申请上海交通大学工学硕士专业学位论文

动态二进制翻译中基于 profile 的优化算法研究

学　　校： 上海交通大学

院　　系： 电子信息与电气工程学院

班　　级： B0503392

学　　号： 1050339035

工学硕士生： 史辉辉

专　　业： 计算机软件与理论

导　　师： 管海兵

上海交通大学电子信息与电气工程学院

2008 年 1 月

A Dissertation Submitted to Shanghai Jiao Tong University for

Master Degree

Study of Profile-based Optimization Algorithms in Dynamic Binary Translation

Author： Shi Huihui

Specialty: Computer Software and Theory

Advisor: Guan Haibing

Shanghai Jiao Tong University

Shanghai, P.R.China

Jan, 2008

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

动态二进制翻译中基于 profile 的优化算法研究

摘要

二进制翻译（Binary Translation）是指在不需要可执行程序源代码的情况下，把源机器平台上的二进制程序经过一定的转换之后运行在目标机器平台上的过程。所谓动态二进制翻译就是边翻译边执行，并在翻译的过程中进行动态优化。动态二进制翻译为解决代码遗留，代码移植以及构建分布式虚拟计算环境等问题提供了一个良好的解决方法，因此在近年来得到了越来越广泛的关注和研究。由于动态二进制翻译和优化中的所有工作都是在运行时完成，对系统性能有较高的要求，因此应该采取跟静态编译和优化不同的途径。

本文主要研究在动态二进制翻译中如何有效地收集运行时的 profile 信息，以及如何用这些信息来进行有针对性的优化，研究内容主要包括三方面：程序运行时的 profile，热代码的识别和优化，以及基于 profile 信息的代码 cache 管理。

Profile 是指对程序运行时信息的统计和收集。传统的动态二进制翻译器都是采用基于基本块的 profile，其缺点是工作量大，信息缺乏连贯性。本文提出了一种高效的基于路径的 profile 方法，实现上稍为复杂，但可以在一定程度上克服传统方法中的缺点。

热代码的识别和优化分两步，首先根据收集的 profile 信息找出频繁执行的一段连续的基本块序列，然后把这些基本块合并成具有单一入口和多个出口的超级块，这样做一方面可以减少因基本块结束而

引发的上下文切换，另一方面，超级块中包含更多的指令，给中间代码的优化提供了广阔的空间。

代码 cache 的调度和管理算法主要是指已翻译代码的替换算法。由于基本块不定长，不合理的替换将在 cache 中产生很多碎片，降低 cache 的利用率。而采用 FIFO 等不会产生碎片的算法又往往降低替换的准确性，即把当前的热块替换出去，造成频繁翻译。本文提出了多层次 cache，即将代码按照他们的 profile 信息存放到对应的 cache 层次中，最大限度地减少碎片率和错误替换次数。

关键字：动态二进制翻译，动态优化，热路径，代码 cache

Study of Profile-based Optimization Algorithms in Dynamic Binary Translation

ABSTRACT

Binary translation is the process of converting binary code from one ISA to another ISA without the presence of source code. In dynamic binary translation (DBT), translation is done at run-time, and optimization algorithms are applied on the translated code at the same time. DBT is especially useful in solving the problem of legacy code, code migration and building a virtual computing environment. It has become a hot spot in recent research. Because all the translation and optimization are done at run-time, it has a high demand on the efficiency of the system.

The work of this thesis includes three major parts: run-time profiling, hot code recognition and optimization, and the management of code cache according to the profile information.

Profile involves collecting run-time information of the system. Traditionally, the object to be profiled in a binary translation system is basic block, but it requires a lot of work, and the information it collected is relatively isolated. In this thesis, we proposed an efficient path-based profiling method.

Hot code recognition is the process of finding the highly frequently executed blocks by using the profiling information. Then the hot blocks are merged into superblocks, which contain single entry but multiple exits.

This will save the system a lot of cost on context switch, and also provide great opportunity for intermediate level optimization.

Code cache management focuses on the eviction of obsolete blocks. Because blocks may be different in length, fragmentation will be caused by inappropriate eviction. This thesis proposed a multi-level cache to solve the problem by using profiling information.

Keywords: Binary Translation, Dynamic Optimization, Hot path, Code Cache

目录

第一章　绪论

软硬件间的接口使得计算机软件和硬件依赖于特定的指令集体系结构，针对不同指令集体系结构的软件和硬件不能相互组合，而新的处理器要想得到广泛的应用，必须有大量应用程序的支撑。因此，软件的移植是处理器更新换代所面临的重要问题。一方面，新的处理器可能因为缺少有关应用软件的支持而无法广泛应用；另一方面，没有广泛应用的处理器也很难得到其他软件开发商的支持，从而进一步影响新处理器的推广。在这种情况下，处理器的发展和创新都必须兼容以前的版本，尽管它们有一些不容置疑的缺陷。因此，研究不同平台之间的软件移植，不仅对软件重用有重大意义，更可以解脱处理器发展的束缚，促进处理器的不断创新。

为了解决遗留代码和代码移植的问题，一般来说有以下几种解决方法[1]：第一种是提供专门的运行模式执行老代码，如 Intel 的 Itanium 处理器存在专门执行 x86 代码的硬件；第二种是采用重新编译的方法，即把应用程序的源代码重新编译使得它能在新的平台上执行；第三种是采用软件方法解释或翻译应用程序。采用第一种方法显然无法利用新处理器的一些先进特性，同时还增加的新处理器的硬件复杂度，甚至还会影响原有代码的执行效率。第二种方法具有很高的效率，前提是必须获得应用程序的源代码。有些程序依赖于共享代码库，而这些共享代码往往以目标代码的形式出现，很难获得源代码，因此，这种方法在很多情况下都缺乏可行性。鉴于此，第三种方法，二进制翻译的方法便应运而生了。它是一种可直接翻译执行二进制代码的技术，能够把一种 ISA 上的二进制代码直接翻译到另一个 ISA 上执行。而且二进制翻译采用软件的方法实现，具有很大的扩展性，可以同时支持多个 ISA 的翻译，解脱了硬件发展的束缚，为处理器的发展和创新提供了广阔的空间。

二进制翻译器是位于应用程序和计算机硬件之间的一个软件层，它很好地降低了二进制应用程序和底层硬件之间的耦合，使得二者可以相对独立地发展和变化，因此，具有广阔的应用前景。

1.1　二进制翻译技术简介

二进制翻译也是一种编译技术，它与传统的编译技术的区别在于它们所编译处理的对象不同。传统编译技术编译的是某一种高级语言，经过编译处理之后生成某种机器的目标代码；二进制编译处理的对象是某种机器的二进制代码，该二进制代码是经过传统编译器生成的，它经过二进制翻译处理后直接生成另一种机

器的二进制代码。按照传统编译程序前端、中端和后端的划分，我们可以认为二进制翻译是拥有特殊前端的编译器。

1.1.1 二进制翻译结构与分类

二进制翻译系统在概念上可以分为三个部分[3]：前端解码器、中端分析优化器和后端代码生成器，它们在二进制翻译的不同阶段起着不同的作用，共同完成从源到目标的翻译。

前端解码器根据源机器的指令结构特点，以及可执行文件的格式规定，通过指令模式匹配对二进制代码进行处理，完成类似反汇编的功能。解码器的输出是某种形式的抽象中间表示，以便对其进行分析和优化。

中端分析优化器的功能是对中间表示进行一定的转换，除去代码中源机器特性，并对程序进行分析和部分优化。在二进制翻译系统中，引入中间表示可以降低前端和后端的耦合度，可以用来实现多源多目标的翻译器。

后端优化编码器类似于一般编译器，其功能是从一种中间语言生成优化的目标代码。它根据目标机器的特点，将中间代码翻译为目标机器上可执行的二进制代码，综合了常规编译系统中的后端代码优化和生成器，以及类似链接器和装载程序的功能。

按照实现方法的不同，可以把二进制翻译分为三类：解释执行，静态翻译和动态翻译[1]。

解释执行对源处理器代码中的每条指令实时解释执行，系统不保存也不缓存解释过的指令，不需要用户干涉，也不进行任何优化。解释器相对容易开发，比较与老的体系结构高度兼容，但效率很差[1][3]。

表 1 三种翻译方法比较总结

	优点	缺点
解释执行	开发容易，不需要用户干涉，高度兼容	效率很差
静态翻译	离线翻译，可以进行更好的优化，效率较高	依赖解释器、运行环境的支持，需要终端用户的参与，给用户造成了不便
动态翻译	无需解释器和运行环境支持，无需用户参与，利用动态信息来发掘优化机会	翻译的代码效率不如静态翻译高，对目标机器有额外的空间开销

静态翻译是在源处理器代码执行之前对其进行翻译。静态翻译器离线翻译程序，有足够的时间进行更完整和细致的优化，生成代码的质量较高，优化效果较

大。但是它不能获得运行时的信息，可能无法完整地翻译一个程序[4]，因而需要依赖解释器的支持[3]；而且静态翻译器作为一个独立的工具，需要中端用户的参与[1]，这给用户带来了很大的不便。

动态翻译在程序运行时对执行到的代码块进行翻译，克服了静态翻译所无法解决的一些困难，如运行时动态信息收集，代码挖掘，自修改代码和精确中断问题[3]。而且动态翻译器对用户完全透明，无需用户干预[5]。虽然动态翻译有上述诸多优点，翻译过程却由于受到动态执行的限制而不能像静态翻译那样进行完全细致的优化，使得翻译生成的代码效率比静态翻译器差[1]。表 1 总结了上述三种翻译方法的优缺点[1][3]：

1.1.2 动态二进制翻译系统框架简介

动态二进制翻译是一种兼顾正确性和效率的二进制翻译方式，其基本结构和流程也相对固定。图 1 给出了常见的二进制翻译器的大致结构和翻译流程[6]。

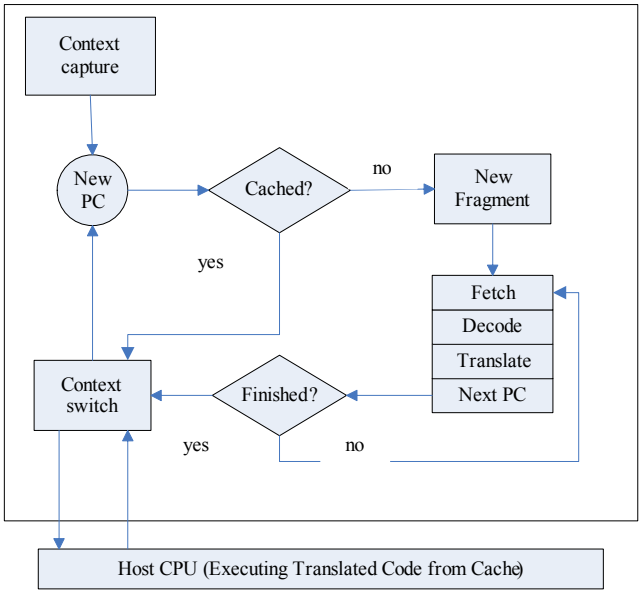

图 1 二进制翻译的流程

Figure 1 Work flow of Binary Translation

一个动态二进制翻译系统通常包含控制块、启动块、翻译模块、代码 cache 管理和本地执行模块等。整个翻译过程如下：当系统要执行一个新的基本块时，控制模块首先检查该基本块是否已经被翻译并存放于 cache 当中。已经存在了，则直接执行已经存在的基本块，否则，控制器将启动翻译模块，生成一个空的代码块，然后将指令逐条插入，直到遇到基本块结束条件，然后将新生成的基本块进行翻译，并将其存放在代码 cache 之中。

1.2 动态优化技术简介

动态优化技术既可以作为一种独立的代码优化技术，又可以作为二进制翻译系统的一部分。而由于动态二进制翻译的动态特性，使得动态优化技术与动态二进制翻译技术具有密切的联系。

动态优化技术是在应用程序的运行时刻对程序的信息进行收集和分析，并对程序的关键部分有选择性地进行必要的优化，从而提高程序的性能。由于在动态时刻对程序进行上述处理需要花费一定的时间，因此动态优化必须及时发现程序的关键部分，并采用快速而又高效的分析、优化方式，以便得到理想的优化效果。

动态二进制翻译要兼顾正确性和高效性。在实现不同平台二进制代码的移植时，首先要保证翻译的正确性，使得翻译后的代码与源代码等价；同时，由于翻译而带来的运行时开销需要通过优化生成高质量的目标代码来进行弥补，因而动态优化成为了动态二进制翻译中的一个必要的环节。

动态优化器的吸引人之处在于它能够根据动态的运行时信息对正在执行的程序进行调整，使之适应具体的执行环境，而这些信息用静态的方法是无法获取的。动态优化器的目标就是要能够优化正在执行的二进制代码。对于动态二进制翻译器的构造，有如下几个设计要求[7]：

（1）必须能够观察和修改正在执行的指令，这是最基本的要求。观察指程序运行过程中各种 profile 信息的收集；

（2）必须能够中断程序的执行。为了能够初始化和进行代码优化，必须做到可以在程序中的任何地方中断程序的执行；

（3）必须在优化算法的复杂度和效率之间作权衡，最终目标是提高系统的整体效率；

（4）健壮性：这对动态优化器来说尤为重要，在设计时必须使其尽可能健壮，从而最大限度的提高程序的健壮性；

（5）透明性：最理想的状态是完全透明，有些情况下，优化器可能会降低透明性，获取一些有用的信息，如符号表，栈信息，以及一些函数调用信息，在动态二进制翻译中，优化器是二进制翻译器的一部分，它需要的信息完全来自于二进制翻译器；

