享的，所以每个核心对应一个 L1 Cache 行为自动机。由于 L1 自动机的行为较为简单，我们也可以把指令在 L1 Cache 上的执行时间直接建模到程序自动机中，以提高模型的运行效率，这里为 L1 Cache 单独建立自动机主要是为方便陈述问题。

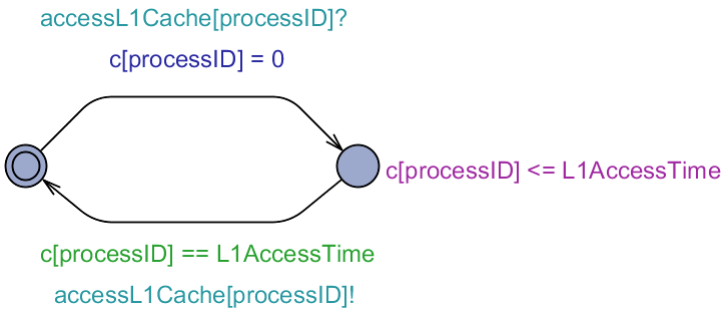


图 5.8 L1 Cache 行为自动机

Fig. 5.8 The automaton modeling L1 Cache

对 L2 Cache 进行建模，最关键的一点是需要考虑两个核心上的程序并行对 L2 Cache 进行访问的问题。例如在某一时刻，核心 1 和核心 2 上各有一条指令访问 L2 Cache，如果两条指令恰巧映射到同一个 Cache 组中，那么需要谨慎处理两条指令更新 Cache 的次序。在本章的模型中，如果出现上述情况，我们允许处理器选择任意一个核心上的指令先得到执行，模型检测器会在验证属性的过程中考虑所有可能的情况。此外，如果一条程序在 L2 Cache 中是命中，那么将瞬间更新 L2 Cache 的内容，之后完成一个延时，模拟数据从 L2 Cache 搬运到处理器核心的时间；如果一条指令在 L2 Cache 中是不命中，那么将首先完成一个延时，模拟从内存读取所需数据的时间，在此之后触发一个写操作，将从内存新加载的数据写入 L2 Cache，写操作同样也是瞬间完成的。根据上述需求，需要有一个自动机专门用于协调多个核心对对共享 L2 Cache 的访问，以及对 Cache 内容的更新。因此，L2 Cache 的行为被建模为两个层次：在第一个层次中，我们设计了 L2 Cache 行为自动机，该自动机主要建模 L2 Cache 访问的时间特性，每个核心拥有一个独立的 L2 Cache 行为自动机；在第二个层次中，我们设计了 L2 Cache 更新自动机，主要用来接受由上一层的 L2 Cache 行为自动机发送来的更新请求，并对 Cache 内容进行相

应的操作。L2 Cache 更新自动机的所有操作都不会导致时间流逝，它主要是把来自各个核心的并行更新请求串行化。

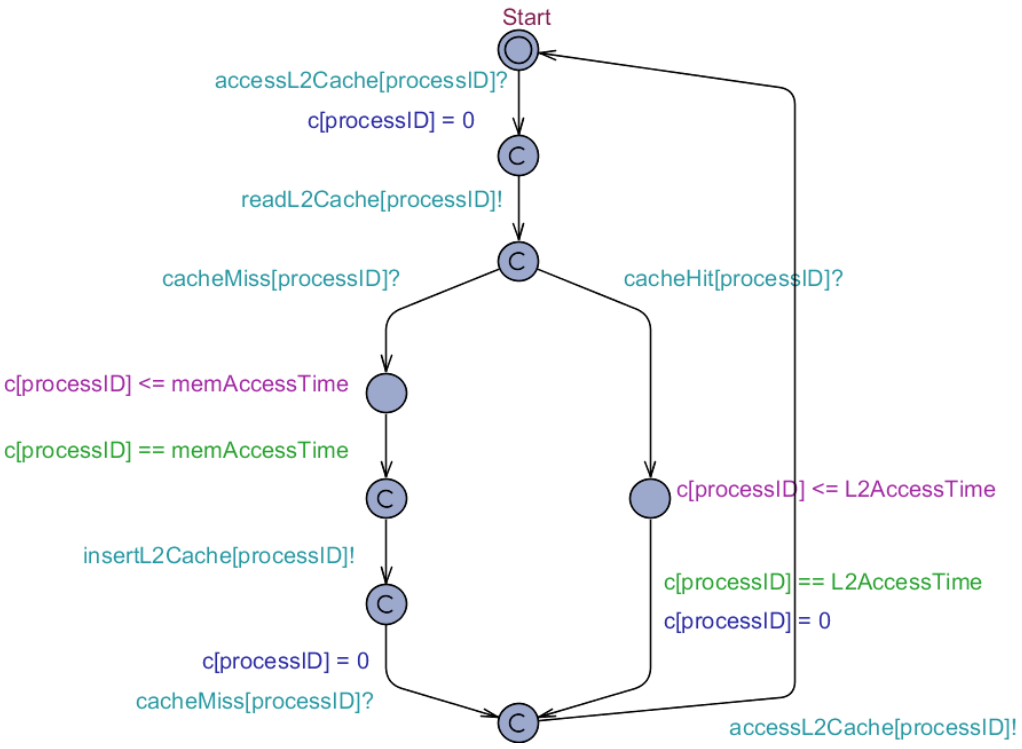


图 5.9 L2 Cache 行为自动机

Fig. 5.9 Automaton modeling L2 Cache

图 5.9 表示的程序访问 L2 Cache 的行为，即 L2 Cache 行为自动机。这一自动机在接收到程序自动机通过 accessL2Cache[processID]通道发送来的消息后启动执行。该自动机首先通过 readL2Cache[processID]通道发送消息给 L2 Cache 更新自动机（这一自动机将在后面介绍），后者通过查找 Cache 内容，得到本次访问是否命中的信息。如果指令在 L2 Cache 中命中，那么自动机将执行右边的路径，主要功能是延时 L2AccessTime 时间，用以模拟 L2 Cache 命中时的访问时间。如果指令在 L2 Cache 中不命中，将执行左边的路径，首先延时 memAccessTime 时间，模拟 L2 Cache 不命中后从内存读取数据的时间，之后通过 insertL2Cache[processID]通道向 L2 Cache 更新自动机发送消息，驱动该自动机将新的数据更新到 L2 Cache 中。上述操作完毕后，将通过 accessL2Cache[processID]通道向程序自动机发送消息，表明 L2 Cache 的访问已经完毕。

图 5.10 的自动机用来建模对 L2 Cache 的查找以及更新操作。当该自动机收到由 L2 Cache 行为自动机发送来的更新消息后被启动。该自动机根据给定的程序地址 processInstAddr[tempID]判断当前指令是否在 L2 Cache 中，这一操作是通过 isInL2Cache()函数完成的。如果当前指令命中，那么调用 updateL2Cache_LRU()函数对 Cache 内容进行更新，并通过 cacheHit[processID]通道向 L2 Cache 行为自动机发送消息；如果当前指

令不命中，那么将通过 cacheMiss[processID]通道向 L2 Cache 行为自动机发送消息，后者将等待 memAccessTime 时间，之后通过 insertL2Cache[processID]通道向 L2 Cache 更新自动机发送消息，完成 Cache 内容的更新操作。特别注意到，L2 Cache 更新自动机的所有动作都不消耗时间，即本模型中所有的 Cache 更新操作都在瞬间完成，这一假设是为了简化分析。在图 5.10 的自动机中，有两个函数需要特别说明一下：isInL2Cache(processInstAddr)函数主要是用于判定给定的指令地址是否存在于 Cache 中；updateL2Cache_LRU(processInstAddr)函数主要是完成对 Cache 状态的更新。本章主要基于 LRU 替换策略进行分析。如果需要对 FIFO，PLRU 等策略进行分析，只需修改 Cache 更新函数即可。

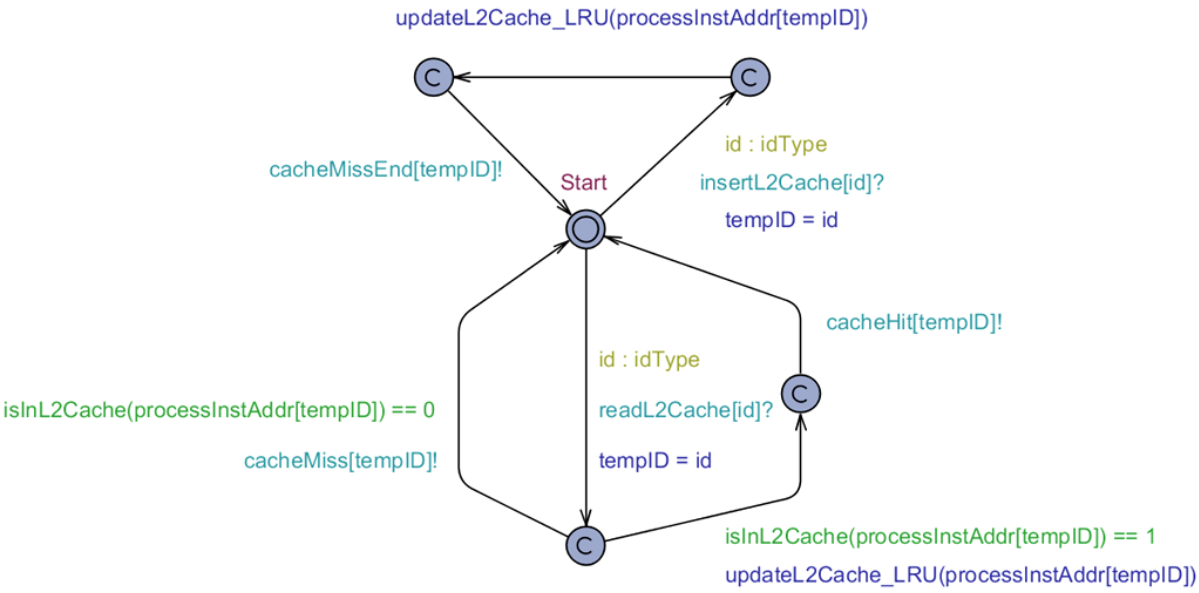


图 5.10 L2 Cache 更新自动机

Fig. 5.10 Automaton modeling L2 Cache update

以上介绍了采用 UPPAAL 对多核共享 L2 Cache 进行分析的全部内容。给定处理器核心的个数，为每个处理器生成对应的程序自动机、L1 Cache 行为自动机和 L2 Cache 行为自动机，并为所有核心生成一个共同的 L2 Cache 更新自动机，就得到了一个完整的分析模型。将这一组网络时间自动机交给 UPPAAL 进行验证，根据第 3 章所提出的二分搜索方法，就可以求出每个核心上的程序在考虑了共享 L2 Cache 上的干涉后的 WCET 值。

5.3.4 对基本模型的进一步优化

上一小节给出的是一种最基本的，同时也是最标准的多核共享 Cache 分析的建模方法，主要目的是为了展示本章所采用的主要分析思想以及分析框架。但是上述模型并非

是效率最高的，因此我们采用一些手段对 5.3.3 小节所提出的建模方法进行优化，力求达到降低模型的状态空间的目的。我们主要从两个角度简化原有模型：一是简化对指令的建模；二是简化 Cache 行为自动机的设计。下面给予详细介绍。

图 5.6 中访问 Cache 的延时和程序指令的延时是分开建模的。具体的说，我们各采用了一个 UPPAAL 的节点来描述上述两个延时，其结果是一条指令的执行将造成两次时钟重置。我们知道时钟变量是影响 UPPAAL 模型的状态空间的主要因素之一。在 UPPAAL 中，状态空间的大小与时钟重置的次数有着密切关系——时钟重置次数越多，造成对时间段的切分越多，对应生成的状态空间就越大。因此尽可能降低时钟重置的次数是缓解状态空间爆炸的方法之一。显然在图 5.6 的指令建模中，指令如果是访问 L1 命中，两次时钟重置就可以被简化为一次，此时的延时为 (L1AccessTime + InstExeTime[i])。

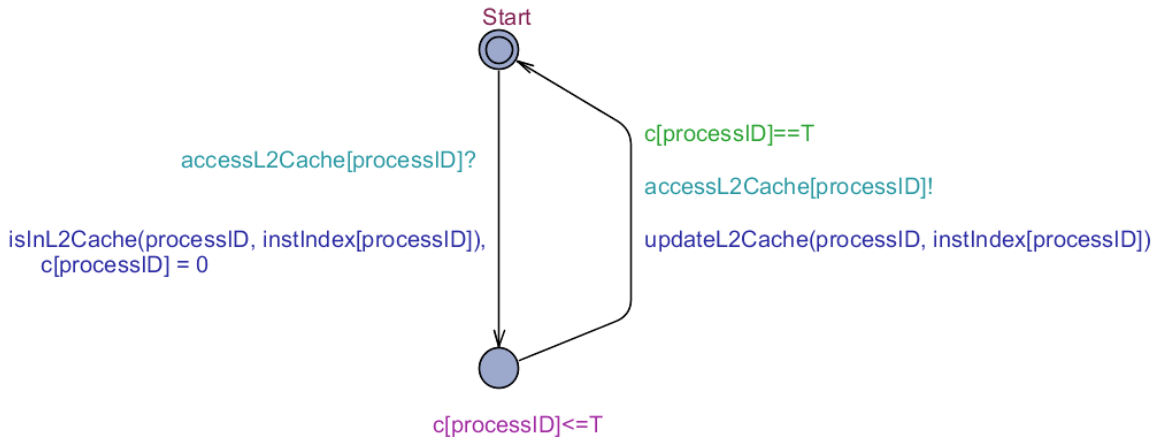


图 5.11 改进的 L2 Cache 行为自动机

Fig. 5.11 Imroved L2 Cache behavior model

UPPAAL 的模型如果被设计为多个相互关联的自动机，那么状态空间原则上就是各个自动机状态空间的乘积。因此删除或缩减不必要的自动机可以使状态空间减小。在我们的优化中，首先删除了 L1 Cache 行为自动机，当一条指令需要访问 L1 Cache 的时候，访问延时直接在程序自动机对应的位置被建模，对延时的处理方法已在上一段进行了解释。此外，在 L2 Cache 行为的设计中，L2 Cache 更新自动机的意义主要是防止多个自动机同时访问共享的全局变量时产生不一致，以及建模多个程序自动机同时访问 Cache 时的时间不确定性。通过对 UPPAAL 工具的深入学习，我们发现这个自动机的设计是不必要的，其功能完全可以放到 L2 Cache 行为自动机中。当两个程序的行为自动机同时访问保存 L2 Cache 内容的全局变量的时候，在 UPPAAL 的分析中是一定会被串行化的，且所有可能的次序都会被考虑到。依照这一思路，我们重新简化设计了 L2 Cache 访问的部分，新的 L2 Cache 行为自动机如图 5.11 所示。

5.4 实验结果与分析

本节首先介绍实验环境的建立和本章所采用的测试程序，之后介绍具体的实验结果，并从分析精度和分析效率两个角度对实验结果进行分析和评价。

5.4.1 实验方法与设置

本章实验的程序实现部分，主要是基于抽象解释的 L1 Cache 分析模块和基于模型检测技术的共享 L2 Cache 分析模块。本章仍旧使用 Chronos 工具所提供的反编译功能和 CFG 提取功能，在此基础之上，根据现有理论实现了一个基于抽象解释的 L1 Cache 分析器，其中包括了 5.3.2 小节对原分析理论的修改。基于模型检测技术的共享 L2 Cache 分析模块的本质是一个 UPPAAL 模型生成器。生成出来的模型交付 UPPAAL 模型检测器进行验证。寻找 WCET 值采用二分搜索。

表 5.1 测试程序

Table 5.1 Benchmark programs

程序名称	程序描述	指令条数	循环个数	循环嵌套层次
bs	冒泡排序	78	1	1
edn	FIR 过滤器程序	896	12	3
fdct	快速离散余弦变换函数	647	2	1
fir	FIR 过滤器程序	145	2	2
insertsort	插入排序	106	2	2
jfdctint	离散余弦变换	691	3	1
matmult	矩阵乘法运算程序	287	7	3

表 5.1 列出了本章实验所采用的测试程序。每个程序分别列出了编译后的总指令条数，循环个数，循环嵌套层数等反映程序复杂程度的参数。所有循环的上限值均设置为 16。大部分的测试程序选自第 3 章所采用的测试程序。在实验中我们模拟双核处理器上程序干涉的情况，所以上述 7 个程序两两干涉就有 21 种组合方式。对于每种组合我们将分别计算两个程序对应的 WCET。

7 个测试程序的平均指令个数为 407。为了测试我们的方法在程序高度冲突情况下的分析性能，我们设置了小容量的 Cache。同时为了和相关工作进行对比，实验中的 Cache 都设置为直接相联的。我们的分析模型是设计用来分析组相联 Cache 的，但是由于直接相联是组相联的一种特例，我们的分析程序不经额外修改就可以分析直接相联的 Cache。本章实验中，L1 Cache 的块大小为 16 字节，也就是一个 Cache 块可以放两条指令，组

的个数为 32，也就是说 L1 Cache 仅能存放 64 条指令。这个大小远远小于程序的平均指令个数，因此程序必将对 L2 Cache 产生大量的访问。L2 Cache 的块大小为 32，可以存放 4 条指令，组的个数为 128，相当于最大可存放 512 条指令。L1 Cache、L2 Cache 以及主存的访问延迟分别为 2、6、20 个时钟周期，7 个测试程序在内存中的分布从 bs 开始到 matmult 依次排列，每两个程序之间地址间隔 20 字节。由于起始地址不同，程序在 L2 Cache 中可能只是局部交叠的。但是由于本章的 Cache 设置的相对较小，因此仍然能够保证比较密集的程序间冲突。

