内存管理中的基于扩展伯克利包过滤器的工作集大小估算

Zhilu Lian*
软件工程学院
中山大学
珠海 ，中国
lianzhlu@mail2.sysu.edu.cn

Yangzi Li*
软件工程学院
中山大学
珠海 ，中国
liyz25@mail2.sysu.edu.cn

陈志祥
软件工程学院
中山大学
珠海 ，中国
chenzhx69@mail2.sysu.edu.cn

单诗文
软件工程学院
中山大学
珠海 ，中国
shanshw@mail2.sysu.edu.cn

Baoxin Han
软件工程学院
中山大学
珠海 ，中国
hanbx3@mail2.sysu.edu.cn

Yuxin Su
软件工程学院
中山大学
珠海 ，中国
suyx35@mail.sysu.edu.cn

arXiv:2303.05919v1 [cs.PF] 17 Jan 2023

摘要——**工作集大小估算（WSS）对于提升现代操作系统中程序执行效率和内存管理效率具有重要意义。以往的研究提出了多种估算 WSS 的方法，包括自膨胀、Zballoning 等。然而，这些基于虚拟机的方法通常会产生很大的开销，因此使用这些方法估算 WSS 并不切实际。本文提出了一种新颖的框架，利用扩展伯克利包过滤器（eBPF）高效估算 WSS。eBPF 是一项前沿技术，通过附加到内核来监控和筛选数据。我们将 eBPF 程序固定在内核中，从而获取缺页次数及其他内存分配信息。此外，我们使用原生工具收集工作集大小数据，以训练预测模型，并借助 LightGBM 完成估算。LightGBM 是一种实用工具，擅长生成连续值决策树。实验结果表明，与传统方法相比，我们的框架能够精准估算工作集大小，同时将开销降低 98.5%**。

索引词—**扩展伯克利包过滤器、工作集大小估算、LightGBM**

I. 引言

随着云计算服务的快速发展，数据规模不断扩大，内存需求也急剧攀升。然而，问题在于每台物理机的内存容量都有限。因此，最大限度地提高内存利用率是解决内存短缺问题的当务之急。

根据美国国家标准与技术研究院 [11]，，云计算资源包括基础设施即服务（IaaS）、软件即服务（SaaS）和平台即服务（PaaS）。资源分布广泛，使云计算能够覆盖全球更多地区。这扩大了云计算的应用范围，尤其是在高性能需求场景中，进而引发资源管理问题，特别是内存管理问题。此外，虚拟化云基础设施采用资源超额承诺，以更好地利用资源。分配更多虚拟资源给一台机器（或一组机器），超过其（或它们）实际能够使用的数量，这称为资源超额承诺。由于大多数应用程序并非始终会充分使用所申请的资源，这种做法可以节省内存供更多用户使用，从而获得更高收益 [13]。在满足服务级别协议（SLA）与采用超额承诺[6] 之间取得平衡的困难，使这些问题更加严重。

因此，高效的内存管理技术对云计算意义重大。其中，最常见的技术是**工作集大小（WSS）估算**技术。WSS 是进程在一段时间内维持运行所需的内存量 [12]，，即实际将要访问的页面集合。因此，采用非侵入式、低开销的方法收集虚拟机中的 WSS 至关重要。WSS 估算技术通常包括数据收集和数据处理。WSS 估算的主要挑战在于数据收集，因为估算的准确性主要取决于所提供数据的质量。大量论文致力于更准确、高效地估算 WSS，因为这对改善内存管理起着至关重要的作用。然而，以往的 WSS 估算技术难以兼顾效率与可用性：高效的技术实现起来较为复杂，而易于实现的技术性能则较差。此外，以往性能和准确性较高的方法在工业领域并不实用，因为它们采用侵入式运行方式，因此大多数客户无法接受 [4]。

其中，使用重用距离方法构建未命中率曲线（MRC）是最广泛使用的方法。对于重用距离方法，需要努力降低内存访问带来的空间开销和时间开销，并在开销与 MRC 的准确性之间取得平衡。使用内部计数器作为指标来估算工作集大小 [10]，这种方法说服力较弱，因为它没有说明使用了哪些内部计数器来获取相应信息

*同等贡献

正在使用它们来获取相应的信息。

与以往方法不同，我们提出了一种新方法，用于收集用户空间进程中的各种数据，从而更准确地评估工作集大小。为实现这一目标，我们使用**扩展伯克利包过滤器（eBPF**）监控工作程序的执行状态。eBPF 是 BPF 的扩展。*BPF* 是一种用于数据包过滤的内核架构。eBPF 将 BPF 转变为一种通用的内核虚拟机。eBPF 程序可以使用多种语言编写，例如 C语言和 Python。与其他数据收集工具相比，eBPF 工具性能卓越且不会造成干扰 [4]„ 能够以极低的开销动态监控和跟踪大量内存活动，例如页面分配、页面交换等。

