代　号	10701	学　号	0910092006
分类号	TP311.5	密　级	公开
U D C		编　号	

题（中、英文）目 基于 UEFI 技术的 BIOS 系统分析及其 API 性能测试研究

Analysis of BIOS System based on UEFI Technology and Research on its API Performance Test

作 者 姓 名 华东 学校指导教师姓名职称 武波 教授

工 程 领 域 软件工程 企业指导教师姓名职称 令杰 高工

提交论文日期 二 O 一二年六月

西安电子科技大学
学位论文独创性（或创新性）声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。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做了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切的法律责任。

本人签名：_____ 日期_____

西安电子科技大学
关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。同时本人保证，毕业后结合学位论文研究课题再撰写的文章一律署名单位为西安电子科技大学。

（保密的论文在解密后遵守此规定）

本学位论文属于保密，在　　年解密后适用本授权书。

本人签名：_____ 日期_____

导师签名：_____ 日期_____

摘要

Unified Extensible Firmware Interface 是下一代计算机固件接口标准。UEFI 是用面向对象来设计，采用模块化设计思路来组织驱动和应用程序的，模块之间用标准协议通信。UEFI 的提出目的是更新传统 BIOS，它是能够用高级语言和现代的软件工程方法设计的接口程序。UEFI BIOS 的广泛推广，使得更多的 OEM、个人，都可以自行实现开发自己的专用软件。但目前在测试这些软件的性能方面，没有现成的测试软件。

论文文详细研究了 TIANO 系统框架及 EDK 模块，并在 EDK 下实现了具体的应用程序，旨在深入介绍新一代 UEFI 的所独具的技术特点。

论文设计完成了一款 PEI 阶段的底层 API 性能分析测试模块，成功地在 UEFI SHELL 下实现了对 MDE Library 的分析。最后通过对 RUNTIME 阶段底层 API 的研究，实现了在 LINUX 下底层 API 的实时调用，且编写应用程序代码获取底层 TimeService 提供的准确时间，进而直观地测试了底层 API 的性能。

关键词： UEFI　TIANO　EDK　API　NT32　LINUX

Abstract

Unified Extensible Firmware Interface is the next generation firmware interface standard. It is object oriented and consists of drivers and applications that communicate with each other via standard protocols. UEFI is subjected to replace legacy BIOS and become a firmware interface that implemented with high level programming language and modern software engineering methodology.

This paper first studied in detail TIANO framework and EDK module, and the EDK under a specific application, designed to in-depth description of unique technical features of a new generation of UEFI.

This paper designed and completed a PEI phase of the underlying API performance analysis test module, the analysis of the MDE Library on UEFI Shell. Finally, after the research the RUNTIME phase of low-level API, real time LINUX underlying API call, the paper applied application code obtain underlying time service provide accurate time, and then visually tested the performance of the underlying API.

Key words: UEFI TIANO EDK API NT32 LINUX

目　录

第一章　绪论

1.1 UEFI 概况和技术特点

作为连接操作系统与硬件体系之间的桥梁，传统的 BIOS (Basic Input/Output System) 为 PC 的发展做出了重要贡献。BIOS 是硬件与软件程序之间的转换器，也是它们之间的接口，BIOS 是一段程序，以满足硬件相对需求，同时具体地执行软件需要硬件提供的服务。它的作用主用体现在操作系统初始化，功能还包括对系统配备的检查以及连接操作系统与硬件。由于本身是用汇编语言编写，BIOS 是以 16 位编码、寄存器的参数作为调用方式、静态的链接，及以 1MB 为上线的固定内寸编址形式服务了计算机应用的初始阶段。CPU 更新了一代又一代，但加电后启动 16 位的实模式一直延续至今。虽然后来 PC 界应用了大实模式，但它只适用在系统固件的范畴。这种落后的运行过程，迫使了 Intel、AMD 等在开发它们新款的 CPU 的同时，需要促使系统系能降低的被动兼容的形式。开发者对开发过程中的体验也是非常的困惑，编写代码的过程过于复杂。在这样的前景下，Intel 公司提出了一个新的方案，即 EFI 的技术。

2001 年，Intel 公司向 PC 界展示了新一代的 BIOS 程序 EFI，并将 EFI 技术应用于安腾服务器的平台上。EFI 的提出，旨在替代传统的旧的 BIOS 程序。旧的 BIOS 程序都是采用汇编编写，面对新的来自各方的需求显得无能为力。在这方面，EFI 技术具有明显的优点，它采用了模块化的、动态链接的以及类似于 C 语言的风格对应的参数堆栈的传递方式来构建系统，相对汇编来说，更容易实现。并且，EFI 的驱动是不依赖于 CPU 的类型，这大大提高了 EFI 技术的 BIOS 的兼容性能，保证了其在不同 CPU 的架构上的畅通。EFI 是 Intel 定的标准的、可扩张的固件之间的接口规范。这样它就有别与传统的 BIOS 是用固定、基于经验、缺乏文档参考的实际标准。

从体系架构上看，EFI 可以看成是一个简单的 OS，它的目的是连接硬件与操作系统的枢纽。跟传统的 BIOS 不同，EFI 技术添加了图像显示的驱动，使之能提供对应的清晰的色彩图形接口。另一个区别是 EFI 技术的 BIOS 是采用高级的 C 语言编写，丢弃了传统 BIOS 的晦涩难懂的 16 位汇编语言。这就使得可以有更多的个人、厂家加入到开发过程中，对 EFI 技术的发展有着相当的推动作用。当前，EFI 技术的应用进一步的推广到了个人 PC 机的上面，苹果公司在其 x86 架构的 PC 机上也采用了最新的 EFI 的技术[1]。同时，EFI 技术也在向电子商务、家电领域不停的延伸。例如，因为 EFI 技术的深入应用，用户可以在没有进入 OS 的情况下就

实现网络应用。无疑，安腾服务器平台对 EFI 技术的成功应用，让开发者们看到了 EFI 技术带来的便利。2004 年，在整个 IT 界的共识下，Intel 公司把 EFI 规范交给由微软、HP、AMD 等公司组成的联盟管理，并且开放了其实现的核心代码。在这个背景下，EFI 也更名成 UEFI(统一的可扩展固件接口)。UEFI 的联盟负责对 EFI 技术的开发、维护及管理工作。

UEFI 的联盟是由 Intel、Hp、微软等公司在 2005 年发起并成立的阻止。组织定期地召开对于 UEFI 技术研讨大会。联盟现在已经发展到了 86 个企业的成员。所有企业成员可以分成三类：推广的、接受的及贡献的。他们各自分担不同的职责和义务。UEFI 的联盟工作组四个，他们是负责规范的工作组 USWG、负责测试的工作组 UTWG、负责平台的初始化的工作组 PIWG 以及负责联络的个成员的工作组 ICWG，他们通过对业内进行大量的、多样的教育和推广，让 IT 的各个成员更深切地了解、感受 UEFI 技术[2]。

目前，UEFI 联盟最新发布两个刚形成的规范，即 UEFI 2.3 以及新 PI(初始化平台规范)v1.2。规范 UEFI2.3 定义了固件与高级的应用软件和操作系统之间的程序接口。v1.2 规范保证了不同企业开发的固件的可兼容性和互操作性 [3]。

2006 年，UEFI 的联盟技术大会第一次在中国的南京举行。联盟的常务会执行总裁魏东的演讲，让国内的众多业界厂商第一次全面地认识了 UEFI 联盟以及他们所应履行的职责。

UEFI 的联盟监督了该技术的全程开发。该组织负责人表示，当初设计出的 BIOS 预计仅供 25 万台机器使用，而这一数字早已被超越。当 64 位计算变得更为常见，改装机性能远远超越普通台式和笔记本电脑时，BIOS 早已跟不上现代社会快速的发展节奏。为了适应现代个人电脑，BIOS 往往要在开机前花上一段时间进行预热，因此，配置 BIOS 的电脑在开机时会耗费上许多时间。而新型 UEFI 技术，全称统一的可扩展固定接口， 是一种详细描述全新类型接口的标准。这种接口用于操作系统自动从预启动的操作环境，加载到一种操作系统上，从而使开机程序化繁为简，节省时间。UEFI 将提供一个明确的在操作系统与开机时启动固定于硬件中的软件平台之间的接口规范，此外还支持一种用于初始化插卡的独立于计算机架构的装置。

从而，基于 UEFI 的 BIOS 系统得到了越来越多的应用，传统的 BIOS 必将逐渐被取代。

1.2 UEFI 对比与传统 BIOS 的优势与局限性

20 世纪的 80 年代 BIOS 技术开发并应用了，该技术主要应用在 IBM 的 PC/AT 和 PC/xT 系统中。BIOS 固化在主板的 ROM 中，可以是 EEPROM。PC 启动后，

BIOS 首先被运行，BIOS 的代码接管、负责 PC 的硬件识别、检测和初始化的工作。BIOS 程序抽象出不同的 PC 平台硬件组织，这样就可以在基于 X86 的架构平台上的不同 PC 运行应用程序。

随着硬件的设备在种类和功能的不断发展，现代的计算机通常把 BIOS 程序代码分离到主板 ROM 和其它存在在硬件的适配器 Option Rom 里。主板的 BIOS 程序包括了基本硬件的识别、代码的检测，同时负责了加载在 Option Rom 中特定的硬件驱动的代码。在 Option Rom 中对应的代码的开发者可以是生产商本身。

传统的 BIOS 采用软件抽象方式向应用程序和操作系统提供统一来访问系统的资源对应接口，这样的特点使之得到了广泛应用。随着硬件的不断更新发展，传统的 BIOS 遇到了很多瓶颈，就算用 Option Rom 的指定方式也不能得到有效的解决。对应的问题如下。

第一，传统的 BIOS 一直定义了一些与操作系统无关的应用接口，这些接口依赖中断的形式组织 BIOS 软件结构，非常依赖 X86 提供的中断模型。换言之，该 BIOS 的软件接口不是独立硬件的。

第二，传统的 BIOS 运行在实模式 Real Mode 下，寻址的能力是 1MB。虽然在理论及技术的层面上可在 Real Mode 下提供更大寻址空间，但技术的适用使程序的耦合度大增。Option ROM 对应的地址内存空间也仅限在 1MB，这就限制了 Option Rom 的驱动程序扩展功能。1986 年 Intel 公司发布了 32 位的 CPU，当前的 CPU 总线是 64 位，这样一来传统的 BIOS 寻址能力造成了资源的浪费。

第三，传统的 BIOS 是采用汇编编码，软件工程的实施和管理效率比较低。开发者通常采用一定的汇编语言的技巧编写 BIOS 程序，所以汇编语言编成的 BIOS 程序的扩展性及维护性比较差。矛盾的是，硬件的快速发展对 BIOS 程序的扩展性及可维护性提出了更高的要求。

在技术遇到瓶颈的情况下，Intel 公司开发了 Extensible Firmware Interface 以替代传统的 BIOS。最新的 UEFI 技术标准已成了国际的开放标准。 UEFI 的规范平台固件与操作系统的链接定义了接口，该接口是由数据表来组成，包括平台的相关信息及对 OS 和载入的程序可以有效启动并满足实时的服务请求。这个过程定义了完整的、先进的环境用来启动 OS 和导入应用程序。UEFI 的 BIOS 与传统 BIOS 相比而言，最明显的优点是它具体有模块化、可扩展的，规范化的特点[3]。UEFI 的 BIOS 程序采用高级 C 语言来编写，在启动的过程中可以实现很多的扩展功能。UEFI 的 BISO 作用像一个 OS，在 BIOS 程序进入到 DXE 阶段的时候，用户可以执行 UEFI Shell 工具作为用户与硬件系统之间的交互接口。这样就可以在进入到操作系统之前就可以实现信息检测、网络应用、软硬件的调试等扩展的功能。但是，到 UEFI 技术 BIOS 完全取代传统 BIOS 还有很长的时间，在 DXE 阶段的 UEFI Shell 环境下开发的程序只有一小群人在应用，推广这种底层应用程序的必要性可

想而知[4]。UEFI 技术的 BIOS 为开发者提供了结构层次清楚的、模块化的、网络应用的、高级语言开发等优点的情况下，也带出了很多的安全问题，如下所列。

1. UEFI 的 BIOS 技术在进入到 DXE 阶段引入了网络应用，比如 UDP、TCP、MTFTP 网络功能支持。这个过程同样对网络的安全带来了一定的隐患，将带来另一个重要的网络安全问题。

2. UEFI 的 BIOS 技术的扩展性能提供了更多接口给开发者使用，其核心的代码开源性使得黑客们不需要反汇编的代码，仅仅通过阅读就可以找到其核心代码的漏洞和缺陷。

3. UEFI 的 BIOS 技术采用了高级语言编写开发，相比传统 BIOS 应用的汇编语言，提高了 UEFI BIOS 程序的接口调用效率，尤其对与 DXE 的阶段 UEFI Shell 调试接口来说，可能引起黑客们新一轮的针对 BIOS SMM (System Mangement Mode）的攻击。

4. UEFI 的 BIOS 技术新颖的同时有其复杂的不同阶段划分，每个阶段有其单独的核心，比如 DXE 和 PEI 阶段都是由其对应的分发器来负责的。对于普通 PC 的平台，尤其对嵌入式的设备来说，阶段的衔接会造成系统的启动速度比不上传统 BIOS 及 Bootloader[5]。

1.3 新一代的固件架构与 UEFI 结构分析

EFI 规范是平台固件与操作系统之间的接口。规范所定义接口内容包括平台信息数据表及启动后所提供的服务。启动服务包括对不同设备，总线文件服务上支持的文本和图形控制台以及运行的服务。EFI 还定义了一个小型的命令行处理环境，在没有启动进入操作系统的情况下，用户可以选择选择进入 EFI Shell 的命令行处理程序，这样一来就抛弃了原有的 BIOS 所承载的只有进入了操作系统才能处理各种应用程序的弊端[6]。在这里要说明的是 shell 俗称壳，用来区别于核的概念，指的是“提供用户使用界面”的软件，是命令解析器。它类似 DOS 下的提 command.com。EFI Shell 是 EFI 特有的命令解析器。

EFI 启动的管理器装载于操作系统上，不需要专有的启动装载器的机制进行辅助。EFI 扩展可连接在 PC 上，用固定存储设备进行装载。Framework 定义为固件架构，其为 EFI 固件的接口实现，可以替代传统 BIOS 程序 [7]。

整个 Framework 是模块化、层次化的结构，用 C 语言编写实现，它的结构如图 1.1。

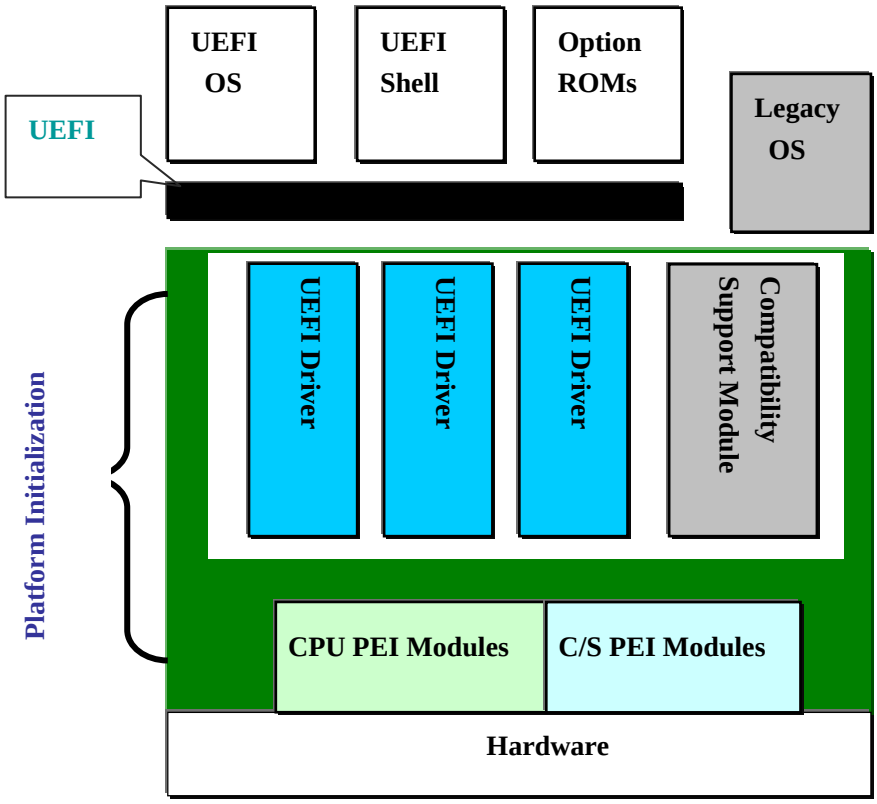

图 1.1 Framework 结构图[8]

