计算机科学
COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240800023

基于系统追踪与模式挖掘的缺页异常性能分析方法研究

魏书文　　王宝会
北京航空航天大学软件学院　北京 100191
(weishuwen@buaa.edu.cn)

摘　要　缺页异常处理是 Linux 内核内存管理子系统中的核心模块,对于数据库等内存密集型应用,其处理效率直接影响全系统性能,常常成为性能瓶颈。优化内存使用和减少缺页异常发生已经有大量的研究工作,但在实际环境下缺页异常的处理模式及其延迟分布却鲜有人关注。文中提出了一种基于执行跟踪的方法,用于分析缺页异常的处理模式及其延迟分布,通过该方法,提出了两种经典内存密集型负载的缺页异常的处理模式及其延迟分布,为优化内存密集型应用提供了重要参考。

关键词:性能分析;系统追踪;缺页异常;操作系统
中图分类号　TP316

Study on Performance Analysis Methods for Page Faults Based on System Tracing and Pattern Mining

WEI Shuwen and WANG Baohui
School of Software,Beihang University,Beijing 100191 China

Abstract　Page fault handling is a core module in the memory management subsystem of the Linux kernel. For memory-intensive applications such as databases, the process efficiency will directly impact the overall system performance and often becomes a performance bottleneck. There has been extensive research on optimizing memory usage and reducing the occurrence of page faults. However, the handling patterns of page faults and their latency distribution in real-world environments have received little attention. This paper proposes a method based on execution tracing to analyze the handling patterns and latency distribution of page faults. With this method, the handling patterns and latency distribution of page faults under two classic memory-intensive workloads is studied, which providing important references for optimizing memory-intensive applications.

Keywords　Performance analysis,System tracing,Page fault,Operating system

1　引言

在现代操作系统中,内存管理是确保系统性能和资源有效利用的关键组成部分。缺页异常[1]处理作为一个重要机制在内存管理中扮演着至关重要的角色。图 1 给出了 CPU 访存的过程,当 CPU 尝试访问内存时,首先会通过 MMU 将虚拟地址转换为物理地址,MMU 会先查找 TLB 以确定是否存在虚拟地址到物理地址的快速映射。如果 TLB 命中,则直接进行数据访问;如果 TLB 未命中,则 MMU 需要通过查找页表来进行地址转换。如果在页表中未找到有效映射,或因为权限问题无法访问页面,便会触发缺页中断。缺页异常处理的效率直接影响系统的响应时间和性能,尤其是对于数据库、文件处理和大规模计算应用等内存密集型应用,因为这些应用需要频繁地处理大量缺页异常。在这类应用中,精确地识别、刻画和分析缺页异常的行为细节,显得尤为重要。

目前,大部分研究主要关注于降低缺页异常的发生频率[2]和处理开销[3],以及通过高级内存管理策略[4]、替换算法[5]和硬件优化技术来提高系统的整体性能和响应速度。然而,对于不同的内存密集型应用,缺页异常在内核层面实际处理模式和延迟分布的研究相对较少。此外,现有的系统级追踪工具虽能提供大致的性能概览,但在细化分析方面仍显不足。因此,本文提出了一种性能分析方法,旨在从系统跟踪日志中挖掘缺页异常处理的具体模式,从而识别并定位潜在的性能瓶颈,并通过详尽的系统性能分析,为性能优化提供更加精确的指导。

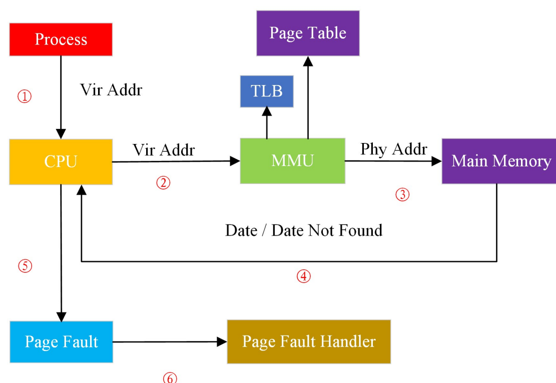


图 1　CPU 访存流程示意图

Fig.1　CPU memory access and page faulttrig gering

为了从大量、复杂并且包含众多错误和冗余信息的系统追踪日志中挖掘出有效的缺页异常的处理模式和延迟分布,本文提出了一个高效分析框架,如图 2 所示。在这个框架中,我们采用了一种基于函数调用层级的逐层分析方法,这种策