具体而言，我们的工作可分为三个部分：

1) **数据收集**。如上所述，eBPF 是我们用于监控内存分配的实用工具。本文使用 Python 编写 eBPF 监控程序。成功编译后，验证器会检查程序是否合法。随后，准备好的 eBPF 程序将被加载到内核中。连接到一个称为钩子的接口后，eBPF 程序便可正常运行。eBPF 收集的数据是在一段时间内的工作集大小和缺页次数。这两项是工作集大小估算中最常用的指标。

2) **对收集到的数据进行预处理**。不同测量单位会导致特征向量的特征值存在较大偏差，这可能会对估算结果产生不利影响。为缓解这一问题，我们使用归一化方法处理收集到的数据。

3) **建模与估算**。我们使用处理后的数据建立并训练 $LightGBM$ 模型，这是一种经久不衰的机器学习模型，可更准确地预测工作集大小。为减少过拟合，我们在估算模型中添加惩罚项，这对提高模型的泛化能力至关重要。

研究结果表明，我们的方法能够准确估算且性能优异，同时具有非侵入性，平均均方根误差为0.0744，时间开销减少了65倍。总之，本文的贡献如下：

- 据我们所知，我们首次以非侵入方式估算程序执行的工作集大小。我们采用机器学习方法建立页面错误次数与实际工作集大小之间的关系。
- 我们借助扩展伯克利包过滤器，对 Linux 内核进行轻量级观测，以实现一种非侵入式缺页计数方法。
- 实验结果表明，我们提出的框架具有很高的准确性，平均均方根误差为 0.0744；同时效率也很高，估算工作集大小所需的时间开销减少了 65 倍。

II. 背景

A. 工作集大小估算

有许多论文探讨如何计算工作集大小（WSS）。此前的工作 [12] [14] 总结了几种实现这一目标的方法：

- **自气球**。 这是一种基于虚拟机的方式，根据内部计数器的值计算工作集大小。
- **Zballoning**。 它通过不同事件手动调整内部计数器的值来估算工作集大小，然后根据调整后的值估算工作集大小
- **Geiger**。 Geiger 会先分配少量内存。如果分配的内存不足，幽灵缓冲区便会发挥作用。因此，工作集大小可以表示为以下公式：

$$WSS = M_{allocated} + M_{ghost_buffer}$$

- **基于 LRU 的 MRC 构建**。它根据 LRU 栈绘制 MRC 图，可据此计算被引用页面的重用距离并估算工作集大小。

本文使用原始 WSS 估算工具[1] 获取工作集大小。该工具通过重置内存中的引用标志，并检查标志恢复的页面数量来估算工作集大小。我们将得到的工作集大小视为正确值，并据此评估训练好的模型。

B. 内存管理中的缺页

当工作集大小（WSS）不满足用户程序的需求时，就会发生缺页异常。显然，当 WSS 大于或等于程序请求的内存大小时 [7]，便会出现这种情况。因此，可以逐步增加程序请求的内存量来引发缺页，从而估算 WSS。

本文通过运行一个申请较大内存的程序，收集缺页次数及缺页发生的时间。我们改变请求的内存大小，以获得不同的数据。具体细节将在第三节中说明。

C. *eBPF* 技术

扩展伯克利包过滤器（eBPF）是伯克利包过滤器（BPF）的扩展。BPF 是一种用于数据包过滤的内核架构 [9]。eBPF 将 BPF 转换为通用的内核虚拟机。eBPF 程序可以使用多种语言编写，例如 C语言和 Python。本文使用 Python 编写 eBPF 监控程序。图 2 展示了 eBPF 的框架。eBPF 结构封装了 11 个 64 位寄存器、一个 521 字节的栈、一个映射以及一些辅助函数 [15]。

有几种结构体支持 eBPF 程序的执行：

（1）编译器。如前所述，扩展伯克利包过滤器通常使用 C 语言编写，并由后端为 LLVM 的 Clang 编译器编译。Clang 编译器按照 C 语言的规则编译扩展伯克利包过滤器程序。不过，C 语言中的某些

