arXiv:2601.02238v1 [cs.SE] 5 Jan 2026

NQC²：非侵入式 QEMU 代码覆盖率插件

Nils Bosbach 亚琛工业大学，德国亚琛

Alwalid Salama 亚琛工业大学，德国亚琛

Lukas Jünger MachineWare GmbH，德国亚琛

Mark Burton 高通技术公司，法国波尔多

Niko Zurstraßen 亚琛工业大学 亚琛，德国

Rebecca Pelke 亚琛工业大学 亚琛，德国

Rainer Leupers 亚琛工业大学 亚琛，德国

摘要

代码覆盖率分析已成为软件开发中的标准方法，有助于评估测试套件的有效性、识别测试不足的代码段以及发现性能瓶颈。在需要测量嵌入式系统软件的代码覆盖率时，传统方法很快就会遇到局限。常用方法之一是在源文件中插入额外代码，以便在运行时收集并转储覆盖率信息。这些插入的代码通常依赖于操作系统和文件系统的存在，以转储收集到的数据。对于在嵌入式系统上执行的裸机程序而言，这些功能并不可用。

为了解决这一问题，我们推出了 NQC2，一款用于 QEMU 的插件。NQC²会在运行时从 QEMU 中提取覆盖率信息，并将其存储在主机的文件中。这种方法还兼容经过修改的 QEMU 版本，且不需要对目标软件进行插桩。NQC² 的性能比赛灵思的一种可比方法高出最多 8.5 倍。

CCS 概念

• **软件及其工程** → **软件维护工具**;
• **硬件** → 仿真与模拟。

关键词

代码覆盖率、QEMU、TCG 插件、性能优化

ACM 参考格式：
Nils Bosbach、Alwalid Salama、Lukas Jünger、MarkBurton、Niko Zurstraßen、Rebecca Pelke 和 Rainer Leupers。2024年。NQC²：非侵入式 QEMU 代码覆盖率插件。载于面向设计的快速仿真与性能评估（*RAPIDO '24*），*2024*年*1*月*18*日，德国慕尼黑。ACM，纽约，纽约州，美国，共 6 页。https://doi.org/10.1145/3642921.3642924

1 引言

代码覆盖率分析是软件开发中的一项基础实践，可作为可靠的工具，用于评估测试套件的有效性、识别测试不足的代码段，并找出

允许免费制作本作品的全部或部分数字版或纸质副本，供个人或课堂使用，但不得为营利或商业利益而制作或分发副本，且副本首页必须保留本声明和完整引文。对于本作品中归其他作者所有的组成部分，必须尊重其版权。注明出处后可以摘录。若要以其他方式复制、再版、发布到服务器或分发至列表，则须事先获得特定许可和/或支付费用。请向 permissions@acm.org 申请许可。
RAPIDO '24，德国慕尼黑© 2024 版权归所有者/作者所有。出版权已授权给 ACM。ACM ISBN 979-8-4007-1791-8/24/01
https://doi.org/10.1145/3642921.3642924

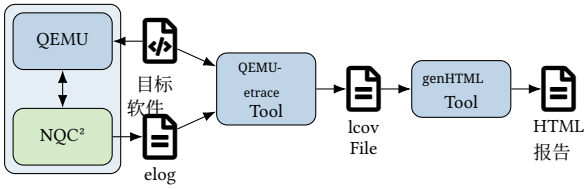


图1：NQC²与 QEMU 的交互、涉及的文件以及使用的后处理工具。

性能瓶颈。它统计软件程序在程序运行时每一行代码的执行次数。1963年，Miller 和 Maloney 发表了这一代码覆盖率理念 [13]，，如今该理念已成为工业界和研究领域的标准 [9, 16]。

在实践中，代码覆盖率分析广泛应用于软件开发的若干关键领域，例如测试覆盖率分析 [9]、瓶颈检测以及为模糊测试器提供指导 [14]。尽管代码覆盖率分析在标准软件开发中发挥着至关重要的作用，但将其应用于嵌入式领域会带来一些挑战。这些挑战源于传统的基于插桩的代码覆盖率分析方法：在编译前、编译期间或编译后向目标可执行文件中注入代码，以收集覆盖率数据。这种方法会引入对编程语言和编译器的依赖，改变可执行文件，而且在许多情况下，还必须在操作系统（OS）中运行目标软件，以便使用系统调用转储覆盖率数据。

为应对这些挑战，本文提出了一种利用 QEMU [1]，（一种开源仿真软件）的方法，以非侵入且可移植的方式改进代码覆盖率分析。本文的贡献如下：

- **QEMU-TinyCode 生成器（TCG）插件 NQC²：** 我们概述其工作原理和实现细节。
- **性能优化：** 我们通过合并、缓冲和异步写入器来降低 NQC²的运行减速。
- **分析：** 我们评估了 NQC²在不同场景下的性能，结果表明，其性能最高可达赛灵思基于 QEMU 的覆盖率解决方案的 8。5 倍。

