FRACTAL: Fast Reverse Address translation over Coherence for Tracing And Logging

Francesco Ciraolo
Boston University
Boston, USA
fciraolo@bu.edu

Patrick Carpanedo
Boston University
Boston, USA
pfcarp21@bu.edu

Daniele Ottaviano
Technical University of Munich
Munich, Germany
daniele.ottaviano@tum.de

Matias Ou
Boston University
Boston, USA
matiasou@bu.edu

Marco Caccamo
Technical University of Munich
Munich, Germany
mcaccamo@tu.de

Renato Mancuso
Boston University
Boston, USA
rmancuso@bu.edu

***Abstract*—Fine-grained tracing of memory activity is useful for analyzing software behavior and understanding software-hardware interactions. Existing approaches rely either on software instrumentation, which introduces high overhead, or on architecture-specific hardware tracing mechanisms with limited flexibility and filtering capabilities. In particular, current solutions provide limited support for process-specific filtering and/or simultaneous visibility of virtual and physical addresses.**

This paper presents FRACTAL, a hardware module that operates in the coherence domain and enables tracing of memory transactions for a target process. FRACTAL passively observes coherence traffic and performs reverse address translation to reconstruct virtual addresses from physical memory accesses. To achieve this, it follows the page table walks of the monitored process and maintains a reverse translation structure that mirrors the processor TLB. The design is largely microarchitecture-agnostic, does not require binary instrumentation, and does not affect the performance of the traced application.

We implement FRACTAL on a commercial heterogeneous multiprocessor system-on-chip platform, the KRIA KV260, with programmable logic integrated in the cache-coherent interconnect. Experimental evaluation using vision benchmarks shows that the system can do fine-grained tracing of selected virtual address ranges with negligible overhead. The results demonstrate that coherence-based observation combined with reverse translation is a practical approach for low-intrusion memory tracing.

***Index Terms*—Memory Profiling, FPGA, Cache Coherence, Reverse Address Translation, Hardware Tracing, Performance Monitoring**

I. INTRODUCTION

Leading efforts in the systems community have highlighted the importance of low-overhead fine-grained tracing of applications [1] [2]. Fine-grained application tracing refers to the ability to capture, in real time, memory accesses and/or instructions executed by the application with rich information about their nature, timing, and location. This capability enables researchers and system engineers to study software-hardware interactions in detail, revealing optimization opportunities, performance bottlenecks, and execution anomalies that enable the design of faster, more efficient, and interpretable systems.

The earliest efforts to provide insight into the program execution come from software instrumentation. Tools like Pin [3] and Valgrind [4] dynamically insert tracing logic to capture the control flow of the program being executed. However, they come with serious overheads. Pin reports slowdowns of 1.4x to over 20x [3] depending on the benchmark and optimization level, while Valgrind incurs a 4.3x to 22.2x slowdown [4]. This *probing effect* fundamentally distorts the temporal behavior being measured. Furthermore, these tools operate entirely in the virtual address space and cannot observe physical memory behavior, providing an incomplete view of system behavior.

Recognizing these limitations, processor manufacturers have integrated hardware debugging support directly in/into silicon. Performance monitoring units (PMUs) [5] are widely implemented but often provide hardware counters that track only aggregate metrics (e.g., cycle counts and cache misses) with negligible overhead, typically below 2% [6]. While useful for coarse-grained profiling, PMUs are unable to attribute counter values to specific instructions or memory locations. To address these limitations, modern architectures include dedicated tracing infrastructure. Intel introduced Processor Trace (PT), which records branch outcomes to enable full control flow reconstruction with low overhead [7]. Similarly, ARM's CoreSight provides embedded trace macrocells (ETM) for instruction and data trace for real-time debugging and verification [8]. In addition, both Intel- and ARM-based systems have introduced hardware-level support for statistical profiling via Intel Timed Process Event-Based Sampling (TPEBS) and ARM Statistical Profiling Extension (SPE). TPEBS and SPE are state-of-the-art mechanisms capable of collecting a set of records about a class of instruction of interest, such as branch instructions or memory instructions.

