Software-Shaped Platforms

Renato Mancuso
rmancuso@bu.edu
Boston University
USA

Shahin Roozkhosh
shahin@bu.edu
Boston University
USA

Denis Hoornaert
denis.hoornaert@tum.de
Technical University of Munich
Germany

Ju Hyoung Mun
jmun@bu.edu
Boston University
USA

Tarikul Islam Papon
papon@bu.edu
Boston University
USA

Manos Athanassoulis
mathan@bu.edu
Boston University
USA

ABSTRACT

This paper outlines the vision for a new type of software-shaped platforms, or *SOSH platforms* for short, that can be implemented in commercial CPU+FPGA platforms. At the core of the SOSH paradigm is the idea of exposing direct control over the flow of data exchanged between hardware components in embedded System-on-Chips (SoC). Data flow manipulation primitives are synthesized in reprogrammable hardware and interposed between central processors, memory modules, and I/O devices. A new layer of system software is then introduced to leverage such primitives and to achieve fine-grained control and introspection over the interaction of SoC resources. By turning memory and I/O data flows into manageable entities, a new degree of internal awareness can be achieved in complex systems. We first review recent works that are well aligned with the concept of data flow manipulation primitives that can be deployed in SOSH platforms. Next, we outline future research avenues concerning the use of the SOSH paradigm for workload profiling and prediction, to implement advanced memory models, and to perform security threat identification and mitigation.

KEYWORDS

software-shaped platforms, predictability, architectures

ACM Reference Format:
Renato Mancuso, Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, and Manos Athanassoulis. 2023. Software-Shaped Platforms. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 9–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3576914.3587546

1 INTRODUCTION

Computing systems with strict reliability and confidentiality requirements support the modern world and society. Avionic systems, nuclear plant supervisory systems, orbit controllers in satellites, driverless cars, and unmanned aerial vehicles (UAVs) are a few notable examples, often referred to as safety-critical systems. But safety alone is not enough. Modern safety-critical systems are expected to build and exploit knowledge of their environment and make complex decisions based on a multitude of sensory inputs. In other words, they must meet ever-evolving expectations about their ability to intelligently interact with their environmental context. We refer to one such ability as *environmental awareness*.

At the same time, a safety-critical system must continue to function despite unexpected overloads or failures. To do so, predicting and assessing overload conditions, violation of timing requirements, and presence of security threats is equally important. In many ways, we demand that our systems become aware of their internals to tell apart what is *expected* from what is *aberrant*. This form of *internal awareness* is in stark contrast with the aforementioned environmental awareness.

Indeed, achieving environmental awareness calls for an increase in system complexity. The fusion of high-bandwidth sensing devices, the addition of specialized hardware accelerators, and the use of machine-learning data-intensive analytics are now commonplace. The result is a highly complex, often under-optimized software stack on top of equally sophisticated hardware. As system complexity soars, it is impossible to fully understand and predict the exact interplay between software components, software and hardware, or even between individual hardware modules. This fundamental lack of understanding severely limits the degree of internal awareness that can be achieved in traditional computing systems.

Data Flows as Assets: In complex systems, having static knowledge of all the emerging interactions between hardware and software components is hard, if not impossible. Thus, towards achieving internal awareness, it is necessary to have direct control over all interactions in the system. Such interactions occur as *flows of data* exchanged between components.

The need to enact precise monitoring and control over on-chip data flows is not a novel idea. Radical system redesigns like CoQoS [7] and PARD [9] pioneered this principle more than a decade ago. What is novel is the intuition (corroborated with a substantial array of practical use cases) that emerging trends in augmenting existing SoCs with an on-chip programmable logic (FPGA) empower seamless and continuous observability and control over on-chip data flows. This implies that (1) advanced flow analysis and control primitives can be carried out in existing commercial hardware and (2) that the original vision of QoS-oriented data flow management can be significantly broadened.