通信作者:王宝会:(wangbh@buaa.edu.cn)

略可以构建一个全面的性能影响图景，每一层的分析都为下一层提供了上下文和数据支持，并且随分析的深入可以逐步揭示性能问题的根源。此外，在数据预处理阶段，根据每层的依赖关系来动态地调整数据处理的范围。这种动态预处理策略大大提升了数据处理的灵活性和针对性，从而优化了分析流程的效率，并增强了结果的准确性和可靠性。

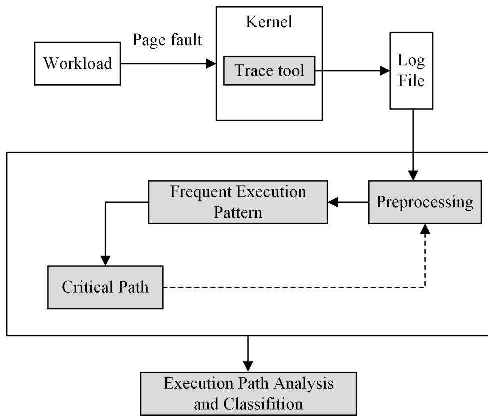


图 2 缺页异常性能分析框架

Fig. 2 Automated log file analysis framework

2 相关工作

系统级分析工具[6]在系统调优[7]和性能分析领域具有重要作用，可以帮助开发人员寻找性能瓶颈和优化机会。作为一个轻量级而功能强大的跟踪工具，ftrace[8]专注于 Linux 内核的细节，能够跟踪各类内核函数。通过使用 ftrace，研究人员可以详细观察到内核操作的执行过程，从而有效地揭露和定位潜在的性能问题。然而，ftrace 输出数据庞大且复杂，使得解析和分析过程既耗时又容易出错。

在相关研究中，Sauradip Ghosh[9]的研究提供了对如何利用软件跟踪技术实时分析并识别系统性能问题的深刻见解。通过广泛回顾并建立性能分析模式的目录，Ghosh 的工作强调了系统化方法在识别和解决运行时性能问题中的重要性。Christopher LaRosa 等[10]开发了一个分析内核追踪日志的框架，主要通过应用频繁模式挖掘技术来识别操作系统中的进程间通信和运行时行为模式。尽管该研究提供了一种有效的方法来分析内核操作，但其关注重点为内核日志中的一般模式识别，未专门针对操作系统特定子系统进行深入研究。

本文进一步聚焦于 Linux 内核内存管理子系统[11]中的一个特定问题——缺页异常的执行模式及其延迟分布。通过提出一个自动化分析框架，自动从内核跟踪日志中分析缺页异常的发生模式和延迟，以提供更为精确的性能优化建议。

3 研究方法

本文的核心目的是通过分析内核跟踪日志，揭示缺页异常处理的关键模式及其延迟分布。我们重点关注了日志中频繁出现的函数调用序列，并采用了高效的序列模式挖掘方法来提取这些信息。本章详细介绍了数据收集、预处理及分析方法，旨在确保所得结果准确反映系统的性能特征，进而揭示性能瓶颈的根本原因。

3.1 数据收集与预处理

本文的目标是揭示内核中缺页异常的实际处理模式，即内核层面的函数调用栈。为此，我们选择了 ftrace 作为追踪工具，并采用 function_graph 作为追踪器。function_graph 追踪器擅长捕捉内核函数的调用序列和嵌套结构，提供了一个关于函数执行时间和调用层次的详细视图。通过 function_graph 收集的原始数据以文本格式存储。图 3 给出了 function_graph 结果文件的格式。

```
4) redis-b-2330522 |               |  __do_page_fault() {
4) redis-b-2330522 |   0.063 us    |    fault_in_kernel_space();
4) redis-b-2330522 |   0.061 us    |    down_read_trylock();
4) redis-b-2330522 |               |    find_vma() {
4) redis-b-2330522 |   0.061 us    |      vmacache_find();
4) redis-b-2330522 |   0.178 us    |    } /* find_vma */
4) redis-b-2330522 |               |    handle_mm_fault() {
```

