Hardware Support for Statistical Methods Applied to Hardware-Software Debugging

Debanjan Das, Ryan Robucci, Dhananjay Phatak
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County, Baltimore, MD, USA
{ddas1, robucci, phatak}@umbc.edu

***Abstract*—In this paper we demonstrate a novel integration of the Bloom filter in hardware (RISC-V based OpenTitan®) in order to bolster hardware and/or software de-bugging. We implemented and demonstrated the novel concept by realizing it on an FPGA (for rapid proto-typing). The new methodology yields the following additional capabilities: (i) Setting an arbitrary number of break-points at the hardware level at a substantially lower cost. All other known methods substantially increase the number of registers and comparators (in linear proportion) with the number of break-points set. (ii) All the de-bugging can be done on-line and at-speed (especially as compared to software breakpoints) . The cost is false-positives triggered at the hardware target which can be easily ignored by debug hosts.**

***Index Terms*—FPGA-based silicon verification, hardware design verification, FPGA-design, Bloom filter, statistical error detection, on-chip hardware debugger, remote debugging, GDB**

I. Introduction

Debugging is an important task in hardware-software design. Traditionally on-chip debug support within microcontrollers and embedded cores provides a limited number (e.g. one or two) of hardware breakpoints. Software breakpoints are usually used to augment the set of hardware breakpoints, but can severely degrade speed and introduce timing irregularity. Also, they are not appropriate in cases where Execute in Place (XiP) is used and read-write cycles should be limited. Specialized hardware for recording processor activity or interacting with cores for debugging [1] can be used if on chip high-speed buses can be bridged to such external hardware. However these products can be costly and inconvenient to obtain and integrate into existing projects. Furthermore, our aim is to develop a foundational functionality that is extensible to allow users to define complex trigger conditions.

The challenge of verification for hardware-software co-design is that most tools in these fields are designed for either hardware verification or for software verification assuming operation on known-platforms [2]. Compatibility of specifications in the software and hardware domains is difficult to ensure without full system testing or simulation. Tools such as gem5 [3] achieve microarchetecture simulation and allow for software debugging along with other hardware models, but incorporation of a breath of models and tools provided in the domain of electronic design automation (EDA) is desired. Hardware-related verification using detailed models and designs provided as RTL can be performed via (1) **Simulation** or (2) **FPGA-based silicon verification**. Simulation offers fine-grained observability but is slow, limits the number of test vectors, and cannot record all nodes due to storage constraints. FPGA-based silicon verification executes tests at high speed but exposes minimal internal state, making large-scale internal reporting infeasible. This work targets the second method: rapid, in-system FPGA verification [4].

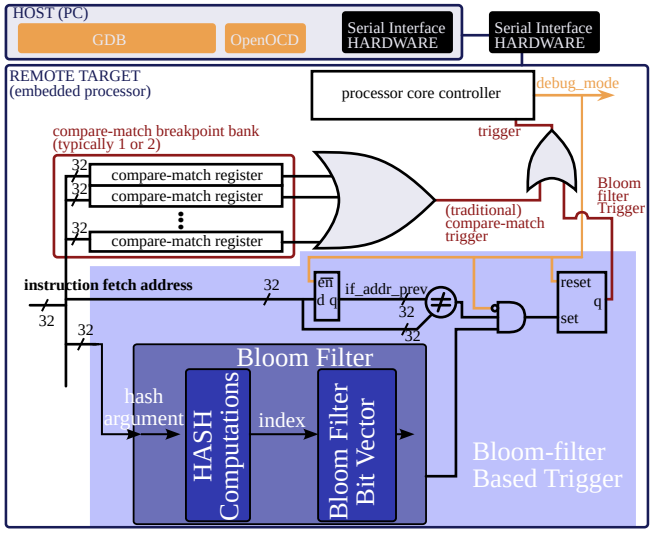

Fig. 1: Bloom Filter-based on-chip debugger support. On-chip hardware usually supports one or two hardware breakpoints and expansion requires a multitude of duplication. The novel approach uses Bloom filter hardware to support a large set of breakpoints updated dynamically via debug tools like GDB.