（6）控制权：在程序执行过程中，优化器必须拥有对程序的决定控制权，绝对不能丢失。

常见的动态优化技术有热路径优化，寄存器优化，高效的代码 cache 管理，以及硬件辅助优化等。随着研究的不断深入，越来越多的动态优化技术正在不断地涌现。

1.3 常见二进制翻译及优化系统介绍

二进制翻译和优化系统可以分为固定源和目标的二进制翻译系统，多源和多目标的二进制翻译系统，以及专门的动态优化系统。

常见的固定源和目标的二进制翻译系统有以下几种：

DEC 公司 1996 年研发的 FX!32 系统[8]是一个基于 profile 信息的二进制翻译器，目的是为了能将运行在 x86/WinNT 系统下的应用程序运行在 Alpha/WinNT 下。它结合静态翻译和动态解释，具有正确，高效而透明的特点。

HP 公司 1999 年开发的 Aries 软件仿真器[5]是一个基于软件的 IA64 转换设备。该系统结合快速解释和动态翻译两种手段，可以仿真 PA-RISC 全部指令集，无需用户干涉。Aries 只动态翻译经常使用的代码，仿真过程结束后丢弃所有翻译的代码，而不修改原来的应用程序。因此，动态翻译既可提供快速仿真，又不破坏仿真的 PA-RISC 应用程序的完整性。

Daisy 和 BOA 系统是 IBM 公司分别于 1996 年和 1999 年开发的，虽然都用到二进制翻译优化技术，解决了诸如精确中断和自修改代码等二进制翻译通常会遇到的难题，但这两个系统的研究开发目标并不相同。Daisy 是用于仿真现存体系结构的二进制翻译系统[9][10]，以使旧体系结构上现存的软件（包括操作系统内核）可以在超长指令字（VLIW）体系结构下运行。BOA 动态二进制翻译系统[11][12]的目标是简化硬件，通过结合二进制翻译和动态优化，填补 PowePC RISC 指令集和更简单的硬件原语之间的语义差别。BOA 系统关心的不是每条指令的周期数目（CPI）最小化，而是希望通过简化了的硬件指令，极大地提高处理器频率。

Transmeta 公司 2000 年开发的 Code Morphing 软件[13]，使自己研发的芯片能够兼容 X86 的软件，利用二进制翻译技术开创了一个新的软、硬件开发模式。Transmeta 公司生产的 crusoe 芯片是由逻辑上被 Code Morphing 软件裹着的硬件引擎构成的。Crusoe 芯片与 Code Morphing 软件的结合说明微处理器可以被当作软硬件的混合体来实现，软件部分的升级独立于芯片，硬件设计系统和应用软件分开使得硬件设计师更新设计时不用担心影响遗留的软件。

Intel 公司 2003 年开发的 IA-32 Execution Layer（EL）软件[14]，通过软件的方法在 IPF（Itanium Process Family）上执行 IA32 的应用程序，实现兼容，从而简化硬件的复杂度。IA-32 EL 是一个应用程序级的翻译器，它运行在本地的 64 位 OS 之上，能够支持 windows 和 Linux 系统。EL 把 IA32 上的应用程序加载到翻译器自己使用的地址空间上，翻译器为了能够在多平台上运行，把操作系统无关的翻译算法放在一个操作系统无关的模块〔BTGeneric)，而把库函数调用这种操作系统相关的封装到 BTLib 模块中，这两个模块之间通过约定好的 API 进行

通信。IA-32 EL 是两个阶段动态翻译的翻译器。运行时把翻译的代码缓存起来，一旦这个进程结束，就把已经翻译的代码扔弃。第一阶段的翻译，是冷代码的翻译，在这个阶段，要求翻译开销小，做了最少的优化，而且需要为第二阶段的翻译提供热点第二阶段的翻译，也就是热代码的翻译。冷代码的翻译，基本上以基本块为单位，一个基本块平均包含 4-5 条 IA32 指令.对于热代码的翻译，以热路径为单位，一般包含 20 条左右的工 A32 指令。虽然 Itanium 2 上也可以硬件执行 IA32 的程序，但现在 IA32 EL 的执行效率已经超过了硬件提供的执行效率。

上述 FX!32，Aries 翻译系统和 IA32 EL 软件都是在操作系统环境之上运行，着眼于应用程序的翻译;而 Daisy, BOA，Transmeta 系统则是对整个系统的翻译，包括应用程序、操作系统、以及其它特权级的指令[2]。

常见的多源和多目标二进制翻译器有以下几种：

Queensland 大学先后研发了支持多源和多目标的静态（UQBT[15]）和动态（UQDBT[21]）二进制翻译系统框架，引导了二进制翻译研究的一个新的研究方向。UQBT 框架根据不同的二进制文件格式描述文件，自动生成文件编解码器；根据不同的编解码器描述文件自动生成指令编解码器；还要根据不同的语义描述文件自动生成语义抽象转换器，从而使机器不相关部分的工作分离出来，给二进制翻译器编写者提供可重用的代码。

奥地利维也纳大学研究开发了机器可适应的 Bintran 动态二进制翻译系统[17].它的最终目标是希望在不同机器描述协助下，能够实现所有的 CISC、RISC 以及 VLIW 体系结构之间的代码翻译。

Transitive 公司拥有的 Dynamite 软件技术[18]，是英国曼彻斯特大学的超过 20 人年的研究成果。该技术可以使本地代码和非本地代码的程序都能无缝透明的执行。

专门的动态优化系统不像二进制翻译系统那样名目繁多。HP 公司开发的 Dynamo[29][19]是一个动态优化器的原型。它的输入是本地二进制可执行代码，通过解释执行并观察程序的行为而不需要任何采样代码，不需要对代码进行预分析，也不需要为以后的执行写出信息。它解解释执行程序时收集 profile 信息，帮助动态选择执行频繁的热路径，然后对这些热路径进行优化，并将优化后生成的代码存放在一个软件 cache 中，当再次执行到这些路径的时候就不解释而直接执行 cache 中优化后的代码，从而使程序的执行效率得到大幅度提升。

此外，Java 虚拟机中也用到了动态优化技术。Java 编译器将 java 源代码编译成平台无关的 Java bytecode 格式，然后被分发到各种平台，由 Java 虚拟机(JVM)实时解释执行。在高性能实现的 JVM 中，Just-In-Time (JIT)编译器实时地将 Java 字节码翻译成本地码，来减少解释执行的开销。由于翻译本身在程序执行期间进行，编译时间成为程序执行时间的一部分，因而编译中的优化需要考虑动态优化

的特点，即需要在优化的代码质量和编译时间之间进行权衡。许多 Java 虚拟机中都采用了 JIT 编译，如 Sun 公司商业虚拟机 JDK[50]、汉城国立大学和 IBM 合作开发的开放源码 LatteVM [30], Intel 的开放源码 MRL VM [31]。

1.4 二进制翻译技术面临的挑战

二进制翻译技术可以应用在不同领域，满足不同的需求。在设计二进制翻译系统的时候可以从很多方面来权衡和选择。二进制翻译可以是解释执行或者翻译/优化；静态翻译或者动态翻译；模拟一个虚拟机器或者模拟真实机器；完整系统或者用户级；操作系统相关或者操作系统独立；同一指令集或者不同指令集等等。不过不管设计者如何选择，二进制翻译技术都要面临以下几个方面的挑战[2]：

1. 自修改代码：源机器的代码被修改时，与该代码段对应的任何翻译都要被置为无效。

2. 异常的精确性：同步异常(如页故障)和异步异常(如时钟中断)发生时，翻译器的异常处理机构必须提供一个与原结构状态一致的、正确的状态。

3. 地址翻译：不同硬件结构的地址空间设计存在一定的差异，对 I/O 地址的处理也大相径庭，全系统的二进制翻译还必须完成虚拟地址和物理地址之间的转化。

4. 自引用代码：自引用代码会进行自校验或者查看自己的代码，所以二进制翻译系统必须保存源机器代码的备份。

5. 代码 Cache 的管理：代码 Cache 的大小是有限的，当代码 Cache 满时，需要为新翻译生成的代码腾空间；被置为无效的翻译，也要在代码 Cache 中标识。

6. 实时行为：二进制翻译系统必须根据代码是否已经被翻译、被如何翻译来决定执行速度的不同，执行时间是一个不定因素。

7. BOOT 和 BIOS 代码：在全系统的二进制翻译中，控制源机器的最低级代码必须被如实地翻译到宿主机上。

由于研究人员的不懈努力，二进制翻译技术已经日趋成熟，这些二进制翻译面临的挑战也已经能得到比较妥善的解决，翻译的开销能够被控制在可以忍受的范围之内。

1.5 本文要研究的问题

1.5.1 研究目标

在动态二进制翻译和优化中，协调质量与效率的关系是要解决的重要问题。实际上，动态翻译和优化可以采取与静态编译时孑然不同的路线，静态编译优化侧重语义分析，而动态翻译优化可以更依赖统计信息，静态编译优化强调全面，追求算法的最优性，而动态翻译优化则可以针对重点，采用简单而效率高的算法来达到较好的效果。

本文将基于上述思想展开研究。本文的研究目标是在分析总结现有优化算法的基础上，针对其中的缺点和不足，重点研究其中的技术难点，找出新的优化方法， 并在实验平台上对新旧方法进行对比分析，提高动态二进制翻译器的整体效率。具体来讲，主要包含以下工作：

- 程序运行时如何对代码做 profile；
- 热代码的识别和优化；
- 基于 profile 的 Cache 的调度和管理。

1.5.2 全文结构

本文的剩余部分将按如下的方式来进行组织：

第二章系统介绍了动态二进制翻译技术，并以 valgrind 为实例来详细介绍其组成，工作原理以及优化方法。

第三章内容包括两方面。第一方面是动态二进制翻译中的 profile 以及实现方法，这是所有以 profile 为基础的优化方法的前提；第二方面介绍几种常见的热路径识别方法，然后提出了一种改进的热路径识别算法，并给出了实验结果和对结果的分析。

第四章讨论了代码 cache 的管理方法。在总结常见的代码 cache 管理策略的基础上，针对热路径的特点，提出了一种基于 profile 信息的代码 cache 管理方法，将热路径和普通代码基本块区别对待，以达到最佳效果。

第五章回顾总结了全文，并提出了未来工作。

第二章　动态二进制翻译系统

2.1 引言

将一些指令序列从一台机器翻译到另一台机器可能并不困难，但要真正实现一个二进制翻译器却并不容易。在静态二进制翻译中，代码在运行之前被离线翻译，使用目标机器的指令结构生成一个新的程序。不过，静态二进制翻译有其局限性。现在使用的计算机，其基本工作原理是存储程序和程控，它是由著名数学家冯•诺依曼提出的，其指令和资料一起存储，以相同的方式表示。因此，要静态地发现一个程序中所有的指令代码一般是不可能的。例如，控制语句中的间接跳转指令（如寄存器上的跳转）的目标指令有时候是很难静态地分析出来的。因此，一个静态翻译的程序通常会使用一种回跳机制，其表现形式一般是一个解释器。解释器在运行期处理所有未被翻译的代码，并且一旦发现一条适合的路径就返回到翻译过的代码。

动态二进制翻译克服了静态二进制翻译的局限性，但损失了性能。在动态二进制翻译器中，代码在运行期的空闲阶段被翻译，这样用户就可以在目标机器上获得接近于源机器上的运行性能。动态二进制翻译与仿真器不同，它生成本地代码并按需（on-demand）实现代码优化。代码中的热点在运行期被优化，从而提高了这些代码的运行性能。而且，一些不能静态实现的优化方法也可能动态的实现。

本章将以 valgrind 为例来详细介绍动态二进制翻译以及动态优化系统的原理与实现。

2.2 Valgrind 简介

Valgrind[32][33]是一个 DBI（Dynamic Binary Instrumentation）Framework，首先它是一个开源的、基于 GPL 的项目，其任务和目标是任何人都可以通过 Valgrind 提供的接口开发出自己的 DBA（Dynamic Binary Analysis）工具，用于调试和剖析（debugging and profiling）以及修改基于多平台（3.0 以后的版本）的 linux 二进制代码。像 linux 一样，Valgrind 的开发者来自世界各地，任何人都可以加入这个团队，对其作出贡献，比如为 Valgrind 开发出新的工具，或者将 Valgrind 移植到其它平台上。目前 valgrind 的最新版本是 3.3.0。其中第二版系列和第三版系列最大的区别是：第二版只能基于 X86 平台，而第三版有了一个突飞猛进的进展，可以基于多源多目标，目前支持 X86、ARM、PowerPC、AMD64

四种平台，而且还在进一步的扩展之中。

Valgrind 是一种执行驱动（execution-driven）的 JIT（Just in time）系统，几乎所有的常规代码都可以处理。这些常规代码包括正常的可执行代码，动态链接库，动态生成的代码等。唯一不在系统控制之下的是系统调用，但是它可以通过间接的方法来观察和监控。图 2 中左边部分给出了正常程序的执行层次。用户程序可以直接访问机器中用户层的部分（如通用寄存器），但是要访问系统层部分时，只能通过操作系统来进行访问。图 2 中右边部分给出了当用户程序运行在 valgrind 控制之下时的情景。用户程序和 valgrind 属于同一进程的不同部分，而且 valgrind 完全把用户程序置于它的监控之下，会干涉到用户程序所做的所有事情。

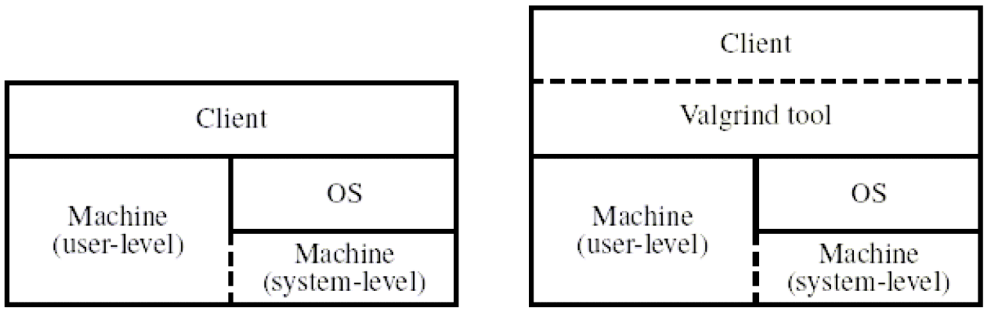

图 2 valgrind 在程序执行中的位置

Figure 2 valgrind is a layer below client

Valgrind 采用中间代码的方法对源机器的二进制代码进行动态翻译，即以基本块为单位作为输入产生中间代码，通过对中间代码进行修改以达到控制程序的目的，最后修改的中间代码再被翻译成目标机器代码并运行。其中间代码的设计采用了类似 RISC 的两地址指令集的方案，该方案有以下优点：