UPPAAL 模型检测器运行于 IBM Blade Server 集群服务器，其中每个刀片配置有 2 个 4 核心的 Xeon 1.6GHz E5310 处理器以及 8GB 内存。由于 UPPAAL 模型检测器仅能在一个核心上运行，因此该平台的多核优势没有被利用。

5.4.2 实验结果与分析

我们假定，如果分析时间超过 36,000 秒则认定分析是时间不可行的，如果分析所需要的内存超过 8G 则认定分析是空间不可行的。在此规定下，本章实验成功的分析出了 7 个程序对应的 42 组实验中的 39 组。

对于每个程序，我们分别计算 4 种数据：（1）假设所有 L2 Cache 都命中的 WCET，这个作为模型检测的搜索下限；（2）假设所有 L2 Cache 都不命中的 WCET，这个作为模型检测的搜索上限；（3）文献[31]的分析方法得到的 WCET；（4）本章基于模型检测的分析方法得到的 WCET。其中（1）-（3）都采用了隐式路径枚举的 WCET 计算框架。实验的主要目的是对比文献[31]（这里称为“Yan 的分析方法”）与本章所设计的分析方法。除了作为搜索上限，计算情况（2）的另外一个目的是用于观测 Yan 的分析方法在程序密集冲突时的悲观程度。对于本章的分析方法，我们分别记录得到 WCET 所需要的分析时间和内存使用情况，同时罗列了进行二分搜索的步数。本章的分析方法在性能上与文献[31]的性能是无法比拟的，但是优势在于分析精度。下面我们将针对具体实验结果进行分析和评价。

我们首先评价 Yan 的分析方法的精确性。在 42 组实验中，有 19 组实验的分析结果已经完全退化为搜索上限的值，也就是假定 L2 Cache 全部是不命中的 WCET 值。这种分析精度显然是非常差的。造成这种结果的原因来自 Yan 的分析思想。在 Yan 的分析方法中，如果两个程序之间存在地址上的冲突，就会将对应的 L2 Cache 访问分析为在一定执行次数范围内的不命中；而且，如果地址冲突的指令位于程序的循环体内，那么这样的指令被分析为一定不命中。其中 insertsort 程序在和其他每个程序相互干涉的试验中都出现了结果退化的问题。这是因为，insertsort 程序的主体就是一个两层循环，因此

程序的执行时间主要就取决于这个循环体的执行时间。并且循环体指令数量比较少，所以循环体指令只占有少数的一些 Cache 块。当和其他具有较大循环体的程序相互干涉的时候，insertsort 的循环体所在的 Cache 块和另外一个程序循环体所需要占用的 Cache 块发生冲突的可能性就非常大。因此 Yan 的方法在分析 insertsort 程序的时候所有的结果都退化成了最悲观的情况。

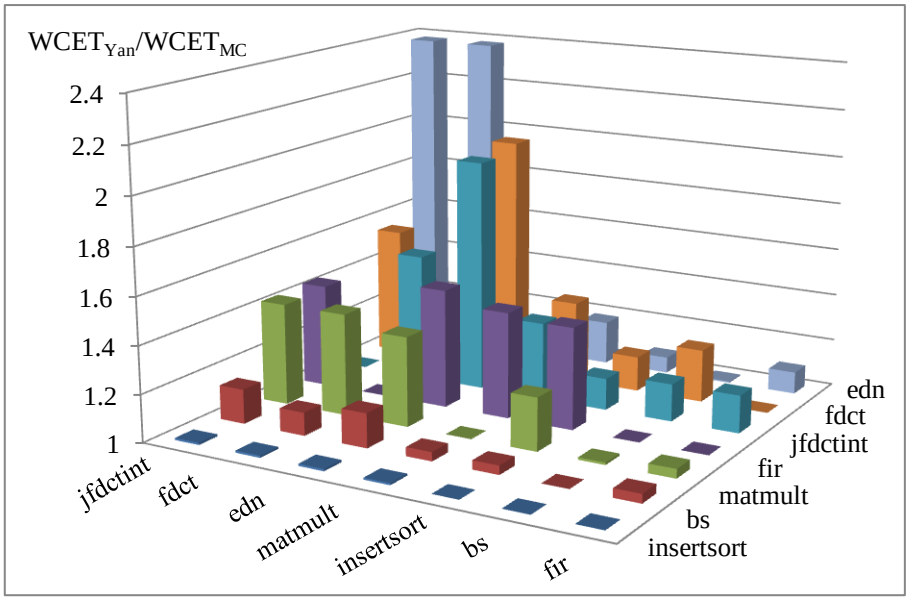


图 5.12 分析精度对比

Fig. 5.12 Comparison of analysis precision

相比之下，基于模型检测技术的分析方法在分析精度上就非常有优势。首先，在所有 Yan 的方法出现结果退化到最悲观情况的实验中，基于模型检测的分析方法都没有出现退化问题；其次，在 Yan 的方法未出现退化问题的实验中，基于模型检测的分析方法依然能够表现出更好的分析精度。图 5.12 描述的是 Yan 的分析方法和基于模型检测的方法的结果精度的对比，每组实验所对应的数据为 $WCET_{Yan}/WCET_{MC}$，其中 $WCET_{Yan}$ 表示 Yan 的方法得到的 WCET 值，$WCET_{MC}$ 表示本章的方法得到的 WCET 值。比值较大的数据反映的是上面所分析的 Yan 的方法出现结果退化的实验。同时需要特别指出的是，在本章实验的参数设置中，两级缓存的访问延时基本符合实际处理器的情况，而内存访问延时要比实际系统中的内存访问延时小的多，如果将这一参数设置为 100 或更大，那么基于模型检测技术的分析方法的精度优势将被放大。为避免这种放大效应，我们在实验中将内存访问延时设置的较小。

总的来说，Yan 的方法之所以分析精度低，主要是因为只考虑了干涉程序在地址上的潜在冲突，而没有探索潜在冲突的发生时间。而相比之下，由于基于模型检测的技术隐含的考虑了所有可能的执行情况，并完全的模拟程序在访问 Cache 时的行为，因此就

能够有效的发掘出哪些地址冲突由于在时间上错开而根本不会发生。这种时间上的错开有两种表现。第一种是全局的时间错开。例如两个相互干涉的程序 A 和 B，A 的执行时间为 1000，B 的执行时间为 10000，两个程序同时启动。那么在这种情况下，只有在前 1000 个时钟周期内，B 才可能受到来自 A 的干涉，而在后 9000 个时钟周期内，由于 A 已经执行完毕，B 根本不可能受到来自 A 的干涉。如果对程序间干涉的分析仅限于地址冲突，那就等于假定 A 对 B 的干涉将一直持续到 B 执行结束，这显然是不符合实际情况的。第二种是局部的时间错开。也就是说，A 程序的某个局部 A1 和 B 程序的某个局部 B2 在地址上存在若干冲突，但是在实际执行过程中，完全可能 A1 在 B2 开始执行之前就已完成，或者 B2 在 A1 开始执行之前已经完成。如果这种执行次序发生，那么 A1 和 B2 之间也不可能存在实际冲突。这些都是仅考虑地址冲突的分析方法所不能够判断的信息；而在基于模型检测的分析方法中，这些行为信息能够完全被捕捉到，并纳入 WCET 分析中。

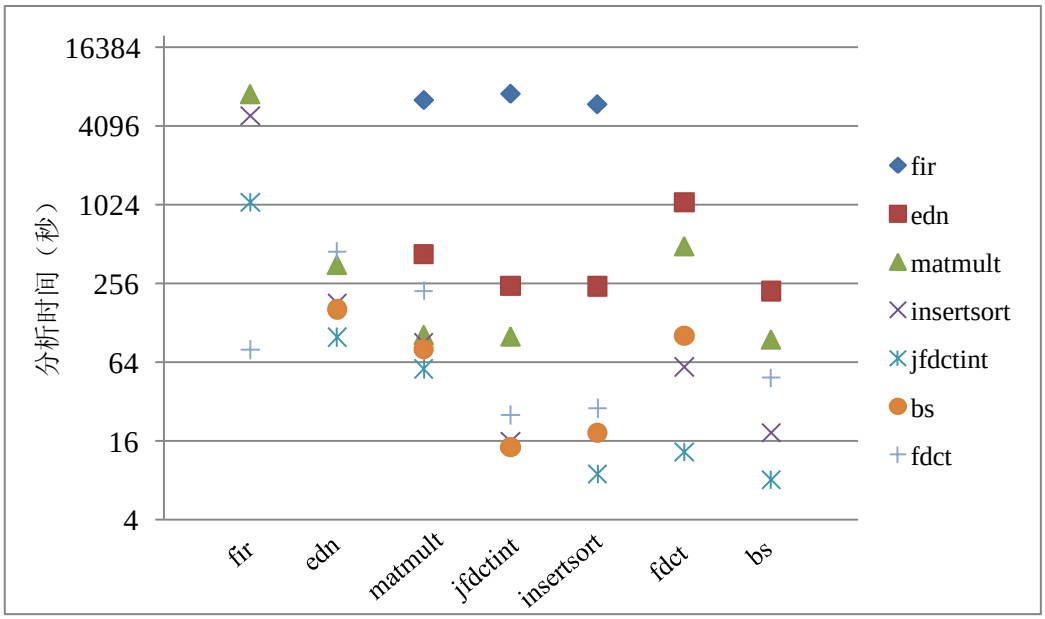


图 5.13 基于模型检测分析方法的分析时间

Fig. 5.13 Analysis time of the model checking based analysis method

下面我们将对基于模型检测技术的分析方法的效率问题进行讨论。图 5.13 和图 5.14 分别列举了本章所提出的方法在各组实验中所消耗的时间和内存，其中时间以秒为单位，内存使用量以 KB 为单位。从实验数据可以看出，在通常情况下，分析时间与内存消耗呈正相关。内存消耗越大，所需要搜索的状态空间就大，因此需要更多的分析时间。回顾部分测试程序在第 3 章实验中的时间和空间消耗不难发现，当两个程序相互干涉的时候，模型还是出现了状态空间爆炸的问题。例如，fdct 和 insertsort 两个程序，在第 3 章的实验中计算 WCET 需要的内存仅为 3MB，但是在考虑程序干涉的情况下，内存使用

则达到了平均 94MB 和 240MB。产生爆炸的原因是新模型的状态空间是两个干涉程序的状态空间的乘积。例如，假定 A 程序和 B 程序各有 100 条路径，也就是各有 100 种可能的执行情况，那么当两个程序发生干涉的时候，对于 A 程序的任何一条路径，B 程序都有 100 种干涉可能，因此理论上就有 10000 种可能的执行情况。但是需要指出的是，本章实验的目的之一也是探索在密集干涉下分析的可行性，因此设置了容量远远小于实际处理器配置的 Cache；随着 Cache 容量的增加，在 Cache 上发生冲突的可能性将会减少，因此如果基于实际的处理器参数进行分析，空间爆炸的程度要小于本章实验所反映出来的效果。

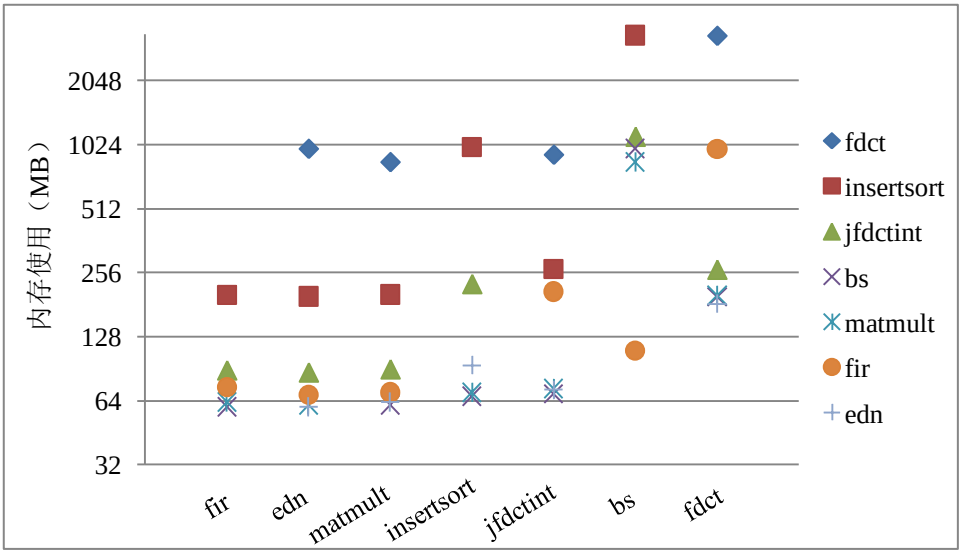


图 5.14 基于模型检测的分析方法的内存使用

Fig. 5.14 Memory usage of the model checking based analysis method

尽管不可避免的出现了状态空间爆炸，39 个得出结果的实验都能够在不超出系统内存的情况下完成分析，其中大多数实验的内存使用量在 1GB 以内，分析时间不超过 1000 秒，我们认为这一时间和存储空间的消耗是在可接受的范围内的。其中状态空间和分析时间爆炸现象比较严重的是 fir 程序，在该程序在和其他所有程序干涉的实验中都出现了这一现象。其根本原因是 fir 程序的循环体内分支数量过多，导致其程序路径数量也最多，于是在与其他程序相互干涉的时候造成了状态空间的急剧膨胀。通过对时间和空间消耗的分析，我们可以得到如下几个结论：

第一，状态空间的爆炸主要取决于相互干涉的程序各自路径的数量，对 Cache 行为的建模并没有显著的增加状态空间和分析时间，这一现象同时也符合我们在前面章节的讨论中所做的分析；

第二，程序指令数量本身并不直接影响状态空间，因此具有较多指令的程序与其他程序干涉时未必发生严重的状态空间爆炸。典型的例子就是 edn 和 jfdctint，这两个程序

是测试程序中指令条数最多的两个，但是程序结构相对简单，这两个程序与其他程序干涉的时候状态空间爆炸并不严重，分析能在较短的时间内完成；

第三，较大的循环体并不会造成状态空间的显著变化，但是对于 Yan 的方法却会造成分析精度的显著下降。典型的例子是 jfdctint 程序，该程序的三个循环体的指令个数都比较多，于是在 Yan 的分析中都退化为了最悲观的情况。具体原因我们已经在本小节一开始就给予了分析。而模型检测技术却不会因为大循环体的存在而产生分析精度下降的问题。

综合 39 组实验的结果，我们认为基于模型检测的方法可以应用于分析双核系统共享 Cache 的行为，从实验结果来看，只要程序循环体内结构不复杂，即使处理指令条数超过 1000 条的程序也不会产生状态空间的爆炸，分析时间和内存使用都在合理范围内，普通的桌面计算机的配置就足以完成对小型程序的分析。状态空间的爆炸主要还是取决于程序的路径数量，因此基于模型检测的分析方法目前尚不能处理所有类型的程序。在未来工作中，我们将继续探索如何在多核共享 Cache 的分析中引入剪枝思想，或类似的不可行路径分析方法，削减状态空间，达到提高分析方法可伸缩性的目的。

5.5 小结

本章首先探讨了多核共享 Cache 这一全新处理器体系结构给 WCET 分析带来的问题及挑战，通过分析相关工作存在的不足，提出了一种基于模型检测技术的多核共享 Cache 行为分析的方法。实验结果表明，由于模型检测技术能够探索指令间干涉发生的时间次序，因此能够有效的提高分析的精度。同时，我们从时间和内存消耗的角度分析评价了本章分析方法的可伸缩性。

第6章　实时操作系统中的 WCET 分析与应用

前面两章主要介绍了对应用程序进行 WCET 分析的相关技术。目前，这一领域的研究已经取得的相当数量的成果——在学术界已经拥有近 10 个分析工具，而在工业界，已经有 aiT、Bound-T、RapiTime 等工具应用于飞机和汽车等工业设计领域。但是随着实时系统复杂度的提高，目前几乎所有的实时系统都使用了实时操作系统来管理系统资源，调度任务运行。因此，实时系统的时间特性不仅仅取决于应用程序，同时也取决于所采用的实时操作系统。为获取整个系统的最坏情况执行时间，就必须对应用程序和实时操作系统程序都进行 WCET 分析。同时，了解系统最坏情况下的时间行为，也是实时操作系统设计过程中所必须重视的一个主要问题。

广义上讲，实时操作系统的可预测性主要包含以下几个方面：（1）系统的所有行为具有最坏情况的执行时间且可分析；（2）需要有软件机制避免应用程序执行过程中出现不可预测的阻塞延迟；（3）需要控制硬件行为以避免不可预测时间延迟的出现（例如存储器管理中的请求调页机制就是造成不可预测时间延迟的一个重要原因）。本章的研究重点集中在对实时操作系统时间特性的分析这一问题上，具体讲，就是对操作系统所提供的系统调用（System Call）的 WCET 进行分析。

除了以 API 方式提供的系统调用以外，在一个实时操作系统中，我们需要特别关注系统的“特定代码段”的时间特性。特定代码段的一个典型例子就是禁止中断区间（Disabled Interrupt Regions）。所谓禁止中断区间，指的是系统代码中的一些特殊代码段，在执行这些代码段的过程中需要关闭中断响应。例如，系统在访问互斥区的时候，需要读写共享的系统变量，此时需要禁止中断以防止中断服务程序打断原程序的执行并修改系统变量，而导致数据不一致现象的出现。禁止中断区间主要是用来保护对共享系统变量的访问。由于中断被关闭，在执行禁止中断区间的过程中，系统无法响应外部事件，这就降低了系统的响应性。实时操作系统的设计通常要求提供较好的实时响应特性，因此要求这些禁止中断区间的设计都尽可能短，这也成为了实时操作系统时间特性的一个重要指标。

