

NQC²: A Non-Intrusive QEMU Code Coverage Plugin

Nils Bosbach
RWTH Aachen University
Aachen, Germany

Alwalid Salama
RWTH Aachen University
Aachen, Germany

Lukas Jünger
MachineWare GmbH
Aachen, Germany

Mark Burton
Qualcomm Technologies, Inc.
Bordeaux, France

Niko Zurstraßen
RWTH Aachen University
Aachen, Germany

Rebecca Pelke
RWTH Aachen University
Aachen, Germany

Rainer Leupers
RWTH Aachen University
Aachen, Germany

ABSTRACT

Code coverage analysis has become a standard approach in software development, facilitating the assessment of test suite effectiveness, the identification of under-tested code segments, and the discovery of performance bottlenecks. When code coverage of software for embedded systems needs to be measured, conventional approaches quickly meet their limits. A commonly used approach involves instrumenting the source files with added code that collects and dumps coverage information during runtime. This inserted code usually relies on the existence of an operating and a file system to dump the collected data. These features are not available for bare-metal programs that are executed on embedded systems.

To overcome this issue, we present NQC², a plugin for QEMU. NQC² extracts coverage information from QEMU during runtime and stores them into a file on the host machine. This approach is even compatible with modified QEMU versions and does not require target-software instrumentation. NQC² outperforms a comparable approach from Xilinx by up to 8.5 x.

CCS CONCEPTS

- **Software and its engineering** → **Software maintenance tools**;
- **Hardware** → *Simulation and emulation*.

KEYWORDS

Code Coverage, QEMU, TCG Plugin, Performance Optimization

ACM Reference Format:

Nils Bosbach, Alwalid Salama, Lukas Jünger, Mark Burton, Niko Zurstraßen, Rebecca Pelke, and Rainer Leupers. 2024. NQC²: A Non-Intrusive QEMU Code Coverage Plugin. In *Rapid Simulation and Performance Evaluation for Design (RAPIDO '24)*, January 18, 2024, Munich, Germany. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3642921.3642924>

1 INTRODUCTION

Code coverage analysis is a fundamental practice in software development, serving as a reliable tool for assessing the effectiveness of test suites, identifying under-tested code segments, and pinpointing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RAPIDO '24, January 18, 2024, Munich, Germany

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1791-8/24/01...\$15.00
<https://doi.org/10.1145/3642921.3642924>

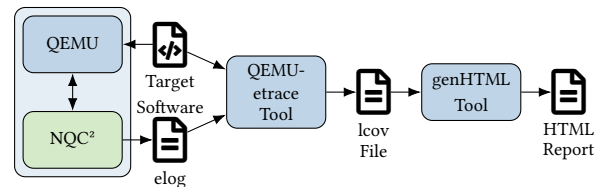


Figure 1: Interaction between NQC² and QEMU, involved files, and used postprocessing tools.

performance bottlenecks. It counts how many times each line of a software program is executed during program runtime. In 1963, Miller and Maloney published this idea of code coverage [13], which became a standard in industry and research nowadays [9, 16].

In practice, code coverage analysis finds application in several key areas of software development like test coverage analysis [9], bottleneck detection, and guidance of fuzzers [14]. While code coverage analysis has a pivotal role in standard software development, its application to the embedded domain causes some challenges. These challenges arise from the traditional instrumentation-based approach of code coverage analysis, which involves injecting code into the target executable before, during, or after compilation to gather coverage data. This approach introduces language and compiler dependencies, alters the executable, and, in many cases, necessitates the execution of the target software within an Operating System (OS) to use system calls for dumping the coverage data.

In response to these challenges, this paper introduces an approach that leverages QEMU [1], an open-source simulation software, to enhance code coverage analysis in a non-intrusive and portable manner. We present the following contributions:

- **QEMU-Tiny Code Generator (TCG) plugin NQC²:** We sketch the working principle and implementation details.
- **Performance optimization:** We reduce the slowdown of NQC² by merging, buffering, and an asynchronous writer.
- **Analysis:** We evaluate the performance of NQC² for different scenarios showing that we can outperform Xilinx’s QEMU-based coverage solution by a factor of up to 8.5.