Motivation. Unfortunately, even state-of-the-art support listed above is not suited for a number of important analysis scenarios. First, consider the case of live application progress tracking. In this case, a key capability is the ability to install instruction watchpoints at many points in the control flow graph of an application, which constitute important *progress milestones* [9]. Intel PT and ARM ETMs allow specifying a handful (typically 4) of instruction ranges for which tracing

TABLE I: Qualitative comparison of proposed FRACTAL vs. state-of-the-art methods.

Feature	FRACTAL	Intel PT	ARM SPE	Intel PEBS	AMD IBS	ARM ETM	AccProf
Monitoring Fundamentals							
Placement	**Coherence Port**	Inside Core	LSU/Pipeline	PMU	PMU	Inside Core	PMU
Observation	**Passive Sniffing**	Logging (Trace)	Sampled (1 every N)	Sampled (1 every N)	Sampled (1 every N)	Signal Trace	Injected Instructions
Primary Data	**Coherence Traffic**	Control Flow	Sampled LD/ST	Event Samples	Event Samples	Data/Inst. Trace	Event Samples
Visibility	**LLC Misses/Snoops**	Every Branch	Statistical	Selected Events	Selected Events	Full Stream	Selected Events
VA vs. PA	**VA + PA**	VA only	Both (Sampled)	VA + PA	VA + PA	VA only	VA only
Data Bandwidth	**Low (Filtered)**	Very High	Medium	Very Low	Very Low	Extremely High	Low
System Integration & Scalability							
Perf. Overhead	$\sim 8\%$	1–5%	$< 1\%$	$< 1\%$	$< 1\%$	$\sim 0\%$	N/A
Multi-Range Track	**High**	Low (1-2 pairs)	Low (Fixed num.)	✗	✗	Low (1-4 pairs)	Static (compile time)
Obs. Range Reconfig.	✓	✗	✗	✗	✗	✗	✗
Micro-arch. Indep.	✓**(ACE/CHI/CXL)**	✗	✗	✗	✗	Medium	✗
HW Extensibility	✓	✗	✗	✗	✗	✗	✓
Chiplet/CXL Ready	**Native**	✗	✗	✗	✗	Limited	✓

is enabled. Changing which addresses are monitored requires disabling, reconfiguring, and re-enabling the tracing subsystem, resulting in a monitoring *blackout window* [9]. Conversely, Intel TPEBS and ARM SPE support limited filtering in hardware, with dynamic changes to monitored address ranges causing blackout-window effects. Moreover, all the aforementioned solutions are architecture-specific.

Contribution. In this paper, we propose FRACTAL, short for *Fast Reverse Address translation over Coherence for Tracing And Logging*: the blueprint and proof-of-concept implementation for an off-core hardware module capable of performing low-overhead, dynamically programmable memory profiling for a target application. Compared to similar hardware modules, the proposed FRACTAL is unique for the following reasons. First, it relies on predominantly passive analysis of coherence traffic to collect information about memory transactions—this makes the module microarchitecture-agnostic. Second, it employs a novel approach, based on MMU translation tracing, to filter out traffic that does not belong to the workload of interest. Third, it leverages a novel approach to perform on-the-fly reverse address translation, recovering virtual addresses from physical addresses, enabling effortless logging of both virtual and physical addresses.

We also propose an FPGA module that implements the FRACTAL model and acts as an manager on the coherency domain. It can be instrumented to perform Reverse Address translation on a VAs' range for a specific target process. The data it collects enable users to specify application milestones using VAs, construct Heatmaps of addresses accessed, and tie the latter to their representation in the code.