传统的实时操作系统时间特性分析主要采用动态分析（也就是基于模拟执行）技术。如第 2 章所述，动态分析技术的一个主要问题是无法保证分析结果是安全的，这在硬实时系统中是不允许的。为此，需要使用静态分析技术对实时操作系统的系统调用和禁止中断区间进行分析，其结果可以帮助系统设计者发现系统执行时间与响应性的瓶颈，并完善系统的设计。

尽管在编译成机器指令后，应用程序和实时操作系统程序并没有显著的区别，但是由于实时操作系统的动态执行特性，导致采用静态分析技术分析实时操作系统要比分析应用程序复杂得多。简单的把为分析应用程序而设计的 WCET 分析技术应用于实时操作系统的分析将会导致较低的分析精度，甚至出现分析错误。下面举两个简单例子。操作系统代码中的循环上限的确定是主要难题之一，这是因为操作系统中的大多数循环上限值取决于任务的动态执行特性。例如，需要释放某种系统资源的系统调用执行的时候，可能有不确定数量的任务在等待该资源，那么根据系统语义，可能需要将所有等待任务唤醒。唤醒过程所花费的时间与正在等待的任务数量有关，而正在等待的任务数量是在系统启动执行前难以精确确定的。此外，对应用程序的 WCET 分析通常假设程序的执行是不可被打断的。但是实时操作系统会引入任务抢占，从而导致任务的执行经常被打断，其结果是当被打断任务恢复执行的时候，处理器状态已经被抢占任务所修改，这就导致了被打断任务后半段的执行时间发生变化，且具体变化难以预测。在可能出现时间异常的体系结构的处理器上，任务切换导致分析难度进一步加大。

如果缺少相应的分析手段，那么使用了实时操作系统的硬实时系统的时间特性就难以得到完整的分析和保证。从另一个角度，对系统时间行为分析的匮乏，会进一步限制实时系统在特定领域中的应用。对实时操作系统进行 WCET 分析，能够让开发人员确知系统的实时特性，结合对应用程序的分析，可以在开发阶段就得到整个系统实时特性的完整分析结果，因而确保系统的硬实时特性得到满足。

基于上述需求，本章对 μC/OS-II 实时操作系统的系统调用和禁止中断区间进行了完整的 WCET 分析。一方面，研究实时操作系统 WCET 分析对于探讨 WCET 分析理论在实际系统中的可用性问题[6]具有很大帮助；另一方面，具体的分析结果本身就是对 μC/OS-II 实时操作系统的实时特性的一个完整的量化描述，它对于使用者了解该系统的时间特性，以及对于系统开发者改善系统的实时特性都具有重要的应用价值。

本章后续章节将首先介绍实时操作系统 WCET 分析领域的相关工作以及 μC/OS-II 实时操作系统；之后介绍具体的实验方法和实验设置，其中主要是如何对禁止中断区间的提取和分析；最后，分析实验结果的精确性，并对尚存在的问题进行阐述，对于下一步工作需要继续深化研究的问题进行讨论。

6.1　相关工作

Colin 与 Puaut 等人采用 WCET 分析工具 Heptane 对 RTEMS[33]实时内核进行了分析，这是第一个采用静态分析技术分析实时操作系统的研究工作[34]。RTEMS 实时操作系统具有 85 个系统调用，但是他们仅对其中的 12 个系统调用进行了 WCET 分析，主要包

括任务调度系统调用和部分资源管理系统调用。Colin 等人的研究首次给出了采用静态分析技术分析实时操作系统的诸多问题，包括“动态函数调用中调用目标无法静态确定，循环上限的具体值与系统的动态特性相关，阻塞式的系统调用给 WCET 分析带来困难”等。此外，调度器本身的设计也造成分析的困难，这是因为 RTEMS 的调度器本身是一个循环体，而循环次数又与调度器执行过程中发生的中断次数有关系，这种动态特性导致对调度器执行时间分析非常困难。把上述问题综合到一起，最终分析结果的过度估计（Overestimation）达到了平均 86%，这在 WCET 分析中是非常不精确的。

Carlsson 与 Sandell 等学者对 OSE 实时内核[35]进行了分析，Carlsson 主要使用 WCET 分析工具 SWEET 完成了对该内核的禁止中断区间的分析[36,37]。OSE 实时内核包含了超过 1200 个的禁止中断区间，作者分析了其中的 612 个。实验表明 OSE 内核的禁止中断区间代码结构都比较简单，但是分析结果精度不好，主要是因为缺少对 Cache 的分析，以及设置循环上限难度较大。Sandell 主要使用 WCET 分析工具 aiT 完成了对该内核的系统调用的分析[38,39]。Sandell 主要遇到了三类问题，包括“系统调用的执行时间严重受制于系统参数，循环上限值与系统的动态执行特性有关，系统在处于不同模式的时候，执行时间存在巨大差异”等。在整个分析的过程中，需要用户的大量介入，因此分析的自动化程度非常低。Sandell 提出单一的 WCET 分析值已经不足以刻画实时操作系统的时间行为，但是并没有提出任何解决办法。文献[119]中总结了将 WCET 分析技术用于工业应用中所遇到的主要问题，其中一些问题与 Sandell 的结论是一致的。

Petters 及其研究小组对 L4 实时内核[40]进行了 WCET 分析，旨在探索对实时操作系统进行 WCET 分析的自动化程度如何[41]。他们所采用的工具是一个混合 WCET 分析工具，路径分析采用基于语法树的技术，而对于每个基本块的执行时间采用测量的方法获得[120]。Petters 所遇到的第一类问题来自程序代码的结构。由于采用的工具在路径分析上采用了基于语法树的技术，因此无法处理 L4 实时内核中大量出现的 break 和 goto 等非结构化的程序结构。第二类问题来自由于编译器优化造成的“索引跳转（Indexed Jumping）”，作者通过设计一个跟踪器（Tracer）来分析跳转地址。第三类问题是动态函数调用，这一问题在 Colin 的工作中同样出现。Petters 指出，目前对实时操作系统进行分析，要求分析者对于 WCET 分析工具和待分析的实时操作系统都具有深入的了解——这实际上说明实时操作系统 WCET 分析的自动化程度还较低。

Puschner 等人从另外一个完全不同的角度解决实时操作系统时间可预测性的问题[121]。其基本思想是，通过各种可能的手段，在硬件、系统软件、应用软件的设计中尽量避免那些可能导致执行时间不可预测的特性，从而使得整个系统的时间行为几乎完全可预测。同时，专门的软硬件设计大大减少了 WCET 分析的难度和工作量。主要采用

的技术手段包括：（1）采用“单路径编程（Single-Path Programming）”技术[122]编写程序以减少任务代码内部的不可预测性；（2）强制每个处理核心上只运行一个任务以减少任务间干涉；（3）使用简单硬件（如顺序流水线等）降低程序执行的状态空间；（4）采用静态调度的方法管理任务对共享资源的访问。Khyo 和 Puschner 等人设计了一个系统，其中硬件和操作系统都采用上述设计思想 [123,124]。但是按这种思想设计的系统，其灵活性较差，因此不适用于那些动态性较强的系统。

表 6.1 实时操作系统 WCET 分析相关工作对比

Table 6.1 Comparison of related work on WCET analysis of RTOS

分析者	Colin	Sandell	Petters	Puschner
被分析的实时操作系统	RTEMS	OSE	L4	
分析工具	HEPTANE	aiT/SWEET	Petter’s Tool	
过度估计平均值	86%	n/a	n/a	
因程序结构特性造成的问题				
不可规约的程序结构	P2		P2	
索引跳转			S	
因缺乏应用程序信息而导致的问题				
循环上限的设置	P2,3	P2, P3	N	
动态函数调用	P4		P4	
带阻塞语义的系统调用	N			
缺乏系统调用上下文信息				
缺乏操作系统运行模式信息		P2		
因任务切换和任务间干涉造成的问题				
任务切换造成的问题				P4
任务抢占造成的时间异常				P4
任务切换造成的执行时间变化	N	N		P4
多核任务间干涉			N	N

我们对上面的相关工作进行简要的总结。所有的问题可以分为三大类：由于程序结构造成的问题、由于缺乏应用程序信息造成的问题、以及由于任务切换和任务间干涉造成的问题。由于程序结构造成的问题主要和采用的 WCET 路径分析技术有关，实际上目前多数技术都能够处理非结构化的程序，因此这不是一个关键问题。缺乏应用程序信息将会导致实时操作系统 WCET 分析结果精度较差。设置循环上限就是最典型的一个例子。动态函数调用和带有阻塞语义的系统调用仍旧是当前 WCET 研究所面临的主要

问题。如果能够找到一些办法从应用程序中提取一些信息，将动态函数调用目标等在执行之前就确定下来或给定一个范围，那么就可以开展对动态函数调用的分析，同时也会提高分析的精确性。具体的总结结果见表 6.1。其中，S 表示该问题已经得到解决，N 表示该问题在相关研究中被回避，因此尚未解决；P 表示该问题部分被解决，但仍需要继续研究。其中 P1 表示该问题可能存在可伸缩性问题，P2 表示该问题的解决需要用户大量介入，P3 表示该问题目前得到的结果精确性很差，P4 表示采用的解决办法限制性过大。

6.2　μC/OS-II 实时操作系统简介

μC/OS-II 实时操作系统[125,126]是由 Micriμm 有限公司开发的开源实时系统。该实时操作系统使用 C 代码编写，并完全符合 ANSI C 标准。μC/OS-II 体积小巧，且可以由用户自由配置，编译后的体积为 5KB 到 24KB，非常适合资源受限的实时嵌入式系统使用。虽然 μC/OS-II 内核不大，但是却具备了与 RTEMS、VxWorks 等大型实时操作系统类似的几乎所有特性，包括：基于优先级可抢占的实时调度；（2）基于信号量、互斥信号量、消息队列、以及消息邮箱等机制的任务间通信功能；（3）基于优先级继承协议的资源共享管理；（4）时间管理；（5）简单的内存管理等。

μC/OS-II 是目前工业界使用最广泛的实时操作系统之一。目前该系统已经被移植到 22 家微处理器公司的超过 50 种的微处理器上，几乎覆盖了嵌入式设计领域的所有主流处理器。同时该系统已经授权给超过 100 家的嵌入式系统开发公司，产品覆盖了包括航空、医药设备、数字通信设备、白色家电、手机与个人数字助理、工业监控设备、汽车电子等众多应用领域。μC/OS-II 已经通过了美国联邦航空管理局（FAA）的认证并可以适用于商用航空领域；同时该实时操作系统符合 SIL3/SIL4 IEC 协议，可用于运输和核工业领域。

如上所述，μC/OS-II 基本具备了一个实时操作系统所应具备的所有功能，并且广泛应用于重要的工业及民用领域；但是另一方面，目前尚无对于该实时操作系统的实时特性的量化分析。因此，本章选择 μC/OS-II 实时操作系统作为分析对象。μC/OS-II 所具备的丰富功能有助于探索采用静态 WCET 分析技术分析操作系统代码的可行性及存在的主要问题；μC/OS-II 的广泛应用则使得对该系统的实时特性的量化分析结果具有很好的实际应用价值。

6.3　μC/OS-II 禁止中断区间 WCET 分析

本节首先介绍禁止中断区间的分析框架，其次详细介绍了分析框架中核心功能模块

“禁止中断区间提取模块”的详细设计。

6.3.1 禁止中断区间分析框架

禁止中断区间是操作系统代码中的一些特定片段，在其执行的过程中要求系统中断关闭。禁止中断区间通常位于系统调用代码中，以中断禁止指令/函数开始，以中断恢复指令/函数结束。图 6.2 就是 μC/OS-II 的一个禁止中断区间的例子。大多数的 WCET 分析工具往往只能分析完整程序，而不能处理程序片段。本章所采用的分析工具 Chronos 也不例外。Chronos 所能够接受的输入是由程序的可执行代码转化而来的 TCFG。这里，TCFG 是一个单体的控制流程图，它是通过将所有子函数的 CFG 根据函数调用关系组合在一起而生成的。为此，我们需要从待分析程序的 TCFG 中，将禁止中断区间所对应的部分提取出来。禁止中断区间的 CFG 实际上就是待分析程序的 TCFG 的一个子图，我们称之为 Sub-TCFG。Sub-TCFG 提取出来之后，我们就可以对其进行 WCET 分析。

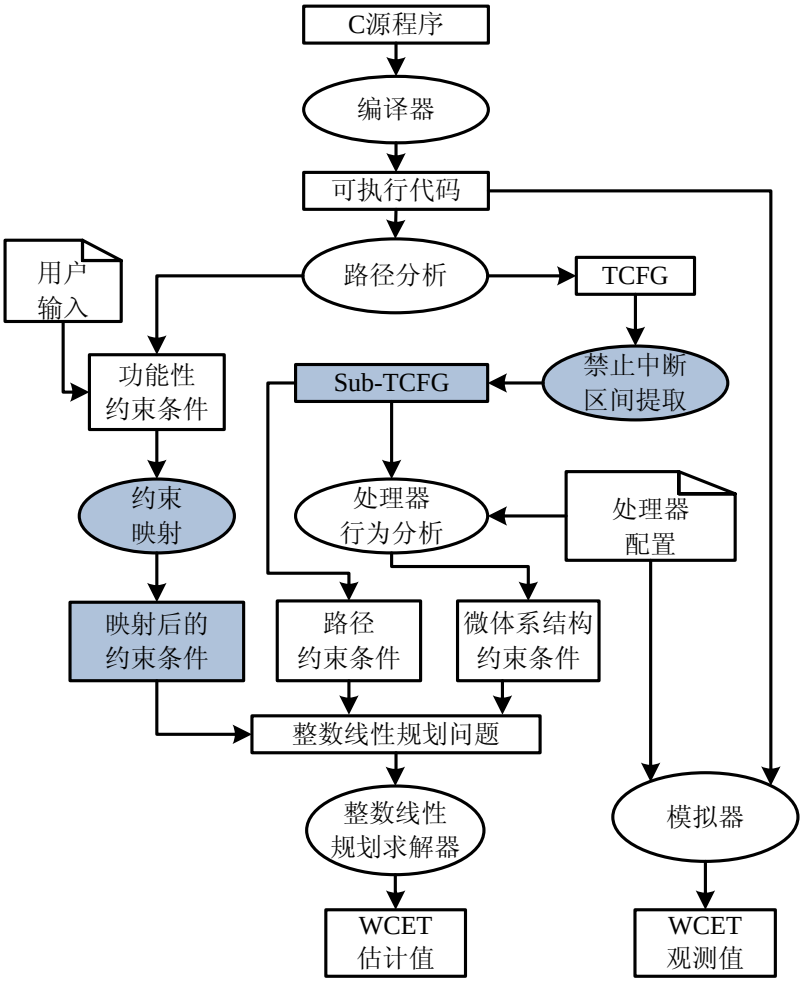


图 6.1 改进后的 Chronos 工作流程

Fig. 6.1 The enhanced Chronos work flow

图 6.1 是改进后的 Chronos 工作流程，增加的功能用阴影表示。首先我们添加了禁止中断区间的提取功能。原工具所生成的整个待分析程序的 TCFG 就是禁止中断区间提取功能的输入，通过识别禁止中断区间的起止标记，我们能够从 TCFG 中提取出对应的禁止中断区间的 Sub-TCFG，Sub-TCFG 以和 TCFG 相同的数据结构表示，因此能够被 Chronos 的其他分析步骤所接受。一方面，工具从 Sub-TCFG 中提取控制流程信息，生成对应的控制流程约束条件；另一方面，Sub-TCFG 的基本块的执行时间通过原有的处理器行为分析功能分析得到。禁止中断区间内部也可能包含有循环结构，因此在计算禁止中断区间的 WCET 时同样需要知道循环上限。原分析工具已经提供了循环上限的输入功能，需要改进的就是把用户输入的功能性约束映射到 Sub-TCFG 上，这一功能是通过“约束映射模块”实现的。下面，我们将进一步介绍提取 Sub-TCFG 的具体实现。

6.3.2 禁止中断区间的提取

为提取 Sub-TCFG，首先要完成识别功能。在 μC/OS-II 中，禁止中断区间的入口和出口分别是两个宏定义，而其具体实现是两个函数：

```
#define  OS_ENTER_CRITICAL()  {cpu_sr = OSCPUSaveSR();}     // 入口
#define  OS_EXIT_CRITICAL()   {OSCPURestoreSR(cpu_sr);}     // 出口
```

在 Chronos 工具中，函数调用都将导致一个新的基本块的产生。因此，我们仅需要跟踪 OSCPUSaveSR()函数和 OSCPURestoreSR()函数在 TCFG 中出现的位置，即可确定禁止中断区间的边界。