1. 可以很容易的实现多源多目标；
2. 可以根据需要修改中间代码，以达到监控程序的目的；
3. 可以针对中间代码做优化；
4. 可以充分利用目标机器的寄存器。

2.3 Valgrind 的翻译单元

Valgrind 翻译和执行的基本单位是基木块，即一段以控制转移指令为结束的顺序代码(与编译中定义的基本块稍有区别，为了便于动态实现)，并以每个基本块的源二进制代码的首地址作为其唯一的标识。通常一个基本块包括了 4 到 7 条指令。不同于单条指令为单位的翻译，它可以省去很多和函数调用相关的操作，比如说堆栈的维护，包括参数和局部变量以及返回地址的处理。同时可以充分挖

掘基本块内的指令并行性，给编译器提供了更多的优化空间。此外还可以充分利用宿主机的优势，用尽可能少的宿主机指令来实现源机器程序中多条指令的语义。

Valgrind 中所有中间代码基本块的最后一条或两条指令都是跳转指令。如果源二进制代码中的跳转指令是无条件跳转，那么对应的中间代码基本块的最后一条指令必定是无条件跳转。如果源二进制代码中的跳转指令是条件跳转，那么对应的中间代码基本块的最后两条指令必定是一条条件跳转和一条无条件跳转。Valgrind 中的无条件跳转和条件跳转指令如下所示：

jmp_lit(UCodeBlock* cb, Addr d32)

jcc_lit(UCodeBlock* cb, Addr d32, Condcode cond)

其中 UCodeBlock 是一个存放基本块信息的结构体，Addr 是一个四字节的地址，Condcode 是条件码，即满足该条件时才发生跳转。

2.4 valgrind 的模块及工作流程

Valgrind 的系统框架由六个模块组成，分别是装载器，调度器，前端解码器，后端编译器，工具加载模块以及代码 cache。其中代码 cache 又可分为普通代码 cache 和快速代码 cache。其框架如图 3 所示：

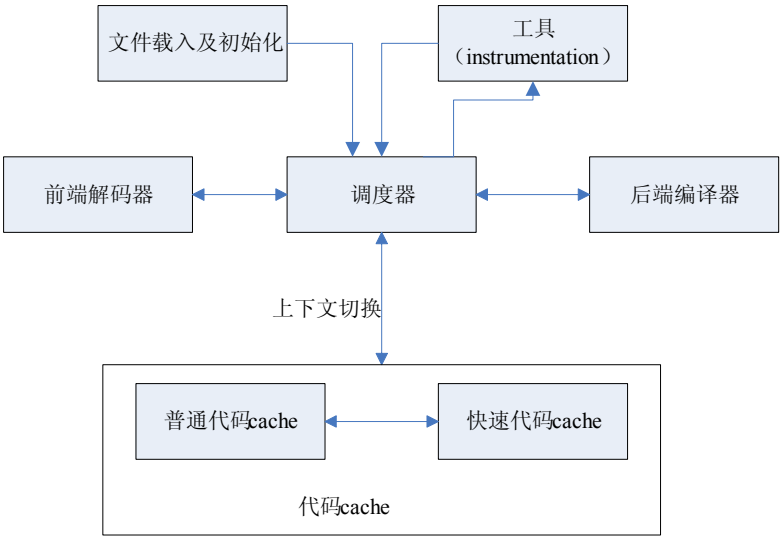

图 3 valgrind 的构架

Figure 3 The architecture of valgrind

文件装载和初始化模块是系统启动后首先调用的模块，它负责二进制文件的载入和客户端环境的设定，如建立 client 栈，建立 client 数据空间等；前端解码器负责从源二进制指令解码生成中间代码（IR）基本块；后端编译器负责从中间代码基本块到本地二进制代码的翻译，同时它还包括了一些初级的优化；而工具则是一些可加载的选项，可以根据不同的需要编写不同的工具，由调度器来进行

加载，例如可以编写一个 profiling 的工具来对程序做 profile；调度器是整个系统的核心，它负责各个模块之间的切换，以及翻译和执行时的上下文切换。代码 cache 是一块用来存放翻译后的本地代码的内存，它包括两部分：第一部分是普通的代码 cache，以基本块首地址为标识，将目标代码基本块依次存入；第二部分是快速代码 cache（fast cache），它存放了当前执行的路径，即当前执行的几个基本块的序列，每次调度器都先在快速代码 cache 中查找，如果没有则在普通代码 cache 中查找。由于程序的执行具有局部性原理，这样做可以在一定程度上减少查找的次数。

Valgrind 的工作流程如图 4 所示。首先将要翻译和执行的源可执行文件及其依赖的库文件装入进程空间，做必要的初始化工作，然后携带源可执行程序入口 pc（程序计数器）进入主循环，直到满足结束条件时退出。主循环的迭代过程如下：

第一步：首先在代码 cache 中查找以当前 pc 值为标识的基本块是否已经存在了，如果没有则调用解码器将源二进制代码翻译生成一个中间代码基本块。

第二步：对该中间代码基本块加载一些必要的工具（instrumentation），然后做一些比较初步的优化工作，如冗余代码的删除，寄存器重新分配等。

第三步：由后端翻译器将该中间代码基本块翻译成本地代码，并将翻译后的代码存放在代码 cache 当中，并将控制权交还给调度器。

第四步：调度器将上下文切换到本地执行状态，跳入翻译好的本地代码执行。本地代码执行完之后，调度器获得下一个 pc，进入新的一轮迭代。

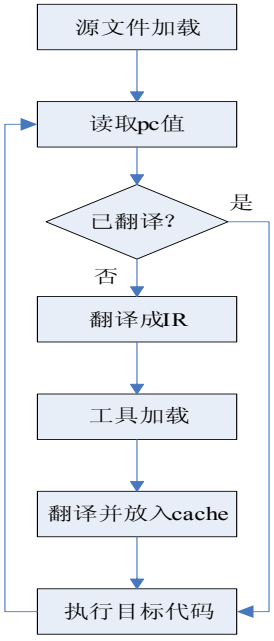

图 4 valgrind 工作流程图

Figure 4 The work flow of valgrind

2.5 Valgrind 基本块的翻译过程

Valgrind 对基本块的翻译是按需（on-demand）进行的[34]。在生成一个代码基本块时，valgrind 逐条读取指令，直到遇到下列条件中的一个：

1. 达到基本块中可以容纳的上限，一般 50 条左右，视不同的系统构架而定；
2. 遇到一个条件跳转；
3. 遇到一个目标地址未知的跳转分支；
4. 有明确跳转地址的无条件跳转指令达到三条。

Valgrind 中基本块的翻译过程包括以下几个步骤：

第一步：解码，从机器码解码成树状中间代码（tree IR）。在该步中，前端解码器将机器码解码成未经优化的树状中间代码。每条指令都被独立地解码成一条或多条中间代码指令。源指令中的寄存器被存入临时寄存器中，经操作后再写回。

```
0x24F275: movl -16180(%ebx,%eax,4),%eax
1: ------ IMark(0x24F275, 7) ------
2: t0 = Add32(Add32(GET:I32(12), # get %ebx and
   Shl32(GET:I32(0),0x2:I8)),    # %eax, and
   0xFFFFC0CC:I32)               # compute addr
3: PUT(0) = LDle:I32(t0)         # put %eax

0x24F27C: addl %ebx,%eax
4: ------ IMark(0x24F27C, 2) ------
5: PUT(60) = 0x24F27C:I32        # put %eip
6: t3 = GET:I32(0)               # get %eax
7: t2 = GET:I32(12)              # get %ebx
8: t1 = Add32(t3,t2)             # addl
9: PUT(32) = 0x3:I32             # put eflags val1
10: PUT(36) = t3                 # put eflags val2
11: PUT(40) = t2                 # put eflags val3
12: PUT(44) = 0x0:I32            # put eflags val4
13: PUT(0) = t1 # put %eax

0x24F27E: jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32       # put %eip
16: t4 = GET:I32(0)              # get %eax
17: goto {Boring} t4
```

图 5 解码：从机器码到树状 IR

Figure 5 Decoding: from machine code to tree IR

图 5 给出了一个 x86 机器码解码的例子。在该例子中，3 条 x86 指令被分解成了 17 条树状中间代码指令。

指令 1，4 和 14 是 IMarks，即标识一条指令开始的空操作，它同时还记录了该指令的地址和所占的字节数。这些空操作可以帮助 profiling 工具来识别指令的边界。指令 2 将一个树状表达式赋值给一个临时变量 t0；它说明了怎样将一条 CICS 指令转化成多条中间代码指令。GET:32 读取一个 32 位的整型寄存器；偏移量 12 和 0 分别表示源指令中的寄存器%ebx 和%eax。Add32 是一个 32 位的加法操作，而 Shl32 是一个 32 位的左移操作。指令 3 将寄存器%eax 的值写回。指令 5 和 15 更新程序计数器（%eip）。指令 17 是一条跳往 t4 的无条件跳转指令。

第二步：初步优化，将树状中间代码转化成简单中间代码（flat IR）。初步优化除了将树状中间代码简化之外，还做了一些其他的优化，包括冗余的 get 和 put 的删除，拷贝和常量传播（copy and constant propagation），常量结合（constant folding），死代码的移除，公共子表达式的删除，甚至还有基本块内的循环展开（loop unrolling）。

```
*   1:   ------ IMark(0x24F275, 7) ------
    2:   t11 = GET:I32(320)          # get sh(%eax)
*   3:   t8 = GET:I32(0)             # *get %eax
    4:   t14 = Shl32(t11,0x2:I8)     # shadow shll
*   5:   t7 = Shl32(t8,0x2:I8)       # *shll
    6:   t18 = GET:I32(332)          # get sh(%ebx)
*   7:   t9 = GET:I32(12)            # *get %ebx
    8:   t19 = Or32(t18,t14)         # shadow addl 1/3
    9:   t20 = Neg32(t19)            # shadow addl 2/3
   10:   t21 = Or32(t19,t20)         # shadow addl 3/3
* 11:   t6 = Add32(t9,t7)            # *addl
   12:   t24 = Neg32(t21)            # shadow addl 1/2
   13:   t25 = Or32(t21,t24)         # shadow addl 2/2
* 14:   t5 = Add32(t6,0xFFFFC0CC:I32)   # *addl
   15:   t27 = CmpNEZ32(t25)     # shadow loadl 1/3
   16:   DIRTY t27 RdFX-gst(16,4) RdFX-gst(60,4)
         ::: helperc_value_check4_fail{0x380035f4}()
                                  # shadow loadl 2/3
   17:   t29 = DIRTY 1:I1 RdFX-gst(16,4) RdFX-gst(60,4)
         ::: helperc_LOADV32le{0x38006504}(t5)
                                  # shadow loadl 3/3
* 18:   t10 = LDle:I32(t5) # *loadl
```

图 6 加载工具之后的简单 IR

Figure 6 Flat IR with instrumentation

该阶段对图 5 中 IR 的修改如下：指令 2 中的复杂树状指令被分解成了 5 条简单指令，包括 2 个 GET，2 个 Add32，一个 Shl32；指令 3 中的 PUT 被改成了对一个临时变量的赋值；指令 5 被删除，这是因为指令 15 对%eip 重新赋值，使得它成为了冗余指令；指令 6，7 和 16 被移除，这是因为指令 2 展开后，使得它们的 GET 成为了冗余。

第三步：对简单中间代码进行工具加载（instrumentation）。在该步中，代码基本块被传递给了一个工具，该工具可以对基本块中的指令进行必要的修改。在将中间代码基本块传递给工具之前一定要将其简化，这将极大地降低工具处理的难度。加载工具之后的中间代码如图 6 所示，其中加*标记的指令是加载工具之前就有的指令。该例子中的工具是一个 shadow value 的工具，即为需要做工具的寄存器或变量提供副本。

第四步：树状指令重建。该步将加载工具后的简单中间代码指令重建成复杂树状指令，为下一步中的指令选择做准备。如果有一个表达式对一个临时变量赋值，而该临时变量只使用了一次，那么可以直接在使用它的地方把它用相应的赋值表达式来代替。结果是 load 操作的顺序可能会有所改变，但是 load 和 store 的顺序绝不会颠倒。

第五步：指令选择。在该步中，指令选择器将树状中间代码转化成一个使用了虚拟寄存器的指令列表。指令选择器使用的是一种简单，贪婪，自顶向下的树匹配算法。

第六步：寄存器分配。在该步中，一个线性扫描的寄存器分配器将指令列表中的虚拟寄存器替换成宿主机的实际寄存器。尽管目标代码指令是与特定平台相关的，但是寄存器分配器却是与平台无关的，它采用了一些回调函数来找出每条指令都分别对哪些寄存器进行了读和写的操作。图 7 中给出了一个寄存器分配的例子，其中虚拟寄存器用%%vrNN 给出。由例子中可以看出，寄存器分配器可以移除许多寄存器之间的相互 move 操作。

```
-- t21 = Or32(t19,Neg32(t19))
movl %%vr19,%%vr41                movl %edx,%edi
negl %%vr41                       negl %edi
movl %%vr19,%%vr40
orl %%vr41,%%vr40                 orl %edi,%edx
movl %%vr40,%%vr21
```

图 7 寄存器分配之前和之后

Figure 7 Before and after register allocation

第七步：编译。最终，指令列表中的指令被转化成了本地的机器代码。

源二进制代码在经过了一系列的转换和优化后，最终被翻译成了本地二进制

代码，然后将翻译后的代码存放在代码 cache 当中，由调度器调度执行。

2.6 Valgrind 中的优化

Valgrind 的主要工作不是做翻译，但是却附带实现了翻译的功能，如果写一个什么都不做的或者只针对二进制翻译优化的工具，那么 Valgrind 就是一个动态二进制翻译系统。这个翻译系统无疑很理想：利用中间代码很容易实现多源多目标、利用中间代码进行各种跳转处理等等。