[1]https://github.com/brendangregg/wss

函数无法用于扩展伯克利包过滤器程序，例如*printf()*。

（2）验证器。内核中的验证器会在编译后的 eBPF 程序加载之前对其进行检查。为确保安全性，验证器会遍历整个程序，检查其是否符合内核要求。

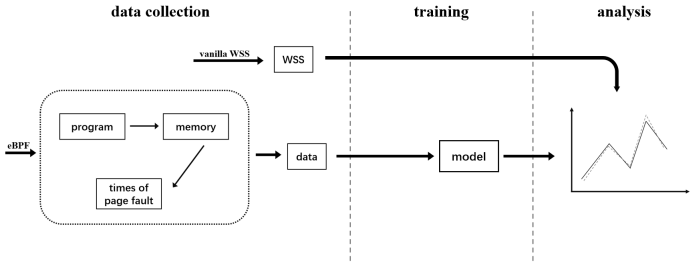


图1：总体框架

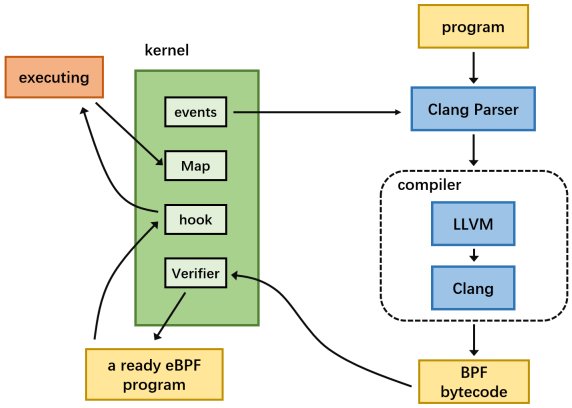


图2：扩展伯克利包过滤器的总体框架

（3）映射。映射用于存储扩展伯克利包过滤器程序的键值。键值映射会将程序映射到系统调用。扩展伯克利包过滤器程序应声明一个 *struct bpf map def* 类型的全局变量，以访问映射。每个映射由若干文件描述符组成，这些描述符会在加载过程中由文件加载器传递。验证器会将这些描述符转换为指针。

（4）辅助函数。扩展伯克利包过滤器程序的类型不仅决定其参数，还决定程序将使用哪些辅助函数，以及程序附加到哪个钩子。

（5）尾调用。扩展伯克利包过滤器程序可以通过尾调用调用其他程序。与函数调用相比，尾调用开销较低，还能简化复杂程序。尾调用需要使用一种特定的映射，称为*BPF MAP TYPE PROG ARRAY*，用于存储扩展伯克利包过滤器程序的引用。尾调用通过名为*bpftail call*的辅助函数执行。

编译成功后，验证器会检查程序是否合法。随后，准备好的扩展伯克利包过滤器程序将被加载到内核中。将其附加到名为钩子的接口后，扩展伯克利包过滤器程序便可正常运行。

III. 方法

本文通过监测缺页次数来估算工作集大小。我们准备了一些容易引发缺页的程序，方法是持续申请内存，并利用 eBPF 技术获取内存中缺失页面的数据。然后，我们使用 LightGBM 根据这些数据训练工作集大小估算模型。LightGBM 是一种可将数据训练为梯度提升决策树（GBDT）的工具。图 1 展示了总体框架。

A. eBPF 检查

当遍历到相应代码路径时，扩展伯克利包过滤器程序会被执行，这对于网络程序很有用。本文将 eBPF 技术应用于监控内存分配，并使用 eBPF 技术收集数据。要执行扩展伯克利包过滤器程序并收集目标数据，需要完成一些工作。

1) 源代码：在我们开发的扩展伯克利包过滤器程序中，我们使用了扩展伯克利包过滤器软件包中定义的若干函数和变量。图3展示了扩展伯克利包过滤器程序各部分的函数及其连接关系。

- **内核探针**。内核探针是 BPF 工具中的一种特定事件，它可以通过创建动态映射并筛选所需数据，将后续函数与内核函数关联起来。
- **BPF_哈希表(name, key_type, leaf_type, size)**。在扩展伯克利包过滤器内核中定义的函数可以创建一个哈希表*name* 来存储键值对。本文中的键值对将进程标识符与相应的缺页次数关联起来。在我们的程序中，名称设为*events*，键类型设为 u32。
- **BPF 性能输出(name)**。此函数会创建一个名为 name 的输出表，通过每个环形缓冲区传递事件数据。在我们的程序中，名称设为*events*。
- **处理内存管理故障(处理器寄存器结构体 *ctx)**。此函数在内核探针之后运行，用于记录特定事件的缺页信息并修改表计数。内置内存比较函数*()* 用于确定页面是否缺失。发生缺页时，events 和 counts 等引用变量的值会被修改。在我们的程序中，特定事件设为*myprogram*，用户可以自定义此项。
- **BPF(text = prog)**。此函数返回一个指向固定在内核中的代码的引用。我们可以通过该引用获取内核中的数据。在我们的程序中，我们将此引用赋给变量*b*，以完成后续工作。
- **打开性能缓冲区(function)**。此函数会打开一个性能环形缓冲区，用于存储由*BPF* 性能输出传递的数据。同时，它会调用作为参数传入的函数。在我们的程序中，该函数是*print event()*。
- **print event()**。该函数接收性能缓冲区中的数据，并打印缺页的信息（次数和时间）。
- **性能缓冲区轮询函数**。该函数会等待性能缓冲区中的数据，为打印下一条数据做好准备。

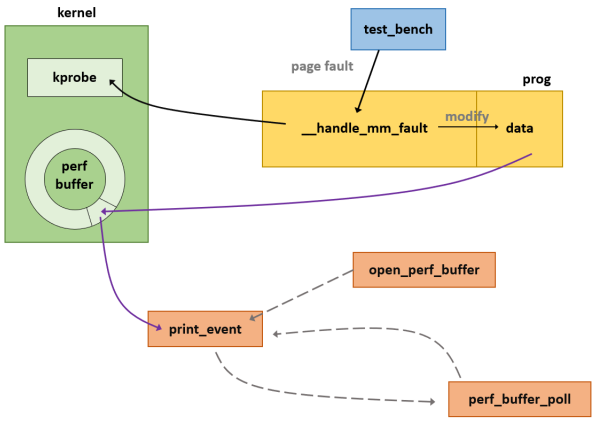


图 3：我们 eBPF 程序的实现

我们将这些函数组织起来。内核探针将函数处理内存管理故障*()* 关联到内核函数，该内核函数会将缺页信息记录在由*BPF* 哈希映射和*BPF* 性能输出创建的表中。随后，数据会被传送到每 CPU 缓冲区，并由函数*print_event()*打印。要运行我们的程序，需要完成以下几个步骤。

