学号：__2016300030028__

密级：_____

武汉大学本科毕业论文

面向 Intel PT 追踪数据的并行解码器设计与实现

院(系)名 称 ： 弘毅学堂

专 业 名 称 ： 计算机科学与技术

学 生 姓 名 ： 吉凯

指 导 教 师 ： 何发智 教授 左志强 助理教授

二〇二〇年五月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名: _____ 日期: _____

摘　　要

基于硬件机制的嗅探技术借助处理器已有的硬件模块，实现在底层对程序执行信息的收集与跟踪，对源程序无侵入，引入较低的额外开销。英特尔处理器追踪 (Intel PT) 是一种新的在英特尔处理器上支持硬件嗅探技术的硬件机制, 它追踪程序在中央处理器上的分支执行，通过对追踪数据的解码能够重构程序执行的整个控制流。不过，虽然 Intel PT 减少了程序执行时的额外开销，但追踪过程中产生的大量数据使得解码的效率通常比追踪的效率低几个数量级，实际上将开销转移到了解码阶段。

为了提高解码过程的效率，降低解码阶段的开销，本课题在利用 Intel PT 实现对 C/C++ 程序的嗅探时，完成了对 Intel PT 追踪数据解码的并行化设计与实现。本课题主要完成了三个方面的内容：追踪，配置 Intel PT 在程序执行的过程中记录并导出不同处理器执行下的 Intel PT 数据流；解码，实现 Intel PT 数据包、事件、指令三个层次的解码，利用数据包层解码器切分数据流，利用指令级解码器并行化解码切分后的 Intel PT 的追踪数据流，重构机器指令的执行；映射，将机器指令映射到源代码，实现对源程序的嗅探。

实验中追踪和解码了 SIR 的 10 个 C/C++ 程序，评价了追踪的开销和解码的效率，在 Linux 5.3 内核，ubuntu18.04 操作系统，Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 处理器的环境下，追踪阶段产生的额外开销小于 5%，解码阶段减少了非并行化解码器 60%-70% 的解码时间。

关键词：英特尔处理器追踪；嗅探；并行化算法

ABSTRACT

Hardware-based profiling takes advantage of existent features of Center Processing Unit(CPU), to achieve tracing and information collecting during the execution of programs in the hardware level. It is non-intrusive to source codes and introduces little cost. Intel Processor Trace is a new hardware feature in Intel processors which can help profiling. It can reconstruct the whole execution flow of programs by decoding the branch execution collected during running time. Though it reduces the extra cost during the execution of programs, great number of data collecting during tracing period results the efficiency of decoding to be several orders of magnitude lower than the efficiency of tracing. It actually shifts the cost to decoding period.

To improve the performance of decoding period and reduce the cost of decoding, this project designs to implement a parallel decoder based on Intel PT while using it to implement a profiling tool for C/C++ programs. Three parts are mainly achieved in this project: First, trace. Record with Intel PT after being configured correctly and dump the Intel PT trace data stream. Second, decode. Implement PT packet-level, event-level and instruction-level decoder. Use packet-level decoder to split the trace stream and instruction-level decoder to parallel decode the trace stream and reconstruct the execution of machine codes. Second, map. Map the execution of machine codes to source codes, so we can profile the source codes.

In the experiment, We traced and decoded 10 C/C++ programs in SIR, then evaluated the overhead in tracing period and efficiency in decoding period. In Linux 5.3, ubuntu18.03, processor of Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, the overhead was lower than 5% in tracing period, and the the decoding time reduced 60% to 70% compared to non-parallel decoding.

Key words: Intel Processor Trace; Profiling; Parallel Algorithms

目　　录

1 绪论

1.1 研究背景及意义

嗅探技术 (Profiling) 是一种被广泛应用的技术，它通过获取程序的运行时信息，能够分析程序的运行过程，并被应用到程序开发的各个阶段,包括调试、测试、问题修复、性能分析、日志等。通过 Profiling，开发人员可以了解到程序运行过程中在哪些部分花费更多的时间，程序运行过程的函数调用图等信息，利用程序运行时得到的 Profiling 信息，一方面，通过分析可以找到比预期执行慢的程序，从而进行优化并使其获得更快的执行效率，另一方面，通过分析出函数调用次数与预期的比例，可以发现程序的潜在错误。[1]

通常来讲，可以将 Profiling 分为两大类：基于插桩 (Instrumentation) 实现的软件 Profiling 以及借助底层实现的硬件 Profiling。软件 Profiling 在软件层面通过程序插桩技术，在原程序的基本块中注入额外的用于收集执行信息的输出代码，将源程序重新编译后从而在程序运行时获取程序执行过程中的控制流、数据流等信息，用以分析程序的执行过程。[2]

gprof 和 gcov 是 GNU 的 Profiling 工具实现，gprof 能够较为精确的提供函数级别的 Profiling 信息，评估程序在每一部分的执行时间，然而这类信息较为有限，gcov 则通过编译前的插桩，能提供程序的行级别的 Profiling 信息，分析每行代码的执行频率，获得测试代码覆盖率等信息。[3][4] 然而基于软件方式的 Profiling 通常是侵入式的，额外代码的引入一方面可能会影响程序的正常执行，另一方面要获取精准信息必然会对运行时程序带来较大的额外开销 (插桩后的程序执行时间可能会比原来要慢 50% 以上)，这给对实际应用程序的 Profiling 造成了极大的困难。[5] 于此相比，基于硬件机制的 Profiling 可以借助已有的 CPU 硬件相应模块，实现在程序运行底层对程序执行信息的收集与跟踪，硬件级别的 Profiling 是非侵入式的，引入的额外开销较低，具有较高的研究价值。

数字连续 Profiling 基础架构是一种基于采样的 Profiling 系统，它通过对处理器提供的程序计数器的采样进行数据收集和分析，支持对生产应用程序的连续 Profiling，基于对采样信息的分析，它能够确定指令停顿的原因，例如高速缓存未命中、资源争用、分支错误预测等，提供细粒度的指令级 Profiling，指导用户和自动优化器找出性能问题的潜在原因，并提供解决问题的可能思路。[4]

最后分支记录 (Last Branch Record, LBR) 和分支追踪存储 (Branch Trace Store, BTS) 是 Intel 处理器提供的硬件 Profiling 机制，LBR 能够记录最后若干个分支记录到一组特定于模型的寄存器 (MSR) 中, 利用 LBR 可以对分支执行进行采样，每次采样能够获取到最后 8 至 32 个分支记录信息，然而由于 LBR 只能记录最后的若干个分支，能够获取的 Profiling 信息十分有限，BTS 能提供更多分支信息的记录，它将分支信息保存在缓冲区中，并在缓冲区满时产生中断信号导出到磁盘中，然而 BTS 的引入会给程序的执行造成巨大的额外开销，同时对每个分支记录的 24 字节信息使得 BTS 的记录信息十分庞大，这使得它并没有被广泛使用。[6]

Intel 处理器追踪 (Intel Processor Trace, Intel PT) 是一种在 Intel 处理器上支持硬件 Profiling 的新硬件机制，从第五代核处理器 BROADWELL 开始，Intel 处理器提供了对于 Intel PT 的支持，并在接下来的处理器中提供了普遍的支持。与 BTS 相比，Intel PT 支持以更低的额外开销追踪，并且能够收集时间等额外信息，同时对于不同的分支，Intel PT 对于追踪数据的生成采取了各种不同形式的压缩方式，生成更简洁的记录信息。[7][8]

Intel PT 通过追踪 CPU 上的分支执行，能够重构代码执行的整个控制流。通过 Intel PT 实现硬件 Profiling 主要包含两个过程：（1）Intel PT 追踪程序运行时分支跳转信息，产生一系列的数据包并压缩到二进制数据流中；（2）解码器以可执行文件和收集得到的压缩数据流为输入，进行解码从而重新构建程序执行时完整的控制流。[9]

基于 Intel PT 的硬件 Profiling 降低了对源程序执行产生的影响，减少了追踪过程产生的开销，但实际上将开销转移到了解码阶段。追踪过程中产生较多的追踪信息 (每个核每秒成百兆字节) 通常使得解码阶段的开销比追踪阶段慢几个数量级。提高 Intel PT 的解码效率，减少 Intel PT 解码阶段的开销，对于基于 Intel PT 的硬件 Profiling 具有重要的意义。

1.2 研究现状

Intel 在引入 Intel PT 的新特性后受到了广泛的关注，它通过在硬件层面收集的大量的控制流等程序运行时信息，通过离线的解码重新构建完整的程序执行信息，在调试、测试、程序修复、性能优化等问题上都可能会有着重要的用途。不过，由于 Intel 提供 Intel PT 硬件机制时间不长，现在关于 Intel PT 的很多研究和使

用仍然处在初始阶段。

为解决生产过程中出现的故障在测试中难以重现的问题，Failure Sketching 利用 Intel PT 的控制流追踪，结合程序静态分析和程序动态分析策略，实现了一个能够自动化提供生产过程错误产生的根本原因的调试工具，该工具能够以较小的开销给出导致程序故障根本原因的语句，显示程序成功和失败运行的区别。[10]

在 GRIFFIN 中，作者发现利用具有类似于 Intel PT 能够重新构建程序执行流的特性的硬件辅助控制流完整性 (Control Flow Integrity, CFI) 实施系统具有较高的效率和较好的灵活性，并基于 Intel PT 设计了一种并行化的方法实施各种类型 CFI 策略，在 Linux 4.2 内核中 GRIFFIN 能够在包括火狐浏览器及其即时编译的代码等多种软件得到完整的 CFI 策略实施，比基于软件的插桩具有更低的额外开销。[11]

将 Intel PT 的追踪信息应用于硬件延迟的有效探查、追踪，和系统调用的准确分析也得到了比较好的结果，相比于基于插桩的 Profiling 方式，它对程序性能的影响大大降低，平均只有 2% 至 3%，同时能够获得更为精确的分析信息。[12]