II. Background

Multiple software (SW) based tools have existed throughout the years [10] providing fine-grained monitoring of memory/instruction events in relation to specific instructions in code. Similarly, x86 designers have provided in-house SW solutions that leverage their own hardware [11] [12] which has been mirrored by ARM with [13]. However, even hardware-assisted profiling via PMU counters and event-based sampling introduces measurable overhead [6] [14], motivating the dedication of additional transistor budgets towards debugging and profiling infrastructure to benefit the overall implementation of processing elements (PE). ARM allotted a considerable amount of the budget towards the complex Coresight Debug Infrastructure [15] [16]. This infrastructure at the basic level is capable of reporting branch instructions statistics and, if implemented, the memory address related to those instructions. It is capable of being configured through a 3rd party debugger [17] or internally through software setup [18]. The infrastructure is capable of stalling specific cores to install hardware breakpoints without software instrumentation. Despite these capabilities, CoreSight does come with a number of disadvantages:

- The implementation of the infrastructure can be partial or not documented
- Usage is difficult given the complexity and modes of the infrastructure
- There is a bandwidth limit imposed by each module in the infrastructure

Similarly, Intel PT offers branch instruction tracing [7], while Intel PEBS [11] and AMD IBS [12] provide hardware-assisted sampling of individual instructions with micro-architectural event attribution. AMD also introduced Lightweight Profiling (LWP), though documentation is sparse and adoption is limited. These mechanisms are architecture-specific and offer little to no address range filtering, and their flexibility is limited in comparison to Coresight.

Community efforts have instrumented FPGAs to serve as a custom HW profiler. There have been multiple efforts on augmenting the existing infrastructure to provide additional abilities [8]. Others have shown in-depth methods of creating a custom debug infrastructure on FPGA-based systems to precisely define the information wanted [19]. A middle ground is taken by AccProf [20], which offloads profiling computation to an FPGA to reduce cache pollution, but still requires compile-time binary modification via a custom LLVM pass. None of the aforementioned approaches provides passive, binary-

transparent profiling with simultaneous VA and PA visibility.

Most of the commercial-off-the-shelf (COTS) boards that offer Programmable Logic (PL) alongside traditional CPU clusters utilize a snoop-based coherence protocol between the processing cluster(s) and the PL to provide system-wide cache coherence. While such protocols can be a powerful tool to trace CPU activity [21], they operate with physical addressing only. Thus, naïvely logging snoop traffic is only useful in systems with a static virtual-to-physical mapping.

Table I offers an overview of some solutions discussed in this section in comparison to FRACTAL.

III. Design

The design of the proposed FRACTAL, is focused on portability, requiring only: *(i)* a slice of programmable logic (PL) with a snoop-based cache coherent interconnect with the main Processing Element (PE), *(ii)* to observe the coherence traffic and *(iii)* optionally, to issue Distributed Virtual Memory (DVM) and cache maintenance commands.

Most of the commercial off-the-shelf that satisfy *(i)* also satisfy the other requirements. Boards from families such as the Xilinx Ultrascale+ ZynqMP and the Altera Stratix 10. Furthermore, the x86 space offers CXL 2.0 (a cache-coherent interconnect standard) compatible platforms with companies (e.g. Supermicro, Dell, etc) providing motherboards and CXL-compatible accessories to utilize the standard.

FRACTAL and its interaction with a multi-core processing cluster is depicted in Fig. 1. For simplicity, we consider a FRACTAL module that handles a single **target application/process**. Simultaneous handling of multiple processes is beyond the scope of this proof-of-concept implementation. The target process is identified by its Page Global Directory (PGD) and Address Space ID (ASID). Moreover, a range of Virtual Addresss (VAs) of interest can be defined to focus the observations collected by the system.

FRACTAL operates, mainly, as a *passive* observer of the CPU's activity, and its internal modules partially mirror some of the CPU's ones. With reference to the overview shown in Fig. 1, a description of FRACTAL's key sub-modules and their interactions with the processing cluster follows.

