第 5 章　软件时间分析方法

在开发嵌入式软件时，每个开发阶段都会遇到一系列与时间相关的问题、任务或挑战。例如，是否应该在项目的早期阶段，甚至在硬件可用之前比较不同的操作系统配置和通信概念？或者，软件的第一个版本已在运行中，但是仍然有一些尚未解决的零星问题需要调查？又或者，你或许需要确保软件时间保持稳定，以免在项目后期进行自动化测试时引起意外？可能你已经完成开发，但是在最终产品正常运行的过程中，需要通过分析组件来监控软件时间。

对于所有这些用例，可使用的软件时间分析方法存在很大差异。全面了解它们的所有可能性、优点和缺点以及使用它们的必要先决条件，对于高效执行时间分析至关重要。这里的“高效”是指以较低的成本和尽量少的精力实现正确的时间分析。如果没有本书中介绍的工具和方法，提供高可用性的安全嵌入式系统就是一件很难的事。

本章介绍在不同的开发阶段使用的各种软件时间分析方法，在后面的第 9 章“开发过程中的方法技巧”中，将使用这些方法。

除了介绍每种时间分析方法以外，本章还包含就相关方法对相关专家进行的简短访谈。

5.1　概览及在不同层面上的分类

图 5.1 显示了本章中详细介绍的软件时间分析方法。图中的纵轴表示可以执行时间分析的层级或粒度。以下三节将提供详细的描述。

5.1.1　通信层级

通信层级的时间通常与网络总线上的经过时间有关。其中，报文响应时间、带宽、利用率和缓冲区大小起到了重要的作用。而关注的重点是通信层级上端到端的时间（例如，从传感器到执行器），或者从软件中的一个事件到服务器上另一个事件的时间差。

产品之间：每当要开发的产品与外界交换数据时，时间方面也会产生影响。下面将以用于车联网或通过云端连接服务器的 Car-2-X 为例进行说明。其中一个用例是，在道路上检测到危险的车辆向跟随在后方的车辆示警。显然，此示警信息到达接收车辆的时间不应持续太久。

网络：这里的“网络”一词是指产品中（例如，机器或车辆中）控制单元和它们之间的总线所构成的网络。

ECU：对单个 ECU 进行时间分析意味着要查看安装在 ECU 中的处理器，并检查二者之间通过 SPI 进行的通信。如果 ECU 仅有一个处理器，则 ECU 层级也就是处理器层级。