2）准备工作：准备阶段包括编译器和验证器。

(1) 编译器。该编译器是基于 LLVM 架构的 Clang 编译器。LLVM 为扩展伯克利包过滤器提供了一个平台，其中包括 C语言的一个子集和一些辅助函数。该子集有若干限制，例如 C语言函数的使用、变量类型以及栈的大小。

(2) 验证器。为确保内核的安全，代码编译完成后、加载到内核之前，验证器会进行工作。它会从多个方面检查程序的源代码和可执行与可链接格式：a) 验证器检查程序大小，程序指令数应少于 10^6 。b) 验证器对 eBPF 程序执行深度优先搜索。如果程序中的循环有界，则可将程序转换为有向无环图（DAG）。c) 验证器通过状态机遍历从第一条指令开始的所有可能路径，以检查深度是否低于指定大小。d) 验证器检查程序中函数的许可证，以确保其可用且兼容。e) 验证器对程序执行边界检查，以避免溢出。

3）加载：eBPF 程序在执行前必须挂接到一个钩子上。钩子是一种可由用户自定义的接口。用户可以在钩子中注册用于处理若干事件的程序。eBPF 程序所挂接的钩子由其类型决定。当发生已注册的事件时，eBPF 程序便会执行并跟踪目标数据。本文讨论的这些事件与缺页异常有关。

4) 执行：经过准备和加载阶段后，eBPF 程序便可执行并监控目标数据。它会在特定事件发生时运行。在此过程中，我们前面提到的映射、辅助函数和尾调用会发挥作用，以完成数据收集任务。

(1) 映射 如图3所示，eBPF 程序会创建一个映射，用于存储进程标识符和缺页次数的键值。执行时，程序会修改该映射中的数据，以记录我们需要的数据。

映射的生命周期由内核中的引用计数器（refcnt）决定。当进程创建一个新映射时，内核会将该映射的引用计数设为 1。该计数器用于统计 eBPF 程序使用该映射的数量。如果 refcnt 达到 0，内核就会销毁与该计数器相关的 eBPF 程序，这意味着映射的生命周期结束。

为了使映射保持有效，可以将其固定到内核中的文件系统。这种方法称为映射表固定，可以通过多种工具实现，例如 *bpf()* 系统调用和*libbpf* 。

（2）辅助函数eBPF 程序可以调用内核中的辅助函数来执行相应任务，这是扩展伯克利包过滤器与 cBPF 之间的区别之一。这些任务如下：a) **交互**。辅助函数可以与钩子上下文以及内核中的结构体（例如映射）进行交互。b)**修改**。辅助函数可以修改进程中的数据。c) **打印**。辅助函数可以打印目标数据，以帮助用户监控程序的执行状态。

辅助函数的声明大多位于*bpf helpers.h*中。在编译过程中，编译器只需编译包含这些函数签名的头文件，而不必编译它们对应的 .c 文件，因为这些函数的实现由内核提供。

B. 预测模型设计

LightGBM 是一个用于实现 GBDT 算法的高效框架。GBDT 是一种基于提升的优秀模型。在提升算法中，为了提高学习模型的效果并弥补前一个模型的错误， y_i 的输入不仅包括输入序列 x，还包括上一个训练完成的模型的训练误差 y_{i-1}。它可以表示为以下函数：

$$H(x) = \sum_{i=1}^{N} \alpha_i h_i(x)$$

其中， $h_i(x)$ 是使用部分数据训练得到的单个分类器。at 是相应错误的权重。 α_i 的表达式为：

$$\alpha_i = \frac{1}{2} ln \frac{1-\epsilon_i}{\epsilon_i}$$

其中， ϵ_i 为：

$$\epsilon_i = P_{x \in D_i}(h_i(x) \neq y)$$

之前模型的误差可以通过权重相加来修正。LightGBM 基于直方图算法优化 GBDT，该算法可将连续值离散化为 k 个整数，并创建宽度为 k 的直方图。处理连续值时，会将这些值映射到直方图中，然后通过直方图搜索最佳分割点。LightGBM 还可以通过两个节点相减，轻松得到少量叶节点

，以计算直方图的差值。本文使用 LightGBM 获得训练好的模型。模型的输入是缺页次数以及缺页发生的时间。

IV. 实验

A. 实验设置

本文的实验环境基于 Linux 5.13.0，我们借助 bcc v0.23.0 使用 Python 3.7 编写 eBPF 程序，并通过 LightGBM 3.3.2 训练模型。

