QEMTRACE: A MULTI-PLATFORM MEMORY TRACER BASED ON SYSTEM EMULATION

Alexy Torres Aurora Dugo
Jean-Baptiste Lefoul
Felipe Göhring de Magalhães
Gabriela Nicolescu

Department of computer engineering
Polytechnique Montréal
Montréal, QC, Canada
alexy.torres-aurora-dugo@polymtl.ca

Dahman Assal

MANNARINO Systems and Software
Montréal, QC, Canada
dahman.assal@mss.ca

ABSTRACT

Memory simulation is now widely used in system design processes. This process allows debugging, profiling and sometimes validating all memory-related tasks. The step before memory profiling is memory tracing, which consists in collecting memory events. State-of-the-art solutions rely on well-known environments provided by operating systems. Such environments might not be available when designing custom operating systems or bare-metal applications (e.g., applications directly running on hardware). In this context, aside from hardware-assisted approaches, no memory tracing mean is available. In this paper, we propose QEMTrace, a QEMU based open-source tool for memory tracing. QEMTrace provides a flexible way to trace memory accesses on operating systems, bare-metal applications and regular applications. After traces collection, we inject them in a cache simulator to show its usage. A shared memory interface to allow co-simulation of trace driven simulators is also presented.

Keywords: simulation, tracing, memory, QEMU, virtualization

1 INTRODUCTION

Memory accesses are the critical path to optimize when developing high performance applications (Desnoyers and Dagenais 2006). The increasing complexity of the memory hierarchy forces developers and software designers to take this memory hierarchy into account (e.g., cache locality, memory latency minimization). Depending on the application's memory profile, users will select multiple characteristics and parameters of the hardware supporting it. Moreover, memory accesses profiling helps better understanding software's behavior. We often use this method to verify and debug applications.

Profiling has also been used to increase security in the system. This method can be used to detect potential attacks. In (Cristalli et al. 2016) the authors use an hypervisor environment to detect and prevent memory-related attacks. In (Xu et al. 2017) memory tracing is used to detect the behavior of malware in the system. The authors analyze the memory access traces to provide a classification of software running in the system.

Memory profiling often use hardware and memory emulation systems to trace and analyze memory accesses. However, it is sometimes useful to rely on simulations to carefully analyze the behavior of a program or the hardware it is related to (memory buses, RAM, etc.). Such analysis is usually achieved thanks to the

simulation of memory components (caches, RAM, etc.). To run these simulations, we use memory traces as input data to drive the simulation. The hardware is designed at the same time as the software it will embed. During the HW-SW co-design process, simulators are used.

Few existing simulators (Binkert et al. 2011) allow generating memory accesses required by the simulated CPU. However, such simulators are slow to execute and impracticable when designing software. To overcome this issue, fast simulators / emulators were created. Unfortunately, they do not provide the ability to trace memory accesses. In this paper, we prose memory tracer embedded in the QEMU fast emulator. The traces are consumed "on the fly" by a trace driven simulator or can be stored for further use.

Memory tracing can be achieved through code instrumentation. This technique relies on source code modification. Developers add instructions in the programs to track memory accesses. Such methods are intrusive and may modify the behavior of the program. Binary rewriting uses additional instructions to gather memory events (Brais et al. 2019). Additionally, less intrusive (or non-intrusive) methods can rely on additional hardware (tracing probes) or virtualization (Qureshi et al. 2019). Those methods are non-intrusive but require additional resources such as hardware components or virtualization tools (virtual machine, emulators, ...). Usually, memory tracing introduces a computational overhead that will translate into an increased execution time and memory requirements.

In this paper we present QEMTrace, a QEMU based memory tracer. QEMTrace allows bare-metal and OS memory access tracing in a non-intrusive manner. It uses the virtualization provided by QEMU to collect information about the memory accesses. We also present the different trace collection means put in place and introduce a Shared Memory Interface (SMI) used to connect QEMTrace to any client application (simulators) that uses memory traces as input. Compared to previously proposed contributions, our approach allows a non-intrusive means to collect a wide variety of information concerning memory accesses such as their type, privilege level, or their size. QEMTrace is an open source tracing engine for QEMU that seeks for improvement and extension by the community. The GitHub repository is available in (A. Torres, et al. 2020).