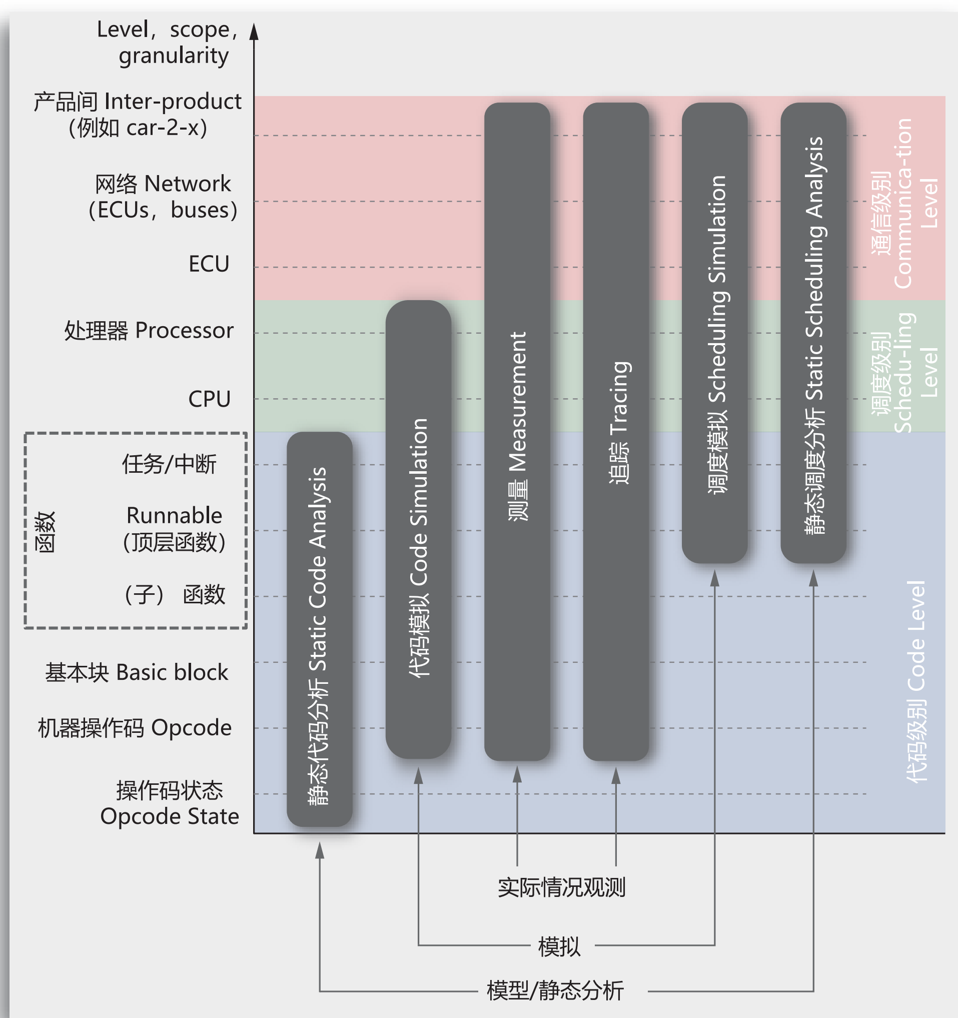

图 5.1 时间分析方法概览及其应用领域

5.1.2 调度层级

调度层级（RTOS 层级）的时间会影响操作系统所有与时间有关的行为。因此，调度层级也被称为操作系统层级或 RTOS 层级。调度层级的关键时间参数是任务的响应时间（参阅 4.1 节“时间参数”）。

处理器：处理器是一种半导体器件，上面有一个或多个 CPU、各种存储器和外设（参阅图 2.1）。CPU 会相互通信，通常也会相互干扰，例如在访问共享存储器时。此外，在不

同 CPU 上运行的代码部分在某些时候必须同步。

处理器层级是指执行多核调度的层级。这意味着提供多核调度的操作系统（例如基于 POSIX 的操作系统）必须“监控”整个处理器及其 CPU。例如，用户可以将线程的处理从一个 CPU 迁移到另一个 CPU。

但如果是更高的层级，则不由操作系统负责。例如，用户无法轻松地在另一个处理器上启动任务或在另一个处理器上启动线程。

多核任务时间方面的内容将在第 7 章“多核及多 ECU 环境下的软件时间”中进行更加详细的讨论。

CPU：CPU（中央处理器）是一种执行单元，一次只能处理一个线程（一串指令）。这里专门排除了超线程 CPU，因为嵌入式系统中很少用到这类 CPU。如果要执行另一串指令（例如中断），则必须先中断当前这一串指令。

CPU 层级是指执行单核调度（如 OSEK 所提供的调度）的层级。

5.1.3 代码层级

对代码层级的元素执行时间分析时，重点是元素的处理和所需的时间。代码层级的中央时间参数是净运行时间（CET，核心执行时间）。

由于中断等原因造成的打断在代码层级不予考虑。也就是说，如果在代码层级进行分析时（例如测量 CET 时）出现了打断，则应予以扣除。

函数：这里的“函数”一词用于指代所有与函数类似的结构，包括常规的 C 函数（如 `int giveMeANumber(void){ return 42; }`）以及 OSEK 任务或中断服务例程。

扩展此定义的主要原因是可以很好地表示层级，如图 5.1 所示（自上而下）。

TASK, ISR：AUTOSAR CP 或 OSEK 任务和某些中断服务例程均将接受调度，因此在运行时由 AUTOSAR CP 或 OSEK 操作系统组织。

Runnable（顶层函数）：AUTOSAR CP 任务一般会依次调用分配给它们的 Runnable。和任务一样，Runnable 一般也是 `void-void` 函数。

但是，即使是不使用 AUTOSAR 操作系统的应用程序也通常遵循“顶层函数”的概念，使用这些函数“完成”操作系统的任务。

（子）函数：顶层的 Runnable 或函数会调用函数本身，而后又会调用其他函数。

基本块：所有代码（包括函数）均可分成基本块。需要注意的是，基本块是一系列不会跳入或跳出的机器指令。因此，基本块的所有命令将无一例外地从第一个命令开始按顺序进行处理。