A memory-mapped **Configuration port** allows the user to (1) specify the target process and virtual address range of interest and (2) to tweak with the internal configuration to leverage between trace accuracy and overhead. FRACTAL can be configured to passively log VA-resolved memory transactions and to install specific observation points. Passive logging corresponds to the least intrusive configuration, in which the FRACTAL module initiates no explicit interactions with the CPU, resulting in low performance overhead all the while collecting a complete trace of LLC misses. If more precise tracing of specific memory objects (at the cacheline granularity) is desired, FRACTAL can be configured to install multiple Observation Points (OPs). The observability of OPs is ensured by FRACTAL through **active invalidation** of the traced cachelines from the processing cluster's caches.

The core of the proposed methods is the **Reverse MMU (rMMU)**, further divided into two sub modules; It mirrors the MMU's purpose by collecting and maintaining a set of Physical Address (PA) to VA mappings. Upon observing the snooped PA, a naïve approach to obtaining the associated VA would require a reverse walk through the page tables of the target process. Unfortunately, however, a reverse page table walk is an expensive operation. As such, the **Reverse MMU (rMMU)** must rely on an alternative strategy to efficiently perform a reverse PA-to-VA translation on the fly for each captured snoop.

On the other hand, it is possible to leverage the method described in [22] to follow the MMU's page walks and reconstruct the VA associated with each Page Table Entry (PTE). This task is delegated to the **Translation Stalker (TS)**, which transparently participates in the coherence exchanges, identifies the snoops that are *(i)* MMU's read requests to translate *(ii)* a VA in the range of interest for *(iii)* a target process. Note that by following each CPU translation walk, the TS always obtains the most up-to-date mapping possible, accounting for changes, e.g., in case of page migrations. The overhead on the CPU by employing such a strategy is negligible, and the results are cached in the Translation Lookaside Buffer (TLB) to avoid too-frequent walks.

Symmetrically, the **Reverse TLB (rTLB)** is a cache structure that stores the (reverse) mappings, indexed by PA, and enables the other elements of FRACTAL to look up and quickly recover the corresponding VAs. Once the **TS** produces a complete mapping, a **refill** operation will store it in the rTLB. This element does not require a huge amount of memory, since its size can be comparable, if not smaller, than the size of the monitored CPU's TLB, since with high probability the VA/PA mapping in the CPU's side will be evicted before the one in the rTLB.

At FRACTAL setup time, to guarantee that the selected VA range can be correctly and efficiently resolved, we must ensure that the TS is able to observe, at least once, a full page table walk when the corresponding PAs are accessed for the first time. As such, FRACTAL performs a full TLB flush directly via the coherence channel. This allows the relevant rTLB entries to be incrementally populated as the application accesses the monitored VA range.

Later on, if due to rTLB space constraints one of its entries needs to be evicted, FRACTAL forces the content of the rTLB to remain in synch with that of the TLB. Specifically, for each evicted rTLB entry, FRACTAL issues two invalidation commands towards the processing cluster directly through the coherence channel: *(i)* one to the TLB, to invalidate the evicted VA, and *(ii)* a *MakeClean* targeting the cache line containing the evicted PTE. This **active eviction** process necessitates that the CPU will perform at least the last step of the translation walk at the next access to the VA, allowing the TS to recapture the mapping and re-populate the corresponding rTLB entry.

The **Snoop Listener (SL)** is responsible for obtaining and filtering the PA being read by the CPU; it observes traffic on the coherence channel and filters read snoops—namely,