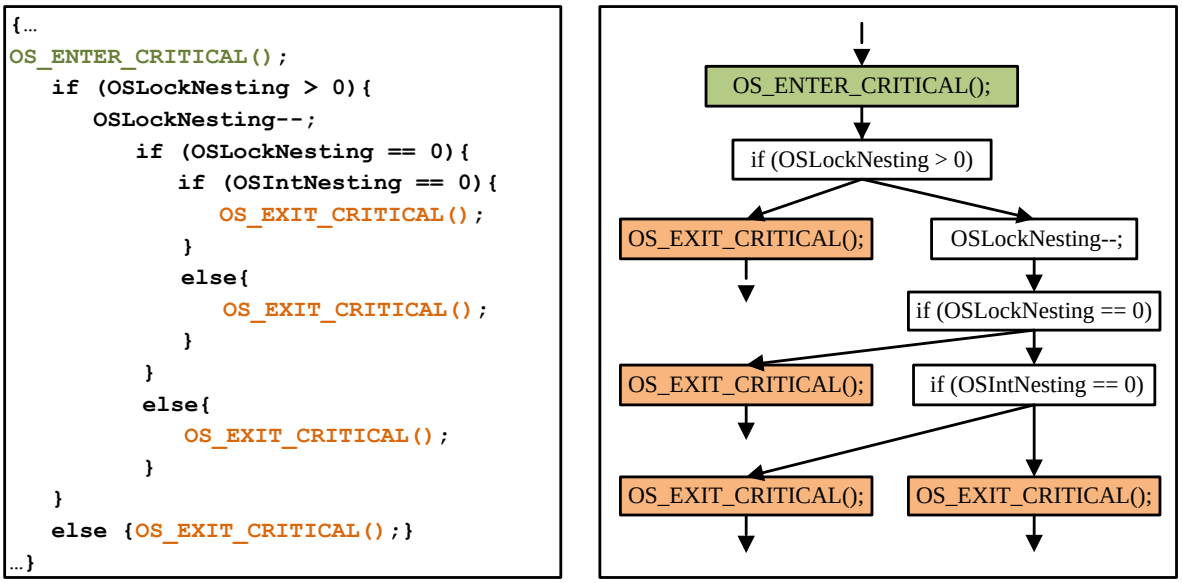


图 6.2 多出口禁止中断区间示例

Fig. 6.2 An example of multi-exit disabled interrupt region

分支和循环结构在禁止中断区间中是比较常见的，因此在提取的过程中都必须进行相应的处理。μC/OS-II 的禁止中断区间普遍存在的一个现象就是“单入口，多出口”。如图 6.2 所示，当程序关闭中断之后，需要根据系统状态的不同执行不同的程序分支。但是不管执行哪个分支，在对应功能完成后，都应将中断重新开启，这就造成了“单入口、多出口”的结构。

注意到在我们所分析的程序中，所有的分支结构都只有两条可能路径，因此我们采用一个递归算法对禁止中断区间进行提取。图 6.3 是该算法的基本流程。从禁止中断区间的入口节点开始，我们根据待分析程序的 TCFG 的控制流程结构，不断沿可行路径遍历，直至遇到出口节点，或遇到已经遍历过的节点。由于遍历过程是按照图中的有向路径执行的，因此当禁止中断区间存在循环的时候，会陷入死循环。因此，我们每遍历一个节点，都会对其进行标记。当程序终止的时候，所有属于该 Sub-TCFG 的节点都已经存放于集合 S_{node} 中。再将原 TCFG 中的那些起点和终点都在 S_{node} 中的边找出来，与 S_{node} 中的点就可以组合成该禁止中断区间对应的 Sub-TCFG。

算法6.1　根据禁止中断区间的入口结点提取对应Sub-TCFG

函数名称：*void lookup_node (**node****)***
输入：（1）待分析程序的TCFG；（2）入口结点
输出：该入口结点对应的Sub-TCFG

```
if ( node已经在结点集合S_node中 )
    return;
else
    将结点node放入结点集合S_node;
if ( node == 出口结点 )
    return;
if ( node->out == NULL )
    return;
else {
    lookup_node (node->out);
    if ( node->next_out != NULL )
        lookup_node (node->next_out);
}
return;
```

图 6.3 提取一个禁止中断区间的递归程序

Fig. 6.3 The recursive algorithm for extracting a disabled interrupt region

以上是提取一个禁止中断区间的递归程序。图 6.4 是提取所有禁止中断区间的算法。尽管一个 μC/OS-II 中可能含有多个禁止中断区间，但是所有的禁止中断区间都不存在嵌套的情况。因此提取程序中所有的禁止中断区间，仅需从每一个入口节点开始，各执行一次图 6.4 中的算法即可。

算法6.2　提取一个TCFG中的所有禁止中断区间

输入：待分析程序的TCFG
输出：所有禁止中断区间对应的Sub-TCFG的集合—S

扫描TCFG，识别所有的禁止中断区间入口，将每一个入口e_i放入集合$S_{entries}$;
while ($S_{entries}$不为空)
　取得一个入口结点e_i;
　遍历TCFG，调用***lookup_node()***找到该入口对应的禁止中断区间的Sub-TCFG的结点集合S_{node};
　收集所有跟Sub-TCFG的结点相关的边;
　if (该Sub-TCFG的出口数 > 1)
　　增加*dummy*结点用以合并出口结点;
　end if
　把上面提取出来的点和边构造成Sub-TCFG放入集合S;
end while
return S

图 6.4 提取所有禁止中断区间的算法
Fig. 6.4 The algorithm to extract all disabled interrupt regions

约束映射模块完成循环上限等功能性约束的映射功能。通过 Chronos 原有的机制，用户可以手工输入循环上限信息。循环上限控制约束的形式如第 3 章的公式 3.3 所示，主要是通过约束循环体执行次数和循环体入口边的执行次数的关系来表达的。由于在上述遍历过程中，遍历到的节点在原 TCFG 中的信息都得以保留，所以我们很容易根据已知的约束信息，找到对应的节点在 Sub-TCFG 中的编号，并根据基本块编号的对应关系，生成新的功能性约束条件。其他类型的功能性约束条件可以采用类似的方法进行映射。

6.4　实验方法及实验设置

本节首先介绍实验环境的设置，主要是分析工具参数的设置及采取这种设置的理由；其次介绍了一种简单直观的获取系统调用的 WCET 的方法，并说明了这种方法的正确性。这一方法同样应用于禁止中断区间的实验中。

6.4.1 分析工具的配置

本章主要采用 Chronos 作为主要分析工具。Chronos 工具的一大优势就是允许用户定制目标体系结构的各种特性，如流水线参数、Cache 的大小等参数、分支预测的策略等。实验中无论设置何种体系结构配置，Chronos 内建的分析方法能自动进行相应的调整，从而实现对多种不同体系结构的 WCET 分析。SimpleScalar 模拟器也能够根据配置的修改而自行调整。

表 6.2 Chronos 配置对分析精度的影响

Table 6.2 The effects on analysis precision due to Chronos configurations

超标量	Cache 块大小	访存时间	WCET	模拟时间	过度估计
1	8	10	2,624	2,232	17.56%
2	8	10	2,639	2,044	29.11%
4	8	10	2,639	2,040	29.36%
1	8	10	2,624	2,232	17.56%
1	32	10	976	664	46.99%
1	64	10	726	390	86.15%

我们在实验的过程中发现，Chronos 分析结果的精确性与体系结构有很大的关系。表 6.2 是本文对系统调用在不同系统配置下进行 WCET 分析的结果，通过实验数据可以看出，如果设置较大的流水线超标量、较大的 Cache 块大小等都将严重影响结果的精确性；相反的，流水线的指令预取队列长度、流水线 ROB 大小、访存时间、以及 Cache 的总数及相联度都不会对分析结果的精确性造成显著的影响。由于本课题的一个主要目标是探索实时系统的独特性是否对 WCET 分析的精确性产生影响，所以为便于区分不精确性产生的原因，应该让分析工具本身所导致的不精确性尽量减到最小。根据这一要求，最终的体系结构参数配置如表 6.3 所列。此外，由于技术问题，Chronos 不可行路径分析功能在本章的分析中被关闭。

表 6.3 Chnoros 工具的配置

Table 6.3 The configurations of the Chronos tool

处理器特性	参数值
Superscalarity	1
Instruction Fetch Queue Size	4
Reorder Buffer Size	8
The Number of Cache Sets	64
Cache Block Size	8
Cache Associativity	8
Main Memony Access Latency	30
Branch History Table Size	16
Branch History Register Width	1

6.4.2 实验方法

对系统调用进行实际分析时的一个主要问题是，目前版本的 Chronos 无法对一个程

序中的某个函数进行单独的分析，它只能分析程序主函数 main()所包含的部分，因为只有存在 main()函数的完整程序才能够顺利通过编译，并被模拟器执行。因此，在分析每一个系统调用的时候，本文必须将其封装在一个 main()函数中才能够顺利完成分析。封装后的程序如图 6.5(a)所示。

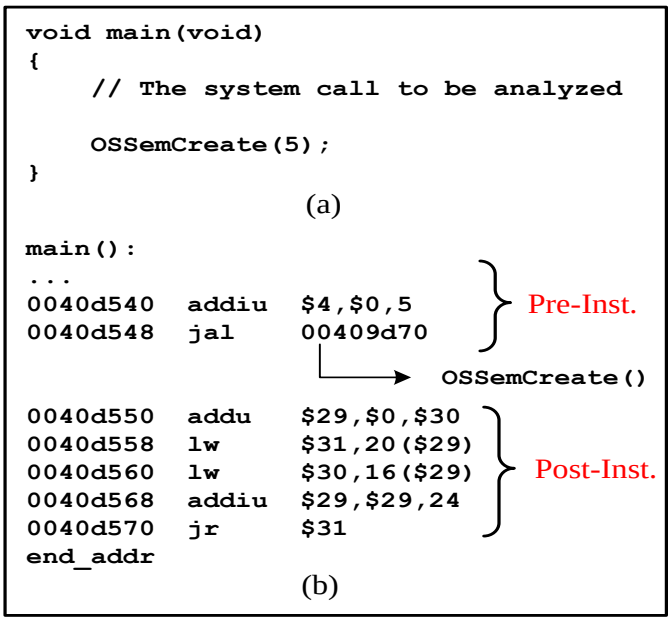


图 6.5 main()函数的封装程序

Fig. 6.5 The wrapper main() function

但是这种办法本身也存在问题。即使一个main()函数中仅仅包含一个系统调用，编译后的可执行代码中，仍然会有少量指令出现在系统调用的前面以及后面。而系统调用的WCET应该是整个main()函数的WCET减去这些额外代码的执行时间，这样得到的才是系统调用本身的WCET。可以看出，出现在系统调用前后的代码不包含任何分支和循环，因此它们仅执行一次，这些指令在Cache中将会产生“首次访问失效（cold miss）”。所以这些额外指令引入的执行时间包括了这些指令本身的执行时间和由于执行这些指令所引入的Cache失效时间。从上一小节介绍的系统配置可知，流水线的Superscalarity被设置为1，因此每个时钟周期最多只可能有一条指令被提交，且提交指令的次序与指令出现在程序中的次序相同。同时注意到，不管什么类型的指令，每条指令的执行时间最小值为1。因此对于系统调用之前和之后的每一条指令，本文都对应减去一个时钟周期以及一个Cache失效的周期数，所得到的结果作为系统调用的WCET，依然是安全的。计算系统调用的WCET的方法如公式6.1所示，其中N_{pre}和N_{post}分别表示系统调用之前和之后的指令的条数。

$$Calculated\_WCET = Estimated\_WCET - (Cache\_miss\_penalty + 1) \times (N_{pre} + N_{post}) \quad (6.1)$$

同时注意到，在真实系统中，这些系统调用能够得到正常执行的前提条件是它们所需要的数据结构在系统启动的时候或调用这些系统调用之前已经成功建立。由于本文的上述方法，导致系统调用在main()函数中是独立出现的，为使程序能够正常在模拟器中得到执行，本文还要为系统调用准备出它所需要的运行环境，也就是在调用之前手工创建一些系统调用所需要的数据结构。这些额外的环境将进一步引入额外的指令，对这些指令的处理方法是类似的。

6.5　实验结果与分析

这一节主要对实验结果进行分析和评价。首先评价主流静态 WCET 分析技术用于分析实时操作系统时的精确性如何，其次评价对 μC/OS-II 的禁止中断区间的分析结果。

6.5.1 对系统调用分析精度的评价

μC/OS-II 内核共设计有 79 个系统调用，本文实验成功得到了其中 61 个系统调用的 WCET 值。其中 9 个系统调用属于非关键功能的系统调用，直接排除；另外还有 9 个系统调用因工具问题未能成功分析。μC/OS-II 的时钟管理模块（Timer Management）的 8 个系统调用都未能成功分析，主要原因是这几个系统调用的代码中含有动态函数调用（以函数指针的形式实现）。从可执行代码的角度看，动态函数调用位置的跳转指令所需要的跳转地址不是一个固定值，而是存放在某个寄存器中的值，且其具体值要根据系统的运行状态具体确定。本文所采用的 WCET 分析工具 Chronos 采用的是静态 WCET 分析技术，它要求所有被调用函数的目标地址是静态确定的。任务管理模块的OSTaskCreateExt()系统调用也未能被成功分析，主要原因是这个函数无法通过 simprofile。Simprofile 是 SimpleScalar 模拟器的一个关键模块，其主要功能是为待执行的程序生成信息文件，它是程序模拟执行之前必不可少的一个步骤。

所有 μC/OS-II 系统调用的 WCET 分析实验数据列于附录 A。附录 A 为每个系统调用分别列出了 WCET 估计值，SimpleScalar 模拟执行的观测值，以及由此计算出来的过度估计值。整体分析结果表明，对于 61 个系统调用的分析，平均的过度估计值为 17.57%。虽然这一估计值比起分析应用程序的平均情况要差一些，但是基本上还是在可接受的范围内。本文可以分几种情况对这一结果进行分析解释。

首先，大约一半的系统调用其过度估计的值不超过 5%。这是由于这些系统调用的程序结构都非常简单，基本没有循环且分支个数较少。对于这种简单程序，Chronos 工具能够较好的处理，因此过度估计值比较小。第二类系统调用是那些用来获得共享资源的系统调用，例如事件组管理模块的 OSFlagPend()、信号量管理模块的 OSSemPend()等。

这些系统调用的过度估计值较大，是造成平均过度估计偏高的主要因素。通过对分析结果的详细分析，本文认为造成这一结果有两个原因：一方面，这些系统调用的共性就是含有大量的分支结构和函数调用，在这种情况下，模拟器按照程序的最坏情况路径执行比较困难，因此过度估计不完全由分析工具产生，其中一部分来自执行路径问题；另一方面，分析工具确实存在不精确的问题，本文在实验中发现分析工具给出的 Cache 失效的个数大于模拟器执行得到的 Cache 失效次数。

```
1   // some code here …
2   OSTimeDly ( (INT16U) ticks );
3   while ( loops > 0 ) {
4         OSTimeDly ( (INT16U) 32768U );
5         OSTimeDly ( (INT16U) 32768U );
6   }
7   // some code here …
```

图 6.6 OSTimeDly()函数部分代码

Fig. 6.6 code segment of OSTimeDly()

实验中还出现了一个特例：OSTimeDlyHMSM()系统调用的过度估计值高达 248.94%。分析发现，造成这一结果的原因来自于图 6.6 所示的代码段。在模拟器模拟执行的过程中，第 2 行调用 OSTimeDly()函数的执行出现的是 Cache 失效，但是由于这次执行已经将这个函数的代码调入 Cache，因此在第 4、5 两行再次调用这个函数的时候 Cache 命中。但是在分析端，Chronos 对于第 2、4、5 三行的 OSTimeDly()函数的执行都分析出是 Cache 失效，对于后两行函数的错误分析是导致如此大的过度估计的根本原因。

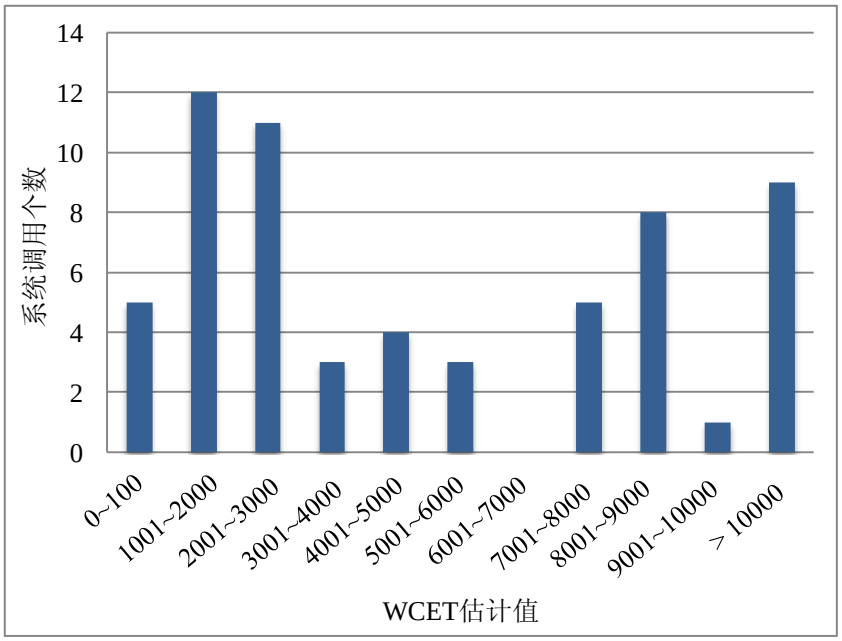


图 6.7 61 个系统调用 WCET 值分布情况

Fig. 6.7 The distribution of the WCET of 61 system calls

图 6.7 是所有被分析的 61 个系统调用 WCET 值。该图粗略的反应了这些系统调用

的最坏情况执行时间的特征。首先，本文发现有三分之一的系统调用，其执行时间在 3000 个时钟周期以内。这说明相当一部分的系统调用都能在较短的时间内执行完成。同时也注意到，有 9 个系统调用的执行时间在 10,000 个时钟周期以上，这些系统调用相当于是整个 µC/OS-II 内核部分的性能瓶颈，需要对其进行进一步的分析。

