# 基于动态指令编译的软件性能分析方法

程克非[1] 张 聪[2] 张 勤[3] 汪林林[1]
(重庆邮电学院计算机系 重庆 400065)[1] (重庆交通学院计算机系 重庆 400074)[2]
(重庆大学计算机学院 重庆 400044)[3]

**摘 要** 进行精确的软件性能分析,需要在代码中插入测量和控制代码,并根据具体运行状态动态的检查多个不同的参数。但是,用静态类型的程序语言,如C语言等书写的代码,一经编译和链接,其处理逻辑即不可更改。因此,在无法获取源代码或者重新编译和重新启动代价较高的应用中,对软件进行动态性能分析非常困难。本文将介绍一种在软件运行时刻动态插入监控点的动态指令编译技术对软件进行监控,从而在上述情况下达到对软件的监控目的。这种方法是基于DynInst API和PAPI技术的。实验表明,这种方法在去掉了对源代码的依赖的情况下,仍然与在源代码级插入监控点的方法具有同等的采集效率,在很大程度上增强了基于硬件性能计数器方法的软件监控技术的应用范围,达到了较好的性能分析效果。

**关键词** 动态指令编译,性能分析,Dyninst API,PAPI,硬件性能计数器

## Software Performance Analysis Using Dynamic Instruments Compiling Technology

CHENG Ke-Fei[1] ZHANG Cong[2] ZHANG Qin[3] WANG Lin-Lin[1]
(Dep. of Computer, Chongqing University of Posts and Telecommunications, Chongqing 400065)[1]
(Dept. of Computer, Chongqing Jiaotong University, Chongqing 400074)[2]
(School of Computer, Chongqing University, Chongqing 400044)[3]

**Abstract** In wish to do dynamic software performance analysis, many different qualities should be checked according to the running context, and survey and control instructions must insert into code dynamically. But the inner process logic of a program written by static type languages like C is stable and fixed after compiled and linked. So, when source code is not free to get and modify, or when rewriting and restarting a running program is not an option, it is difficult to do software performance analysis. A new dynamic instruments compiling technology is introduced here. It is based on the technology of Dyninst API and PAPI. Using dynamic instruments compiling, performance analysis is independent to source code, and only associates to running status and image of a program. This paper shows how to use dynamic instruments compiling and hardware performance counters to do dynamic software performance analysis.

**Keywords** Dynamic instruments compiling, Hardware performance counter, PAPI, Dyninst API

## 1 引言

软件系统性能分析就目前的现状而言,主要在三个方面有其实际意义:一是嵌入式设备软件系统,主要研究如何以更小的代码空间获得更大的性能;二是高性能的科学计算,主要研究如何让代码以更高的效率和更短的时间运行;第三则是一些商业服务性软件系统,如数据库系统和用户接入系统等,主要研究如何具有更高的数据吞吐量和接入率,以及更为稳定的性能。这三类系统中,除了第一类一般来说可以经常重新编译和启动外,另外两类系统如果经常重新编译和启动都会造成或多或少的时间和经济的损失。传统的软件系统性能分析或者采用修改源代码的方式,加入性能数据采集或分析指令进行性能分析;或者采用对系统进行仿真的方式达到性能分析的目的。

本文将介绍一种在运行程序中插入代码的动态指令编译技术。通过这种技术,我们可以在程序运行过程中创建需要的代码并插入到目标中运行,从而实现对软件系统的性能数据的采集和分析。动态指令编译技术可以用到不同的应用系统中,在调试系统中可以采用这种技术调试没有源代码的程序或者不方便重新编译启动的程序;更为重要的,可以对现有应用系统在不经过重新编译和启动的情况下,进行性能分析和优化。