REPT 是利用 Intel PT 实现了一个能够对于部署系统的软件故障进行反向调试的系统，通过对 Intel PT 追踪数据的解码与对二进制文件的静态分析，REPT 能够准确地重建程序的执行历史，集成到 Windows 的调试工具并对 16 个漏洞进行评估，实验表明，它能够有效和精确的还原这些漏洞的数据值。[13]

这些工具的实现与应用基于 Intel PT，因此都离不开对于 Intel PT 追踪数据的解码，Intel PT 在追踪过程中将信息以压缩数据包的形式收集到二进制流中，只有通过对 PT 数据的解码才能够获取我们所需要的程序执行时信息。目前提供对 Intel PT 数据进行解码的工具主要有：LIBIPT(Intel PT Decoder Library) 是 Intel 对于 Intel PT 解码器的官方参考实现，它可以作为一个独立的库被调用或者部分或全部的集成到其他工具中，LIBIPT 提供了解码过程中不同层次的抽象，通过调用 LIBIPT 提供的函数接口，可以实现对 Intel PT 追踪数据的解码。[9] [14] SimplePT 是 Andi Kleen 实现的 Linux 内核模块，支持在 Linux 系统上收集和导出 Intel PT 的信息 (不支持 PT 数据的连续导出，只能保存程序结束前的部分 PT 数据到磁盘)，解码器基于 LIBIPT 实现，解码结果仅包括函数级别的信息。[15] Perf 是 Linux 系统自带的性能分析工具，从内核 4.1 版本开始，可以通过 perf_event 系统调用接口配置 Intel PT 收集信息，从内核 4.3 版本开始，perf 添加了对 Intel PT 事件的支持，且支持持续 PT 数据的导出，perf 工具的解码结果提供了指令级 Profiling 信息，它包括程序

执行时每个分支的方向。[16] 不过，现有工具对于 Intel PT 的解码过程均是顺序执行的，因此解码会花费较多的时间，为减小 Intel PT 解码过程开销，有必要实现一个并行化的 Intel PT 解码器。

1.3 研究内容

本课题主要研究内容为在利用 Intel PT 进行硬件层面的 Profiling 时，减少解码阶段的开销，主要考虑通过解码阶段的并行化来进行实现。本课题利用 Intel PT 实现了一个简单的 C/C++ 程序 Profiling 的工具，能够提供源程序函数、行两个级别的 Profiling 信息，包括源程序每个函数被调用的次数，每一行被执行的次数，主要工作包括三个方面的内容：1. 在 Linux 5.3 内核下，利用 Linux 提供的系统调用 perf event 内核接口配置 Intel PT 进行 PT 数据的捕获，并能够连续向磁盘中导出捕获的数据；2. 基于 1 中捕获的数据和程序可执行文件进行解码，重构原始程序在机器指令级别的完整执行流，在解码的过程中实现并行化，对 Intel PT 追踪数据进行划分，对划分后的数据进行并行的解码，从而减少解码阶段的开销，提高解码的效率；3. 利用源程序编译 (编译器为 gcc/g++，编译选项需添加-g 参数) 阶段的调试信息实现解码后的机器指令到 C/C++ 源程序的映射，从而得到程序执行过程中在源程序行、函数两个级别的 Profiling 信息。

对 Intel PT 的解码阶段进行并行化设计主要思路是首先借用 Intel 官方解码参考实现 LIBIPT 的思想，并在在其基础上进行改进，对 Intel PT 追踪数据的解码进行不同层次的封装，能在数据包层、事件层、指令层三个不同的层次对收集的 Intel PT 数据进行解码；然后，通过 Intel PT 数据包层的解码器对 Intel PT 追踪数据进行较为平均的划分，对划分的每一段 Intel PT 数据分别分配不同的指令层解码器进行独立的解码，实现解码阶段的并行化，同时在从机器指令到源程序映射的过程中，采用同样的并行策略，对每一段 Intel PT 数据解码得到的机器指令进行并行化的到源程序的映射。实验中追踪和解码了 SIR(Software-Artifact Infrastructure Repository) 中的 11 个 C/C++ 程序，分析评估了追踪过程的额外开销，比较了并行化解码器与普通解码器的效率，同时在最后给出了本课题实现的 C/C++ 程序 Profiling 工具在源码上的输出结果。

1.4 论文组织结构

本论文在第一章概述了 Profiling 相关的背景知识和利用 Intel PT 进行硬件 Profiling 的优势，简介了 Intel PT 的研究现状和对追踪数据并行化解码的意义，并给出了课题的主要研究点和论文组织结构。

第二章简要介绍了 Intel PT 相关的知识，给出了利用 Intel PT 进行硬件 Profiling 的流程和常用的几种 PT 数据包形式与作用以及利用 Intel PT 进行程序追踪的几种方法 (具体介绍了本课题使用的 perf event 系统调用)。

第三章是基于 Intel PT 并行化解码器的设计方案和算法实现，首先具体介绍了数据包、事件、指令层三个级别解码器的具体功能和调用方法，然后给出了利用数据包、指令两层解码器进行并行化解码的方案和算法实现。

第四章是实验部分，追踪和解码了 SIR 提供的 10 个 C/C++ 程序，分析了追踪过程开销和解码过程效率，最后介绍了利用编译阶段生成的调试信息实现机器指令到源程序映射的方法，给出了实现的 Profiling 工具的最终结果。

第五章给出了本文的总结与展望。

2　Intel PT 背景知识

2.1　Intel PT 简介

英特尔处理器追踪 (Intel Processor Trace，Intel PT) 是 Intel 处理器结构的扩展，是一项最新的硬件功能，它使用专用的硬件设施捕获程序的运行时信息，只造成极小的额外开销。正确配置后，Intel PT 追踪程序在 CPU 上的执行，收集控制流转移等相关信息，并对收集信息进行编码产生追踪数据。将记录的 Intel PT 追踪数据与程序二进制文件作为输入，软件解码器可以重新构建程序执行的精确控制流，图 2.1给出了利用 Intel PT 进行硬件 Profiling 的简要流程。

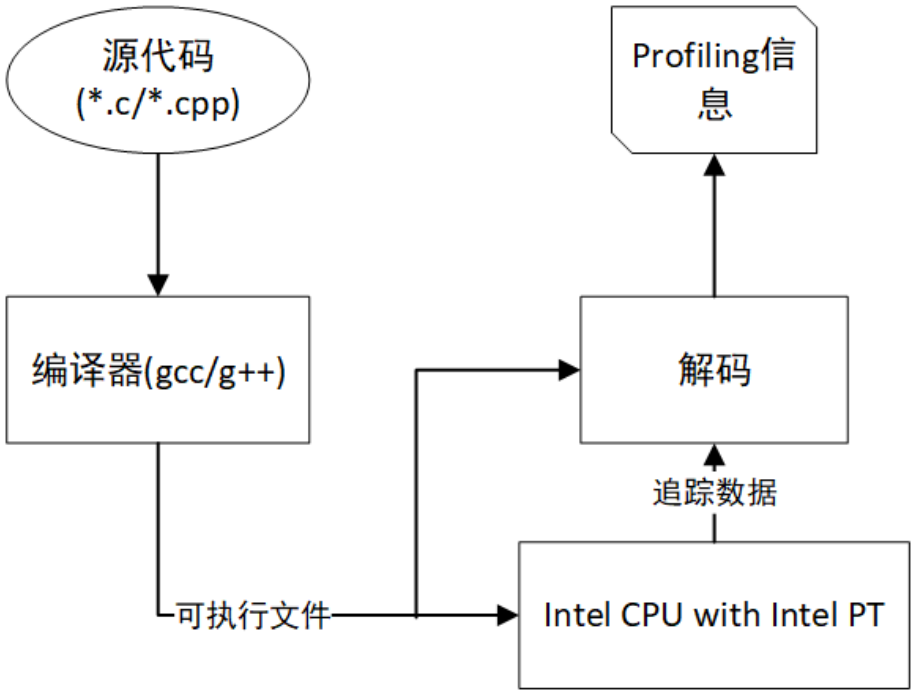

图 2.1　Intel PT **硬件** Profiling **流程**

Intel PT 追踪数据由一系列的数据包组成，数据包能够记录控制流信息，例如间接分支目标的指令指针（IP）地址和条件分支方向 (分支是否发生)，除了控制流信息，数据包还记录了如上下文、时间戳等其他信息，从而可以对应用程序进行功能和性能调试。Intel PT 具有多种形式的控制和过滤功能，可通过对特定的 MSR 寄存器编程进行配置，正确配置后能用于自定义收集的跟踪信息，并附加其他处理器状态和时序信息以启用调试。例如，有些模式允许基于当前特权级别（CPL）或 CR3 的值，以及指令指针地址的值对数据包进行过滤。[9] 为了解 Intel PT 如何

启用控制流跟踪，表2.1 给出了主要的 Intel PT 生成数据包的类型、大小和功能：

表 2.1　Intel PT 数据包

数据包	大小 (字节)	功能
PSB	16	Packet Stream Boundary, 定期生成，充当数据包所构成的二进制流的边界，为解码器提供了一个同步点。
PGE	≤8	Packet Generation Enable, 标志着一段控制流追踪的开始，提供开始的指令指针地址
PGD	≤8	Packet Generation Disable, 标志着一段控制流追踪的结束，提供结束时指令指针地址
TNT	1 or 8	Taken-Not-Taken, 提供直接分支跳转的方向 (发生或者未发生)
TIP	≤8	Target Instruction Pointer, 提供间接分支跳转、异常、中断等的目标指令指针地址
FUP	≤8	Flow Update Packet, 提供无法异常、中断等无法推导源指令指针地址的异步事件的源指令指针地址
TSC	8	Timestamp Counter, 固定产生时钟周期计数值，和其他与时间相关的数据包一起能提供较准确的时间信息