基本块已经在 2.2 节“代码执行”中做过解释。

机器指令码：机器指令码是指单一指令。大部分追踪和测量工具的粒度和精确度都止于此层级。比如这样的测量任务示例：测量在地址 X 执行指令和在地址 X+Y 执行指令之间的 CET。无法以比这更高的精确度解析测量或追踪结果。

操作码状态：如 2.6 节“流水线”(Pipeline）中所示，每个机器指令均分多个步骤处理，这

些步骤即操作码状态。在时间分析中，仅静态代码分析会考虑此层级的影响，有时代码模拟也会考虑。

5.2 术 语 定 义

在详细说明时间分析方法之前，我们先对一些在后续章节中起重要作用的术语进行解释。

5.2.1 追踪

追踪是指在存储器中记录事件并附加时间戳的过程。在以后的某个时间点，此追踪存储内容可用于重现原始事件发生时的情况。

以下将通过一些例子来进行说明。CAN 追踪使我们能够分析 CAN 总线上的通信状况或对其进行“重现”。在 CAN 总线上传输的每条报文都将与报文 ID、用户数据和时间戳一起记录。

如果以某种方式扩展 OSEK 或 AUTOSAR CP 操作系统，使其记录所有任务的激活、开始和终止以及任务 ID 和时间戳，则可以在稍后可视化并分析何时激活、运行、中断哪些任务等。

即便和嵌入式软件不搭边的一个例子，即游艇的航行日志条目，也可以说明追踪的概念。游艇日志上会记录每一次进出港口事件以及相应的日期和时间等，以后可通过此日志追踪游艇的航行路线。此示例也很有帮助，因为它可以清楚地表明追踪数据通常只是实际发生情况的摘录。日志中的信息只能大致重现游艇的航行路线。条目中并不会显示游艇在各港口之间的具体航行路线。

我们将在后文中更详细地介绍调度追踪，即将任务和 ISR 或进程和线程随时间的变化可视化。在许多方面，调度追踪与示波器相似，本书的后续章节将多次使用这种类比。调度追踪将记录任务状态并在时间轴上显示任务状态，而示波器记录的则是电压。

最后，要注意避免混淆。术语“追踪”和“可追踪性”虽然具有相同的词根，但含义不同。前文已经解释过追踪的含义。而可追踪性则描述了在开发过程中追踪不同项目的能力。例如，测试结果应与测试关联。而测试本身则应与测试的功能关联，而该功能又与构成开发基础的需求关联。

5.2.2 分析、时间测量和（再次）追踪

分析（Profiling）是指访问运行中嵌入式系统或模拟的时间参数的过程。图 5.2 显示了两种分析方法，其中一种方法是执行运行时间测量，以直接确定所需的时间参数；另一种方法则是通过追踪间接进行。在追踪时，最初仅收集追踪数据，然后便可从中提取时间参数。

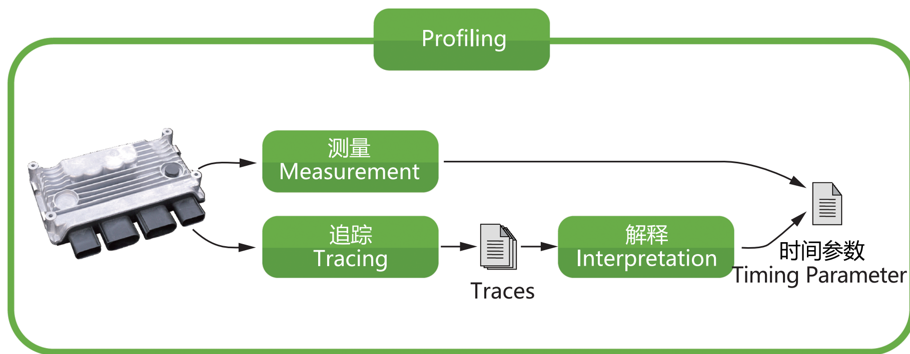

图 5.2　用于获取时间参数的两种方法

5.3　静态代码分析

这里的静态应理解为“离线”，即在分析时，不实际运行被分析的软件。静态代码分析器有很多，但用于分析软件运行时间行为或软件时间的很少。此外，还有分析堆栈需求的工具、尝试检测软件错误的工具以及检查是否符合编码准则的工具。

在下文中，我们将探讨侧重于软件时间的静态代码分析。更准确地说，我们将研究如何确定特定代码片段（例如函数）的可能最大内核执行时间。可能的最大 CET 是指 WCET（最坏情况执行时间），实际应表述为 WCCET（最坏情况核心执行时间）。在多数情况下，静态代码分析还能确定 BCET （最佳情况执行时间），但这并无太大意义，因为它对于软件的安全性和可靠性并不是很重要。