一般来说,程序在没有运行前,特定的插入代码是未知的,如程序可能在运行后调用某个动态连接库,或者因为某些条件而只执行某些部分的代码。如果无法确定哪些是需要的,哪些是不需要的,则(1)在程序编制中加入所有可能需要的代码,这样保证程序在运行时刻总能满足用户需要,不过这要付出一些额外的代价,包括设计和编制的复杂度的提高[2];同时因为需求的不定性,就算设计非常完善的系统,也可能因为外界环境的变化和用户需求的更改而导致可能需要删除或者添加新的功能。(2)用户可以只启动需求的最小集,然后在新的变化到达时,再加入并重新运行程序;这样可以使程序具

程克非 讲师,在读博士生,主要研究方向为高性能计算应用系统的性能监控、分析与优化,计算机网络应用等;张 聪 讲师,主要研究方向为人工智能与博弈论;张 勤 教授,博士生导师,主要研究方向为故障诊断与人工智能;汪林林 教授,硕士生导师,主要研究方向为数据库理论,数据挖掘,GIS与空间数据库等。

有一定的可变性，当然，这对于一个短暂运行的程序而言，肯定是可行的，但对于商业的 7×24 小时的服务性或者科学仿真运算系统，系统的重启可能导致巨大的经济损失和时间浪费。通过动态指令编译技术，则可以在不影响当前程序运行的情况下，插入需要的新的代码，更改原有的代码和数据结构，或者查看当前的一些运行状态值。

动态指令编译的一个关键就是在程序运行时刻插入和更改指令，这种技术目前主要依赖于动态代码技术 Dyninst API[2~4]。Dyninst API 提供了一个很小的便于使用和理解的 API 接口，并以一种抽象的程序集合以及一些简单的定义插入代码的方法实现。插入的代码可以以动态库的形式或者以动态代码编译形式插入到程序中。另外，程序员可以在自己的用户权限下调用 Dyninst API，从而避免不必要的安全和管理问题。

为了进行精确的性能分析，软件运行的底层性能特性数据，如 Cache 线占用情况，指令运行的时钟周期数是非常必要的。本文采用基于硬件性能计数器的 PAPI 技术采集软件运行时刻的性能数据[5,6]，并将这种技术融合动态指令编译技术 Dyninst API，实现在程序运行时，将 PAPI 插入到程序中采集性能数据进行性能分析。

在现代 CPU 的设计和实现中，一般都加入了一类特殊的寄存器，称为硬件性能计数器(CHPC)，同时 CPU 中存在一条和常规指令执行并行的流水线，以达到对 CPU 指令执行事件的高效记录。对这些事件的存取需要具有一定的操作系统特权，对于普通用户程序而言，它们是不可见的。由美国航天航空局(NASA)和能源部支持的 PAPI 项目[5,6]，针对各种不同的 CPU，封装了对不同 CPU 硬件计数器的低层访问函数集，构成了 PAPI(Performance Application Programming Interface)，使得进行性能分析的研究人员可以不用关心底层存取权限、平台和编程语言的差别而获取硬件性能计数器数据。PAPI 接口中实现了在静态源代码基础上的对 CHPC 的采集。

第 2 节中，我们将对动态指令编译和 Dyninst API 作深入的介绍；第 3 节将建立基于 Dyninst API 的性能数据采集模型；第 4 节是具体的数值实验，最后给出本文的结论和今后研究展望。

## 2 动态指令编译与 Dyninst API

一个程序的正常生命周期从需求开始，一般需要经历设计、编制、编译、调试、运行和维护等阶段。从编制开始，到交付用户运行，都离不开对整个系统代码的依赖。就是说，如果编制完成，在调试过程中出现任何需要修改的地方，则必须在源代码中修改，然后重新经历编译调试和运行过程。动态指令编译指令打破了这种常规的软件生命周期，抛开运行程序目标代码对软件源代码的依赖，实现了指令的运行时刻动态编译和插入，可以在程序运行时期，改变其中一些函数代码的功能，插入一些当前程序没有实现的一些功能，在一定程度上弥补了原有程序生命周期的不足。