其实在 Valgrind 里无处不体现着二进制翻译和优化的思想。除了前面提到的在翻译时进行的冗余代码删除，常量复制与传播，死代码的移除等编译器的常规优化之外，还在运行期进行了跳转优化。Nicholas Nethercote（Valgrind 的主要开发者之一）在他的博士论文中提出当控制流从一个基本块转移到另一个基本块时，执行速度由快到慢分为三个层次：chainning、dispatch、scheduler。

chaining 就是直接跳转的概念，通过调用一段手写的汇编代码 VG_(patch_me) 函数实现直接跳转。每次一个基本块执行完毕之后，系统就知道它下一个跳转的基本块位于 cache 中的什么位置。这样，可以直接把跳转地址修改为 cache 中的地址，下次执行时就不用再通过调度器而直接跳过去执行。如图 8 所示，对于那些频繁的跳转，可以节省很多上下文切换的时间。

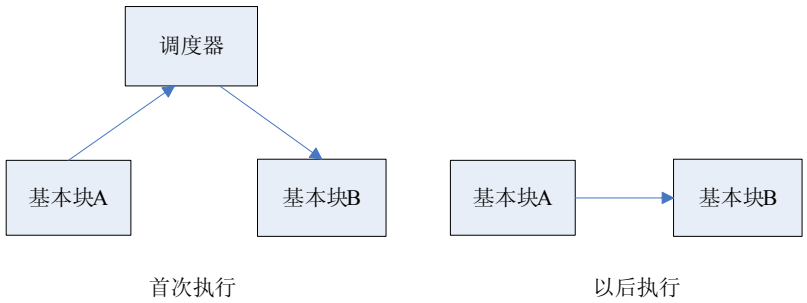

图 8 chaining 之前和之后

Figure 8 Before and after chaining

虽然通过 chaining 可以节省很多上下文切换的开销，但是在发生 cache 替换时则不得不考虑代码基本块之间的 chaining。由于 chaining 是根据代码 cache 中的地址来进行的，如果上图中的基本块 B 被替换出去，而放入了一个新的基本块 C，而此时如果基本块 A 的 chaining 还没断开的话，下次执行时就会直接跳往基本块 C，这显然是我们不希望看到的结果。因此，在每次代码 cache 发生替换时，valgrind 都会对 cache 中的基本块进行扫描，断开所有与替换出去的基本块之间的 chaining。

dispatch 发生在基本块结束同时又无法进行直接跳转的时候去查找一个更小的 Fast Cache（所有代码 Cache 的一个子集）。dispatch 用汇编写成，指令不超过

20 条。Fast Cache 中存放了近期执行过的代码基本块。由于程序执行有代码局部性原理，近期执行过的代码在未来很有可能被执行到，因此每次都先跳转到 Fast Cache 当中进行查找。

仅当 dispatch 没有命中时，控制权才转回到 scheduler，此时调度器按照常规的做法来进行处理。首先在普通代码 cache 中查找需要执行的基本块，如果查找成功，则将其存放在 Fast Cache 之中，然后执行；否则，说明该基本块还没有被翻译，调度器启动翻译模块进行翻译和优化并将翻译后的代码存放在代码 cache 之中。

2.7 本章小结

本章以开放源代码的 valgrind 为实例介绍了动态二进制系统的构成以及工作流程。Valgrind 的特色之一是各种可以加载的工具，以及定义良好的工具接口，使用户可以很方便地编写自己的工具来对程序做 profile 和分析。总结起来，valgrind 有以下优点：

1. 它功能强大，除了做二进制翻译之外，还提供了功能丰富的各种工具，可以对不同平台的二进制代码进行剖析；

2. 它在翻译机制中引入了中间代码，容易实现多源多目标，可以针对中间代码进行修改，达到各种目的，如监控程序等；从二进制翻译的角度看，利用中间代码做基本块连接以及跳转的优化是比较方便的；中间代码还有一个好处是可以充分利用目标机器的寄存器；

3. 模块化较好，平台的差异以库（二进制翻译库 VEX）的方式提供，以回调函数的方法允许第三方开发者开发针对其它平台的二进制翻译系统；

4. 提供可选项，供第三方开发者有选择的添加或删除某些功能模块；还可以使第三方快速开发自己的各种二进制代码分析工具。

同时，由于 valgrind 提供的功能过于庞大，性能也随之下降，而且强大的功能还使得系统自身的结构变得复杂。但是作为一个为第三方提供的开发和研究平台，valgrind 还是一个非常理想的平台。

第三章　Profile 和热路径优化

3.1　引言

动态优化是提高动态二进制翻译以及其它运行时系统性能的重要手段。在进行动态二进制翻译时，异地代码被沿着控制流的顺序，边翻译边执行，用户看到的 “执行”过程伴随着翻译、执行等不同阶段的交替，因此除了保证正确性外，高效的性能是实现可用性的关键。由于在运行时，系统不适合展开类似静态编译时的那些复杂度高、开销大的优化，由此便需要引进一种专门针对运行时系统新型的优化模式，这就是动态优化。动态优化的独特之处在根据程序行为选择运行时的热区域作为优化的单位，这样既提高了优化的利用率，又降低了不必要的优化开销。

除了优化动态二进制翻译这个应用背景外，动态优化本身具有更广泛而深刻的技术前景[19]。随着软件技术的发展，为了提高软件重用性，广泛采用延迟绑定成为软件开发的流行趋势，例如面向对象的语言以及动态链接库等，由于静态编译时难以跨越动态绑定的边界进行分析和优化，因此延迟绑定导致的一个后果是阻碍编译优化的开展，虽然提高了重用性，但性能势必有所下降。动态优化恰好可以弥补静态编译的不足，由于运行时可视范围扩大到整个程序的执行轨迹，因此优化可以跨越动态链接或者虚函数等静态编译器无法跨越的边界。

从另外一个角度看，动态优化是现代编译领域基于反馈优化思想的一个发展方向。一些研究者强调[20]应该摆脱根据程序员定义的代码边界划定优化单位的传统，取而代之的是将优化单位的确定建立在程序运行行为基础上，动态优化正是这种思想的体现。因此相对所要进行的优化内容而言，有关如何捕捉程序的动态行为以及划定优化单位的研究更有意义，这也是本章内容的研究动机。

对于动态优化来说，要选择这样的优化区域：首先，要体现程序的执行轨迹，具有动态局部性和执行顺序性；第二，代码的执行频率高，具有高度重用性；第三，便于在其上展开高效的优化。根据以上的标准，程序执行的热路径是非常合适的优化区域。所谓热路径是指频繁执行的动态指令序列。首先它们可以反映程序动态执行的特征轨迹，具有很好的局部性和顺序性，基于热路径的代码布局有利于提高指令 Cache 和分支预测的命中率。另外由于控制结构简单，在路径上易于快速高效的实施优化，比如展开基于热路径的指令调度[21]。

由此，热路径的选择成为动态优化中的关键问题之一。在基于 profile 反馈的编译优化中通常也需要选择热路径作为后端优化的依据，然而这里 profiling

的性质是离线的，即通过程序的一次预备执行来专门收集 profile 信息。而动态优化中 profile 信息的采集和利用是在同一次程序执行时完成的，因此对运行效率要求比较高，低开销的 profiling 是动态优化的基本要求。另外一点不同是，离线 profile 的目的在于总结程序的行为，而动态优化中的 profile 在于预测[22]，因而更讲究实效性。运行时要通过当前已经收集的 profile 信息及时的预测下一个热路径是什么，这个观察的过程不能太长，以免既浪费了优化机会，又增加 profiling 的开销。

有些硬件提供了程序历史的采样记录机制[23]，这为热路径的选择提供了比软件 profiling 更高效的途径，并且由于硬件可以检测到处理器的微体系结构行为，因而可以利用其在热路径上开展更细粒度的优化。但硬件支持并不具有通用性，难以在大多数平台上应用，因而本文仅探讨利用软件 profiling 选择热路径的方法。

就像计算机科学中很多问题一样，动态热路径的选择不是一个求唯一最优解法的问题，而是要权衡各方面相互制约的因素，给出在某些情况下的适应系统实际情况的解决办法，这类问题需要的是一种有效的方法，而非一味追求全局最优。在动态选择热路径时，存在这样一些相互制约的因素，例如，我们希望能够较早的预测到程序中确实热的路径，这就需要降低预测时的热度阈值，这样一来大量并不很热的路径也通过了筛选，不但干扰了优化，同时也增加了对内存缓冲区大小的需求，对于某些内存紧张的系统，如嵌入式系统来说，对内存的需求可能是极其昂贵的。由此阈值的设置不宜过小。反之，如果提高预测的阈值，虽然可以淘汰掉大量非热的路径，但预测延迟的拉长会导致 profiling 开销的加大，并且浪费了真正热路径的优化机会，另外热路径的预测效果并不一定由于预测期延迟的加长而更加准确，例如某些程序具有 phase 行为[24]，热路径会在不同的 phase 呈现不同的走势，程序执行前期预测的结果可能在后期一点也不适用。因此阈值的设置也不宜过大。本文后面有 profiling 开销随阈值变化的分析，具体的系统可以根据相应的情况来调整阈值设定。

本章余下部分将详细介绍动态二进制翻译中的 profile 方法以及热路径的识别和优化方法。结构如下：第 2 节介绍动态二进制翻译中的 profile；第 3 节介绍二进制翻译中常用的热路径识别方法；第 4 节介绍一种改进的热路径识别和优化方法；最后第 5 节给出实验结果和分析评价。

3.2 动态二进制翻译中的 profile

Profile 是一种对特定对象的统计信息，它一般采用二元组<id，counter>的形式来表示，其中 id 是被统计对象的唯一标识，counter 是该对象的统计信息。Profile

常用于程序动态执行信息的获取，程序执行过程的检测等方面。在二进制翻译中，profile 的对象可以是简单的基本块，控制流程图中的一个跳转，也可以是整条路径。这些信息可以用来识别程序中频繁执行的代码，即热代码，并对其优化，如代码重排，超级块的生成[25]等。

在静态二进制翻译中，profile 信息的收集是单独进行的，即先运行程序，获取各部分代码执行的详细信息，从中找出热路径进行优化，然后再重新编译。早期的静态二进制翻译器如 VEST [35]等，都比较成熟，而且在这方面也有比较优秀的算法，如[36]中的高效热路径识别。但是在动态二进制翻译中，翻译是即时进行的，对效率有较高的要求，因此不可能采用复杂的算法，也不可能收集程序的全部信息。一般的动态二进制翻译器都采用热路径预测的方法，即先运行程序一段时间，收集在这段时间内代码重复执行的情况，然后在此基础上做出预测。一旦识别出热路径，就对其进行优化，包括热代码连接，建立间接跳转 cache 等[25]。

由于在动态二进制翻译中对 profile 的性能有较高的要求，要能进行快速的查找和更新 profile 信息，因此常用 hash table 的方式来存放收集到的 profile 信息。

3.3 动态二进制翻译中的热路径识别

在动态二进制翻译中，常见的 profile 和热路径识别方法有三种：基于基本块 profile 的热路径识别，基于边 profile 的热路径识别，以及基于路径 profile 的热路径识别。三者预测的准确率递增，但实现的难度和算法的复杂度也随之增加。在实际应用中可以根据具体情况来选择一种或集中的组合。

3.3.1 基于基本块(basic block)profile 的热路径识别

基于基本块 profile 的热路径识别实现最为简单。在程序运行过程中，对每个基本块的执行次数进行统计，一旦达到预先设定的阈值，便认为当前代码已经足够“热”，开始生成热路径。方法如下：

首先找到循环的入口基本块，把他作为热路径的开始块；然后每次遇到条件跳转时就简单地比较两个目标块执行的次数，并把执行次数较多的那个块作为热路径上的下一个基本块；最后，当遇到循环出口时便认为热路径结束。至此，一条热路径便被识别出来，将其优化翻译之后以合理的方式保存在代码 cache 之中，下次被运行时可以直接调用。

3.3.2 基于跳转边(edge)profile 的热路径识别

利用上述基于基本块 profile 的算法可以以较低的代价生成一条热路径，但是由于 profile 的对象是孤立的基本块，收集的信息不够全面，预测错误的概率较大。这是因为上述算法只是简单的考虑了每个基本块被执行的次数，而没有包含基本块之间的跳转信息。

基于跳转边 profile 的热路径识别算法是对上述算法的改进。在该算法中，不去收集基本块被执行的次数，而是记录了基本块之间的跳转次数。这里我们用一个二元组(A, B)来表示从基本块 A 到基本块 B 的一次跳转，用 N(A,B)表示该跳转发生的次数。显然和每条边相关联的基本块有两个，即源块 A 和目标块 B，它准确的反映了基本块之间的跳转信息。我们对 2.1 节中的算法稍作修改：在遇到条件跳转时，我们比较以当前块 A 为源块的两条边(A,B)和(A,C)的跳转次数，若 N(A, B)>N(A, C)，则把 B 作为热路径上的下一个基本块，否则选择 C。

该算法需要的信息比基于基本块的要多，但预测更准确；同时其实现简单，但预测的准确率却和基于路径的算法相当，因此该算法为许多二进制翻译器所采用。[26]为基于边的算法提供了理论和算法基础。

3.3.3 基于路径(path)profile 的热路径识别

虽然基于边的热路径算法预测较为准确，但是在路径有较多重叠的基本块时，预测难免发生错误。原因是每条边只是一条路径的一部分，用部分来预测整体明显不够准确。改进的方法是为每条从循环入口到出口的边建立一个档案，记录每条路径被执行的次数。这种方法预测的准确率最高，但是由于每条边有多个基本块，我们必须保存每条边所包含的基本块的信息。另外，随着循环内部跳转增多，从入口到出口的路径数目会非常庞大。因此必须找到一种高效的算法。