静态代码分析主要有两种类型，一种是从源代码开始，另一种则是从可执行文件开始。前者在学术环境之外几乎没有任何实际应用，因此，下面将只介绍基于可执行文件的分析。

5.3.1　基础功能和工作流

图 5.3 显示了执行静态代码分析时有哪些相关的数据，同时也显示了整个的工作流。首先，静态代码分析读取可执行文件并将其反汇编，即将二进制机器代码转译为汇编程序指令（另请参阅 1.3.5 节“汇编器”）。通过反汇编的代码，可以推导出控制流和函数调用树。所谓函数调用树，顾名思义，就是它能告诉你哪个函数调用哪些其他函数。此外，此分析还能确定最大循环迭代次数。

收集的数据现在将合并在一起，可能的最大运行时间将沿着控制流累加。可执行文件包含所有机器指令和数据的存储地址，这就是为什么此分析甚至可以考虑缓存和流水线对运行时的影响。为此，除了存储器配置（如指定闪存访问的等待状态）外，此分析还需要非常精确的处理器模型。在很多情况下，可使用处理器厂商的 VHDL 或 Verilog 数据创建此模型。VHDL 是指超高速集成电路硬件描述语言，可被视为作用于实现处理器的源代码。

如果所有时间数据均以秒或纳秒为单位，则分析时还需要有关处理器时钟的信息。这是由所用晶振和处理器时钟单元的配置决定的。

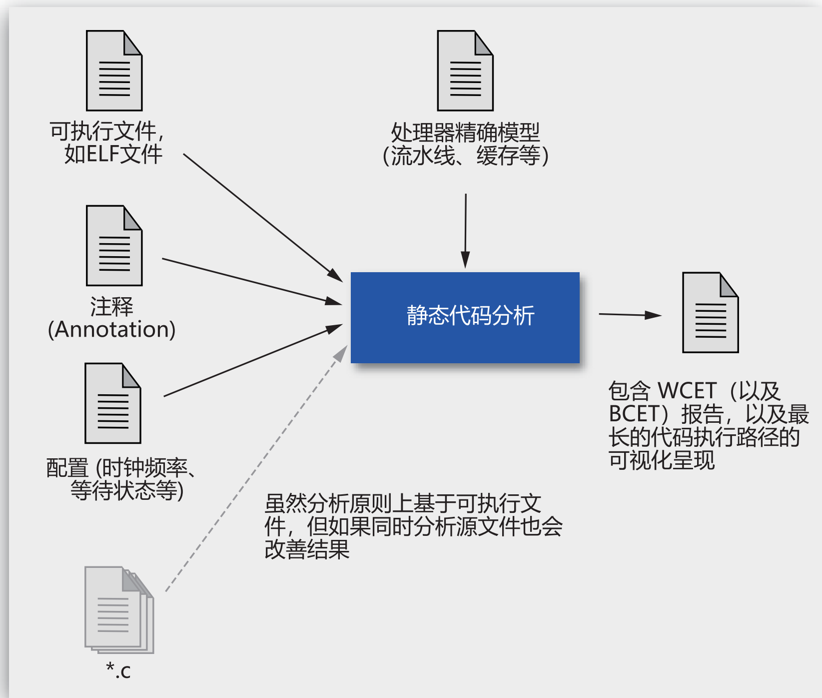

图 5.3　用于 WCET 计算的静态代码分析工作流

最后应该注意的是，在大多数情况下，实际 WCET 并不能在有限的时间内计算出来，但是可以计算出一个保证大于 WCET 的值 X。因此，分析的结果总是在安全的一侧，可以理解为安全上限，请参阅图 5.4。