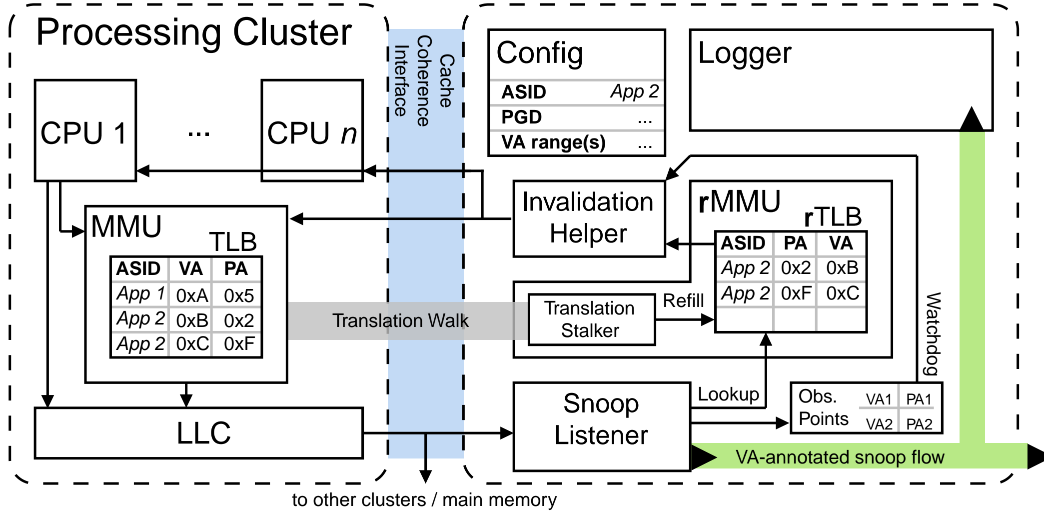

Fig. 1: Overview of FRACTAL's design, on the right, interactions with the PE and internal components connections.

excluding Distributed Virtual Memory (DVM) traffic and cache-maintenance messages. Then, for each snooped address, it performs a lookup in the rTLB, both to identify the ones of interest and to obtain the related VA; in parallel, the SL also feeds the filtered PA to the OPW (discussed below). Finally, in case of a rTLB hit, it emits a VA-annotated snoop for logging and/or for further analysis downstream. Note that because TLB and rTLB are kept in synch as discussed above, a rTLB miss guarantees that the snooped PA does not correspond to any of the VAs of interest.

The infrastructure described so far enables efficient tracing and resolution of any relevant snoops that naturally result from a LLC miss in the processing cluster. Furthermore, FRACTAL also allows users to specify a set of VAs (at the cache-line granularity) for which every CPU-initiated access should be logged. These are effectively treated as *observation points* (OPs) and handled by the **Observation Points Watchdog (OPW)** sub-module. OPs have a dedicated mapping storage that is updated contextually upon rTLB **refill**. To ensure that OPs are logged at every access, the OPW issues corresponding cache invalidation requests at OP setup time and after every OP hit through the coherence channel.

The **Invalidation Helper (IH)** serves as a cache-coherent protocol adapter to handle invalidation requests issued by the OPW or the rTLB. Note that the abstraction of architecture-dependent aspects—e.g., the exact format of invalidation requests, page table entries, and snoop transactions—is handled by the IH and TS modules which allows FRACTAL to operate fully independent and performant despite being *off-core*. Despite the lack of portability of IH and TS, they allow the rest of the logic to remain architecture-independent.

Finally, the VA-annotated snoops can be stored by the **Logger** in a slice of BRAM or directly into the system DRAM. The stored entries can be customized as needed, allowing the storage of physical and virtual addresses, timestamps, and other metadata. Opting for a BRAM slice limits the trace size but prevents FRACTAL from adding any stress on the main memory bus.

IV. Implementation

The KRIA KV260 was chosen as the platform for instantiating and evaluating FRACTAL. This platform contains the AMD/Xilinx Zynq™ UltraScale+™ MPSoC System-on-Module which is composed of four ARM Cortex-A53 operating at 1.3 GHz with 1 MB of shared unified L2 cache, two ARM Cortex-R5, and an FPGA block [23]. The FPGA is capable of communicating directly to the main memory controller via a dedicated link using the memory-mapped Advance eXtensible Interface (AXI) protocol. Furthermore, communication with the CPUs through the coherent domain is accomplished with dedicated ports on the CCI-400 [24] which uses an AXI Coherency Extension (ACE) interface. This allows The FPGA to snoop coherent memory transactions transparently and to directly initiate memory fetches without the involvement of any other PEs. Our system is synthesized on the FPGA using Vivado 2024.2, and the resource utilization, including a BRAM sized 256K used to store the trace entries, is shown in Table II.

