Database Driven RTL Simulation: Fighting Billion-Gate Verification Bottlenecks

Victor Besyakov
BTA Design Service, IC Verimeter, Ottawa, Canada

Abstract-As AI ASIC designs scale beyond billions of gates, conventional RTL simulation runtimes grow exponentially. Prolonged execution time significantly reduces verification productivity by extending debug cycles and slowing test regression. This paper introduces the SVDB Gateway, an open-source SystemVerilog (SV) Direct Programming Interface for C (DPI-C) library that offloads verification data to external SQLite databases during runtime. This approach reduces memory footprint, enhances compilation speed, and improves simulation throughput for data-centric verification tasks. The solution is specifically designed to address infrastructure bottlenecks in register management, configuration databases, transaction logging, and regression data analysis, where data storage and retrieval represent the primary performance constraints. It provides a scalable data management solution with minimal modifications to the existing UVM testbench infrastructure.

I. INTRODUCTION

A. ASIC Architectural Trends

Artificial intelligence (AI) workloads require parallel processing of massive datasets for both neural network training and real-time inference. Modern AI accelerators must process more data per clock cycle than traditional SoCs. They should minimize communication overhead when integrated into system-in-package designs or multi-chip systems. As a result, contemporary devices integrate heterogeneous third-party IP blocks, large arrays of parallel compute units, and complex hierarchical networks-on-chip (NoCs), along with multi-level memory systems that incorporate both on-chip and off-chip components [1][2][3][4].

These trends drive AI ASICs to grow in size and complexity. Adding more parallel compute units and larger data buffers expands design scale beyond traditional limits. Chiplet integration addresses these demands by enabling multiple tiles to work together without the constraints of a single monolithic die. This combination of scale and modularity makes billion-gate, multi-tile AI ASICs the industry standard for today's most demanding workloads. This architectural complexity directly translates into fundamental verification challenges.

B. Verification Bottlenecks

The growing size of AI ASICs, reaching multi-billion-gate capacity, pushes traditional RTL simulation to its limits. The bottleneck arises mainly from four sources: excessive simulation runtime, massive log and waveform files, multi-platform result fragmentation, and slow coverage closure [5][6][7].

Simulations for large designs can run for days or weeks, generating terabytes of log and waveform data. Results from simulation, emulation, and FPGA prototyping come in different formats, making cross-platform comparison difficult. Coverage closure is slow because multi-component interactions leave many coverage points hard to hit [6][8][9]. Hardware-software co-verification adds non-determinism from firmware. This can reopen previously closed coverage and extend debug cycles.

These cumulative issues result in a situation where design complexity outpaces verification capability. This imbalance is known as the Verification Productivity Gap 2.0 [7][8]. The evidence of this gap is clear: more designs now require respins. A recent survey shows that first-time success with silicon dropped to 14%, down from approximately 30% historically [6].

C. Motivation for Database Offload

External database offloading provides a fundamentally different approach to verification scalability. Rather than managing data within simulation memory or dumping it to files, offloading moves storage and retrieval operations to an external database during runtime. This keeps the simulation process lean, reduces memory pressure, and enables efficient data access via indexed queries instead of sequential file scans. Data management represents a notable performance constraint in large-scale designs. This paper examines specific examples: very large register models, high-volume transaction logs, UVM configuration database operations, and multi-source verification data. Each example presents distinct challenges that standard simulation frameworks struggle to resolve.

Large register models consume significant memory due to static instantiation patterns. Traditional UVM register abstraction layers (RAL) [16] build the complete register hierarchy by creating dedicated class objects for each field, register, and block. Complex SoCs with many register fields instantiate all objects upfront regardless of test

requirements. The factory mechanism alone introduces substantial overhead when managing large class counts. Simulation performance suffers because unused register objects occupy memory throughout execution, increasing both process memory consumption and compilation time. This becomes critical in designs where register models dominate the class count and strain simulator resources.