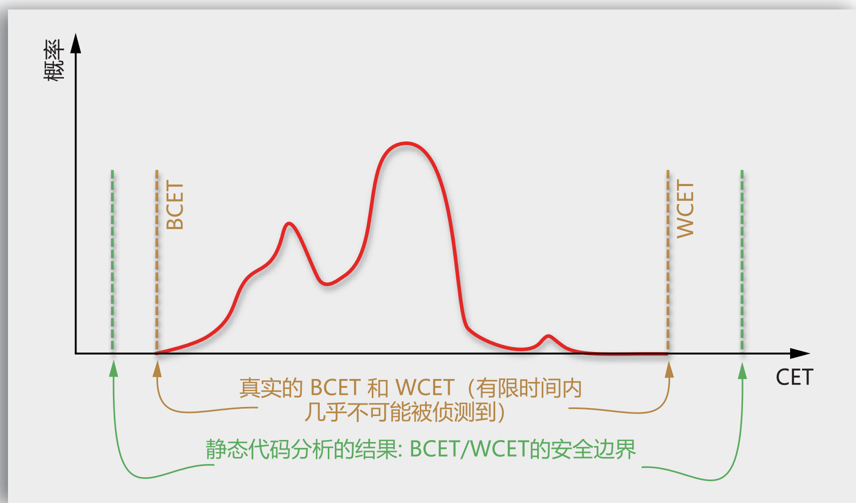

图 5.4　某函数（具有 BCET 和 WCET）的 CET 分布

5.3.2　用例

实际上，当需要确定最坏情况下的时间时，就必须要进行静态代码分析。某些安全性标准将静态代码分析视为“强烈推荐”的分析方法（参阅第 11 章“功能安全, ISO 26262”）。

抽象解译是适用于大部分情况的一种基本静态分析方法。它是一种形式化的方法，当应用于 WCET 分析并正确实现和配置时，它可以被认为是 WCET 的数学推导。

至关重要的是，分析结果必须完全独立于所使用的任何测试变量，这样可确保无须考虑这些变量导致最大运行时间的因素。

由于静态代码分析使用未执行的现成可执行文件，因此可以在不考虑硬件可用性的情况下进行分析。

另一个用例是构建过程中的自动 WCET 验证。每次编译软件时，都会进行静态代码分析，以检查某些函数的 WCET 是否超过预定义的限制，或者 WCET 相比于上一个软件版本的增量是否超过 $X\%$。

如果评估足够详细，则静态代码分析也可以用于运行时优化。

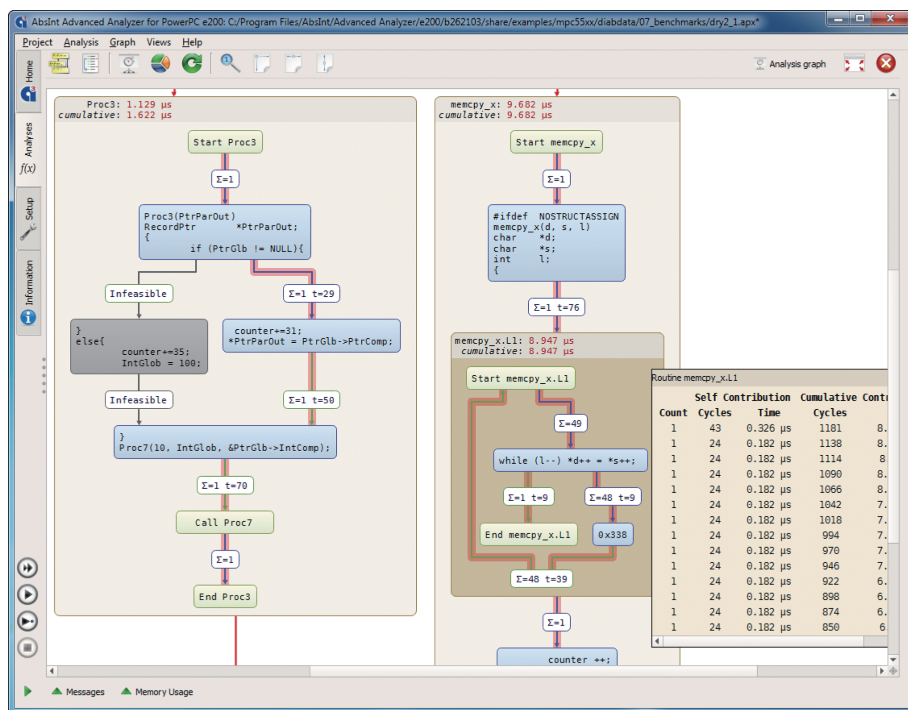

图 5.5　两个函数的 WCET 路径（工具：AbsInt[13] 的 aiT[12]）

图 5.5 显示了在 aiT[12]（AbsInt[13] 开发的一种静态代码分析工具）中分析的两个函数的结果，左侧为 `Proc3`，右侧为 `memcpy_x`。蓝色方框中显示了源代码片段。路径上的白色方框表示相应路径的最大执行次数。如果可执行文件中包含死代码（即任何情况下都无法执行的代码），则相应代码块显示为灰色而不是蓝色，并且相应路径上显示“Infeasible”（不可行）。最长执行路径（函数的 WCET 路径）以粉色突出显示，并有蓝色或绿色箭头（有