Intel PT 定期产生 PSB 数据包，标志着每段数据流的边界，为解码器提供了一个同步点，在解码时 PSB 数据包应该是解码器应该找的第一个数据包。Intel PT 通过 PGE 和 PGD 数据包记录一段控制流追踪的开始和结束，在配置 Intel PT 时进行追踪时，可能添加了类似于当前用户权限 (CPL)，CR3 寄存器，指令指针地址等过滤条件，当满足追踪条件时，会产生 PGE 数据包标志着一段控制流追踪的开始，当不满足时，则通过 PGD 标志一段控制流追踪的结束，但控制流追踪结束的情况，PT 仍然有可能会产生时间、模式等相关的数据包。

为了压缩产生的信息，最小化生成数据包的大小，Intel PT 对不同的分支指令采用了不同的方式记录，对于 JMP (E9 xx, EB xx), CALL (E8 xx) 等无条件跳转指令，它们的目标地址嵌入在指令中，目标地址可以直接从二进制文件中推导，Intel PT 忽略了此类跳转信息的记录，对于条件分支指令 (JCC, J*CXZ) 和 LOOP 指令，Intel PT 使用 TNT 数据包中的一个比特记录了分支的方向 (发生/未发生)，1 个字节的 TNT 数据包最多能记录 6 个条件分支，8 个字节的 TNT 数据包最多能记录 47 个条件分支，对于类似于 JMP (FF /4), CALL (FF /2) 等间接分支指令和 RET (C3, C2 xx) 等返回指令，Intel PT 用 TIP 数据包记录了目标指令指针地址，为了减小 TIP 数据包的大小，如果高位地址字节与先前记录的地址匹配，则英特尔 PT 压缩目标地

址，最多可压缩六个字节，此外，对于 RET 指令，若目标地址与先前的调用匹配，Intel PT 将其压缩为 TNT 的一位。对于中断、异常等异步事件指令，Intel PT 除了用 TIP 数据包记录了目标指令指针地址，还用 FUP 数据包记录了发生此事件的源指令指针地址。Intel PT 的数据包压缩机制会导致数据包之间存在依赖关系，这对并行化的处理提出了挑战。

Intel PT 的输出机制独立于追踪和过滤机制，输出选项可能随处理器和平台的不同而变化。Intel PT 将追踪数据包直接输出到物理内存中，绕过缓存和 TLB 以减少对程序性能造成的影响，目前内存配置有三种方式：1. 物理地址空间的单一连续区域；2. 可以通过使用类似于表的数据结构来配置使用物理地址空间中不连续的多个缓冲区，当缓冲区已满，可以触发中断；3. Intel PT 支持一种特定于平台的子系统输出机制。

Intel PT 支持所选进程/线程的用户空间和内核空间的追踪，通过相应的配置后，能够仅追踪特定的进程/线程在用户空间/内核空间的执行，在本课题中实现 C/C++ 程序的 Profiling 时，我们更关注对某一进程的用户空间执行的追踪。

2.2 Intel PT 追踪配置

Intel PT 追踪数据包的生成通过特定于模型的寄存器（MSR）集合来启用和配置，目前在 Linux 系统上提供 Intel PT 追踪实现获取数据的工具主要有：

1. simple-PT 是 Intel PT 在 Linux 系统上的简单实现，它支持使用备用的内核驱动程序在 Linux 上捕获 Intel PT，使得 Intel PT 能以适度的开销追踪 CPU 在硬件级别执行的控制流。该项目包括一个配置 Intel PT 的内核驱动模块，一个导出 PT 数据的模块，一个能显示函数、指令级追踪的解码模块、一个仅显示 PT 数据包的快速解码模块。不过 simple-PT 不支持中断，不能够长期导出 PT 数据，当追踪生成的 PT 数据超过为其所配置的内存时，将会丢失。[15]

2. 从 4.1 版开始，Linux 内核通过 perf event 内核接口支持对 Intel PT 的配置追踪。从 4.3 版开始，用户空间 Profiling 工具 perf 也支持 Intel PT，perf 通过调用 perf event 内核接口实现对 Intel PT 的配置，支持两种模式下的追踪：Full Trace 模式下，perf 会连续向磁盘中导出追踪过程中的全部 PT 数据，Snapshot 模式下，生成的 PT 数据会不断向内存中重写，直到停止追踪，并只导出追踪结束前的数据，这种模式对软件的调试有重要的帮助。[17]

3. 除此以外，Intel 的 Profiling 工具 VTune[18] 提供了对 Intel PT 的支持，它具有较为强大的功能支持；gdb[19] 从 7.10 开始的版本通过利用 perf event 内核接口也提供了对 Intel PT 的支持，很好地实现了进程的记录和重放，在断点或故障时，能够很容易的看到之前的指令流，在利用 gdb 进行调试时有很好的效果。

为了使追踪模块更简洁且易于控制，本课题主要利用了 perf event 内核接口实现了简单的 Intel PT 的追踪配置和数据导出。

下面具体介绍了 perf event 内核接口的配置方式：perf event open 内核接口提供了在 Linux 用户态建立性能监控的途径，指定参数后，系统调用 perf_event_open 返回一个文件描述符，每一个文件描述符对应一项被监控的系统事件，多个文件描述的组合可用来监控多种事件，此文件描述符可以被以后的系统调用 (ioctl, prctl, read, mmap 等) 所使用。ioctl、prctl 系统调用控制事件的启用或禁用，事件分为两种形式，计数事件通过系统调用 read 访问，采样事件通过系统调用 mmap 配置采样结果在内存的写入和访问。参数列表中参数 pid 和 cpu 指定了监控某个进程/线程和 cpu 的事件，group_fd 允许创建一个事件组，首先创建的事件该参数为-1，该组的其他事件指定 group_fd 为首先创建事件的返回值。参数 flags 指定了该事件的部分标识，包括是否支持系统级监控、是否忽略 group_fd 等。参数 attr 具体配置了事件的相关信息，包括事件类型、该类型事件的具体配置、是否监控用户空间、是否监控内核空间等。

为了配置 Intel PT 追踪进程在用户空间的执行，我们主要需要进行两个方面的配置：

1.Intel PT 事件，通过读取 Linux 下的/sys/bus/event_source/devices/intel_pt/type 文件可以获取 Intel PT 的动态 PMU 值，根据需要，具体配置 PT 事件相关的 attr 参数信息，然后对所期望追踪的进程和每一个 CPU 进行监控，通过 mmap 系统调用设置每一个 CPU 产生 PT 追踪数据的内存，通过 ioctl 系统调用控制事件的启动和禁用。

2. 为了对 Intel PT 进行解码，除了需要配置 Intel PT 相关的事件，我们还需要实现对其他信息的记录，例如二进制文件的加载 (需要将某个时刻的 PT 记录信息与其记录执行的二进制文件对应)，线程切换信息 (为了得到当前执行的线程信息，获得对多线程的支持) 等。首先配置监控事件的类型为 PERF_TYPE_SOFTWARE，具体配置所需相关事件的 attr 相关信息，同样需要监控对预期追踪的进程以及每

一个 CPU 进行监控，利用 mmap 系统调用设置该事件产生结果写入的内存，利用 ioctl 控制事件的启动和禁用。

3　并行化解码器设计与实现

相应的配置后，Intel PT 能够追踪程序在 CPU 上的执行，并会产生一系列的追踪数据；在程序执行完成后，通过对追踪数据和可执行二进制程序的解码能够重新构建程序的完整执行过程。图 3.1简要给出了利用 Intel PT 进行硬件 Profiling 中解码阶段的主要流程。

关注到解码阶段产生的巨大开销，为降低解码阶段的开销，在此章中给出了解码阶段并行化的实现方案。为了对 Intel PT 追踪数据进行并行化解码，首先借用 Intel 官方解码参考实现 LIBIPT[14] 的思想，并在其基础上做了部分改动，实现了数据包、事件、指令三个层次的解码抽象，每个层次的解码器可用于完成特定的解码要求，数据包层的解码器实现了对 Intel PT 追踪数据纯数据包导出，通过对 Intel PT 追踪数据的扫描，迭代解码追踪产生的每一个数据包；事件层解码器能够通过对不同 PT 数据包组成的事件进行处理，完成了对事件的封装，并能够通过询问的方式询问下一个控制流转移语句的方向；指令层解码器是对事件层解码器的进一步封装，它增加了对可执行二进制文件的处理，通过设置当前 Intel PT 追踪数据所对应的上下文的内存映像后，指令层解码器能够对程序执行的指令进行逐条还原。

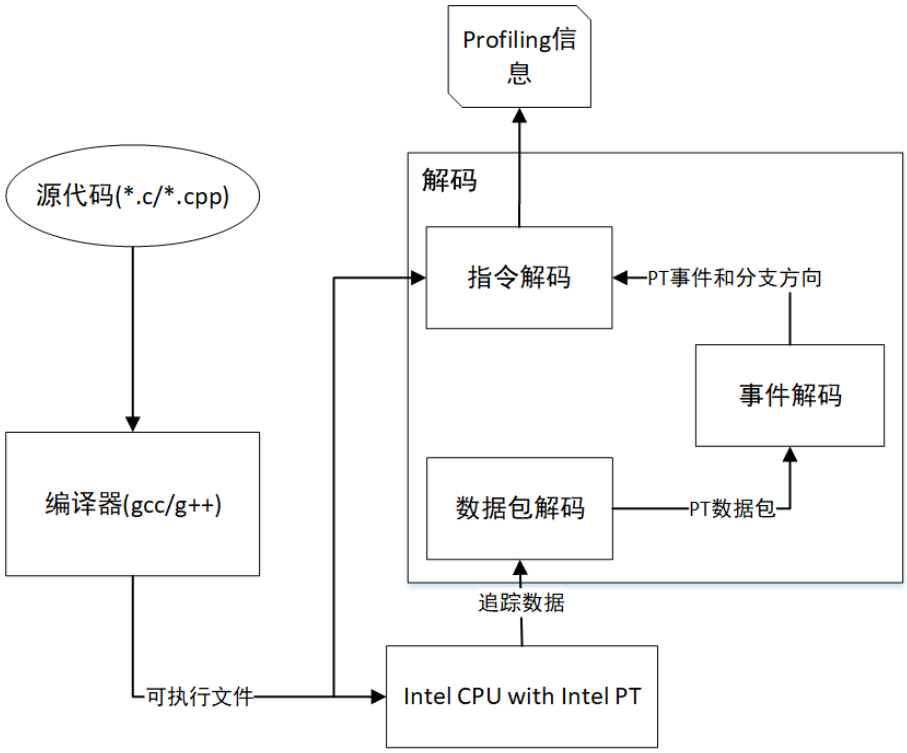

图 3.1　Intel PT **硬件** Profiling：**解码**

在本章中，首先具体介绍了数据包、事件、指令三层解码器的实现和使用，然后重点给出了并行化解码的具体算法实现，并在解码的最后同时给出了利用编译过程的调试信息完成到 C/C++ 程序源码级别 Profiling 的实现。