从图中可以清晰地看到 UEFI 不是对硬件来进行操作，事实上 UEFI 通过调用底层的 SAL 和 PAL 来调控硬件。让 UEFI 引导服务（Boot Service）及运行服务（Runtime Service）去调用 SAL 及 PAL 相应的功能。UEFI 另外提供了系统调用的工具软件，这些就是 UEFI 技术 API 的调用。概括地说，UEFI 技术在 Firmware（平台固件）模型框架里起到一个连接作用、达到承上启下的效果。

UEFI 模块与其余各模块的联系区别如图 1.2，UEFI 引导服务及运行服务的过程采用调用 SAL 及 PAL 相应的功能。

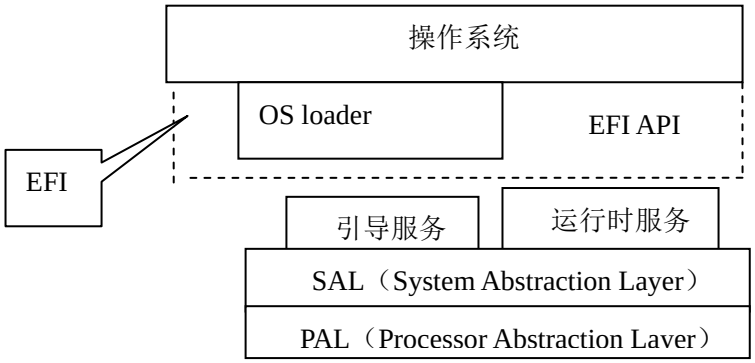

图 1.2 UEFI 与其余模块的联系[9]

不同于传统 BIOS 的实现，Tiano 基于现代软件体系设计的思想，按不同阶段初始化系统平台，将从上电到启动到进入操作系统流程程分为四个重要阶段，即

SEC 阶段、PEI 阶段、DXE 阶段及 BDS 阶段[10]。

第一阶段即安全阶段（Security， SEC），上第一个执行的阶段。SEC 阶段支持系统执行最初操作码，来确保所选平台的固件映像完整。此阶段通常要硬件的支持，而新一代的 CPU 及芯片组都支持相应的功能。该体系提供扩展的接口以支持新功能的添加。

由于高级语言 C 语言的支持，此阶段需要设计一个任务，即寻找初始化过内存，让 C 语言代码能够运行在所给的系统上。PEI 阶段及 DXE 阶段划分也参照此原则。

第二个 PEI 阶段，负责简化工作寻找初始化的内存，很多芯片及其它硬件初始化过程一直到驱动程序的执行环境DXE阶段启动并运行之后开始执行。最早PEI阶段代码更倾向用适合的机器汇编代码来编写。发现内存之后，PEI 的阶段就用状态信息描述平台资源图。再初始化内存，最后跳到 DXE 的阶段为止。PEI 阶段到 DXE 阶段之间的转化是单向过程。程序进入到 DXE 阶段之后，PEI 阶段的代码停止执行，此时 DXE 阶段成为了设备很齐全的环境。

DXE 的阶段是系统初始化的执行阶段。DXE 的阶段里有些组件，包括了 DXE 分派程序、DXE 核及一段 DXE 驱动程序。DXE 核提供启动的服务，即运行服务及 DXE 的服务。而 DXE 的分派程序主要负责按正确顺序发现以及运行 DXE 的驱动。DXE 的驱动程序则主要负责初始化芯片组、平台组件及处理器。同时也为系统提供抽象软件服务、控制和启动设备。它们一起负责初始化的工作，为操作系统提供相应的服务[11]。

BDS（Boot Device Selection）的阶段与 DXE 阶段共同创建软件控制台，尝试从待命的设备启动 OS。BDS 阶段是将控制权交于 OS 的最后阶段，显得很关键。在 BDS 的阶段里，OEM 可以通过用户的界面来指定 DXE 阶段对应的系统。

基于 UEFI 技术的 API 性能测试设计实现的主要信息来源于 UEFI 技术的开源社区中，这个开源社区有四个和 UEFI BIOS 相关开源的项目，它们是 EDK（EFI Dev Kit）、EDK Shell、EFI ToolKit 和 EDKII。其中，EFI Shell 工具是区别于 EDK 及 EDKII 外 DXE 阶段的应用界面环境。在此环境里，已经开发了一些基本应用，如进退目录、设置 IP 地址、编辑文本、复制文件等等。然后，里面非常缺对 API 性能分析及优化的功能。

EFI Shell 是 UEFI 的 BIOS 中功能齐全的调试和执行环境。随着 UEFI 的 BIOS 技术的推广，更多的制造厂商，甚至个人，都可以在该环境下开发专有应用软件。然而，衡量及测评开发的应用软件在 UEFI 的 BIOS 实际平台上执行的性能，当前还没有具体解决的方案[12]。同时，对于从事此类开发的程序员而言，通过大量地研习 EFI Shell 自带应用工具的实例，可以更快的掌握 UEFI 的开发经验。那么，怎样在 EFI Shell 环境里编写测试的用例，完成基于 EFI BIOS 的内部模块 API 性

能进行一定分析，是技术难点。

除此之外，随着 EFI 技术不仅可实现在服务器中，同样可以将它嵌入到嵌入式的软件工程当中。这就提出了要求，那就是 EFI 开发的库函数必须兼容在别的编译器里，可以编译出相应的可执行的文件，这样就可以提高 UEFI 代码的容错性和兼容性[13]。

紧接着，本文将提出怎样在 Linux 的环境下实现 EFI BIOS 实现阶段系统调用的方案，其技术难度在于 BIOS 完成了初始化工作后将底层系统的参数传递给操作系统，怎样将这些参数供给上次用户使用。本文采用的方案是用执行系统调用。以此来实现操作系统里调用底层 API 应用的功能。

随着基于 UEFI 技术的 BIOS 推广与应用，对 UEFI 技术的应用程序的功能需求提出了越来越多的要求。UEFI 的 BIOS 下 API 接口功能实现，在一定程度下提供了 UEFI 环境里开发模块软件的函数接口，这也体现了本文的实用价值。性能分析的模块是针对 UEFI 的 DXE 阶段尚没有应用程序的路径信息和分析模块情况。这样的开发，为从事 UEFI 环境中开发应用程序的开发者提供了分析函数的调用关系及函数所执行的时间手段。另外，工作还尝试了在 UEFI 的 BIOS 下利用 VFR 语言来嵌入对应的应用模块，基于应用程序去实现图形的界面设计。此外，在 UEFI 的 BIOS 执行环境下，通过对 NT32 模拟环境的实现对 PEI 阶段底层的 API 性能测试分析的实例，直观地分析了 UEFI 的 BIOS 其底层的功能函数的性能[14]。整个设计与开发过程充分地体现了在 UEFI 技术的 BIOS 下开发设计相对应传统的 BIOS 开发的优点。

1.4 本章小结

本章对 UEFI BIOS 的目前国内外概况和发展趋势进行了简要的描述，明确阐述了 UEFI BIOS 较传统 BIOS 的优点以及其本身的一些局限点。接着，对 UEFI 技术下 BIOS 的整体设计框架和启动过程各个阶段进行了较为深入的分析。在此基础上，论文的整体结构及各章节的写作陈述内容如下。

本文共分为六章。首先要介绍的是作为新一代 BIOS 的背景知识，即 UEFI 技术的特性，Tiano 框架和模块设计的分析。在研究学习了 UEFI 技术背景、EDK 工具用法等一系列技术知识提前下，基于 EDK 的架构下文中具体实现了应用程序的开发。描述了整个开发的流程，演示了程序运行的结果。在实习工作期间，需要在 UEFI Shell 下对每一代新的开发板与其对应的 UEFI BIOS 的 API 性能进行测试，以来确定每一款产品是否适用。在这样的工作要求下，文中完成了两个测试功能模块，第一个是在 NT32 平台下对 PEI 阶段的 MDE 中的 strcpy 函数接口进行了性能测试的设计与实现。另一个是在 LINUX 下对 RUNTIME 阶段底层 API 性能测试

的设计与实现。下面简要介绍一下论文工作上的章节安排。

第一章 绪论。本章首先阐述了 UEFI 背景概况、国内外概况和发展趋势。描述了 UEFI BIOS 与传统 BIOS 相比的优点，以及其本身的安全隐患和其局限性问题。

第二章 Tiano 系统框架及 EDK 模块的研究及分析。首先，阐明了 Tiano 系统框架及实现原理。接着详细地分析了 EDK 的功能模块及各模块相互之间的依赖关系。着重地对 EDK 下的 MdePkg 模块进行了研究，列出和学习了 MdePkg 模块里的类库和单元库。为论文后期在 EDK 下具体设计开发应用程序做了一个技术支持。需要说明的是，论文在写作之初，花了大量的时间来研究与分析 EDK 模块的架构与开发实现的流程。

第三章 EDK 下应用程序的设计与实现。这一章首先详细地阐明了 EDK 工具的下载配置及 Debug 的调试方法。接着具体地设计与实现了 EDK 下的应用程序的嵌入，详细地描述了开发流程。为进一步在 EDK 下添加更高级的应用程序，如网络应用、文件操作、硬软件管理等，作出了一个明确的实例演示。

第四章 API 性能测试在 NT32 平台及 LINUX 下的实现。本章首先设计完成了一款基于 PEI 阶段的底层 API 性能分析测试模块，成功地在 UEFI Shell 环境下实现了对 MDE Library 中的 strcpy 函数接口的分析。接着对 LINUX 的 RUNTIME 阶段底层 API 分析研究，通过系统调用的方式实现了在 LINUX 下底层 API 的实时调用，且编写应用程序代码获取底层 TimeService 提供的准确时间。

第五章 总结与展望。

第二章 Tiano 系统框架及 EDK 模块的研究与分析

与传统的 BIOS 实现机制不同，Tiano 系统为了模块化，启动过程分成多个阶段，每个阶段分别用不同的模块来实现，各个阶段之间都有自己定义好的接口。从平台上电到初始化，最后到操作系统启动完成，共分为七个具体阶段。文中阐述了各个阶段的功能。EDK 是个开源的 EFI BIOS 发布框架，包含了大量的开发示例和底层库函数，开源部分涉及 NT32、Unix 和 DUET 三种平台架构。本章首先详细地分析研究了 Tiano 的系统架构。接着详细地分析 EDK 的功能模块及各模块之间的依赖关系，并着重地列举了其中依赖最多的 MdePkg 模块，为第四章详细对 MDE Library 的 API 性能测试作出了一个详细的指引。

2.1 Tiano 系统框架及 EDK 模块介绍

Tiano 系统采用模块化设计理念，基于现代软件体系设计的思想，将 Tiano 框架流程详细地划分为 SEC 阶段、PEI 阶段、DXE 阶段、BDS 阶段、TSL 阶段、RT 阶段和 AL 阶段共七个阶段。UEFI/TIANO 系统框架如图 2.1。

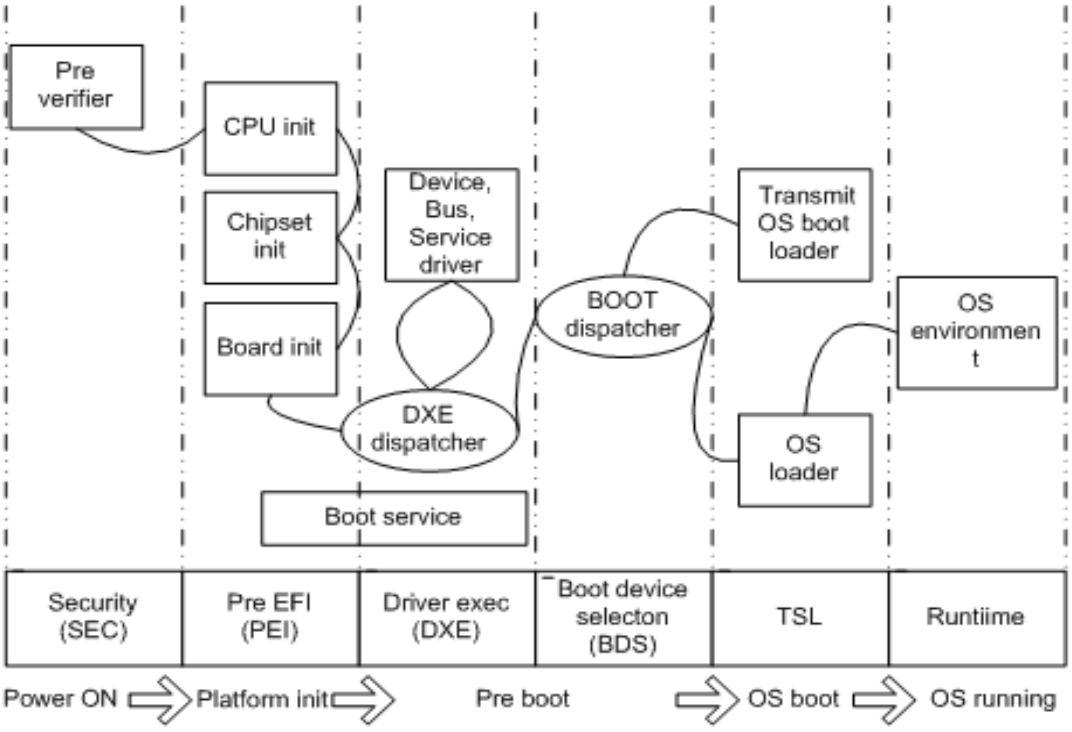

图 2.1 UEFI/Tiano 系统框架[15]

Tiano 系统过程启动简单流程是 Power on-> Platform init-> Pre boot-> Os boot-> OS running，其内 Power on 到 OS boot 过程详细的四个阶段分析如下。

1. SEC 阶段，安全阶段是上电执行的第一步。SEC 能支持检测系统操作码，确保所选平台固件的映像没遭到破坏。此阶段通常是需硬件的支持，新一代处理器和芯片组是支持这些功能的。

2. Pre PEI 阶段，主要负责尽可能少地工作来寻找和初始化内存，这个阶段的代码倾向于用适合机器的汇编代码编写。

3. DXE 阶段，它是大多数的系统初始化执行的一个阶段。PEI 到 DXE 阶段的转化是顺行的，一旦 DXE 阶段的初始化装入程序完成后，PEI 阶段代码将不可用了，DXE 就成为了一个健全的操作环境。

4. BDS 阶段，BDS 是 Boot Device Selection，顾名思义，该阶段是尝试从可用的启动设备来启动 OS。在 BDS 阶段可以向用户展示界面，也可以被 OEM 修改用来指定 DXE 所对应的系统[16]。

EDK 是开源的 EFI BIOS 框架，其中包含了开发的示例及大量的基本底层的库函数，它的开源部分共涉及到 NT32、DUET 和 UNIX 这三种平台。开源了核心代码之外，EDK 环境同时是开发及测试一些 EFI 的驱动程序综合平台[17]。近年来，Intel 在 EDK 的模块上扩展了功能，定义了更为强大的、集成了多种功能模块的 EDKII。此 EDKII 在开发类、库机制及 PCD 机制等方面展开了扩展，着眼于怎样让开发者开发出更规范、更容易的自定义模块。如图 2.2 所示，描述了 EDKII 的主要功能模块之间的依赖关系。

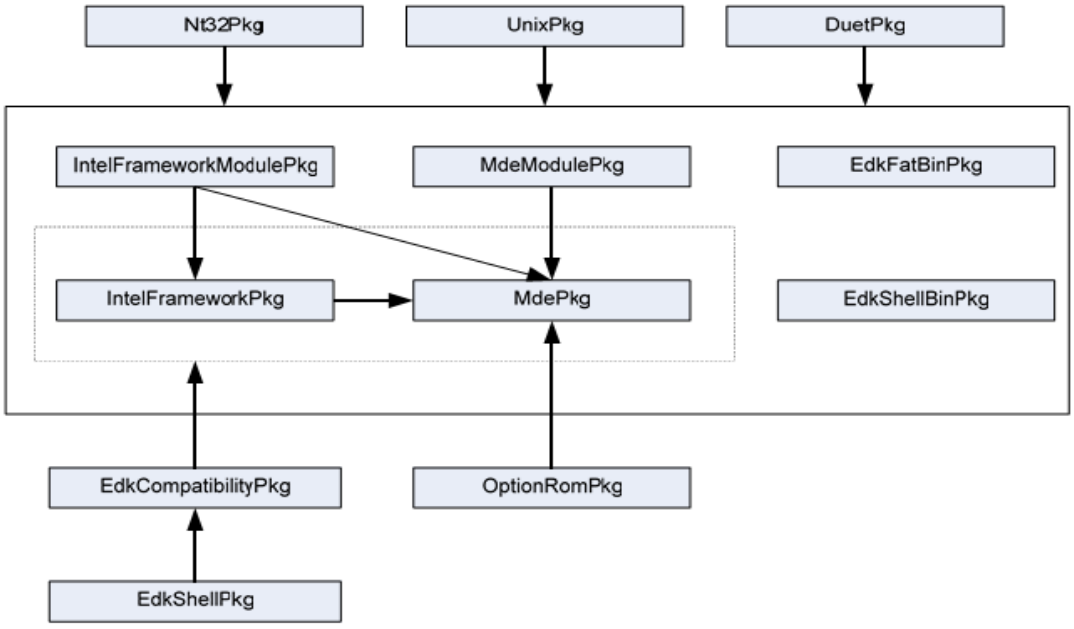

图 2.2 EDKII 的主要模块间依赖关系[16]