图 3 function_graph 追踪结果示意图

Fig. 3 Function_graph tracing results

3.2 数据分析

3.2.1 频繁模式识别

在内核跟踪日志这样的顺序序列数据中查找频繁出现的模式是一项挖掘任务，称为顺序模式挖掘[12]。设 S 为序列数据库，σ 为最小支持度阈值，频繁序列 p 的支持度为 $sup(p)$，则 p 是频繁的当且仅当：

$$sup(p) \geqslant \sigma \tag{1}$$

本文采用了 PrefixSpan 算法[13]来实现频繁序列的挖掘。该算法是由 Pei 等于 2001 年提出的，其主要目的是从大量序列数据中找到频繁出现的序列模式。与基于候选生成的传统算法（如 Apriori 算法）相比，PrefixSpan 减少了候选序列的生成数量，从而提高了挖掘效率。

假设有以下序列数据库，如表 1 所列。

表 1 原始序列

Table 1 Original sequences

SID	Sequences
S	$\langle abcd \rangle$
S	$\langle cabc \rangle$
S	$\langle acbac \rangle$

如上述序列中“a”是频繁项，对于频繁项“a”，构建如下“投影数据库”，如表 2 所列。

表 2 投影序列

Table 2 Projected sequences

SID	Sequences
S	$\langle bcd \rangle$
S	$\langle bc \rangle$
S	$\langle cbac \rangle$

该数据库包含“a”后面的所有子序列。随后，算法对每个投影数据库递归重复此过程，每次递归都会将一个元素添加到当前的频繁序列前缀中，直至不再发现新的频繁序列。

考虑到序列数据库中数据的时间连续性这一特性，本文对基础的 PrefixSpan 算法进行了必要的优化。鉴于每个序列的首个元素准确标记了函数调用的初始点，优化策略侧重于对这些首元素频率的精确分析。此方法显著减少了冗余投影数据库的生成，此改进保持了 PrefixSpan 的核心原理，同时通过简化数据处理流程，使算法更适应序列数据的特定需求，从而提高了分析的针对性和实用性。

算法 1 Optimized PrefixSpan

1. Procedure OptimizedPrefixSpan(S, min_sup, Prefix=[])
2. Input: Sequence database S, minimum support threshold min_sup
3. F ← empty list

```
4. Counter ← new HashMap()
5. for each sequence seq in S do
6. first_item ← first element of seq
7. Counter[first_item] ← Counter[first_item] + 1
8. end for
9. for each item, count in Counter do
10. if count >= min_sup then
11. F.append(item)
12. end if
13. end for
14. for each item in F do
15. NewPrefix ← Prefix + [item]
16. Output NewPrefix
17. ProjectedDatabase ← BuildProjectedDatabase(item, S)
18. if not empty(ProjectedDatabase) then
19. call OptimizedPrefixSpan(ProjectedDatabase, min_sup, NewPrefix)
20. end if
21. end for
22. End Procedure
23. // Build the projected database
24. Procedure BuildProjectedDatabase(item, S)
25. ProjectedDatabase ← empty
26. for each sequence seq in S where item is the first element do
27. Rest ← the part of seq after item
28. if not empty(Rest) then
29. Projected Database.append(Rest)
30. end if
31. end for
32. return Projected Database
33. End Procedure
```

3.2.2 贡献最大函数

一个内核函数的执行开销主要由其本身的指令开销以及函数调用栈内各部分的开销构成，但通常会有对总开销贡献最大的部分。在完成频繁项集挖掘后，我们通过分析序列中各元素执行时间及其方差来识别影响总执行时间波动的关键因素，以便深入探索性能问题的根本原因。

4 实验

4.1 实验设置

硬件平台如下：实验在单处理器的 intel 机器上进行，机器具体为 11th Gen Intel(R) Core(TM) i7-11700F @ 2.50GHz，具有 8 核心 16 线程，64 GB DDR4 内存，和 512 GB SSD 存储。该机器为单节点，排除了 NUMA 的影响。