3.1 数据包层解码器

数据包层解码器的主要功能在于实现对 Intel PT 数据包的逐个解码，通过正确的设置与分配后，数据包层解码器能够逐个导出 Intel PT 追踪过程的数据包。我们知道，Intel PT 追踪数据的最终表达形式是存储的二进制数据流，数据包层解码器给出了将此二进制流还原为一系列 PT 数据包的实现，因此数据包层解码器主要需要完成根据 Intel PT 编码格式完成对二进制流的解码，以及对 Intel PT 数据包的数据结构的设计，使得解码后的数据包的表达形式易于理解。

通常情况下，存储在二进制流中的每一个 Intel PT 数据包是由数据包头和数据包负载组成的，数据包头标识了数据包的类型，每种数据包有着特殊的编码格式，因此对于不同的数据包也应该采用不同的解码方式。因此，数据包层解码器对 Intel PT 的追踪数据流进行扫描，根据数据包头判断数据包的类型，并根据不同数据包的编码方式进行数据包的解码。除此以外，在实现数据包解码器的时候，增加了额外的单纯对 PSB 数据包的扫描，PSB 数据包标志着一段追踪数据的边界，提供了 Intel PT 追踪数据的同步点，每次解码的开始或解码过程中发生错误后都应该首先识别一个 PSB 数据包，这对 Intel PT 追踪数据的划分有重要作用。

数据包层解码器能够逐个解码追踪过程中产生的数据包，每个数据包中标识了该数据包的类型、大小，数据包携带着数据包类型所对应的不同的负载数据，如 TNT 数据包会带有小于 6 位或小于 48 位的条件分支发生/未发生的指示，TIP 数据包带有一个小于等于 8 个字节的指令指针地址和对该指令指针地址是否被压缩的指示，TSC 数据包带有一个当前的 64 位的硬件时间戳等。

Intel PT 追踪数据中 PSB 数据包指定了 PT 数据的边界，在实现解码器时，首先识别 PSB 数据包进行 Intel PT 数据的同步。数据包层解码器中提供了三种同步功能，可向前 (pt_pkt_sync_forward)、向后 (pt_ppkt_sync_backward) 迭代同步点 (一个 PSB 数据包在 Intel PT 二进制流的偏移位置)，或者手动设置同步点 (pt_pkt_set_sync)。解码器会记下解码时上一个 PSB 数据包的偏移量，随后调用 pt_pkt_sync_forward 和 pt_pkt_sync_backward 以此为起点。通过 pt_pkt_get_offset 可以获取当前解码器

表 3.1　二进制程序

… 00000000000005fa <fun>: 5fa: push %rbp 5fb: mov %rsp,%rbp 5fe: mov %edi,-0x14(%rbp) 601: mov %esi,-0x18(%rbp) 604: mov %edx,-0x1c(%rbp) 607: mov -0x14(%rbp),%eax 60a: cmp -0x18(%rbp),%eax 60d: jle 617 <fun+0x1d> 612: mov %eax,-0x4(%rbp) 615: jmp 61d	<fun+0x23> 617: mov -0x18(%rbp),%eax 61a: mov %eax,-0x4(%rbp) 61d: mov -0x1c(%rbp),%eax 620: cmp -0x4(%rbp),%eax 623: jle 62b <fun+0x31> 625: mov -0x1c(%rbp),%eax 628: mov %eax,-0x4(%rbp) 62b: mov -0x4(%rbp),%eax 62e: pop %rbp 62f: retq …

表 3.2　数据包解码结果

00000000: PSB 00000010: PAD 00000016: TSC 0x725df346c80c 0000001e: PAD 00000026: TMA CTC 0xfd1c FC 0000002e: PAD 00000030: CBR 0x1b 00000034: PSBEND 00000036: MTC 0xa4 …	000006ce: MTC 0xcd 000006d5: MODE.Exec 64 000006d7: TIP.PGE 0x55f1a5e415fa 000006df: PAD 000006e0: TNT TN (2) 000006e1: TIP.PGD 0x165f 000006e4: MTC 0xce 000006e6: MTC 0xcf …

位置，通过 pt_pkt_get_sync_offset 可以获取上一个同步点的位置。解码器同步后，通过调用 pt_pkt_next 可以逐个获取数据包，此外，数据包层解码器提供了一个对未知数据包回调函数的配置，在分配解码器时，通过对回调函数的设置，可以实现自己对未知数据包的处理。

表3.1为一个简单的可执行程序中函数 fun 的汇编代码表示，fun 的主要功能在于返回三个整数的最大值，程序中调用了 fun(1, 2, 3)，在追踪时通过指令指针地址过滤配置了 Intel PT 仅追踪程序中的 fun 函数，表3.2给出了利用数据包层解码器进行简单的原始 Intel PT 数据包导出的结果输出的示例。PSB 数据包标志着追踪数据的界限，并伴随着其他相关的数据包，PAD 数据包仅作为填充数据，无实际意义，TIP.PGE 标志着一段追踪的开始，其中包含的指令指针地址对应着 fun 函数的起始地址，接下来的 TNT 数据包给出了 fun 函数中的条件跳转发生与否，最后生成 TIP.PGD 数据包标志此段追踪结束，需要注意的是，尽管仅对 fun 函数进行追踪，但是在对应的 TIP.PGE 之前或者 TIP.PGD 之后仍然有一系列的数据包生成，

这些数据包与时间戳 (TSC、MTC) 和总线频率 (CBR) 等附加信息有关。

3.2 事件层解码器

事件层解码器提供了对于数据包组合成的更高级事件的处理，输入类似于数据包层解码器，仅为 Intel PT 的追踪数据。事件层解码器并不提供对于可执行文件的处理，但是利用事件层解码器，可以通过自己实现对二进制文件的解码处理，利用事件层提供的控制语句跳转信息，重新构建可执行程序完整的执行过程，并能够获取 Intel PT 追踪数据较细粒度的信息 (指令层解码器所无法获取的)。

在事件层解码器实现的过程中，我们主要需实现对事件的封装，通过对数据包的顺序扫描识别相互依赖的数据包，并完成对特定数据包的组合以返回相应的事件，当发生相应的事件时，指示解码的用户对此事件进行相应的处理，除此以外，应该提供对二进制文件无法确定的控制流转移方向的查询 (通过 TNT 数据包提供条件转移指令的发生/或未发生，通过 TIP 数据包提供间接跳转的目的指令指针地址)，同时由于 Intel PT 追踪过程中的压缩机制，指令指针地址在数据包中的表示并不完整，因此在事件层解码器中应该完成必要的解压缩以还原完整的指令指针地址。

事件层解码器通过对 Intel PT 追踪数据包的组合与处理，将其封装成更高层次的 Intel PT 事件，并提供无法直接从可执行文件得到的控制流转移方向。与实现的 Intel PT 数据包数据结构类似，事件的数据结构中包含着该事件的类型和事件的基础信息，如是否包含时间戳 (若包含则提供此事件发生的时间戳信息)，指令指针地址产生被抑制的指示等，除此以外，不同事件携带着不同的负载信息，例如普通的 Intel PT 追踪启用/禁用事件会提供发生该事件的指令指针地址，异步的 Intel PT 追踪禁用事件还会提供发生该异步事件的源指令指针地址等。

类似于数据包层解码器，在创建事件层解码器时，除了必须的 PT 数据、CPU 信息等配置外，同样可以配置对未知数据包进行解码的回调函数，若指定未知数据包大小，解码器可以忽略未知数据包。解码的回调函数如果未被指定，解码将会终止，并返回-pte_bad_opc 错误信息。

同样，在使用解码器之前，首先应该识别追踪数据的 PSB 数据包，将其同步到 Intel PT 数据包流中，事件层解码器提供了同样的向前 (pt_qry_sync_forward)，向后 (pt_qry_sync_backward) 迭代同步点，以及手动设置同步位置 (pt_qry_sync_set)

的三种同步方式。

同步成功后，事件层解码器会继续读取 PSB 数据包的附加数据包初始化内部状态，如果在此同步点启用了追踪，则会得到起始解码的指令指针地址，如果在此同步点禁用了跟踪，返回的状态位 (pts_ip_suppressed) 表明此处无可用指令指针地址。同步后，可以根据得到的指令指针地址对应二进制文件的指令开始解码，如果能够通过二进制文件确定下一条指令 (顺序执行或无条件直接跳转指令)，可以对此二进制文件继续解码，当遇到无法直接确定控制流转移的分支指令时，可以询问事件层解码器此分支指令的方向：

对于条件分支指令，pt_qry_cond_branch 提供了此条件分支的发生或未发生，对于间接分支指令，pt_qry_indirect_branch 提供了此间接分支的目的指令指针地址。如果 Intel PT 追踪时配置了返回压缩，对于 RET 指令，首先应该用 pt_qry_cond_branch 条件分支查询，如果查询成功，返回值被压缩，并应采用此分支，表明此次返回地址对应着调用栈的首个调用，否则，若查询失败，则应继续使用 pt_qry_indirect_branch 查询。