我们的方法克服了传统代码覆盖率分析的局限，并提供了一种解决方案，可轻松用于经过自定义修改的 QEMU实现。所提出的工具流程如图 1 所示。我们的 QEMU-TCG 插件 NQC² 可在运行时由 QEMU 加载。QEMU 执行目标软件时，会将已执行代码的信息传递给 NQC²。该插件将收集的数据存储在执行日志（elog）文件中。此 elog 文件可与包含调试符号的已执行目标软件的可执行与可链接格式（ELF）文件一起处理，以生成

覆盖率报告。要直观呈现此报告，可以使用*lcov genHTML* [10] 工具。

2 背景与相关工作

如今，代码覆盖率分析已成为业界成熟的工具 [9]，自20世纪60年代末 [16]起便已得到应用。它衡量程序在运行时每一行代码被执行的频率。这些信息能为程序员提供有价值的洞见，揭示测试套件覆盖了软件的哪些部分，或指出性能瓶颈所在。收集到的信息还可作为进一步方法（如模糊测试 [14] ）的指导。

覆盖率数据在运行时收集。常见的方法是在编译之前、期间或之后对软件进行插桩。表1概述了现有的代码覆盖率工具。现代 C/C++ 编译器（如 GNU 编译器套件（GCC）或 *clang*）都集成了在编译期间进行覆盖率插桩的支持。它们分别使用 *gcov* [6] 和 *llvm-cov* [11], 对代码进行插桩。程序执行期间，插桩会捕获覆盖率信息并将其存储在文件中。生成的输出可以转换为覆盖率报告。通常，图形化工具能够生成报告，例如将源代码与行执行次数的注释一同显示，从而便于评估。其中一种工具是 *genHTML*，它属于 *lcov* 项目 [10]。它会生成基于 HTML 的报告。除 C/C++ 外，许多其他语言也有可用的代码覆盖率工具。

在分析嵌入式软件的代码覆盖率时，还会遇到其他挑战。尤其是对于基于插桩的方法，只有在操作系统中运行目标软件时，才能使用 gcov 等标准工具。这是因为插桩后的代码依赖系统调用，例如创建并写入用于存储所收集数据的文件。分析裸机软件的代码覆盖率时，由于没有文件系统和操作系统，这些系统调用无法运行。为解决这一问题，*embedded-gcov* 由 *NASA* 喷气推进实验室 [15]提出，Blasum 等人 [3] 也提出了一种方法，建议将覆盖率信息直接转储到嵌入式目标设备的内存中。执行完成后，需要从目标设备中提取转储数据，并存储在主机上进行分析。

基于插桩的方法的缺点在于它依赖编程语言，并且会改变目标软件。通过向目标软件添加插桩，收集覆盖率信息时执行的二进制文件与正常运行期间执行的二进制文件并不相同。这一事实可能会因内存布局发生变化以及添加了额外指令而导致不同的行为。第二个限制出现在插桩

表1：代码覆盖率方法。

方法	Non-侵入性	独立性语言	OS	标准-独立
gcov [6]	✗	✗	✗	✓
llvm-cov [11]	✗	✗	✗	✓
embedded-gcov [15]	✗	✗	✓	✓
Blasum 等人 [3]	✗	✗	✓	✓
赛灵思的 QEMU [21]	✓	✓	✓	✗
NQC² （本研究）	✓	✓	✓	✓

由编译器添加时。尽管可以在编译后添加插桩 [2]，但大多数最先进的工具（如 gcov）都会在编译前对代码进行插桩。这限制了这些工具的使用范围，使其只能用于开发它们时所针对的特定语言。

为了解决这一问题，可以使用虚拟平台（VP）。虚拟平台是一种基于软件的模拟器，用于模拟完整片上系统（SoC）的行为。它可用于在 x86 通用个人电脑等主机上开发、运行和分析未经修改的目标软件。目标程序的指令是针对目标指令集架构（ISA）编译的，由虚拟平台 CPU 模型中的指令集模拟器（ISS）执行。现代基于动态二进制翻译（DBT）的指令集模拟器会将目标指令集架构中的基本块（BB）（即一组连贯且不包含分支的指令）翻译为宿主指令集架构中的指令。这些经过翻译的基本块称为翻译块（TB）。模拟器可以缓存翻译块，因此每个翻译块只需翻译一次。再次执行同一个翻译块时，即可使用缓存版本。