软件如下：操作系统为 UOS Server 20 1060a，内核版本为 4.19。

测试负载如下：评估包括两种典型应用负载，即 Redis 和 MySQL。测试的数据库系统分别是 Redis 5.0.5 和 MySQL 8.0，运行在默认配置下。实验负载分别采用 redis-benchmark[14] 进行标准测试，Redis-benchmark 是 Redis 提供的性能测试工具，它可以通过执行一系列预定义的命令来模拟客户端请求，并生成报告，显示每秒能处理的请求数（TPS）、每个请求的平均延迟等信息。以及使用 TPC-C[15] 基准测试模拟 MySQL 的数据库操作。TPC-C 是一个标准的数据库性能测试基准，它模拟了一个完整的订单处理系统，其中包含了商品的入库、订单的生成、支付处理、订单状态查询和库存管理等多种交易类型。TPC-C 通过模拟真实的业务环境来评估数据库管理系统（DBMS）的性能和吞吐能力。

4.2 执行路径分析

图 4 给出了我们挖掘到的缺页异常的几种主要执行模式。由于从__do_page_fault 函数开始跟踪，因此将其定位为第一分析层。接下来，将按照从整体性能到局部性能的顺序对缺页异常的性能特征进行逐层分析。

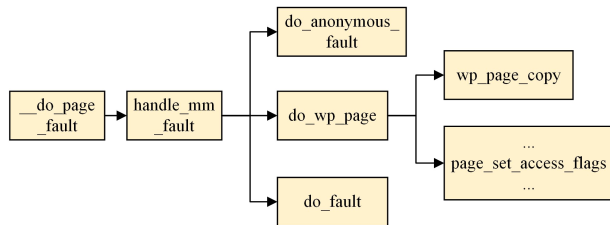
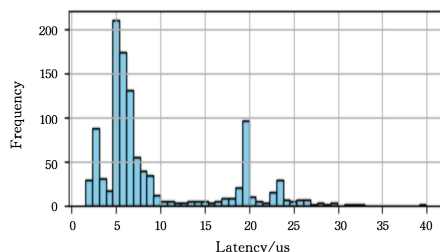


图 4　缺页异常处理模式示意图

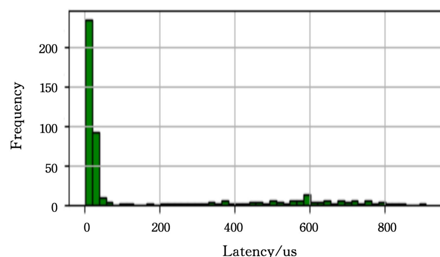
Fig. 4　Page fault handling pattern

4.2.1 __do_page_fault 函数

图 5 给出了__do_page_fault 函数开销的分布，在缺页异常处理的分析中，将__do_page_fault 函数视为第一层。这层是整个处理流程的起点，它直接对应于操作系统内核在遇到页面错误时的首个响应点。当一个进程访问无效或未映射的内存页时，便会触发__do_page_fault 函数的调用。通过对该层的分析可以了解页面错误处理的初步复杂度和可能的性能瓶颈。



(a) Distribution of redis latency durations



(b) Distribution of MySQL latency durations

图 5　__do_page_fault 函数开销分布图

Fig. 5　__do_page_fault function overhead distribution

图 5 中，redis-benchmark 数据的主要分布集中于 0～40 的范围内，并在此区间内形成了 3 个显著的峰值，分别位于 0～5 区间，5～10 区间和 15～25 区间，表明在这些范围内有较强的趋势或模式存在。

值得注意的是，在 TPC-C 的结果图中数据不仅在 0～40 的区间内分布，而且在 200～1000 的较宽范围内也呈现出明显的分布，大约占总数据的四分之一，这说明在 TPC-C 负载下可能存在额外延迟。

通过对比这两种负载下各自的执行模式发现，在 TPC-C

负载下，调用栈中多了一部分 down_read()函数的调用，当一个线程调用 down_read() 时，它请求读取访问权限。如果有写操作正在进行(或等待进行)，这个调用会阻塞，直到写操作完成。在对跟踪输出文件的上下文分析中，我们观察到该文件记录了多次调度事件，涉及到 TPC-C 的多个线程。在频繁的调度和缺页异常中，我们发现部分开销明显异常，特别是在调用 down_read() 过程时，其开销显著增加。图 6 给出了 down_read()函数开销分布，这基本解释了 200～1000 区间内分布的根源。