除了 pt_qry_cond_branch 接口能够用于查询下一个条件分支是否发生，pt_qry_indirect_branch 接口能用于查询下一个间接分支的目的指令指针地址之外，事件层解码器还提供了其它的功能：pt_qry_event 用于查询下一个事件，pt_qry_time 用于查询当前时间 (追踪程序时执行到此的时间)

事件层解码器的每个函数调用都会都返回一个状态位 (可能返回该状态位的负值作为错误代码)，其中 pts_ip_suppressed 表明当前没有可用的指令指针地址 (调用间接分支查询和同步函数时可能返回)，pts_event_pending 状态表明当前有一个事件等待处理，在重构程序执行控制流前，若有事件未处理应该先查询该事件并进行相应的处理，pts_eos 状态表明追踪的结束，并在后续的查询都将返回-pte_eos。

仍然以表3.1所示的 fun 程序为例，表3.3给出了利用事件层解码器将数据包组合成事件的输出，其中 Enable 事件标志着一段追踪的开始，Disable 事件标志着一段追踪的结束，在这两个数据包之中，可以通过查询的方式询问控制流转移的方向，Branch Taken 表示 fun 中的第一个条件跳转分支发生，Branch Not-Taken 表示 fun 中第二个条件跳转分支未发生，除此以外，CBR 事件指示处理器的总线频率可能变化，exec_mode 事件表示了执行模式为 64 位。

表 3.3　事件解码结果

CBR 0x1b	tsc: 0x725df346c80c	
CBR 0x1b	tsc: 0x725df346fa28	
CBR 0x1b	tsc: 725df3477878	
Enable	IP: 0x55f1a5e415fa	tsc: 725df35173e0
EXEC_MODE	IP: 0x55f1a5e415fa	tsc:725df35173e0
	Branch Taken	
	Branch Not Taken	
Disable	IP: 0x55f1a5e4165f	tsc: 725df35173e0

3.3　指令层解码器

指令层解码器是对事件层解码器的封装，指令层解码器调用事件层解码器对 Intel PT 数据处理的方式，获取 Intel PT 追踪数据包产生的事件和控制流转移的方向，同时指令层解码器提供了对二进制文件的处理，通过对内存映像 (image) 的设置，指令层解码器能够利用事件层解码器得到的控制流转移方向，顺序遍历程序执行的机器指令。

因此除了 Intel PT 相关的配置之外 (Intel PT 数据等)，还需要对指令层解码器设置当前追踪数据所对应的内存映像 (加载到内存的二进制文件)，实现中内存映像由 pt_image 对象表示，内存映像设计为段 (pt_section) 的不重叠的内存区域的集合，pt_section 内包含其对应的二进制文件的名字、大小、加载在内存中的位置等信息 (这些信息应该由追踪过程中生成的边带信息获得)，在解码时，当获取一个指令指针地址时，通过读取内存映像，可以将其映射到具体二进制文件的偏移位置，获取 Intel PT 数据所对应的正在追踪的机器指令。

内存映像通过 pt_image_alloc 进行分配，通过 pt_image_free 进行释放，每个解码器不能够共享内存映像，必须通过重新分配或者复制 (pt_image_copy) 分配新的内存映像。通过重复调用 pt_image_add_file 或 pt_image_add_cached 可以对 pt_image 进行段的填充，如果新添加的段与现有段有重叠，则现有段将被截断或拆分给新段腾出空间，调用 pt_image_copy 复制另一个 pt_image 的所有 pt_section。在程序执行的过程中，内存映像可能会改变，通过 pt_image_remove_by_filename 可以通过二进制文件名删除先前添加的段，通过 pt_image_remove_by_asid 可以删除某个地址空间内的所有段。除了可以通过添加二进制文件的方式设置内存映像，pt_image 可以通过调用 pt_image_set_callback 注册回调函数，若无法从当前内存映像中找到目的指令指针地址时，将会调用回调函数，可以用于处理没有被添加到

内存映像中的程序追踪的解码，并可以实现较好的控制。

内存映像只能被单独的解码器使用，不能被多个解码器所共享，与此相比内存映像缓存 (pt_iscache) 提供了给多个解码器共享使用的策略，对于每一段二进制程序，内存映像缓存仅映射一次，通过调用 pt_iscache_alloc 可进行内存映像缓存的分配，使用 pt_iscache_free 可进行内存映像缓存的释放，但释放缓存并不会不会破坏添加到缓存中的段，它们将保持有效，直到不再使用为止。通过调用 pt_iscache_add_file 实现二进制程序到内存映像缓存中的添加，并返回一个用于识别此缓存中该段的段标识符（ISID）。使用 pt_image_add_cached 可以将内存映像缓存中的段添加到内存缓存中。能够同时向多个内存映像中添加该内存映像缓存。

设置内存映像后，指令层解码器根据 Intel PT 追踪数据提供的指令指针地址对应到可执行文件的偏移位置对指令进行还原，在实现中，每条指令的数据结构提供了指令指针地址，字节大小，指令的机器码，以及该指令的所对应的内存映像段的标识符，和当前执行模式等信息，同时将指令粗略地分类为近程调用、近程返回、近程无条件跳转、近程条件跳转、远程调用、远程返回、远程跳转等十种类型。

与数据包层、事件层解码器类似，在分配指令层解码器时，除了必需的配置字段之外，可以指定可选的对未知数据包进行解码的回调函数，指定后，对于未知的数据包格式将会调用此函数，可以通过设置未知数据包的大小从而忽略未知数据包。如果未指定解码回调，在处理未知数据包时，解码器的解码过程中止，返回-pte_bad_opc 错误信息。同样，在使用解码器之前，首先应该识别追踪数据的 PSB 数据包，将其同步到 Intel PT 数据包流中，指令层解码器提供了向前 (pt_insn_sync_forward)，向后 (pt_insn_sync_backward) 迭代同步点，以及手动设置同步位置 (pt_insn_sync_set) 的三种同步方式。解码器会记住它解码的上一个同步数据包的偏移位置。随后对 pt_insn_sync_forward 和 pt_insn_sync_backward 的调用将以此为起点。

通过 pt_insn_get_offset 可以获取当前指令解码器位置作为 Intel PT 缓冲区的偏移量，通过 pt_insn_get_sync_offset 可以获取上一个同步点的位置作为 Intel PT 缓冲区的偏移量。解码器同步后，通过重复调用 pt_insn_next 能够实现追踪程序执行过程机器指令的迭代。

如果使用了内存映像段缓存，则可以使用内存映像的段标识符将指令映射包

表 3.4　指令解码结果

0x55f1a5e415fa	push %rbp
0x55f1a5e415fb	mov %rsp,%rbp
0x55f1a5e415fe	mov %edi,-0x14(%rbp)
0x55f1a5e41601	mov %esi,-0x18(%rbp)
0x55f1a5e41604	mov %edx,-0x1c(%rbp)
0x55f1a5e41607	mov -0x14(%rbp),%eax
0x55f1a5e4160a	cmp -0x18(%rbp),%eax
0x55f1a5e4160d	jle 617 <fun+0x1d>
0x55f1a5e41617	mov -0x18(%rbp),%eax
0x55f1a5e4161a	mov %eax,-0x4(%rbp)
0x55f1a5e4161d	mov -0x1c(%rbp),%eax
0x55f1a5e41620	cmp -0x4(%rbp),%eax
0x55f1a5e41623	jle 62b <fun+0x31>
0x55f1a5e41625	mov -0x1c(%rbp),%eax
0x55f1a5e41628	mov %eax,-0x4(%rbp)
0x55f1a5e4152b	mov -0x4(%rbp),%eax
0x55f1a5e4152e	pop %rbp
0x55f1a5e4152f	retq0

含此指令的二进制文件中，映射二进制文件后，可以通过二进制文件中包含的调试信息将二进制机器指令映射源代码。pt_insn_next 可能指示返回的指令之后发生的错误，如果设置了其 iclass 字段，则返回的指令有效。指令层解码器使用类似于事件层解码器的事件系统。待处理事件由 pt_insn_sync_forward，pt_insn_sync_backward，pt_insn_next 和 pt_insn_event 返回的状态标志位中的 pts_event_pending 标志指示。当返回时设置了 pts_event_pending 标志时，可通过重复调用 pt_insn_event 得到尚未被处理的事件并一一处理，处理完成后继续循环调用 pt_insn_next 以继续执行指令解码。

仍然以表3.1程序为例，指令层解码器通过对二进制可执行文件的处理重新构建了表3.4所示的程序中 fun 函数的完整执行流，表3.4顺序给出了 fun 函数动态运行时的每一条指令，分支指令的后续直接为此次分支发生/不发生后执行的指令。

3.4　并行解码器

Intel PT 数据包层解码器读取压缩的 Intel PT 追踪二进制流数据，能够解码还原为一系列的追踪数据包；Intel PT 事件层解码器提供了对 Intel PT 追踪数据包组合所形成的更高级别的事件的解码，通过自己实现对二进制文件的解码处理，事件层解码器能够通过查询的方式提供不能直接通过二进制文件获得的控制流方向，

重新构建程序的完整执行过程；Intel PT 指令层解码器读取压缩的追踪数据，同时根据设置的内存映像对应的可执行文件进行指令解码，迭代还原程序执行过程中的机器指令，通过调用事件层解码器对追踪数据的处理，在每次到达条件分支或者间接分支时，通过查询追踪数据中所对应的数据包 (TNT、TIP 数据包)，获取分支进行的方向，除此以外，Intel PT 指令层解码器还会还能够获取时间信息，报告异步事件 (例如异常、中断、事务终止等)。

为重新构建程序的完整执行过程，主要需要用到指令层的解码器。然而追踪过程中产生的大量追踪数据 (每个内核每秒数百兆字节)，通常使得解码阶段的开销比追踪过程的开销要多几个数量级，解码阶段过于庞大的时间开销对 Profiling 的效率造成了较为严重的影响。因此，本节在实现的数据包层、事件层、指令层三个层次解码器的基础上，实现了 Intel PT 解码阶段的并行化。主要思想是利用数据包层的解码器提供的同步函数实现对 Intel PT 追踪数据的划分，对于划分后的不同数据创建不同的线程并分配不同的指令层解码器对其进行独立的解码，从而提高解码阶段的效率。此外，在对解码得到的机器指令到源码的映射的过程中，同样利用到了解码阶段的并行化策略，对不同指令层解码器得到的机器指令，同样实现并行的到源码的映射，同时提高了此阶段的效率。