Our approach addresses the limitations of traditional code coverage analysis and offers a solution that can be easily used with QEMU implementations that contain custom modifications. The proposed toolflow is depicted in Fig. 1. Our QEMU-TCG plugin NQC² can be loaded by QEMU during runtime. While QEMU executes the target software, it passes information of the executed code to NQC². The plugin stores the collected data in an Execution Log (elog) file. This elog file can be processed together with the debugging-symbol-containing Executable and Linkable Format (ELF) file of

the executed target software to generate an *lcov* coverage report. To visualize this report, the *genHTML* [10] tool can be used.

2 BACKGROUND AND RELATED WORK

Code coverage analysis is nowadays an established tool in industry [9] that has been used since the late 1960s [16]. It measures how often a line of a program is executed during runtime. This information is a valuable insight to the programmer revealing which parts of the software are tested by a testing suite or pinpointing where the performance bottlenecks are. The collected information can be also used by further approaches like fuzzing [14] as guidance.

Coverage data are collected during runtime. A common approach is the instrumentation of the software before, during or after compilation. An overview of available code coverage tools is presented in Table 1. Modern C/C++ compilers like the GNU Compiler Collection (GCC) or *clang* have integrated support for coverage instrumentation during compilation. They use *gcov* [6] and *llvm-cov* [11], respectively, to instrument the code. During execution, the instrumentations capture the coverage information and store it in a file. The produced output can be converted into a coverage report. Usually, graphical tools are available that create reports that, e.g., display the source code together with the annotation of line-execution counts. This facilitates the evaluation. One of those tools is *genHTML* of the *lcov* project [10]. It creates a HTML-based report. Besides C/C++, coverage tools are available for many different languages.

When it comes to coverage analysis for embedded software, additional challenges arise. Especially for the instrumentation-based approach, standard tools like *gcov* can only be used if the target software is executed within an OS. This is a requirement because the instrumented code depends on system calls such as creating and writing to the file that stores the collected data. When analyzing the code coverage of bare-metal software, these system calls do not function due to the absence of a file system and OS. To circumvent this issue, *embedded-gcov* from the *NASA Jet Propulsion Laboratory* [15] and the approach by Blasum et al. [3] suggest to directly dump the coverage information into the memory of the embedded target. After execution, the dump needs to be extracted from the target and stored on a host machine to be analyzed.

The drawbacks of the instrumentation-based approach are that it is programming-language-dependent and changes the target software. By adding instrumentations to the target software, the binary that is executed when coverage information is collected differs from the one that is executed during normal operation. This fact can lead to different behavior due to a changed memory layout and added instructions. The second limitation appears when instrumentations

are added by the compiler. Although it is possible to add instrumentations after compilation [2], most state-of-the-art tools like *gcov* instrument the code before compilation. This limits the usage of those tools to the specific language they have been developed for.

To overcome this issue, Virtual Platforms (VPs) can be used. A VP is a software-based simulator that mimics the behavior of a full System-on-a-Chip (SoC). It can be used to develop, run, and analyze the unmodified target software on a host machine like an x86 general-purpose PC. The instructions of the target program, which have been compiled for the target Instruction-Set Architecture (ISA), are executed by the Instruction-Set Simulator (ISS), which is part of the CPU model of the VP. Modern Dynamic Binary Translation (DBT)-based ISSs translate Basic Blocks (BBs), which are groups of coherent instructions without branching, from the target ISA to the host ISA. Those translated BBs are referred to as Translation Blocks (TBs). TBs can be cached by the simulator so the translation is only needed once per TB. When the same TB is executed a second time, the cached version can be used.

A widely-used VP that has an DBT-based ISS is QEMU [1]. QEMU can simulate several target architectures and run on different simulation hosts. Its internal ISS is called TCG. QEMU can be modified to capture tracing data during execution. This has been done by Xilinx in their QEMU fork [21]. Xilinx added a feature called Execution Trace (*etrace*) to their QEMU version, which collects traces from the TCG during execution. These traces are dumped into an *elog* file on the host machine, which can be converted to a *lcov* file by the *qemu-etrace* tool [8]. The strength of this approach is that the target software does not need to be instrumented. There is no dependency on the compiler or used language and no recompilation is needed for the analysis. However, the drawback is that Xilinx's solution requires running QEMU with disabled TB chaining. TB chaining is an optimization technique that allows the execution of multiple TBs without switching back to QEMU's main loop in between. When this feature is disabled, the simulation performance is reduced.