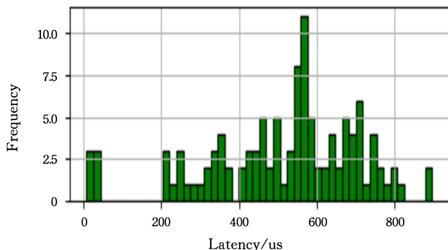


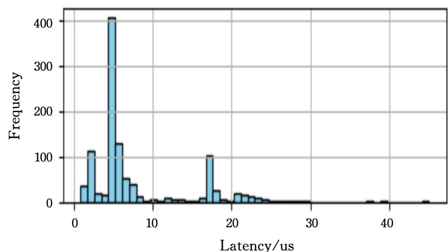
图 6 down_read 函数开销分布图

Fig. 6 down_read() function overhead distribution

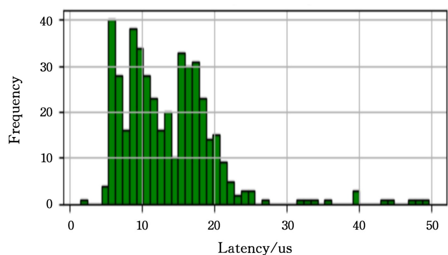
4.2.2 handle_mm_fault 函数

进一步分析发现，这一部分性能异常的根本原因在于 MySQL 的多线程竞争资源。由于多个线程同时争夺访问权限，导致频繁的调度和锁竞争，因此引发性能波动和异常行为。

图 7 给出了 handle_mm_fault()的开销分布，handle_mm_fault()是处理实际内存错误的核心函数，它会根据错误的类型和位置采取不同的措施。可以发现数据分布的形状与图 5 中的分布大体一致。这也表明，handle_mm_fault()是影响上一层分布的主要因素。



(a) Distribution of redis latency durations



(b) Distribution of MySQL latency durations

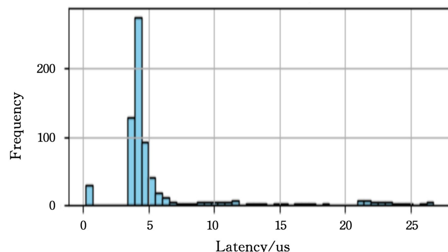
图 7 handle_mm_fault 函数开销分布图

Fig. 7 handle_mm_fault function overhead distribution

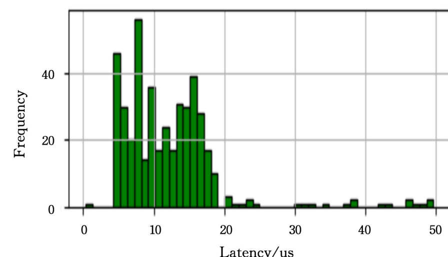
4.2.3 缺页异常类型分析

本文采集到三类缺页异常，分别为未映射匿名页面缺页异常、文件映射缺页异常[16]和写时复制缺页异常[17]。需要注意的是，在 TCP-C 负载下，仅出现了匿名缺页异常，未检测到明显的文件映射缺页异常和写时复制缺页异常。

未映射匿名页面指物理内存中没有页帧并且页表中也没有对应 PTE 的匿名页面，内核需要分配一个物理页面并将其映射到该虚拟地址。do_anonymous_page()函数主要解决这类问题，图 8 给出了该函数的开销分布，我们发现 redis-benchmark 负载下的开销主要集中在 3～7 区间内，而 TPC-C 在 4～20 区间内均有分布。为了探索造成分布差异的原因，我们又做了进一步的分析，结果显示原因在于函数 try_charge()，该函数的作用是对指定的内存控制组进行记账操作[18]。在这一过程中，TPC-C 频繁地调用 page_counter_try_charge() 和 refill_stock()，反映了其在高负载下对内存的频繁请求和内存控制组配额的频繁检查。造成这种情况的原因可能是 MySQL 和 Redis 的内存访问模式不同。MySQL 需要处理复杂的查询和大量的数据写入操作，这可能导致频繁的内存页面分配和释放[19]。而 Redis 作为键值存储，其内存访问模式可能更加连续和预测，导致较少的内存页面管理开销[20]。这同时也说明不同的应用调优的侧重点也会有差异。