论文的工作是在 UEFI 的 BIOS 最新开发的平台 EDKII 上展开的，其各功能模块间有着紧密的相互依赖关系，其中 MdeModulePkg 所用的协议原型在 MdePkg 中定义。因此，MdeModulePkg 依赖于 MdePkg。

从模块之间依赖的关系图里可以看到，要编译 OptionRomPkg，只要有 MdePkg 和 BaseTools 模块即可。要编译 EdkCompatibiliryPkg 只要有 MdePkg、IntelFrameworkPkg 和 BaseTools 模块即可。要编译 EdkShellPkg，则要有 EdkCompatibilityPkg 和 BaseTools 模块。要编译 Unix、NT32、Duet 模拟的平台，都依赖 MdePkg、MdeModulePkg、IntelFrameworkPkg、IntelFrameworkModulePkg、EdkFatBinPkg、EdkShellBinPkg 和 BaseTools 等功能模块。

EDKII 的主要功能模块所对应功能的分析如下。

1. BaseTools：给出二进制的编译工具以及集合编译环境的配置文件。

2. Conf：是产生编译的环境信息、编译的参数及编译的目标定义保存的路径，配置操作之后，此路径下产生三类配置的文件。

3. MdePkg：包含 EDKII 的各平台底层的库函数、工业标准及协议。IA32、X64、IA64 等平台架构对应的 UEFI BIOS 均可在各自的开发模块里引用对应文件的库函数，减低开发难度。

4. MdeModulePkg：提供了一些跨平台模块，其中包含 MdePkg 的底层库应用的模块示例。

5. EdkShellPkg：提供 Shell 应用程序开发的环境，它也是跨平台开发的环境。

6. EdkFatBinPkg：提供针对不同的处理器架构的原始 FAT。因为 FAT 版权的问题，里面只提供.efi 执行文件，不提供相应的源代码。

7. Nt32Pkg：在 Microsoft 操作系统里可加载的 32 位的模拟器，它提供了 UEFI Runtime 支持的环境。

8. UnixPkg：在 Linux 操作系统里可加载的运行 32 位模拟器，它提供了 UEFI Runtime 支持的环境。

9. DuetPkg：提供了基于传统的 BIOS Runtime 的环境支持库，它可对传统 BIOS 进行一定的改造，可在实际硬件平台构造实际 UEFI BIOS 的工作环境。

10. OptionRomPkg：包含了针对不同处理器的架构而编译的 PCI 兼容实例。

11. EdkCompatibilityPkg：提供了针对传统的 BIOS 及 EDK 来定义库以及协议对应的兼容支持，它保证了 EDKII 对之前 EDK 所有功能的支持[17]。

以上每个文件包各有都有类似结构，例如 MdePkg 有下列子文件夹项。Include 包含了公共文件的头文件，Ia32 是支持 IA32 的架构内部对应的头文件，X64 支持 X64 的架构内部头文件，Ipf 支持 IA64 的架构内部头文件看，Ebc 支持 EBC 的架构内部头文件，Uefi 支持以 Uefi2.3 的规范内部头文件，Pi 支持以 PI1.2 的规范内部头文件，Protocol 包含了各种协议的规范的头文件，Ppi 包含各种 PPIs

的规范的头文件，Guid 包含了各种 GUIDs 的规范的头文件，IndustryStandard 是工业规范的标准公共文件，Library 包含了 MDE 库的文件，Library 是 MDE 库原型。

以下四个软硬件平台是开发应用程序和应用功能模块的环境。

1. Microsoft Windows 2000 or Microsoft Windows XP32 的操作系统。

2. 作为 Intel 32 位体系的结构平台如下。

－ Microsoft Visual Studio .NET 2003 Enterprise

3. 用作 Intel 安腾 CPU 进行开发。

－ Microsoft Windows Server 2004 Driver Development Kit (Driver Developer Kit), build 370

4. ITP 应用工具。

ITP 的全称为 In Target Probe，正如 Linux 下我用 GDB 或内核调试的工具 KGDB 来进行对应的应用程序和内核程序调试，EFI 技术可使用 Intel 公司的 ITP 应用工具调试。

2.2 EDK 下 MdePkg 模块的分析

MdePkg 全称 Module Development Environment Package. 它是 EDK II 可以调用的 API 库的集合，这些库在 BIOS 的启动的过程中提供一些服务，同时 BIOS 在进入系统后也可以提供 RUNTIME 调用的服务，启动 BIOS 时候提供 BootService 服务。MdePkg 功能模块向 BIOS 提供一些基本的函数功能，同时 OS 厂商和个人同样可以 MdePkg 功能模块执行内核模块设计开发。

2.2.1 MDE 里类库

如表 2.1 描述了 MDE 里底层的一些提供 API 接口的类库，他们提供了相应的 API 应用。例如，Base Library 有很多 Stringcopy，Stringcmp 等等底层的 API，当 BIOS 进入到操作系统，就可以调用 API 的应用接口，基于这类 API，用户可以这个基础上开发一些与 UEFI 相关的应用软件和工具。

表 2.1 MdeLib 库列举

名称	描述
Base Library	提供字符串的函数，链表函数，数学函数，同步功能，和 CPU 架构具体的职能。
SAL Library	提供了调用 SAL 的接口服务。
UEFI USB Library	实现了 USB 对应的 API 接口的支持。

（表 2.1 续一）

名称	描述
Cache Maintenance Library	提供维持指令及数据以高速缓存服务。
CPU Library	提供 CPU 的架构对应的功能函数，这些功能函数不能在 Base Library 里定义因为和 PAL Library 的依赖关系。
Debug Library	提调试消息发到调试输出设备服务。
Device Path Library	提供对应库函数构造及解析 UEFI 设备的路径。
DXE Core Point Library	DXE core 模块的入口。
DXE Services Library	提供了简化 DXE 的驱动实现功能。此功能可得到 FFS 的文件数据。
DXE Services Table Library	提供了简化 DXE 的驱动实现功能。此功能可得到 FFS 的文件数据。
ExtractGuid Section Library	提供了获取 GUID 的数据功能服务。
HOB BB Library	提供了构造、擦除内存的 HOB 服务，创造、解析。用在 PEI 及 DXE 的模块。
I/O Library	提供了访问 I / O 的端口及 MMIO 寄存器的服务。
Memory Allocation Library	提供了各种类型存储器内存的分配区服务。
PAL BB Library	提供了 PAL 的调用功能。
PCD BB Library	提供了获取、设置平台的配置数据服务。
PCI BB CF Library	提供了 I / O 端口 0xCF8 进 PCI 参数的空间服务。
PCI Express Library	提供 MMI PCI Epress 窗口进 PCI 的参数空间的服务。
PCI Library	提供了访问 PCI 的配置空间服务。
PCI Segment Library	提供了通过具有多个 PCI 的段数平台进 PCI 参数空间的服务。
PE/COFFEntry Point Library	提供了在 PE/COFF 里提取 PE / COFF 的指针服务。
PE/COFF Loader Library	提供装载和重分配一个 PE/COFF 映像的服务。
PEI Core Entry Point Library	函数库的入口模块。
PEI Services Library	实现所有 PEI 的功能服务的一个库。
PEI Services Pointer Library	提供获取 PEI 的服务表对应指针服务。
PEIM Entry Point Library	PEM 的模块入口库。
Performance Library	提供了记录运行时间及恢复时间服务。
Post Code Library	提供了错误代码信息发到 POST 上的服务。
Print Library	提供了格式化的字符串打印到缓冲区。
Report Status Code Library	提供了记录程序所运行状态服务。

（表 2.1 续二）

名称	描述
Register Table Library	提供服务及处理对应的登记表，包含 I / O 将 MMIO，PCI 的配置活动。
SAL Library	提供了调用 SAL 的接口服务。
Serial Port Library	提供三种常见串行的 I / O 端口的功能服务。
SMBUS Library	提供了库函数访问设备的功能。此类库先要在 SMBUS 的控制手册里说明。
Synchronization Library	提供同步的功能。
Timer Library	提供校准一定延迟及性能的计数服务。
UEFI Application Library	UEFI 应用入口库模块。
UEFI Boot Services Library	提供了获取 EFI 的启动服务返回指针对应的服务，适用在 DXE 的阶段及 UEFI 模块的类型。
UEFI Decompress Library	提供了解压缩的使用缓冲区的服务。
UEFI Library	实现了统一的 UEFI 操作的功能库函数。适用在 DXE 的阶段及 UEFI 的模块类型。
UEFI Runtime Library	实现 UEFI 运行时间的服务功能。适用在 DXE 的阶段及 UEFI 的模块类型。
UEFI SCSI Library	实现了提交 SCSI 的命令服务功能。

2.2.2 单元库介绍

每个单元库（library instance）包括一个或多个单元库类。单元库的实现可以同时占用一个或多个其它的库类，也可以选择产生的构造和析构函数。为图书馆的实例构造函数和析调用语义随模块类型。阿库实例的另一个目的是联系在一起的模块组特定的类型。如果库实例被声明为一个，那么，可以与任何类型的模块连接基地，模块类型[18]。

论文工作需要对函数库进行分析，开发出一套函数库来为上层提供服务。在 MDE 模块中主要利用 PCD 入口来实现函数间的调用，说明一个 API 的调用是通过对应的 PCD 值来获取参数的。

2.3 本章小结

本章首先详细描述了新一代 BIOS 的设计框架流程的七个阶段 SEC、PEI、DXE、BDS、TSL、RT 和 AL。其次介绍了 EDKII 主要模块之间的相互依赖关系及模块本身的功能分析。最后详细介绍了 EDK 下的 MdePkg 模块，分别介绍了 MdePkg 里面的类库和单元库。

第三章 EDK 下应用程序的设计与实现

上一章详细分析了 Tiano 系统框架及 EDK 模块。在这样的基础上，完成论文工作的第一步深入地研究学习 EDK 的下载配置及其 Debug 的调试方法，并在 UEFI Shell 的环境下具体地设计实现了应用程序的嵌入，编译并执行能够在实际运行的.efi 应用程序，为在 EFI Shell 添加更高级的应用程序作了先一步的研究。

3.1 EDK 的下载配置及 Debug 调试方法

EDK 是一个开源的 EFI BIOS 的发布框架，其中包含一系列的开发示例和大量的基本底层库函数，开源部分涉及 NT32、Unix 和 DUET 三种平台架构。除了开源核心代码，EDK 同时也是一个开发、调试和测试 EFI 程序的综合平台。

在完成论文工作的过程中，学习 EDK 下载配置是必不可少的一步。学习的下载配置步骤如下。首先，需要完成搭建一个 EDK 开发平台，通过 TortoiseSVN 客户端在本地更新到指定的 EDKII 开发版本，将开发的框架放在 C：\EDKII 目录下。进入 C：\EDKII\EdkShellPkg 路径下，修改 EdkShellPkg.dsc 配置文件，设定宏路径“DEFINE EDK_SHELL_DIR”添加开发中所用到的库文件，要增加对图形界面语言 VFR 的支持，就要添加在 Libraries 中添加如下的文件。

EdkCompatibilityPkg/Foundation/Library/Dxe/EfiIfrSupportLib/EfiIfrSupportLib.inf。

然后在 Components 中添加所开发模块的索引文件，再根据具体情况在 Build Options 部分修改编译器对工程的编译选项，针对 IA32 架构的编译选项配置部分为。

DEBUG_VS2005xASL_IA32_CC_FLAGS = /nologo /c /WX /GS- /W4 /Gs8192 /D UNICODE /O1ib2 /GL /FIAutoGen.h /EHs-c- /GR- /GF /Gy /Zi /Gm

搭建平台完成之后，就可以进行相信的编译工作，进入 Visual Studio 2005 command Prompt，进入开发路径，运行 edksetup 自动设置默认的开发配置，然后输入如下命令。

build -a IA32 -a X64 -a IPF -p EdkShellPkg\EdkShellPkg.dsc

程序运行成功，可以得到能在 EFI Shell 环境下运行的.efi 的可执行应用程序。程序的开发过程，调试是重要的一方面。NT32 的模拟器下的 Debug 的调试环境界面如图 3.1 所示。

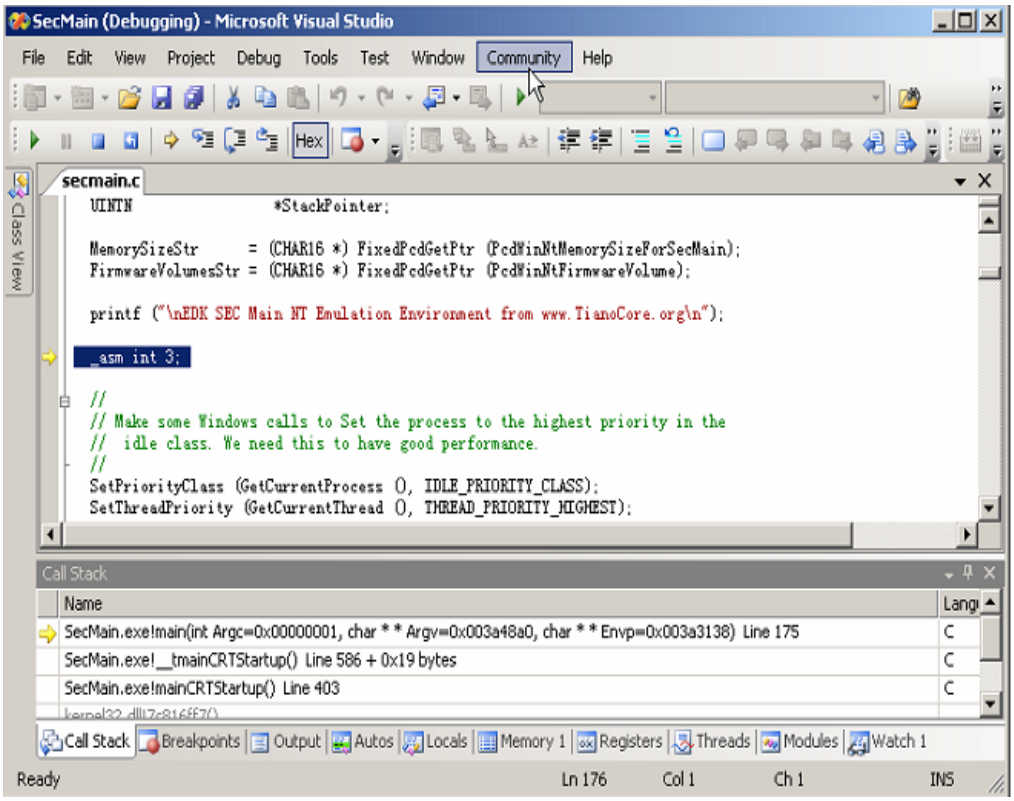

图 3.1 NT32 的模拟器 DEBUG

基于 EDK 的平台应用的程序开发过程，完成论文工作中学习了两种 DEBUG 调试的方法、技巧。

第一种相对容易，针对的是 IA-32 执行程序，可直接对应 NT32 的模拟器用 Microsoft Visual Studio 2005 enterprise 进行。

首先，在调试的程序代码里插入“_asm int 3”。接着，重新编译源代码，在 NT32 的模拟器里运行。程序执行至需要 Debug 的地方时，进而触发应用程序的一个异常，选择“Cancle”和“New Instance of Visual Studio 2005”的选项，这就进了调试的环境，即如图中所示的界面。

第二种比较复杂，通过 Intel ITP 工具来直接对 PC 的处理器实现硬件调试。这个调试方式很通用，可以对 X64、IA-64 和 IA-32 所架构硬件的平台执行单步跟踪，检查内部的寄存器、分析硬件的端口信息这些高级操作。

具体实现的方法是在 DEBUG 的运行程序中加入一段代码，这段代码能造成程序进入死循环，可以加入如下代码。

```
volatile unsigned int Index;
for (Index = 0; Index == 0;);
```

程序执行这一段的时候会陷入设置的死循环，此时打开之前的 ITP 硬件的调试工具，输入“halt”命令，暂停程序，然后输入“loadthis”指令，根据需要调试

文件所有的路径信息来加载对应的本地盘符源代码。本地编译是常规需求，若要远程调试，要先把远程的编译目录对应到当前 PC 的盘符。

由于临时变量一般存在 eax 寄存器中。所以，当前这个死循环一直在判断 eax 寄存器的数值。这时，在 ITP 工具命令行中输入”eax=1”，就可以跳出该死循环，进入需要调试的程序。查看当前内存调试的现场截图，如图 3.2 所示。