3.3.4 基于跳转边的 profile 与基于路径的 profile 的对比

基于跳转边的 profile 方法是传统和主流的作法，根据 edge profile 可以容易确定那些高度频繁执行的热路径，例如图 9[26]中控制流图 g1 中的路径{e1, e2, e3}，以及 g2 中的{e1,e5,e3)。根据 edge profile 选择热路径的算法非常简单，热路径的选取按照从热到冷的顺序，首先是最热的路径:先选择一个起点，既可以是一个节点，也可以是一条控制流边，一般来说选择该编译区域的入口节点作为热路径的起点。而后沿着执行频率高的一边将后续节点添加进去，直到编译单元的结束节点。最热的路径选出之后，挑出先前未被选中的出口节点作为下一个路径的起点，继续如法炮制选出次热的路径。

但基于 edge profile 的热路径预测并非一定行之有效，例如图 9 中的 g3 和 g4, edge profile 对路径走向的限制是非常有限的。g4 中 8 条路径的执行次数可能是平均的，也可能其中只有两条路径很热，还有可能出现其它复杂的情况。g3 的情况比 g4 稍微简单一些，热路径只能有 2 到 4 条。这种情况下就要通过 path profile 来预测热路径了。Ball& L arus 提出的一种高效的 path profiling 算法[36],所考察的路径是过程内无环正向路径，它们以控制流回边或者过程的返回节点为结尾。该算法需要先对程序控制流结构进行预分析，以便建立一个最小化的路径编码。而后应用生成树算法选出代价最小的边集进行插装。被插装的控制流边携带了足够的信息在运行时唯一确定每条执行路径的路径号。应用 Ball& L arus 的算法导致的运行时开销大约为 30%，而一般来说，通过插装提取 edge profile 或者 blockpr ofile 信息会导致大约 16%的运行时开销[26].B all& L arus 的算法极大的提高了 path profiling 的效率，但算法实现的复杂性也随之提高。

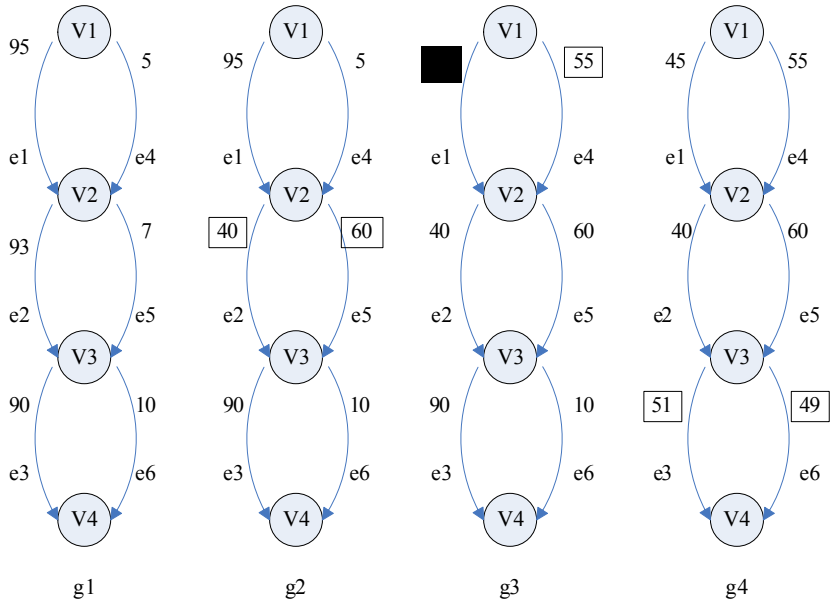

图 9 edge profile

Figure 9 edge profile

根据 edge profile 预测热路径已经具有相当的可用性，这在现代编译器的应用中已经得到体现，那么是否有必要采用复杂的 path profiling 技术来代替 edge profiling 呢？Ball 等做了理论上的研究[26]来比较两者的实用性，即什么情况下 edge profile 足以成为预测热路径的依据，什么情况下不能，为了描述 edge profile 能够提供的约束信息，定义路径的两个度量，一个是“必然频率”，另一个是“可能频率”。必然频率是指根据控制流图的 edge profile 信息可以确定某条路经至少经过的次数，而可能频率则是根据 edge profile 判断的该路经最多有可能经过的次数。例如 g1 中的{el,e2 ,e3 }至少经过了 78 次，因为非该路径总共经过的次数最名不超过(5+7+10)=22 次，因此由 gl 可以得出{e1,e2 ,e3 }的必然频率是 78。{e1,

e2, e3}最多可能经过 90 次，也就是取路径上各条边执行次数的最小值，于是{e1,e2 ,e3 }的可能频率为 90。而后根据必然频率和可能频率分别推测出必然热路径和可能热路径。经过与 path profile 预测热路径算法的比较发现，如果程序中必然热路径的比例很高，则基于 edge profile 的热路径预测比较准确，相反如果必然热路径的比例比较低，则 edge profile 的预测成功率下降，此时需要 path profile 来帮助准确地寻找热路径。

3.3.5 NET 动态热路径预测策略

Dynamo [29]是 HP Lab 开发的动态优化系统，最初基于 PA-RISC 体系结构的机器。作为一个出色的动态优化系统，Dynamo 采用了新颖的热路径预测方式，被称为 NET(NextE xecutionT ail) [22]。NET 预测的目标是，在显著降低 profiling 开销的前提下，实现和 path profile 预测相当的效果。由于其新颖和高效性，很多系统采用了类似 Dynamo 的热路径选择方法，例如 DynamoRIO [37],Walkabout [16], Mojo [38]等。

Dynamo 的 NET 同样也是一种“先热先选择”式的策略。在 NET 中，路径被分为路径头和路径尾两部分，路径头是指路径的起始点，路径尾则指剩下的部分。Dynamo 通过仅对路径头进行 profiling 而投机预测路径尾来降低 profiling 的开销，这种策略的出发点是一个热的路径头意味着程序正执行于热区域，而紧接其后执行的路径则很有可能在那个区域中。

Dynamo 中的路径起始条件为：

1．向回跳转成功的目标(target of a backward taken branch)；

2．已生成热路径的出口目标。

当解释到满足上述条件之一的地址时，关联的 profiling 计数器加 1。如果计数器达到热度阈值，则进入热路径生成模式，这也就是“先热先选择”策略中的热路径触发部分。

进入热路径生成模式后，紧接着路径头执行的动态执行序列将被纳入新生成的热路径中，直到满足下面的路径终止条件之一：

1．成功的向回跳转(a backward taken branch)；

2．缓冲区已满。

一旦终止条件满足，新的热路径生成之后，与该路径起始地址关联的 profiling 计数器被清零回收，用于以后 profile 其它的路径起始地址。

基 Dynamo 的 NET 策略选出的热路径则反映了程序执行瞬间的行为状态，或者叫程序执行的一个快照(snapshot)，由于是程序执行的快照，因而由 Dynamo 方法选出的热路径有可能体现程序控制行为的关联性，这一点是该策略最突,的优点之一。

3.4 改进的热路径识别和优化算法

不管是前面提到的基于基本块的热路径算法，还是基于跳转边的热路径算法，每当发生一次跳转时都要更新计数器。这样频繁的更新计数器必然会带来不小的开销。基于路径的算法只对整条路径进行计数，而忽略了中间细节。但是我们必须找一种有效的方法来记录一条路径所包含的基本块。

3.4.1 热路径的编码表示方法

我们可以用编码的方法来表示一条路径。在每个基本块结束时，我们忽略直接跳转，而条件跳转有两个目标地址，因此我们可以用二进制编码的方法来记录跳转信息，如图 10 所示：

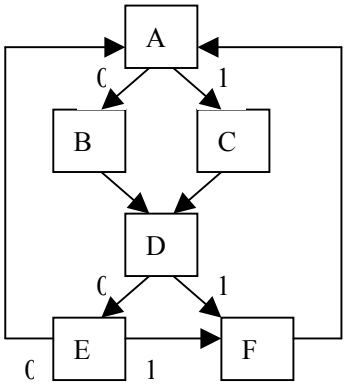

图 10 路径的编码

Figure 10 Coding a path

如果跳转的目标地址是 address1，我们用 0 来表示；如果跳转目标地址是 address2，我们用 1 来表示。用上面的编码方法，每条路径唯一的对应于一个码字，如表 2 所示。

表 2 路径及其编码

路径	ABDEF	ABDE	ABDF	ACDEF	ACDE	ACDF
码字	001	00	01	101	10	11

由于(B,D)和(C,D)之间的跳转是直接跳转，只有一个跳转地址，我们不必对其编码，这样可以有效减短码字的长度。

3.4.2 基于路径的热路径算法的分析

我们前边定义的编码方法只针对条件跳转，码字的每一位都对应一个分支。如果我们将这些分支用二叉树来表示的话，则从根节点到叶子节点的每条路径都对应从循环入口到出口的一条路径，如图 11 所示。

假设路径的平均长度为 k，即二叉树的高度为 k，那么不同路径的数目约为 2^k，它将随路径的长度呈指数型增长。当 k 比较大时，为了找出一条热路径，我们不可能为每条路径都设置一个计数器。文献[22]中用 NET 预测算法，通过路径头(即入口基本块)的信息来预测热路径。该算法可以减小长度，从而降低预测延迟，但仍有一定的盲目性，而且在生成热路径时还需要做一些额外工作。

基于路径的热路径算法的另一个缺陷是，随着路径长度的增加，路径的数目急剧膨胀，我们预测的命中率也就急剧下降。最坏的情况是当每条路径被执行的概率差不多时，这时热路径命中的概率为 2^{-k}。由于程序的运行有局部性原理，每条路径的概率都不一样。但是在复杂程序中，频繁执行的路径不止一条，假设为 m，则命中概率为 $1/m$，随 的增加而迅速减小。这种情况同样存在于另外两种热路径算法。

因此，我们一方面要尽可能减少信息收集的工作量，另一方面我们要针对各条路径均匀执行的情况进行特别的处理，以提高优化后代码的利用率。

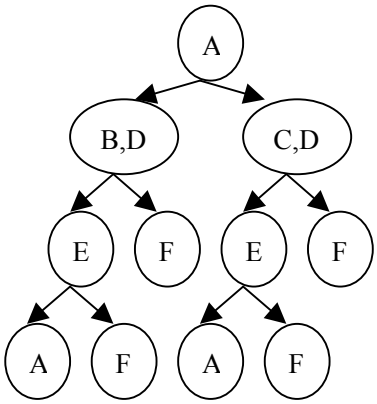

图 11 路径的二叉树表示

Figure 11 representing paths by a binary tree

3.4.3 基于编码的路径的形式化定义

由于每条路径都唯一对应一个码字，我们可以定义起始地址为 A，长度为 k 的路径如下：

$$p(A,k)=\{A,x_1x_2\cdots x_k \mid x_i=0,1\}。$$

即一条路径可唯一的表示为一个基本块的地址和一串 0，1 序列，其中的 0 或 1 代表了从循环入口到出口之间所经历的基本块。

同理我们可以定义一条路径上的一个段(fragment)如下：

$$f(B,l)=\{B,x_1x_2\cdots x_l \mid x_i=0,1\}$$

它表示从循环中任一个基本块 B 开始，长度为l的一段路径。

有了路径和段的概念之后，我们定义两条路径的交运算如下：

$$F=p(A,k)\cap p(B,j)$$

其中F是一个段的集合：

$$F=\{f(B_i,l_i)\mid f(B_i,l_i)\subseteq p(A,k),\\ f(B_i,l_i)\subseteq p(B,j),1\le l_i\le \min(k,j)\}$$

F中包含了两条路径$p(A,k)$和$p(B,j)$相重叠的基本块序列。

假设在热路径被识别出之后路径 p 被执行的次数，即执行频率为$q(p)$，阈值为 N，那么当某个循环内部有 k 条路径时，总的循环次数大约为$\sum_{i=1}^{k}q(p_i)$+N*k。

我们定义路径$p_j(1\le j\le k)$的命中率为：

$$h(p_j)=q(p_j)\Big/(\sum_{i=1}^{k}q(p_i)+N*k)$$

那么预测错误的概率即为 $1-h(p_j)$。我们的目标就是用尽可能低的代价来提高命中率$h(p_j)$。

3.4.4 算法实现

为了准确识别出热路径，必须进行较为详尽的信息收集。传统的做法是对所有的基本块执行情况进行跟踪，并通过不同的计数器来保存执行信息。这显然是一笔不小的开销。由于热代码只占所有代码的一小部分，因此我们应该尽可能只收集那些有可能成为热代码的基本块的信息。我们的做法是寻找那些往回跳的边，这往往是一个循环的开始。因此，我们可以只收集可能在循环内部的基本块的信息，这样就大大减少了开销。

用二进制编码的方法可以使码字与路径一一对应，为每条路径维护一个计数器即可。路径的出口发生在回跳处或路径达到规定的最大长度处。设所有路径的最大长度为 L，这样防止了程序在顺序执行时路径过长。

当某条路径的计数器达到阈值 N 时，便认为它是热路径，其码字决定了它所包含的基本块的信息。当一条路径的计数器值远远大于同一循环内其他路径时，

我们称它为绝对热路径，否则称其为相对热路径。对于绝对热路径，由于它的所有基本块都是热块，可以直接进行优化。但是在有些情况下，特别是循环体内条件分支比较多的情况下，会出现两条或多条路径平均执行的情况。这时识别出的是相对热路径，它的基本块即有热的，也有冷的。我们只想把优化作用在热代码中，以便减少不必要的开销。为此，我们要识别出相对热路径中的热段。

热段在路径中有三种可能出现的位置，如图 12 所示。当热段位于路径两端，实现比较简单；当热段位于路径中部时，实现较为复杂，但由于这种情况相对于另外两种出现的频率较低，因此这种情况将不予考虑。在实际应用中，大部分热段都位于路径首部，而且这种热段识别起来最为简单，本文将主要考虑这一种情况。