Herein use of Bloom filters [5] to monitor systems states in remote targets and provide meaningful interaction with debugging tools on hosts is proposed. Work to support a number of desirable monitors in both hardware and software domains ensues, but the focus within this manuscript is the incorporation of the method in existing debug cores for microprocessors. This is valuable for verification of software through execution which serves to validate both software and hardware. The task of detecting systems states and in processors specifically breakpoints is viewed approached as an approximate set membership problem [6]. To this end Bloom

filters are incorporated into triggers for debug systems Fig. 1 alongside traditional compare match registers used to store *hardware breakpoints* . As demonstrated in hardware, both can be updated through use of existing debug tools like the GNU Debugger (GDB) [7]. The limitation of traditional breakpoints is that typically a small number is provided. The new methodology can support extremely large number of breakpoints, and can eventually be designed as an add-on to existing embedded memory to re-purpose any available memory for debugging functionality. With this capability new automated in-system tests could be possible, involving selection of large sets of instructions matching a criteria of interest or supporting novel testing needs associated with automatically generated code.

II. Bloom Filter

Bloom Filters (BFs) [5], [8] are probabilistic data structures that provide fast, space-efficient set-membership testing [9], [10]. Their low memory footprint and one-sided correctness guarantee have led to widespread adoption in both software and hardware. Representative applications include general query engines [11], large-scale database searches (e.g., healthcare, social-security), browser URL screening, crypto wallet validation [12], spell-checking, and high-speed network routers implementing firewall rules and packet filters.

After training on a set $\mathcal{S}$, a BF answers set-membership queries

$$x \stackrel{?}{\in} \mathcal{S}. \tag{1}$$

A BF may report $x \in \mathcal{S}$ even if $x \notin \mathcal{S}$ (false-positive), but it never returns a false-negative:

$$\left.\begin{array}{l} \text{If BF reports } x \in \mathcal{S}, \text{ result may be TRUE or FALSE;} \\ \text{If BF reports } x \notin \mathcal{S}, \text{ result is always TRUE.} \end{array}\right\} \tag{2}$$

Let $N = |\mathcal{S}|$ be the number of stored elements, M the number of bit-buckets, k the number of hash functions, and ϵ the false-positive probability. Then [9]

$$\epsilon = \left(1 - \left[1 - \frac{1}{M}\right]^{kN}\right)^{k}. \tag{3}$$

Ignoring integrality, the value of k minimizing ϵ is

$$k = \frac{M}{N} \ln 2, \tag{4}$$

implying the optimal bits-per-element:

$$\frac{M}{N} = -\frac{\ln(\epsilon)}{(\ln 2)^2} \approx -2.08 \ln(\epsilon), \tag{5}$$

and an equivalent expression for k:

$$k = -\frac{\ln(\epsilon)}{\ln 2}. \tag{6}$$

III. Implementation

A Bloom Filter was implemented in hardware with parameters M, N, and K defining total storage, number of tracked elements, and hash count, which together set the false-positive rate. SystemVerilog RTL was used for hardware development.

Prime sizes of 11 and 12 bits were chosen resulting in a decent memory capacity for exhaustive testing; the implementation used $2027, 2081, 2111$, and 2113. The Bloom Filter determines if a value is *potentially present* or *definitely absent*. These remainders index the filter bit-vector.

The implementation of the Bloom filter consists of four parallel Bloom filter banks implemented using FPGA registers where each bank corresponds to a separate hash function all operating on breakpoint-related execution information. During breakpoint learning, the processor's program counter (PC) values corresponding to user-defined hardware breakpoints are passed through the modulo hash functions explained above, generating multiple hash indices simultaneously. These indices are then used to set bits across several distributed bit-vector banks, allowing the Bloom filter to compactly encode a large history of breakpoint locations without requiring a dedicated comparator register for each breakpoint. During runtime execution, the live PC, specifically the instruction fetch address from the instruction pipeline, is continuously hashed using the same set of hash functions, and the indexed bits from all Bloom filter banks are checked in parallel. If all corresponding output bits are asserted, the filter reports a Bloom filter hit, indicating that the currently executing instruction address likely matches a previously programmed breakpoint entry. No true breakpoints are missed, but false break points can be triggered which are easily checked by the host followed by instruction to continue. The parameters of the Bloom filter design allow control over the number of supported breakpoints for a given false-trigger rate. Because the design is fully registered, all bit-vectors are implemented directly using synchronous registers or LUT-based distributed storage, enabling single-cycle lookup behavior, deterministic timing, and fast parallel access. This registered realization is especially useful for tightly integrated processor debug systems because it minimizes lookup latency while allowing thousands of potential breakpoint signatures to be represented using compact hash-based storage rather than large numbers of dedicated physical breakpoint comparators.