一种广泛使用的虚拟平台（VP）是基于 DBT 的指令集模拟器 QEMU [1]。QEMU 可以仿真多种目标架构，并运行在不同的仿真主机上。其内部指令集模拟器称为 TCG。QEMU 可以修改为在执行期间捕获跟踪数据。赛灵思已在其 QEMU 分支版本 [21] 中实现了这一功能。赛灵思在其 QEMU 版本中添加了一项名为执行跟踪（etrace）的功能，可在执行期间从 TCG 收集跟踪数据。这些跟踪数据会转储到主机上的 elog 文件中，并可通过 *qemu-etrace* 工具 [8]转换为 lcov 文件。这种方法的优势在于，无需对目标软件进行插桩。它不依赖编译器或所用语言，分析时也无需重新编译。不过，这种方法的缺点是，赛灵思的解决方案要求运行 QEMU 时禁用 TB链接。TB链接是一种优化技术，可在多个翻译块之间执行，而无需在其间切回 QEMU主循环。禁用此功能会降低仿真性能。

另一个缺点是依赖赛灵思定制的 QEMU 分支版本。当主 QEMU 分支获得更新和新功能时，这些更改可能需要较长时间才能合并到赛灵思的版本中。截至2023年，QEMU GitHub代码库已有超过 5,000 个分支 [17],，这表明社区对增强和调整 QEMU 以满足特定需求具有浓厚兴趣。鉴于 etrace 与赛灵思的 QEMU 深度集成，要在其他版本中复用此功能颇具挑战性。

为解决可移植性问题，QEMU 的 TCG 在 4.2 [5] 版本中引入了插件功能。TCG 插件是一种可在运行时由 QEMU 加载的共享库。在此过程中，QEMU 会调用插件的 qemu_plugin_install 函数，使插件能够注册回调函数。有关插件应用程序编程接口（API）的全面概述，请参见 [18]。重要的是，插件可用于各种 QEMU实现，即使 QEMU 后续发生更改或扩展，也能确保插件可重复使用。与直接修改 QEMU 不同，这项功能使扩展具备可移植性。

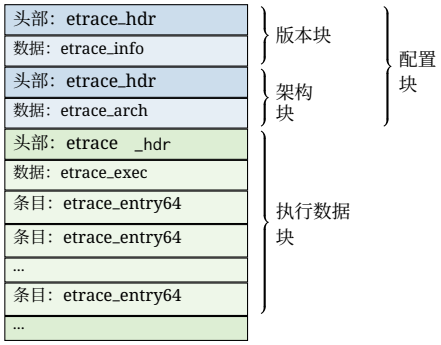


图2：elog 文件结构。

3 实现

NQC² 是一个用于 QEMU 的 TCG 插件，它利用插件 API 的功能生成 elog 文件。该文件为现有工具的处理提供了基础，并有助于生成全面的覆盖率报告，如图 1 所示。

elog 文件是一种二进制数据文件，其结构由多个串联的代码块组成，如图2所示。每个代码块包含两个主要部分：头部和数据段。代码 1 所示的头部结构决定了后续数据段的类型和长度。每种数据段类型都有其自身定义的结构。

```
struct etrace_hdr{ uint16_t type; // 后续数据的类型 uint16_t unit_id;// CPU ID uint32_t len; // 后续数据的长度 } __attribute__ ((packed));
```

代码 1：etrace头结构体。

对于覆盖率评估而言，代码 2 中给出的条目类型尤为重要。对于已执行的翻译块，etrace_entry64 结构体会在目标地址空间中存储已执行指令的起始和结束地址，以及以纳秒为单位的执行时长。为便于组织和管理数据，可以将多个 etrace_entry64 代码块组合成一个执行数据块，如图2所示。etrace_hdr 的 len属性定义了 etrace_exec 和 etrace_entry64 条目的累计大小。运行时，NQC² 从 QEMU 收集数据，将其存储在 etrace_entry64 结构体中，并写入 elog 文件。

1 struct etrace_exec { // type = 2 uint64_t start_time; // 首个翻译块执行的时间戳 3 } __attribute__ ((packed)); 5 struct etrace_entry64 { 6 uint32_t duration; // 执行时长（ns） 7 uint64_t start , end; // 起始和结束地址 8 } __attribute__ ((packed)); 代码 2：已执行翻译块的条目。