Dyninst API 是基于对程序及其运行时刻特征的抽象[2]，通过程序的指令特征和程序编译链接时的符号表，生成进程的程序结构，从而决策能够插入代码的位置。Dyninst API 的两个主要的抽象概念是 point 和 snippet。Point 是程序运行时，可以插入代码的位置；snippet 是插入到程序 point 位置的一段小的可执行代码。如果我们希望查看程序中某一个函数的调用次数情况，则 point 应该是该函数的第一条指令的地址，而 snippet 则可以创建一组指令完成对某个计数器变量的计数。Snippet 可以包含几乎所有的正常指令流，如条件、循环、函数调用等。

Dyninst API 可以插入一段代码到单机的多个进程中，为此，引入了另外两个抽象概念：thread 和 image。Thread 代表一个运行的线程(根据实际系统实现，thread 可以是一个实际的进程，也可以是线程，即轻量级的进程)。Image 是程序的一个静态实例，它含有 point 对象，表达在这个 image 中可以插入代码的位置。

图 1 中 Dyninst API 的总体结构。左边的进程调用 API 完成对右边进程的监控和代码的插入。右边是实际的被监控程序，其实际的代码可插入位置如图 1，从上到下分别为 BPatch_entry，BPatch_subroutine，BPatch_exit，表示函数的入口，子程序调用及函数的出口。

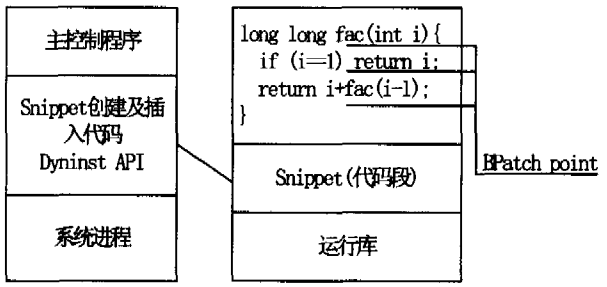



图 1 Dyninst API 控制结构

Dyninst API 实现了一些简单的系统类型，如整型(integer)、字符串(strings)和浮点型(float)，同时也支持数组和系统数据结构，以及对用户自定义结构的修改等。

下面将描述在 API 中的几个主要接口及对象，以及它们之间的相互关系。首先是对运行中的程序的控制类，BPatch 和 BPatch_thread。然后是对程序结构的存取类 BPatch_image，BPatch_module 和 BPatch_function。最后是构建新的代码片段并将其插入到程序的类 BPatch_snippet 和 BPatch_point。下面分别对这些类作简单描述。

BPatch：该类表达整个 Dyninst API 库。在一个应用中，只允许唯一的 BPatch 实例存在。BPatch 可用于函数调用，获取函数信息，以及定义特定事件的回调函数。

BPatch_thread：操作管理或者创建一个运行进程。可以使用 BPatch_thread 的方法实现对线程的启动，暂停或者中止。同时线程类也可用于将代码插入到程序中。对于一个多线程的程序，BPatch_thead 代表其中的某一个线程。API 实现中保证了将代码插入到某个线程并只在该线程中执行，而不会影响到其他线程的运行。对于一个无线程的程序而言，BPatch_thread 则代表进程。

BPatch_image：代表当前运行中的程序的静态镜像。考虑到一个程序在运行时刻，可以因为不同的条件而调用不同的动态库，所以 BPatch_image 仅是 BPatch_thread 的一部分，以保证 BPatch_image 和 BPatch_thread 的一致性。

BPatch_module：是运行程序的一部分。

BPatch_function：运行程序中的函数。

BPatch_point：指出运行程序中可以插入一段代码的位置。它可以是一个符号，如函数入口；也可以是函数中的某种结构入口，如一个循环的开始；或者指明程序中的某一个虚拟的地址，如某一条指令。