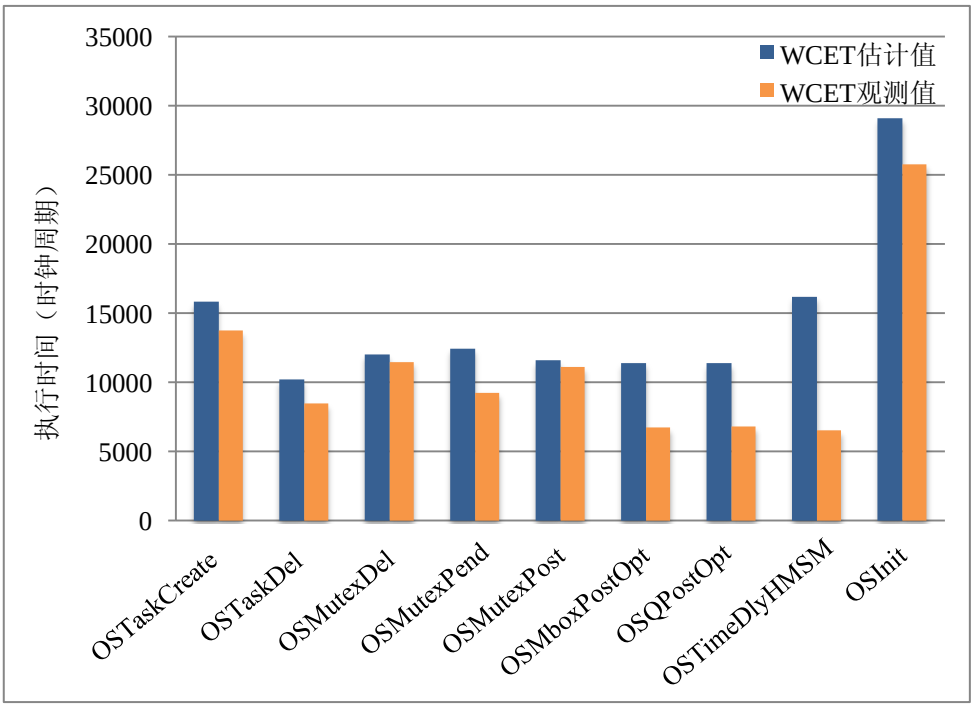


图 6.8 WCET 值在 10,000 以上的系统调用

Fig. 6.8 System calls with WCET values above 10,000

详细观察这 9 个 WCET 值过大的系统调用，发现有相当一部分系统调用的 WCET 分析的过度估计比较大。也就是说，除去过度估计的因素，实际的程序 WCET 值并没有分析结果汇报的那么大。这些系统调用主要集中在互斥信号量管理功能模块，这和本模块的系统调用所实现的功能语义有很大的关系。

6.5.2 对禁止中断区间 WCET 分析的评价

µC/OS-II 共有 79 个系统调用，我们成功的分析了其中的 57 个系统调用中所存在的 76 个禁止中断区间。所有的详细分析结果列于附录 B。某些系统调用中会含有多个禁止中断区间，我们采用数字编号进行区分。对于每个禁止中断区间，我们分别统计了其包含的基本块的个数以及分支的个数，这些属性是对禁止中断区间代码复杂度的一个粗略描述。由于禁止中断区间是系统调用的一个局部程序，而 SimpleScalar 模拟器是无法模拟局部程序的，因此在禁止中断区间的分析结果中不存在执行时间观测值，而仅有 WCET 估计值。禁止中断区间的分析精度在总体上应与系统调用的分析精度类似。函数 OS_Sched()中也存在一个禁止中断区间，而这个函数在很多系统调用中都会出现，因此

我们把 OS_Sched()中的禁止中断区间的分析结果单独列出来。

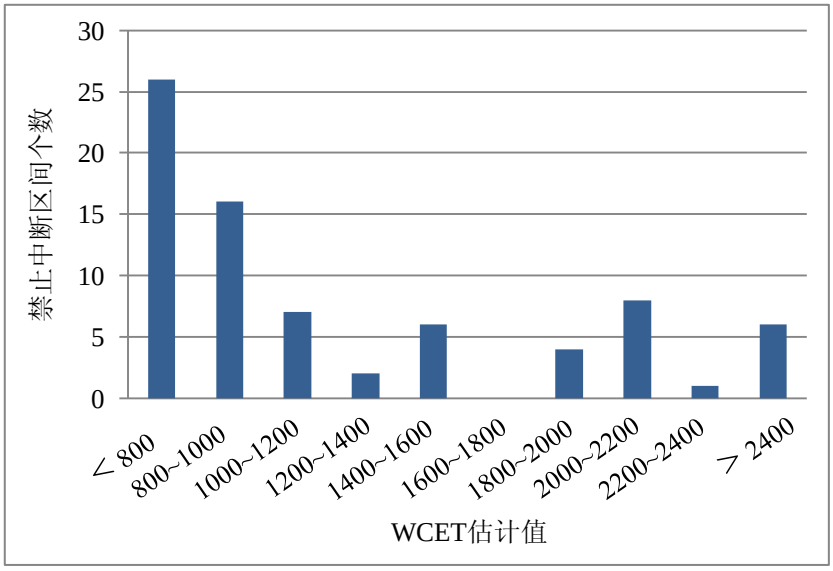


图 6.9 禁止中断区间 WCET 值分布

Fig. 6.9 The distribution of the WCETs of DI-Regions

总的来说，大多数的禁止中断区间所包含的基本块的个数并不是很多，但是有些区间所包含的分支数量较多，导致其结构复杂。图 6.9 表示了所有禁止中断区间 WCET 值的分布情况。大约 2/3 的禁止中断区间具有较短的执行时间，但是有 6 个区间的 WCET 值要远远超过其他的区间（>2400）。这 6 个禁止中断区间主要存在于 OSSemPost()、OSMboxPost()、OSMboxPostOpt()、OSQPost()、OSQPostFront()和 OSQPostOpt()中。进一步分析发现，这 6 个系统调用的功能都是用以释放某种系统资源，在释放系统资源的时候，所有等待该资源的任务都将被设置为“就绪态”，而这个操作是造成执行时间过长的主要因素。分析表明，μC/OS-II 的代码是在性能上进行了充分优化的，因此难以通过改善具体代码实现来进一步提高性能。为去除禁止中断区间执行时间的瓶颈，需要对资源管理功能（主要是资源释放）的语义进行重新设计。具体的设计已超出本章讨论范围，这里不详细讨论。

6.6 目前尚存在的问题

本节主要讨论本章对实时操作系统进行 WCET 分析尚存在的问题，包括单值 WCET 分析用于分析实时操作系统时的可用性问题，以及由于任务切换导致的结果不准确。这些问题为下一步研究工作的开展提供了新的研究方向。

6.6.1 单值 WCET 分析的不足

在对 μC/OS-II 的系统调用进行初步的 WCET 分析的过程中，我们发现 μC/OS-II 实

时内核的代码结构和大多数的应用程序不同。通常情况下，循环结构是影响程序执行时间的关键因素；但是在 μC/OS-II 的设计中，为优化性能，内核中的循环结构的个数已经通过各种设计优化技术尽可能的减到了最少，因此循环对于大多数的系统调用的影响几乎是不存在的。但是，μC/OS-II 在执行时间上的特点是：几乎所有的系统调用内部都存在大量的分支结构，且程序执行不同的分支其执行时间的差异非常大。

表 6.4 不同执行情景下执行时间的变化

Table 6.4 Execution times in different execution scenarios

系统调用	敏感因素	Est.	Sim.1	Est./Sim.1	Sim.2	Est./Sim.2
OSMutexPend	资源是否存在	12,461	2,428	5.13	9,251	1.35
OSMboxPost	等待任务个数	7,440	1,560	4.77	6,347	1.17
OSSemDel	函数调用参数	8,548	1,963	4.35	7,437	1.15

表 6.4 以 OSMutexPend、OSMboxPost 和 OSSemDel 三个系统调用为例，说明了在不同的执行情景下，执行时间的变化情况。OSMutexPend 在一个任务要获取一个互斥信号量的时候被调用执行，它的执行时间主要根据资源是否存在而不同。如果调用该系统调用的任务发现互斥信号量未被占用，那么任务将立即获得该信号量；如果当前该互斥信号量已经被占用，那么任务将根据优先级继承协议对占用该资源的任务的优先级进行调整，并将自己放入等待队列，同时调用调度器执行任务切换。该系统调用在后一种情况下的执行时间要明显高于第一种情况。从表 6.4 可以看出，对 OSMutexPend 系统调用进行 WCET 分析得到的估计值是 12,461，第一种情况的模拟执行时间是 2,428，第二种情况下的模拟执行时间是 9,251。如果采用上述估计值来代表第二种情况的执行时间，那么过度估计为 35%；但是如果采用这一估计值来代表第一种情况的执行时间，那么过度估计高达 413%。倘若系统在大多数情况下工作于第一种状态，那么不区分执行情景而获得的 WCET 值的可用性就非常差。类似的，OSMboxPost 系统调用的执行时间主要与等待消息的任务的个数有很大关系。OSSemDel 系统调用的执行时间与调用该函数的参数有密切关系。

在 μC/OS-II 中，大多数的系统调用都具有类似的特点，其他实时操作系统的情况也应该是类似的。主要原因是，实时操作系统的设计目标就是根据不同的系统状态执行不同的系统行为，这种控制密集型的代码往往在执行时间上具有很大的动态性。尽管传统的 WCET 分析能够给出安全的分析结果，但是“单一值”的 WCET 结果显然已经无法精确的描述以控制密集型代码为主的实时操作系统的执行时间特征。因此需要设计新的分析目标和分析技术，对实时操作系统代码的执行时间进行更加细致的刻画，以提高结果的可用性。

在未来的研究工作中，可以通过“逐步求精”的办法来实现上述目标。首先，我们可以进行最好情况执行时间（BCET）的研究，来确定操作系统系统调用的执行时间下限。一组 BCET 和 WCET 组成的执行时间区间可以粗略的反映某个系统调用在执行时间上的动态特性。其次，可以采用参数化的方法分析实时操作系统的 WCET，进而对系统的时间特性进行更加细致的刻画。Bydge 和 Lisper 等人提出了一种参数化 WCET 分析的框架[127]，其基本思想是生成关于待分析程序的执行时间的一组公式，通过代入不同的参数求得不同情况下的 WCET。他们的工作可以作为参数化分析实时操作系统 WCET 的一个基础，但是还有一个必须解决的重要问题就是“实时操作系统中的参数如何定义”。操作系统中的参数主要是指操作系统的不同执行情景，通常不同的情景是程序的一组控制变量的集合，其取值影响到操作系统程序的执行流程。如何对操作系统的“情景”进行建模，并将情景信息正确映射到控制流程分析中，是值得深入探讨的问题。

6.6.2 任务切换对分析结果的影响

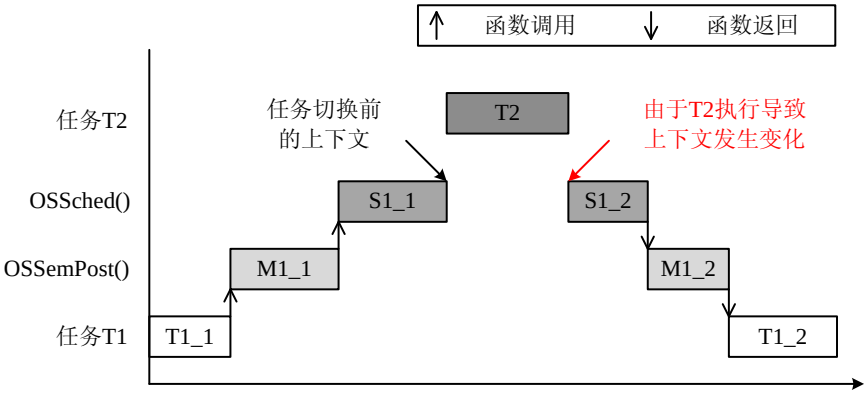


图 6.10 µC/OS-II 任务切换示例

Fig. 6.10 An example of context switch in µC/OS-II

本章研究中遇到了另外一个主要的问题就是由于任务切换造成的分析结果不准确。µC/OS-II 通过函数 OS_Sched()完成任务调度的功能，这一函数在相当一部分系统调用中都会出现。以图 6.10 为例，任务 T1 在执行过程中通过调用 OSSemPost()函数来释放一个信号量资源。一种可能的执行情况是，有更高优先级的任务正在等待该资源，那么这一资源一旦释放，更高优先级的任务将抢占当前任务获得执行。为实现这一目的，在 OSSemPost()的实现中会调用 OS_Sched()函数，在 OS_Sched()的执行过程中完成向更高优先级任务的切换。在这种情况下，任务 T1、OSSemPost()、以及 OS_Sched()的执行都将被任务 T2 的执行切分为两个部分。在 T2 执行完毕后，流水线和 Cache 中的内容都已经被修改，对于任务 T1 而言，此时的执行上下文已经发生了改变。例如，T1 的某些指令可能在前半段 T1_1 的执行过程中被载入了 Cache，但是 T2 的执行将这部分内容从

Cache 中替换了出去，那么任务 T1 的后半段 T1_2 执行的时候，就要产生额外的 Cache 不命中，进而导致 T1_2 的执行时间比不被打断的时候更长。本章所采用的分析技术和工具将 OS_Sched()识别为一个普通函数，因此不能有效识别任务切换，进而不能分析由于任务切换造成的执行时间的变化。这样可能造成分析结果不安全。

因此需要一种新技术，能够比较精确的获知任务或系统调用在执行的过程中，在什么时间、发生过多少次任务切换，任务切换导致的系统状态变化的具体情况又如何。对于一个系统调用而言，如果调用了调度器，就有可能发生调度，且只发生一次（不考虑中断的情况下），那么需要根据整个系统的信息，分析得到关于此刻发生切换的可能性，从而计算出更加准确的切换次数。同时，应该分析程序在切换时可能发生的对 Cache 和流水线的修改，对这个信息了解的越精确，得到的分析结果也就越精确。

对于普通任务，在执行的过程中，还可能被高优先级的任务所抢占，这一行为由高优先级任务到达造成。同时仅仅关心一次到达还不够，因为低优先级的任务可能被抢占多次。因此还要知道被抢占的次数，才能精确的计算低优先级任务在考虑抢占情况下的 WCET。抢占信息可以从可调度性分析中得到，同时，考虑抢占之后计算出的延长的 WCET 可能进而影响到可调度性分析，因为执行时间增加了，被抢占的次数也可能增加。因此，如果考虑抢占对 WCET 的影响，那么需要将可调度性分析和 WCET 分析集成在一个框架中。这方面的研究工作目前正逐渐增多，但是切换造成的 WCET 变化的问题需要继续深入研究。目前，已经有学者在“考虑任务切换行为的 Cache 分析”问题上展开了研究[128-131]，主要目的是安全并且精确的分析由于任务切换造成的 Cache 状态的变化，以及这些变化对 WCET 计算的影响。但是上述研究都是在理论环境下开展的，应用到实际的实时操作系统环境下需要做何种改进尚不清楚，这在一定程度上也限制了上述成果在实际系统中的应用。我们将在未来工作中研究在实际的实时操作系统环境下如何计算考虑了任务切换的程序 WCET。

6.7　小结

本章对工业界广泛使用的 μC/OS-II 实时操作系统进行了静态分析，包括对系统调用以及禁止中断区间的分析，得到了该系统实时时间特性的完整描述。在此基础上，评价了传统技术在分析实时操作系统代码上的精度，以及尚存在的问题。

第7章　支持多核体系结构的 WCET 分析工具的设计与实现

本文第 2 章至第 6 章分别探讨了基于模型检测技术的 WCET 计算方法，基于剪枝思想的单核指令 Cache 分析，基于模型检测技术的多核共享指令 Cache 分析，以及实时操作系统 WCET 分析。本章基于上述各章的技术与方法，在新加坡国立大学开发的 Chronos 工具基础之上，设计实现了一个 WCET 分析工具。本章将对这一工具的结构与功能进行介绍。

7.1　概述

WCET 分析是实时系统研究领域的一个比较具有挑战性的课题方向，对程序进行完整意义上的分析，通常需要多种理论和技术的支撑，因此需要设计相关分析工具来验证分析理论的正确性。目前，成熟的 WCET 分析工具（如 aiT、Bound-T 等）已经在工业界获得了应用。同时，为探索新的分析理论和技术，相关研究组织也都开发了实验性质的 WCET 分析工具，Chronos、Heptane、OTAWA 等工具属于这一类。基于本文第 2 章至第 6 章所研究的方法和技术，我们在 Chronos 工具的基础上设计并实现了一个综合性的 WCET 分析工具，该工具与其他分析工具的最大区别是：提供了一种基于路径的 WCET 计算框架，以及基于这一框架的 Cache 行为分析。由于反编译和 CFG 抽取功能与分析功能相比并不复杂，但是实现所需代码量巨大，因此我们主要沿用了 Chronos 工具的反编译和 CFG 抽取功能模块，同时也保留了 Chronos 工具的图形用户界面。在此基础上，增加了基于模型检测技术的 WCET 计算模块、基于抽象解释的独立 Cache 分析模块、以及基于模型检测技术的多核共享 Cache 分析模块。同时，第 6 章的实时操作系统分析技术也在我们的工具中单独实现为一个功能模块。下面小节中，我们将对该 WCET 分析工具的主要结构与工作原理进行详细介绍。

7.2　系统功能设计

本文所设计的 WCET 分析工具的总体结构如图 7.1 所示。其中虚线内部的部分为该分析工具所包含的主要功能模块，虚线以外是与工具进行交互的外部环境，包括目标程序、编译器、用户输入环境、整数线性规划求解器、模型检测器、模拟器等。其中，淡蓝色部分为工具中所保留的 Chronos 原有的功能模块，橘黄色部分为本文新增的分析模