图3展示了 NQC² 与 QEMU 交互的方式和时机。QEMU 加载插件后，会调用 qemu_plugin_install

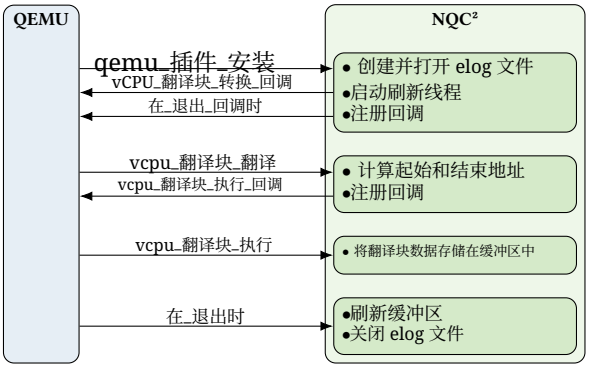


图 3：NQC²架构。

函数。随后，NQC² 会创建一个 elog 文件并打开以供写入，然后转储包含版本和架构信息的配置块。一个异步写入器函数会在新的 POSIX 线程（pthread）中执行，之后便由它负责将数据写入 elog 文件。我们将在第 3.1 节详细讨论这一性能优化。在 qemu_plugin_install 函数末尾，会通过 QEMU 的 TCG 插件 API 注册两个回调函数。第一个回调函数 vcpu_tb_trans 会在 TCG 翻译新的基本块时通知 NQC² 。第二个回调函数 at_exit 则会在虚拟 CPU（vCPU）退出时执行。

每当 TCG 翻译一个基本块时，就会调用 vcpu_翻译块_翻译函数。在此函数中，会计算翻译块第一条和最后一条指令的地址，并将其存储在一个结构体中。另一个回调函数 vcpu_翻译块_执行会注册为在每次执行翻译块时调用。注册时，会传递一个指向包含翻译块起始和结束地址的结构体的指针。此后，每次调用 vcpu_翻译块_执行时，都会传递该指针。

每次执行翻译块（TB）后，都会从 vcpu_tb_exec 函数中传入的结构体里提取翻译块的起始和结束地址。相关信息会被复制到 anetrace_entry64 结构体中，如代码 2 所示。etrace_entry64 结构体会被放入缓冲区。缓冲区填满后，收集到的 etrace_entry64 结构体会连同前置头部（如代码 1 所示）以及 etrace_exec 块（如代码 2 所示）写入 elog 文件。

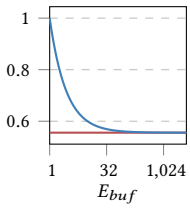
当 vCPU 在仿真结束时退出，缓冲区中剩余的 etrace_entry64 结构体会被写入 elog 文件。随后关闭该文件。之后可以使用 QEMU-etrace 工具 [8] 对其进行后处理，以生成 lcov 覆盖率报告，如图 1 所示。

etrace_entry64 结构体会先收集到缓冲区中，然后再写入 elog 文件。在将代码块写入 elog 文件之前对其进行缓冲，可以根据图2对代码块进行分组，从而减小文件大小。这样一来，所需的头部数量也会减少。elog 文件的大小 S_{elog}可根据公式（1）计算得出。

$$S_{elog} = S_{conf} + \frac{\#TB}{E_{buf}} \cdot \left(S_{hdr} + S_{exec} + E_{buf} \cdot S_{entry64}\right) \quad (1)$$

$$= 124\,\mathrm{B} + \frac{\#TB}{E_{buf}} \cdot \left(16\,\mathrm{B} + E_{buf} \cdot 20\,\mathrm{B}\right) \quad (2)$$

$$\approx \frac{\#TB}{E_{buf}} \cdot \left(16\,\mathrm{B} + E_{buf} \cdot 20\,\mathrm{B}\right) \quad (3)$$



$$\frac{S_{elog}(E_{buf})}{S_{elog}(E_{buf}=1)} \approx \frac{4}{9}\cdot\frac{1}{E_{buf}}+\frac{5}{9} \quad (4)$$

图 4：缓冲对 elog 文件大小的缩减。

大小 S_{elog} 由一个固定量 S_{conf} 和一个与工作负载相关的量组成。固定量由开头一次性写入的版本和架构信息所产生。与工作负载相关的量取决于已执行 TB 的数量（$\#TB$）、etrace_hdr（S_{hdr}）、etrace_exec（S_{exec}）和 etrace_entry64（$S_{entry64}$）结构体的大小，以及缓冲区中可容纳的 etrace_entry64 结构体数量（E_{buf}）。由于固定部分相对较小，可以忽略不计，因此该公式可简化为公式（3）。

为了估计缓冲对 elog 文件大小的影响，可以根据公式（4）计算启用缓冲时的 elog 文件大小 S_{elog} 与未启用缓冲时的 elog 文件大小 $S_{elog}(E_{buf}=1)$ 的比值。由此得到的文件大小相对缩减量如图 4 所示。结果表明，对 etrace_entry64 结构体进行缓冲和打包，最多可将 elog 文件大小缩减 44%。即使只打包 32 个元素，文件大小也已缩减 43%。进一步增加打包数量对文件大小的影响有限。

3.1 多缓冲

etrace_entry64 条目的缓冲和捆绑具有双重作用，不仅有助于减小 elog 文件大小，还能提升性能优化的可能性。当缓冲区已满时，其中包含的数据需要写入 elog 文件。在此过程中，QEMU 会被暂停，从而降低性能。为解决这一问题，我们建议实现一个异步写入器 pthread，并配合使用多个缓冲区。