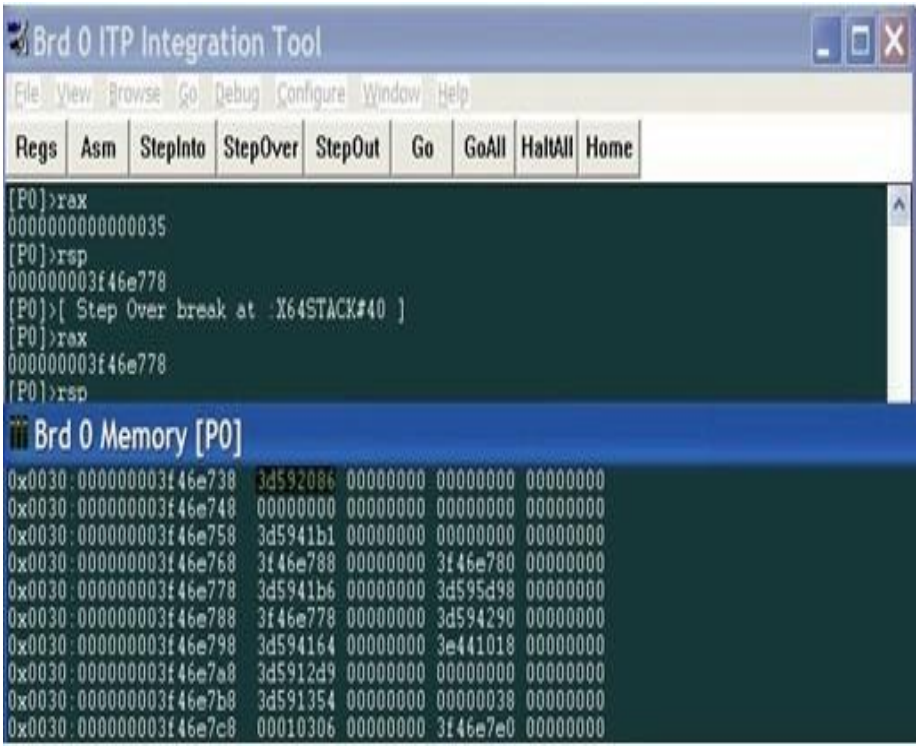

图 3.2 Intel ITP DEBUG 现场截图

系统的调试流程如下，在 X64 的架构目标机中，通过 Intel ITP 工具将待测版本 EFI BIOS 的 Image 烧写到 Flash 中。重启 PC，进入到 EFI Shell 命令行下，转到工具在的目录，运行应用程序。通过串口及目标机的屏幕来观察程序运行的情况。若程序发生了异常，可以用 ITP 硬件的工具进行对应的在线调试[19]。

3.2 应用程序在 EDK 模块下的实现

论文第一大块的工作就是在 EDK 模块下实现应用程序的开发，通过在 EDK 下实现一个应用程序的编译，可以清楚地学习掌握 EDK 模块下对应用程序的源程序是如何处理并实现的。

首先选择应用程序运行的环境平台，文中选用 IA32 模拟平台即 NT32 平台，这是一个用 VS2005 编译器生成的、模拟的 32 位的 EFI BIOS 运行环境。其操作流

程如下。

在 C:\EDK\Platform\Nt32\Build\Nt32.dsc 的 Libraries.Platform 部分添加，

Other\Maintained\Application\Shell\Library\EfiShellLib.inf

这个过程是将 UEFI Shell 运行环境加载到 NT32 平台里面，为应用程序提供一个运行的场所。

接着将 Other\Maintain\Application\Shell\HelloWorld\HelloWorld.inf，FV=NULL 加到 Components 部分，注意应用程序的位置，在 dsc 文件中明确限制了程序所占模块的位置。这里的 HelloWorld.inf 文件相当于是一个 Makefile 文件，但不能等同于 Makefile 文件。HelloWorld.inf 类型定义了程序的入口函数，GUID，源代码、支持的编译器等等。

3.2.1 编译应用程序

编译过程就是把高级语言的程序变成计算机可以识别的二进制语言。编译程序把一个源程序翻译成目标程序的工作过程分为五个阶段，词法分析、语法分析、语义检查和中间码生成、代码优化、目标码生成。词法分析和语法分析又称为源程序分析。在 NT32 环境下，这个过程操作如下所述。

程序完成后，在 shell 下输入如下命令，Cd C:\EDK\Sample\Platform\Nt32\Build

设置程序所用运行环境变量的路径，C:\EDK\Sample\Platform\Nt32\Build>Set EDK_SOURCE=C:\EDK，接着 Build，C:\EDK\Sample\Platform\Nt32\Build>nmake

编译结束后程序的二进制目标码将出现在下列路径，根据机器的路径而定，

C:\EDK\Sample\Platform\Nt32\Build\IA32\HelloWorld.efi

3.2.2 执行应用程序

应用程序编结束功后，在 C:\EDK\Sample\Platform\Nt32Build 目录中可以会看到一个 HelloWorld.efi 的文件，此文件就是一个在 NT32 平台上的二进制执行代码文件。.efi 文件运行的命令很简单，首先要模拟器 Secmain.exe 打开，运行进入 Shell，在该环境中，已经存在了一些基本的应用，比如进入退出目录、设置当前 IP 地址、编辑文本文件、复制移动文件等。但其中，却非常缺少一种对 API 的性能分析及其优化方案的支持功能不用任何参数，在 C:\EDK\Sample\Platform\Nt32Build 目录中直接打 HelloWorld.efi，按回车即可，HelloWorld.efi 应用程序的执行结果如图 3.3 所示。

```
EFI Shell version 2.30 [10]
Current running mode 1.1.2
Device mapping table
  fant0:BlockDevice - Alias f10
        VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)/VenHw(0C95A935-A006-
11D4-BCFA-0080C73C8881,00000000)
  fsnt1:BlockDevice - Alias f11
        VenHw(58C51B1-76F3-11D4-BCEA-0080C73C8881)/VenHw(0C95A935-A006-1
1D4-BCFA-0080C73C8881,01000000)
  blk0:BlockDevice-Alias(null)
        VenHw(58C518B1-76F3-11D4-BCEA-008073C8881)/VenHw(0C95A928-A006-1
1D4-BCFA-0080C73C8881,00000000)
  blk1:BlockDevice-Alias(null)
        VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)/VenHw(0C95A92F-A006-
11D4-BCFA-0080C73C8881,01000000)

Press ESC in 5 seconds to skip startup.nsh,any other key to continue.
Shell>fsnt0:

fsnt0:\>HelloWorld.efi
UEFI Hello World!—Hua.dong

fsnt0:\>
```

图 3.3 Hello world 运行结果

3.2.3 应用程序代码分析

图中可以清晰地看到在 EFI Shell 下输出了想要的“UEFI Hello World”，这样一个简单的应用程序嵌入就根本完成了，HelloWorld.c 的代码如下。

```
EFI_STATUS
EFIAPI
UefiMain(
    IN EFI_HANDLE              ImageHandle,
    IN EFI_SYSTEM_TABLE        *Systemtable
)
 {
UINT32 Index;
Index=0;
//
// Three PCD type（FeatureFlag. UINT 32 and String)are used as the sample.
//
```

```
If   (FeaturePcdGet (PcdHelloWorldPrint Enable)){
    for (Index=0;Index<PcdGet 32(PcdHelloWorldPrintTimes);Index ++) {
//
//Use UefiLib Print API to print string to UEFI console
//
Print ((CHAR16*)PcdGerPtr(PcdHelloWorldprintString));
}
}
return EFI_SUCCESS;
}
```

该函数 UefiMain (

IN EFI_HANDLE ImageHandle,

IN EFI_SYSTEM_TABLE *SystemTable

)

是应用程序的入口，函数里面首先定义了一个 32 位的整形变量，并且赋初值为 0，使用了一个 for 循环， PcdHelloWorldPrintEnable 在 platform 里面永远是 true 的一个 GUID，等同于 C 语言里面定义的一个宏，EFI 里面一般都将需要用的数据初始值定义在了 Platform 的 MdeModulePkg.dec 文件里面。因为在 EDK 模块下用到 string，要用 PCD 调用方式以实现，EDK 模块属于底层系统的框架，和高级的 VC++ 等有些区别，没有库提供调用。HelloWorld.c 的依赖代码如下。

```
// HelloWorldStrings.uni
//
//String   resources for HelloWorld   application
//
/=#
#langdef   eng "English"
#langdef   fra   "Fran?ais"
#string   STR_HELLOWORLD   #langusge   eng   "Hello, world!\n"
#language   fra "Hello,world!\n"
#string   STR_GOODBYE #language eng "Goodbye,cruel   world!\n"
#language   fra "Goodbye .cruel   world!\n"
Composing an EFI Shell Appliceation
Version   0.91 June 27,   2005   22
#string STR_SHELLENV_GNC_COMMAND_NOT_SUPPORT
#language eng "%hs:   This   command   is   not   supported   in   the   \n"
"underlaying   EFI version, required   EFI   version   \n"
"should   be %hd.%hd or above\n\n"
#language   fra "%hs:This   command is not supported in the \n"
"underlaying EFI version, required EFI
```

可以在 MdeModulePkg.dsc 里面将初值修改， MdeModulePkg.dsc 未做修改的情况，Hello_World.c 文件是用 MdeModulePkg.dec 的初值。这样一来无论驱动或是

应用程序在用的时候都无须定义，用 PCD 的函数调用是可以实现的，其次 for 循环的 PcdHello_WorldPrintTimes 在 MdeModulePkg.dec 文件里面是定义成 1,这种情况下，for 循环运行了一次。要多次运行的话，需要在 MdeModulePkg.dec 文件里将 PCD 对应的值改大，这样程序最后被修改成 Print (CHAR16*)PcdGetPtr (PcdHello_WorldPrintString)。程序中可以看到，语句首先是用 PcdGetPtr 函数获得 PcdHelloWorldPrintString 的值，这个值同样被定义于 MdeModulePkg.dec 中，得到这个值后将它强制转换成 16 位的 char 型输出。这个 HelloWorld 所用到的 PCD 的值不一定放在 MdeModulePkg.dec 文件中，也可以放在 Nt32Pkg.dec 中。

3.3 本章小结

本章首先介绍了 EDK 模块的下载配置及程序 Debug 的调试方法。在深入研究学习 EDK 的下载配置及其 Debug 的调试方法的基础上，完成了 UEFI Shell 的环境下具体应用程序的嵌入，编译并执行能够在实际运行的.efi 应用程序，并分析了函数的源代码及依赖代码，为在 EFI Shell 添加更高级的应用程序作了先一步的研究学习。

第四章 API 性能测试在 NT32 平台及 LINUX 下的实现

上一章详细地描述了在 UEFI Shell 环境下开发出了一个实际的应用程序的流程。类似的，今后完全可以在 EFI 下加入更多实用的功能，比说说可以不进入系统，在 EFI 里就可以进行文件操作，在 EFI 里实用网络功能等等。本章分别详细地给出了 API 性能测试在 NT32 平台及 LINUX 下的实现。首先设计完成了一款基于 PEI 阶段底层 API 的性能分析测模块，成功地在 UEFI Shell 下实现了对 MDE Library 中 Strcpy 函数的分析。接着通过对 LINUX RUNTIME 阶段底层 API 的研究，实现了 LINUX 系统环境下底层 API 的实时调用，且编写应用程序代码获取底层 TimeService 提供的准确时间，进而直观地测试了 RUNTIME 阶段底层 API 的性能。

4.1 PEI 阶段底层 API 性能测试模块的总体设计及结构

4.1.1 测试模块系统分析

UEFI BIOS 下 API 功能的实现，可以从一定程度上提供 UEFI 环境下开发各种模块软件函数接口。在 UEFI Shell 环境中，已经存在了一些基本的应用，比如进入退出目录、设置当前 IP 地址、编辑文本文件、复制删除文件等。但其中，非常缺少一款对底层 API 的性能分析及其优化方案的支持功能。相应的，论文的工作就是来实现这样的测试底层 API 的性能功能模块。在这样的研究支持下，可以为 UEFI 软件开发者提供相应的系统函数调用及函数运行时间的调用手段。其中，在 NT32 平台下实现的 PEI 阶段底层 API 的性能测试的实现，直观地给出了 UEFI BIOS 的底层 API 的性能。

论文开始的时候设想的测试的功能模块应该支持 IA32,X64,IA64 系统架构的编译器，而且可以在系统启动的六个阶段进行测试，比如对 BASE 模块在 PEI、DXE、 SMM、 Runtime、 PEI Core 和 Dxe Core 阶段都可以测试。其中 SMM 阶段是一种特殊用途的运行模式，用来处理影响整个系统功能的模式。这里说的影响整个系统功能的模式有电源管理，系统硬件的控制等，或者也可以用这个模式来处理专有 OEM Designed 代码。SMM Mode 只面向系统固件，而不是由软件来申请使用的。

论文工作以 PEI 阶段为例，因为同时对六个阶段进行测试的工作量是相当大的，为了更有效地完成工作，论文选择了在 PEI 阶段对 MDE Library 中的 Strcpy 函数分析为研究课题。论文对 PEI 阶段 MDE Library 中 Strcpy 的性能测试系模块

设计分析如下。

1. 首先是 Proxy Driver，Proxy Driver 的作用是把一个单元库封装到一个独立的模块中。这个模块中包括了一些待测试的 API。其中没有 Driver 的意义，它只是提供一个临时 API 的活动场所。它的实现可以参考 EFI 下如何实现 Driver 的相关文档。

2. 然后执行 “Proxy Driver”的入口函数去插入该库的接口函数到这个模块中，来让测试程序对单元库进行测试。

3. 最后在这个模块中执行测试程序并输出测试结果。整个测试流程如下图 4.1 所示。

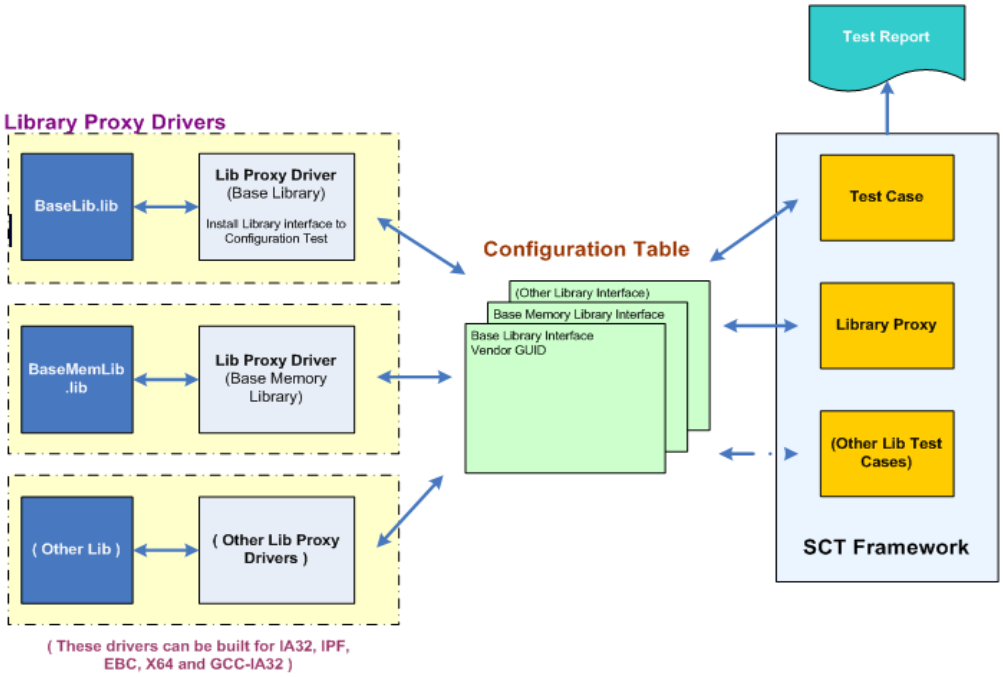

图 4.1 MdeTestLib 原理图

代理驱动将 MDE 中的单元库抽象出来，通过测试代码获取抽象出来的地址，这两个驱动建立了两个空间，提供分析案例活动的空间，分析程序获取 Lib Proxy Driver 转换过来的 Interface，调用该 API 并对它输入相应的参数，从而获取函数返回值，跟 MDE_Library 中对应的函数实现功能，判断程序能否实现相应功能，是否在功能及性能上的缺陷。最后通过黑盒及白盒测试的思想得出其性能解析[21]。

图中，GenTestPkg.bat 的作用是生成需要的二进制式代码测试包。Cleanall.bat 文件作用是删除对应代理驱动及测试案例之间的软连接，另外，它可以同时删除之前的测试源码。MdeTestPkg.spd 文件提供说明的文档。MdeTestPkg_*.fpd 文件作用是描述待测试源码包，概括测试架构及测试案例支持驱动、库连接，并标示了