BPatch_type：定义系统中使用的数据类型。

BPatch_snippet：代表一段待插入的代码。

下一节我们将建立基于 Dyninst API 的性能数据采集模型。

## 3 Dyninst-PAPI 模型

根据上一节对 Dyninst API 的描述，我们建立图 2 所示的性能数据采集模型 Dyninst-PAPI。

在该模型中，由监控进程初始化 Dyninst API 和 PAPI 库，并负责初始化硬件性能计数器 CHPC，然后由 Dyninst API 分析目标进程的结构，得到监控点 BPatch_point，主要在类型为 ENTRY_POINT 和 EXIT_POINT 的位置，插入必要的 PAPI 性能数据采集接口。也可根据需要在 CALL_POINT 和 RETURN_POINT 处插入必要的代码。之后由插入的 PAPI 采集代码收集性能数据，提交回上层数据整理和性能分析接口，完成性能分析，并最终提交给用户。

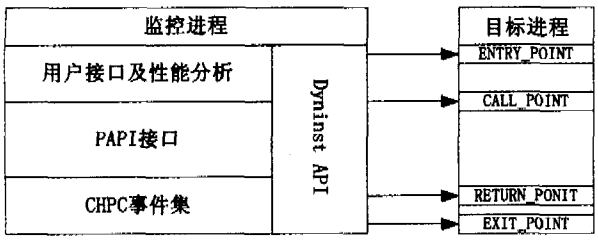



图 2 Dyninst-API 模型

在下一节中，将以一个实例来说明如何使用 Dyninst-PAPI 模型在运行程序中插入监控点，采集性能数据用于性能分析。

## 4 数值实验

本节将以一个简单的递归方式求 1～n 的和的程序为例，说明模型 Dyninst-PAPI 的可行性以及动态指令编译技术如何通过 Dyninst API 应用到程序性能数据和分析中。

本节的实验环境为 Fedora Core 3，Linux-2.6.11，GCC-3.4.2-fc3，381M 内存，其他见下面描述，为程序运行时获得。

处理器：730 Mhz GenuineIntel Intel Pentium III rev 0x1 (1-way)

采集的性能计数器：

PAPI_TOT_CYC：本地指令运行周期数；

PAPI_CA_SHR：本地独占共享 Cache 线请求数。

监控的函数个数：2

```
long long fac(int i){
  long long temp;(A)
  if(i==1)(B)return i;
  temp=(C)fac(i-1);
  sleep(1);
  (D)
  return i+ temp;(E)
  }
```

```
int main(void){
int i=15
(F)
(G)printf("fac(%d)value is:%11u\
n",i,(H)fac(i));
(I)
return 0;
(J)
}
```

图 3 待监控的程序实例 fac

图 3 中是本文中所采用的被监控的主要程序代码，使用 GCC-3.4.2-fc3 编译链接，可以加上-g（调试编译）选项，也可不加，对本实验无影响。对于 PAPI 库，为了便于使用，我们对其做了进一步的封装，以便于采用 Dyninst API 插入到运行程序中。为了放大结果，在函数 fac 中加入了 sleep(1)。

图 3 中标识的(A)，(B)，(C)，(D)，(E)，(F)，(G)，(H)，(I)，(J)各点就是这段代码中的 BPatch_point，一般来说程序中的 BPatch_point 位于一个函数的开始和结束，分别定义为 ENTRY_POINT 和 EXIT_POINT，如图 3 中的(A)和(E)，(F)和(J)。需要注意的是，一般常识中，return 指明函数的返回，EXIT_POINT 位置不可能执行到，但在编译连接后，函数的返回是以汇编指令 ret 说明的，它和 C 语言中的 return 可以不一致，则 EXIT_POINT 所给出的是 ret 指令的位置。而 return 的位置由 RETURN_POINT 指明。另外对函数的调用则用 CALL_POINT 指明，其可能出现的位置如(H)所示。

表 1 采样分析

| 模块 | 样本数量 | 比重 | 调用数 |
|---|---|---|---|
| main | 151000 | 100 | 1 |
| -fac.0 | 119000 | 78.6 | 1 |
| -f80482f4.1 | 25500 | 16.9 | 1 |
| fac | 1430000 | 100 | 15 |
| -fac.0 | 1310000 | 91.3 | 14 |
| -f80482d4.1 | 55600 | 3.88 | 14 |