图5以四个缓冲区为例对这一概念进行了概述。我们不再使用一个填满后才刷新的单一缓冲区，而是采用多个缓冲区。每个缓冲区都会被分配一种状态，可以是 空、填充中、已满 或 刷新中。在 NQC2 初始化期间，会创建一个异步写入器 pthread（参见图3）。每个缓冲区都可以由两个不同的 pthread 访问：写入器或主 NQC² pthread，后文称为 收集器 。开始时，所有缓冲区均为空。收集器将第一个缓冲区的状态更改为填充中，并持续向其中添加 etrace_entry64 结构体，直到缓冲区已满。随后，它将状态更改为已满。当下一个缓冲区处于空状态时，收集器将其状态更改为填充中并继续执行。如果下一个缓冲区处于已满或刷新状态，收集器就必须等待写入器将其状态更改为空。

写入器在异步 p- 线程中与收集器并行运行。仿真开始时，它会等待收集器将第一个缓冲区的状态更改为满。随后，写入器将状态更新为正在刷新，将数据刷新到 elog 文件中，并将状态设为空。它会等待收集器将下一个缓冲区的状态更改为满，然后继续向 elog 文件写入数据。

收集器等待缓冲区变空以及写入器等待缓冲区变满的过程，均使用 POSIX 条件变量进行同步。每当

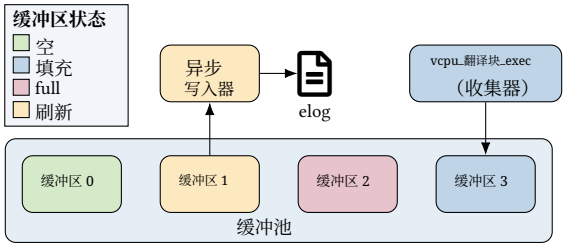
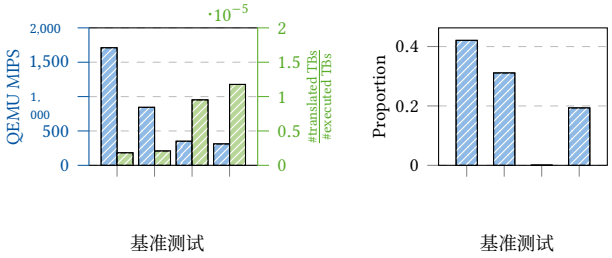


图 5：NQC²多缓冲方案。



(a) QEMU 执行统计。

(b) 可合并部分的比例 etrace_entry64 代码块。

图 6：QEMU 执行统计。

状态从刷新变为空，或从填充变为满时，都会通知一个条件变量，以提醒可能正在等待的收集器或写入器。

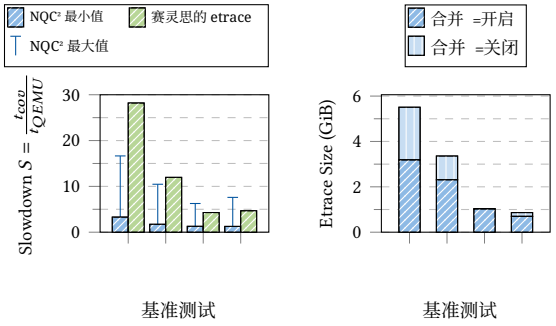
3.2 合并

NQC² 将 etrace_entry64 结构体的合并作为第二项优化。在 vCPU_TB_执行回调中，将新条目添加到缓冲区之前，会先检查缓冲区是否为空。如果不为空，则比较缓冲区中最后一个元素的结束地址与已执行翻译块的起始地址。如果两者匹配，则可以更新缓冲区中的最后一个条目，而不是向缓冲区添加新条目。具体做法是将最后一个条目的结束地址设置为当前条目的结束地址。如果标注了计时信息，则需要将当前翻译块的执行时长加到更新后条目的 duration 字段中。

4 结果

为了测量 NQC2 带来的性能下降，我们评估了广泛使用的裸机基准测试 Coremark、Dhry-stone、Stream 和 Whetstone 的代码覆盖率。所有基准测试均在 QEMU 8.1.1 [17] 版本上分别启用和不启用 NQC² 的情况下进行评估，同时也在赛灵思的 QEMU 分支版本 2023.1_更新 1 [21]上进行评估。模拟的目标架构为 *aarch64*。使用的主机 CPU 是 AMD Ryzen 9 3900X 12 核处理器。加载 NQC² 后，可以配置缓冲区数量、每个缓冲区中的 etrace_entry64 结构体数量以及合并选项。