为了找出热段，我们将同一循环内两条执行次数最多的路径对应的码字分别按高位对齐进行按位异或，结果码字中连续为 0 的片断对应的段即为候选热段。当候选热段在原路径中的首基本块相同时便可被确认为热段。

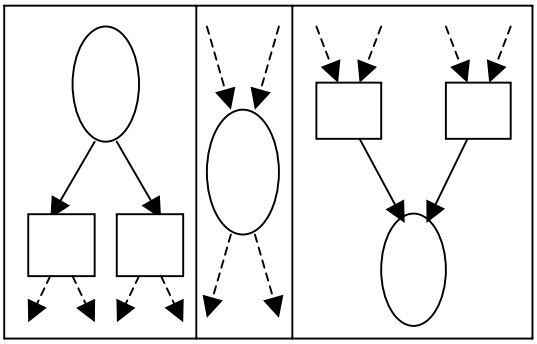

图 12 热段在路径中的位置

Figure 12 The position of a fragment in a path

假设有两条相对热路径： $p(A,7)=\{A,0110010\}$， $p(A,6)=\{A,011110\}$，按高位对齐位异或后结果为：

$$0110010\text{^}011110 = 000111$$

这样前三个基本块序列便可被确认为一个热段。由于热段长度一般都比较短，我们可以将其中的基本块合并为一个超级块，为多条路径所共用。

超级块是程序中频繁执行的几个基本块序列，它一般有一个入口，多个出口，如图 13 所示，箭头上的数字表示该跳转发生的次数。那么，由于 A，B，D，F 频繁执行，可以将其合并成一个超级块。如果程序在超级块内部运行，将不用通过调度器查找下一个基本块，减少了上下文切换的次数。

在生成超级块时，首先从首块开始，依次将下一个块的代码插入。如果当前块最后是无条件跳转，则删除该跳转指令，直接将下一块的指令插入；否则，则保留一个失败出口地址，然后将下一块的指令插入，直至所有热段上基本块都合并成一个超级块。

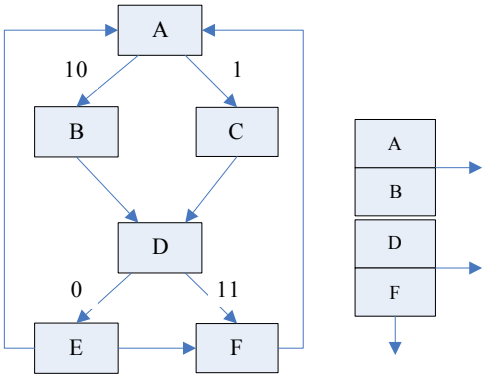

图 13 超级块

Figure 13 Superblock

3.4.5 实验结果与分析

该实验在 valgrind 平台上进行。识别热路径的第一步是进行信息收集，采用不同的策略在效率上有较大的差异。表 3 中给出了三种策略的具体数据。我们看到，跳转边的数目最多，为每条边维护一个计数器将会为系统增加较大的负担。相比之下，路径的数目要少得多。由于每条路径仅需一个码字即可表示，因此不但可以节省内存空间，同时减少了更新计数器的频率，可以以相对较低的开销收集到更为详尽的信息，作出更准确的预测。

表 3 计数器个数统计

Benchmark	基本块	跳转边	路径
Num Sort	1666	2186	700
String Sort	1793	2358	750
Bitfield	1703	2235	713
Emulation	1889	2526	827
Fourier	1701	2215	697
Assignment	1761	2339	762
Idea	1754	2333	763
Huffman	1743	2296	739

我们在前边定义了绝对热路径和相对热路径，两者的处理方法略有不同。图 14 给出了 benchmark 中各个程序所包含的绝对热路径和相对热路径的情况。从结果中可以看出，一般情况下绝对热路径数目略多，但是在 Assignment 和 huffman 中，相对热路径和绝对热路径平分秋色。这是因为这些程序中条件语句和 switch 语句数目众多，路径执行情况比较均匀。

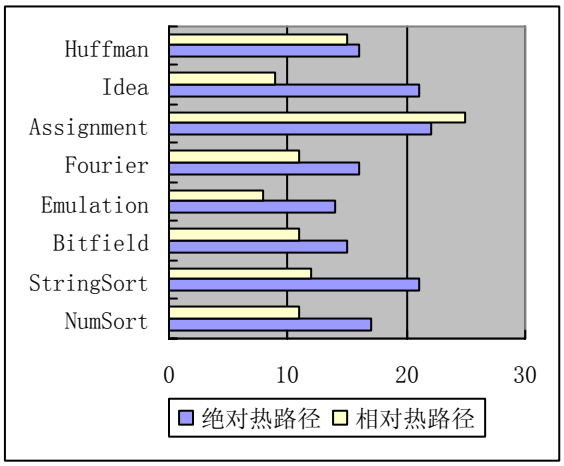

图 14 两种热路径对比

Figure 14 Absolute and relative hot path

在图 15 中，我们给出了三个有代表性程序的平均命中率。其中横坐标表示预测延迟[12]，即热路径在达到阈值之前执行次数占总次数的百分比。我们在定义命中率时已经考虑到了预测延迟。这是因为，如果单从命中的概率来说，当然预测延迟越大预测越准确。但是我们希望热路径带来的收益尽可能的大，因此希望预测延迟尽可能的小。最终我们选择从总体收益来衡量。我们从图中可以看出，大部分程序的最高命中率在 10%左右的预测延迟处获得。此时热路径已经被正确地识别出，如果继续增大预测延迟，只会减少热路径的收益，因此从 20%往后，命中率将平滑地下降，按照我们的定义，当预测延迟达到 100%时，命中率为 0，即此时热路径将不会带来任何收益。

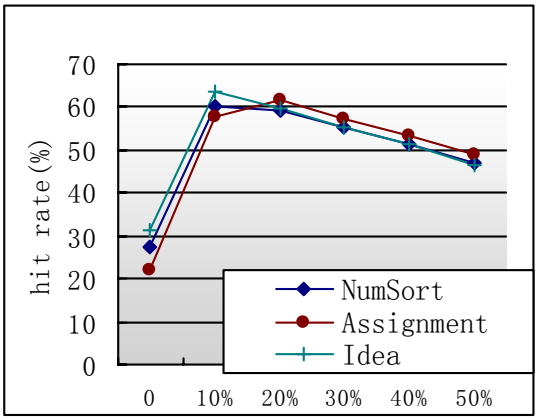

图 15 命中率

Figure 15 Hit rate

从图 15 中可以注意到，在预测延迟为 0，即进行盲预测时，Assignment 的命中率最低。这是因为其中的相对热路径最多，直接预测时选择较冷路径的概率较大。这和图 14 中相对热路径的统计结果一致。

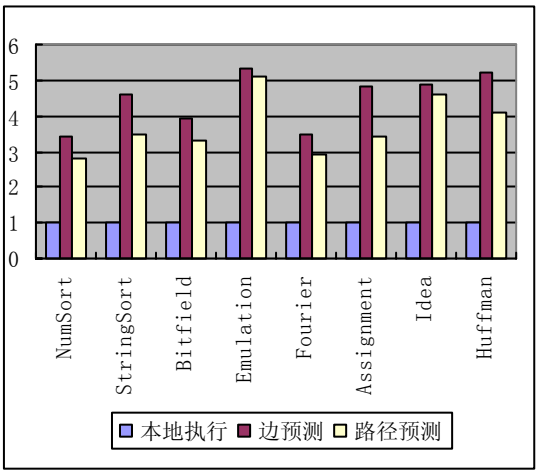

图 16 运行时间

Figure 16 Total run time

图 16 中所示是以本地直接执行为基准的运行时间。纵坐标表示的是边预测和路径预测相对于本地执行花费时间的倍数。由前面的分析知道，Assignment 和 Huffman 中条件分支众多，相对热路径数目相对较大，因此边预测效果应该和路径预测相差较大。这和实验结果基本吻合。

总之，改进的路径预测算法无论在提高预测准确度方面，还是在降低系统开销方面都有着明显的优势。

3.5 相关研究

3.5.1 静态 profiling

在应用于动态优化之前，控制流 profile 就已经被广泛用于辅助编译优化。例如 superblock 调度[21]就是基于控制流 profile 展开的。在控制流 profile 中，path profile 提供了比 edge profile 更精确的路径信息，但是实现比较复杂，开销较大。Ball 和 Larus 设计的 path profiling 算法[36]可以高效的记录过程内各路径的执行频率，而 Young 和 Smith[39]则通过类似串匹配的方法将 profile 的路径形态扩展到更一般的形式，并通过该 path profile 进行静态关联分支预测，取得了比较好的效果。为了降低 profile 的开销，还出现了基于统计采样的低开销 profiling[28]方法。但静态 profile 是对程序整个执行过程的平均行为的记录，因而缺乏时效性。

3.5.2 基于硬件采样的动态优化

基于硬件解决方案的动态优化采用特殊的硬件机制支持来收集和描述程序

的性能和行为信息。在[23]中，特别设计了“热点”检验硬件机制，将频繁执行的分支序列存储于一个分支行为缓存(BBB)，并记录下分支的偏好信息。在[40]中，用一种类似 trace cache 的结构捕捉频繁执行的热路径，并将优化后的热路径存储起来用于在执行中替换已有的代码，这种机制的有点是可以透明而低开销的检查到热代码以及用于 profiling 的分支热度信息。在[41]中，通过指令路径协处理器不断的将 CPU 执行过的执行序列转化成更有效的形式，并且实现了 stride 数据预取以及指针列表预取。在[42]中，硬件只是用来收集原始的性能数据，而热路径的选择通过软件来完成，相对专用硬件实现机制来说，软件方法会引起更多开销，但软件的优势在于易于更新性，并且可以利用更大的缓存。

通过统计采样来检测程序热点的系统包括 DCPI[43]，Morph[27]以及 Spike[44]，[28]中的分支预测采样也利用了类似的机制。

3.6 本章小结

动态 profiling 以及热路径的识别和优化是动态二进制翻译优化中非常重要的一个方面，几乎所有具有动态优化功能的二进制翻译系统都做了相关的优化。与静态基于控制流的 profile 不同，动态热路径预测需要更简便、更具实效性的 profiling 算法。动态 profiling 策略可分为两大类，一类是纯软件实现的 profiling 方式，另一类是利用特殊硬件支持的统计采样方式。硬件方式可以减少运行时开销，但降低了通用性和易更新性，通常在研究领域采用硬件采样作为 profile 信息的来源。为了顾及通用性，本文选择软件 profiling 作为研究的内容。

本章首先介绍了三种基本的 profiling 和热路径识别算法，并详细分析了它们的优劣，然后在次基础上提出了一种改进的基于路径 profile 的热路径识别也优化算法，其特点是算法简单，但是包含的信息量却很大，是一种行之有效的热路径优化算法。

第四章　基于 profile 的代码 cache 管理

4.1 引言

动态二进制翻译器在翻译，执行源机器的二进制程序代码时，通常将翻译和优化后的代码通常放在代码 cache 中，以提高代码的重用率。当程序再次执行到这个基本块或热路径时就执行代码 Cache 中相应的代码块，从而避免重新翻译并提高程序执行速度。但是代码 Cache 不可能无限制增大，随着程序的运行代码 Cache 必定会被翻译生成的代码填满。根据程序执行的时间局部性特征，代码 Cache 中保存的代码有可能是很久以前运行的代码，现在和将来都不会再使用到。所以使用一定的策略用把这些旧代码替换出代码 Cache，代之以新翻译生成的代码是非常重要的优化方法。有效的管理代码 Cache 能充分利用空间，保证上面保存的代码都是将来最有可能被运行到的部分，从而提高动态二进制翻译器的性能。

从一般意义上来说代码 Cache 的空间越大，所能容纳的基本块也就越多。同时被丢弃的基本块和被重复翻译基本块也会相应减少。这些都非常有利于提高二进制翻译的性能。然而并不是所有的系统都能够分配出非常大的空间给代码 Cache，特别是在一些资源紧缺的嵌入式设备。而且随着代码 Cache 空间的不断增大，管理的开销（动态二进制翻译系统的开销 ＋ 操作系统的开销）也会相应的增加。在选择代码 Cache 大小的时候需要综合考虑以下三个因素：

1．实际硬件能为代码 Cache 提供的内存空间的大小；

2．源机器程序的大小以及翻译的代码膨胀率；

3．管理代码 Cache 空间的开销。

代码 Cache 大小选取将直接影响到动态二进制翻译器的性能。这非常类似于在设计计算机的时候寄存器数量的选择。寄存器虽然访问的速度非常快，可以大幅度的提高计算机的性能。但是寄存器并不能无限制的增加，它也考虑到寄存器的成本，寄存器在执行程序中的使用效率，和寄存器管理开销（需要对寄存器进行编号，寄存器的增加将需要更长的机器指令）等因素。综合考虑这些因数，现在的 RISC 机器都往往只使用 32 个通用寄存器。

在代码 Cache 空间不够大的时候，代码 Cache 的管理显得特别重要。代码 Cache 的管理非常类似于操作系统中的物理内存管理。用户程序不可能一次全部都加载到物理内存里面，因为物理内存一般没有足够的空间，而且这么做也是没有必要的。操作系统利用请求调页策略来实现内存和二级存储器之间来回传送存

储页。内存被不断新调入的页框所占据，当空闲页框数少于预定的阈值时，就要把内存中的一些页框写回二级存储器，为新调入的页框腾出空间。操作系统和代码 Cache 的管理一样，都要面对同一个问题：如何选择被换出的代码。所不同的是操作系统把内存的页框写回二级存储器，而且要保证这两者的一致性。而代码 Cache 直接丢弃被换出的基本块，如果万一程序下次还要运行这段代码，就必须重新翻译。

Cache 中代码块的替换算法必须要尽可能的简单，不给系统带来较大的额外开销，同时要考虑到代码执行在时间上的局部性，尽可能把那些现在和近期内活跃的代码块保留在 cache 中，避免相同基本块的频繁换出和重新翻译。现有的一些动态二进制翻译和优化器都采用了某种代码 cache 的替换算法，如 Dynamo [19]，Mojo [38]，以及 Wiggins Redstone 等。它们都是先进行翻译和运行时的 profiling，然后对热代码进行优化，翻译和优化后的代码暂存在 cache 中，下次执行时直接从 cache 中读取。