The Bloom filter is integrated within the Control and Status Register (CSR) module of the OpenTitan [13] Ibex core to extend hardware breakpoint functionality. It is tightly coupled with the trigger CSR path (tselect, tdata1–3), where breakpoint values written by the debugger are reused for Bloom filter learning. A multiplexer selects between the trigger register write data and the program counter, allowing the same datapath to serve both learning and lookup phases. During learning, breakpoint addresses are hashed using the modulo-based hash functions and stored in compact bit-vector banks. During execution, the program counter is hashed and compared against the stored bits to generate a probabilistic hit. The Bloom filter output is ORed with the exact comparator-based trigger

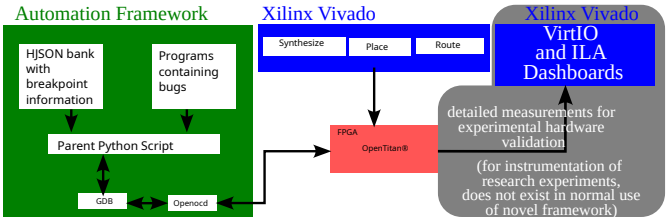

Fig. 2: Exhaustive Test Automation Flow.

match, enabling scalable breakpoint detection beyond the limited hardware trigger count with minimal hardware overhead. Because of our current implementation with minimal change to the existing processor hardware, debug mode is entered on the following instruction. This could be mitigated by setting a breakpoint on the previous instruction using the large set of breakpoints supported, though additional signaling and control within the processor controller is being investigated for future improvement.

A. Automating Breakpoint Execution and Data Collection

The experimental framework is built around a Python-based automation script that orchestrates compilation, program loading, breakpoint programming, and execution control for the RISC-V based OpenTitan core. Each experiment is described using a structured HJSON configuration file, which specifies the source program, toolchain parameters, breakpoint locations, and GDB command sequence. The HJSON files explicitly reference ("source") a set of buggy C programs designed to exercise specific control-flow or data-flow anomalies, enabling systematic evaluation of the proposed Bloom filter–based debugging mechanism.

For each experiment, the automation script invokes a cross-compilation toolchain (in this case, `riscv64-unknown-elf-gcc`) to generate an ELF binary, which is then converted into a raw binary using objcopy. The binary is loaded into the target memory via GDB, which communicates with the hardware using OpenOCD [14] and a JTAG interface. The script initializes the processor state by explicitly setting the program counter and stack pointer to safe memory regions before execution begins.

A key feature of the workflow is the controlled insertion of hardware breakpoints. Due to the limited number of hardware trigger registers in the Ibex/OpenTitan design, the script employs a serial programming strategy: it issues an `hbreak` command to program a breakpoint into the trigger CSR, followed by a deletion to free the hardware resource (compare-match register). Although only one or few physical breakpoint exists at a time, each previously programmed address is captured by the integrated Bloom filter logic, allowing multiple breakpoint locations to be accumulated efficiently. Execution is driven using bounded stepping or controlled run commands to prevent indefinite execution. All interactions, including compilation output and GDB responses, are logged for analysis. Importantly, the automation to test the new debug platform supports sequential execution across experiments decribed in multiple HJSON files that define codesets and breakpoint sets. By disabling processor reset between runs, the Bloom filter retains its state and is incrementally populated across different programs.

IV. Testing Results

A large-scale automated demonstration was conducted to evaluate the proposed Bloom filter based breakpoint-history and debug-trigger infrastructure integrated within the OpenTitan Ibex RISC-V core. A total of 1000 diverse bare-metal C programs were generated and executed using a Python-driven experimentation framework. The generated programs were intentionally designed to exercise a wide range of control-flow and computational behaviors commonly encountered in embedded software debugging workloads. The corpus included arithmetic kernels, branch-heavy conditional logic, nested-loop computations, array traversal operations, checksum and parity calculations, bit-manipulation routines, state-machine style logic, iterative accumulators, and pointer-based update functions. For each generated C program, five breakpoint candidates were selected and recorded in the HJSON file. Another 13 programs were hand-crafted for a total of 1013 execution results whose logs where analyzed. These breakpoints were chosen from locations placed at critical execution points, such as arithmetic assignments, loop-body updates, state-machine transitions, array-processing statements, checksum/hash updates, branch-controlled computations, calls from `main`, and final result/trace updates. Each generated program incorporated a dedicated `_start` entry point to support standalone execution without relying on operating system services or standard runtime initialization libraries.