CPS-IoT Week Workshops '23, May 9–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0049-1/23/05…$15.00
https://doi.org/10.1145/3576914.3587546

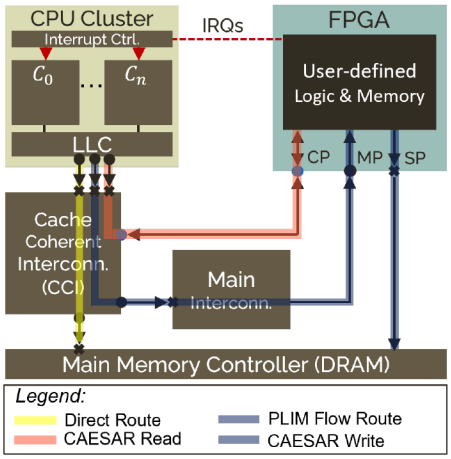

Figure 1: CPU+FPGA SoC overview. Direct route to main memory (yellow) and through-FPGA alternative route possible via PLIM [17] (blue).

Software-Shaped Platforms: With programmable control over on-chip data flows, we argue that a new paradigm for software-shaped platforms, or **SOSH** for short, is possible. In the SOSH paradigm, the software instantiates, as needed, hardware modules to constantly monitor data-flows, define policies and performance envelopes for data-flow exchanges, and define actions that affect both hardware and software components in case of policy violation. SOSH platforms are designed to be implemented using commercially available hardware and thus exhibit competitive performance from their inception. The enabling technology we leverage is the co-location of embedded processing elements and dynamically reconfigurable FPGA in modern SoCs. In SOSH platforms, the FPGA is not primarily used to build accelerators. Instead, FPGA modules are defined to monitor, profile, predict, and control the flow of data exchanged between on-chip resources — i.e., between processors, I/O peripherals, memory modules, and accelerators.

In this paper, we outline (1) a broad vision for SOSH platforms and review (2) key enabling mechanisms and (3) concrete use cases for data flow observation and management that constitute the pillars of the SOSH paradigm.

2 BACKGROUND: CPU+FPGA SYSTEMS

CPU+FPGA Platforms. CPU+FPGA SoCs are key enablers of the SOSH vision. A simplified block diagram is provided in Fig. 1. In these SoCs, a multi-core (C_0,...,C_n) computing cluster (CPU) and programmable logic (FPGA) are co-located. Today, the two major players in general purpose computing, i.e., Intel® and AMD®, own the two leading companies in FPGA technology, i.e., Altera® (June 2015) and Xilinx® (February 2022), respectively. **CPU+FPGA SoCs are poised to become a standard computing model** in both the embedded market and enterprise computing segments. Typically, the CPUs within the cluster share a single LLC. The downstream memory hierarchy components must resolve LLC misses (cache refills) following a write-back write-allocate (WBWA) policy. By WBWA, dirty lines are written back upon eviction, and a write miss allocates a new line in the LLC. If a cache refill is for a line present in main memory, the yellow route in Fig. 1 is followed by memory transactions. Memory transactions pass through one or more point-to-point bus segments. The initiating end is called a *main* port (MP, ● in Fig. 1), the replying end is called *secondary* port (SP, ✖ in Fig. 1). **Memory-mapped FPGA:** the FPGA is assigned a static aperture within the physical addressing space. LLC refills (write-backs) whose physical address falls within the FPGA aperture initiate a read (write) memory transaction towards the FPGA. This approach to initiate CPU-to-FPGA communication is termed *memory-mapped semantics*. **CPU-to-FPGA interfaces:** when memory transactions reach the FPGA, they arrive through an MP (from the CPU cluster's perspective). The FPGA can initiate transactions towards main memory (DRAM) via an SP of comparable performance. Additionally, an emerging technology [2, 3] is that of connecting the CPU cluster with the FPGA directly via a bi-directional coherence port (CP, ● in Fig. 1).

3 ENABLING MECHANISMS

The ability to dynamically instantiate hardware components to manage on-chip data flows is enabled by three complementary techniques, namely: (1) Partial Dynamic Reconfiguration (PDR), (2) Programmable Logic In-the-Middle (PLIM), and (3) Coherence-Aided Elective and Seamless Alternative Routing (CAESAR). We briefly discuss these key mechanisms in the following.