B. 数据收集

我们将 eBPF 程序插入内核态，利用钩子捕获缺页，以收集测试程序发生缺页的次数。

1) 数据记录：

• **哈希表的建立**为了记录所观察进程的缺页次数，我们通过辅助函数 $BPF\ HASH()$ 建立一个名为 *counts* 的哈希表，其中每个条目的键是进程的进程标识符，对应的值是缺页次数。我们还建立了另一个名为 *frq* 的哈希表，用于记录缺页发生的频率。当达到阈值 $frequency$ 时（在实际情况下，该阈值将动态调整），数据将提交到稍后讨论的 perf 环形缓冲区。如果样本成功写入存储文件，*counts*将被清空。

• **时间记录** 我们定义了一个全局变量 *begin time*，用于监控 eBPF 程序的执行时间，之后将用它来计算收集每个数据样本所需的时间。

•**系统调用钩子** 我们使用 $kprobe$（一种 eBPF 程序）来钩住系统调用函数 处理内存管理故障，即函数 $int\ kprobe\ handle_mm_fault(structpt\ regs * ctx)$。每当进程调用该系统调用时，程序都会先进入预设的 eBPF 程序进行数据收集，然后再由处理函数处理。

• **eBPF 程序中的数据操作**当进程陷入预设的 eBPF 程序时，程序会捕获进程标识符和进程名称，以检查其是否为被观测的进程。如果是，哈希表中缺页次数的值 *counts* 将自动递增。同时，*frq* 的值也会递增，以记录缺页频率。

2）数据交互：

• **建立 perf 环形缓冲区** 为了实现从内核空间到用户空间的数据交互，我们使用 BPF 表，通过调用辅助函数 $BPF\ PERF\ OUTPUT()$，经由名为*event* 的 perf 环形缓冲区推送数据。

• **输出数据**Linux 内核提供了许多用于操作 perf 环形缓冲区的辅助函数。在本文中，我们使用辅助函数 $perf\ buffer\ poll()$ 和 $open\ perf\ buffer$来输出数据。每次调用 $perf\ buffer\ poll()$ 时，都会调用附加到 $open\ perf\ buffer$每个条目的回调函数。我们将回调函数重新命名为打印事件，以输出数据。在打印事件函数中，考虑到数据稳定性，我们舍弃了小于 $min\ value$ 的值，然后将符合条件的数据连同时间戳输出到用户空间，并重定向至存储文件。

3）*WSS* 基准真值近似值： 本文使用 [3] 提供的工具，在收集缺页的同一时间段内测量工作集大小（WSS）。目标测试程序会创建一个长度可选的整数数组，范围为 $[2^7, 2^{22}]$, ，并在无限循环中为每个元素赋值。程序将持续运行，直到达到设定的时间阈值，以确保 Linux 能够承受其消耗的内存。

我们收集了 20 多个数据集，其中大多数包含 1200 多个样本。将估算出的近似工作集大小标记为缺页次数的标签后，即可训练预测模型，利用缺页次数估算未来的工作集大小，具体内容将在下文讨论。

C语言. 模型构建

1) 数据预处理：为确保模型的稳健性和稳定性，我们从收集的数据中剔除异常数据。此外，数据收集记录了测试程序在特定时间段内的缺页次数。为了更好地训练模型，我们需要根据公式计算各时间段的时间间隔：$\Delta t_i = t_{i+1} - t_i$此外，我们还进行数据归一化，以提高模型的收敛速度和准确性。当存在多指标评价时，通常需要进行归一化，因为不同评价指标的性质、维度和数量级各不相同。如果指标维度不同，直接使用原始指标进行评价时，数值较大的指标将在综合评价中占据主导地位，从而削弱数值较小的其他指标的作用。本文使用两个指标，即时间间隔和缺页次数。因此，我们使用归一化将收集的数据映射到 $[0, 1]$ 的范围内。归一化表达式为：

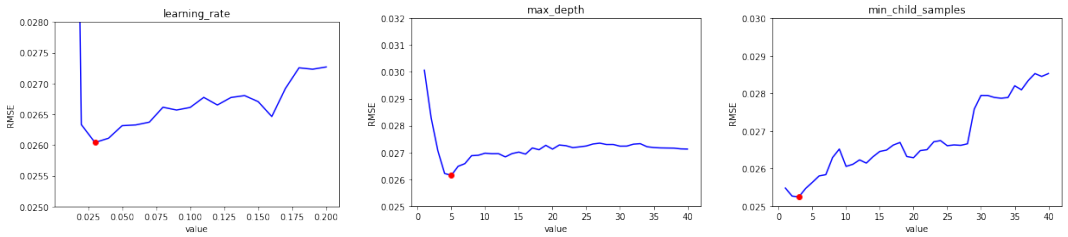
$$X_{standard} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