(a) Distribution of redis latency durations



(b) Distribution of MySQL latency durations

图 8 do_anonymous_page 函数开销分布图

Fig. 8 do_anonymous_page function overhead distribution

写时复制缺页异常指请求的内存页已在物理内存中但需重新映射或权限调整，无需进行磁盘 I/O 操作[21]的情况。函数 do_wp_page() 主要负责处理该类问题，具体分为两种方式。第一种方式是修改页面权限直接在原页面进行更新，这一方式的开销相对较低，大约在 1～1.2 μs 之间，如图 9 中的第一个峰值。同时也贡献了图 7 和图 5 中的第一个峰值分布。第二种方式涉及写时复制，由 wp_page_copy 函数执行，该过程需要建立该页面的副本并建立新的映射。此种处理方式的开销主要集中在 4～10 区间以及 13～20 区间内。

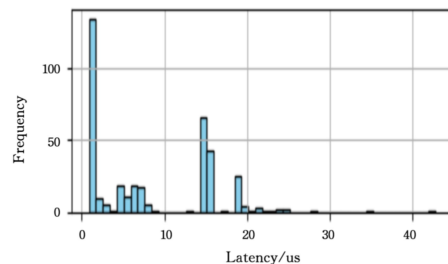


图 9 do_wp_page 函数开销分布图

Fig. 9 do_wp_page function overhead distribution

文件映射缺页异常指当进程访问一个尚未读入物理内存的文件映射页面时,内核需要从文件系统中读取相应的数据块并将其加载到物理内存中。函数 do_fault() 主要负责处理该类问题。图 10 给出了该函数的开销分布。观察到该函数的开销分布较为分散,主要的峰值集中在 15 与 20 μs 左右。该类型主要贡献了图 5 中 18~25 区间内的部分数据分布。

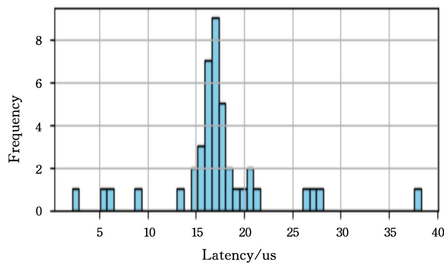


图 10 do_fault 函数开销分布图

Fig. 10 do_fault function overhead distribution

5 讨论

通过实验,本文分析了两种应用环境下缺页异常的处理模式及其延迟分布。研究结果显示,不同类型的应用触发的缺页异常及其延迟分布存在显著差异。此外,即使是同一类型的缺页异常,在不同的负载条件下,延迟分布也呈现不同。这些发现指导我们在系统调优过程中,针对不同的负载条件制定具体的优化策略。同时,我们还识别出在 TPC-C 负载条件下的两个潜在性能瓶颈。

在逐层分析的过程中,我们归纳总结了函数各部分开销的分布情况。本文发现大多数函数具有相对固定的开销,其中大部分函数的处理时间低于 1 μs。除了极少数例外,很少有函数在单次页面错误处理中被重复调用。另一方面,我们着重分析了具有显著特征的函数,这也是第 4 节的重点。通过第 4 节的分析,我们识别了几条关键路径,最终将其划分到 3 种主要缺页异常类型中。另外,本文对不同类型负载下的非函数开销进行了详细分析,分别测量各级函数的非函数开销。非函数开销指除了明确的函数调用外,系统在执行过程中产生的其他时间开销。实验结果表明,不同类型负载下的非函数开销差别很小,基本分布在 1 μs 之内。

结束语 本文深入分析了 Linux 内核中缺页异常的执行模式及其延迟,并明确划分了 3 种主要执行模式,对每类错误的延迟进行了细致的评估。此外,本文还识别出两种异常开销模式,并明确了它们的根本原因,从而揭示了系统中潜在的性能瓶颈。这项工作为深入理解缺页异常处理机制及其对系统性能的影响提供了一定的参考,并为后续的性能优化工作指明了方向。

尽管本文识别了几个潜在的性能瓶颈,但关于如何针对这些瓶颈进行有效优化的具体策略尚未进行详细讨论。未来的工作可以专注于开发和测试针对这些特定瓶颈的优化措施,以提高系统性能。

参 考 文 献

[1] CHO S,CHO Y. Page fault behavior and two prepaging schemes [C]// Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference onComputers and Communications. IEEE,1996:15-21.