The remaining of the paper is organized as follows. In section 2 we present the related work in the literature. In section 3 we introduce the approach used to develop the QEMTrace tool. Then section 4 presents the results we gathered using QEMTrace applied on a small bare-metal application. Finally, section 5 concludes our work.

2 RELATED WORK

Prior research proposed memory tracing tools capable of collecting multiple information about the accesses made by a program. In (Marathe et al. 2007), the authors propose to trace virtual addresses for each access and the address of the instruction generating the access. Such tracing is achieved thanks to dynamic binary rewriting. (Zhao et al. 2011) uses the same approach while collecting more data about accesses during the program's execution. The main advance brought by this tool is the exploiting of multi-core architecture to reduce the overhead induced by memory tracing. Along with the traced program, multiple threads, running on different cores perform trace analysis and storage.

In (Janjusic et al. 2013), the authors propose Gleipnir, a memory tracing framework that uses Valgrind (Nethercote et al. 2007) to parse debug information. Gleipnir is capable of tracing virtual and physical memory addresses. Even-though Gleipnir cannot trace the OS itself, it can trace multiple processes at the same time. Compared to our approach, this tool is only able to output textual based information of applications running on an operating system. To avoid the overhead of such output method, our approach relies on binary output. We also extend the tracing capabilities to any operating system of bare-metal applications.

More recently (Brais et al. 2019) propose a novel tracing approach based on dynamic binary rewriting. The authors describe a framework capable of extensive information collection. The tracing tool not only gather virtual addresses for each access, but also, among other information, the value of the accesses, their size and related instructions. Additionally, the authors define a novel adaptive trace management mechanism to reduce the tracing overhead. In (Rittinghaus et al. 2015) the authors propose a client-server based architecture. The tracing framework traces all memory access from the applications running in the system but also from the operating system itself. However, this approach requires the tool to be integrated with the operating system and run on top of it. Our approach is OS agnostic and does not require the OS to be compatible with our tracing tool.

Profiling and tracing methods can rely on virtualization and simulation to reduce the underlying tracing complexity and intrusiveness level. SNIPER (Carlson et al. 2011) is a system simulator that allows trace driven simulations. This feature enables system simulation based on memory access traces and execution traces. Along with other systems simulators (e.g., GEM5-X (Qureshi et al. 2019)) these tools provide convenient ways to collect execution traces (program execution flow, memory access traces, etc.). Operating systems aided tracing methods present in the literature rely on performance counters. In (Itzkowitz et al. 2003) the authors use such a mechanism to gather run-time information on applications.

Compared to prior work, our approach is capable of collecting memory traces for both data accesses and instruction fetches. We also enable the tracing of the OS itself and not only the applications running on it.

The existing memory trace approaches require their execution in a specific environment like an OS. To the best of our knowledge, there is no solution enabling memory tracing for bare-metal applications or in-development OS. The only solution applicable for these cases is the hardware probing. Tracing probes are expensive and may not be compatible with a wide variety or hardware platforms. Thus, memory tracing can become a huge part of the budget when developing bare-metal applications or OS on multiple platforms.

In this paper, we propose a method to trace memory access for a wide range of applications that do not have software tracing mean. We rely on the QEMU fast emulator to provide tracing means to any application running directly on hardware systems (bare metal applications or operating systems).

2.1 QEMU Fast Emulator

To accelerate the design and development of such applications, QEMU (Bellard 2005) offers a system-level emulation of many platforms. The emulator allows developers to design and tests their applications on many hardware platforms and CPUs. QEMU is an open-source software used and extended by thousands of collaborators.

In (Bellard 2005) the authors present QEMU, a system emulator. Since 2005, QEMU has been subject to constant improvements and is well known for its use in the academic and industrial worlds. This emulator focuses on performance instead of replicating each internal mechanism of the hardware. QEMU supports many processor architectures such as x86, PowerPC, ARM and RISC-V. Although a lack of support for some features (for instance Performance Monitoring Units are not emulated), one can easily modify the available code to add a peripheral, a feature or a new architecture to emulate.

Many research projects use QEMU and extend it to fit different needs. In (Patel et al. 2011), the authors propose a cycle accurate extension for the x86 platform based on QEMU. In (Becker et al. 2012), the authors present a framework that allows applying mutation to the software executed in QEMU. Such mutation enables extensive testing of the emulated software. Run time fault injection approaches sometimes rely on QEMU to test systems and application resilience (Höller et al. 2015).