Each configuration HJSON specified the source file, compilation settings, OpenOCD/GDB target parameters, stack and load addresses, and the selected breakpoint locations distributed throughout the code. The selected breakpoints were automatically inserted into the Ibex debug infrastructure using GDB's hardware breakpoint mechanism operating with `always-inserted mode:on`, enabling all breakpoints to be transferred so that the Bloom filter hardware would learn the associated program counters (PCs) from the set of all previously entered hardware breakpoints.

Experiments were conducted using the Digilent Nexys Video Artix-7 FPGA Trainer Board which is a FPGA development platform built around the Xilinx Artix-7 XC7A200T FPGA. It provides substantial on-chip resources including about 33,650 logic slices, 13 Mbits of block RAM, 740 DSP slices, 285 I/O, 32 BUFG, and 10 PLL blocks. The design was synthesized with a 100 MHz system clock specified in the Xilinx Design Constraints (XDC) file, which serves as the operating frequency of the OpenTitan RISC-V processor core and the integrated Bloom filter debugger. All the implementations were carried out using Xilinx Vivado 2024.2.2 on a 64 core server with 512 GB RAM. The parameter DbgHwBreakNum with the OpenTitan processor RTL was changed to 5000 in the control status register module in order

TABLE I: Breakpoint vs Bloom Filter Utilization

	Resource	Utilization	Available	Util.(%)
(a) Synthesis estimates of core with 5000 hardware breakpoints	LUT	223748	134600	166.23
	LUTRAM	1364	46200	2.95
	FF	232209	269200	86.26
	BRAM	250	365	68.49
	DSP	30	740	4.05
	IO	49	285	17.19
	BUFG	23	32	71.88
	PLL	1	10	10.00
(b) Synthesis estimates of core with Bloom filter using 12-bit long primes	LUT	121767	134600	90.47
	LUTRAM	1364	46200	2.95
	FF	75492	269200	28.04
	BRAM	250	365	68.49
	DSP	30	740	4.05
	IO	49	285	17.19
	BUFG	23	32	71.88
	PLL	1	10	10.00
(c) Implementation metrics of core with the Bloom filter	LUT	117884	133800	88.10
	LUTRAM	838	46200	1.81
	FF	74920	267600	28.00
	BRAM	250	365	68.49
	DSP	30	740	4.05
	IO	49	285	17.19
	BUFG	18	32	56.25
	PLL	1	10	10.00

to create the required breakpoint capacity in the core. The design failed at the implementation stage due to a lack of LUT resources generating error code DRC UTLZ-1 and it was reported that the design would have required 221110 LUTs which happens to be 165.2% of the available resource limit. The synthesis metrics depict 166.23% LUT utilization and 86.26% FF utilization in part (a) of Table I. On the other hand, the Bloom filter integrated with the core required a total of 117884 LUTs comprising of 88.10% of the total available LUTs as shown in part (c) of Table I while the core itself utilizes 79% of LUTs. This proves that when it is impossible to implement large number of hardware breakpoints in the traditional sense, a moderately sized Bloom filter constitutes only about 9% of available resources.