[2] MUTHUSUNDARI S,BERLIN M A,GEETHA P. A buffer based page replacement algorithm to reduce page fault[J]. Materials Today:Proceedings,2020,33:4557-4560.

[3] TIRUMALASETTY C,CHOU C C,REDDY N,et al. Reducing minor page fault overheads through enhanced page walker[J]. ACM Transactions on Architecture and Code Optimization(TACO),2022,19(4):1-26.

[4] GANNON D,JALBY W,GALLIVAN K. Strategies for cache and local memory management by global program transformation[C]// International Conference on Supercomputing. Berlin, Heidelberg:Springer Berlin Heidelberg,1987.

[5] CHU W W,OPDERBECK H. Program behavior and the page-fault-frequency replacement algorithm [J]. Computer, 1976, 9(11):29-38.

[6] GREGG B. Linux Performance [EB/OL]. https://www.brendangregg.com/Slides/Percona2018_Linux_Performance.pdf.

[7] LOUKIDES M K. System performance tuning[M]. O'Reilly & Associates,Inc. ,1996.

[8] BIRD T. Measuring function duration with ftrace[C]// Proceedings of the Linux Symposium. Ottawa,ON,Canada:Citeseer, 2009,1.

[9] GHOSH S,HAMOU-LHADJ A,EZZATI-JIVAN N. System and Application Performance Analysis Patterns Using Software Tracing[D]. Montreal Concordia University,2022.

[10] LAROSA C,XIONG L,MANDELBERG K. Frequent pattern mining for kernel trace data[C]// Proceedings of the 2008 ACM Symposium on Applied Computing. 2008:880-885.

[11] CHO H,EGGER B,LEE J,et al. Dynamic data scratchpad memory management for a memory subsystem with an MMU [C]// Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages,Compilers,and Tools for Embedded Systems. 2007.

[12] HAN J,KAMBER M. Data Mining:Concepts and Techniques (nd ed)[M]. San Francisco,CA:Morgan Kaufmann Publishers.

[13] JIAN P,HAN J,MORTAZAVI-ASL B,et al. Prefixspan:Mining sequential patterns efficiently by prefix-projected pattern growth[C]// Proceedings of the 17th International Conference on Data Engineering. IEEE,2001.

[14] LEHMANN R. Redis in the yahoo! cloud serving benchmark [R]. Cloud Serving Benchmark,2011.

[15] LEUTENEGGER S T,DIAS D. A modeling study of the TPC-C benchmark[J]. ACM Sigmod Record,1993,22(2):22-31.

[16] CROTTY A,LEIS V,PAVLO A. Are you sure you want to use mmap in your database management system? [C]// Conference on Innovative Data Systems Research(CIDR). 2022.

[17] SANTHOSH K T , MISHR A D, PAND A B,et al. CoW-Light:Hardware assisted copy-on-write fault handling for secure deduplication[C]// Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and

Privacy. 2019.
[18] HUANG J, QURESHI M K, SCHWAN K. An evolutionary study of Linux memory management for fun and profit[C]// Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16). 2016.
[19] HUNTER A H, CAPITAL J S, KENNELLY C, et al. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator[C]// Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). 2021.
[20] KIM J, CHOE W, AHN J. "Exploring the design space of page management for Multi-Tiered memory systems"[C]// Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21). 2021.
[21] LESOKHIN I, ERAN H, RAINDEL S, et al. Page fault support for network controllers[J]. ACM SIGARCH Computer Architecture News, 2017, 45(1): 449-466.

WEI Shuwen, born in 1997, postgraduate. His main research interests include Linux kernel and Linux kernel performance.

WANG Baohui, born in 1973, professor-level senior engineer, master supervisor. His main research interests include cybersecurity, big data and artificial intelligence.