QEMU offers two emulation modes:

- The user mode allows emulating the system and the guest operating system (executed as part of the emulated environment). This mode makes user mode applications easy to prototype and port on different architectures.
- The full system mode emulates the complete system, without any form of software attached to it. This is useful when developing bare-metal applications or operating systems.

To allow guest emulation, QEMU relies on online binary translation. This method consists of parsing the guest's binary code and translating it into binary executed by the host (the system that executes the emulator). The host code (generated by QEMU) has to manage the emulator's internal state. Thus a simple instruction for the guest application might result in multiple instruction to update the QEMU's internal state.

QEMU uses TBs (Translation Blocks) and a translation cache to improve simulation speed. A translation block is a sequence of guest instructions to be translated. When a TB is translated, QEMU stores its translation in the translation cache. If a similar block has to be translated in a near future, the translation routine recovers the stored TB from the translation cache.

3 QEMTRACE SOLUTION

In this section we introduce our approach and explain the design choices made. We further describe how to reduce the overhead generated by the memory tracing mechanisms. Finally, we introduce the SMI (Shared Memory Interface) library that allows the user to connect to the tracer via shared memory.

To leverage the issues presented in introduction, we propose QEMTrace, a memory access tracer based on the QEMU machine emulator. The aim of QEMTrace is to provide users with detailed memory accesses for bare-metal applications or operating system prototyping, debugging or profiling. This process is usually difficult to achieve or requires expensive hardware such as debug probes. We decided to base our memory tracer on QEMU since it is the emulator of choice for prototyping and debugging on multiple architectures. QEMU relies on dynamic binary translation to execute the guest code on the host machine. Dynamic binary translation allows us to inject tracing code that will be executed at run time without affecting the system's behavior. This method is further explained in the next section.

To control the tracing engine, we use special assembly instructions designed for this purpose. The instructions allow the user to start and stop memory tracing at any moment during the system's execution. Furthermore, the guest OS or application can propose an Application Programming Interface (API) to start and stop memory tracing when running on QEMTrace. We designed the instructions to be the same on all currently supported architectures.

We propose a trace structure that contains the memory access information. Each trace stores the address (virtual or physical depending on the user's choice) and its access time. The tracing routine also embeds different flags to add information about the access. Three gathering methods are available: file storage, trace print and shared memory storage. The latter uses a Shared Memory Interface (SMI) to store and shared traces. We provide users with a library allowing easy integration of the SMI with their applications.

3.1 Dynamic Code Translation

As explained previously in this paper, QEMU uses dynamic code generation to translate guest instructions to host instructions. We rely on this approach to integrate our tracing method.

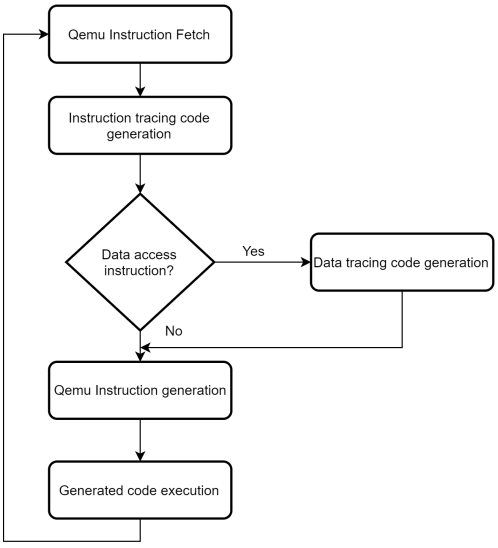


Figure 1: QEMTrace tracing call injection flow

At TB generation, QEMU generates different instructions to emulate the system's behavior. During this process, QEMTrace injects trace generation code that the host will execute. When the TB's instructions are executed, the injected code will call the QEMTrace engine to collect tracing data.

During emulation, the machine's state can be retrieved. This state consists of the CPU registers' values, the MMU (memory management unit) state and other hardware-specific information. The MMU state permits QEMTrace to translate the virtual addresses used by the CPU to physical addresses. We also rely on the MMU to infer trace metadata such as cache inhibition or cache coherency requirements. CPU registers are used to get the rest of the metadata such as the current privilege level or cache state.

To achieve instruction fetch tracing, the QEMTrace engine injects the access tracing instruction for each instruction fetched by the guest. We use the same flow for data tracing when the guest executes a memory-related instruction. This execution flow is shown in Figure 1. Instructions such as cache locking instructions also generate both an instruction fetch trace and the related special event trace (defined in the Trace Formats section).