块。下面我们将详细介绍该工具中各功能的工作原理，以及采用这一工具进行 WCET 分析的主要流程。

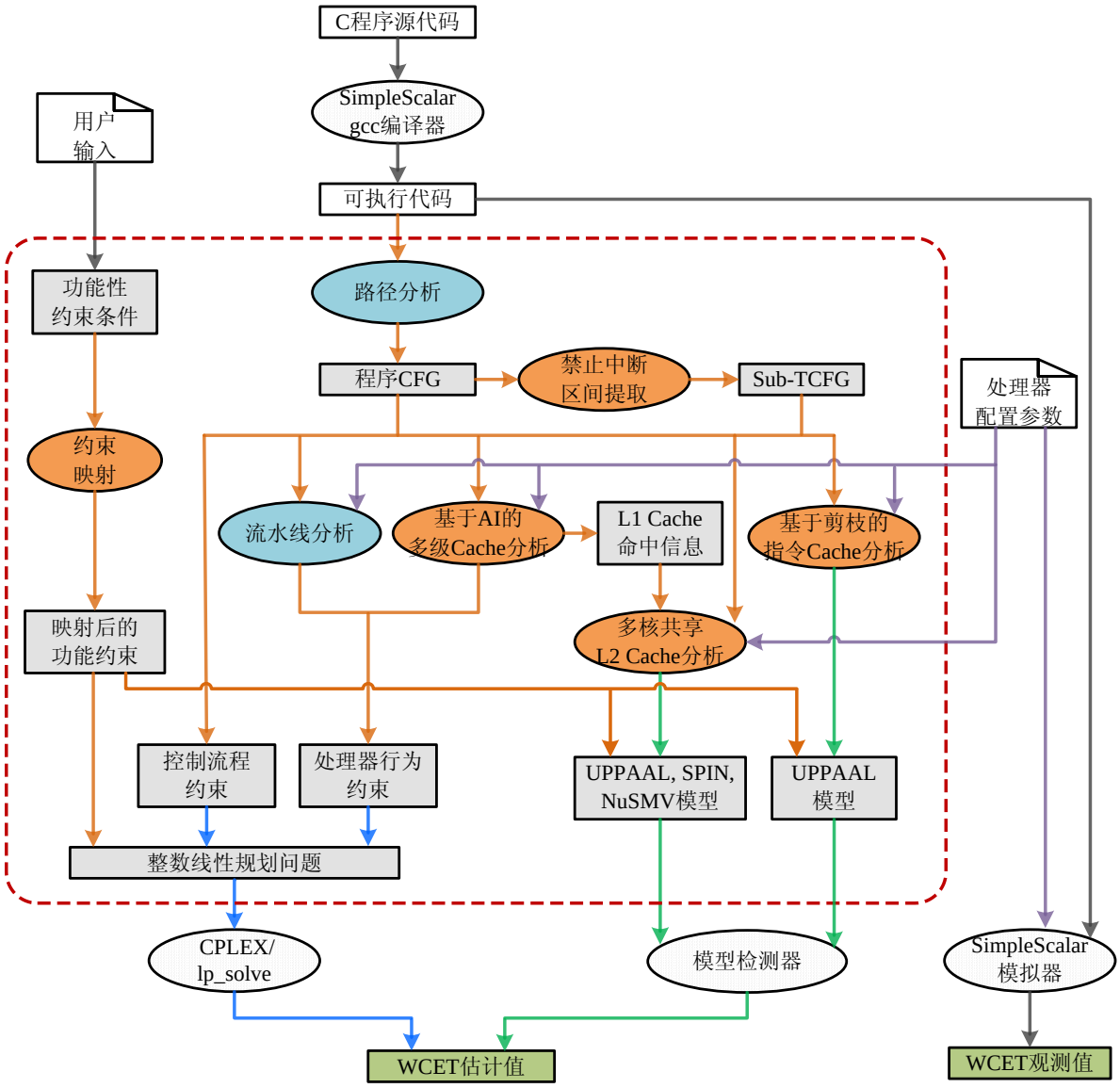


图 7.1 WCET 分析工具总体框架

Fig. 7.1 The framework of the WCET analysis tool

待分析的程序为 C 代码编写，通过面向 SimpleScalar 平台的 gcc 交叉编译器，可以将源程序编译为 PISA 指令集的可执行程序，这个可执行程序就是 WCET 分析工具的直接输入。程序读入工具之后，采用 Chronos 原有的路径分析功能模块，将可执行程序进行反编译，提取详细的指令信息，并生成程序的 CFG。如果是对操作系统的禁止中断区间进行分析，那么还需要从程序的 CFG 中提取禁止中断区间对应的 Sub-TCFG。程序的 CFG 或 Sub-TCFG 表示了程序的流程结构，这一信息将成为处理器行为分析的输入。处理器行为分析主要包括两个部分，流水线分析和 Cache 分析。Chronos 工具的主要贡献就是对超标量乱序流水线的分析，而 Cache 分析相对薄弱。本文在处理器行为分析方面

的研究主要集中在 Cache 分析上，因此工具保留了原有的流水线分析，并设计了全新的 Cache 分析模块。其中 Cache 分析模块分为三个部分：（1）一个基于抽象解释技术的多级 Cache 分析，这个部分主要采用了现有工作的一些分析方法。在不存在多核共享 Cache 的情况下，这一分析模块可以处理单核多级 Cache；（2）第 4 章关于“基于剪枝的单核系统指令 Cache 分析”单独实现为一个模块；（3）第 5 章所做的研究主要体现在“多核共享 L2 Cache 分析”模块，该模块能够对多核环境下的共享 Cache 进行分析。循环变量、不可行路径等信息可以由用户手工输入，我们称这些输入为“功能性约束条件”，根据下一步计算的需要，这些约束条件将被映射为不同模型中所需要的形式。

在程序流程、处理器行为分析、以及功能性约束等信息都齐备以后，需要采用某种手段计算求解程序的 WCET。蓝色线条部分表示的是 Chronos 工具原有的计算方法——基于隐式路径枚举的计算方法。根据计算需要，程序流程信息、处理器行为信息、功能性约束等分别被转化为线性约束描述的“控制流程约束”、“处理器行为约束”以及“映射后的功能约束”等，这些约束与目标函数结合起来，就形成了一个整数线性规划的问题，这一问题可以交付商业的求解软件（如 CPLEX）或开源的数学规划求解软件（如 lp_solve）进行求解，最终得到程序的 WCET 值。本文主要研究了采用了模型检测技术的 WCET 计算方法，该功能对应的是绿色线条的部分。无论是程序流程信息，还是处理器各部件的行为，都可以通过自动机的形式给予描述（也就是本文第 3、4、5 章所涉及的技术），并最终生成不同模型检测器所能够接受的不同格式的模型文件。将这些模型文件交付对应的模型检测器进行验证，就可以求得程序的 WCET 值。在另外一条分支上，可执行程序可以交给 SimpleScalar 模拟器进行模拟执行，用以获得程序的 WCET 观测值，它通常用来和 WCET 估计值进行比较，以粗略的分析后者的精确程度。下面对本文所涉及的几个重要功能模块给予进一步的介绍。

（1） 约束映射模块

该模块主要完成用户输入约束向其他功能模块所需约束形式的映射。以循环上限为例，Chronos 内建的功能是让用户根据源代码的行号，输入对应循环的上限，这一信息进一步被 Chronos 转化为程序基本块之间的执行次数的约束关系。这一约束可以直接用于基于隐式路径枚举技术的计算方法中，但是如果是对操作系统的禁止中断区间进行分析，或采用模型检测技术进行计算，则需要转换约束的形式。在计算禁止中断区间的 WCET 时，由于新生成的 Sub-TCFG 的编号与原程序的 CFG 的编号不同，因此转换主要是保证循环信息被正确的映射到对应的 Sub-TCFG 的基本块上。在基于模型检测技术的 WCET 计算中，循环上限对应于有限状态自动机的某一部分的行为，这里约束映射的功能主要是完成线性约束向有限状态自动机行为语义的转化。

（2） 禁止中断区间提取模块

这一功能模块主要是在分析实时操作系统的禁止中断区间的时候被使用，其它时候可以关闭。这一模块的功能主要是从程序的 CFG 中提取出禁止中断区间代码所对应的局部 CFG（称为 Sub-TCFG），所以它的功能属于程序流程分析的范畴，与处理器行为分析等功能模块没有交叠。无论是原程序的 CFG 还是禁止中断区间的 Sub-TCFG，都采用完全相同的数据结构进行表示，因此对于其他模块都是透明的，都可以作为处理器行为分析的输入，以及用于控制流程约束的生成。

（3） 基于抽象解释技术的多级 Cache 分析模块

这一模块的主要功能是进行独立的单核多级 Cache 分析，且任何一级 Cache 的分析结果都可以单独提取出来。例如 L1 Cache 的分析结果用于多核共享 L2 Cache 分析。该模块所采用的理论和技术主要是现有的基于抽象解释的分析技术[29]，第 5 章中对 Cache 持久性分析的修改也已经被实现到这一模块中。该分析模块的结果形式是所有的指令在各级 Cache 中的命中情况，包括“一定命中”、“一定不命中”、“首次不命中”和“不可确定”。这一信息需要经过转化，使用到计算基本块 WCET 的步骤中。这和 Cinderella 工具[23]所采用的基于“Cache 冲突图”的解决方法本质上是不同的。

（4） 基于剪枝思想的单核指令 Cache 分析模块

这一模块对应于本文第 4 章的主要贡献。该模块以程序 TCFG 为输入，根据第 4 章介绍的剪枝技术，对原程序的流程图进行剪枝操作。其结果是一个分支更少的流程图，同时由于程序变形，局部程序可能与原始流程图不直接对应。采用模型检测技术分析单核指令 Cache 的优势已经在第 4 章进行了详细阐述，主要是该技术可以分析目前几乎所有的 Cache 替换算法。在得到了经过剪枝的程序的 TCFG 后，给定一个 Cache 替换策略，我们就可以生成对应的模型，通过将这一模型交付模型检测器分析，得到程序的 WCET。这部分工作也需要使用到约束映射模块的输出。

（5） 多核共享 L2 Cache 分析模块

这一模块是本文 WCET 分析工具的核心模块之一。其主要功能是采用模型检测技术，完成对多核共享 Cache 的行为分析，并结合基于模型检测技术的 WCET 计算方法，求解程序的 WCET。这一模块根据给定的程序流程结构信息、L1 Cache 访问命中信息，以及处理器的配置，为待分析程序自动生成对应的程序自动机，为处理器部件（这里主要是 Cache）生成对应的行为自动机。这些自动机之间通过消息机制进行同步，共同构成一个自动机网络，模型检测器即是对这个自动机网络进行属性验证。目前，在 WCET 计算方面，该模块可以生成 UPPAAL，SPIN，NuSMV 三种模型检测器所能够接受的模型；在多核共享 Cache 分析方面，可以生成 UPPAAL 所能够接受的模型。未来有可能

尝试采用 SPIN 或其他模型检测器对多核共享 Cache 进行分析。通过对该模块的简单扩充，就可以完成上述设计。

7.3　系统实现

原 Chronos 工具的图形用户界面采用 JAVA 语言编写，分析引擎部分采用 C 语言编写，这是目前学术界很多 WCET 工具和模型检测工具通常所采用的设计结构。由于本文的功能实现主要集中在分析引擎中，我们沿用了 Chronos 原有的图形用户界面，因此所采用的开发语言主要是 C 语言。

系统实现的主要代码量集中在基于抽象解释的多级 Cache 分析，以及基于模型检测技术的单核独立 Cache 分析和多核共享 Cache 分析。实现过程中重新定义和实现了多级 Cache 这一核心数据结构。从程序的 TCFG 转换出来的 UPPAAL、SPIN、NuSMV 等模型都具有较为固定的格式，因此我们针对上述模型文件设计了对应的数据结构，以方便模型文件的生成。由于受原有 Chronos 基本框架的限制，开发主要基于 C 语言。我们针对每种模型文件的生成，模拟 C++语言中类的概念设计了对数据结构进行操作的成员函数，这样可以简化设计并减少程序错误的出现。例如在多核共享 Cache 分析中，最多的程序需要生成超过 1000 个节点和边的自动机。采用模拟面向对象设计的数据结构及其相关操作函数，可以比较容易的实现对数量众多的节点和边的操作，同时降低程序出错的概率。

开发环境主要是在 Windows 宿主机上安装 VMware 虚拟机，并在虚拟机中安装 Linux 操作系统（发行版本为 Fedora Core 10），所有的程序均运行于 Linux 环境。在虚拟机环境中而非直接在裸机上安装 Linux 主要是出于开发方便。给定待分析的 C 语言源代码程序，通过带有交叉编译功能的 gcc-2.7.3 编译器将代码编译成 PISA 指令集的可执行代码，该代码交付给本文所设计的 WCET 分析工具即可求得程序的 WCET；同时该可执行代码也可以交付给 SimpleScalar 3.0 模拟器模拟执行，该模拟器能够给出模拟执行程序所需要的时间，精度为时钟周期。通过对比模拟执行得到的 WCET 观测值，可以评估静态分析的分析精度。

7.4　小结

在本文对 WCET 计算技术与处理器行为（主要是 Cache 行为）的分析基础之上，本章基于 Chronos 工具设计并实现了一个以模型检测技术为主要框架的 WCET 分析工具。本章详细介绍了该 WCET 分析工具的总体结构、工作原理以及各模块功能。

第8章 结 论

WCET 分析是实时系统时间验证的重要任务之一。在硬实时系统中，为保证分析结果的安全性，通常采用静态方法分析程序的 WCET。本文主要针对静态 WCET 分析中的关键技术展开了研究，涉及的主要问题包括：WCET 计算框架，单核系统中独立 Cache 的分析，多核系统中共享 Cache 的分析，实时操作系统 WCET 分析等。本文对以上问题进行了深入研究，提出了相应的解决方案，通过大量实验验证了本文提出的方法与技术的可行性和有效性。

8.1 本文的主要贡献与结论

（1）提出了一种全新的基于模型检测技术的 WCET 计算方法。考虑到隐式路径枚举等传统 WCET 计算技术在描述能力上的限制，本文采用模型检测技术作为静态 WCET 分析的基础性框架，充分利用了模型检测技术搜索最优解的能力，使得分析结果具有更高的精度。采用了 SPIN、NuSMV、UPPAAL 等不同工作原理的模型检测器进行分析，探讨了分析方法的时间可伸缩性和空间可伸缩性，并给出了基于模型检测技术的 WCET 计算方法的适用范围。

（2）提出了一种基于剪枝思想的单核系统指令 Cache 分析方法。在基于模型检测技术的 WCET 计算框架的基础上，研究了利用这一技术分析单核系统 FIFO 替换策略的方法。针对模型检测技术在 WCET 分析中可能出现的状态空间爆炸问题，提出了基于剪枝思想的分析方法。该方法通过削减程序模型中符合特定条件的程序分支，能够在不损失分析精度的前提下，有效提高分析的性能以及可伸缩性。

（3）提出了一种面向多核共享 Cache 的处理器行为分析方法。现有的方法在分析这一问题的过程中只考虑了不同核心上的程序在地址上的冲突情况，导致分析结果的过度估计过高。基于这一问题，本文采用模型检测技术对多核共享 Cache 的行为进行了建模，模型能够在更细的粒度上挖掘冲突发生的时间关系，从而大大提高了分析的精确性。这部分研究对于加深理解程序在多核共享 Cache 上的时间特性起到了重要作用。

（4）对工业界广泛使用的 μC/OS-II 实时操作系统进行了 WCET 分析。本文对 μC/OS-II 实时操作系统的系统调用和禁止中断区间进行了静态 WCET 分析，详细分析了传统技术在分析实时操作系统时的精度以及尚存在的主要问题。同时，实验结果本身就是对 μC/OS-II 实时操作系统实时特性的一个完整的量化描述，它对于使用者了解系统的时间特性，以及帮助开发者改善系统的实时性能都具有重要的应用价值。

（5）本文在新加坡国立大学开发的 Chronos 工具的基础上，设计并实现了一个静态 WCET 分析工具，实现了本文所提出的基于模型检测技术的 WCET 计算方法、基于剪枝思想的单核系统指令 Cache 分析、多核系统共享 Cache 分析、实时操作系统 WCET 分析等具体技术和方法，验证了这些方法的正确性和有效性。

8.2 进一步的工作

随着处理器新体系结构的不断出现，静态 WCET 分析将面临越来越多的挑战。对于新的体系结构特征，研究兼具高精度与高可伸缩性的分析方法，并验证这些分析方法在实际系统中的可用性是静态 WCET 分析研究领域的关键课题。基于已经获得的研究成果，在未来工作中，我们将在以下几个方面进一步深入研究：

（1）面向具体体系结构特性的剪枝技术。本文研究内容的核心就是采用模型检测技术进行静态 WCET 分析，这种分析技术能够得到非常好的分析精度，但是面临的主要问题是状态空间爆炸。面向具体应用的剪枝技术是解决状态空间爆炸的有效途径。本文研究了基于 FIFO 替换策略的剪枝技术，未来应继续深入探讨面向其他体系结构特性的剪枝技术，这对于扩大模型检测技术在静态 WCET 分析中的应用具有重要意义。

（2）多核共享 Cache 静态分析方法。本文探索了采用模型检测技术对多核共享 Cache 进行分析的可能，取得了分析精度上的收益。但是随着深纳米技术的发展，未来的处理器将会在一个芯片上集成更多的处理核心，共享 Cache 上的干涉问题将更加严重，模型检测技术的状态空间爆炸问题也因此更加严峻。在不过度丧失分析精度的前提下提出具有高可伸缩性的分析方法将是多核时代 WCET 分析的重要课题。

（3）参数化的 WCET 分析方法。实时系统的系统程序和应用程序日益复杂，单一的 WCET 值在描述程序的时间属性方面将变得越来越不可用。因此未来需要研究参数化的 WCET 分析方法，分析得到程序执行时间的某种函数描述形式，通过给定不同参数，得出系统在不同环境下对应的 WCET 值。

（4）考虑程序 WCET 的编译技术。静态分析的基本出发点是给定待分析程序，通过分析手段得到程序的执行时间特性。但是基于分析的方法并不能从根本上改善程序自身的可预测性。一种更加主动的办法就是在编译程序的过程中，加入对程序 WCET 的考虑，合理分配处理器资源，使得编译后的程序在执行时间上具有更好的可预测性。这将从根本上提高系统的时间可预测性，同时也将大大简化 WCET 分析的复杂度。

参考文献

1. Raj Kamal. Embedded Systems: Architecture, Programing and Design [M], United States: The McGraw-Hill Companies, Inc., 2003.
2. Jane W.S. Liu. Real-Time Systems [M]. United States: Pearson Education, 2002.
3. Quirk, William J. Verification and Validation of Real-Time Software [M]. New York: Springer-Verlag, Inc., 1985.
4. Lui Sha, Tarek Abdelzaher, Karl-Erik arzen, and et al. Real-Time Scheduling Theory: A Historical Perspective [J], Real-Time Systems, 2004, 28(2-3): 101-155.
5. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, and et al. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools [J], ACM Trans. Embedded Computing Systems, 2008, 7(3): 1-53.
6. Jan Gustafsson. Usability Aspects of WCET Analysis [A], Proceedings of 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing [C], 2008, 346-352.
7. Alan C. Shaw. Reasoning about Time in Higher Level Language Software [J], IEEE Transactions on Software Engineering, 1989, 1(2): 875-889.
8. ChangYun Park and Alan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema [J], IEEE Transactions on Computers, 1991, 24(5): 48-57.
9. ChangYun Park. Predicting Deterministic Execution Times of Real-Time Programs [D]. Unitd States: University of Washington, 1992.
10. Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration [A], in Workshop on Languages, Compilers and Tools for Real-Time Systems [C], 1995, 456-461.
11. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software [A], Proceedings of the IEEE Real-Time Systems Symposium [C], 1995, 298.
12. www.ait.com [EB/OL], 2009.
13. Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. Chronos: A Timing Analyzer for Embedded Software [J], Science of Computer Programming, 2006, 69(1-3): 56–67.