Another drawback is the reliance on Xilinx's customized QEMU fork. When the main QEMU branch receives updates and new features, there can be a significant delay before these changes are incorporated into Xilinx's version. As of 2023, there are more than 5,000 forks of the QEMU GitHub repository [17], indicating a strong community interest in enhancing and adapting QEMU to specific requirements. Given *etrace*'s deep integration with Xilinx's QEMU, the ability to reuse this feature in other versions is challenging.

To address the issue of portability, QEMU's TCG has a plugin feature that has been introduced in version 4.2 [5]. A TCG plugin is a shared library that can be loaded by QEMU at runtime. During this process, QEMU invokes the *qemu_plugin_install* function of the plugin, which enables the plugin to register callback functions. A comprehensive overview of the plugin Application Programming Interface (API) can be found in [18]. Importantly, plugins can be employed across various QEMU implementations, ensuring their reusability even when QEMU has undergone alterations or extensions. In contrast to intrusive modifications of QEMU, this feature enables the portability of extensions.

Table 1: Code Coverage Approaches.

Approach	Non-Intrusive	Independenciy Language	OS	Stand-alone
<i>gcov</i> [6]	⊗	⊗	⊗	⊙
<i>llvm-cov</i> [11]	⊗	⊗	⊗	⊙
<i>embedded-gcov</i> [15]	⊗	⊗	⊙	⊙
Blasum et al. [3]	⊗	⊗	⊙	⊙
Xilinx's QEMU [21]	⊙	⊙	⊙	⊗
NQC ² (this work)	⊙	⊙	⊙	⊙

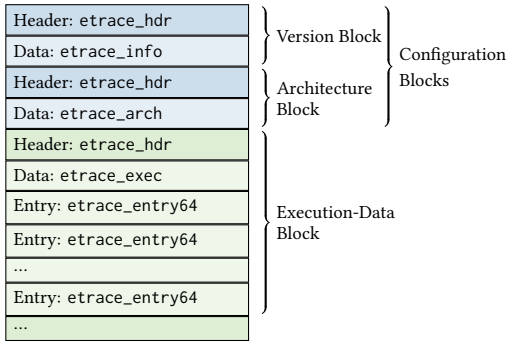


Figure 2: The elog file structure.

3 IMPLEMENTATION

NQC² is a TCG plugin for QEMU that leverages the capabilities of the plugin API to generate an elog file. This file serves as the foundation for processing by existing tools and facilitates the generation of a comprehensive coverage report as depicted in Fig. 1.

The elog file is a binary data file that exhibits a structured layout composed of concatenated blocks as illustrated in Fig. 2. Each block comprises two main components, a header and a data segment. The structure of the header, depicted in Code 1, determines the type and length of the subsequent data segment. Each data segment type has its own defined structure.

```

1 struct etrace_hdr {
2     uint16_t type; // type of the subsequent data
3     uint16_t unit_id; // CPU ID
4     uint32_t len; // length of the subsequent data
5 } __attribute__((packed));

```

Code 1: etrace header struct.

Of particular significance for the coverage evaluation is the entry type presented in Code 2. For an executed TB, the `etrace_entry64` struct stores the start and end addresses of the executed instructions within the target’s address space and the execution duration in nanoseconds. To enhance the organization and manageability of the data, multiple `etrace_entry64` blocks can be grouped into an execution-data block as visually depicted in Fig. 2. The `len` attribute of the `etrace_hdr` defines the accumulated sizes of both the `etrace_exec` and the `etrace_entry64` entries. During runtime, NQC² collects the data from QEMU, stores them in `etrace_entry64` structs and dumps them to the elog file.

```

1 struct etrace_exec { // type = 1
2     uint64_t start_time; // timestamp of first TB exec
3 } __attribute__((packed));

5 struct etrace_entry64 {
6     uint32_t duration; // execution duration (ns)
7     uint64_t start, end; // start & end addresses
8 } __attribute__((packed));

```

Code 2: Entry for an executed TB.