绿色箭头的部分在粉色背景下显示为橙色)。蓝色箭头标记了常规执行路径，绿色箭头标记了执行条件跳转的路径，例如在满足跳转条件时。此外，aiT 还提供了有关该函数各个代码部分为 WCET 贡献了多少时间的详细信息。

5.3.3 静态代码分析的限制

静态代码分析实际上并不像看起来那样简单。即使在构建函数调用树时，此分析通常也会遇到在没有外部帮助（例如，来自用户的帮助）的情况下无法解决的问题。

5.3.3.1 间接函数调用

间接函数调用（即将要调用的函数作为参数指针传递给调用者）可能很难分析。在某些情况下，静态分析将无法确定此参数可以取哪些值。在这种情况下，函数调用树并不完整。静态分析“知道”有更多函数要添加到其中，但是具体涉及哪些函数仍处于未知状态。此时需要在调用树上的这个点放置一个红色问号，而不是分出另一个分支。

5.3.3.2 递归

递归（即函数调用自身）的情况与此类似。这里有一个有趣的问题：递归将达到多大的深度，即该函数调用自身的最大次数。

5.3.3.3 循环上界

即使函数调用树完整，分析仍然会达到某些循环的极限，并且无法确定最大循环迭代次数。但是，计算 WCET 需要此数据。与其使用实际的循环上界，不如使用一个可能过大的值进行分析，从而导致明显的高估。

5.3.3.4 注释

用户必须通过提供其他信息来手动阐明这三个“绊脚石”(未解析的间接函数调用、递归以及错误识别的循环次数上限)。也就是说，必须注释（Annotation）代码。

小贴士： 通过要求负责创建代码的人员提供必要的注释，可以非常轻松地解决这种问题。例如，如果某家软件公司交付了一组包含间接函数调用的函数，则除了代码以外，还必须要求该供应商提供注释文件（类似于规范），这些文件将清楚地说明每次间接函数调用可能有哪些调用目标。

德国的一家汽车行业大型供应商在几年前就引入并实施了这种要求供应商完全注释其所交付代码的方法。

5.3.3.5 运行模式和互斥代码

对于已经正确识别或注释了所有间接函数调用和递归以及所有循环边界的应用程序，通常仍需要附加注释。

应用程序一般都能支持不同的工作模式。以飞机作为类比，这些模式可能是“在地面上”“飞行中”和“控制单元更新”。静态代码分析时各个不同工作模式的代码部分并不总是以互斥的方式彼此分离，至少从静态代码分析的角度来看并非如此。

当详细检查 WCET 高得出奇的代码时，开发人员会发现路径中存在互斥的代码。现在，他们有两个选项：要么重写代码，以便静态分析发现两个部分不能在同一路径中；要么必须添加其他注释。

5.3.3.6　高估

即使可以完全完成分析并注释应用程序工作模式，分析结果通常 WCET 仍会出乎意料的高。造成这种情况的原因之一可能是高估，即报告的 WCET 上界与实际 WCET 之间的差异很大。

还应注意的是，高估的时间也会被计入。例如，如果你要执行具有 500 个 Runnable 的 AUTOSAR CP 任务，并想确定该任务的 WCET，则分析时将选择通过所有这 500 个 Runnable 的最长路径。然后就是一个该值是否仍然相关的概率问题了。但是，WCET（或者更普遍地说，图 5.4 中所示的该任务的曲线）的发生概率无法通过计算得出。

如果这一点可以实现，则不必根据分析报告的上界来调整系统；相反，可以使用与所需概率相对应的 CET 值。

但是，在实践中并不会这样做，因此我们要么必须忍受这些高估，要么以相应较低的使用率运行系统，要么使用其他分析方法。

5.3.3.7　中断、多核和暂时性错误

如上所述，静态代码分析是一种代码层级的方法，因此不涉及任何调度方面。存储器接口上的中断或访问冲突（多核设备上经常发生）将不予考虑。

下面将用示例来进行说明。使用静态代码分析，将函数 F 的值 x 确定为 WCET 的上界。假定在实际执行中满足达到 WCET 的所有先决条件，即启动该函数时缓存和流水线将处于最差状态，且该函数使用的所有数据和参数都具有最长路径所需的值。在执行过程中，达到了 WCET。根据静态代码分析方法，此执行过程不会出现任何中断或冲突。