Intel PT 追踪数据中的 PSB 包指定了一段追踪数据的边界，PSB 数据包的产生有固定的周期性，配置追踪时可确定每产生一定数量的数据包时产生一个 PSB 数据包，同时在实现数据包层、事件层、指令层三个层次的解码封装时，我们独立实现了一个仅对于 PSB 数据包扫描的函数 (同步函数)，利用对 PSB 数据包进行扫描的同步函数可以对 Intel PT 追踪数据进行较为准确且良好的划分，并对划分后的数据进行独立的解码。

数据包层、事件层、指令层三个层次的解码器均通过同步函数提供了寻找向前或向后或手动设置 PSB 数据包偏移位置的方法，并行化解码器实现的主要思路在于以全部 Intel PT 追踪数据为输入，分配数据包层解码器 (更小的时间/空间开销)，利用数据包层解码器的同步函数搜索全部 PSB 数据包在二进制流中的偏移位置，实现对 Intel PT 追踪数据的划分，之后利用指令层解码器对划分后的 Intel PT 追踪数据进行独立的解码。图 3.2给出了并行化解码的流程。

1. 数据划分：利用数据包解码器划分 PT 数据的过程如下：(1) 以完整 Intel PT 追踪数据为输入，配置数据包层解码器相关参数，分配数据包层解码器 (2) 连续调

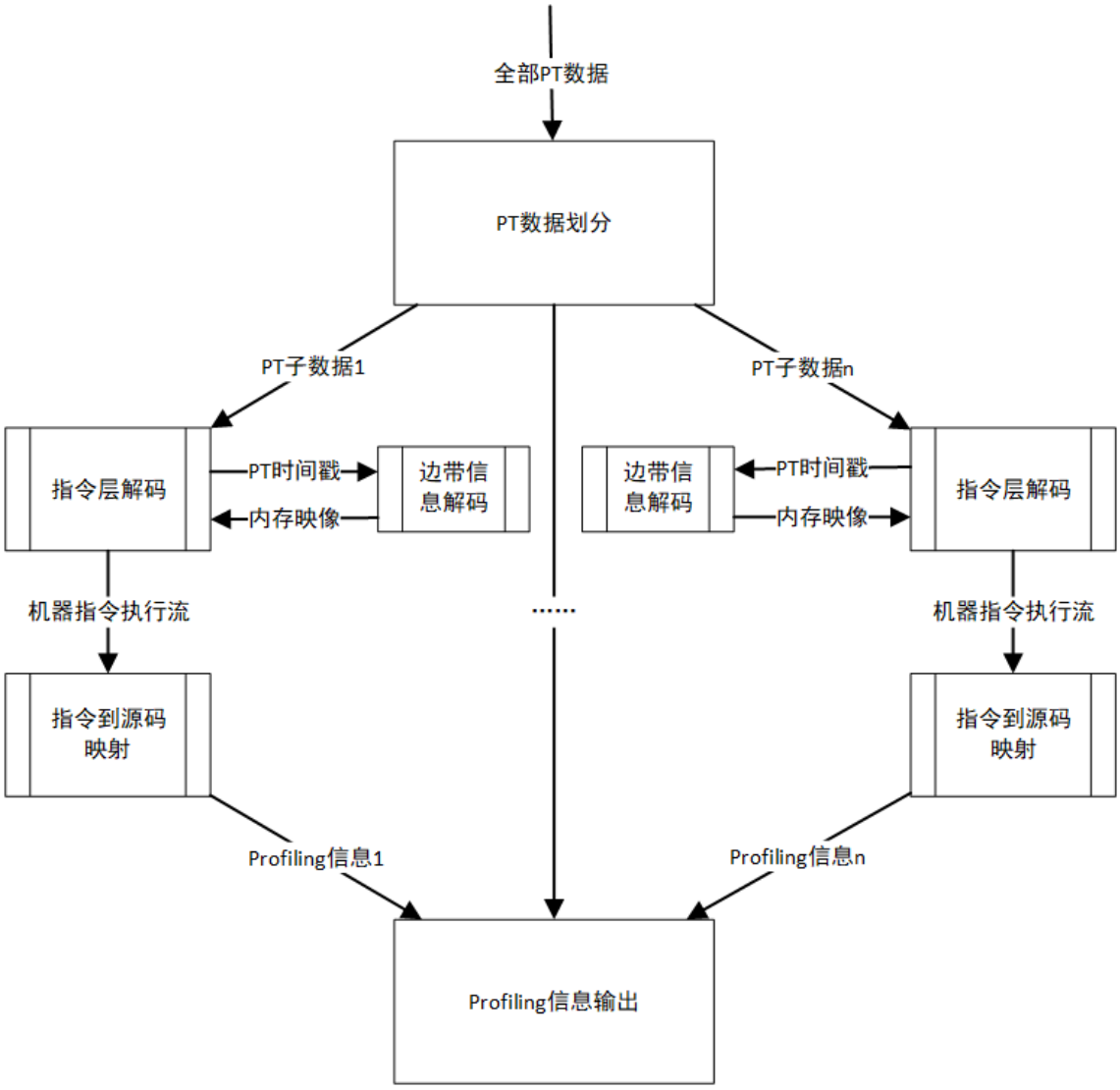

图 3.2　并行解码流程图

用 pt_pkt_sync_forward 同步函数向前迭代 Intel PT 追踪数据的同步点，记住每一个同步点的偏移位置 (3) 根据设置的并行级数 (分配多少个指令层解码器进行独立和同步的解码) 以及记录的所有同步点位置，将 Intel PT 追踪数据划分为较为平均的并行级数份。

2. 指令层并行解码：分配并行级数个指令层解码器，每个指令层解码器以 1 中划分后的一段 Intel PT 追踪数据为输入，并配置其他相应的参数。对每个指令层解码器新建对应的线程，每个线程对应执行该解码器的解码函数，开始重新构建程序完整的执行过程，迭代产生执行的每一条指令。每个指令层解码器在进行解码时，应该将 Intel PT 事件与边带信息相协调，需要利用 Intel PT 事件的时间戳信息与边带事件的时间戳信息，对于 Intel PT 追踪数据产生事件的每一个时间戳，

应该查看在此事件时间戳之前的所有边带事件 (边带事件记录了所有的线程切换、二进制文件加载等信息)，边带事件能够帮助了解追踪数据对应的进程、线程信息，以设置对应的内存映像 (解码实现中为每一个进程设置了一个内存映像，根据边带事件像二进制文件添加或删除对应的二进制文件段，根据当前追踪数据对应的进程设置相应内存映像)，明确当前的执行线程 (帮助多线程的程序的 Profiling，Intel PT 追踪每一个 CPU 生成相应的 PT 数据，线程信息能够帮助明确该 CPU 此时的线程，在对每个 CPU 解码完成后，根据需要，可以利用时间戳信息对线程在每个 CPU 上的执行流进行拼接，从而获得此线程的完整执行流)。

3. 机器指令向源码的映射：根据 2 中解码迭代的每一条指令，我们可以明确该指令所对应的二进制文件和在此二进制文件的偏移位置。对于每个 C/C++ 程序，在利用 gcc/g++ 编译时我们应该添加-g 参数生成它们的调试信息，我们通过对可执行二进制文件的调试信息的处理，可以得到每一条机器指令所对应的源码信息，这些信息包括二进制文件中每一条机器指令所对应在源码的行号、函数名等，从而实现在源程序级别的 Profiling。同时，在对源码进行 Profiling 的时候，同样实现了并行化的设计，对源码 Profiling 的并行化采用了解码阶段的相同并行化策略，对于 2 中每一个解码器的结果进行了独立的向源码的映射，从而实现了并行化的 Profiling。

4 实验结果

4.1 性能评价

本节利用 Software-artifacts Infrastructure Repository (SIR) 测试基准中提供的 C/C++ 程序，对追踪过程的开销和解码过程的效率进行了评价。对于追踪阶段的开销，本实验比较了利用 gcov 进行的软件 Profiling 和利用 Inel PT 进行的硬件 Profiling 对程序执行造成的额外开销，包括编译后的可执行文件大小和程序运行的时间；对于解码过程，实验比较了设置不同的线程级数后对 Intel PT 追踪数据进行解码所耗费的时间。

实验过程中，配置了 Intel PT 追踪指定程序在每一个 CPU 上的全部运行，对每个 CPU 分配 512KB 的内存空间用于记录边带信息，分配 4MB 的空间用于记录 Intel PT 追踪数据，选取了 SIR 中的 10 个 C/C++ 程序，通过 gcc/g++ 在优化选项为-O2 下编译，完成运行时的追踪和运行后的解码。实验运行环境为 ubuntu18.04 操作系统，Linux 5.3 内核版本，Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 8 核处理器，gcc-7.5.0 编译器。

表 4.1 源程序信息

名称	版本	代码行	函数/类	测试用例	语言	描述
flex	1.1	10459	162	525	C	生成词法分析器
grep	1.2	10068	146	470	C	查找字符串
gzip	1.5	5680	104	214	C	压缩文件
printtokens	2.0	726	18	4130	C	词法分析
printtokens2	2.0	570	19	4115	C	词法分析
schedule	2.0	412	18	2650	C	优先级调度
schedule2	2.0	374	16	2710	C	优先级调度
totinfo	2.0	565	7	1052	C	信息统计
vim	1.0	122169	1999	975	C	编辑器
concordance	1.0	1034	5	372	C++	文本分析

