Enabling Heterogeneous Performance Analysis for Scientific Workloads

Maksymilian Graczyk*, Vincent Desbiolles‡, Stefan Roiser*, and Andrea Guerrieri‡
*CERN, Geneva, Switzerland
‡School of Engineering, HES-SO Valais-Wallis, Sion, Switzerland

***Abstract*—Heterogeneous computing integrates diverse processing elements, such as CPUs, GPUs, and FPGAs, within a single system, aiming to leverage the strengths of each architecture to optimize performance and energy consumption. In this context, efficient performance analysis plays a critical role in determining the most suitable platform for dispatching tasks, ensuring that workloads are allocated to the processing units where they can execute most effectively. *Adaptyst* [1] is a novel ongoing effort at CERN, with the aim to develop an open-source, architecture-agnostic performance analysis for scientific workloads. This study explores the performance and implementation complexity of two built-in eBPF-based methods such as Uprobes and USDT, with the aim of outlining a roadmap for future integration into *Adaptyst* and advancing toward heterogeneous performance analysis capabilities.**

I. Introduction

In recent years, heterogeneous platforms enable optimized execution of workloads according to their computational characteristics. However, the inherent complexity of heterogeneous systems poses significant challenges for performance optimization. Efficient utilization of resources requires a deep understanding of workload behavior across different components. This necessitates robust profiling methods capable of capturing fine-grained performance metrics, identifying bottlenecks, and guiding optimization strategies. Profiling techniques, ranging from hardware counters and sampling-based tools to trace-based analysis, play a critical role in bridging the gap between system potential and actual performance. *Adaptyst* [1] is a novel, open-source performance analysis and software-hardware co-design tool designed to scale from embedded to high-performance computing and incorporate several static and dynamic analysis methods, including sampling-based profiling. Its modular nature as shown in Fig. 1 and integration with stateful dataflow multigraphs [2] make our work easily adaptable to already-existing and new solutions on both the software and hardware sides while considering the big picture of computing systems like memory hierarchies, networking, operating systems, and storage. *Adaptyst* can profile on-CPU and off-CPU activity of single- and multi-threaded codes in any compiled language running on Linux. This is made possible through an internal module named *linuxperf* (based on a customized version of *perf*) capable of inspecting CPU performance of arbitrarily-sized code blocks of a program using the cache-aware roofline modelling method. *Adaptyst* has been successfully tested on x86-64, arm64, and RISC-V instruction set architectures.

II. Related Work and Contribution

The profiling ecosystem includes several state-of-the-art tools [3], [4], [5], [6], [7], [8], but few offer extensible support

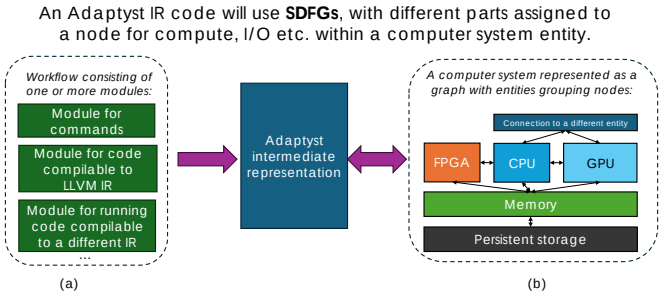

Fig. 1. The modular design of *Adaptyst* as a software-hardware co-design framework. Stateful dataflow multigraphs [2] are used as the IR. (a) Every module describes how a corresponding *Adaptyst* IR should be generated for its use case. (b) Every node is assigned to a backend module which describes how a specific system component should be modelled/profiled, how it can connect to other nodes, and (later) how a match between the component and the *Adaptyst* IR region should be calculated.

for heterogeneous or non-standard architectures. Additionally, not all are open-source or vendor-portable, limiting their broader applicability. In software-hardware co-design, there are several works targeting different parts of the software-hardware spectrum. This list include standard, modular or transparent compilers [9], [2], domain-specific languages or vendor-specific programming models [10], [11], [12], [13], or high-level synthesis tools.

Adaptyst is designed to interact with existing tools rather than compete with them: our work aims to bridge the gaps between these tools by ensuring language-agnosticism, automatic hardware and code selection, and considering the broader picture beyond solely compute units. However, short-term *Adaptyst*'s development faces some challenges: (1) in the case of *linuxperf*, the necessity of calling a separate instance of "perf" for profiling CPU activity of codes instead of built-in kernel methods like *perf_event_open* or eBPF, (2) the requirement of compiling programs with frame pointers for complete stack traces.

In this work, we focus on addressing the first limitation of *Adaptyst* by first investigating the preliminary performance and implementation complexity of two eBPF-based built-in alternatives: Uprobes and USDT (User-Static Defined Tracing).

Adaptyst uses three types of profiling: sampling-based CPU walltime, tracing of context switches for off-CPU activity analysis, and tracing of spawning and exiting threads/processes of a given program. The aforementioned eBPF alternatives (Uprobes and USDT) are related to the latter type, and they may also be expanded later when looking into removing the frame pointer compilation requirement.