Partial Dynamic Reconfiguration (PDR): Modern FPGAs that support PDR allow the definition of hardware components at runtime and thus without a system reset. Modular reconfiguration of the FPGA is supported, meaning that some of the hardware modules defined on the FPGA can be reprogrammed while others continue to operate. If a library of modules is synthesized ahead of time, the overhead paid for their deployment can be kept at a minimum. An excellent array of studies on the reconfiguration bandwidth of modern FPGAs were conducted in [1, 12, 13, 19]. As highlighted in [13], modern FPGAs can support reconfiguration throughputs upwards of 0.9 Gbps, meaning that sizable ~1 MB modules can be swapped in/out with millisecond-scale overheads.

Programmable Logic In-the-Middle (PLIM): The work in [17] pioneered the idea of interposing programmable logic between processor-side memory fetches and main memory with the PLIM technique. Under PLIM, memory transactions originating at the CPUs and targeting main memory are *re-routed* through the FPGA, following an *alternative* route denoted in blue in Fig. 1. Doing so incurs acceptable overheads [17] and opens up new avenues to design and deploy resource management primitives previously available only through hardware re-design. It also creates new opportunities in defining OS-level and hypervisor-level technologies that leverage PLIM modules to improve performance management, self-awareness, and online workload characterization.

Coherence-Aided Elective and Seamless Alternative Routing (CAESAR): A promising technique with cache-coherent on-chip FPGAs via the CP is *coherence backstabbing* which was recently introduced in [15]. Coherence backstabbing refers to the possibility of leveraging the coherence port exposed to the FPGA for purposes other than maintaining cache coherence. The term *backstabbing* wants to evoke that the FPGA inserts itself on the back of a protocol where (1) useful information about the behavior of the CPUs is transferred (**observability**); (2) the state of the upstream caches,

and the semantics and timing of downstream data accesses can be impacted by programmable logic (**programmability**). Coherence backstabbing enables the CAESAR approach for memory traffic management, which extends PLIM capabilities in three main ways. First, CAESAR enables memory traffic to follow a route that is an alternative to what is followed with traditional memory-mapped semantics used by the original PLIM approach. Because such a route bypasses slower SoC interconnects meant for I/O traffic, it also enables lower through-FPGA re-routing overheads. The results outlined in [15] suggest that using the coherence port unlocks up to 40% better throughput for read traffic compared to using memory-mapped semantics. Second, it allows re-routing data and metadata (e.g. access timestamps and physical addresses) with different trade-offs between seamlessness and resulting overheads. Third, it enables dynamic activation of the alternative route (for cache-line refill traffic) without the need to modify page-table translations.

4 SOFTWARE-SHAPED PLATFORMS

At the core of the SOSH paradigm is the idea of exposing direct control over the flow of data exchanged between hardware components in an embedded System-on-Chip (SoC). Data flow manipulation primitives are synthesized in reprogrammable hardware and interposed between central processors, memory modules, and I/O devices. The SOSH paradigm leverages PDR capabilities to define management and monitoring primitives as needed. For instance, a SOSH platform can instantiate on-the-fly hardware modules to perform accurate application profiling; gather online statistics about the usage of hardware resources; enforce I/O data pre-processing and filtering rules; re-routing application traffic.

On the other hand, the dynamically-instantiated modules leverage PLIM and CAESAR techniques to enact programmable management policies for on-chip data flows. The deployed PLIM/CAESAR components allow exporting on-demand and fine-grained profiling and control mechanisms back to the software layers. Thus, a new layer of system software is then introduced to leverage such primitives; and to achieve fine-grained control and introspection over the interaction of SoC resources. Further run-time PDR policies can get triggered based on the nourished real-time profile. By turning memory and I/O data flows into manageable entities, a new degree of self-awareness can be achieved in complex systems.

As suggested by the name, a SOSH platform can be re-shaped as needed by the software in light of the current operating conditions. And while doing so, the software can retain control and observability over the most performance-critical inner-most hardware-application interactions.