特殊案例。RunNt32.bat 文件是执行 SCT 文件的脚本。脚本被执行前提是 NT32 的模拟环境需要被编译成功。

4.1.2 测试模块函数框架说明

下面主要描述 EDK II 开发测试的框架所需的单元库，描述测试框架模型，并分析 API 的功能及性能。MdeTestPkg 的组成机构如表 4.1 所示。

表 4.1 MdeTestPkg 组成

名称	描述
Debug Test Library	函数作用是 debug，函数设定了一个断点及跳转至保存返回点，当程序发生死循环的情况下可以跟踪检查 CPU 找出程序出错位置。
Pei BB Proxy Main Library	Pei 阶段 proxy driver 模块的入口函数。
Pei BB Test Case Main Library	Pei 阶段测试程序模块的入口地址。
Pei Standard Test Library	提供了在 Pei Core 及 Pei 阶段的测试结果。
Pei Core BB Proxy Main Library	Pei Core 阶段的测试程序的入口模块驱动代理。

在 PEI 的阶段所有模块都叫 PEIM，英文的全称 Pre EFI Interface Module，这类模块由三种类型驱动组成，分析如下。

第一类驱动负责插入点调试，在 BIOS 的启动阶段，处理器执行任务过程是单任务的模式，当运行过程中发生了死锁，程序即无法来继续执行，程序员并无法介入当中调试，在这种情况下 MDE 模块必须加入调试模块来测试模块的 Debug Test Library。设计模块，可以在程序放生死循环的前面打上一个标记，用 ITP 调试工具来进行单步跟踪，找到发生死锁的地方。

第二类模块分两种，一种是 Pei Core BB Proxy Main Library，另外一种 Pei BB Proxy Main Library，Pei Core BB Proxy Main Library 驱动作用是加载特殊函数的接口，为了之后的模块可以方便调用相应的程序接口。Pei Core BB Proxy Main Library 为 Pei BB Proxy Main Library 提供需要的函数接口，然后 Pei BB Proxy Main Library

再把 MDE Library 的函数接口转移到 Pei Core BB Proxy Main Library 驱动中，以方便下个阶段的调用。

第三类的模块由 Pei Core BB Test Case Main Library 和 Pei BB Test Case Main Library 组成，它们的组成类似 Pei Core BB Proxy Main Library 和 Pei BB Proxy Main Library，Pei Core BB Test Case Main Library 将装载分析案例到驱动内，Pei BB Test Case Main Library 装载测试的代码，使得其他各阶段可以调用。

4.1.3 Debug 模块的实现

论文工作对测试模块 Debug Test Library 按下面的流程实现。EDK II 的开发及测试环境中，调试用到的测试库，通过返回指针来返回 ASSERT ，其目的是提供了在不同的阶段，其功能不变的调试接口函数，该调试函数基于 UEFI BIOS 的启动流程中任何阶段都可以对各模块通过对 Set_return_point 设置相应程序的返回点。如 Pei 的阶段，Dxe 阶段和 Runtime 阶段。这样就对 SetJump 及 LongJump 的服务进行相应的封装。函数的返回点实现是通过宏调用 SetJump 保存处理器信息来得到共享内存的数据，设置 SetJumpFlag 参数进而实现的。当 SetJumpFlag 参数为 0，说明第一次的调用 SetJump 成功。SetJumpFlag 参数为 1，说明了参数基于 JumpToReturnPoint 返回。SetJumpFlag 参数为 2，说明寻不到共享的内存中对应参数的数据。JumpToReturnPoint 函数结构的申明如下。

```
VOID
  EFIAPI
  JumpToReturnPoint (
      VOID
      )
  {
      BASE_LIBRARY_JUMP_BUFFER                *DebugTestLibJumpBuffer;
      BASE_LIBRARY_JUMP_BUFFER                *JumpBufferWithZero;
  }
```

此过程相当于系统触发了一次中断，处理器从用户的模式切换至系统模式下，CPU 执行的第一件事情既是保存现场，然后去中断的向量表里去找触发了中断所对应的事件函数功能。在 EDK 模块里用 SET_RETURNPOINT 来保存已知共享内存中 SetJumpFlag 的值。然后由 JumpToReturnPoint 函数找到需要运行的函数接口，并准确地找到需要的执行函数。它的功能是通过搜索已知共享内存来获取处理器情况，然后调用 LongJump 服务，恢复返回值 1。从测试案例中获取测试参数的 GetBbTestParametersFromTestCase 函数实现如下。

```
EFI_STATUS
EFIAPI
GetBbTestParametersFromTestCase    (
```

```
  OUT   EFI_GUID                          **TestGuid,
  OUT   EFI_BB_TEST_ENTRY_FIELD           **TestEntryField,
  OUT   EFI_BB_TEST_PROTOCOL_FIELD        **TestProtocolField
  );
```

该函数功能是从分析函数中读取测试的库，测试的入口等信息，其中 TestGuid 是测试案例的 GUID，TestEntryField 是记录所测案例所属的库，TestProtocolField 是记录测试案例属于的协议。EDK 开发测试环境中，调试测试库通过提供所定的返回指针返回 ASSERT ，其目的是提供一个在不同阶段，其功能不变的调试函数，也就是说这个调试函数在 UEFI BIOS 启动过程中的任何一个阶段均可以对不同的模块设置程序返回点，函数具体操作实现如下。

```
Status = GetBbTestParametersFromTestCase   (
            &TestGuid,
            &TestEntryField,
            NULL                                  //it is not useful in PEI phase.
            );
if  (EFT_ERROR   (Status))    {
    return   Status;
}
```

4.1.4 Proxy 模块的实现

Proxy 模块分 Pei Core BB Proxy Main Library 和 Pei BB Proxy Main Library 。如上所提的 Pei Core BB Proxy Main Library 驱动是用来加载一些特殊函数接口的驱动，为了能让大部分函数接口可以被传递给下一个阶段的模块使用。Proxy Driver 的作用是把一个单元库封装到一个独立的模块中。这个模块中包括了一些待测试的 API。其中没有 Driver 的意义，它只是提供一个临时 API 的活动场所。这两个库在 PEI 的阶段提供了代理服务，MED Library 的 API 地址被放到对应的这些空间里，这类 API 接口是重定向在这些空间的，如 C 语言里的指针。这样看来，此两驱动均是为映射对应 API 接口的作用。Pei Core BB Proxy Main Library 函数实现如下。

```
EFI_STATUS
EFIAPI
PeiCoreBBProxyMain   (
    IN EFI_PEI_STARTUP_DESCRIPTOR   *PerStartupDesriptor ,
    IN VOID                         *OldcoreData
    )
{
  EFI_STATUS     Status;
  Void           *StartAddr;
  VOID           *ProxyInterface;
```

```
  UINTN          ProxyInterfaceLength;
  EFI_GUID       TestGuid;

  proxyInterface  =  NULL;
}
```

Pei Core BB Proxy Main Library 中的 PeiStartupDescriptor 启动信息指针，OldCoreData 指针指向之前的初始化所用的核心部分，此函数需要接受参数传递的地址，并且为初始化下个函数 Pei BB Proxy Main Library 做准备，只有初始化了之后才可执行。Pei BB Proxy Main Library 的函数实现如下。

```
EFI_STATUS
EFIAPI
PeiBBProxyMain  (
    IN          EFI_PEI_FILE_HANDLE              FileHandle
    IN CONST EFI_PEI_SERVICES                    **PeiServices
    )

{
  EFI_STATUS     Status;
  Void           *StartAddr;
  VOID           *ProxyInterface;
  UINTN          ProxyInterfaceLength;
  EFI_GUID       TestGuid;

  ProxyInterface = NULL;
}
```

Pei Core BB Proxy Main Library 为 Pei BB Proxy Main Library 准备好一些基本的函数接口，然后 Pei BB Proxy Main Library 再去将大部分的 MDE Library 函数接口装载在这个驱动中以便调用。Pei Core BB Proxy Main Library 功能是 Pei Core BB Test Case Main Library drive 入口的地址，另一个 GetBbTestParametersFromProxy，通过 proxy drivers 插入其相应测试用地 GUID。而最后是 InitializeProxyInterface，它负责了初始化测试程序的接口对应的函数。因此，函数调用之前需要判断接口能否被调用，这里用 malloc 函数，若返回值为 NULL，说明已经没内存可以分配使用。

4.1.5 Test 模块的实现

API 性能测试模块里设计的 Test 模块分 Pei Core BB Test Case Main Library 和 Pei BB Test Case Main Library。前面详细介绍了 Pei Core BB Proxy Main Library 和 Pei BB Proxy Main Library 的区别与联系，Proxy Driver 和 Test Driver 之间的区别很

简单。一个是 proxy 类型的，另一个 test 类型的，一个是提供源 API 的地址的，另一个分析案例的地址的。案例连接对应的 API 地址对应看属于 Pei Core BB Proxy Main Library 还是 Pei BB Proxy Main Library。Pei BB Test Case Main Library 的实现如下。

```
EFI_STATUS
EFIAPI
PeiBBProxyMain   (
    IN EFI_PEI_FILE_HANDLE              FileHandle
    IN CONST EFI_PEI_SERVICES                   **PeiServices
    )

{
  EFI_STATUS                              Status;
  EFI_GUID                               *TestGuid;
  EFI_BB_TEST_ENTRY_FIELD                 *TestEntryField;
  VOID                                   *TestInterface;
}
```

Pei Core BB Test Case Main Library 的实现部分如下。

```
EFI_STATUS
EFIAPI
PeiCoreBBProxyMain   (
    IN EFI_PEI_STARTUP_DESCRIPTOR   *PerStartupDesriptor ,
    IN VOID                         *OldCoreData
    )
{
  EFI_STATUS                              Status;
  EFI_GUID                               *TestGuid;
  EFI_BB_TEST_ENTRY_FIELD                *TestEntryField;
  VOID                                    *TestInterface;
}
```

Pei Core BB Test Case Main Library 将分析案例装载到驱动中后，Pei BB Test Case Main Library 装载测试代码，以便在其它各阶段调用。

论文的完成过程中 debug 调试起到至关重要的作用，如何发现问题并调回源代码进行修改显得很关键。在测试 Strcpy 函数的过程中就发生了函数死循环的情况。需要做的是在函数死循环之前设置断点。考虑到 BIOS 启动到 PEI 阶段后先是加载 PEIM，可以在程序到死循环前的任意地方来设置断点，就是在离死循环的程序附近函数里设置一个断点。很显然，最近的就是 Pei BB Test Case Main Library，在函数里插入_asm int 3。重新编译程序代码，并在 NT32 的模拟平台下运行。程序运行到要调试的地方后，会触发到执行的异常，选择 Cancle 和 New Instance of Visual Studio 2005 选项，进入到 DEBUG 的环境，Pei BB TestCase 函数设置断点的

代码如下。

```
EFI_STATUS
EFIAPI
PeiBBProxyMain    (
    IN EFI_PEI_FILE_HANDLE                FileHandle
    IN CONST EFI_PEI_SERVICES                    **PeiServices
    )

{
  EFI_STATUS                                  Status;
  EFI_GUID                                    *TestGuid;
  EFI_BB_TEST_ENTRY_FIELD                     *TestEntryField;
  VOID                                        *TestInterface;

_asm    int 3;
}
```

程序重新编译运行后发现不能正常执行，弹出的出错的对话框如图 4.2 所示。

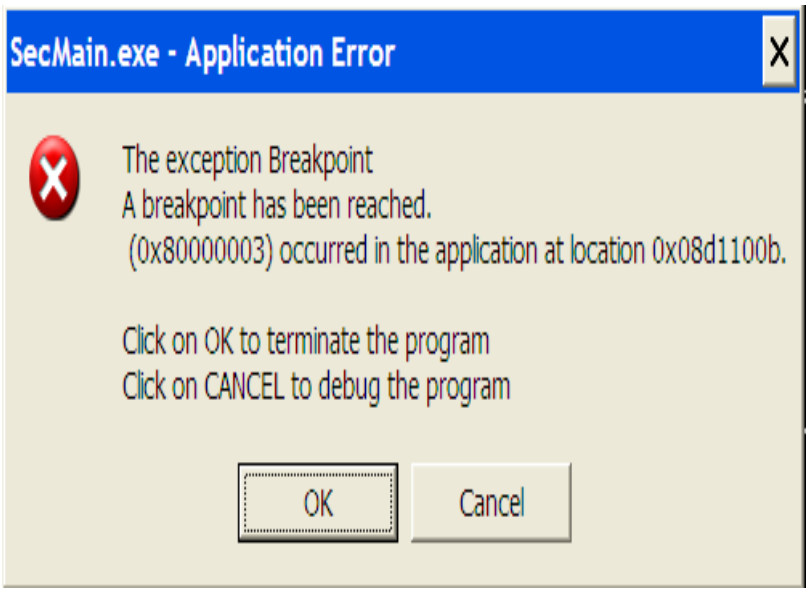

图 4.2 SecMain 进入死循环

点击 Cancel 键进入 VS2005 按 F9 进行调试，如图 4.3 所示。

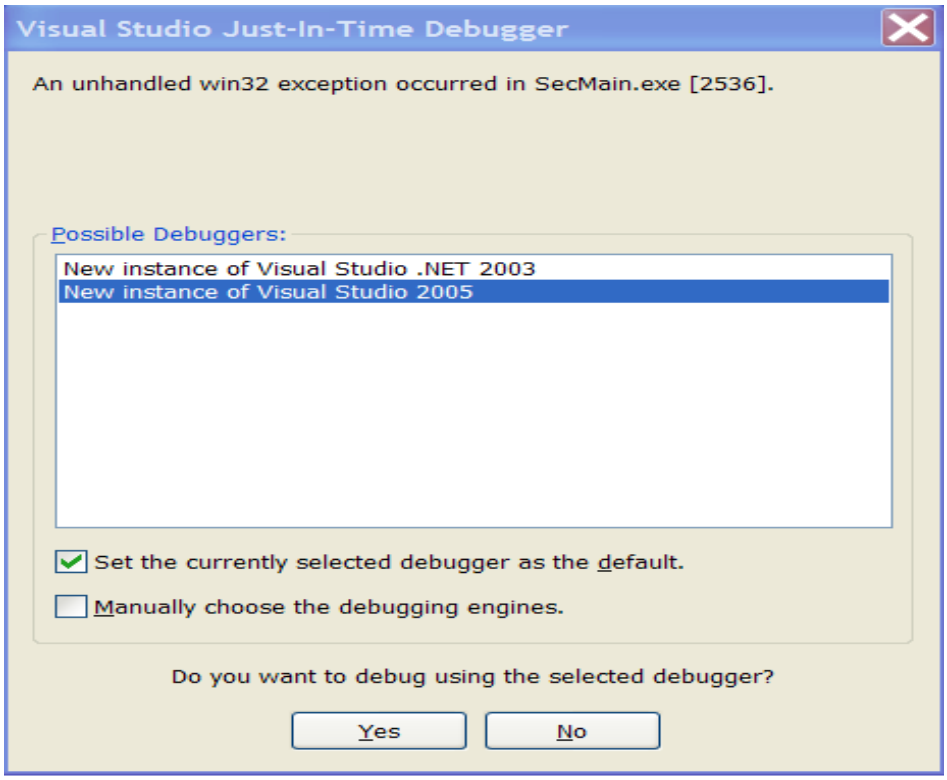

图 4.3 调试窗口

接着选择点击 Yes 进入调试窗口，在 VS2005 里面可以跟踪源代码是如何一步一步执行的。可以在 CPU 出现死循环的时候查看到对应出问题的代码。分析出引起程序发生死循环的原因，进而反复修改直到问题解决为止。

4.2 PEI 阶段底层 API 性能测试在 NT32 平台的实现

4.2.1 测试流程分析

PEI 阶段是整个 BIOS 启动过程中的一个阶段，其主要功能是为下一个阶段初始化硬软件环境。PEI 阶段里面需要对 MDE 提供的 API 进行分析，实际工作中 API 没有实现其功能这对下阶段的工作将带来不可想象的灾难。基本 base 函数不能实现 copy 功能，内存信息不能传递到下一个阶段等等多种情况的发生都会对后期的驱动初始化环境带来问题。在这样的背景下，对 PEI 的阶段 MDE 提供的 API 性能测试有个不可替代的作用。PEI 阶段到 DXE 的阶段转化是单向过程，在 DXE 阶段初始化并装入程序后，PEI 阶段的代码不可再用，此时 DXE 就相当于设备完整的操作系统环境[22]。而 PEI 的阶段的代码不能再使用，其提供的 API 服务就很有必要在之前就完成其功能性能的测试，以满足系统的需要。论文设计的测试的框架模块分为 DebugTestLibJumpBuffer、Porxy Driver、Test Driver，测试设计的框