本文使用 Python 3.7 提供的函数$MinMaxScaler()$进行归一化。

2）敏感性分析：LightGBM 算法提供了用于训练和优化模型的可选参数 [1] ，我们分析了其中许多参数，以找出超参数。它们如下所列：

• **学习率**：用于控制深度学习模型的进展，其值应大于零。

• **最大深度**：决策树的最大深度。它用于限制决策树的深度。其值应为正整数。



(a) 学习率的均方根误差 (b) 最大深度的均方根误差 (c) 最小子节点样本数的均方根误差

(d) 叶子节点数的均方根误差 (e) 估计器数量的均方根误差 (f) 按树列采样的均方根误差

图4：超参数分析结果

- **叶子节点数**：单个决策树的最大叶子节点数。其值应为正整数。
- **估计器数量**：本文采用提升算法。该参数表示提升迭代次数，应为正整数。
- **最小子样本数**：单个叶节点中的最小数据量，应为正整数。
- **随机状态**：用于生成其他随机种子。由于其优先级低于其他随机种子，默认情况下不使用该种子。其值设为 $[0, 10^6$ 范围内的随机值）。
- **按树列采样**：特征选择的百分比。默认值为 1，表示选择所有特征。设为小于 1 的值时，每个树节点都会选择部分特征。其值应大于 0 且小于或等于 1。
- **子采样**：用于随机选择数据，不进行重采样。其值应大于 0 且小于或等于 1。

我们尝试通过在 NNI [2]，（即神经网络智能）上运行程序，找出每个超参数的最优区间和次优区间。NNI 是微软提供的一款用于自动调参的工具。图 4a-4f 展示了各超参数对应的均方根误差（RMSE）。从图 4a-4f 可以轻易得出结论：几乎每张图都呈现出一条微笑曲线，显示出一个最优区间和一个次优区间。然而，子采样的曲线表明，其取值对 RMSE 没有影响，因此可以忽略其对应的图。

根据各均方根误差图得出的每个超参数的最优区间和次优区间列于表 I。对于学习率，较大的取值会使我们的模型收缩，从而获得局部最优结果，而非全局最优结果。与此同时，较小的值会大幅增加模型训练的时间开销。对于最大深度，较大的值会导致过拟合问题，而较小的值则会导致模型欠拟合，使预测值不准确。对于估计器数量，较大的值会增加模型训练时间，而较小的值会导致准确性不足。对于最小子样本数，选择合适的值有助于防止过拟合。叶子节点数在一定程度上决定了树的复杂度；较小的值会导致欠拟合，而较大的值会导致过拟合。至于按树列采样，它可用于加快训练速度，合适的值也有助于解决过拟合问题。显而易见，不恰当的超参数取值会对模型乃至整个工作造成灾难性影响。当每个超参数的取值都处于其最优区间时，预测结果会达到最佳，低估值和高估值也会较少；而处于次优区间时，可能会出现一些异常峰值。更糟糕的情况是，显而易见，不恰当的超参数取值会对模型乃至整个工作造成灾难性影响。当每个超参数的取值都处于其最优区间时，预测结果会达到最佳，低估值和高估值也会较少；而处于次优区间时，可能会出现一些异常峰值。当取值落在最优区间或次优区间之外时，情况最为糟糕：大多数预测值都会严重偏离真实值，模型也会剧烈振荡。因此，经过数百次尝试后，我们训练所得模型的超参数取值如下：学习率=0.04，最大深度=5，叶子节点数=255，估计器数量=200，最小子样本数=3，

表 I：最优区间和次优区间

参数	最优区间	次优区间
学习率	$[0.0245, 0.0425]$	$[0.023, 0.0245) \cup [0.0425, 0.073]$
最大深度	$\{4, 5\}$	$\{3, 6, 7\}$
叶子节点数	$[25, 300]$	$[8, 23]$
估计器数量	$[100, 200]$	$\{99\} \cup [201, 310]$
最小子样本数	{2,3}	{1,4}
按树列采样	$[0.75, 1]$	/

按树列采样=0.999，子采样=0.57。这些超参数保证了模型的高准确性。

3) 模型训练：为了训练模型，我们将预处理后收集的数据分为六组，分别是$train\ X$、$train\ Y$、$valid\ X$、$valid\ Y$、$test\ X$和$test\ Y$。具体来说，它们是训练集、验证集和测试集。模型从训练数据中学习并不断改进和优化自身，根据输入验证集时的结果来获得更好的预测结果。我们手动调整每组数据的大小，以使模型达到最佳效果。回归模型通过 LightGBM 进行训练。用于设置模型的超参数已在上文讨论。

D. 工作集大小估算

为了验证我们工作的准确性，我们使用从测试程序中收集的 4 个测试数据集来测试模型。每个测试数据集的样本数分别为 444、458、561 和 621。此外，我们使用测试程序模拟特定进程向操作系统请求内存的过程，这一点此前已有介绍。每个测试程序创建的数组大小分别为 200、1024、1024 和 25，范围从少量内存到大块内存。结果见图 5a–5d。各测试数据集的均方根误差统计结果列于表 II。我们的模型平均均方根误差为 0.0744，准确性极高。

此外，在估算的时间开销方面，

表 II：测试数据集的均方根误差

	测试数据集索引			
	数据集 *1*	数据集 *2*	数据集 *3*	数据集 *4*
大小[a]	444	458	561	621
RMSE	0.0014	0.0873	0.0291	0.1799