Data Flow Management Templates: Fig. 2 provides an overview of the key super-classes of data-flow management primitives—depicted as black boxes—that SOSH platforms can instantiate. **Merging** primitives allow enacting complex/programmable rules to join data flows originated by different modules. An example is provided in Section 5.1. **Reordering/Filtering** primitives will carry stream-processing operations to (1) reduce the amount of moved data and (2) optionally perform re-ordering to improve locality. A concrete example of one such block is presented in Section 5.2. Next, **Profiling/Logging** primitives support extracting data flow characteristics for prediction and state/progress tracking of application workload, as briefly described in Section 5.3. Lastly, **Splitting** primitives will

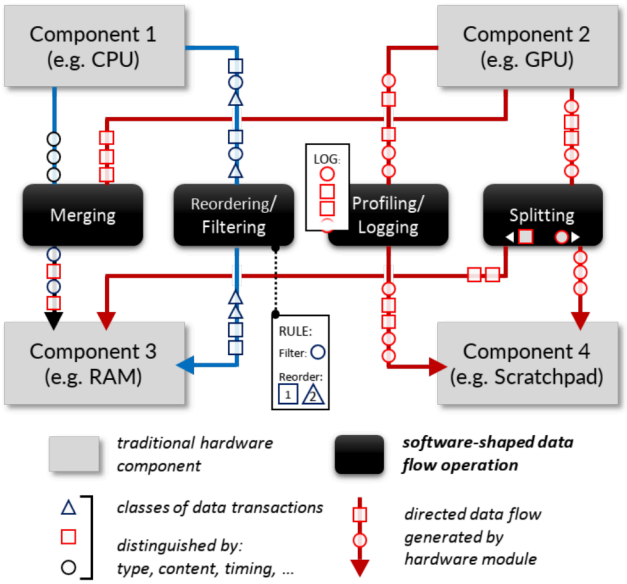

Figure 2: High-level semantics of key data-flow manipulation primitives (black boxes) in software-shaped platforms.

allow selectively re-routing sub-flows to improve timing and/or relieve congestion.

5 EXISTING SOSH MODULES

This section briefly reviews concrete designs and implementations of data flow operation blocks that have been studied in the past. These represent a bootstrap to the library of modules that a SOSH platform can instantiate.

5.1 Scheduler in-the-Middle (SchIM)

The Scheduler in-the-Middle, or SchIM for short, was proposed in [6]. The SchIM is a configurable module that leverages PLIM to interpose itself between the last-level cache (LLC) of the CPU cluster and the memory controller in a CPU+FPGA system. By recognizing the regions of physical memory mapped to the different software partitions, the SchIM can classify the traffic source on a per-core and potentially per-application basis. By leveraging this information, the SchIM performs memory transaction-level scheduling following policies that can be customized and configured at runtime by the software. The original work showcased two proof-of-concept memory scheduling policies: Time Division Multiple Access (TDMA) and Fixed Priority (FP), while Traffic Shaping (TS) has also been incorporated more recently.

The SchIM is a good example of a SOSH block providing data flow merging (see Fig. 2) operations. So long as the source of the traffic can be identified, the SchIM allows the software layer to define the logic according to which memory transactions originated by different upstream components—in this case, different CPUs—are forwarded to a downstream component—in this case, main memory.

Fig. 3 was obtained with a similar setup as that in [6], and it demonstrates the fine-grained level of control that the SchIM introduces when deployed and configured to policy the traffic originating

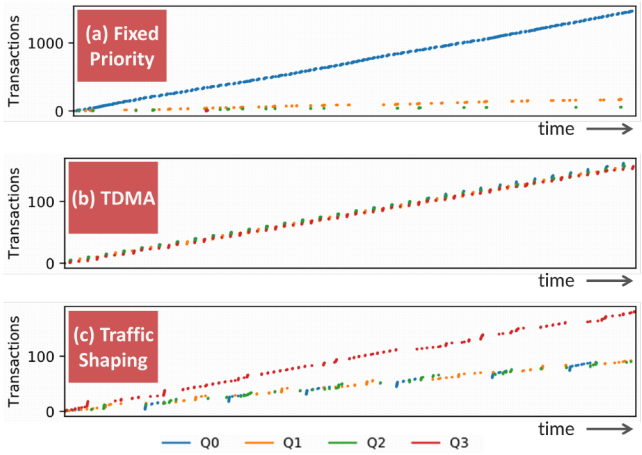

Figure 3: Trace of memory transactions collected on-chip via a real-hardware instantiation of the SchIM module [6].