III. Preliminary Results

For our eBPF investigation, we want to evaluate two primary aspects: (1) performance overhead and stability, (2) implementation complexity. The first aspect, performance overhead, involves measuring and comparing the runtime overhead introduced by each profiling tool during program execution. This allows for a detailed understanding of the efficiency of each tool and its suitability for real-time or high-performance applications. The second aspect, implementation complexity, focuses on assessing the practical challenges associated with deploying each tool.

To evaluate the profiling methods, a lightweight yet sufficiently complex C program that computes approximate square roots of integers from 1 to 100 has been used. This allows a fast-executing workload for serial benchmarking (multiple runs to average measures), to reveal differences between profiling methods. Compilation was performed with GCC version 13.3.1 using the flags `-fno-omit-frame-pointer` and `-mno-omit-leaf-frame-pointer`, ensuring consistent stack traces. The experiments were conducted on an Intel Xeon Silver 4216 with 32 cores, 180 GB RAM, running Gentoo Linux. To minimize scheduling noise, each program was pinned to a dedicated CPU core.

A. Performance Overhead and Stability

Profiling results (presented in Table I) show minimal overhead across all configurations, respectively 5.1% for USDT and 4.8% for Uprobes. The standard deviation is modest across all cases, indicating stable performance. While the overall distribution is consistent, the USDT configuration shows slightly higher variance and maximum execution time, suggesting a marginal increase in profiling overhead. Figure 2 shows the breakdown of system and user time for the same configurations. It highlights that Uprobes incurs more system time than USDT or baseline, likely due to kernel-level instrumentation.

TABLE I
TIMING RESULTS FOR DIFFERENT PROFILING METHODS. BENCHMARKING WAS CONDUCTED USING *hyperfine* [14], WITH 100 WARM-UP RUNS AND 1000 MEASUREMENT RUNS. RESULTS ARE EXPRESSED IN MILLISECONDS.

Type	Mean (ms)	Stddev (ms)	Median (ms)	Min (ms)	Max (ms)
Baseline	1.026	0.199	0.963	0.827	1.949
USDT	1.079	0.211	1.004	0.839	2.083
Uprobes	1.076	0.207	1.000	0.833	2.054

B. Implementation Complexity

In terms of deployment complexity, eBPF can run concurrently with existing applications without code changes. However, developing eBPF-based solutions involves significant complexity due to intricate data structures and a multi-stage compilation process.

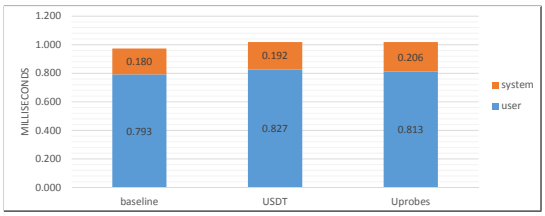

Fig. 2. System versus User Breakdown over 100 warm-up runs and 1000 measurement runs. Uprobes uses more system time than USDT.

IV. Conclusion and Future Work

This study presented a preliminary analysis of two eBPF-based profiling tools focusing on operational overhead and deployment complexity. The results show that both USDT and Uprobes introduce minimal overhead compared to the baseline. While all methods demonstrate consistent performance, Uprobes contributes slightly more system time, and USDT exhibits marginally higher variability. Future work includes the integration of e-BPF into *Adaptyst*, as well as the support for non-CPU devices such as GPUs and FPGAs.

Acknowledgements

This work is funded by European Union's Horizon Europe Programme grant 101092877 (SYCLOPS), 101129744 (EVERSE - HORIZON-INFRA-2023-EOSC-01-02), and HES-SO.

References

[1] "Adaptyst," https://adaptyst.web.cern.ch/.
[2] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *SC'19*, 2019.
[3] "perf: Linux profiling with performance counters," https://perfwiki.github.io/main, access: 2025-02-23.
[4] Intel, "Fix performance bottlenecks with intel vtune profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, access: 2025-02-23.
[5] AMD, "Amd μprof," https://www.amd.com/en/developer/uprof.html, access: 2025-02-23.
[6] "Gnu gprof," https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html, access: 2025-02-23.
[7] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.
[8] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
[9] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *Proceedings of CGO*. IEEE, 2021, pp. 2–14.
[10] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *DAC*, 2012, pp. 271–276.
[11] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak *et al.*, "A matlab compiler for distributed, heterogeneous, reconfigurable computing systems," in *Proceedings of FCCM*. IEEE, 2000, pp. 39–48.
[12] T. Grosser and T. Hoefler, "Polly-acc transparent compilation to heterogeneous hardware," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–13.
[13] H. Riebler, G. Vaz, T. Kenter, and C. Plessl, "Transparent acceleration for heterogeneous platforms with compilation to opencl," *ACM TACO*, vol. 16, no. 2, pp. 1–26, 2019.
[14] "Hyperfine," https://github.com/sharkdp/hyperfine.