(a)PAPI_TOT_CYC

| 模块 | 样本数量 | 比重 | 调用数 |
|---|---|---|---|
| main | 4144 | 100 | 1 |
| -fac.0 | 3632 | 87.64 | 1 |
| -f80482f4.1 | 356 | 8.591 | 1 |
| fac | 30300 | 100 | 15 |
| -fac.0 | 26800 | 88.5 | 14 |
| -f80482d4.1 | 2428 | 8.016 | 14 |

(b)PAPI_CA_SHR

在实验中，可以在主控程序中创建和启动目标程序，使用 BPatch_thread::createProcess 实现，也可用 BPatch_thread::attachProcess 建立和运行状态的程序的关联。通过 BPatch_thread::loadLibrary 方法加载和初始化 PAPI 动态链接库；定位 initcode，begincode，endcode，exitcode 等待插入代码段的位置；查找和分析程序中的插入点，根据其插入点属性(BPatch_entry，BPatch_exit，BPatch_return，BPatch_subroutine)，使用 BPatch_thread::insertSnippet 插入相应监控代码段。最后使用 BPatch_thread::continueExecution 开始或恢复程序运行，开始数据采集和分析。本文的监控程序数据表达借鉴了 Dynaprof[7] 的一些表现形式。

实验中设 $i=5$，采用递归方式实现对 1～15 的求和，表 1 给出程序运行完成后的实验结果。在数据采集时同时采集 PAPI_TOT_CYC 和 PAPI_CA_SHR 两个性能参数。表 1(a)中是 PAPI_TOT_CYC 的分析值：在 main 函数中一共的数值为 151000 个时钟周期，其中调用 fac.0(.0 表示该主函数中的第一个调用函数，以下同)119000 时钟周期，f80482f4.1 函数(因该函数名称不定，以其地址代替)为 25500 时钟周期；fac.0 占 78.64%的处理时间。进一步查看 fac 函数中的内部情况，可以看出 fac.0(显然这是对 fac 函数的递归调用)在 fac 函数中占据了 91.27%的处理时钟周期，而另外一个函数 f80482d4.1 只有 3.88%的占用率，剩下的 4.85%为该函数中

的其他一些未知指令占用。

同样的，考察表1(b)PAPI_CA_SHR的情况，main函数中fac.0的共享Cache线独占申请数占87.64%，fac函数本身中fac.0(递归调用)占88.5%。这样，我们有理由相信，在本实验例子中，fac函数是该程序的主要性能瓶颈，如果需要对程序优化，则主要应该考虑对fac的递归的优化。

**结论与展望** 本文以一个简单的实例介绍了动态指令编译技术，并结合PAPI技术，建立了Dyninst-PAPI模型，实现了Linux平台中对运行时程序的性能数据采集和分析。通过实验证明，这种技术可以比较有效地对无源代码，或者不能随意重新编译、重新启动的程序进行性能数据采集和分析。所不足的是，目前的实现是在字符控制台中，通过对本地进程的接管和结构分析(路径和关键点的选择还主要依赖分析人员对程序的熟悉程度)，插入监控代码并采集和分析，使得Dyninst-PAPI模型的应用存在较大局限性。今后的研究中，将考虑实现对远程进程的监控以及良好的用户使用接口。如何能够在得到进程的运行镜像后，分析进程的关键路径，自动决策插入监控点位置是一个很有意义和挑战性的研究。在文[8]中介绍了通过分析并发进程间的循环层次结构得到程序的关键路径的方法，但是它的分析主要基于对静态程序结构和数据的分析；同时，通过程序结构分析，得出图形化的程序关键路径表达，提供给普通程序开发和维护人员使用，也是一个很有意义的研究工作。

## 参考文献


1 Zhang Charles, Jacobsen HansArno. TinyC2: Towards building a dynamic weaving aspect language for C[C]. Foundation of Aspect Oriented Languages Workshop in conjunction with 2nd AOSD Conference 2003, Boston, MA

2 Buck B, Hollingsworth J K. An API for Runtime Code Patching [J]. Jour. of High Performance Computing Applications, Winter 2000, 14(4): 317～329