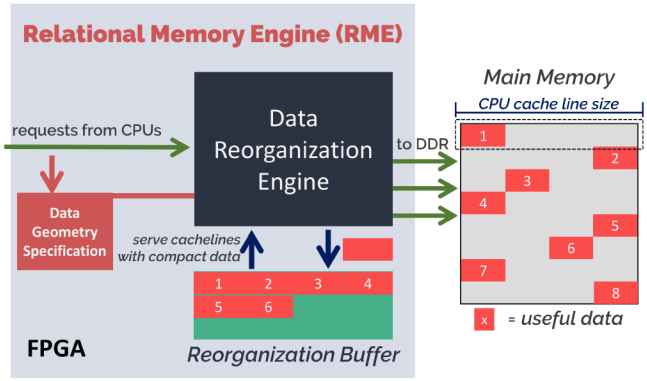

Figure 4: High-level organization of the RME. When the CPU makes a request for a cache line of reorganized data, main memory is accessed. Useful data items are extracted on the fly based on the provided geometry specification, and cache lines with only useful data are issued in response [16].

from CPUs C_0 through C_3. In Fig. 3(a), FP is used, and the priority of traffic en-queued by C_0 (Q0) is configured to be the highest. Therefore C_0 completes substantially more requests per unit of time compared to the other cores. In Fig. 3(b), equally sized transmission slots are configured for all the cores, resulting in similar traffic patterns across all the cores, with a recognizable slotted transmission pattern. Finally, in Fig. 3(c), TS is used to assign to C_3 (Q3) twice the bandwidth allocated to the other cores. This results in the visible separation between the rate of transmissions from Q3 and that from the other queues in Fig. 3.

5.2 Relational Memory Engine (RME)

Roozkhosh et al. [16] propose the concept and implementation of the Relational Memory Engine (RME). The RME represents an excellent instance of a data flow reordering/filtering operational block (see Fig. 2). Indeed, the RME is a configurable near-data transformation engine positioned between main memory and LLC. The engine captures CPU-originated memory requests and converts on-the-fly data from a row-major format to arbitrary column groups by leveraging the PLIM paradigm [17]. A high-level visual representation of the engine is provided in Fig. 4.

Relational data are organized in relational tables with tuples stored as rows containing several columns (also called attributes). The data is physically laid out either in a row-first format or column-first format. Maintaining each row together is beneficial for workloads that target reading or updating individual rows—i.e, *transactional workloads*—while storing data in column-first fashion benefits more workloads that calculate aggregates and statistics of large chunks of data—i.e., *analytical workloads*. Traditional data management systems have been aiming to support one of the two workloads, so applications that have both workloads need multiple systems and pay the cost of gradually converting data from a transactional-friendly layout to an analytics one [10, 11, 14].

The core idea of RME is to create reorganized aliases of the data (*ephemeral variables*) that are not physically stored in main memory, yet that the CPU can use as if they were. In other words, the RME can deliver to the CPU relational data with arbitrary data layouts that are accessible with the ideal spatial locality. Thanks to the RME, a system is able to store data in a row-major format in memory and efficiently absorb incoming data. At the same time, it can deliver arbitrary groups of columns for read-only analytical queries with no overhead. Because unnecessary data items are excluded from cache lines, a net reduction in cache footprint is also achieved. But more broadly, the RME showcases the transformational capabilities of redefining the semantics of memory accesses in a configurable way. Indeed, we are able to decouple data addresses from their placement and organization in main memory.

5.3 Silent Application Profiler

Coherence backstabbing and the CAESAR approach [15] (see Section 2) open the door to the design and implementation of non-intrusive profiling/logging data flow operation blocks. Perhaps most prominently, the concept of a silent application profiler, called the *Silent Observer*, was proposed in [15]. The profiler is capable of collecting important and fine-grained metadata about the main memory traffic produced by the CPUs. In particular, by monitoring the CP (see Fig. 1), the module collects the physical addresses of all the cache lines that were accessed and resulted in a cache miss in the LLC. In the same way, the exact timestamp of those transactions is acquired by the module.

The high-level operation of a CAESAR-aided memory profiler is depicted in Fig. 5. Upon (1) a cache miss encountered by an application core in LLC, the coherence fabric generates a (2) snoop transaction towards the FPGA. The timestamp at which the snoop is received corresponds to the timestamp of the cache miss. The payload of the snoop transaction carries the physical address. The module can (3) immediately signal the coherence fabric that it does not hold a copy (nor an updated version) of the requested cache line. Contemporarily, it (4) logs said transaction metadata for later analysis in a dedicated and off-the-critical-path memory block—e.g., an FPGA-side scratchpad.