架如图 4.4 所示。

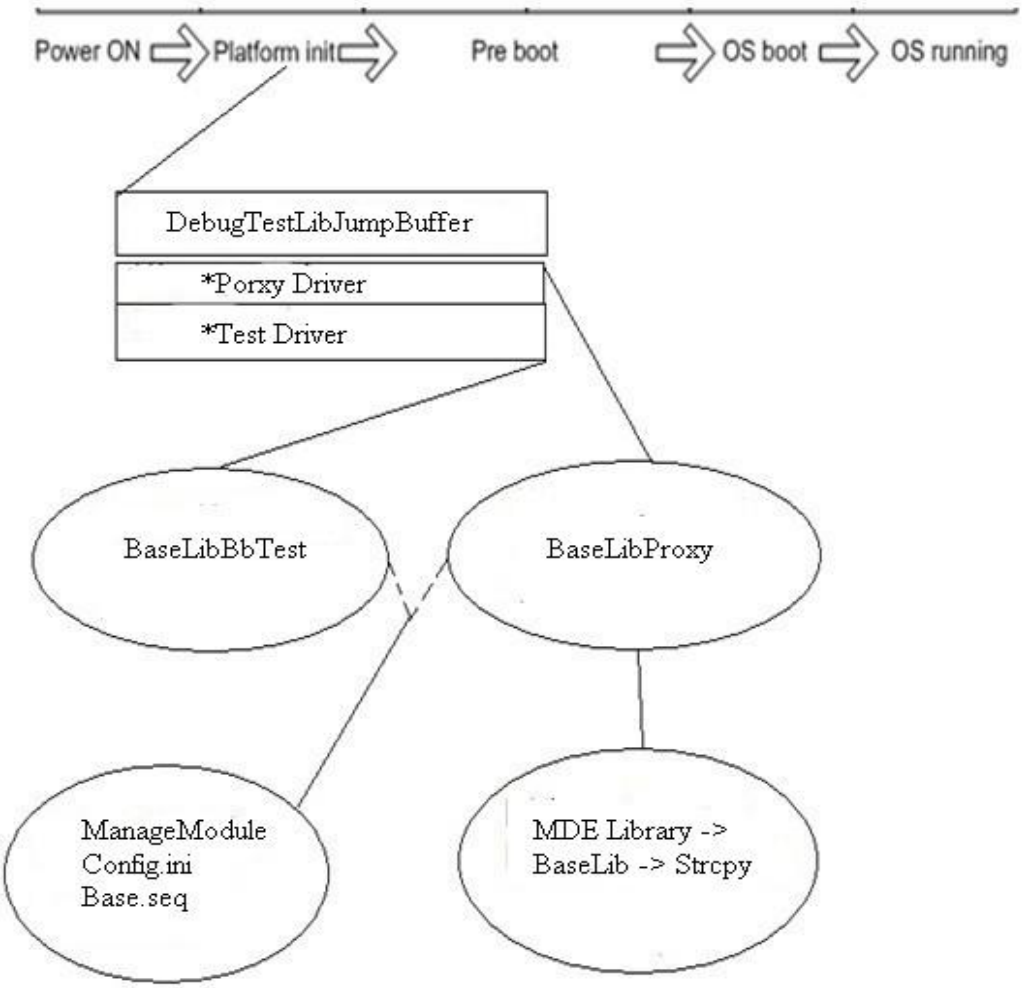

图 4.4 分析代码的工作流程

如图中所示一个系统从上电到 boot 到 OS 的流程。在 BIOS 上电后，首先进行安全检查，接着进行一些初始化，在 PI(Platform Initiation)阶段，整个阶段包含了 PEI 阶段。PEI 阶段需要做的事情就是初始化 MDE 函数接口，图中描述了测试程序的设计框架，通过嵌入一个模块完成对 MDE Library 中 Strcpy 的性能进行分析。设计的思路如下，测试的模块分为 DebugTestLibJumpBuffer、Porxy Driver、Test Driver 三个部分，Porxy Driver 提供存放 BaseLibProxy 对应的 API 的地址空间，通过设计实现的 ManageModule 模块与 BaseLibTest 里的地址进行比对测试，从而得到 API 性能测试分析结果。该功能模块的难点就是在于如何实现 ManageModule 的功能。

论文的工作选定了在 NT32 的平台上面模拟出整个分析测试的过程，NT32 是用 VS2005 提供的库为函数库的，其可以编译成一个完全仿真模拟的 BIOS 环境。任何装载了 VS2005 软件的 PC 都可以模拟仿真的 EFI BIOS 环境，其 NT32 的源代

码的框架如图 4.5 所示。

图 4.5 NT32 平台的源代码

Mdepkg 是底层的 API 函数库集合，MdeTestPkg 是分析的函数库集合，Nt32Pkg 是模拟 NT32 环境所需代码库。它是其它平台对应的代码库的集合，例如 ArmPkg、BearleBoradPkg 等等。在 Nt32Pkg 下实现的 API 性能分析，前提是要在 Nt32Pkg 嵌入对应测试的 MdeTestPkg 代码，此类代码都支持在 32 位的模拟环境里进行编译。Nt32Pkg 文件夹里有两种重要的文件，一个 Nt32Pkg.dsc 文件，另一个 Nt32Pkg.fdf 文件，Nt32Pkg.dsc 文件的作用是来制定 Nt32 的平台下编译得到的二进制的代码，Nt32Pkg.fdf 文件的作用则是把编译得到的二进制码进行对应封装，就是把得到的二进制码重新组成映像文件。

4.2.2 分析库嵌入到 NT32 平台

论文的工作首先是将 MdeTestPkg 测试代码嵌入到 Nt32Pkg.dsc 文件中。Nt32Pkg.dsc 文件所制定的 NT32 的平台编译得到的二进制码，这有这样才能让其在 NT32 的平台上被识别。将分析源代码嵌入到 NT32 的平台过程是通过调试库、代理驱动由分析函数入口嵌入执行的，这个过程是自动执行的。调试库文件是 TestPkg/TestInterface/Pei/DebugTestLibJumpBuffer/DebugTestLibJumpBuffer.inf，代

理驱动的文件是 TestPkg/Proxy/BaseLibProxy/Pei/BaseLibProxy.inf，分析程序函数入口文件是 TestPkg/TestCase/BaseLib/Pei/BaseLibBBTest.inf。

论文接着的工作是将编译好的二进制代码放入到映像文件中，实现的代码如下。

```
INF MdeTestPkg/TestInfrastructure/Pei/DebugTestLib/DebugTestLibJumpBuffer.inf
INF MdeTestPkg/Proxy/BaseLibProxy/Pei/BaseLibProxy.inf
INF MdeTestPkg/TestCase/BaseLib/Pei/BaseLibProxy.inf
```

论文完成的过程，对 Nt32Pkg.dsc 和 Nt32Pkg.fdf 关系的理解可以拿盖房子来打个比方。盖房子需要搬运工，搬运工会根据老板给的建筑材料说明将砖头，瓦片等材料搬到建筑地上，然后由瓦工根据房屋设计图盖成大楼。Nt32Pkg.dsc 相当于搬运工手里的材料明细 ，Nt32Pkg.fdf 相当于瓦工手上的房屋设计图，它们协同来搭建整个大楼。

4.2.3 分析库编译到 NT32 平台

完成了上面的工作之后，开始实施分析库编译。编译首先设置环境变量，即指定编译路径，编译选项，编译工具等。具体步骤如图 4.6 所示。

```
\SysInternals;c:\OSHook\CPSTools\WInXPSupportTools;c:\ProgramFiles\UltraEdit;c:\Progr
amFiles\Microsoft\SQLSeruer\90\Tools\binn\;c:\WINDOWS\system32;c:\WINDOWSS;C:\
WINDOWS\System32\Wbem;c:\OSHook\CPSTools;c:\OSHook\CPSTools\SysInternals;c:\
OSHook\CPSTools\WInXPSupportTools

     WORKSPACE          = c:\Q1Release
EOK_TOOLS_PATH        = c:\Q1Release\BaseTools                    //编译

WORKSPACE              =c:\qlrelease
ECP_SOURCE            =c:\qlrelease\edkcompatibilitypkg
EDK_SOURCE             =c:\qlrelease\edkcompatibilitypkg
EFI_SOURCE            =c:\qlrelease\edkcompatibilitypkg
EDK-TOOLS_PATH        =c:\qlrelease\basetools

TARGET_ARCH            =IA32                                       //编译平台架构
TARGET                 =DEBUG                                      //调试
TOOL_CHAIN_TAG         =MYTOOLS                                    //编译工具

Active  Platform     =c:\qlrelease\Nt32Pkg\Nt32Pkg.dsc
Flash Image Definition   =c:\qlrelease\Nt32Pkg\Nt32Pkg.fdf

     Processing   meta-data...
```

图 4.6 编译 NT32 平台

分析库程序的核心代码如下。

```
EFI_STATUS
EFIAPI
StrCpyBbTestEntry   (
  IN VOID                          *This,
  IN VOID                          *ClientInterface,
  IN EPI_TEST_LEVEL                TestLevel,
  IN EPI_HANDLE                    SupportHandle
  )
{
 EPI_TEST_ASSERTION                AssertionType;
 BASE_LIBRARY_INTERFACE            *BaseLibraryInterface;
 UINTN                             CheckPoint;
 UINTN                                 Index;
 CHAR16                            *DestinationArray[4];
 CHAR16                            *SourceArray[4];
 CHAR16                            *ReturnValue;
 INTN                              DestinationResult;
 INTN                              ReturnResult;

 BaseLibraryInterface = (BASE_LIBRARY_INTERFACE *) clientInterface;
}
```

代码中 EFI_STATUS 和 EFIAPI 是 EDK 里面定义的一个宏，EFI_STATUS 宏的申明是 typedef unsigned long EFI_STATUS，表示以下函数的返回值的类型是一个无符号的长整型。EFI_API 没有特殊意义，它仅是用来标示此处将实现对应函数，其意义就如路标一样，告知驾驶员驾驶的方向。在 API 的入口处 IN 、OUT 等等这样类似宏的作用和 EFIAPI 功能是一致的，IN 代表输入参数，而 OUT 代表的函数的输出参数。但代码中出现的 UINTN、CHAR16、INTN 等宏就不一样了，它们通过 typedef 把类型重定义。EDK 里统一了申明类型，论文中把 C 语言里的类型重定义，这样可以方便的进行应用。代码实现有两个地方着重进行了分析设计。

第一个 AsertionType，该变量 EFIASERTION 是定义的返回值，函数设定了两个返回值，要么成功，要么失败。实现的代码如下。

```
if  (  (  DestinationResult  |  ReturnResult  )  != 0  )
 {
  AssertionType = EFI_TEST_ASSERTION_FAILED;
}   else
 {
  AssertionType = EFI_TEST_ASSERTION_PASSED;
}
```

第二个是 BASE_LIBRARY_INTERFACE，这个定义的*BaseLibraryInterface 指针指向 Proxy 里 MDElib 的 API 应用，实现的代码如下。

```
BASE_LIBRARY_INTERFACE *Interface;

Interface          =  (BASE_LIBRARY_INTERFACE *)  *ProxyInterface;

//
//String functions
//

Interface->StrLen                                =          Strlen;
Interface->StrCmp                                =          StrCmp;
Interface->StrCmp                                =          StrCmp;
Interface->StrCmy                                =          StrCmy;
Interface->StrSize                               =          StrSize;
Interface->StrCmp                                =          StrCmp;
```

在函数 InitializeBaseLibrary_Interface.c 里面将 Baselib 里的函数初始化并指向另一个地址，在 BaseLibBbTestString.c 里得到该地址后可以对它对应的 API 性能进行测试分析，函数的代码如下。

```
LibRecordAssertion (
mStandardLibHandle,
AssertionType,
gBaseLibBbTestStringAssertionGuid001,
L”BaseLib_StrCpy.Destination is NULL”,
L”%a:%d:It should be ASSERT() and ReturnValue should be 1 but it returns %d.”,
_FILE_,
_LINE_,
AssertReturnValue
);
```

其中 LibRecord_Assertion 函数的功能类似于常用的 C 语言的 print 函数，不过 LibRecord_Assertion 函数里定义了许多类型的输出，并且将这些输出类型规范统一，将程序测试分析结果定义到里面 AssertionType，AssertionType 是枚举变量，该变量决定了下面结果输出是成功或是失败，枚举变量在这里起到关键作用。另外还输出了 eLibBbTestFunctionAssertionGuid，它是测试分析程序的 GUID，文中描述了 GUID 的作用，但在机器语言里仅可以通过 GUID 来查找程序入口并记录分析的数据。ReturnValue 是 Strcpy 函数的原型中的返回地址，字符串复制结束后返回对应的类型地址，获取该地就可比较 Stcpy 的函数功能能否正确的实现字符串的复制。

4.2.4 测试工具的设计与实现

在 Shell 下实现的 ManageModule 测试工具代码如下。

```
Status = InitManageModule();
If (EFI_ERROR(Status))
{
Print(L”Init Fail!\n”);
goto end;
}
Status=RecordUserSelection();
if (EFI_ALREADY_STARTED==Status)
{
    Status=GenerateReport();
    if (EFI_ERROR(Status)){
    }
    }
Status = SelectedItemAvailable(&ItemRecord);
if (EFI_NOT_FOUND==Status)
 {
     NeedReset=FALSE;
     }
if(Needreset){
     BurnFd(imageHandle,Fdlocation[ItemRecord.ItemId]);
     gsT->RuntimeServices->ResetSystem(EfiResetCold,EFI_SUCCESS,0,NULL);
     }
     //return Status;
     end:Print(L"\n\n");
     Status=Fremove(CHECK_LIST_FILE);
     return Status;
}
```

函数执行初始化，接着测试的程序里读取要分析 GUID，通过所获取变量把 GUID 值写到 BIOS 的 PEI 阶段对应的变量里，重新运行软件工具，测试的程序会完成刚写入 GUID 的对应的 API 的测试。在 API 入口处的 IN 还有 OUT 等等这样类似的宏的意思和 EFIAPI 的用处是一样的，IN 表示输入的参数，OUT 是函数输出的参数。但代码中出现的 UINTN、CHAR16、INTN 等宏就不一样了，他们是经过 typedef 将类型重新定义了一下。其中 InitManageModule 即是初始化的工具 ManageModule，初始化如成功则程序继续执行，若是失败则会打印出失败的提示，最后退出程序。RecordUserSelection()函数的作用是到文件里取需要进行测试 API 对应的 GUID，开发者需要写进待分析 API 对应的 GUID，ManageModule 把 GUID 的值写入 BIOS 中，接着 BIOS 会根据链接寻找到程序对应的入口地址，进而完成测试过程。整个分析流程要执行三次 ManageModule 工具的。前两次已经描述了，第三次是实现获取分析的数据，分析结果如图 4.7 所示。

```
918D5531-4DAA-4D5C--DEFD7A8CCB/BaseLib_StrCpy-withtwounicode strings:
c:\qlrelease\MdeTestPkg\TestCase\BaseLib\BaseLibBbTestString.c:474:SourceString=,
DestinationString=,ReturnValue=

918D5531-4DAA-4D5C-B33FAILURE BaseLib_StrCpy-with two unicode strings:
c:\qlrelease\MdeTestPkg\TestCaseCommon\BaseLibBbTestString.c:474:SourceString=
abcdefg, DestinationString, ReturnValue=

918D5531-4DAA-4D5C-B33682:PASS|BaseLib_StrCpy-with two unicode strings:
c:\qlrelease\MdeTestPkg\TestCase\BCommon\BaseLibBbTestString.c:47SourceString=
  DestinationString=, ReturnValue=

918D5531-4DAA-4D5CCCB82:FAILURE BaseLib_StrCpy-with unicode strings:
c:\qlrelease\MdeTestPkg\TestCase\Pei\Common\BaseLibBbTestString.SourceString=
hijklmn, DestinationString=hijklmn, ReturnValue=
```

图 4.7 分析结果(1)

在函数 InitializeBaseLibraryInterface.c 里面将 baselib 里面的函数进行初始化指向一个地址，在 BaseLibBbTestString.c 里面拿到这个地址就可以这个 API 进行分析。从上图中可以看到，结果有两个输入的参数返回了失败的值。说明了 Strcpy 函数本身可以有待解决的问题。完成论文工作的过程是再一次在 Shell 下运行测试，看看测试的结果。在 Shell 下执行测试模块结果如图 4.8 所示。

```
startup.nsh> cd NT32PEITEST
startup.nsh> ManageModule.efi
918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASSIBaseLib_StrCpy – Unicode stri
ngs ;
    c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibTestString.c:474
:SourceString=, DestinationString=, ReturnValue=
918D5531-4DAA-4D5C-B336--DEFD7A8CCB82:FAILUREBaseLib_StrCpy-Unicode
Strings ;
     c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibTestString.c:474
:SourceString=abcdefg, DestinationString=abcdefg, ReturnValue=
918D5531-4DAA-4D5C-B336--DEFD7A8CCB82:FAILUREBaseLib_StrCpy-Unicode
Strings ;
     c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibTestString.c:474
:SourceString=hijklm, DestinationString=hijklm, ReturnValue=
```

图 4.8 分析结果(2)