尽管动态二进制翻译中的代码 cache 和传统的硬件 cache 有相似之处，但是至少在以下方面有较大的差别：首先，代码 cache 中的基本块的长度是不固定的，这主要由源代码基本块的长度来决定，可以是数条指令，也可以是数十条指令；其次，为了跳转优化，基本块之间往往有跳转链接，当一个基本块被替换出去时，必须将与其相关的链断开；最后，当一个基本块被替换出去后，如果后面又要用到，只能重新翻译，因此缺失代价比较高。所有这些差别都对替换算法的选择有较大的影响。

通过对 nbench 中的测试程序进行基本块信息的统计，得到的基本块的大小信息如表 4 所示。从表中可以看出，在程序当中，基本块大小的变化幅度比较大，最大块和最小块的大小相差悬殊，因此不可能采用 cache 定长分配的方法，因为在替换中极易造成 cache 碎片。

表 4 基本块大小统计

Benchmark	最小	最大	平均
Num Sort	32	660	125
String Sort	48	672	128
Bitfield	31	628	105
Emulation	32	742	121
Fourier	32	656	115
Assignment	35	665	102
Idea	32	683	128
Huffman	32	560	99
平均	34	658	115

本章余下部分将详细介绍动态二进制翻译中的各种代码 cache 替换方法和策

略，以及一种新的基于 profile 信息的代码 cache 管理策略。结构如下：第 2 节介绍动态二进制翻译中的常见代码 cache 替换策略；第 3 节介绍基于 profile 信息的代码 cache 替换算法；第 4 节给出实验结果和分析评价。

4.2 常用替换算法

在传统的 cache 管理中，替换算法主要包括 LRU(Least Recently Used)，最佳大小匹配(Best-Fit Element)，全清空(Full Cache Flush)，以及分阶段清空(Preemptive Flush)等[45]。

4.2.1 LRU 策略

代码 cache 从低地址到高地址，按照基本块的翻译顺序依次存放每个基本块，当出现代码 cache 空间不足时，就将丢弃最近最久未使用的基本块，如果替换后代码 cache 剩余空间还不足，则继续丢弃与刚替换出去的基本块物理相邻的基本块，直至代码 cache 空闲空间足够分配给新翻译的基本块。这个算法的优点在于它考虑了程序的执行特性，以程序最近执行情况来预测程序将来的执行倾向，也考虑了程序的时间局部性。某些现代操作系统内核已经提出了 LRU 的置换算法，但是往往都要借助于计算机硬件平台的支持，比如说一些大型主机的 CPU 可以自动更新在每个页表项中包含的计算器，这些计数器能够准确的记录该页的年龄和被使用的情况。而在一般的情况下（如 X86 处理器）都没有提供类似的支持，因此 Linux 等操作系统也无法使用真正的 LRU 算法。

LRU 对传统的 cache 管理有较好的作用，但是在动态二进制翻译环境下却并不是最佳选择。首先，必须记录每个基本块的最近使用时间，以后每当一个基本块被执行时，都要更新它对应的 LRU 信息，这将给系统带来不小的开销；其次，LRU 算法选择的基本块很可能在 cache 的中部，而新的块又不一定刚好填满，从而造成大量的 cache 碎片。如图 17 所示，其中的 H，J，L 都是因替换而产生的碎片。

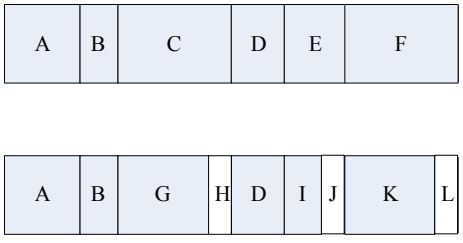

图 17 cache 碎片

Figure 17 Fragmentation in Cache

4.2.2 FIFO 策略

代码 cache 从低地址到高地址，按照基本块的翻译顺序依次存放每个基本块，当出现代码 cache 空间不足时，丢弃最先进入代码 cache 的基本块。如果替换后代码 cache 剩余空间还不足，则继续丢弃先进入的基本块，直至代码 cache 的空闲空间足够分配给新翻译的基本块。Dynamo RIO [46] 系统就是采用这个策略。FIFO 策略考虑了程序的时间局部性，而且由于替换一个基本块而牵连换出的基本块块也是下一次即将要换出的基本块，因而那些比较热的基本块被牵连换出概率也比较少，故其效率比较高，但是它没有充分考虑程序的执行特征，先进入代码 cache 的基本块不一定可以先被换出，进入代码 cache 的顺序并不能准确的反应基本块是否在将来被频繁使用。比如说有些程序就是会频繁的使用程序开头部分的代码，所以它还是有一定的局限性。

4.2.3 粗粒度的 FIFO 替换算法

粗粒度的 FIFO 替换算法将整个代码 cache 分割成若干个大小相等的大块，当代码 cache 被填满需要替换时，每次都将其中的一个大块进行全清空，而各个大块之间则采用 FIFO 的方法来进行管理。该方法即避免了 FIFO 频繁替换所带来的开销，同时又降低了全清空算法的盲目性。该算法由 Mojo [38]首先使用，它只将代码 cache 分成了两个相等的部分。Hazelwood[47]提出，当将代码 cache 分成 6 个相等的大块时能取得最优效果。全清空替换算法可以看作一种特殊的粗粒度 FIFO 替换算法，它只有一个大的块。

4.2.4 全清空算法

当代码 cache 没有足够空间时将其全部清空的算法是实现起来最简单和便捷的替换算法[48]。该算法有其自身的优点。首先，实现简单，不会带来额外的开销。其次，超级块是由一条频繁执行的路径来生成的。但是在经过一段时间之后，热路径可能会改变，即选择不同的分支执行。全清空的算法可以清除过时的热路径，按照当前的执行情况重新生成新的热路径。

但是该算法最大的缺点就是将所有的基本块全都清空，当然也包括当前活跃的那些块。这些活跃的块在全清空后将不得不重新翻译，导致清空之后的一段时间里有较高的缺失率。

4.2.5 分阶段清空算法

分阶段清空算法为 Dynamo [19]所采用它是在全清空算法基础上的改进算法。大多数程序的运行都有阶段性，而阶段的改变也就意味着工作集的移动，其结果是引入新的代码区域，从而导致大量的 cache 替换发生。通过对一段时间内的 cache 替换次数进行跟踪可以识别出程序阶段的变化，此时将代码 cache 清空的话损失的只是一小部分活跃的块，它带来的缺失率将小于全清空替换算法。

4.2.6 最佳大小匹配替换算法

最佳大小匹配替换算法是为了在不全清空的情况下尽量减小碎片的总的大小。该算法首先扫描整个代码 cache，找出能够容纳新块且大小和新块最接近的基本块，然后将其替换出去。该算法片面地减少碎片率，但无法从根本上消除碎片；而且由于没有考虑到程序的时间局部性，很有可能将当前活跃的基本块替换出去，从而增加缺失率。此外，该算法虽然在一定程度上减小了碎片的大小，但是每次替换时都要扫描整个代码 cache，在发生频繁替换时在效率上会有较大牺牲。

4.3 基于 profile 信息的替换算法

动态二进制翻译中的代码 cache 的替换和垃圾回收机制相似，都是将无用的对象清理出去，给新的对象分配空间。但是垃圾回收器能确切知道一个对象是否不活跃，而代码 cache 在程序运行过程中无法知道一个基本块将来是否会被使用。

由于代码 cache 中的大部份基本块都有这样的特性：要么该块长久驻留在 cache 中，要么该块只在 cache 中做短暂的停留。这是因为程序中的热代码一般都会反复执行，从而要长时间停留在 cache 中；而大部分代码都不是热代码，因此执行次数较少，一般执行几次之后就变成垃圾等待回收。图 18 列出了几个测试程序中代码基本块执行次数的分布情况。

从图 18 中可以得知，大部分基本块执行次数较少，用过之后后面将很少会再次用到。因此，要尽量将那些执行次数较多的热代码保留在 cache 中，即用 LRU 替换算法。为了克服 LRU 算法的高开销和高碎片率等缺点，可以将 cache 中的代码基本块按照它们的执行次数，即热度来分类，然后将它们存放在 cache 相对应的区间中，在逻辑上形成一个多层次的代码 cache。最上层存放超级块，中间层存放除超级块之外的其他热代码基本块，它们都采用 FIFO 的方法来进行替换，以便尽可能把它们保留在 cache 中；最下层存放执行次数较少的代码基本块，由于这些代码很少再次被使用，可以采用粗粒度 FIFO 替换算法来进行替换。

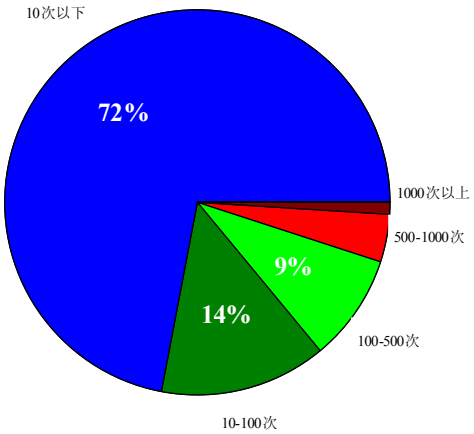

图 18 基本块执行次数分布

Figure 18 Distribution of execution times

4.4 实验分析与评价

采用基于 profile 信息的代码 cache 管理方法的首要目的是减少 cache 的缺失率，一方面尽量把有用的代码基本块保留在 cache 中，另一方面尽量提高代码 cache 的利用率，即在任意时刻都尽量把代码 cache 中的剩余空间填满。

代码 cache 的缺失率是指翻译新的基本块的次数占所有基本块执行次数的比例，其定义如下式所示：

$$missrate = \frac{m}{\sum_i r_i}$$

其中 m 表示缺失所发生的总次数，r_i 表示第 i 个基本块的总的执行次数。

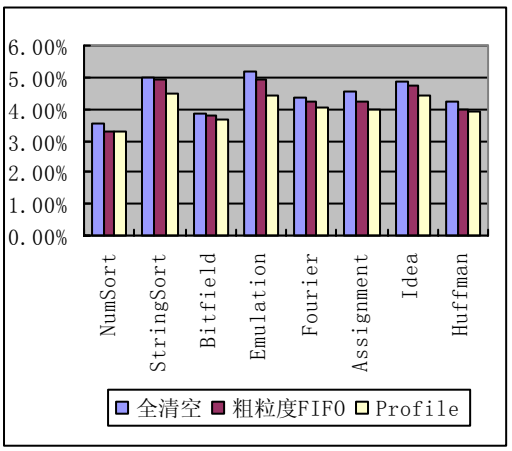

图 19 缺失率

Figure 19 Miss rate

目前大多数动态二进制翻译系统都采用了全清空或粗粒度的 FIFO 方法。下图 19 给出了基于 profile 信息的分层次 cache 管理和这两种替换算法在缺失率方面的对比。由图可知，由于基于 profile 的代码 cache 管理算法具备了较为完备的信息，在预测方面占有优势，因而相比另外两种方法有相对较低的缺失率。而且程序越大，这种差别就越明显，因为程序规模越大，需要发生替换的次数就越多，基于 profile 的替换就越能显示出其威力。

程序整个运行过程中代码 cache 的利用率反映了 cache 使用的效率，同样可以用 cache 的闲置率来反映。代码Cache 的利用率可以用下式所示的方式来定义：

$$usage = \frac{\sum_i u_i \times r_i}{size \times totalrun}$$

其中 u_i 表示本次代码 cache 内容改变之后的已使用部分的大小，r_i 表示本次代码 cache 内容改变到下次改变之间代码基本块的执行次数。Size 表示代码 cache 总的大小，而 totalrun 表示在程序运行过程中所有代码基本块总的执行次数。

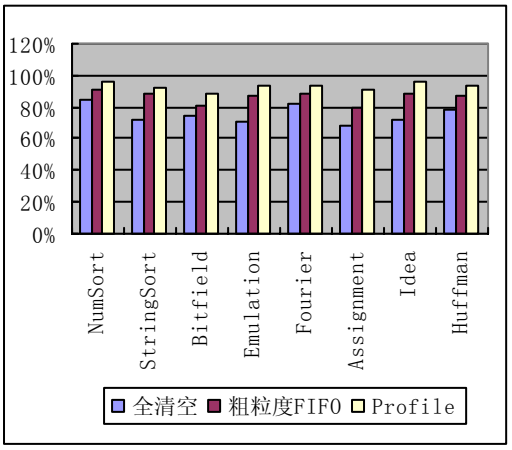

图 20 利用率

Figure 20 Cache usage

由于全清空和粗粒度的 FIFO 替换算法在每次发生清空之后，代码 cache 的空间利用率都会降至最低，而基于 profile 的替换算法在热代码部分没有发生清空，最大限度保持了 cache 的利用率，因此较其他两种方法具有较高的 cache 利用率。如图 20 所示。

由图 20 可知，由于采用基于 Profile 信息进行替换的代码 cache 中任何时刻都有较高的 cache 使用率。使用粗粒度 FIFO 替换的代码 cache 每次都会发生部分清空，使用率会周期性降低，从而平均 cache 使用率较前者低。而采用全清空替换的 cache 由于每次都要将整个代码 cache 清空，它的 cache 使用率最低。

4.5 本章小结

代码 cache 是动态二进制翻译系统中一个重要的组成部分，也是动态二进制翻译与解释执行的重要区别之一。尽管现有的动态二进制翻译系统都使用了代码 cache，而且采用了不同的代码 cache 替换策略，但都是把所有的代码一视同仁，还没有专门对热路径进行区别对待的代码 cache。然而，热路径往往是系统最为关注的对象，需要与其它基本块有所区别，因而本章提出了一种基于 profile 信息的代码 cache 管理。