Fig. 3 visualizes how and when NQC² interacts with QEMU. After the plugin has been loaded by QEMU, the `qemu_plugin_install`

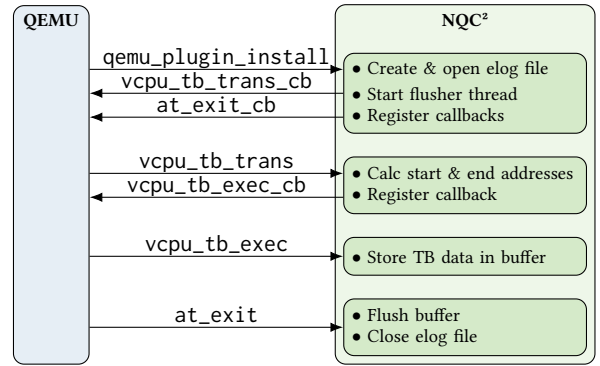


Figure 3: NQC² schema.

function is called. NQC² then creates an elog file, opens it for writing, and dumps configuration blocks containing version and architecture information. An asynchronous writer function is executed in a new POSIX Thread (pthread) which is from then on used to write data to the elog file. This performance optimization will be discussed in detail in Section 3.1. At the end of the `qemu_plugin_install` function, two callback functions are registered using QEMU’s TCG-plugin API. The first callback function, `vcpu_tb_trans`, notifies NQC² when the TCG translates a new BB. The second callback function, `at_exit`, is executed once a virtual CPU (vCPU) exits.

Every time the TCG translates a BB, the `vcpu_tb_trans` function is called. In this function, the addresses of the first and last instructions of the TB are calculated and stored in a struct. A third callback function, `vcpu_tb_exec`, is registered to be called every time the TB is executed. During the registration, a pointer to the struct containing the start and end addresses of the TB is handed over. This pointer is then passed to every call of `vcpu_tb_exec`.

After each execution of a TB, the start and end addresses of the TB are extracted from the handed-over struct in the `vcpu_tb_exec` function. The information is copied to an `etrace_entry64` struct as shown in Code 2. The `etrace_entry64` struct is placed in a buffer. Once the buffer is full, the collected `etrace_entry64` structs are written to the elog file with a preceding header, as depicted in Code 1, and an `etrace_exec` block, as presented in Code 2.

When a vCPU exits at the end of the simulation, the remaining `etrace_entry64` structs from the buffer are written to the elog file. The file is then closed. It can be post-processed using the QEMU-etrace tool [8] to generate a lcov coverage report as shown in Fig. 1.

The `etrace_entry64` structs are collected in a buffer before they are written to the elog file. Buffering the blocks before writing them into the elog file reduces the file size by grouping blocks according to Fig. 2. Thereby, the number of required headers is reduced. The size of the elog file, S_{elog} , can be calculated according to Eq. (1).

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

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

$$\approx \frac{\#TB}{E_{buf}} \cdot (16 \text{ B} + E_{buf} \cdot 20 \text{ B}) \quad (3)$$

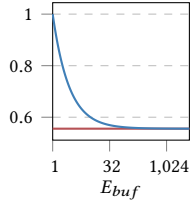


Figure 4: Reduction of the elog file size due to buffering.

The size S_{elog} is composed of a constant amount S_{conf} which is caused by the version and architecture information that is written once at the beginning. The workload-dependent amount is determined by the number of executed TBs ($\#TB$), the sizes of the `etrace_hdr` (S_{hdr}), `etrace_exec` (S_{exec}), and `etrace_entry64` ($S_{entry64}$) structs, and the number of `etrace_entry64` structs that fit into the buffer (E_{buf}). Since the constant part is relatively small and thereby negligible, the equation can be simplified to Eq. (3).

To estimate the influence of buffering on the elog file size, the ratio of the elog file size with buffering, S_{elog} , to the elog file size without buffering, $S_{elog}(E_{buf} = 1)$, can be calculated according to Eq. (4). The resulting relative reduction of the file size is plotted in Fig. 4. It shows that buffering and bundling of the `etrace_entry64` structs can reduce the file size of the elog file by up to 44%. If only 32 elements are bundled, the file size is already reduced by 43%. Further bundling has only a limited influence on the file size.

3.1 Multi-buffering

The buffering and bundling of `etrace_entry64` entries serve a dual purpose, benefiting not only in the reduction of the elog file size but also in enhancing performance-optimization possibilities. When the buffer is full, the contained data need to be written to the elog file. While this is done, QEMU is suspended which reduces the performance. To circumvent this issue, we suggest the implementation of an asynchronous writer pthread together with multiple buffers.