查看函数的具体实现以论证程序出错的地方，打开 Strcpy 函数的代码如下。

```
CHAR16 *
EFIAPI
StrCpy (
    OUT       CHAR16                  *Destination,
    IN        CONST CHAR16            *Source
    )
   {

   //
   //Destination cannot be NULL
   //
   ASSERT (Destination != NULL);
   ASSERT (((UINTN) Destination & BITO)==0);

   //
   //Destination and source cannot overlap
   //
   ASSERT ((UINTN)(Destination - Source)>StrLen (Source));
   ASSERT ((UINTN)(Source - Destination)>StrLen (Source));

   while (*Source !=0){
     *(Destination++) =*(Source++);
     }
   *Destination =0;
   return Destination;
   }
```

函数的实现运用函数指针，看它是否返回正确地址而实现的，此函数的内部定义两个 char 类型指针，Source 是输入的，Destination 是输出的，Source 指针用 CONST 修饰说明其实不可被修改的，目的是保护传进的字符串的值不会变。下面两个宏来判断对应的指针是否为 NULL。若为 NULL 则 程序产生异常并退出，另外下面的两个宏来判断内存重叠是否发生。若它们的地址之间长度没源字符串长度大的话，会造成内存的覆盖。若目标的字符串覆盖了原来的字符串，那会不可容忍的，安全检查结束后开始执行字符串的复制。具体实现首先判断 while (*Source != 0)，当*Source 不为 0 时，程序会继续执行，当*Source=0 时，程序跳到*Destination = 0;对 Destination 末尾赋的值为 0，是字符与字符串重要的区别，到了最后在 return Destination 出错了，说明上面的 while 循环判断的过程中，其内部会让 Destination++，进而 Destination 地址里的值不得而知，所以需要先要将 Destination 地址先保存一份，以方便之后的程序返回查看，修改之后代码的实现如下。

```
CHAR16 *
EFIAPI
StrCpy (
     OUT       CHAR16                    *Destination,
     IN        CONST CHAR16              *Source
     )
    {
     CHAR16                             *ReturnValue;
        //Destination cannot be NULL
    ASSERT (Destination != NULL);
    ASSERT (((UINTN) Destination & BITO)==0);

        //Destination and source cannot overlap
    ASSERT ((UINTN)(Destination - Source)>StrLen (Source));
    ASSERT ((UINTN)(Source - Destination)>StrLen (Source));

    ReturnValue = Destination;
    while (*Source !=0){
       *(Destination++) =*(Source++);
       }
    *Destination =0;
    return ReturnValue;
    }
```

修改完需要重新编译，若仅修改了源代码，却没进行重新编译是不行的。原因是代码一旦被编译后在内存里是不会改变的。 重新编译的步骤与上面提到的相同。重新测试结果如图 4.9 所示。

```
918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASSIBaseLib_StrCpy – Unicode stri
ngs ;
    c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibTestString.c:474
:SourceString=, DestinationString=, ReturnValue=
918D5531-4DAA-4D5C-B336--DEFD7A8CCB82:FAILUREBaseLib_StrCpy-Unicode
Strings ;
     c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibTestString.c:474
:SourceString=abcdefg, DestinationString=abcdefg, ReturnValue=
918D5531-4DAA-4D5C-B336--DEFD7A8CCB82:FAILUREBaseLib_StrCpy-Unicode
Strings ;
     c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibTestString.c:474
:SourceString=hijklm, DestinationString=hijklm, ReturnValue=
```

图 4.9 分析结果(3)

在 Shell 环境执行函数的结果如图 4.10 所示。

```
Startup.nsh>cd.NT32PEITEST
Startup.nsh>ManageModule.efi
918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS|BaseLib_StrCpy-with two unicode strings:
  c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
SourceString=,DestinationString=,ReturnValue=
918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS|BaseLib_StrCpy-with two unicode strings:
  c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
SourceString=abcdefg,DestinationString=abcdefg,ReturnValue=abcdefg
918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS|BaseLib_StrCpy-with two unicode strings:
  c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
SourceString=,DestinationString=,ReturnValue=
918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS|BaseLib_StrCpy-with two unicode strings:
  c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
SourceString=hijklmn,DestinationString= hijklmn,ReturnValue= hijklmn

fsnt0:\NT32PEITEST>
```

图 4.10 分析结果(4)

上面论文工作是对 Strcpy 进行功能的分析，而对它的性能分析基于功能分析之上完成的，即性能测试分析的前提是功能分析完成，并执行得很好。工作中设计把性能分析分为两种情况，第一种情况对 Strcpy 函数输入错误的值，来看它是否有错误提示的返回。第二种情况就是对 Strcpy 函数功能实现的内部代码分析，这种情况就要 MDE_Library 里面的 string.c 里面 Strcpy 的实现，其源代码的实现思想如下。

函数实现通过函数的指针，来返回正确地址实现其目的，函数定义两个 char 类型指针，Source 是输入的，Destination 是输出的，Source 指针用 CONST 进行修饰，表明它的值不可修改，目的是保证传入的字符串不可变动。

接着定义另一个指针，定义 Destination 不能为空，在程序中让 Destination 赋一个空指针，接着实现具体的函数部分[22]。分析程序对 Strcpy 的性能分析的实现代码如下。

```
EFI_STATUS
EFIAPI
StrCpyBbComformanceTestEntry(
IN VOID                                 *This,
```

```
IN VOID                          *ClientInterface,
IN EFI_TEST_LEVEL                TestLevel,
IN EFI_HANDLE                    SupportHandle
)
{
EFI_TEST_ASSERTION               AssertionType;
BASE_LIBRARY_INTERFACE           *BaseLibraryInterface;
UINTN                            CheckPoint;
UINTN                            Index;
CHAR16                           *DestinationArray[4];
CHAR16                           *SourceArray[4];

UINTN                            AssertReturnValue;
UINT32                           MaxStringLength;
CHAR16                           *SourceString;
CHAR16                           *DesString;
UINTN                            SourceLength;
CHAR16                           *UnboundaryDestinationString;
CHAR16                           *UnboundarySourceString;
CHAR16                           *AnotherSourceArray[2];
CHAR16                           *AnotherDestinationArray[2];
}
```

函数申明及功能分析的代码类似，下面是分析部分核心的代码。

```
if(AssertReturnValue == 0){
  BaseLibraryInterface->StrCpy(NULL,L”abcd”);
  AssertionType = EFI_TEST_ASSERTION_FAILED;
}else if (AssertReturnValue == 1){
  AssertionType=EFI_TEST_ASSERTION_PASSED;
}else {
  //
  //Must Never Happen
  //
  AssertionType = EFI_TEST_ASSERTION_FAILED;
  CpuDeadLoop();
}

LibRecoudAssertion(
  mStandardLibHandle,
  AssertionType,
  gBaseLibBbTestStringAssertionGuid001,
  L”BaseLib_StrCpy – Destination is NULL”,
  L”%a;%d;It should be ASSERT() and ReturnValue should be 1 but it returns %d.”,
  __FILE__,
  __LINE__,
```

```
  AssertReturnValue
);
```

函数实现运用了函数指针，用指针来返回正确地址，函数里定义两个 char 类型指针，Source 是输入的，Destination 是输出的，其中的 Source 指针用 CONST 修饰。

Strcpy 性能分析和 Strcpy 功能分析的区别就在于对这个函数输入的参数，要是将字符串‘abcd’赋值给空类型 NULL，这个显然是不可以的。对 Strcpy 的性能分析就是看 MDE Libraray 里面的 Strcpy 是如何处理这种情况的，这是性能分析重要的地方。若函数正确返回了错误的值，则说明 Strcpy 函数性能，它在接受错误参数的情况下页是能够正常运行。这就完成了 Strcpy 函数容错性的分析。论文工作完成的分析结果如图 4.11 所示。

```
c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestStringComformBCance.c:1478:It should NOT ASSERT() and ReturnValue should be 0 and it return 0, MaxStrLength=1000000, SourceLength=1000001

02705AA2B-5B91-4628-8005-A24D6E158D63:PASSlBaseLib_StrCpy – Source contains more than PcdMaximumUnicodeStringLength:
c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestStringComformance.c:1510:It should be ASSERT() and ReturnValue should be 1 but it returns 1, MaxStrLength=1000000, SourceLength=1000001

0DA285C5F-ACEA-4862-935F-51D3F7EE70E8:PASSlBaseLib_StrCpy – Source String is not aligned on a 16-bit boundary:
c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestStringComformance.c:1549:It should be ASSERT() and ReturnValue should be 1 but it returns 1,SourceString=79886464, DestinationString=79810560

07AEA9238-3E83-4BE8-837E-E3A38C21C272: PASSlBaseLib_StrCpy – Detination Stringng is not aligned on a 16-bit boundary:
c:\q1release\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestStringComformance.c:1579:It should be ASSERT() and ReturnValue should be 1 but it returns 1, SourceString=79814656, DestinationString=79802368

fsnt0:\NT32PEITEST>_
```

图 4.11 Strcpy 性能分析的结果

将 GUID 写进 BIOS 变量里，以上分析了 GUID 是开发者写入 BIOS 里的，采取的是测试工具将 GUID 值写进 BIOS 变量里，BIOS 重启后能获取变量的 GUID，进而得到待分析的数据的。输入 GUID 文件的过程代码如下所示。

“D915open2.fd”
“D915open2.fd” reboot

[BurnTool]
“” //烧写 flash 工具
[Option]
“” //烧写命令选项
[FdName]
//image 的名称
“.” Reboot //烧好后重启系统
[Sequence]
“BaseLib.seq” //GUID 容器

GUID 是通过*.seq 文件写进 BIOS 里的，例如 Base.seq 文件模型结构如图 4.12 所示。

```
[Test Case]
Revision=0x10000
Guid=bb31d31c-844d-478a-839d-dab839eebb26   //测试函数库入口的 GUID
Name=BaseLibBbTest
Order=0xf
Iteration=1

[Test Case]
Revision=0x10000
Guid=D9D0F0CD-0B27-4e58-B7F3-C524D4E82E9A          //测试函数接口的 GUID
Name=StrCpy Conformance
Order=0xf                                          //0xf 结尾表示执行
Iteration=1

[Test Case]
Revision=0x10000
Guid=0CFA7C49-03FD-45D3-A8E6-2D802A7135ED          //测试函数接口的 GUID
Name=StrCpy
Order=0x0                                          //0x0 结尾表示忽略
Iteration=1
```

图 4.12 ManageModule 工具配置文件 Base.seq

在这个 seq 文件中，对 BIOS 而言，明显有两个重要的数据，一个是 Guid，一个是 Order，Order 即命令，用户把 GUID 值写进 seq 文件并不一定记着让程序调用其相应测试的程序，可以通过 Order 值来判定，它的过程是程序装载待分析程序 GUID 的值，然后再与上 Order 的值，若结果为 1，即分析；若结果为 0，则丢弃。

4.3 RUNTIME 阶段底层 API 性能测试的实现

如何在 OS 下实现 UEFI BIOS 所提供的服务，取决于 OS 环境是否给出了相应的接口，Windows 内核的代码实现都是保密的，对于程序开发者来说要知道其内部的模块的实现是不现实的，所以要在用户态里获的底层的 API 应用是十分难的。，Linux 作为极新的操作系统，从 1991 年诞生算起，至今不过二十多年，它的发展及成长迅捷无比，无可争议地成为 OS 领域中的一匹黑马。到目前为止，Linux 在全世界范围装机的台数估计最低的也有 300 万，最高数字是 900 万。

作为一类 Unix 操作系统，Linux 强大的性能让其余品牌 Unix 失色不少。有专家认为，Linux 系统的推广与普及使它成了 Unix 市场最有活力的 OS。甚至连 Unix 之父 Dennis Ritchie 也认为 Linux 的特点新颖。Linux 公认为是唯一可冲破 Windows 的垄断文化的 OS。其代码的开源性也是选择 Linux 实现实时调用服务的重要原因。

论文将提出如何在 Linux 环境里实现 EFI BIOS 在 RUNTIME 阶段调用的方案，其技术难度是 BIOS 完成了初始化后如何将底层系统的参数传递给操作系统之前的用户。论文中采用的策略通过系统调用来实现，进而实现了操作系统调用底层的 API 功能。

系统调用将时间参数传递给用户空间后，用户空间通过地址强制转换，便可以知道，这个地址下面的其他参数地址，前提是要指定这个结构体的原型，不知道这个结构体的原型情况下是无法获得其中的参数地址。有了结构体原型，可以重新定义这个结构体变量，然后将得到的地址强制转换成一个结构体，并将其中的变量取出来赋值给新的变量，从而可以实现实时调用。实现框架如图 4.13 所示。

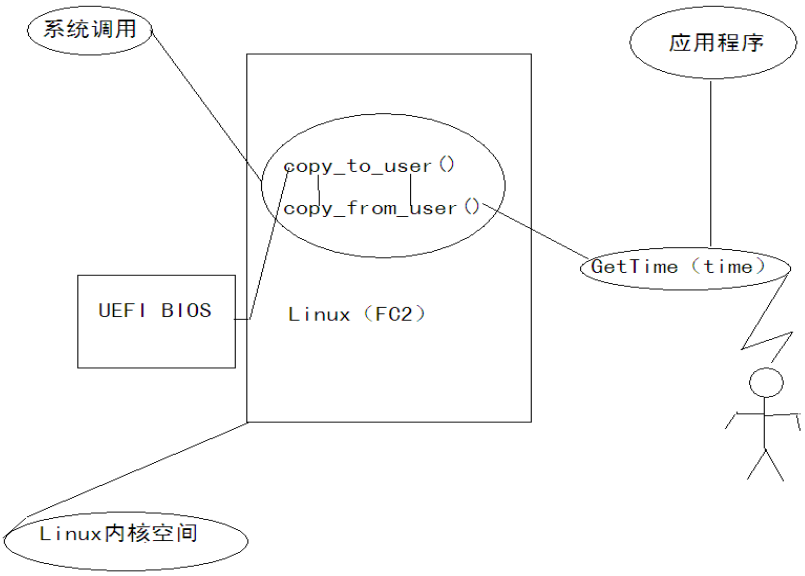

图 4.13 Runtime 阶段系统的调用实现架构

论文的工作首先是在 LINUX 内核中实现驱动，此驱动的目的是把底层 BIOS 接口信息传递入内核的空间里，驱动里实现一个内核于用户之间进行数据的交换功能，这样应用程序即可间接地访问 BIOS 的 API 的应用。

分析设计的整个过程程序流程如图 4.14 所示。

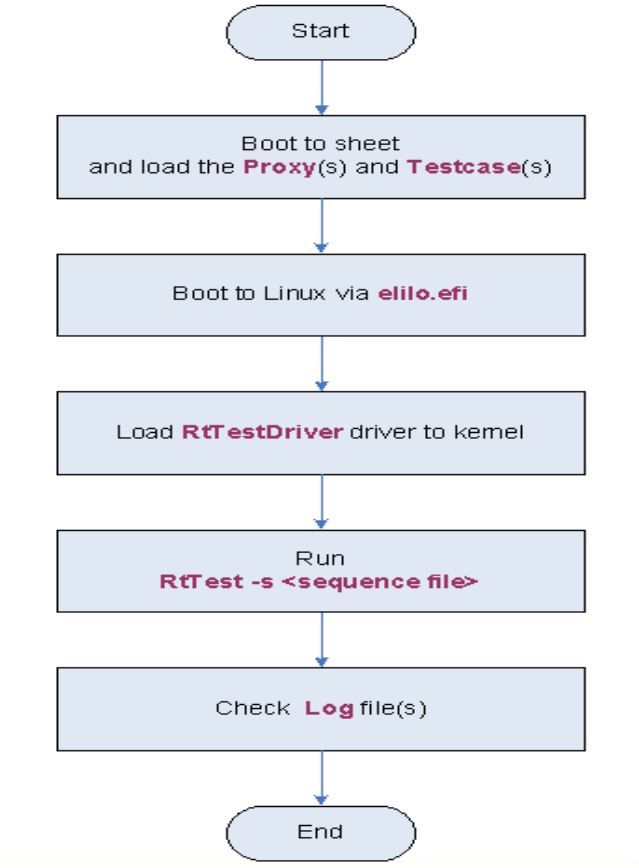

图 4.14 Runtime 阶段的分析案例实现架构

4.3.1 LINUX 系统调用分析

系统调用是操作系统供给用户调用的特殊接口。因而用户程序并可通过这些特殊接口获取操作系统的内核给予的相应服务。如用户可基于文件系统的相关调用来请求操作系统执行读写文件、退出文件等操作，可基于时钟相关连的系统调获得系统的时间、设置系统的时间等等。

逻辑上看，系统调用就是用户与系统之间相互交互的一个接口。系统调用将用户进程请求上传给系统内核，等内核处理完用户的请求后再把处理的结果送回到用户的空间。

系统服务需要用系统调用的方式提供于用户空间，其根本的原因是对系统实

现了保护。Linux 运行空间可分成内核空间和用户空间两种，它们运行在各自不同级别当中，在逻辑上是互相隔离的。用户进程一般情况是不许访问系统内核的数据，也无法用内核的函数，只可在用户的空间里操作数据，调用用户空间的函数。例如，程序 Hello World 执行就是典型的用户空间的进程，它的输出打印函数 printf 即属用户空间里的函数，其打印出的字符 Hello Word，此字符串同样也属用户空间的数据。