To detect the access type (instruction fetch, data load, etc.), the QEMTrace engine offers different method calls in its API. The instructions fetches will be hooked to the instruction tracing methods while data load and data stores will have their handling methods. We also provide different handlers for machine-specific instruction (cache locking, cache flushing, etc.). Each API method will gather the machine state and set the trace flags (see the Trace Formats section) accordingly.

3.2 Traces Management Instruction

Users control the QEMTrace engine thanks to assembly instructions. We define multiple instructions opcodes to control tracing start point, end point, tracing type, etc. We use 32 bits opcodes to represent the instructions but this length be extended or reduced if needed. Trace management instructions are detected during QEMU binary parsing process. At instruction translation from the guest memory, QEMU loads the corresponding binary code. The parser reads binary code and detects the instructions to execute accordingly. To manage the QEMTrace engine, we added the detection of custom opcodes. When such instructions are detected, we call the QEMTrace routines instead of the QEMU regular code. The engine will trace these

```
# Enable tracing #
.long 0xFFFFFFF0
# Traced code #
li   r0, 0
stw r0, 0(r3)
...
# Disable tracing #
.long 0xFFFFFFF1
```

Figure 2: Tracing assembly flow, QEMTrace will register instruction fetches and data access from 4 to 8

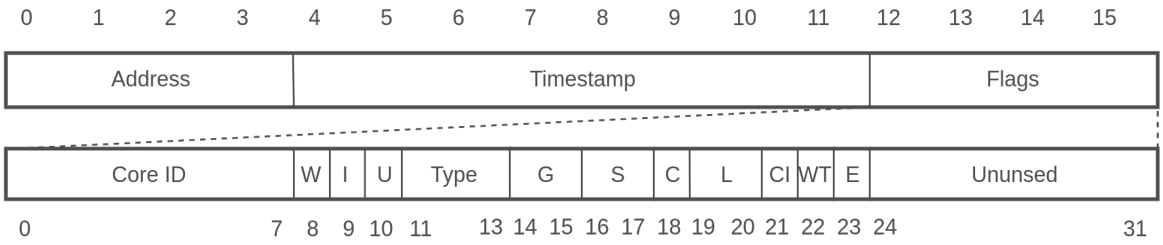


Figure 3: QEMTrace model definition

instructions, which means it will generate an instruction fetch trace. Figure 2 gives an example of assembly flow using the start and stop tracing opcodes.

We define the trace start opcode as *0xFFFFFFF0*. When executed by the host, the QEMTrace engine starts recording all memory accesses. The stop trace opcode is *0xFFFFFFF1*. Both instructions enable or disable tracing routine for all memory accesses executed by the guest. We use *0xFFFFFFF2* and *0xFFFFFFF3* to start and stop data only tracing. In the same way, *0xFFFFFFF4* and *0xFFFFFFF5* only control instructions fetches tracing. Note that using *0xFFFFFFF2* in conjunction with *0xFFFFFFF4* is equivalent to using *0xFFFFFFF0*. The tracing opcodes can be added anywhere in the OS or application's code. Additionally, the OS can provide a service API to manage QEMTrace. There is no limitation on where and how to use the opcodes.

All presented instructions are available in all CPUs privilege modes. Future improvement will study the addition of privilege detection to restrain the usage of the QEMTrace engine to a certain privilege level.

3.3 Trace Format

Three trace production means are available with QEMTrace. The first way is to store them in a file. The second way is to print them through the terminal. When using this method, both virtual and physical addresses are displayed. Finally, an external library enables the use of shared memory between QEMTrace and a client to recover traces. We explain this method in the next section. All output methods use the same trace formats (short and long). Figure 3 shows the different component of a single long access trace. The trace model is presented in Table 1. We leave the bits 24 to 32 free. Users can use the free bits to store additional metadata.

We also define a short trace format. In short traces, the address is on four bytes and the subsequent byte contains the flags. The first bit of the flags is the W bit, the second is the I bit and the next three are the TYPE identifier.