Because the profiler module does not actively interact with the coherence port, profiling is performed off-band and introduced near-zero overhead, as demonstrated in [15]. Fig. 6 provides a visualization of the kind of insights that can be gathered via this

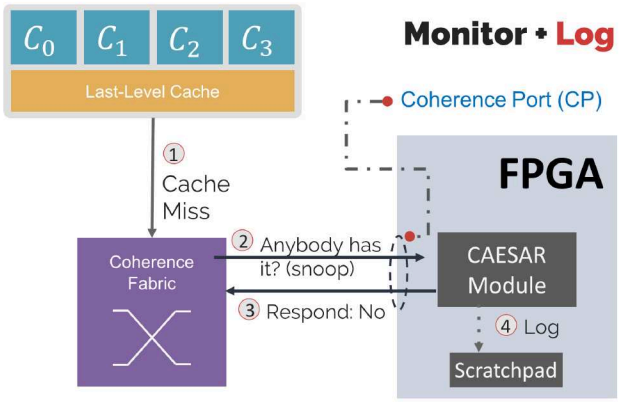

Figure 5: High-level operational principle of a CAESAR-aided memory profiler. The module passively listens to coherent interconnect-generated snoop transactions, which carry the timestamp and physical address of requested cache lines. These metadata are logged on a dedicated scratchpad [15].

profiling technique. The two insets depict a partial visualization of the memory traffic generated by two applications. These are (a) `tracking` and (b) `mser` from the San Diego Vision Benchmarks Suite (SD-VBS) [18]. In particular, the top plot in each inset shows the cumulative number of memory transactions (y-axis) over time (x-axis). The bottom plot in each inset focuses on those 100 pages recorded to have the largest number of accesses throughout the profiled runs. For each of them (y-axis), each cell in the heatmap is colored on a blue-green gradient scale to encode the number of accesses to cache lines within that page over a discretized notion of time (x-axis). The qualitative difference between insets (a) and (b) is representative of the different characteristic memory access patterns of the two applications.

Importantly, instead of being logged for later analysis, the information extracted by the profiling module can be stream-processed to inform the software layers of a SOSH platform about application progress, unexpected access pattern behaviors, and/or to detect the usage of and enforce memory bandwidth reservation quotas.

6 ENVISIONED SOSH CAPABILITIES

The concrete instances of data flow operation blocks reviewed in Section 5.1–5.3 highlight the feasibility of the SOSH paradigm. More research is required to complement the SOSH vision with key additional capabilities. A non-exhaustive list of such capabilities that we envision to be within reach is reviewed in the following.

6.1 Progress Tracking and Behavior Prediction

Progress Tracking: When an application with a known profile is launched on a SOSH platform, the corresponding profile can be used to monitor its progress and anticipate its needs. A progress tracking module that leverages data-flow monitoring can be designed by recognizing the characteristic *fingerprint* [4] of different execution phases in the application under analysis. Tracking application progress can provide unprecedented internal awareness in a SOSH platform. First, clashes over system resources between two or more co-running applications can be easily predicted and