现在假设此执行过程被 ISR 中断。中断服务例程位于缓存中并不存在的存储区中。因此，必须将代码以及 ISR 使用的数据加载到缓存中，这样会覆盖缓存中函数 F 的代码和数据。

处理 ISR 后，函数 F 将继续执行。当然，ISR 的 CET 将从该函数总执行时间（GET）中减去，即 ISR 的执行时间将被扣除。即便如此，缓存现在仍处于“较差”状态，并且在继续执行函数 F 时会出现额外的延迟。当函数 F 执行完成时，其实际执行时间将超过静态代码分析所指定的上限。

原则上，当另一个 CPU 在执行函数 F 期间访问共享存储器并因此延迟其执行时，也会发生非常类似的情况。通过这种方式，可能会出现相当大的延迟，尤其是在访问数组时。但是，即便存在隔离的存储区域，访问共享内部地址和 Crossbar（图 2.1）等数据总线时仍会发生冲突。

为确保完整性，静态代码分析也将忽略暂时性错误（参阅 7.1.3 节“锁步多核”）。

这些限制在实践中意味着什么？结果可靠的真实 WCET 分析仅适用于不会中断的函数，中断、异常和任务抢占等必须予以排除。对于多核系统，几乎不可能进行可靠的 WCET

分析。

但是，它仍不失为多核环境中的一种有效分析方法（参阅 5.3.2 节“用例”中的关键词“构建过程中的自动 WCET 验证”）。

或许也可以摆脱这种严格形式化的方法，通过使用追踪数据[14] 来计算可能的最佳上界。

5.3.4 静态代码分析专家访谈

以下访谈旨在使静态代码分析的主题更加完整，并从另一个角度进行探讨。工具 aiT 和 TimeWeaver 均是 AbsInt[13] 公司的产品，该公司由 Reinhard Wilhelm 教授于 1998 年联合创办。

Peter Gliwa：静态代码分析是一个广阔的领域。我们在这里仅探讨用于确定 BCET 和 WCET 的分析方法。那么简单地说，它的工作原理是什么？

Reinhard Wilhelm 教授：WCET 分析很难用一句话来描述。首要的问题是机器指令执行时间可能存在巨大的差异，具体取决于执行状态，例如缓存内容。对于程序中的每个点，aiT 会计算所有可能执行状态的高估近似值。为了能够预测某个内存块上的缓存命中，该内存块必须处于所有已计算的缓存状态。

aiT 的第一步是通过可执行机器程序重建控制流程图。然后，aiT 将确定上述执行状态的高估近似值。借助于此，aiT 可以安全地高估程序中所有机器命令的执行时间。最后，必须确定此基础上通过程序的最长路径。

Peter Gliwa：有哪些核心用例？

Reinhard Wilhelm 教授：所有安全关键硬实时系统都是。其中大部分都无法通过飞行时间法（Time-of-Flight）测量来确定 WCET，因为这个过程十分复杂，需要测量大量用例，因此监管机构并不认可这种方法。

aiT 通常还应用于非关键型应用程序，因为不需要构建测试用例和测试输入，从而节省了大量的时间和精力。

当在硬件上进行测量的成本过高或无法进行测量时，我们的 TimingProfiler 产品还将使用该技术来实现开发初期的代码优化。

Peter Gliwa：学术界已经就这个话题讨论了很多年，并且进行了深入的研究。但为什么这种分析方法至今还没有在嵌入式软件开发过程中广泛使用呢？

Reinhard Wilhelm 教授：关于静态运行时分析的第一批出版物在 20 世纪 80 年代后期陆续发表，但那时都是针对具有恒定指令执行时间的架构。到了 90 年代，开始首次使用带有缓存、流水线、预测等功能的架构，其指令执行时间取决于执行状态。我们已经解决了由此产生的 WCET 分析问题，并实现了一种解决方案。这种技术在意识到这一问题的用户群体中应用非常广泛。剩下的那些用户则陷入了一种错误的安全感，并依赖一些根本不健全的方法。

Peter Gliwa：不久之后，能够显著提高平均计算能力的处理器功能就从桌面系统迁移到