图 6 展示了基准测试的一般特性以及 QEMU 的执行情况，这些与代码覆盖率分析无关。图 6a 描绘了 QEMU 针对不同基准测试的模拟速度，单位为每秒百万条指令（MIPS）。可以看出，不同基准测试所达到的模拟速度存在差异。Coremark 和 Dhrystone 的模拟速度最高。模拟速度较高的一个原因是，这两个基准测试主要由



(a) 运行时减速。 **(b) 跟踪文件大小。**

图 7：NQC²基准测试结果。

简单的基于整数算术的指令构成 [4]。与 Dhrystone 相比，Coremark 的加载指令数量较少，因此执行速度更快。Stream 主要使用加载和存储指令，因而速度下降幅度更大。Whetstone 包含大量浮点运算，其模拟复杂度高于整数运算。

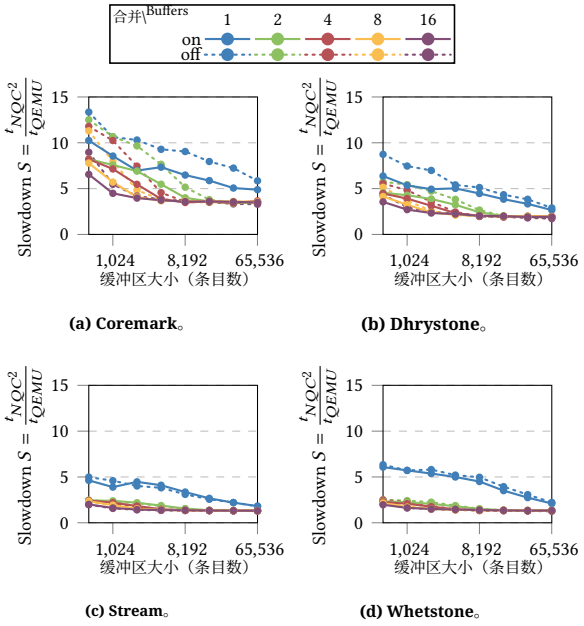
此外，图 6a 展示了翻译块在执行前需要进行翻译的概率。由于 QEMU 会缓冲已翻译的翻译块，对于所有受测基准测试，翻译块过去已被翻译、因而可以直接执行的概率都高于 99.9%。然而，Whetstone 的翻译块翻译概率比 Coremark 高出 6.4 倍以上。较低的翻译块翻译概率可通过减少翻译次数提升 QEMU 的性能。

图 6b 展示了后续的 etrace_entry64 代码块有多频繁地可以合并（参见第 3.2 节）。例如，Coremark 执行所需存储的 295,290,670 个 etrace_entry64 代码块中，有 124,257,227 个（42.08 %）可以与其前驱代码块合并。相比之下，对于 Stream 基准测试，55,024,880 个 etrace_entry64 代码块中，只有 10,481 个（0.02 %）可以合并。

图 7a 展示了启用 etrace 后，NQC² 和赛灵思的 QEMU 相较于 QEMU 8.1.1 所导致的运行时减速。由于 NQC² 的缓冲区数量、缓冲区大小和合并方式均可配置，并会影响性能，因此图中给出了可达到的最小值和最大值。评估的缓冲区数量为 1 至 16 个，每个缓冲区可容纳 512,536 个 etrace_entry64 元素，合并可启用或禁用。

赛灵思的 etrace 实现在 Coremark 基准测试中造成了高达 28.2 倍的运行时减速（图 7a），而对于所有评估的基准测试，NQC² 造成的减速最高可控制在 3.3 倍。对于 Coremark 基准测试，NQC² 的表现比赛灵思的方案高出 8.5 倍。NQC² 性能更好的一个原因是，赛灵思的 etrace 实现必须禁用翻译块链式连接才能正常工作。此外，与赛灵思的实现相比，NQC² 支持多缓冲和可变缓冲区大小功能。赛灵思使用单个缓冲区，可容纳 16,384 个 etrace_entry64 元素。他们使用单个 pthread。

NQC² 的最坏情况减速幅度明显取决于所执行的工作负载。比较图 6a 和图 7a 所示的趋势时，可以发现 MIPS 值的变化与由此产生的减速幅度之间存在一致的规律。似乎在 QEMU 能以较高 MIPS 值执行的工作负载下，NQC² 会带来更大的减速。调整缓冲区