The implementation follows the design described in Sec.III with some adaptation required by the platform's hardware: on the UltraScale+ MPSoCs the Cache Coherent Interconnect (CCI) does not support parallel read requests while a coherent snoop is pending. We therefore altered part of the design to parallelize the accesses.

Our proposed implementation follows the leads of [22]. A few implementation details of the core modules, especially the TS, are noteworthy. The rMMU identifies the beginning of a translation of interest by matching the PGD of the target process and proceeds following the translation through a system of watermarking for the intermediate tables. TS also performs the VA's reconstruction based on the read address

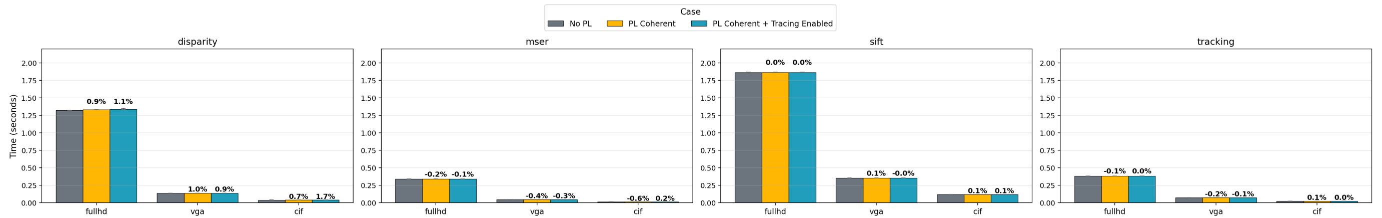

Fig. 2: Exec. time overhead of selected vision benchmarks across three configurations: (i) without PL in the coherence domain, (ii) with PL attached to the coherence domain but tracing disabled, and (iii) with PL enabled and tracing active using FRACTAL.

TABLE II: PL resources used by FRACTAL on a KV260.

FPGA Utilization				
	LUT	LUTRAM	BRAM	FF
Units Used	22257	5170	64	22229
% Used	19.00%	8.98%	44.44%	9.49%

and the data content: *(i)* from offset of the PA, it can extract the index to a cache-line precision, and *(ii)* enumerating the entries in each line, it can fill the VA's missing bits.

The IH allows the other modules complete independence from the specific coherence protocol and enables full portability. This is obtained by offering a generic invalidation interface, limited to the messages' types of interest for the rTLB and the OPW—namely TLB invalidation by VA, data cache invalidation by PA, and instruction cache invalidation by VA—and by translating those to the ones specific to the underlying coherence protocol.

In the following sections, FRACTAL will be evaluated in terms of overhead and tracing capabilities using RT-Bench [25]. The source had minor changes to integrate a library that interfaces with FRACTAL with user-space functions. RT-Bench is an open-source framework that adds standard real-time features and a unified interface for running benchmarks from suites such as the San Diego Vision Benchmark (SD-VBS) [26], the TACLe Benchmark [27], and others.

V. Evaluation

We propose two experimental setups that we carried out to prove the core statements of this work: the first one verifies the effective overhead of the machinery on a subset of the San Diego Vision Benchmark Suite (SD-VBS); we then switch the focus to the evaluation of instruction tracing capabilities, instrumenting FRACTAL to trace the execution of the `canny filter` benchmark.

The benchmarks are compiled within the RT-Bench framework, the latter is extended with a library to simplify the interaction with FRACTAL, with no changes required to the benchmark code for the tracing of any VAs' range.

In all setups, FRACTAL is configured in an intermediately intrusive mode. Namely, invalidation messages are issued to the CPU only if OPs are installed, and one of them hits. In such a context, we can verify the *worst-case* tracing capabilities and showcase how, with negligible overhead, it is possible to achieve meaningful observability. The FPGA bitstream is synthesized at a frequency of 100 MHz.