The concept is sketched in Fig. 5 for four buffers. Instead of a single buffer that is filled and flushed once it is full, we have multiple buffers. A state is assigned to each buffer which can either be *empty*, *filling*, *full*, or *flushing*. During the initialization of NQC², an asynchronous writer pthread is spawned (see Fig. 3). Each buffer can be accessed from two different pthreads, the writer or the main NQC² pthread, called *collector* in the following. At the beginning, all buffers are empty. The collector changes the state of the first buffer to filling. It adds `etrace_entry64` structs to the buffer until it is full. Then it changes the state to full. Once the next buffer is in the empty state, the collector changes the state to filling and continues. If the next buffer is in the full or flushing state, the collector needs to wait until the state is changed to empty by the writer.

The writer runs in parallel to the collector in the asynchronous pthread. At the beginning of the simulation, it waits until the collector changes the state of the first buffer to full. Then the writer updates the state to flushing, flushes the data into the elog file and sets the state to empty. It waits until the collector changes the state of the next buffer to full before it continues writing data to the elog file.

The waiting of the collector for an empty and the writer for a full buffer is synchronized using POSIX condition variables. Every time a state is changed from flushing to empty or from filling to

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

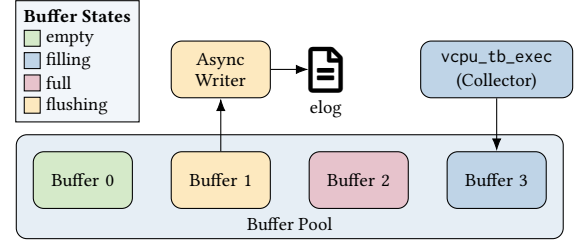
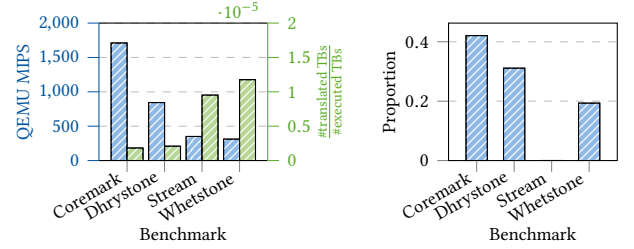


Figure 5: NQC² multi-buffering schema.



(a) QEMU execution statistics.

(b) Proportion of mergeable `etrace_entry64` blocks.

Figure 6: QEMU execution statistics.

full, a condition variable is notified to alert the potentially waiting collector or writer.

3.2 Merging

NQC² includes the merging of `etrace_entry64` structs as a second optimization. In the `vcpu_tb_exec` callback, before adding a new entry to the buffer, it is checked whether the buffer is empty. If that is not the case, the end address of the last element in the buffer and the start address of the executed TB are compared. In the case of a match, the last entry in the buffer can be updated instead of adding a new entry to the buffer. This is done by setting the end address of the last entry to the end address of the current entry. If timing is annotated, the execution duration of the current TB needs to be added to the `duration` field of the updated entry.

4 RESULTS

To measure the slowdown of NQC², we assess the code coverage of the widely-used bare-metal benchmarks Coremark [7], Dhrystone [19], Stream [12], and Whetstone [20]. All benchmarks are evaluated on QEMU version 8.1.1 [17] with and without NQC² enabled, and Xilinx's QEMU fork version 2023.1_update1 [21]. The simulated target architecture is *aarch64*. The used host CPU is an AMD Ryzen 9 3900X 12-core processor. When NQC² is loaded, the number of buffers, the number of `etrace_entry64` structs per buffer, and the merging option can be configured.

Fig. 6 shows general properties of the benchmarks and the execution by QEMU that are independent of the code coverage analysis. Fig. 6a depicts the simulation speed of QEMU for the different benchmarks measured in Million Instructions Per Second (MIPS). It can be seen that the reached simulation speeds differ for the benchmarks. Coremark and Dhrystone reach the highest simulation speeds. One reason for the high simulation speed is that both benchmarks are dominated by simple integer-arithmetic-based instructions [4]. The