(a) Coremark。 **(b) Dhrystone。**

(c) Stream。 **(d) Whetstone。**

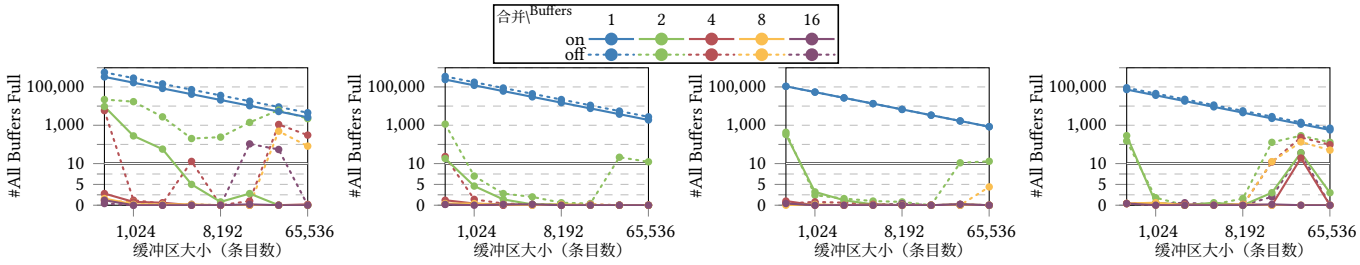
图 8：NQC²减速。

参数和合并选项有助于限制减速幅度。在最佳情况下，NQC² 的表现始终优于赛灵思的 etrace。

图 7b 展示了启用和禁用合并时 elog 文件的文件大小。根据公式（4），文件大小还取决于所用的缓冲区大小。不过，对于所评估的缓冲区大小（大于或等于 512，），其影响可以忽略不计，如图 4 所示。因此，不再进一步评估这种依赖关系。可以看出，elog 文件会迅速增长到数吉字节。合并有助于减小文件大小，但其效果取决于工作负载。合并的影响与可合并的 etrace_entry64 代码块所占比例直接相关，如图 6b 所示。对于代码块很可能可合并的基准测试（例如 Coremark 和 Dhrystone），合并能够带来益处。对于可合并代码块较少的基准测试（例如 Stream），合并无法减小文件大小。

图 8 展示了在不同基准测试和配置下，启用 NQC², t_{NQC^2} 时所需执行时间与未启用 NQC², t_{QEMU} 时所需执行时间的比值。仅使用单个缓冲区时（蓝线），收集器和写入器无法并行工作，因而只能顺序执行，导致运行速度大幅下降。Stream 和 Whetstone 基准测试的结果表明，异步写入器结合多缓冲能显著提升性能。对于这些基准测试，蓝线与其他线之间的差距最大。再次从图 6a 所示的数据中，可以找到这些特定基准测试受到更大影响的原因。每秒百万条指令数较低的工作负载（如 Stream 和 Whetstone）从并行填充和刷新缓冲区中获益最多。对于这些工作负载，QEMU 执行所需的时间更长，因此调用 NQC² 的频率较低。调用频率降低后，写入器便有更多时间清空已满的缓冲区，从而减少缓冲区拥塞。

图 9 进一步印证了这一解释，展示了各种缓冲区和合并配置对缓冲区



(a) Coremark。 **(b) Dhrystone。** **(c) Stream。** **(d) Whetstone。**

图 9：NQC²缓冲区拥塞。

拥塞的影响。图中显示了所有可用缓冲区处于满载状态的频率，此时收集器需要等待写入器清空一个缓冲区。如图 6b 所示，合并通过减少需要存储在 elog 文件中的条目数量，有助于缓解拥塞。

在单缓冲区配置中，收集器和写入器无法并行工作，因此每当缓冲区填满时都必须等待。更大的缓冲区大小会持续减少已填满缓冲区的数量，这一点从蓝色图表的线性斜率中可以看出。增加缓冲区数量或缓冲区大小通常会降低拥塞概率。然而，图 9 显示，对于某些工作负载，更大的缓冲区大小可能会增加拥塞概率。这可能是由于根据已使用的大小，缓冲区清空的时间点不同所导致的。另一个原因是，更大的缓冲区会减少收集器的交换开销，从而缩短写入器能够在不发生拥塞的情况下刷新缓冲区的时间。不过，如图 8 所示，这种略微增加的拥塞概率并不会导致性能下降。当拥塞概率达到零时，继续增大缓冲区大小将不再影响性能，图 8 和图 9 的对比也说明了这一点。

5 结论与未来工作

在软件开发领域，代码覆盖率分析是一项重要实践，可帮助开发人员评估其测试套件的有效性，确定未经测试的代码段，并揭示性能瓶颈。我们提出了 NQC2，这是一款支持对嵌入式软件进行无插桩代码覆盖率分析的 QEMU-TCG 插件。通过使用 QEMU 插件接口，NQC² 也兼容定制版 QEMU。我们介绍了其工作原理、elog 文件格式的结构以及将其集成到 TCG 插件中的方法。我们的性能优化措施，例如异步写入器、使用多个缓冲区以及合并代码块，能够显著降低性能减速。

我们使用多个基准测试评估了 NQC² 的性能。已经发现，NQC² 的减速程度在很大程度上取决于所执行的工作负载。对于以每秒百万条指令计的模拟速度较高的基准测试，相对减速幅度更大。异步写入器可以显著降低减速，尤其是对于每秒百万条指令数较低的基准测试。根据工作负载的 TB 执行顺序，合并条目可以减小 elog 文件大小并提升性能。