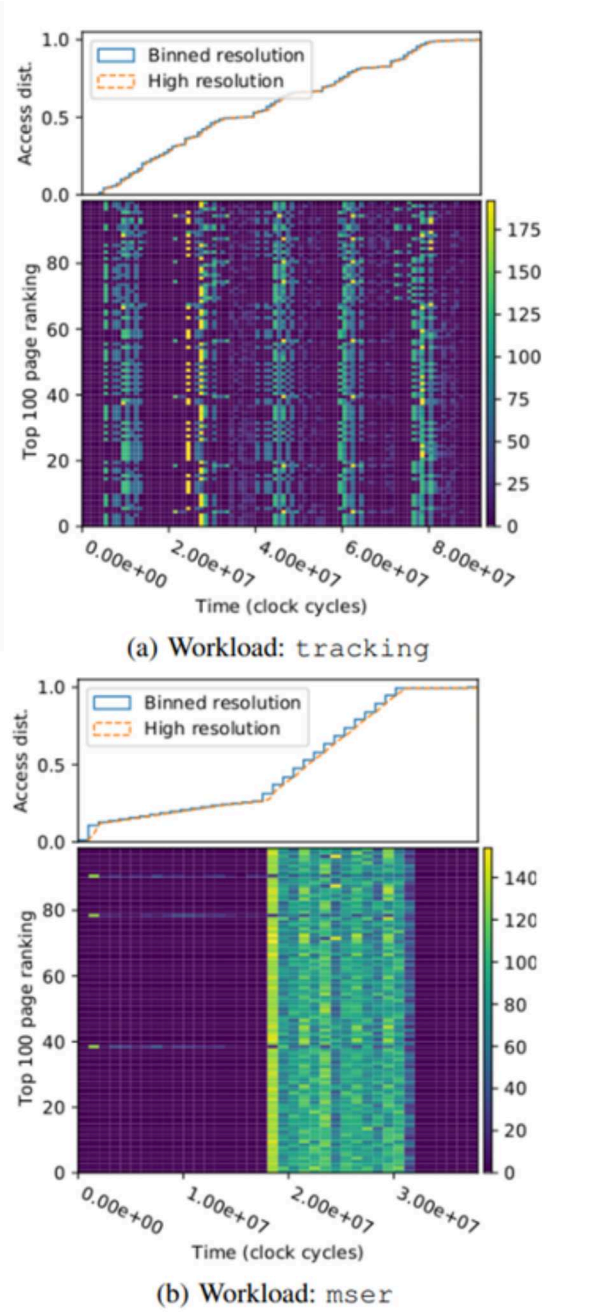

Figure 6: Partial visualization of memory traces obtained via the CAESAR-aided profiling module by observing the memory traffic generated by two SD-VBS [18] applications, namely (a) `tracking` and (b) `mser` [15].

averted. Second, it is possible to discover if an application is running behind or ahead of schedule. Third, application misbehavior due to software bugs or security breaches can be discovered.

Planning Ahead: Most importantly, the ability to track the progress of applications enables tighter cooperation between the platform and the software. Being able to anticipate an application's flow through stages enables strategies to plan ahead. Simply put, the SOSH platform has a way of enacting a number of preparatory actions to optimize the execution of subsequent phases for tracked applications. Indeed, when the two tasks are executed in parallel on different CPUs, they may contend for memory (or I/O) bandwidth allocation. The degree of contention depends on their profiles and their partial progress in time. Given that this information can be tracked in SOSH platforms, the sustainable bandwidth of a targeted resource can be distributed to contending requestors in a way that is aware of each application's temporal constraint (deadline).

6.2 Advanced Memory Models

By adopting the SOSH paradigm it is possible to dynamically redefine the semantics of memory resources.

Historical Memory: Debugging parallel applications is often problematic because, when a thread is paused to inspect memory content, the temporal interaction of the various threads is impacted. By leveraging data-flow manipulation, a more convenient memory semantic can be defined, which we refer to as *historical memory*. In a nutshell, a debugger can deploy a module to monitor a given (set of) memory block(s). Accesses to the monitored blocks are served according to usual read/write semantics. At the same time, a log is created reporting which thread has manipulated any of the monitored memory locations along with the corresponding timestamp of each operation. A similar model can also implement low-overhead application snapshotting and checkpointing.

Self-Destructing Memory: The SOSH paradigm also enables the creation at runtime of memory objects for which an expiration policy is defined. Consider the case where an application is provided with a secret—e.g., an encryption key—to be used only once during execution. After the key has been used, it should be discarded to minimize the chances of its content being leaked. In this case, a SOSH module can be deployed to monitor any access to the in-memory secret. The module can then allow a single access to the content of the secret, after which the content of the secret is erased. Self-destructing memory semantics can be particularly useful to implement One-Time Programs [5], i.e., programs that can be only run once. This basic model can be further generalized. It is possible to create memory content that automatically expires after a given time. Similarly, it is possible to trigger memory self-destruction as an application progresses past those execution phases that require access to the secret.

6.3 Security Threat Mitigation

SOSH platforms can also harden themselves against security vulnerabilities that arise from hardware design flaws.