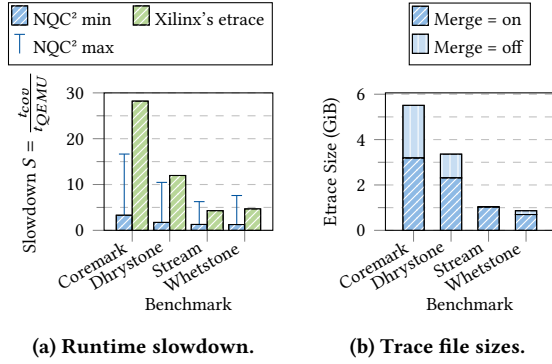


Figure 7: NQC² benchmark results.

higher execution speed for Coremark is caused by the lower amount of load instructions compared to Dhrystone. Stream mainly uses load and store instructions, which causes a higher slowdown. Whetstone consists of many floating point operations, which are more complex to simulate than integer operations.

Furthermore, Fig. 6a shows the probability that a TB needs to be translated before execution. Since QEMU buffers translated TBs, the probability that a TB has been translated in the past and can directly be executed is above 99.9% for all tested benchmarks. However, the TB-translation probability of Whetstone is more than 6.4 x higher than the one of Coremark. A lower TB probability can lead to a higher performance of QEMU due to the reduced translations.

Fig. 6b shows how often subsequent etrace_entry64 blocks can be merged (cf. Section 3.2). For example, 124,257,227 of the total 295,290,670 etrace_entry64 blocks (42.08 %) that need to be stored for a Coremark execution can be merged with their predecessor. In contrast, for the Stream benchmark, merging can only be applied to 10,481 of the 55,024,880 etrace_entry64 blocks (0.02 %).

Fig. 7a shows the runtime slowdowns NQC² and Xilinx's QEMU with enabled etrace cause compared to QEMU 8.1.1. Since the buffer count, buffer size, and merging can be configured for NQC² and influence the performance, the minimum and maximum values that can be achieved are shown. The evaluated number of buffers is between 1 and 16, the buffers can fit between 512 and 65,536 etrace_entry64 elements, and merging is enabled or disabled.

While Xilinx's etrace implementation causes enormous slowdowns by a factor of 28.2 for the Coremark benchmark (Fig. 7a), the slowdown of NQC² can be reduced to a maximum factor of 3.3 for all evaluated benchmarks. For the Coremark benchmark, NQC² outperforms Xilinx's solution by 8.5 x. A reason for the better performance of NQC² is that Xilinx's etrace implementation requires disabled TB block chaining to work. Furthermore, NQC² has, in contrast to Xilinx's implementation, multi-buffering and variable buffer-size capabilities. Xilinx uses a single buffer that can fit 16,384 etrace_entry64 elements. They use a single pthread.

The worst-case slowdown of NQC² exhibits a notable dependency on the executed workload. When we compare the trends depicted in Figs. 6a and 7a, a consistent pattern emerges in the course of MIPS values and the resulting slowdown. It seems that NQC² introduces a higher slowdown when it is used with workloads that can be executed with high MIPS values by QEMU. Tuning buffer