本章首先介绍了几种传统的代码 cache 替换方法，并进行了一定的对比。然后根据统计结果，针对不同的代码（即热路径代码和普通代码）， 提出了基于 profile 信息的代码 cache 替换方法，按照热代码和普通代码各自的行为特征，采样不同的替换方法区别对待，从而很好地解决了替换准确度和代码 cache 碎片之间的矛盾。

第五章　结论

虚拟计算技术的出现，使得软件的跨平台移植成为了可能，而作为实现虚拟计算技术的一种重要方法，动态二进制翻译是软件移植领域的热点，通过它无需通过重新编译即可将已有的应用无迁移到其它平台，既可挽救遗产代码的应用价值，又能为新研制的处理器提供更丰富的软件资源。动态优化是寻求系统性能提升的新途径，软件的延迟绑定将更多的优化机会留给运行时，体系结构的发展对程序执行行为的挖掘越来越深，软硬件协同优化是未来提高提高处理器性能的趋势之一。动态二进制翻译与动态优化无论在应用还是技术层面都是极有价值的研究课题。

本文着重研究了该领域内两个热点问题，在广泛研究国内外他人研究成果的基础上，根据动态系统特有的规律，进行了一定的改进和创新。由于动态系统最主要的特征在于高效性，因此在研究中只针对重点而不盲目追求全面，强调有效性而不过分追求最优性。

现在已经出现了很多独具特色的二进制翻译器。怎样想方设法降低动态二进制翻译系统在翻译时的开销对一个系统的实际应用来说是至关重要的。热路径的识别和优化是改进系统性能的一个重要方法。对一个动态二进制翻译系统来说，算法的简单性是极为重要的。有些算法可能有很好的预测效果，但由于实现过于繁琐，从而整体上效率低下。我们在实现算法时力求简单，摒弃了复杂的部分。在实现热段算法时，我们只考虑了比较容易实现的那种情况。尽管有多种热路径的识别算法，在算法的实现上也是多种多样，但是每种方法都有它的优点和缺点，在不同的场合有不同的实用价值。本文提出的改进的热路径识别算法，易于实现且具有较高的灵活性。它能够提高命中率，从而提高优化代码的利用率。

动态二进制翻译中的代码 cache 管理是一个非常重要的环节，但是由于受基本块不定长以及基本块的近期活跃情况不可预知等条件的限制，使用传统的 cache 替换算法并不能起到预期的效果，如 LRU，FIFO，最佳大小匹配替换算法，以及用在多种二进制翻译平台上的全清空，分阶段全清空和粗粒度 FIFO 替换算法等。在分析对比已有算法优缺点的基础之上，本文提出了基于 profile 信息的分层次代码 cache 替换算法，并从 cache 的缺失率和利用率方面和全清空替换算法，粗粒度的 FIFO 替换算法进行了对比，证明该算法在这两方面确实能产生较好的效果。

本文只是对动态二进制翻译中基于 profile 信息的优化方法进行了初步的探索和尝试，而该领域内的方法和技术博大精深，进一步深入的研究还有待展开。本文做的 profile 都是比较简单的信息收集，只能片面的反映出程序的执行状况，

如果能收集更为全面的信息，能更加详细地反映出程序运行时的细节，将会带来更多的优化机会。本文在识别出热路径后，只是做了非常有限的优化，对于超级块来说，还有很大的优化空间，如冗余代码的删除，更有效的寄存器分配等，这些优化都可以用来生成更高质量的目标代码。

参考文献

[1] Erik R. Altman, David Kaeli and Yaron Sheffer, "Welcome to the Opportunities of Binary Translation," Computer, Vol 33, No.3 , March 2000, IEEE Computer Society Press, page 40-45

[2] Altman ER., Ebcioglu K., Gschwind M., Sathaye S., "Advances and future challenges in binary translation and optimization", In Proceedings of the IEEE, vol.89, no.11 , Nov.2001, pp.1710-1722, Publisher: IEEE,USA

[3] C. Cifuentes, V. Malhotra, “Binary Translation: Static, Dynamic, Retargetable?”,12th International Conference on Software Maintenance (ICSM'96), p. 340

[4] C. Cifuentes and M. Van Emmerik, "Recovery of Jump Table Case Statements from Binary Code," Proceedings of the International Workshop on Program Comprehension, Pittsburgh, USA, May 1999, IEEE-CS Press, pp 192.199

[5] Cindy Zheng and Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," Computer, Vol 33, No.3 , March 2000, IEEE Computer Society Press, pp 47-52

[6] K Scott, N Kumar, S Velusamy, B Childers, “Retargetable and Reconfigurable Software Dynamic Translation”, International Symposium on Code Generation and Optimization, 2003.

[7] E. Duesterwald, “Design and Engineering of a Dynamic Binary Optimizer”, Proceedings of the IEEE, 2005.

[8] Anton Chernoff, Mark Herdeg, Ray Hook-way, Chris Reeve, Norman Rubin, Tony Tye, S. Bhamdwaj, Yadavalli, and John Yates, “FX!32: a Profile-Directed Binary Translator", IEEE Micro, vol.18 ,no.2, 1998

[9] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100 Percent Architectural Compatibility," Proc.ISCA24, ACM Press, New York,1997,pp .26-37

[10] K. Ebcioglu et al., "Execution-Based Scheduling for VLIW Architectures," Proc. Europar99, Lecture Notes in Computer Science 1685,Springer Verlag, Berlin,1999,pp.1269-1280

[11] Michael Gschwind et at., "Dynamic and Transparent Binary Translation," Computer, Vol 33, No 3, March 2000, IEEE Computer Society Press, pp 54-59

[12] Michael Gschwind, Erik Altman, "Inherently Lower Complexity Architectures using Dynamic Optimization," Proc. Workshop on Complexity Effective Design in conjunction with ISCA-2002, Anchorage, A K, May 2002

[13] Alexander Klaibe, “The Technology behind Crusoe Processor”, Transmeta technology report, Jan. 2000, pp 3-12

[14] Leonid Barez, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach, " IA-32 Execution Layer: a

two-phase dynamic translator designed to support I A-32 applications on Itanium based systems", Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36),IEEE-CS Press

[15] C Cifuentes, M Van Emmerik, “UQBT: adaptable binary translation at low cost”, Computer, Volume 33, pp 60-66

[16] C Cifuentes, B Lewis, D Ung, “Walkabout-a retargetable dynamic binary translation framework”, Fourth Workshop on Binary Translation, Sep 22, 2002,Charlottesville,Virginia

[17] Mark Probst, "Fast machine-adaptable dynamic binary translation," In Proceedings of the Workshop on Binary Translation 2001, September 2001

[18] http://www.transitives.corn/pmssroom.htm#brief

[19] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. "Dynamo: A Transparent Dynamic Optimization System", In Proceedings of the ACM SIGPLAN '2000 conference on Programming language design and implementation, PL DI'2000, June, 2000

[20] Michael D. Smith," Overcoming the Challenges to Feedback-Directed Optimization," Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Dec.2000

[21] Wen-mei W. Hwu, Scot A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warier, Roger A . Bringmann, Roland G. Ouellete, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," Journal of Supercomputing, Kluwer Academic Publishing, pp .229--248,1993

[22] Evelyn Duesterwald and Vasanth Bala, “Software Profiling for Hot Path Prediction: Less is More”. ACM SIGOPS Operating System Review, Volume 34, Issue 5, Dec. 2000.

[23] Mathew Merten, Andrew Trick, Erik M. Nystrom, Ronald D. Barnes, Wen-Mei Hwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots", In Proceedings of International Symposium on Computer Architecture, ISCA-27,2000

[24] A. S. Dhodapkar, J. E. Smith. "Comparing Program Phase Detection Techniques", Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003

[25] K. Scott, N. Kumar, B.R. Childers, J.W. Davidson, and M.L. Soffa, “Overhead Reduction Techniques for Software Dynamic Translation”, Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.

[26] Thomas Ball, Peter Mataga, Mooly Sagiy, “Edge profiling versus path profiling: the slowdown”, Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, page 134-148, 1998.

[27] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen and

Michael D. Smith, "System support for automatic profiling and optimization", In Proceedings of the sixteenth ACM symposium on Operating systems principles,1997,pp.15 –26

[28] Thomas Conte, Burzin Patel, J. Cox. "Using Branch Handling Hardware to Support Profile-Driven Optimization", In Proceedings of the 27th Annual International Symposium on Microarchitecture (Micro-27), 1994

[29] Bala V., Duesterwald E., and Banerjla S., "Transparent dynamic optimization: The design and implementation of Dynamo", Hewlett Packard Laboratories Technical Report, HPL-1999-78. June 1999. Dallas. 173-18

[30] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation", In International Conference on Parallel Architectures and Compilation Techniques, Oct.1999.

[31] M. Cierniuk, G. Lueh, and Stichnoth, "Practicing JUDO: Java Under Dynamic Optimizations," In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, October 2000

[32] Nicholas Nethercote," Dynamic Binary Analysis and Instrumentation ",a dissertation for PhD degree at the University of Cambridge, 2004

[33] www.valgrind.org

[34] Nicholas Nethercote, Julian Seward, " Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", Proceedings of the 2007 PLDI conference, 2007

[35] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. “Binary Translation”, Communications of the ACM, 36(2):69-81, Feb 1993.

[36] Thomas Ball, James R. Larus, “Efficient Path Profiling”, 29th Annual IEEE/ACM International Symposium on Microarchitecture, p. 46, 1996.

[37] D. Bruening, E. Duesterwald and S. Amarasinghe, “Design and Implementation of a Dynamic Optimization Framework for Windows”, in Proceeding of Feedback-Directed and Dynamic Optimization, December 2001

[38] W. Chen, S. Lemer, R. Chaiken and D. Gillies, “M ojo: A Dynamic Optimization System”, In Proceeding of the 3rd Workshop on Feedback-Directed and Dynamic Optimization, December 2001

[39] C. Young and M. D. Smith, "Static correlated branch prediction", ACM Transactions on Programming Languages and Systems, vol. 21 ,pp. 111-159, 1999

[40] Sanjay Patel, S. Lumeta, "Replay: A Hardware Framework for Dynamic Optimization", IEEE Transaction on Computers, vol. 50, no. 6, June 2001

[41] Yuan Chou, John Paul Shen. "Instruction Path Coprocessors", in Proceedings of the 27th annual international symposium on Computer architecture, May 2000

[42] Howard Chen, Wei-Chung Hsu, Jiwei Lu, Pen-Chung Yew,

Dong-Yuan Chen. "Dynamic trace selection using performance monitoring hardware sampling", Proceedings of the international symposium on Code generation and optimization, San Francisco, California, 2003, p79-90

[43] Jennifer M. Anderson, Lance M. Berc, Jefrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger and William E. Weihl. "Continuous profiling: where have all the cycles gone?" ACM Transaction on Computer Systems, vol. 15 , no.4, Nov.1997,pp .35 7-39

[44] Robert S. Cohn, David W. Goodwin, P. Geofrey Lowney, "Optimizing Alpha Executables on Windows NT with Spike", Digital Technical Journal, Vol. 9, No. 4, June 1998

[45] James E. Smith, Rail Nair, “Virtual machine: Versatile Platforms for Systems and Processes”, Morgan Kaufmann Publishers

[46] D. Bruening, T. Garnett, and S. Amarasinghe, “An Infrastructure for Adaptive Dynamic Optimization”, in Proceedings of the International Symposium on Code Generation and Optimization, pp. 265–275, IEEE Computer Society, 2003.

[47] Hazelwood, K.Smith, M.D., “Code cache management schemes for dynamic optimizers”, Interaction between Compilers and Computer Architectures, 2002. Proceedings. Sixth Annual Workshop, 2002.

[48] Robert F. Cmelik, David Keppel, “Shade: A Fast Instruction Set Simulator for Execution Profiling”, Sun Microsystems, Inc. Mountain View, CA, USA, 1994

致谢

首先感谢我的导师管海兵教授。研究方向的选择和本文的撰写始终贯穿着管老师深入细致的指导和及时有效的建议。管老师治学严谨，诲人不倦，不管是学术上还是生活上，都对学生充满了关爱。值此论文完成之际，我在此向管老师表达我深深的敬意！

非常感谢梁阿磊副教授。梁老师专业知识精深，对工作一丝不苟，认真负责，我们的学习进步离不开梁老师的悉心指导，离不开梁老师的关怀呵护。梁老师不仅传授给我们知识和技能，更重要的是教会了我们处世和做学问的方法，这些将使我一生受用不尽。

非常感谢 crossbit 项目组的所有成员，感谢胡坤，郑举育，官孝峰，刘可嘉，林凌，马舒兰，姜玲艳，秦鹏等同学，正是与你们的深入讨论才使我能不断进步。我们一起见证了项目组的不断发展壮大，一起经历了其中的曲折和坎坷。

非常感谢上海交通大学计算机科学与工程系计算机软件与理论专业的同学们，我们同处一个集体当中，感谢你们给我生活和学习上提供的帮助。特别要感谢胡学营同学，谢俊同学，王祎同学，是你们不断给我关怀和鼓励，陪伴我克服了一个又一个困难。

最后，感谢我的父母亲人，感谢你们对我的默默支持，是你们给我提供了强大的精神动力。

攻读硕士期间的科研及学术论文

科研项目

2005 年 9 月—2006 年 12 月 参与国家 973 计划重大基础研究前期研究专项“二进制翻译可重定向研究”的研究。
2006 年 12 月—2007 年 12 月 参与国家 863 计划项目 “网络计算环境下的虚拟执行技术”的研究。

学术论文

1. Huihui Shi, Yi Wang, Haibing Guan, Alei Liang, "An Intermediate Language Level Optimization Framework for Dynamic Binary Translation", ACM SIGPLAN notices, volume 42, issue 5.
2. 史辉辉，管海兵，梁阿磊, "动态二进制翻译中热路径优化的软件实现", 计算机工程，2008 年第 2 期
3. 胡坤，史辉辉，管海兵，梁阿磊，"Crossbit 中的代码 cache 管理"，the National Annual Software and Application Conference (NASAC) 2007