Memory Privilege Bypass: The discovery and public release of the Meltdown vulnerability [8] have highlighted the complex interplay between performance-enhancing speculative execution and address space isolation. At the core of the Meltdown vulnerability, there is a violation of a fundamental invariant. That is, while a processor is executing an unprivileged application, a data transaction towards a privileged memory area is initiated. In a SOSH platform, it is possible to intervene by appropriately policing data flows and preventing the completion of cache line refills that, if completed, would leak sensitive information.

Detection of Malware Activity: Signature-based malware detection has been extensively used to identify threats. It can also be efficiently implemented and it introduces low runtime overheads. Unfortunately, most of today's malware is able to alter its signature and avoid detection. Behavior-based malware detection can be employed instead to detect malware capable of altering its signature. Traditionally, sandboxing is used to perform behavior-based detection. A capability made possible with the SOSH paradigm is a new, low-overhead approach for behavior-based detection. By leveraging data flow monitoring, it is possible to analyze the behavior of untrusted software components by analyzing their interaction with memory resources and I/O devices.

7 CONCLUSION

Internal awareness—i.e., understanding and control over the internal interplay of system components—is lost as software and hardware layers grow in complexity. The rise in complexity follows the broadened expectations for environmental awareness in modern safety-critical systems. We propose a novel paradigm of software-shaped (SOSH) platforms aiming to restore strong internal awareness without trading off complexity. SOSH platforms are designed to be implementable in modern FPGA+CPU SoCs and leverage on-chip programmable logic to instantiate data flow control and manipulation primitives. We reviewed existing works that have proposed data flow management modules already well-aligned with the SOSH vision. Finally, we discuss a number of future capabilities that are attainable to complement the SOSH vision.

ACKNOWLEDGMENTS

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grants number CCF-2008799 and CNS-2238476. Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

REFERENCES

[1] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. 2016. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *2016 IEEE Real-Time Systems Symposium (RTSS)*. 1–12. https://doi.org/10.1109/RTSS.2016.010

[2] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 109, 6 pages. https://doi.org/10.1145/2897937.2897972

[3] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 434–451. https://doi.org/10.1145/3503222.3507742

[4] Johannes Freitag, Sascha Uhrig, and Theo Ungerer. 2018. Virtual Timing Isolation for Mixed-Criticality Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 106)*,

Sebastian Altmeyer (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:23. https://doi.org/10.4230/LIPIcs.ECRTS.2018.13

[5] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. One-Time Programs. In *Advances in Cryptology – CRYPTO 2008*, David Wagner (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–56.

[6] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. 2021. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, Vol. 196. 2:1–2:22. https://doi.org/10.4230/LIPIcs.ECRTS.2021.2

[7] Bin Li, Li Zhao, Ravi Iyer, Li-Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. 2011. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *J. Parallel and Distrib. Comput.* 71, 5 (2011), 700–713.

[8] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[9] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, et al. 2015. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 131–143.

[10] C. Mohan. 2016. Hybrid Transaction and Analytics Processing (HTAP): State of the Art. In *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*.

[11] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. https://doi.org/10.1145/3035918.3054784

[12] Marco Pagani, Alessio Balsini, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. 2017. A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*. 96–101. https://doi.org/10.1109/SOCC.2017.8226015

[13] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. 2011. Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model. *ACM Trans. Reconfigurable Technol. Syst.* 4, 4, Article 36 (dec 2011), 24 pages. https://doi.org/10.1145/2068716.2068722

[14] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. *https://www.gartner.com/doc/2657815/* (2014). https://www.gartner.com/doc/2657815/

[15] Shahin Roozkhosh, Denis Hoornaert, and Renato Mancuso. 2022. CAESAR: Coherence-Aided Elective and Seamless Alternative Routing via on-chip FPGA. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 356–369. https://doi.org/10.1109/RTSS55097.2022.00038

[16] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Memory: Native In-Memory Accesses on Rows and Columns. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 66–79. https://doi.org/10.48786/edbt.2023.06

[17] Shahin Roozkhosh and Renato Mancuso. 2020. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 296–309. https://doi.org/10.1109/RTAS48715.2020.00006

[18] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 55–64. https://doi.org/10.1109/IISWC.2009.5306794

[19] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Comput. Surv.* 51, 4, Article 72 (jul 2018), 39 pages. https://doi.org/10.1145/3193827