For the hardware breakpoint testing, execution logs captured detailed runtime information including breakpoint-setting events, resolved hardware breakpoints, instruction traces, register dumps, Bloom filter hits, false-positive detections, and total halt counts (hits). If the PC at a halt corresponds to one of the breakpoints, it is counted as a valid hit. If the PC at a halt does not match any breakpoint, it is counted as a *false positive*. 5065 breakpoints insertion events (not necessarily unique) were performed before executions. Because the total capacity of the Bloom filter is 8332 bits, there were a high number of collisions and *false positives*. The *false positive rate* observed in our experiments was extremely high at 45% which was a useful extreme use case to observe. This experiment took 12 minutes to run. The number of halts recorded was 2781 with a good distribution among the 5 breakpoints (BP1 through BP5): 238 at BP1, 931 at BP2, 695 at BP3, 567 at BP4, and 350 at BP5. This demonstrated the ability of the system to halt at various breakpoints and for GDB to process the events. We note that one type of anomaly was observed only when breakpoints were set at consecutive program addresses, in which GDB would not provide user interaction at the second breakpoint encountered. To ensure that our hardware design was operating as intended, we used an Integrated Logic Analyzer to directly observe entirely correct operation of the new Bloom filter trigger in hardware and even that the processor system controller raised the debug-mode signal at each of the two locations. The cause for the phenomenon of the debug mode being exited before GDB provided user interaction for the consecutive location is under investigation.

V. Future Work and Conclusion

There is an ongoing effort in translating the Bloom filter implementation using Block RAM (BRAM) in order to further optimize resource utilization in the FPGA. The idea behind this is to be able to execute exhaustive debugging capabilities with leftover memory in the FPGA or eventually in first-run silicon prototypes released by manufactures or even final products. On the exhaustive testing side, code mutation tools (e.g. dextool mutate [15]) are being studied and experiments are being designed to inject a plethora of mutations in pieces of C code and observe theextent to which the bloom filter can detect the faults and help in reducing debug time.

The presented results proves that a Bloom filter-based hardware breakpoint mechanism provides a highly scalable alternative to physically increasing the number of dedicated breakpoint registers within a processor core. Instead of allocating expensive comparator logic and debug registers for every additional breakpoint, the Bloom filter compactly stores large numbers of breakpoint signatures using lightweight hash logic and LUTRAM resources, enabling support for hundreds or even thousands of breakpoint entries with minimal area overhead. This approach is significantly more feasible for modern SoC integration because the Bloom filter can be implemented as a reusable synthesizable IP block that interfaces cleanly with existing debug infrastructure, unlike dedicated breakpoint products which often require intrusive architectural modifications and still primarily target only traditional software breakpoint functionality.

References

[1] M. J. Lyons and D. Brooks, "The design of a bloom filter hardware accelerator for ultra low power systems," in *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, 2009, pp. 371–376.

[2] A. Vasudevan and R. Yerraballi, "Stealth breakpoints," in *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005, pp. 10–pp.

[3] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2021.

[4] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, 1st ed. USA: Prentice Hall PTR, 2008.

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[6] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman, "Exact and approximate membership testers," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ser. STOC '78. New York, NY, USA: Association for Computing Machinery, 1978, p. 59–65. [Online]. Available: https://doi.org/10.1145/800133.804332

[7] Free Software Foundation, *Debugging with GDB: The GNU Source-Level Debugger*, GNU Project, 2026, version 18. [Online]. Available: https://www.gnu.org/software/gdb/

[8] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.

[9] Wikipedia contributors, "Bloom filter — Wikipedia, the free encyclopedia," 2025, [Online; accessed 4-April-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bloom_filter\&oldid=1273055182

[10] CS dept. Instructors & Course Associates, Stanford University, "Esoteric data structures: Skip lists and bloom filters," 2021, lecture Viewgraphs for Lecture No. 27 in the course/class CS 106B in Spring 2021. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1216/lectures/27-esoteric/slides

[11] A. Ranjan, "Demystifying bloom filters with real-life examples," 2023, accessed: 2025-04-22. [Online]. Available: https://medium.com/@abhishekranjandev/demystifying-bloom-filters-with-real-life-examples-b66db7e37b37

[12] K. Karantias, "Sok: A taxonomy of cryptocurrency wallets," *Cryptology ePrint Archive*, 2020.

[13] lowRISC CIC and the OpenTitan Contributors, "OpenTitan: Open source silicon root of trust," https://github.com/lowRISC/opentitan, 2026, gitHub repository.

[14] The OpenOCD Project, "Open on-chip debugger (openocd)," https://openocd.org/, 2025, open-source on-chip debugging, in-system programming, and boundary-scan software.

[15] J. Brännström, "Dextool: Suite of c/c++ tooling built on llvm/clang," https://joakim-brannstrom.github.io/dextool/, 2025.