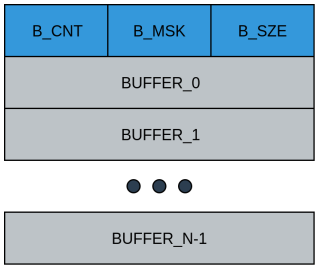


Figure 4: Shared memory sections used by the SMI interface

QEMU allows multi-core emulation. To leverage multi-core memory access tracing, we provide traces with the identifier of the core that requested the access. QEMTrace does not provide trace reordering and output memory accesses traces as they are done by QEMU. In its current version, QEMU serializes multi-core execution, which removes the need for trace reordering.

Finally, we define a file header as follows: The first 8 bytes give the size of the file in bytes; the two subsequent bytes give the size of one trace; the next byte contains the version of the file. We keep the next five bytes free for future use.

Table 1: Trace model fields description

Field	Description
Core ID	Contains the CPU identifier of the generated access. Such identifier is retrieved in different manners depending on the architecture that is traced.
W	The read/write flag. If the access is a memory read, this bit is clear (0); otherwise the tracer sets it to 1.
I	The instruction/data flag. If the traced access concerns data, this bit is clear (0); otherwise, this flag equals 1 if it concerns instructions.
U	The protection level flag. When the system is in kernel mode, this bit is clear (0); otherwise, this flag equals 1 when the access occurs in user mode.
TYPE	The special access flag that can take the following values: regular access (0), cache flush (1), cache invalidate (2), cache flush and invalidate (3), cache lock (4), cache unlock (5), cache prefetch (6), cache line clear (7).
G	The cache granularity flag. For special accesses (see TYPE flag), this flag takes the following values: line (0), set (1), way (2) or global (3).
S	The size flag. The size of an access can be the following: byte (0), half-word (1), word (2), double word (3).
C	The cache coherency flag. If the traced access is not coherent, this bit is clear (0); otherwise this flag equals 1.
L	The level flag. For special accesses (see TYPE flag), this flag takes the following values: all cache levels (0), L1 cache (1), L2 cache (2) or L3 cache (3).
CI	The cache inhibited flag. If access to this address is cache-inhibited, this bit is clear (0); otherwise this flag equals 1.
WT	The cache policy flag. When the cache operates in write-back mode, this bit is clear (0); otherwise, when set to write-through mode this flag equals 1.
E	The exclusive flag. If the traced access is not exclusive, this bit is clear (0); otherwise this flag equals 1.

3.4 Shared Memory Interface

One of the trace-gathering methods relies on shared memory between QEMTrace and an external client. This feature allows co-simulation of trace driven components. Applications of this feature can be the simulation of an external memory, caches or a memory-mapped IO. The QEMTrace engine uses buffers to store the generated traces while the client reads traces in a different buffer. Using multiple buffers increases throughput while enforcing exclusive access to any buffer.

To track the traces order and to ensure their correct transmission, we segregate the shared memory in $3+n$ sections where n is the required number of transmission buffers. n is determined by the user and depend on the number of clients that read the shared memory. The first three sections are used as a header to describe the interface state. Figure 4 gives a schema of the memory sections used by the SMI.

B_CNT represents the number of buffers to use with the SMI. Using multiple buffers increases the throughput of the tracer. While the QEMTrace engine uses a buffer, other full buffers can read by the SMI client.

B_MSK is a bit field that tracks which buffer contains traces that the SMI client can read. When the n^{th} buffer is full, the SMI engine sets the n^{th} bit of ***B_MSK*** to 1. Once the SMI client finish reading the buffer, this bit is cleared to 0. This action is transparent to the user when using the SMI library.

B_SZE gives the size of the buffers that store memory access traces. We express the buffer size in number of traces. The total memory usage of the SMI can be computed as in Equation 1 where $L(Trace)$ is the size of a trace and $L(Header)$ is the size of the SMI metadata header containing B_CNT, B_MSK and B_SZE.

$$Size = B\_CNT \times B\_SZE \times L(Trace) + L(Header) \tag{1}$$

BUFFER_n are the trace buffers. Each buffer is ordered in a FIFO (First In First Out) manner to keep traced access in the correct order. The SMI library offers users an API to manage the shared memory, read traces and manage synchronization between QEMU and the client.