在特定的情况下，很多用户进程也要获取系统服务，即调用系统程序，只可以利用系统供给的用户特殊的接口，即调用系统调用，其特殊性在于限定了用户的进程进入系统内核具体的位置。换言之，用户访问系统内核路径事先是定好的，必须从规定的位置进入系统内核，并不是肆意地跳入系统内核中。这样的访问路径的限制才可以保证系统内核的安全。

论文的主要工作就是通过系统调实现函数功能。首先弄清楚系统调用的原理，学习在 Linux（FC2）下系统调用怎样将 OS 底层的提供 RUNTIME SERVICE 服务的 API 接口参数传递进用户空间，而系统调用的函数如下列举：access_ok()，copy_to_user()， copy_from_user,put_user,get_user 。 access_ok() 函数的原型： int access_ok(int type,unsigned long addr,unsigned long size) 函数 access_ok()用于检测指定的地址能否访问。此参数 type 定义了访问的方式，设为 VERIFY_READ（可读），VERIFY_WRITE（可写）。Addr 指的是待操作地址，size 指的是操作空间的大小（是以字节为单位）。函数若返回 1，则表示可访问，返回 0 则表示不可访问。copy_to_user()和 copy_from_user()函数的原型：unsigned long copy_to_user(void *to,const void *from,unsigned long len)；unsigned long copy_from_user(void *to,const void *from,unsigned long len)

此两函数作用是让内核空间和用户空间的数据进行交换。copy_to_user()作用是数据由内核空间复制到用户空间里，copy_from_user()作用是把数据由用户空间复制到内核空间里。第一个 to 参数定为目标地址用，第二个 from 参数定为源地址用，第三个 len 参数定为待拷贝数据的个数。函数在内部通过调用 access_ok()执行地址检查，其返回值就是未能被拷贝的字节个数。

get_user()和 put_user()函数的原型：int get_user(x,p)、int put_user(x,p)定义为两个宏，作用是用于基本数据拷贝。get_user()作用是把数据由用户空间复制到内核空间里，put_user()作用是把数据由内核空间复制到用户空间里。x 定为内核空间数据，而 p 设定成户空间指针。它们通过 access_ok()执行地址检查。若拷贝成功，则返回 0，否则返回 FAULT。

4.3.2 RUNTIME 阶段底层 API 性能测试设计实现

UEFI BIOS 把 CPU 的控制权给 OS 的同时将底层的初始化完成的函数对应的参数地址会传递进系统内核里，主要用 copy_to_user()和 copy_from_user()函数，将底层函数的地址传递入到用户空间，同时也可以将用户的请求传递到内核空间，在内核中提供一个特殊的系统调用途径，这个限制了用户请求，还保证了内核的其它模块不会受到干扰，这个就是论文工作设计 copy_to_user()和 copy_from_user()函数的主要原因，其实现的函数 efiRT.c 的核心部分如下。

```
transfor_number=copy_from_user(&param, (void*)arg, sizeof(param));

if(param.Time == NULL){
    p1 = NULL;
}else{
    p1 = &eft;
    transfor_number=copy_from_user(p1, param.Time, sizeof(efi_time_t));
}

if(param.Cap == NULL){
    p2 = NULL;
}else{
    p2 = &efc;
    transfor_number=copy_from_user(p2, param.Cap, sizeof(efi_time_cap_t));
}

spin_lock_irqsave(&efi_rt_lock, flags);
param.Status = efi.get_time(p1, p2);
spin_unlock_irqrestore(&efi_rt_lock, flags);

transfor_number=copy_from_user((void*)arg, &param, sizeof(param));
if(p1 != NULL){
    transfor_number=copy_from_user(param.Time, p1, sizeof(efi_time_t));
}
if(p1 != NULL){
    transfor_number=copy_from_user(param.Cap, p2, sizeof(efi_time_cap_t));
}
return 0;
```

论文接下来要做的工作就是把系统调用加入到 Linux 内核当中。efiRT.c 通过 GCC 环境编译完，加载到 Linux 内核当中，加载步骤与内核驱动加载过程是一样的。需要建立设备节点，运行命令 mknod /dev/efiRT.ko c 232 0，建完节点后再执行命令 insmod efiRT.ko，最后可在/dev 里看到 efiRT 设备，这样系统调用的模块就成功加入到了系统内核中。图中片段的函数功能主要是把用户获取 BIOS 的时间服务

的请求传入到系统内核里，同时把 BIOS 传进内核里的参数，传到用户空间里。

此过程用图形描述如图 4.15 所示。

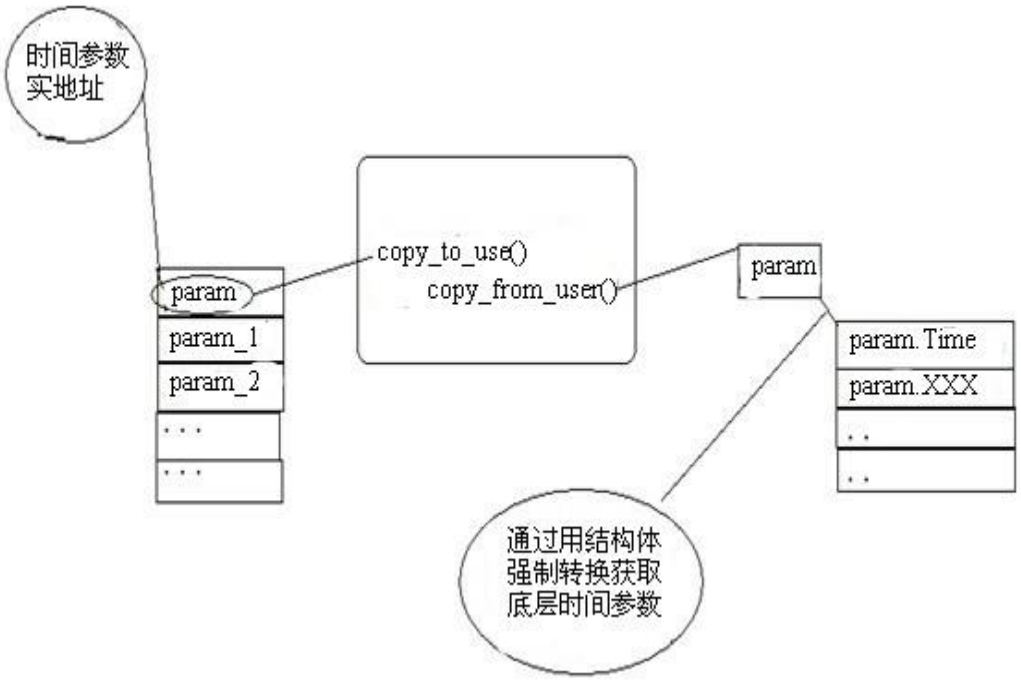

图 4.15 底层函数的地址传到上层的流程

系统调用把时间参数交于用户空间之后，用户通过地址的强制转换，得到该地址下其它参数的地址。这个前提是指定结构体原型，在未知晓结构体原型的情况下是不可获取其参数的地址的。一些结构体的原型，能重新取定义其结构体的变量，把获取的地址通过强制转换，得到一个对应的结构体，把其变量取出并赋于新变量，来实现实时的调用。

论文的工作就到了如何编写具体应用程序。实现 Linux 下系统调用功能后，编写应用程序就比较清晰，在已知系统内核有专门的函数接口供给用户调用的基础上只要编写相应的调用函数到内核里找想对应的函数。应用程序 GetTime 代码如下所示。

```
Status = gtRT->GetTime(
                &OldTime,
                NULL
                );
if (EFI_ERROR(Status)){
    StandardLib->RecordAssertion(
                        EFI_TEST_ASSERTION_FAILED,
                        gTestGenericFailureGuid,
                        "RT.GetTime - Get time",
                        "%a:%d:Status - %r",
```

```
                        __FILE__,
                        __LINE__,
                        Status
                        );
}
//
//Change year
//
Time = OldTime;
if(Time.Year != 2010){
    Time.Year = 2010;
}else{
    Time.Year = 2011;
}
Status = gtRT->SetTime(
              &Time,
              );
if(Status == EFI_SUCCESS){
    AssertionType = EFI_TEST_ASSERTION_PASSED;
}else{
    AssertionType = EFI_TEST_ASSERTION_FAILED;
}
StandardLib->RecordAssertion(
                        AssertionType,
                        gTimeServicesInterfaceTestAssertionGuid003,
                        "RT.SetTime - Change year",
                        "%a:%d:Status - %r,Year %d",
                        __FILE__,
                        __LINE__,
                        Status,
                        OldTime.Year
                        );
```

论文接下来的工作就是编译并运行应用程序实现 RUNTIME 阶段 API 性能测试分析。编写好程序之后，用 GCC 环境编译，该过程先在 Shell 下执行命令 gcc –c GetTimeFunc.c –o GetTimeFunc,‘-c’是制定 GetTime 源程序，‘-o’用来显示可执行的文件，可执行的文件名字即是 GetTimeFunc。编译结束后可在编译目录里找到 GetTimeFunc 的文件，通过将前面的系统调用模块加载到内核中。

在 Shell 下执行./GetTimeFunc，就能出现程序对应的输出结果，即如图 4.16 所示。

```
-------------------------------------------------------------------------------
GetTime_Func
-------------------------------------------------------------------------------
Logfile: "Log/GetTime_Func.log"
Test Started: 03/04/2009    14:12:15
-------------------------------------------------------------------------------
RT.GetTime - Valid parameters -- PASS
7D06EE71-E5F5-4AF3-BC74-ECF471F0F612
GetTimeFunc.c:80:Status - Success, Time 03/04/2009    14:12:15

RT.GetTime - Verify returned time -- PASS
93EF949C-241E-4365-B00B-0842F63F978C
GetTimeFunc.c:95:Status - Success

GetTime_Func: [PASSED]
      Passes..... 2
      Warnings..... 0
      Errors.....…0
-------------------------------------------------------------------------------
Logfile: "Log/GetTime_Func.log"
Test Finished: 05/09/2010    14:12:16
```

图 4.16 Runtime 阶段的分析结果

测试的结果很明显，两次调用 RT.GetTime 函数的结果都是成功的，这样就实现了 LINUX 下系统调用，也验证了 API 的性能测试。

4.4 本章小结

本章分别详细地给出了 API 性能测试在 NT32 平台及 LINUX 下的实现。首先设计完成了一款 PEI 阶段的底层 API 性能分析测试模块，成功地在 UEFI SHELL 下实现了对 MDE Library 的分析。接着通过对 RUNTIME 阶段底层 API 的研究，实现了在 LINUX 下底层 API 的实时调用，且编写应用程序代码通过系统调用的机制获取底层调用提供的准确时间，进而直观地测试了 RUNTIME 阶段底层 API 的性能。

第五章　总结与展望

论文基于最新的 EDKII/Tiano 体系架构，针对 Intel 的主流平台提出底层硬件检测以及 DXE 阶段的程序性能分析设计思路，并完成了编码实现。

主要工作具体有以下几个方面：

1. 研究学习 EDKII/Tiano 系统技术，清楚了 UEFI/Tiano 系统技术的背景和体系结构、主要功能、运行机制和仍存在的问题等。

2. 根据 SMBIOS 的工业规范及 EFI Shell 环境应用程序的开发规范，论文设计且完成了一款基于 PEI 的阶段底层 API 的性能进行分析测试的功能模块。在 NT32 平台上成功地在 UEFI SHELL 环境下实现对了 MDE Library 的分析。

3. 研究学习了程序结构的性能分析基础方法，对所有 EDK 的编译项完成了改进，使它们支持特殊编译。编译得到能在硬件平台上执行的.efi 程序。在实际操作过程中，通过应用 ITP 硬件的调试工具实时分析了 CPU，解决插入函数的不稳定所导致的程序崩溃等实际问题。

4. 通过对 LINUX 的 RUNTIME 阶段的底层 API 的学习，实现了 LINUX 环境下对底层 API 的调用。实现了 LINUX 下的用户对系统调用的需求，编写相应应用程序获取了底层系统调用所提供的时间服务。

论文的工作围绕着对 UEFI BIOS 底层 API 性能分析展开，实现了对 PEI 阶段底层 API 性能测试模块，以及 RUNTIME 阶段底层 API 性能测试模块。在此基础上可以进一步开展的研究工作有。

1. 仅对 PEI 阶段 API 性能分析测试是不够完善的。BIOS 启动过程主要分为了七个大的阶段，要确保模块正常运行在所有的阶段进行对应的分析测试。如 DXE 的阶段下，SHELL 工具 API 的状态等。论文提出了 PEI 阶段 API 测试的方法并实现了，对其余阶段的 API 性能测试实现还需要做大量的工作。

2. 对底层的 API 信息检测，在 PEI 的阶段实现的测试，可以应用到 DXE 的阶段。同样可通过 UEFI BIOS 所提供 RUNTIME 服务获取系统时间起始虚拟的地址实行。

3. 在 RUNTIME 的阶段对底层 API 调用的研究可以进一步扩展，可对 ACPI 接口做深入的研究。UEFI BIOS 程序的结构性能测试分析，是在 Linux 内核里实现的，在 Windows 环境下目前没有具体的方法，原因是 Windows 内核的接口目前是不透明的。

致谢

这一年多来，在 Intel 上海研发中心的生活让我受益匪浅，在此我要感谢亲人，同学，以及同事对我生活和工作上的帮助。

首先要感谢我的导师武波教授。武波导师是我的良师益友，他不仅知识渊博，工作认真，更是我生活，学习以及工作上的导师。他严谨的治学态度让我印象深刻。他是我生活中的榜样。在此要特别感谢一下张立勇老师对我耐心的指导与帮助，我深刻地体会到了张立勇老师在教学和指导论文写作中所付出的真诚与热情。他一丝不苟的作风深深地感动了我。

其次要感谢我的实习单位导师令杰工程师，还有在我实验和论文写作过程中给予我莫大帮助的顾小刚，高杰，王静工程师。在 Intel 的一年里他们教会了我许多，从起初的适应公司氛围，到胜任自己的工作，再到论文的编写过程中给我许多意见和建议，我深深地体会了他们每个人对同事，对他人的热切关怀。

最后我怀着沉痛的心情感谢陪伴我走过了 26 年的三伯，感激有他陪伴的过去的点点滴滴，在这一刻，送上我最深切的敬意。

参考文献

[1] 倪光南，UEFI BIOS 是软件的蓝海，UEFI 技术联盟会议，2006 年 07 月 13 日

[2] 余志超，朱泽民.新一代 BIOS--EFI、CSS BIOS 技术研究，2006(5)

[3] Framework Open Source Community, Framework Open Source Community, June 29, 2005

[4] 胡藉.面向下一代 PC 体系结构的主板 BIOS 研究与实现，南京航空航天大学，2005

[5] 洪蕾.UEFI 的颠覆之旅，中国计算机报，2007 年 07 月 09 日

[6]Framework Open Source Community, Framework Open Source Community , June 27, 2005

[7] 李振华.于 USBKEY 的 EFI 可信引导的设计与实现，北京交通大学，2008

[8]Framework Open Source Community, Pre_EFI Initialization Core Interface, March 2008

[9]Framework Open Source Community, Shared Architectural Elements, Framework Open Source Community, March 2008

[10]UEFI Forum. May 2009. UEFI Specification Version2.3. http://www.uefi.org/specs/

[11] Vincent Zimmer. 2006.Beyond BIOS Intel corporation. 17-32,143-146

[12] 潘登等，刘光明 EFI 结构分析及 Driver 开发， 2006(02) .

[13] 吴松青等，王典洪.UEFI 的 Application 的分析与开发，计算机应用与软件，2007(2)

[14]石浚菁.EFI 接口 BIOS 驱动体系的设计、实现与应用，南京航空航天大学，2006

[15]姚颉文等，谢康林，.基于 EFI 驱动-协议模型的自我认证测试系统，计算机仿真，2005(07)

[16] 杨荣伟.基于 Intel 多核平台的 EFI/Tiano 图形界面系统研究，上海交通大学，2007

[17]张朝华.基于 EFI/Tiano 的协处理器模型的设计与实现，上海交通大学，2007

[18] Intel corporation. August 2008. EDK II Extended INF File Specification Revision Version 1.1

[19] Intel corporation. May 20,2007. EDK Flash Description File (FDF) Specification

[20] Intel corporation. Oct. 12, 2006. EDK II Module Development Environment Library Test Infrastructure

[21] Intel corporation. October 4,2006. Intel Platform Innovation Framework for EFI Human Interface Infrastructure Specification

[22] Intel corporation. October 9,2006. MDE Library Spec