未来的工作可以为 NQC² 添加 elog 文件的写入前即时压缩或直接处理功能，以减少 elog 文件大小。总而言之，NQC² 为各种 QEMU 实现中的代码覆盖率分析提供了一种多用途解决方案，提升了性能，并加深了对影响其效率因素的理解。

参考文献

[1] Fabrice Bellard。2005。QEMU，一种快速且可移植的动态翻译器。。载于 *USENIX* 年度技术会议，*FREENIX* 专题，第 41 卷。美国加利福尼亚州，10–5555。[2] M. Ammar Ben Khadra、Dominik Stoffel 和 Wolfgang Kunz。2020。高效的二进制级覆盖率分析。载于第 *28* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集（*ESEC/FSE* *2020* 10 1145 3368089 3409694)。[3] https://doi.org/ . / . Holder Blasum、Frank Görgen 和 Jürgen Urban。2007。。嵌入式系统上的 gcov。（2007）。[4] Nils Bosbach、Lukas Jünger、Rebecca Pelke、Niko Zurstraßen 和 Rainer Leupers。2023年。基于熵的指令集模拟器基准测试分析。载于*DroneSE* 与 *RAPIDO*：受限嵌入式系统的系统工程会议论文集（*RAPIDO '* *23* 10 1145 3579170 3579267)。https://doi.org/ . / . Emilio G. Cota 和 Luca P. Carloni。2019。。使用快速且可扩展的动态二进制翻译进行跨 ISA 机器插桩。载于第 *15* 届 *ACM SIGPLAN/SIGOPS* 虚拟执行环境国际会议论文集。[6] 自由软件基金会。2023年。Gcov（使用 GNU 编译器套件（GCC））。https://gcc.gnu.org/onlinedocs/gcc/Gcov.html [7] Shay Gal-On 和 Markus Levy。2012。探索 Coremark：一项力求兼顾简洁性和效用的基准测试。嵌入式微处理器基准测试联盟（2012）。[8] Edgar E. Iglesias。2023年。edgarigl/qemu-etrace。https://github.com/edgarigl/qemu-etrace original-date: 2016-02-15T09:42:55Z。[9] Marko Ivanković、Goran Petrović、René Just 和 Gordon Fraser。2019。Google 的代码覆盖率。载于*2019* 年第 *27* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集（*ESEC/FSE* *2019* 10 1145 3338906 3340459)。[10] https://doi.org/ . / . Linux 测试项目。。2023 LTP GCOV 扩展（LCOV）。https://github.com/linux-test-project/lcov original-date: 2014-05-27T14:38:58Z。[11] LLVM 项目。2023年。llvm-cov - 输出覆盖率信息 — LLVM 18.0.0git 文档。https://llvm.org/docs/CommandGuide/llvm-cov.html [12] John D. McCalpin。1995。当前高性能计算机的内存带宽与机器平衡。*IEEE* 计算机体系结构技术委员会（*TCCA*）通讯（1995 年 12 月），19–25。[13] Joan C. Miller 和 Clifford J. Maloney。1963年。数字计算机程序的系统性错误分析。*ACM* 通讯6，2（1963 年 2 月）。[14] Stefan Nagy 和 Matthew Hicks。2019。全速模糊测试：通过覆盖率引导跟踪降低模糊测试开销。载于 *2019* 年 *IEEE* 安全与隐私研讨会（*SP*）。787–802。https://doi.org/10.1109/SP.2019.00069 ISSN：2375-1207。[15] 美国国家航空航天局喷气推进实验室。2023年。nasa-jpl/embedded-gcov。https://github.com/nasa-jpl/embedded-gcov original-date: 2022-02-02T19:25:26Z。[16] P. Piwowarski、M. Ohba 和 J. Caruso。1993。功能测试期间的覆盖率测量经验。载于 *1993* 年第 *15* 届国际软件工程会议论文集。 287 301 10 1109 1993 346035 – 。[17] https://doi.org/ . /ICSE. 。2023 。QEMU。。QEMU。https://github.com/qemu/qemu [18] QEMU。2023年。QEMU TCG 插件 — QEMU 文档。https://www.qemu.org/docs/master/devel/tcg-plugins.html#api [19] Reinhold P. Weicker。1984。Dhrystone：一种合成系统编程基准测试。*ACM* 通讯 27, 10 1984 10 1145 358274 358283 （10 月）。[20] https://doi.org/ . / . Brian A Wichmann。1970。。一些 *ALGOL* 程序的统计数据。英国国家物理实验室中央计算机部门。[21] 赛灵思。2023年。赛灵思的 QEMU 分支。https://github.com/Xilinx/qemu