3.5 Configuration

The QEMTrace engine can be configured to fit different needs. Users may modify different parameters such as the opcodes used to control the QEMTrace engine. The centralized configuration file provided at compilation time allows easy configuration of the tool. We also provide two flags to set when compiling QEMU and the QEMTrace engine:

- *QEM_TRACE_GATHER_META*: used to enable or disable access metadata gathering
- *QEM_TRACE_PHYSICAL_ADDRESS*: used to enable or disable memory address translation

QEM_TRACE_GATHER_META set to 0 will disable metadata gathering. The model used is the compact trace format. QEMTrace only traces the access type (data or instruction, read or write, special events) and its address. *QEM_TRACE_PHYSICAL_ADDRESS* set to 0 will disable memory address translation. All memory access addresses are not translated. When the user sets *QEM_TRACE_GATHER_META* to 0, *QEM_TRACE_GATHER_META* has no effect and memory access addresses are the address used by the CPU (virtual).

3.6 Overhead Reduction

When tracing memory accesses, we discovered the main performance bottleneck was traces storage. Although metadata gathering and trace generation processes introduce overhead, trace saving to the mass storage can take up to 75% of the tracing time. To reduce this overhead, we temporary store traces in a buffer. Users can set the buffer's size before compiling QEMU. When the buffer is full, the QEMTrace engine flushes the buffer to the trace file or SMI buffer. Once the buffer flushed, the engine resumes the QEMU execution as shown in Figure 5.

Based on this buffer architecture, we designed a multi-thread implementation of the trace buffer. We rely on a double-buffered approach. QEMTrace stores traces in the first buffer until it is full. Once the first buffer filled, the QEMTrace engine starts storing traces to the second buffer. An additional thread will proceed to flush the first buffer in the mass storage peripheral or an SMI buffer. This approach reduces the tracing

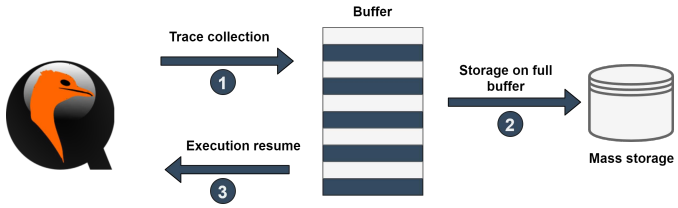


Figure 5: Trace storage execution flow

time by 25% according to our measurements. To get this result we compare the tracing time between trace storage with multi-buffer and with a single buffer

3.7 Cache memory simulation

We integrated our solution with the cache simulator proposed in (Lefoul et al. 2020). The tool uses QEMTrace to run a bare-metal application on the desired platform (here PowerPC). The application starts and stops memory tracing at different points in its execution. These specific points coincide with the cache activation and deactivation.

The cache simulator is linked to the QEMTrace tool through the shared memory interface. When a set of trace is sent to the SMI buffer, the cache simulator is able to retrieve and process the memory accesses. Each access either produces a hit or a miss depending on the cache state.

The simulator does not require the data manipulated by the program. Only the addresses and access features (cache inhibition, privilege level) are required to perform the simulation.

We compared the results of the cache simulation with the results obtained with a hardware probe. The system runs the exact same program and is not recompiled between the two architectures. Results show a difference of miss rate of 0.8%. This error rate takes into account false miss or false hits. The error rate also comes from the code or data prefetch complex processors put in place to increase performance. Such mechanisms are not simulated in QEMU.

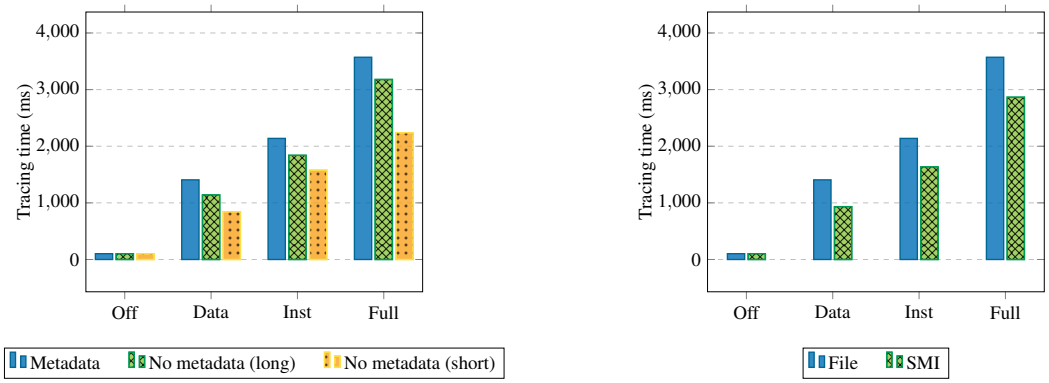
Our integrated platform can be further extended by adding other simulators (memory, buses, peripherals). These additional components can be plugged after the cache simulator in the hierarchy. The trace model can be extended to contain the value of the accesses data. However, the trace model should be kept standard for all components.