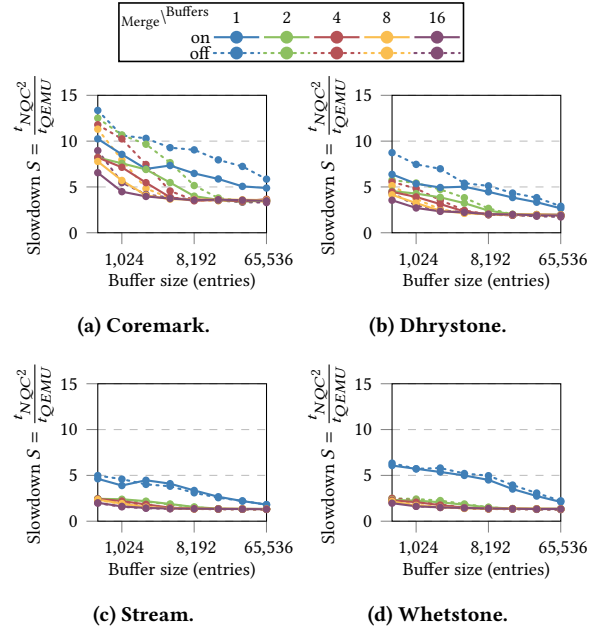


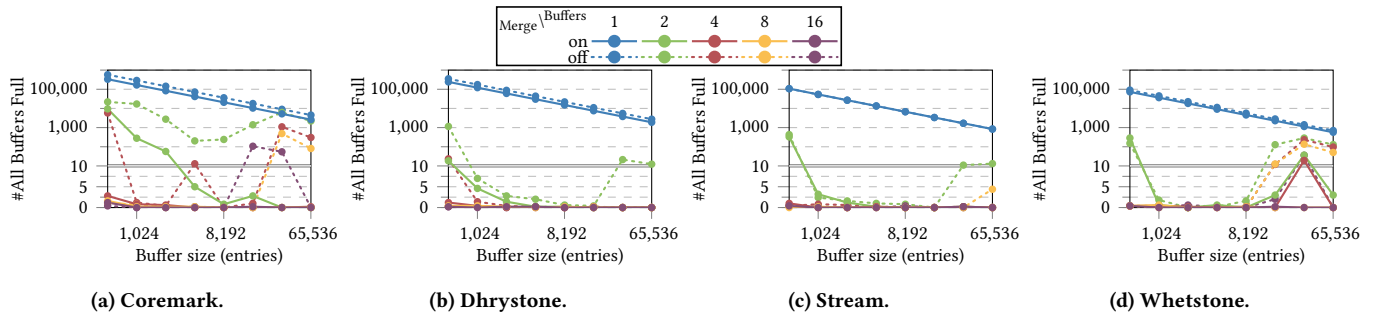
Figure 8: NQC² slowdown.

parameters and merge options can help to limit the slowdown. For the best-case scenario, NQC² always outperforms Xilinx's etrace.

Fig. 7b shows the file size of the elog file for enabled and disabled merging. According to Eq. (4), the file size also depends on the used buffer size. However, for the evaluated buffer sizes, which are larger than or equal to 512, the buffer size has a neglectable impact as shown in Fig. 4. Hence, this dependency is not further evaluated. It can be seen that the elog files can rapidly grow to multiple gigabytes. Merging helps to reduce the file size but the effectiveness depends on the workload. The impact of merging directly corresponds to the proportion of etrace_entry64 blocks that can be merged as depicted in Fig. 6b. Benchmarks that have a high probability that blocks can be merged, like Coremark and Dhrystone, can benefit from merging. Merging cannot reduce the file size for benchmarks with few mergeable blocks, such as Stream.

Fig. 8 shows the ratio of the needed execution time with enabled NQC², t_{NQC^2} , to the one without NQC², t_{QEMU} , for the different benchmarks and configurations. When only a single buffer is used (blue lines), the collector and writer cannot work in parallel which leads to a sequential behavior and thereby an extensive slowdown. The results of the Stream and Whetstone benchmarks show that the asynchronous writer combined with multi-buffering drastically improves performance. For those benchmarks, the gap between the blue and the other lines is the largest. An explanation for the larger influence observed for those particular benchmarks can once again be gleaned from the data presented in Fig. 6a. Workloads that reach a lower MIPS number, like Stream and Whetstone, benefit the most from parallel buffer filling and flushing. For those workloads, the execution performed by QEMU takes longer which leads to less frequent calls to NQC². Less frequent calls result in more time for the writer to empty a full buffer which reduces buffer congestion.

This explanation is underlined by Fig. 9, which demonstrates the impact of various buffer and merge configurations on buffer

Figure 9: NQC² buffer congestion.

congestion. It shows how often all available buffers are full, so the collector needs to wait for the writer to empty a buffer. Merging, as indicated in Fig. 6b, helps to reduce the congestion by decreasing the number of entries that need to be stored in the elog file.

In a single-buffer setup, the collector and writer cannot work in parallel, leading to waiting every time a buffer is full. Larger buffer sizes consistently reduce the number of filled buffers as reflected by the linear slope of the blue graphs. Increasing the number of buffers or the buffer size typically reduces the congestion probability. However, Fig. 9 shows that larger buffer sizes may increase the congestion probability for some workloads. This can be caused by the different points in time at which the buffers are emptied based on the used size. Another reason is that larger buffers reduce the swapping overhead of the collector which lowers the amount of time the writer has to flush a buffer without congestion. However, this slightly increased congestion probability does not lead to reduced performance as shown in Fig. 8. When the congestion probability reaches zero, further increases in the buffer sizes no longer affect the performance, as a comparison between Figs. 8 and 9 shows.