14. www.boundt.com [EB/OL], 2009.
15. http://www.mrtc.mdh.se/projects/wcet/sweet.html [EB/OL], 2009.
16. Friedhelm Stappert, Andreas Ermedahl and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects [A], Technical Report 2001-012 [C], Uppsala University, 2001.
17. Roderick Chapman. Worst-Case Timing Analysis via Finding Longest Paths in SPARK Ada Basic-Path Graphs [A], Technical Report YCS-94-246 [C], University of York, 1994.
18. Roderick Chapman, Alan Burns, and Andy Wellings. Worst-Case Timing Analysis of Exception Handling in Ada: Towards Maturity [A], Proceedings of the 1993 AdaUK conference [C], 1993, 148-164.
19. Roderick Chapman, Alan Burns, and Andy Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada [A], Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems [C], 1994.
20. Sung-Soo Lim, Young Hyun Bae, and et al. An Accurate Worst-Case Timing Analysis Technique for RISC Processors [J], IEEE Tran. on Software Engineering, 1995, 21(7): 593-604.
21. Sung-Soo Lim, Young Hyun Bae, and et al. An Accurate Worst Case Timing Analysis Technique for RISC Processors [A], Proceedings of IEEE Real-Time Systems Symposium [C], 1994.
22. Young Hyun Bae, Sung-Soo Lim, and et al. Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study [A], Proceedings of IEEE Real-Time Systems Symposium [C], 1995, 308-319.
23. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches [A], Proceeding of IEEE Real-Time Systems Symposium [C], 1996, 254.
24. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling [J], ACM Transactions on Design Automation of Electronic Systems, 1999, 4(3): 257-279.
25. Xianfeng Li, Abhik Roychoudhury, Tulika Mitra. Modeling Out-of-Order Processors for Software Timing Analysis [A], Proceedings of IEEE Real-Time System Symposium [C], 2004, 92-103.

26. Xianfeng Li, Abhik Roychoudhury, Tulika Mitra. Modeling Out-of-Order Processors for WCET Analysis [J], Real-Time Systems, 2006, 34(3): 195-227.
27. Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints [A], Proceedings of ACM Symposium on Principles of Programming Languages [C], 1977.
28. Christian Ferdinand and Reinhard Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems [J], Real-Time Systems, 1999, 17(2-3).
29. Henrik Theiling, Christian Ferdinand and Reinhard Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analysis [J], Journal of Real Time Systems, 1999, 17(2-3): 131-181.
30. Jorn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation [A], Proceedings of ACM International Workshop on Languages, Compilers and Tools for Embedded System [C], 1999, 35-44.
31. Jun Yan and Wei Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches [A]. Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2008, 80-89.
32. Wei Zhang and Jun Yan. Accurately Estimating Worst-Case Execution Time for Multi-Core Processors with Shared Direct-Mapped Instruction Caches [A], Proceedings of 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2009, 455-463.
33. http://www.rtems.com [EB/OL], 2009.
34. Antoine Colin and Isabelle Puaut. Worst-case Execution Time Analysis of the RTEMS real-Time Operating System [A], Proceedings of 13th Euromicro Conference on Real-Time Systems [C], 2001, 191-198.
35. www.enea.com [EB/OL], Enea Embedded Technology page, 2009.
36. Martin Carlsson. Worst Case Execution Time Analysis, Case Study on Interrupt Latency for the OSE Real-Time Operating System [D], Sweden: Master Thesis of Royal Institute of Technology, 2002.
37. Martin Carlsson, Jakob Engblom, Andreas Ermedahl, and et al. Worst-Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System [A], Proceedings of 2nd International Workshop on Real-Time Tools [C], 2002.
38. Daniel Sandell. Evaluating Static Worst-Case Execution-Time Analysis for a Commercial

Real-Time Operating System [D], Sweden: Master Thesis of Malardalen University, 2004.

39. Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Bjorn Lisper. Static Timing Analysis of Real-Time Operating System Code [A], Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods [C], 2004, 146-160.
40. Sergio Ruocco. Real-Time Programming and L4 microkernels [A], Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications [C], 2006.
41. Mohit Singal and Stefan M. Petters. Issues in Analyzing L4 for its WCET [A], Proceedings of the 1st Int. Workshop on Microkernels for Embedded Systems [C], 2007.
42. Thomas Lundqvist and Per Stenstrom. Timing Anomalies in Dynamically Scheduled Microprocessors [A], Proceedings of the 20th IEEE Real-Time Systems Symposium [C], 1999, 12–21.
43. Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools [A], Proceedings on Real-Time System [C], 2003, 1038–1054.
44. Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction [J], Journal of Real time Systems, 2000, 18(2-3): 249-274.
45. Iain Bate and Ralf Reutemann. Worst-case timing analysis for dynamic branch predictors [A], Proceedings of the 16th Euromicro Conference on Real-Time Systems [C], 2004, 215-222.
46. Andreas Ermedahl. A Modular Tool Architecture for Worst-Case Execution Time Analysis [D], Sweden: Uppsala University, Dept. of Information Technology, Box 325, 2003.
47. Antoine Colin and Isabelle Puaut. A modular and Retargetable Framework for Tree Based WCET Analysis [A], Proceedings of the 13th Euromicro Conference on Real-Time Systems [C], 2001, 37-44.
48. Antoine Colin and Guillem Bernat. Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis [A], Proceedings of 14th Euromicro Conference of Real-Time Systems [C], 2002, 50-59.
49. Alexander Metzner. Why Model Checking Can Improve WCET Analysis [A], Proceedings of Computer Aided Verification [C], 2004, 334-347.

50. Yanhong A. Liu and Gustavo Gomez. Automatic Time-Bound Analysis for a Higher-Order Language [A], Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems [C], 1998, 75-86.
51. Yanhong A. Liu and Gustavo Gomez. Automatic Accurate Cost-Bound Analysis for High-Level Languages [J], IEEE Transactions on Computers, 2001, 50(12): 1295-1309.
52. Christopher Healy, Mikael Sjodin, Viresh Rustagi and David Whalley. Bounding Loop Iterations for Timing Analysis [A], Proceedings of IEEE Real-time Appplications Symposium [C], 1998.
53. Christopher Healy, Mikael Sjodin, Viresh Rustagi and et al. Supporting Timing Analysis by Automatic Bounding of Loop Iterations [J], Real-Time Systems, 2000, 18(2-3): 129-156.
54. Andreas Ermedahl and Jan Gustafsson. Deriving Annotations for Tight Calculation of Execution Time [A], Proceedings of European Conference on Parallel Processing [C], 1997, 1298-1307.
55. Thomas Lundqvist and Per Stenstrom. An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution [J], Journal of Real-Time Systems, 1999, 17(2-3): 183-207.
56. Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, and et al. Reliable and Precise WCET Determination for a Real-Life Processor [A], Proceedings of Internatinal Workshop on Embedded Software [C], 2001, 469-485.
57. Peter Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs [J], Journal of Real-time Systems, 1989, 1(2): 159-176.
58. Jan Gustafsson, Andreas Ermedahl, Bjorn Lisper, and et al. ALF-A Language for WCET Flow Analysis [A], Proceedings of WCET 2009 [C], 2009.
59. Jan Gustafsson, Bjorn Lisper, and et al. ALL-TIMES - a European Project on Integrating Timing Technology [A]. Proceedings of 3rd International Symposium on Leveraging Applications of Formal Methods [C], 2008, 17, 445–459.
60. Peter Puschner. Worst-case Execution Time Analysis at Low Cost [J], Control Engineering Practice, 1998, 6, 129-135.
61. Jakob Engblom, Andreas Ermedahl, and Peter Altenbernd. Facilitating worst Case Execution Time Analysis for Optimized Code [A], Proceedings of the 10th Euromicro Real-Time Systems Workshop [C], 1998.

62. Bernhard Rieder, Ingomar Wenzel, Klaus Steinhammer, Peter Puschner. Using a Runtime Measurement Device with Measurement-Based WCET Analysis [A], Proceedings of IESS [C], 2007, 15-26.
63. Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking [A], Proceedings of European Design Atuomation Conference [C], 2005, 606-611.
64. http://www.symtavision.com [EB/OL], 2009.
65. http://www.rapitasystems.com/rapitime/ [EB/OL], 2009.
66. http://www.trnicely.net/pentbug/pentbug.html [EB/OL], 2009.
67. http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5r.htm [EB/OL], 2009.
68. Barry Boehm and Victor R. Basili. Software Defect Reduction Top 10 List [J], IEEE Computer, 2001, 34(1):135–137.
69. Boris Beizer. Software Testing Techniques [M], Van Nostrand Reinhold, 1990.
70. James A. Whittaker. What is Software Testing? Why is it so Hard [J], IEEE Software, 2000, 17(1):70–79.
71. Doron A. Peled. Software Reliability Methods [M], Springer-Verlag, 2001.
72. Louis Scheffer, Luciano Lavagno, Grant Martin. Electronic Design Automation for Integrated Circuits Handbook [M], Tyalor and Francis, 2006.
73. Laung-Terng Wang, Yao-Wen Chang, Kwang-Ting (Tim) Cheng. Electronic Design Automation: Synthesis, Verification, and Test (Systems on Silicon) [M], Morgan Kaufmann, 2009.
74. John Rushby. Formal methods and the certification of critical systems [A], Technical Report SRI-CSL-93-7 [C], SRI International, 1993.
75. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic [A], In Logic of Programs, Lecture Notes in Computer Science [C], 1981, 52-71.
76. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR [A], In 5th International Symposium on Programming [C], 1982, 337–351.
77. Christel Baier and Joost-Pieter Katoen. Principles of Model Checking [M], The MIT Press Cambridge, Massachusetts London, England, 2007.
78. Edmund M. Clarke, Orna Grumberg and Doron A. Peled. Model Checking [M], MIT

Press, 1999.

79. Gerard J. Holzmann. The Theory and Practice of a Formal Method: NewCoRe [A], In IFIP World Congress [C], North Holland, 1994, 35-44.
80. Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi and et al. Verification of the Futurebus+ cache coherence protocol: A Case Study in Model Checking [A], Proceedings of the 27th Australasian conference on Computer science [C], 2004, 65-71.
81. David Agnew, Luc J. M. Claesen, Raul Camposano. Compuer Hardware Description Languages and their Applications [M], Kluwer Academic Publishers, 1993.
82. Jorgen Staunstrup, Henrik R. Andersen, Henrik Hulgaard and et al. Practical Verification of Embedded Software [J], IEEE Computer, 2000, 33(5):68-75.
83. Klaus Havelund, Mike Lowry and John Penix. Formal Analysis of a Space-Craft Controller Using SPIN [M], IEEE Transactions on Software Engineering, 2001, 27(8):749-765.
84. Thomas Kropf. Introduction to Formal Hardware Verification [M], Springer-Verlag, 1999.
85. Alberto L. Sangiovanni-Vincentelli, Patrick C. McGeer and Alexander Saldanha. Verification of Electronic Systems [A], Proceedings of the 33rd Annual Conference on Design Automation [C], 1996, 106-111.
86. Michael Yoeli. Formal Verification of Hardware Design [M], IEEE Computer Society Press, 1990.
87. Zonghua Gu, Xiuqiang He, and Mingxuan Yuan. Optimization of Static Task and Bus Access Schedules for Time-Triggered Distributed Embedded Systems with Model-Checking [A], Proceedings of the 44th annual conference on Design automation [C], 2007, 294-299.
88. Zonghua Gu, Mingxuan Yuan, and Xiuqiang He. Optimal Static Task Scheduling on Reconfigurable Hardware Devices Using Model-Checking [A], Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium [C], 2007, 32-44.
89. Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng and Ge Yu. Exact Schedulability Analysis of Global Scheduling on Multiprocessor Platforms by Symbolic Model Checking [A], Proceeding of the 11th IEEE International Symposium on Object-Oriented Real-Time Systems [C], 2008.

90. Zonghua Gu, Mingxuan Yuan, Nan Guan, Mingsong Lv, Qingxu Deng and Ge Yu. Static Scheduling and Software Synthesis of Dataflow Models with Symbolic Model-Checking [A], Proceedings of the 28th Real-Time Systems Symposium [C], 2007.
91. Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP Alone [A], Proceedings of VMCAI [C], 2004, 309-322.
92. Martin Ouimet and Kristina Lundqvist. Verifying Execution Time Using the TASM Toolset and UPPAAL [A], technical report [C], MIT, 2008.
93. Mordechai Ben-Ari. Principles of the Spin Model Checker [M], Springer, 2008.
94. Gerard J. Holzmann. The Model Checker SPIN [J], IEEE Transaction on Software Engineering, 1997, 23(5):279–295.
95. Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia and et al. Nusmv 2: An Opensource Tool for Symbolic Model Checking [A], Proceedings of the 14th International Conference on Computer Aided Verification [C], 2002, 359-364.
96. Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, and et al. Nusmv 2.4 User Manual [A], Technical report [C], ITC-irst, Italy, 2005.
97. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal [A], Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems [C], 2004, 200-236.
98. Kim G. Larsen, Paul Pettersson and Wang Yi. Uppaal in a Nutshell [J], Springer International Journal of Software Tools for Technology Transfer, 1997, 1(1-2), 134-152.
99. Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools [A], In Lecture Notes on Concurrency and Petri Nets [C], 2004, 87-124.
100. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html [EB/OL], 2009.
101. Yan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing Predictability of Cache Replacement Policies [J], Real-Time Systems, 2007, 37(2): 99-122.
102. Douglas Hamilton. Multi-Core Microprocessors in Embedded Applications [M], Freescale Semiconductor White Paper, 2005.
103. Max Domeika. Software Development for Embedded Multi-core Systems – A Practical Guide Using Embedded Intel Architecture [M], Elsevier, 2008.
104. Shekhar Borkar, Norman P. Jouppi, Per Stenstrom. Microprocessors in the Era of Terascale Integration [A], Proceedings of the conference on Design, automation and test

in Europe [C], 2007, 237-242.

105. Guy E. Blelloch, Phillip B. Gibbons. Effectively Sharing a Cache among Threads [A], Proceedings of the 16th annual ACM symposium on Parallelism in algorithms and architectures, 2004, 235-244.
106. Max Domeika. Software Development for Embedded Multi-Core Systems [M], Elsevier, 2008.
107. Steve Daily. Software Design Issues for Multi-core/Multiprocessor Systems [EB/OL], www.embedded.com, 2006.
108. Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, Doug Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures [A], Proceedings of the 27th International Symposium on Computer Architecture [C], 2000, 248-259.
109. Shekhar Borkar. Design Challenges of Technology Scaling [J], In IEEE Micro, 1999, 19(4): 23-29.
110. Krste Asanovic, et al. The Landscape of Parallel Computing Research: A View from Berkeley [A], Technical Report of UC Berkeley [C], 2006.
111. Jon Bradley. Advantages of Using the TMS320C6474 over the TMS320C6455 [M], Texas Instruments White Paper, 2008.
112. The ARM Cortex-A9 Processors [M], ARM White Paper, 2007.
113. QorIQ™ P4080 Communications Processor Product Brief [M], Freescale Semiconductor, 2008.
114. Sanjeev Kumar, Christopher J. Hughes, Anthony Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors [A], Proceedings of the 34th annual international symposium on Computer architecture [C], 2007, 162-173.
115. Alan Zeichick. Coarse-Grained Vs. Fine-Grained Threading for Native Applications [EB/OL], AMD Developer Central, 2006.
116. Sally A. McKee. Reflections on the Memory Wall [A], Proceedings of the 1st conference on Computing Frontiers [C], 2004.
117. Dhruba Chandra, Fei Guo, Seongbeom Kim, Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture [A], Proceedings of the 11th International Symposium on High-Performance Computer Architecture [C], 2005, 340-351.
118. Damien Hardy and Isabelle Puaut. WCET Analysis of Multi-Level Non-Inclusive

Set-Associative Instruction Caches [A], Proceedings of the 2008 Real-Time Systems Symposium [C], 2008, 456-466.