[a]数据集中的样本数量。

工作集大小方面，与 WSS 工具相比，我们的模型展现出压倒性的优势 [3]。如图 6 所示，在我们的测试程序中运行时，WSS 工具平均需要 3.785 秒才能输出估算结果，而我们的扩展伯克利包过滤器程序平均仅需 0.0581 秒即可输出结果。我们的模型只用了 WSS 工具所耗时间的1_{65}，大幅降低了时间开销。

V. 相关工作

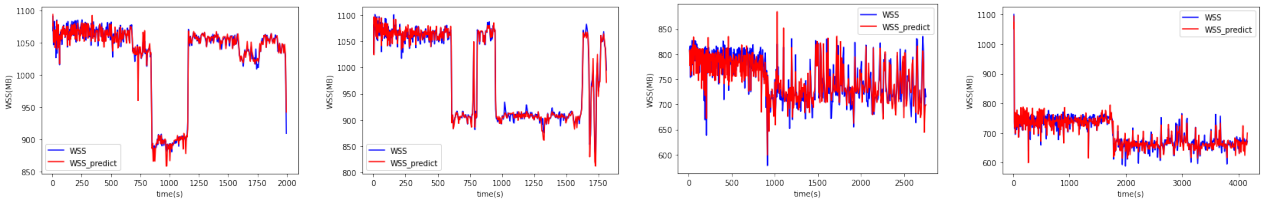
多篇论文致力于以较低的开销和较高的准确性估算工作集大小。重用距离方法是构建内存重用特征曲线时最常用的方法。为构建大页的内存重用特征曲线， [8] 提出了通过调整动态热集来减少大页缺页次数的方案。保留在热集中的大页意味着它们的重用距离较小，因此在跟踪内存访问以计算重用距离时，可以将其排除。为了保证内存重用特征曲线的准确性，作者提出了一种理论启发式算法，用于计算被过滤的重用距离并将其加回。 [16]介绍了一种名为 SHARD 的算法，该算法通过均匀随机的空间采样在线构建内存重用特征曲线。作者提出了 SHARD 的两个版本：第一个版本侧重于固定采样率，第二个版本则侧重于固定大小的样本集 S，即用于存储被引用页面信息的先入先出队列。该算法使用被引用页面的哈希值来决定是否对页面进行采样。两者的区别在于页面被插入S时的处理方式。若 S 已满，第二个版本会淘汰哈希值最大的页面。该算法致力于降低内存访问所带来的空间开销和时间开销。

利用内部计数器也是一种更可取的方法。 [5] [7] 在 Linux 上利用内部计数器 $Committed\ AS$ 。 [5] 设计了一种有限状态机，以动态检测 $Committed\ AS$ ，并在必要时修改其值。该方法提出了三种状态：快速状态、冷却状态和慢速状态。初始时，有限状态机处于快速状态，在此状态下，$Committed\ AS$ 的值每秒降低 5%。每当发生缺页，下一状态便切换为冷却状态，并启动一个时长为 8 秒的计时器。只要 8 秒内发生缺页，系统就会保持在冷却状态。计时器归零后，下一状态转为慢速状态。当 $Committed\ AS$ 的值发生变化时，下一状态将变为快速状态，并将工作集大小估算修改为 $Committed\ AS$的精确值。 [7]提出了另一种与前一种类似的有限状态机，但增加了恢复状态。初始状态为快速状态；不过，每当 $Committed\ AS$ 发生变化，且上一状态不是恢复状态时，系统便切换到恢复状态。在恢复状态下，如果 $Committed\ AS$增加超过 5%，系统便切换到快速状态；否则切换到慢速状态。一旦 $Committed\ AS$降低，系统就会逐步调整，直至恢复稳定。发生缺页时，系统会像前一种有限状态机一样切换到冷却状态。然而，有限状态机的应用需要考虑得更加细致，才能涵盖所有复杂情况。

据我们所知，我们是首个借助 eBPF 工具监控内存活动的研究。我们的工作使用 eBPF 工具收集缺页次数，并利用 LightGBM 预测未来的工作集大小。

VI. 结论

本文提出了一种新颖的非侵入式框架，能够以极高的效率估算工作集大小。据我们所知，这是首项采用非侵入式技术估算进程工作集大小，同时保证低开销的工作。此外，我们对收集到的数据进行预处理



(a) 测试 1 的输出　(b) 测试 2 的输出　(c) 测试 3 的输出　(d) 测试 4 的输出

图 5：4 个数据集的工作集大小估算性能

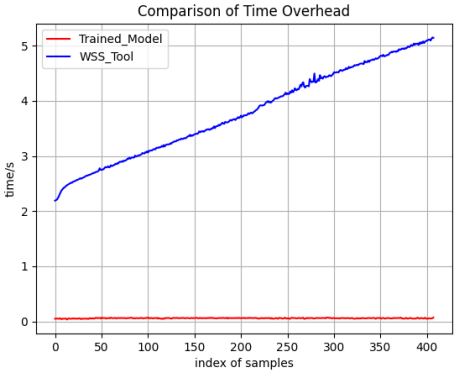


图 6：时间开销比较

并使用 LightGBM 训练模型，以准确估算工作集大小。实验结果表明，与以往工作相比，我们的模型将开销降低了 65.2 倍。同时，均方根误差仅为 0.0774，表明模型具有较高的估算准确性。