实验中选择了 SIR 的 10 个 C/C++ 程序，这些程序均是用于解决实际问题的真实程序，表4.1给出了选取的 10 个程序信息的描述，包括这些程序的名称、版本数、代码行、C 程序的函数数量或 C++ 程序的类数量、测试用例数、语言以及对程序的简单描述。每一个程序的测试选择都涵盖了 SIR 所提供的对应版本的全部测试用例，这些测试用例能够很好的覆盖到该程序的全部代码。在实验过程中，对每个程序的追踪与解码是对程序的全部测试用例进行的，以减少单个测试用例运

行的偶然性。

表 4.2　可执行文件大小和开销

程序名称	普通编译	gcov 选项编译	gcov 选项开销
flex	229.0KB	315.5KB	37.8%
grep	100.4KB	180.6KB	79.8%
gzip	200.5KB	260.7KB	30.0%
printtokens	16.8KB	38KB	126.1%
printtokens2	13.6KB	39.4KB	190.0%
schedule	13.5KB	35.1KB	160.0%
schedule2	13.5KB	34.7KB	157.0%
totinfo	13.3KB	33.2KB	149.6%
vim	762.4KB	1.3MB	70.5%
concordance	38.9KB	67.0KB	72.2%

表 4.3　运行时间和开销

	无 Profiling	gcov		Intel PT	
程序名称	时间 (s)	时间 (s)	开销	时间 (s)	开销
flex	1.000823	1.188910	18.6%	1.025578	2.5%
grep	1.112202	1.219270	9.6%	1.136327	2.4%
gzip	0.602457	0.721433	19.7%	0.613496	1.1%
printtokens	6.492572	7.409447	14.1%	6.684311	2.9%
printtokens2	6.270654	6.973193	11.2%	6.394291	1.9%
schedule	4.110148	4.537002	10.4%	4.155437	1.1%
schedule2	4.222819	4.828388	14.3%	4.294066	1.6%
totinfo	1.742980	2.001803	14.8%	1.757530	0.8%
vim	16.051291	20.318301	26.5%	16.546003	3.1%
concordance	4.390644	4.787759	9.0%	4.429473	0.9%

4.1.1　运行开销

为了评价利用 Intel PT 硬件机制追踪运行时程序而对其造成的影响，实验中主要比较了基于软件的 Profiling 方式 gcov 和利用 IntelPT 进行硬件 Profiling 的开销，表4.2和表4.3给出了 Profiling 对程序运行过程开销对比。主要进行了两个方面的开销比较，首先表4.2给出了编译后源程序的大小，利用 gcov 进行软件层面的 Profiling 需要在编译时增加 gcov 相关编译选项，编译器在编译时向源程序进行插桩，注入额外的代码以收集运行时信息，从表4.2可以看出，这种机制会使可执行程序的大小发生明显的变化，平均会使可执行程序的大小增加 64% 左右，而基于 Intel PT 的硬件 Profiling 借用硬件机制追踪，对源程序无侵入，因此并不会改变可执行程序的大小；表4.3给出了每个程序全部测试用例在无 Profiling、利用 gcov

进行软件 Profiling，利用 Intel PT 进行硬件 Proiling 的执行时间，图4.1直观地给出了两种 Profiling 方式造成额外开销的对比，可以看出基于 Intle PT 的硬件 Proiling 方式造成的额外开销要远小于 gcov 软件 Profiling 造成的额外开销，最差情况下，gcov 在 vim 中造成的额外开销甚至达到了 26.5% 以上，而在 vim 中使用 Intel PT 进行的硬件 Profiling 造成的额外开销在其中仅有 3.1%，在本实验中，总体上，gcov 造成的额外开销在 10% ~ 30% 间，而 Intel PT 仅造成的额外开销小于 5%，远小于 gcov 造成的平均额外开销。

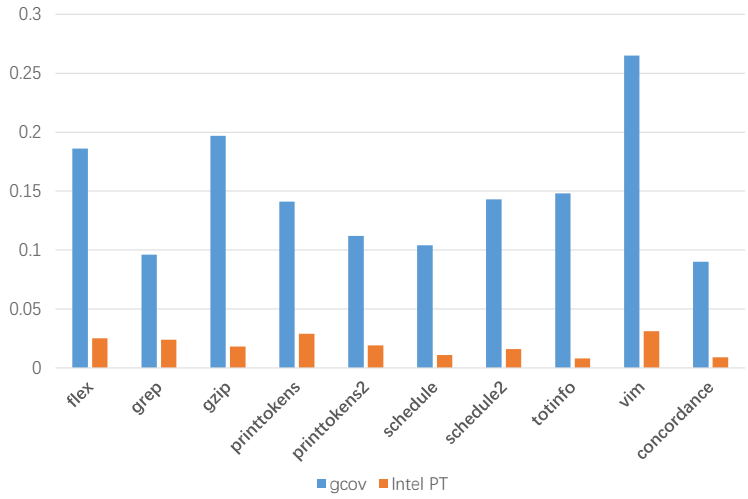

图 4.1　运行开销

4.1.2　解码效率

追踪程序的全部测试用例的执行并产生一系列追踪数据，这些追踪数据被导出到磁盘中，解码器对这些数据进行离线的解码重新构建程序的执行流。追踪过程中会产生大量的数据，表4.4和4.5给出了对每个程序的所有测试用例进行追踪导出的的总数据大小，对每个 CPU 的追踪数据包括记录的 Intel PT 数据和所需要的边带信息，Intel PT 数据即 Intel PT 产生的一系列数据包，边带信息能够帮助我们了解程序执行时的内存和线程等信息，用于内存映像设置和执行线程信息获得等，对这些追踪数据进行解码需要耗费大量的时间，本课题通过实现解码阶段的并行化提高解码阶段的效率。

本节对实验中追踪 SIR 程序全部测试用例的 Intel PT 数据进行解码以评价实

表 4.4　追踪数据大小 (a)(单位:MB)

	flex		grep		gzip		printtokens		printtokens2	
CPU	PT	边带	PT	边带	PT	边带	PT	边带	PT	边带
0	18.7	0.7	0	0.8	9.9	0.1	4.7	1	10.3	1.5
1	21.9	0.6	0	0.1	3.2	0.1	1.2	0.5	8.1	1.4
2	29.2	0.8	3.8	0.1	2.5	0.1	6.1	0.7	13.1	1.7
3	21.2	0.7	35.3	0.3	8.1	0.1	8.3	1.4	6.0	1.4
4	9.7	0.5	1.7	0.1	11.9	0.1	1.4	0.5	4.2	1.2
5	12.9	0.5	16.2	0.2	1.2	0.1	9.4	1.3	14.9	2.0
6	15.2	0.6	6.6	0.1	0.5	0.1	4.5	0.6	7.5	1.5
7	33.9	0.8	0	0.1	5.2	0.1	35.8	3.2	11.2	1.7
总和	168.2		24.8		43.4		80.9		87.9	

表 4.5　追踪数据大小 (b)(单位:MB)

	schedule		schedule2		totinfo		vim		concordance	
CPU	PT	边带	PT	边带	PT	边带	PT	边带	PT	边带
0	10.1	1.0	0.4	0.8	0.6	0.1	50.1	3.6	37.4	0.9
1	18.9	1.6	0.3	0.1	2.4	0.2	67.3	4.1	46.1	1.2
2	1.3	0.2	19.2	0.1	5.6	0.2	79	4.5	51.2	1.1
3	6.5	0.8	12.1	0.3	1.6	0.2	70.2	4.5	31.5	0.8
4	6.5	0.8	14.9	0.1	7.7	0.4	74.8	4.0	43.8	0.7
5	1.0	0.3	6.6	0.2	9.1	0.6	66.1	4.2	27.9	0.8
6	7.3	0.8	0.2	0.1	4.5	0.3	63.5	4.1	43.8	0.9
7	2.7	0.6	7.7	0.1	1.5	0.1	69.4	4.4	41.8	0.9
总和	60.5		67.3		35.5		574.0		373.3	

现的并行化解码器的性能。在对这些追踪数据进行解码时设置不同的并行级数 (解码的线程数) 进行比较，实验中首先测试非并行化的解码器 (不进行 Intel PT 的数据划分，解码线程为 1) 的解码时间，然后将追踪数据划分为 2-8 份，并创建对应的解码线程进行并行解码。

表4.6给出了全部实验的解码时间记录，包括非并行化解码以及设置并行级数为 2 ~ 8 时的解码时间，图 4.2和图 4.3直观地给出了非并行化解码器以及设置不同并行级数解码器的解码时间对照。可以发现，在该实验平台下，此并行化解码器最好情况下解码时间小于原来解码时间的 1/3，表现最好的 flex 减少了非并行解码器 73% 的解码时间，总体上，并行化解码器最好能够减少非并行解码器解码时间的 61%-73%，且随着 Intel PT 追踪数据规模的增加，并行解码器对解码效率的提高依然较为明显，其中较大数据规模的 concardance 的解码时间仍然减少了非并行解码时间的 70% 左右。

表 4.6　解码时间记录 (单位:s)

	1	2	3	4	5	6	7	8
flex	410.1	224.0	158.6	127.4	136.0	121.2	119.6	110.5
grep	257.5	160.0	115.8	94.8	88.7	85.7	79.5	77.6
gzip	384.5	215.9	157.1	127.7	126.1	117.4	109.8	103.8
printtokens	199.8	116.2	84.4	69.4	71.8	65.8	65.2	61.8
printtokens2	204.4	116.9	85.3	71.5	74.0	68.6	67.0	62.2
schedule	148.5	84.4	59.9	49.7	50.5	46.6	46.7	43.2
schedule2	163.8	92.6	66.7	54.2	55.8	51.4	50.6	46.2
totinfo	94.9	52.8	38.4	30.3	32.3	28.6	27.4	25.8
vim	1501.8	898.4	703.0	620.7	623.9	590.2	587.2	578.1
concordance	799.8	438.2	314.4	253.6	265.9	243.2	233.6	220.5