3 Hollingsworth J K, Miller B P, Cargille J. Dynamic Program Instrumentation for Scalable Performance Tools [C]. 1994 Scalable High-Performance Computing Conf., Knoxville, Tenn. 1994. 841～850

4 Mucci P. DynaProf and PAPI: An Object Code Instrumentation System for Dynamic Profiling [C]. Linux Supercluster Users Conference, 2000

5 ICL of University of Tennessee, PAPI Programmer's Reference [EB/OL], http://icl.cs.utk.edu/

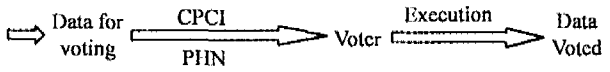
6 Browne S, Dongarra J, Garner N, London K, Mucci P. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters [C]. In: Proceedings of the IEEE/ACM SC2000 Conference November 04 - 10, 2000 Dallas, Texas

7 Mucci P. PAPI and Dynamic Performance Analysis Technology [R]. report at university of Tennessee, Feb. 2003

8 Dey S, Bommu S. Performance analysis of a system of communicating system [C]. C&C Research Laboratgories, IEEE, 1997


(上接第287页)

$+T_3$，表决流程如下：




可以对本结点机的表决时间做一个基本的估计：

$T_1$ 的时间视表决数据的大小而定，一般为μs级；

CPCI的速度为132MB/S或者512MB/S，PHN的速度为1000Mbps且无阻塞，通过这两个路径几百字节的数据到达表决器的总的时间 $T_2$ 为μs级；

表决执行时间 $T_3$ 与表决算法及同步有关，也为μs级；

所以 $T_v$ 为几十μs到几百μs之间。

由于表决执行与对请求的处理分别在两个处理器上并行执行，因此真正的表决时间可以更小。

**进一步工作及结论** ICFTC已经完成研制，在作最后的验证测试。本容错计算机建立在COTS硬件和软件基础上。通过设计一种智能容错管理模块，解决了COTS硬件的可观察性问题，进而实现了将应用任务与容错任务并行处理。在提高容错计算机故障覆盖率方面，ICFTC采用4层故障/错误处理机制，通过对该机制的故障逃逸模型的分析，可以看出ICFTC以较小的设计复杂性和设计开销获得传统定制容错计算机能达到的故障覆盖率。由于容错机制的相对独立性实现，ICFTC结构灵活，具有较好的可扩展性，能满足不同的系统容错要求。

## 参考文献


1 Yuan Youguang. The Reliability Techniques in Real-Time Systems (in Chinese), Beijing: Tsinghua University Press, Sep. 1995

2 Siewiorek D P, Swarz R S. The Theory and Practice of Reliable System Design: Digital Press, USA, 1982

3 Alkalai L, Tai A T. Long-Life Deep-Space Applications. IEEE Computer, 1998, 31: 37～38

4 Tai A T, et al. COTS-Based Fault Tolerance in Deep Space: Qualitative and Quantitative Analyses of A Bus Network Architecture, In: Proc. of the 4th IEEE Intl. Symposium on High Assurance System Engineering, Nov. 1999. 97～104

5 Tai A T, et al. On-Board Maintenance for Long-Life Systems. In: Proc. of the IEEE Workshop on Application-specific Software Engineering and Technology (ASSET'98). Apr. 1998. 69～74

6 Powell D. A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems, London: Kluwer Academic Publishers, 2001

7 Arlat J. Preliminary Definition of the GUARDS validation strategy: [ESPRIT Project 20716 GUARDS Report]. LAAS_CNRS, FRANCE, Jan. 1997

8 Avresky D R. Dependable Network Computing. Kluwer Academic Publishers, USA, 2000. 3～19

9 Shanley T, et al. PCI System Architecture (fourth edition). Adison Wesley Longman, inc, USA, 1999

10 Ou Zhonghong, et al. A Kind of Generic Real-time Dependable Server Architecture With Low Fault-latency Using COTS Components. In: the Proc. of DCABES'2004, Wuhan, Sept. 2004. 103～109