5 CONCLUSION AND FUTURE WORK

In the realm of software development, code coverage analysis is an essential practice that empowers developers to evaluate the effectiveness of their test suites, pinpoint untested code segments, and reveal performance bottlenecks. We present NQC², a QEMU-TCG plugin that enables instrumentation-free code coverage analysis for embedded software. Through the use of QEMU's plugin interface, NQC² is also compatible with customized QEMU versions. We presented the working principle, the structure of the elog file format and the integration into a TCG plugin. Our performance optimizations, such as an asynchronous writer, the usage of multiple buffers, and the merging of blocks, can significantly reduce the slowdown.

We evaluated the performance of NQC² using several benchmarks. It has been seen that the slowdown of NQC² highly depends on the executed workload. For benchmarks that reach a high simulation speed in terms of MIPS, the relative slowdown is higher. An asynchronous writer can noticeably reduce the slowdown, especially for benchmarks that reach lower MIPS values. Depending on the TB-execution order of the workload, the merging of entries can reduce the elog file size and can increase the performance.

In future work, on-the-fly compression of the elog file before writing or direct processing can be added to NQC² to reduce the

elog file size. In summary, NQC² presents a versatile solution for code coverage analysis in diverse QEMU implementations, with improved performance and a deeper understanding of the factors influencing its efficiency.

REFERENCES

- [1] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. California, USA, 10–5555.
- [2] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2020. Efficient binary-level coverage analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. <https://doi.org/10.1145/3368089.3409694>
- [3] Holger Blasum, Frank Görden, and Jürgen Urban. 2007. Gcov on an embedded system. (2007).
- [4] Nils Bosbach, Lukas Jünger, Rebecca Pelke, Niko Zurstraßen, and Rainer Leupers. 2023. Entropy-Based Analysis of Benchmarks for Instruction Set Simulators. In *Proceedings of the DroneSE and RAPIDO: System Engineering for constrained embedded systems (RAPIDO '23)*. <https://doi.org/10.1145/3579170.3579267>
- [5] Emilio G. Cota and Luca P. Carloni. 2019. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*.
- [6] Free Software Foundation, Inc. 2023. Gcov (Using the GNU Compiler Collection (GCC)). <https://gcc.gnu.org/onlinedocs/gcov/Gcov.html>
- [7] Shay Gal-On and Markus Levy. 2012. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* (2012).
- [8] Edgar E. Iglesias. 2023. edgarigl/qemu-etrace. <https://github.com/edgarigl/qemu-etrace> original-date: 2016-02-15T09:42:55Z.
- [9] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. <https://doi.org/10.1145/3338906.3340459>
- [10] Linux Test Project. 2023. LTP GCOV extension (LCOV). <https://github.com/linux-test-project/lcov> original-date: 2014-05-27T14:38:58Z.
- [11] LLVM Project. 2023. llvm-cov - emit coverage information — LLVM 18.0.0git documentation. <https://llvm.org/docs/CommandGuide/llvm-cov.html>
- [12] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.
- [13] Joan C. Miller and Clifford J. Maloney. 1963. Systematic mistake analysis of digital computer programs. *Commun. ACM* 6, 2 (Feb. 1963).
- [14] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. <https://doi.org/10.1109/SP.2019.00069> ISSN: 2375-1207.
- [15] NASA Jet Propulsion Laboratory. 2023. nasa-jpl/embedded-gcov. <https://github.com/nasa-jpl/embedded-gcov> original-date: 2022-02-02T19:25:26Z.
- [16] P. Piwowarski, M. Ohba, and J. Caruso. 1993. Coverage measurement experience during function test. In *Proceedings of 1993 15th International Conference on Software Engineering*. 287–301. <https://doi.org/10.1109/ICSE.1993.346035>
- [17] QEMU. 2023. QEMU. <https://github.com/qemu/qemu>
- [18] QEMU. 2023. QEMU TCG Plugins — QEMU documentation. <https://www.qemu.org/docs/master/devel/tcg-plugins.html#api>
- [19] Reinhold P. Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM* 27, 10 (Oct. 1984). <https://doi.org/10.1145/358274.358283>
- [20] Brian A. Wichmann. 1970. *Some statistics from ALGOL programs*. Central Computer Unit, National Physical Laboratory.
- [21] Xilinx. 2023. Xilinx's fork of QEMU. <https://github.com/Xilinx/qemu>