4 RESULTS

In this section we present the important results obtained when using QEMTrace. The system traced is a light PowerPC kernel. The kernel consists of booting the processor (PPC E500), correctly mapping the memory and launching an application. We use an application provided by the TACLeBench (Falk et al. 2016) benchmark suite. We choose to use the forward DCT function provided in the suite.

Figure 6 (a) shows tracing time that QEMTrace takes to trace the complete application execution in different contexts. We compared the tracing time (in milliseconds) when tracing all the memory accesses, only instruction fetches, only data accesses and when tracing is off. One can see that tracing introduces time overhead. This is mainly due to the trace storage process that is time consuming.

We compared the tracing time when QEMTrace collects metadata and when it does not. In this test, we keep the long trace model for both tracing methods. When metadata collection is disabled, the overhead reduction



(a) (b)

Figure 6: Tracing time in milliseconds, (a) shows the difference between the different trace types in presence of metadata or not (long and short trace format). (b) shows the tracing time difference between file storage and SMI storage (long trace format)

Table 2: Performance measurement on TACLeBench benchmarks

Application	Large traces (LT)	Short traces (ST)	Trace count	LT trace time	ST trace time
adpcm_dec	957.04 MB	299.07 MB	62,720,703	6946.21 ms	4270.74 ms
binsearch	28.38 MB	8.86 MB	1,859,933	1257.58 ms	773.19 ms
bitcount	1225.2 MB	382.87 MB	80,294,581	8513.41 ms	5234.30 ms
dijkstra	2384.89 MB	745.28 MB	156,295,915	15062.62 ms	9261.16 ms
fir2dim	376.02 MB	117.50 MB	24,643,397	3668.57 ms	2255.55 ms
iir	103.66 MB	32.39 MB	6,793,271	1883.79 ms	1158.21 ms
insertsort	105.6 MB	33 MB	6,920,577	1990.05 ms	1223.54 ms
ndes	71.71 MB	22.41 MB	4,699,627	1395.24 ms	857.86 ms
prime	2.68 MB	0.84 MB	176,010	959.45 ms	589.90 ms
recursion	147.44 MB	46.08 MB	9,663,093	1885.11 ms	1159.02 ms
sha	1333.13 MB	416.61 MB	87,368,078	10,354.91 ms	6366.51 ms

Table 3: Trace file size evolution in function of the trace type

Trace type	Large traces	Short traces	Number of traces
Data only	213.5 MB	66.3 MB	13,343,784
Instructions only	427.8 MB	133.7 MB	26,740,084
Full tracing	641.3 MB	200 MB	40,083,868

reaches 10% on average. This shows that even if metadata collection is not the most time-consuming process, its overhead is non-negligible. Finally, we show that using short trace will reduce the tracing time by up to 40%, demonstrating that trace storage is the most time-consuming process. In Figure 6 (b), we compare the tracing time between two storage methods. The first is file storage and the second uses the SMI to send the collected traces. Both methods collect accesses metadata. When using SMI the tracing overhead drops by 20% on average.

Table 2 shows the different performance measures we gathered using QEMTrace. We selected applications from the (Falk et al. 2016). Results show that only the number of traces impacts the tracing time, not the type of application.

Finally, we compared the stored trace size between the different tracing methods. One has to note that disabling metadata gathering will reduce trace size. Table 3 shows the evolution of the trace file size depending on what is traced in the system.

5 CONCLUSION

In this paper we presented QEMTrace, a tracing engine included in the QEMU system emulator. Our work allows tracing memory accesses from the system's point of view and in a non-intrusive manner. QEMTrace provides a novel approach to profile memory accesses made by bare-metal applications and custom-made OSes. We offer multiple ways to collect data and allow in depth memory access attributes gathering that we call metadata. Such metadata, coupled with the presented Shared Memory Interface (SMI) allows co-simulating peripherals during the QEMU system's execution. The use of compact traces allow a reduction of the trace time by up to 40% and relying on a shared memory interface reduces it by up to 60%.

QEMTrace allows the user to select which access type (data or instruction) should be traced and is accessible both in user and kernel mode. The software is in constant development and is open-source (A. Torres, et al. 2020) to allow the community to participate in its improvement and extension.