VII. 致谢

本研究得到广东省重点领域研发计划（编号：2020B010165002）的支持。

参考文献

[1] LightGBM 中所有参数的说明。https://lightgbm.readthedocs.io/en/latest/Parameters.html。访问日期：2022 年 4 月 6 日。[2] 神经网络智能。https://nni.readthedocs.io/en/latest/。访问日期：2022 年 4 月 6 日。[3] BrendanGregg。如何测量 Linux 上的工作集大小。https://www.brendangregg.com/blog/2018-01-17/measure-working-set-size.html。访问日期：2022 年 4 月 6 日。[4] Cyril Cassagnes、Lucian Trestioreanu、Clement Joly 和 Radu State。扩展伯克利包过滤器在非侵入式性能监控中的兴起。载于*NOMS 2020-2020 IEEE/IFIP* 网络运营与管理研讨会，第 1–7 页。IEEE，2020。[5] Jui-Hao Chiang、Han-Lin Li 和 Tzi-cker Chiueh。基于工作集的物理内存气球机制。载于 Jeffrey O. Kephart、Calton Pu 和 Xiaoyun Zhu 编，第十届国际自主计算会议（*ICAC'13*），美国加利福尼亚州圣何塞，*2013* 年 *6* 月 *26–28* 日，第 95–99 页。USENIX 协会，2013。[6] PVSS Gangadhar、M Venkateswara Rao、Ashok Kumar Hota 和 V Venkateswara Rao。云计算环境中的分布式内存与 CPU 管理。国际应用工程研究期刊，12(24):15972–15978，2017。

[7] Ahmed A. Harby、Sherif Fadel Fahmy 和 Ahmed F. Amin。虚拟机中工作集大小的更准确估算。*IEEE Access*，7:94039–94047，2019。

[8] Jingyuan Hu、Xiaokuang Bai、Sai Sha、Yingwei Luo、Xiaolin Wang 和 Zhenlin Wang。虚拟化环境中使用大页估算工作集大小。载于 Jinjun Chen 和 Laurence T. Yang 编，*IEEE* 并行与分布式处理及应用、泛在计算与通信、大数据与云计算、社会计算与网络、可持续计算与通信国际会议，*ISPA/IUCC/BDCloud/SocialCom/SustainCom 2018*，澳大利亚墨尔本，*2018* 年 *12* 月 *11—13* 日，第 501–508 页。IEEE，2018。[9] Steven McCanne 和 Van Jacobson。BSD 分组过滤器：一种用于用户级分组捕获的新架构。载于*Usenix 1993* 冬季技术会议论文集，美国加利福尼亚州圣迭戈，*1993* 年 *1* 月，第 259–270 页。USENIX 协会，1993。[10] Anna Melekhova 和 Larisa Markeeva。利用客户操作系统性能计数器估算工作集大小。*CLOUD Comput*，48 38，2015 33：–。[11] Peter Mell、Tim Grance 等。美国国家标准与技术研究院云计算定义。2011。[12] Vlad Nitu、Aram Kocharyan、Hannas Yaya、Alain Tchana、Daniel Hagimont 和 Hrachya V. Astsatryan。虚拟化环境中的工作集大小估算技术：并非一刀切。*Proc. ACM Meas. Anal. Comput. Syst.*，2(1):19:1–19:22，2018。[13] Frederic Nzanywayingoma 和 Yang Yang。云计算中资源管理技术、问题与挑战的文献综述。*TELKOMNIKA*（电信、计算、电子与控制），15(4):1918–1928，2017。[14] Sai Sha、Jingyuan Hu、Yingwei Luo、Xiaolin Wang 和 Zhenlin Wang。大页友好的虚拟化内存管理。*J. Comput. Sci. Technol.*，35(2):433–452，2020。[15] Marcos Augusto M. Vieira、Matheus S. Castanho、Racyus D. G. Pac´ıfico、Elerson Rubens da Silva Santos、Eduardo P. M. C^amara J´unior 和 Luiz Filipe M. Vieira。使用扩展伯克利包过滤器和 XDP 实现快速分组处理：概念、代码、挑战与应用。*ACM Comput. Surv.*，53(1):16:1–16:36，2020。[16] Carl A. Waldspurger、Nohhyun Park、Alexander T. Garthwaite 和 Irfan Ahmad。使用 SHARDS 高效构建 MRC。载于 Jiri Schindler 和 Erez Zadok 编，第 *13* 届 *USENIX* 文件与存储技术会议论文集，*FAST 2015*，美国加利福尼亚州圣克拉拉，*2015* 年 *2* 月 *16—19* 日，第 95–110 页。USENIX 协会，2015。