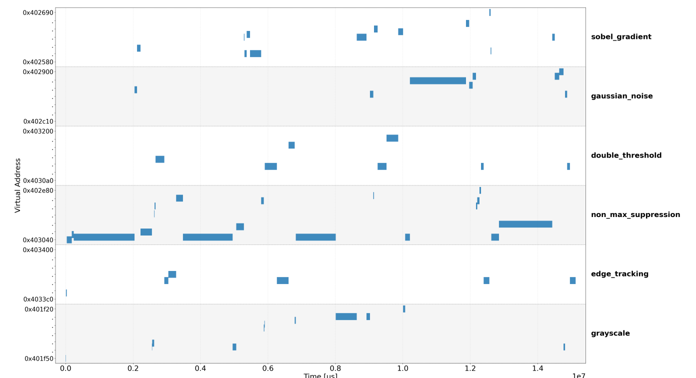

Fig. 3: Progression of *canny* image filter traced via FRACTAL

A. Overhead

From SD-VBS we selected four benchmarks: disparity, mser, sift, and tracking. For each, three image resolutions—fullhd, vga, and cif—are tested under three different conditions: *(i)* PL not included in the coherence domain (*"No PL"*), *(ii)* rMMU included in the coherence domain yet disabled (*"PL Coherent"*) and *(iii)* the prototype tracing the `text` section of the benchmark of interest (*"PL Coherent + Tracing Enabled"*). Fig. 2 confirms that the runtime cost of the machinery is negligible. The majority of the overhead occurs at startup time, in which FRACTAL has to issue up to an invalidation message for each page in the VA's range and for each page table entry in the translation walk.

B. Instruction Tracing

To showcase the instruction tracing potential of FRACTAL, within a minimal overhead setup, we opted for an image filter benchmark, `canny`, and instrumented the machinery to trace its text section. No observation points are instrumented, therefore, the resulting trace is only driven by the PE's cache misses. The benchmark was run with `-t 5` and with a fullhd input image.

The resulting trace is shown in Fig. 3: on the right the functions to which the traced instructions belong and their distribution over time. It is possible to see the limits of each of the 5 runs and identify the order in which the functions are called.

VI. Conclusion

In this work, we presented FRACTAL, a hardware module operating in a cache-coherent FPGA that provides low-overhead memory and instruction tracing for a focused range of addresses of a target user application. In this first prototype, FRACTAL has already proven to be a valuable, low-overhead tool for trace acquisition, remaining competitive with state-of-the-art alternatives. We showed that the main strengths of rMMU are its *(i)* runtime adaptability and its *(ii)* ability to function with minimal software intervention while having *(iii)* transparent hardware-level tracing. We envision a number of possible refinements and improvements in terms of FPGA utilization, user library functionality, and overall execution overhead.

Acknowledgment

This research was supported by the National Science Foundation (NSF) under grants number CSR-2238476. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

References

[1] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. Boston, MA: USENIX Association, Jun. 2004. [Online]. Available: https://www.usenix.org/conference/2004-usenix-annual-technical-conference/dynamic-instrumentation-production-systems

[2] Linux Kernel Development Community, "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/, 2009, available since Linux kernel 2.6.31.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[4] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, p. 89–100, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[5] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, 2024, order Number: 253669.

[6] G. Bitzes and A. Nowak, "The overhead of profiling using PMU hardware counters," CERN openlab, CERN openlab Report, Jul. 2014. [Online]. Available: https://openlab-archive-iv-v.web.cern.ch/openlab-archive-iv-v/sites/test-static-05.web.cern.ch/files/technical_documents/TheOverheadOfProfilingUsingPMUhardwareCounters.pdf

[7] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and Efficient CFI Enforcement with Intel Processor Trace," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 529–540.