QEMTrace is an open source tracing engine for QEMU that seeks for improvement and extension by the community. To add a new architecture, one will have to rely on the QEMTrace API to instrument the architecture of their choice. The users need to add an API call when the emulated architecture loads or stores data as well as when it fetches a new instruction to execute. The GitHub repository is available in (A. Torres, et al. 2020). Further improvement are (but are not restricted to):

- Add customized trace model fit the user's need. One can use a configuration to define the trace model and the metadata needed
- Allow condition tracing: only trace accesses within a certain address range and under certain conditions (privilege level, register values, etc.)
- Extend the SMI to allow full co-simulation. At the moment only one way simulation is possible (from QEMU to the SMI client), but one could add the possibility to interact with QEMU through the SMI.

ACKNOWLEDGMENT

The authors would like to thank NSERQ, MITACS and CRIAQ for their support. We would also thank the reviewers for their insightful remarks to improve this paper.

REFERENCES

A. Torres, et al. 2020. "GitHub repository: QEMTrace". https://github.com/HESL-polymtl/QEMTrace.

Becker, M. et al. 2012. "XEMU: An Efficient QEMU Based Binary Mutation Testing Framework for Embedded Software". In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12.

Bellard, F. 2005. "QEMU, a Fast and Portable Dynamic Translator". In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05.

Binkert, N. et al. 2011. "The Gem5 Simulator". *SIGARCH Comput. Archit. News*.

Brais, H. et al. 2019. "Alleria: An Advanced Memory Access Profiling Framework". *ACM Trans. Embed. Comput. Syst.*.

Carlson, T. E. et al. 2011. "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*.

Cristalli, S. et al. 2016. "Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks". In *25th USENIX Security Symposium (USENIX Security 16)*.

Desnoyers, M., and M. Dagenais. 2006. "The LTTng tracer : A low impact performance and behavior monitor for GNU / Linux". In *OLS (Ottawa Linux Symposium)*.

Falk, H. et al. 2016. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research". In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*.

Höller, A. et al. 2015. "QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks". In *Proceedings of the 2015 Euromicro Conference on Digital System Design*.

Itzkowitz, M. et al. 2003. "Memory Profiling Using Hardware Counters". In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*.

Janjusic, T. et al. 2013. "Gleipnir: A memory profiling and tracing tool". *ACM SIGARCH Computer Architecture News*.

Lefoul, J. et al. 2020. "Simulator-based framework towards improved cache predictability for multi-core avionic systems". In *Proceedings of the 2020 Spring Simulation Conference*.

Marathe, J. et al. 2007. "METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies". *ACM Trans. Program. Lang. Syst.*.

Nethercote, N. et al. 2007. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In *ACM Sigplan notices*.

Patel, A. et al. 2011. "MARSS: A full system simulator for multicore x86 CPUs". In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

Qureshi, Y. M. et al. 2019. "Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms". In *2019 Spring Simulation Conference (SpringSim)*.

Rittinghaus, M. et al. 2015. "Simutrace: A toolkit for full system memory tracing". *White paper, Karlsruhe Institute of Technology (KIT), Operating Systems Group*.

Xu, Z. et al. 2017. "Trace-based Analysis of Memory Corruption Malware Attacks". In *Hardware and Software: Verification and Testing*.

Zhao, Q. et al. 2011. "PiPA: Pipelined Profiling and Analysis on Multicore Systems". *ACM Trans. Archit. Code Optim.*.

AUTHOR BIOGRAPHIES

ALEXY TORRES is a PhD student at Ecole Polytechnique de Montréal. His email is alexy.torres-aurora-dugo@polymtl.ca.

JEAN-BAPTISTE LEFOUL is a PhD student at Ecole Polytechnique de Montréal. His email is jean-baptiste.lefoul@polymtl.ca.

FELIPE MAGALHAES is a postdoctoral fellow at Ecole Polytechnique de Montréal. His email is felipe.gohring-de-magalhaes@polymtl.ca.

DAHMAN ASSAL is a Senior Software Designer / Technical Specialist at Mannarino Systems & Software Inc. in Saint-Laurent, Québec, Canada. His email is dahman.assal@mss.ca.

GABRIELA NICOLESCU is a Professor at Ecole Polytechnique de Montréal. Her email is gabriela.nicolescu@polymtl.ca.