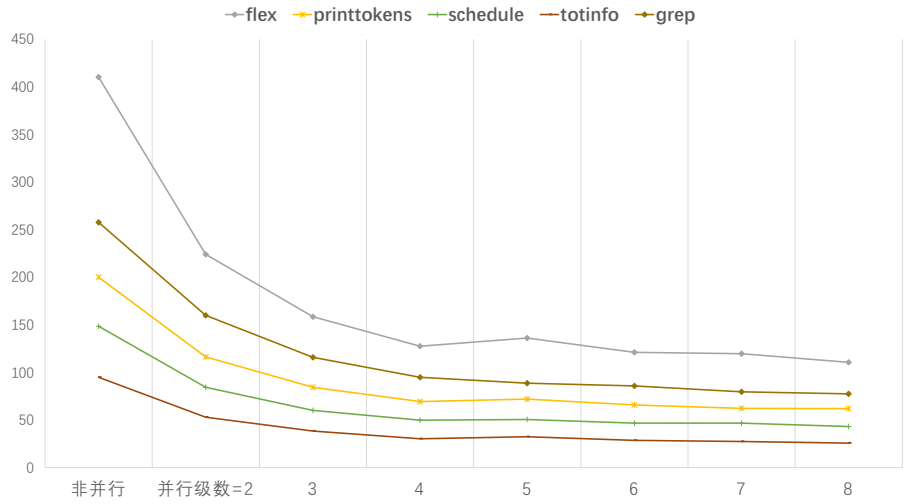

图 4.2　解码时间对照 (a)

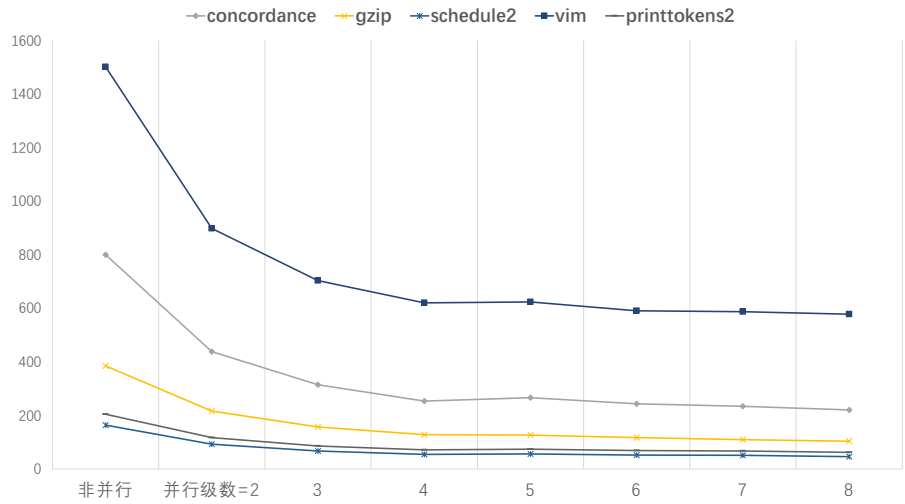

图 4.3　解码时间对照 (b)

4.2 源码 Profiling 结果

此节给出了实现的 C/C++ 程序 Profiling 工具的最终结果输出示例，如表4.7所示，表格左侧给出了源码信息，对于左侧所示的简单 C 程序，最终此 Profiling 工具能够给出类似于表4.7中间所示的函数调用层次关系，以及右侧所示的执行信息图，分别给出了源码所对应的每一个函数的调用次数以及每一行所对应的执行次数。

表 4.7 源码级 Profiling 结果

line 1 : #include <stdio.h>	…	…
line 2 :	main()	…
line 3 : int add(int a, int b){	add()	…/test.c
line 4 : return a+b;	sub()	div(): 9801
line 5 : }	mul()	mul(): 9801
line 6 :	div()	sub(): 9801
line 7 : int sub(int a, int b){	add()	add(): 9801
line 8 : return a-b;	sub()	main(): 1
line 9 : }	mul()	line:3 9801
line 10:	div()	line:4 9801
line 11: int mul(int a, int b){	add()	line:5 9801
line 12: return a*b;	sub()	line:7 9801
line 13: }	mul()	line:8 9801
line 14:	div()	line:9 9801
line 15: int div(int a, int b){	add()	line:11 9801
line 16: return a/b;	sub()	line:12 9801
line 17: }	mul()	line:13 9801
line 18:	div()	line:15 9801
line 19: int main(){	add()	line:16 9801
line 20: int i, j;	sub()	line:17 9801
line 21: for (i = 1; i < 100; i++){	mul()	line:19 1
line 22: for (j = 1; j < 100; j++){	div()	line:21 100
line 23: add(i, j);	add()	line:22 9900
line 24: sub(i, j);	sub()	line:23 9801
line 25: mul(i, j);	mul()	line:24 9801
line 26: div(i, j);	div()	line:25 9801
line 27: }	add()	line:26 9801
line 28: }	sub()	line:29 1
line 29: return 0;	mul()	line:30 1
line 30: }	…	…

5 总结与展望

本课题实现了一个基于 Intel PT 的 C/C++ 程序的 Profiling 工具，能够给出 C/C++ 程序在源码层面的 Profiling 信息，包括源码的每一行的执行次数，每个函数的调用次数，并且能够给出函数调用层次关系。这些 Profiling 信息能够很好的用于程序的性能优化、调试等等方面。在对 Intel PT 追踪数据进行解码时，实现了解码过程的并行化，缩短了解码过程的开销，提高了解码的效率，并通过实验分析了 SIR 中的 10 个 C/C++ 程序的追踪过程开销和解码的效率。

但是，尽管利用 Intel PT 进行硬件层面的 Profiling 在追踪阶段只会引入很低的额外开销，追踪过程中记录程序控制流会产生大量的数据，并需要在追踪过程中将其导出到磁盘，由于 CPU 执行的速度远大于磁盘 I/O 速度，将 PT 数据导出到磁盘时，可能会产生数据缺失，这会造成最终 Profiling 信息的不完整；由于编译期的各种优化措施，机器指令到源程序的映射具有一定的误差，这可能会造成最终 Profiling 信息的不准确。因此更好地提高 Profiling 信息的完整性和准确性是需要继续研究的问题。此外，Intel PT 追踪数据解码后能够重构程序完整的执行流，因此能提供非常丰富的 Profiling 信息，此次毕设仅输出了函数调用层次结构以及源码级别函数、行的执行次数，以更好的形式输出得到的 Profiling 信息也是一个需要完善的部分。

参考文献

[1] GNU. GNU gprof[H]. https://sourceware.org/binutils/docs/gprof, 2020.

[2] BALL T, LARUS J R. Optimally profiling and tracing programs[J]. ACM Transactions on Programming Languages and Systems(TOPLAS), 1994, 16(4): 1319–1360.

[3] GNU. GNU gcc[H]. https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc, 2020.

[4] GRAHAM S L, KESSLER P B, MCKUSICK M K. Gprof: A call graph execution profiler[J]. ACM Sigplan Notices, 1982, 17(6): 120–126.

[5] CONTE T M, PATEL B A, MENEZES K N, et al. Hardware-based profiling: An effective technique for profile-driven optimization[J]. International Journal of Parallel Programming, 1996, 24(2): 187–206.

[6] LWN.NET. An introduction to last branch records[H]. https://lwn.net/Articles/680985/, 2016.

[7] INTEL. Processor Tracing[H]. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[8] LWN.NET. Adding Processor Trace support to Linux[H]. https://lwn.net/Articles/648154/, 2015.

[9] INTEL. Intel 64 and IA-32 Architectures Software Developer’s Manual, vol.3[H]. 2019.

[10] KASIKCI B, SCHUBERT B, PEREIRA C, et al. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures[A]. Proceedings of the 25th Symposium on Operating Systems Principles[C], 2015: 344–360.

[11] GE X, CUI W, JAEGER T. Griffin: Guarding Control Flows Using Intel Processor Trace[J]. Acm Sigarch Computer Architecture News, 2017, 45(4): 585–598.

[12] SHARMA S D, DAGENAIS M. Hardware-assisted instruction profiling and latency detection[J]. The Journal of Engineering, 2016, 2016(10): 367–376.

[13] CUI W, GE X, KASIKCI B, et al. REPT: Reverse Debugging of Failures in Deployed Software[A]. 13th USENIX Symposium on Operating Systems Design and Implementation OSDI 18)[C], 2018 : 17 – 32.

[14] INTEL. an Intel(R) Processor Trace decoder library[H]. https://github.com/intel/libipt, 2015.

[15] ANDI K. A Simple-PT Linux driver[H]. https://github.com/andikleen/simple-pt, 2015.

[16] LINUX. Linux branch with Intel PT Support[H]. https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/perf-intel-pt.txt, 2020.

[17] LINUX. perf-event-open[H]. http://man7.org/linux/man-pages/man2/perf_event_open.2.html, 2020.

[18] INTEL. Vtune[H]. https://software.intel.com/en-us/vtune-help, 2020.

[19] GNU. GNU gdb[H]. https://sourceware.org/gdb/onlinedocs/gdb/, 2020.

致谢

经过努力，我完成了此次毕业设计实验和论文撰写的全部内容。首先，十分感谢武汉大学计算机学院的何发智老师和南京大学计算机科学与技术系的左志强老师的悉心指导，从此次本科生毕业设计的开题、实验到论文写作，他们都给予了我巨大的支持与帮助，没有他们的指导，我是很难成功完成此次毕业设计的，同时，在指导老师的支持与帮助下，于我所感知到的，在毕业设计这个阶段中我的学习能力、思考能力、动手能力很多方面都有了明显的提高。然后，也十分感谢实验室学长以及班上的同学，他们也给了我很多的帮助。最后，感谢家人、朋友的陪伴，2020 年，作为一个十年的开始，本应该是充满希望的一年，然而，由于疫情的原因，却让人倍感寒冷，家人与朋友的陪伴让我们一起熬过了这个难捱的时期，感谢我两岁半的小外甥点亮了我的冬天，同时，感谢所有疫情时期奋战在疫情前线的医护人员。