[8] S. M. Ali Zeinolabedin, J. Partzsch, and C. Mayr, "Analyzing ARM CoreSight ETMv4.x Data Trace Stream with a Real-time Hardware Accelerator," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Feb 2021, pp. 1606–1609.

[9] W. Chen, I. Izhbirdeev, D. Hoornaert, S. Roozkhosh, P. Carpanedo, S. Sharma, and R. Mancuso, "Low-Overhead Online Assessment of Timely Progress as a System Commodity," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. V. Papadopoulos, Ed., vol. 262. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 13:1–13:26. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2023.13

[10] I. Ashraf, M. Taouil, and K. Bertels, "Memory profiling for intra-application data-communication quantification: A survey," in *2015 10th International Design & Test Symposium (IDT)*, 2015, pp. 32–37.

[11] Intel Corporation. (2024) Timed process event based sampling (tpebs). Technical Article. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/timed-process-event-based-sampling-tpebs.html

[12] AMD. (2024) Introduction to ibs (instruction based sampling). Documentation. [Online]. Available: https://docs.amd.com/r/en-US/68658-uProf-getting-started-guide/Introduction-to-IBS-Instruction-Based-Sampling

[13] S. Miksits, R. Shi, M. Gokhale, J. Wahlgren, G. Schieffer, and I. Peng, "Multi-level Memory-Centric Profiling on ARM Processors with ARM SPE," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2024, pp. 996–1005.

[14] M. A. Sasongko, M. Chabbi, P. H. J. Kelly, and D. Unat, "Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1594–1608, May 2023.

[15] Linux Kernel Documentation. (2020) Coresight - arm hardware trace. Linux kernel v5.7 documentation. [Online]. Available: https://www.kernel.org/doc/html/v5.7/trace/coresight/coresight.html

[16] ARM. (2024) Coresight soc-400. Product page. [Online]. Available: https://developer.arm.com/Processors/CoreSight%20SoC-400

[17] Lauterbach GmbH. (2024) Application note for xilinx zynq. Application Note. [Online]. Available: https://www2.lauterbach.com/pdf/app_xilinx_zynq.pdf

[18] Linaro, "Opencsd: Open coresight decoder library," https://github.com/Linaro/OpenCSD, 2024, accessed: 2026-02-08.

[19] I. Parnassos, P. Skrimponis, G. Zindros, and N. Bellas, "SoCLog: A real-time, automatically generated logging and profiling mechanism for FPGA-based Systems On Chip," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

[20] E. Skordalakis, A. Attwood, J. Goodacre, and M. Luján, "Accprof: Increasing the accuracy of embedded application profiling using fpgas," in *Architecture of Computing Systems*, D. Fey, B. Stabernack, S. Lankes, M. Pacher, and T. Pionteck, Eds. Cham: Springer Nature Switzerland, 2024, pp. 192–206.

[21] S. Roozkhosh, D. Hoornaert, and R. Mancuso, "CAESAR: Coherence-Aided Elective and Seamless Alternative Routing via on-chip FPGA," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 356–369.

[22] F. Ciraolo, M. Nicolella, D. Hoornaert, M. Caccamo, and R. Mancuso, "Light virtualization: a proof-of-concept for hardware-based virtualization," 2025. [Online]. Available: https://arxiv.org/abs/2502.15738

[23] AMD, *Kria K26 SoM Data Sheet*, Advanced Micro Devices, Inc., 2024, product specification for Kria K26 System-on-Module. [Online]. Available: https://docs.amd.com/r/en-US/ds987-k26-som

[24] ARM, "ARM® CoreLink™ CCI-400 Cache Coherent Interconnect," Tech. Rep., 2015. [Online]. Available: https://developer.arm.com/documentation/ddi0470/k/functional-description/snoop-connectivity-and-control

[25] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "RT-Bench: an Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 184–195. [Online]. Available: https://doi.org/10.1145/3534879.3534888

[26] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego Vision Benchmark Suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 55–64.

[27] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.