119. Jan Gustafsson and Andreas Ermedahl. Experiences from Applying WCET Analysis in Industrial Settings [A], Proceedings of 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing [C], 2007, 382-392.
120. Stefan Schaefer, Bernhard Scholz, Stefan M. Petters, and Gernot Heiser. Static Analysis Support for Measurement-Based WCET Analysis [A], In RTCSA [C], 2006.
121. Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis [A], Proceddings of WCET [C], 2008.
122. Peter Puschner. Transforming Execution-Time Boundable Code into Temporally Predictable Code [A], Proceedings of the IFIP 17th Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems [C], 2002, 163-172.
123. Guenter Khyo, Peter Puschner, and Martin Delvai. An Operating System for a Time-Predictable Computing Node [A], Proceedings of the 6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems [C], 2008, 150-161.
124. Raimund Kirner and Peter Puschner. Time-Predictable Task Preemption for Real-Time Systems with Direct-Mapped Instruction Cache [A], In ISORC 2007 [C], 2007, 87-93.
125. http://www.micrium.com [EB/OL], 2009.
126. Jean J. Labrosse. MicroC/OS-II: The Real-Time Kernel, [M], CMP Books, 2002.
127. Björn Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis [A], Proceedings of WCET [C], 2003.
128. Jan Staschulat and Rolf Ernst. Scalable Precision Cache Analysis for Real-Time Software [J], ACM Transaction on Embedded Computing Systems, 2007, 6(4).
129. Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, and et al. Bounding Cache-Related Preemption Delay for Real-Time Systems [J], IEEE Transactions on Software Engineering, 2001, 27(9): 805-826.
130. Fadia Nemer, Hugues Casse, Pascal Sainrat, Jean Paul Bahsoun. Inter-task WCET Computation for A-Way Instruction Caches [A], Proceedings of International Symposium on Industrial Embedded Systems [C], 2008, 169-176.
131. Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-Related Preemption Delay Computation for Set Associative Caches [A], Proceedings of WCET [C], 2009.

致 谢

五年的博士生活即将结束，在这里我由衷的感谢五年来给予我无私帮助和关怀的老师、同学、朋友和家人。

感谢我的导师于戈教授。在工作中，于老师高瞻远瞩、治学严谨、作风务实，对学生要求非常严格。在他的言传身教之下，我们深刻的体会到了作为科研工作者所应具备的核心素质。在生活中，于老师为人率真、风趣幽默，常从细微之处关心我们的日常生活。于老师虽身居高位，但我们感受到的却是一位亲和、慈爱的长者。您的治学精神和人格魅力，给予我们的是一生的影响！

感谢我的副导师邓庆绪教授。从硕士阶段开始，我与邓老师的师生之情已有 7 年之久。这些年里，邓老师不仅在我的学术研究上给予了大量的指导，在为人处世方面也传授给我许多宝贵经验。在指导我建立起学术能力的同时，也让我在性格上变得更加成熟。在此向邓老师表示我最诚挚的感谢！

感谢王义老师。您在我博士学业最关键的时期给了我巨大的帮助，在我的前途发展问题上给予我无私的支持，这份恩情终生不忘！

感谢顾宗华老师。谢谢您在我博士最困难时期给予我的帮助，谢谢您那善良的品格所带给我的人生启迪！

感谢我本科的英语老师王文成老师。毕业多年，每次在校园里遇到，您都是那么慈爱的叫我的名字。在我博士最辛苦的阶段，您还打电话嘱咐我注意身体，让我倍感温暖。祝王老师健康长寿！

感谢软件所的鲍玉斌老师、朱靖波老师、申德荣老师、杨晓春老师、王大玲老师、董晓梅老师、李芳芳老师、赵志斌老师、张天成老师、张一飞老师、李晓华老师、林树宽老师、寇月老师、单吉弟老师。这些可爱的老师让我们的软件所变成一个温暖的大家庭。

感谢陈蕊、雷小凤协助我完成了第六章的部分工作。感谢同实验室的张轶、金曦和孔繁鑫。在我博士论文的冲刺阶段，你们帮我分担了很多实验室的工作，特此感谢。感谢国莹、雷晓凤、郭靖，谢谢你们在我最需要的时候所给予的无私帮助。感谢同实验室的王兵、谢利堂、郭英甲、王轶群、罗炎、陈卫涛、刘玮、金瑜、刘诗源、刘柄蔚、宁宝峰、张斌、谷传才等同学，感谢你们多年来对我工作的支持！更要感谢已经毕业的各届研究生师弟师妹，篇幅所限，恕我不能将各位的名字书写于此，谢谢你们从各方面所给予我的帮助和支持，与你们共同学习和生活的日子将是我人生最好的一段回忆。还要

感谢 DEEP WEB 实验室的寇月老师，RFID 实验室的谷峪、王艳秋，谢谢你们对我的每一次帮助。

感谢我那些亲如兄弟般的朋友：刘林波、王彤、李鸣扬、张弛、沈映涛、殷楠、董宁、韩宇、孙永兵、崔进、丁宁、韩陆等。在我最困难的时候，最苦闷的时候，是你们像家人一样听我诉说，像兄弟一样给我关怀。你们的关怀让我充满力量！

特别感谢我的师兄弟关楠。感谢这些年来，在每一次关键时刻上你给予我的经验和建议，感谢在我学业最困境的时期你带给我的希望和帮助，感谢一起打拼的日子里你给我的鼓励和关心！

特别感谢从高中时代开始与我共同学习 12 年的聂铁铮。能在一起共事这么多年，是难得的缘分。感谢你永远不变的那份兄弟情谊！

感谢我的父母、姨、姨夫、姐姐。谢谢您们对我生活最无微不至的关怀。在我遭遇困境的时候，您们比我还揪心；在我成功的时候，您们比我更开心。您们是我今生最大的财富，是我不断努力的动力源泉！

最后感谢人生中给予我最多的亲爱的姥姥，希望您在天堂看到孙子的成绩能够感到欣慰。

攻博期间发表的论文

1. **Mingsong Lv**, Nan Guan, Qingxu Deng, Ge Yu, Wang Yi. Static Worst-Case Execution Time Analysis of the uC/OS-II Real-Time Kernel [J], accepted to Frontiers of Computer Science in China, 2009. (EI 检索源，第一作者)
2. **Mingsong Lv**, Nan Guan, Yi Zhang, Rui Chen, Qingxu Deng, Ge Yu, Wang Yi. WCET Analysis of the uC/OS-II Real-Time Kernel [C], Proceedings of the 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC), 2009, Vancouver, Canada. (EI 检索源，第一作者)
3. **Mingsong Lv**, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, Jianming Zhang. A Survey of WCET Analysis of Real-Time Operating Systems [C], Proceedings of the 6th IEEE International Conference on Embedded Software and Systems (ICESS), 2009, Zhejiang, China, 65-72. (EI 收录: 20094212372971，第一作者)
4. **Mingsong Lv**, Zonghua Gu, Nan Guan, Qingxu Deng, Ge Yu. Performance Comparison of Techniques on Static Path Analysis of WCET [C], Proceedings of the 6th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC), 2008, Shanghai, China, 104-111. (EI 收录: 20091512017854，第一作者)
5. **Mingsong Lv**, Qingxu Deng, Nan Guan, Yaming Xie, Ge Yu. ARMISS: An Instruction Set Simulator for the ARM Architecture [C], Proceedings of the 5th International Conference on Embedded Software and Systems (ICESS), 2008, Sichuan, China, 548-555. (EI 收录: 20083911587221，第一作者)
6. **Mingsong Lv**, Ying Guo, Nan Guan, Qingxu Deng. RTNoC: A Simulation Tool for Real-Time Communication Scheduling on Networks-on-Chips [C], Proceedings of the 2008 International Conference on Computer Science and Software Engineering (CSSE), 2008, Wuhan, China, 102-105. (EI 收录: 20091211962974，第一作者)
7. **吕鸣松**, 张威, 张轶, 邓庆绪. 基于 ARM 和 VxWorks 的嵌入式开发平台设计与实现 [J]，计算机科学, 2009, 36(10), 35-38. (第一作者)
8. Nan Guan, Zonghua Gu, **Mingsong Lv**, Qingxu Deng, Ge Yu. Exact Schedulability Analysis of Global Scheduling on Multiprocessor Platforms by Symbolic Model Checking [C], Proceedings of the 11th IEEE International Symposium on Object-Oriented Real-Time Systems (ISORC), 2008, Orlando, USA, 551-555. (EI 收录: 20083411476194，

第三作者)

9. Qingxu Deng, Fanxin Kong, Nan Guan, **Mingsong Lv**, Wang Yi. On-line Placement of Real-Time Tasks on 2D Partially Run-time Reconfigurable FPGAs [C], Proceedings of the 5th IEEE International Symposium on Embedded Computing (SEC), 2008, Beijing, China, 20-25. (第四作者)
10. Zonghua Gu, Mingxuan Yuan, Nan Guan, **Mingsong Lv**, Qingxu Deng, Ge Yu. Static Scheduling and Software Synthesis of Datafow Models with Symbolic Model-Checking [C], Proceedings of the 28th Real-Time Systems Symposium (RTSS), 2007, Tucson, USA, 353-364. (EI 收录: 20083211446330，第四作者)
11. Nan Guan, **Mingsong Lv**, Qingxu Deng, Ge Yu. A Real-Time Scheduling Algorithm with Buffer Optimization for Embedded Signal Processing Systems [C], Proceedings of the 4th IEEE International symposium on Embedded Computing (SEC), 2007, 772-777. (EI 收录: 20074210874085，第二作者)

攻博期间参与的项目

1. 国家 863 项目（2007AA01Z181），面向可重构计算系统的实时调度问题与操作系统技术的研究，2007 年 8 月~2009 年 12 月，主要研究人员。
2. 教育部科技创新工程重大培育计划（706016），面向智能化装备的嵌入式平台开发及应用示范，2006 年 8 月~2009 年 12 月，主要研究人员。
3. 辽宁省自然科学基金（20082032），基于模型检测的片上多处理器系统实时性分析技术的研究，2008 年 5 月~2009 年 5 月，主要研究人员。

作者简介

吕鸣松，男，汉族，1980 年出生于河北省献县。1998 年考入东北大学计算机科学与技术专业，2002 年毕业，获学士学位。同年，保送本校研究生，从师于戈教授，从事实时嵌入式系统方面的研究工作，2005 年毕业，获硕士学位。同年考入东北大学研究生院，攻读嵌入式系统及应用专业博士学位，从师于戈教授，主要从事实时系统中最坏情况执行时间方面的研究工作。

2005 年起，一直致力于实时系统中最坏情况执行时间的研究工作。研究成果发表（已录用）在包括《RTSS》国际会议、《EUC》国际会议、《ISORC》国际会议、《ICESS》国际会议、《SEC》国际会议、《计算机科学》等国际、国内会议和国内杂志，其中第一作者被 EI 收录 4 篇。

作为主要申请人参与了国家自然科学基金“多核系统中实时调度策略的设计与分析技术的研究（60973017）”的申请和研究工作。作为主要参与人已完成和正在参与的科研项目主要有国家 863 项目（2007AA01Z181）“面向可重构计算系统的实时调度问题与操作系统技术的研究”，教育部科技创新工程重大培育计划（706016）“面向智能化装备的嵌入式平台开发及应用示范”，辽宁省自然科学基金（20082032）“基于模型检测的片上多处理器系统实时性分析技术的研究”等。

附录 A： μ C/OS-II 系统调用分析实验结果

表 A.1 μC/OS-II 系统调用分析实验结果

Table A.1 Experiment data of μC/OS-II system calls

系统调用	分析值	观测值	过度量	系统调用	分析值	观测值	过度量
Task Management (8)							
OSTaskChangePrio	8,134	6,847	18.80%	OSTaskCreat	15,813	13,724	15.22%
OSTaskQuery	4,265	4,035	5.70%	OSTaskDel	10,198	8,476	20.32%
OSTaskDelReq	1,975	1,868	5.73%	OSTaskResume	5,146	4,053	26.97%
OSTaskStkChk	3,201	3,014	6.20%	OSTaskSuspend	5,558	4,367	27.27%
Memory Mangement (4)							
OSMemCreate	3,423	3,335	2.64%	OSMemGet	1,749	1,740	0.52%
OSMemPut	1,780	1,773	0.39%	OSMemQuery	1,994	1,988	0.30%
Event Flages Management (7)							
OSFlagCreate	2,183	2,174	0.53%	OSFlagDel	7,836	6,638	18.05%
OSFlagPend	9,758	7,453	30.93%	OSFlagPendGet	1,056	1,052	0.38%
OSFlagPost	8,865	7,236	22.51%	OSFlagQuery	1,439	1,432	0.49%
OSFlagAccept	2,601	2,550	2.00%				
Mutual Exclusive Semaphore Management (6)							
OSMutexAccept	2,409	2,393	0.67%	OSmutexCreate	3,503	3,490	0.37%
OSMutexDel	12,029	11,499	4.61%	OSMutexPend	12,461	9,251	34.70%
OSMutexQuery	2,907	2,798	3.90%	OSMutexPost	11,575	11,092	4.35%
Semaphore Management (7)							
OSSemAccept	1,594	1,585	5.68%	OSSemCreate	2,728	2,651	2.90%
OSSemDel	8,548	7,437	14.94%	OSSemPend	8,472	5,467	55.00%
OSSemPost	7,344	6,252	17.47%	OSSemQuery	2,461	2,450	0.45%
OSSemSet	1,724	1,711	0.76%				
Time Management (5)							
OSTimeDly	4,348	3,303	32.55%	OSTimeDlyHMSM	16,199	6,507	248.94%
OSTimeDlyResume	5,363	4,272	25.54%	OSTimeGet	1,027	1,023	0.39%
OSTimeSet	996	992	0.40%				

表 A.2 μC/OS-II 系统调用分析实验结果（续）

Table A.2 Experiment data of μC/OS-II system calls (continued)

系统调用	分析值	观测值	过度量	系统调用	分析值	观测值	过度量
Message Queue Management (9)							
OSQAccept	2,251	2,116	6.38%	OSQCreate	4,188	3,658	14.49%
OSQDel	8,796	7,685	14.46%	OSQFlush	1,591	1,585	0.38%
OSQPend	8,658	5,903	46.67%	OSQPost	7,375	6,283	17.38%
OSQPostFront	7,375	6,283	17.38%	OSQPostOpt	11,363	6,816	66.71%
OSQQuery	2,960	2,946	4.75%				
Message Mailbox Management (7)							
OSMboxAccept	1,343	1,337	0.45%	OSMboxCreate	2,635	2,558	3.01%
OSMboxDel	8,548	7,435	14.97%	OSMboxPend	8,565	5,808	47.47%
OSMboxPost	7,440	6,347	17.22%	OSMboxPostOpt	11,428	6,721	70.03%
OSMboxQuery	2,461	2,450	0.45%				
Miscellaneous (8)							
OSInit	29,086	25,758	12.92%	OSIntEnter	286	279	2.51%
OSIntExit	1,619	1,490	8.66%	OSSchedLock	1278	1271	0.55%
OSSchedUnlock	2,115	1,740	21.55%	OSStart	999	992	0.71%
OSTimeTick	4,253	3,496	21.65%	OSVersion	283	279	1.43%

附录 B: μC/OS-II 禁止中断区间分析实验结果

表 B.1 μC/OS-II 禁止中断区间分析结果

Table B.1 Experiment data of μC/OS-II disabled interrupt regions

禁止中断区间	基本块	分支	WCET	禁止中断区间	基本块	分支	WCET
Task Management (15)							
OSTaskChangePrio	19	8	810	OSTaskCreat-1	7	2	624
OSTaskCreat-2	3	0	779	OSTaskResume	13	5	872
OSTaskCreateExt-1	7	2	624	OSTaskCreateExt-2	3	0	779
OSTaskCreateExt-3	5	1	779	OSTaskCreateExt-4	5	1	1,802
OSTaskDel-1	25	11	2,019	OSTaskDel-2	10	2	1,058
OSTaskDelReq-1	3	0	655	OSTaskDelReq-2	7	2	872
OSTaskStkChk	11	4	1,058	OSTaskSuspend	14	5	1,182
OSTaskQuery	15	4	1,058				
Memory Management (4)							
OSMemCreate	5	1	779	OSMemGet	5	1	1,089
OSMemPut	5	1	748	OSMemQuery	3	0	1,182
Event Flag Management (9)							
OSFlagCreate	5	1	1,027	OSFlagDel	31	9	903
OSFlagPend-1	28	10	748	OSFlagPend-2	19	7	1,895
OSFlagQuery	3	0	655	OSFlagPendGet	3	0	655
OSFlagPost-1	53	7	717	OSFlagPost-2	3	0	655
OSFlagAccept	18	6	748				
Mutual Exclusive Semaphore Management (6)							
OSMutexAccept	7	2	1,492	OSmutexCreate	7	2	810
OSMutexDel	29	10	903	OSMutexPend	10	2	1,948
OSMutexQuery	26	11	1,430	OSMutexPost	9	2	2,143
Semaphore Management (8)							
OSSemAccept	5	1	934	OSSemCreate	5	1	779
OSSemDel	24	7	903	OSSemPost	14	4	4,220
OSSemPend-1	9	2	872	OSSemPend-2	9	2	2,143
OSSemQuery	7	1	1,576	OSSemSet	8	2	934