The UVM configuration database presents a different data access problem. Configuration operations rely on string-based lookups with wildcard scope matching, requiring regular expression evaluation across all entries [27]. As designs grow to include many configuration parameters, these lookups exhibit poor scaling behavior with data volume. Build phase delays can stretch for extended periods simply processing configuration set and get operations. Beyond runtime overhead, configuration data persists only during simulation and cannot be compared across multiple test runs or preserved for audit.

High-volume transaction logging stresses I/O and storage systems. Simulations generate transaction volumes measured in millions of entries. Traditional file-based logging achieves only modest throughput, creating storage bottlenecks. Write operations consume both disk bandwidth and capacity. Post-simulation analysis requires random seek access to massive trace files, incurring per-access latency costs. As verification scales, transaction volumes accumulate across regression suites, consuming substantial storage.

Multi-source verification data creates integration challenges. Simulation generates coverage files in one format, emulation produces proprietary formats, C code coverage comes from different tools, and FPGA results appear in trace captures. No unified schema exists across these sources. Fragmented coverage data across verification methodologies makes cross-methodology comparison labor-intensive and prevents automated analysis.

Major simulation vendors use performance optimization strategies like multi-threading, checkpoint/restart frameworks, and distributed simulation to reduce memory and computational overhead. However, these techniques address computation and execution patterns, not the underlying data-volume problem. They do not reduce the fundamental memory footprint or enable efficient data access for verification data that could persist beyond a single simulation run.

A database-driven approach solves these problems through a different mechanism. Offloading register definitions to external storage enables lazy loading, where objects are instantiated only when accessed during runtime. Memory footprint scales with actual register usage rather than total register count. Configuration data is moved to an indexed database, eliminating regex overhead and enabling database queries to run quickly. Transaction logs shift from file-based sequences to normalized database tables, supporting concurrent reads and allowing substantially higher operation throughput. Multi-source verification data maps to a common database schema, enabling vendor-agnostic queries across all coverage sources and verification methodologies. External databases also provide persistence, allowing cross-run regression analysis, automated Continuous Integration/Continuous Deployment (CI/CD) integration, and trend reporting without requiring simulator modifications.

II. SVDB Gateway Architecture and Design

The SVDB Gateway [15] provides a shim layer bridge between SystemVerilog verification environments and external SQLite [20] databases. This section provides a brief overview of the SVDB architecture, component interactions, and data flow mechanisms. For detailed information and implementation guidance, refer to the online documentation available in the SVDB Gateway GitHub repository [15]. Additional information on database features and technical details can be found on the official SQLite site [20]. Please note that all functions starting with the <sqlite3> prefix are from the SQLite C Interface [20].

A. Overall Architecture

The SVDB Gateway implements a three-layer architecture: the SV layer, the DPI-C interface, and the database engine. Each layer performs a specific function with clear separation from the others. This design allows verification data to move from the RTL simulation, through the testbench, and into the external database (Figure 1).

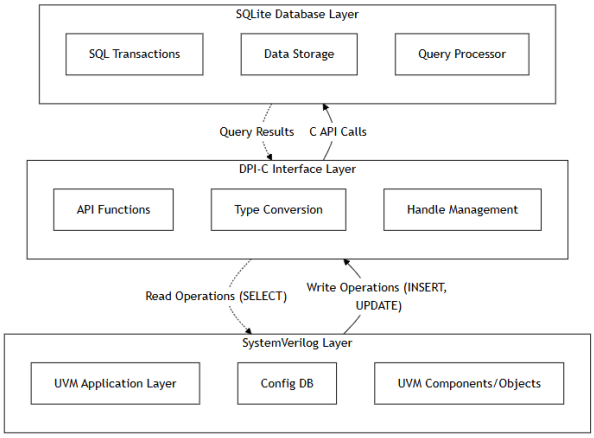

Figure 1. Three-Layer Architecture of SVDB Gateway

The SV layer contains UVM testbench components such as application models, configuration databases, coverage collectors, and test sequences. These components interact with the external database through DPI-C function calls using standard SystemVerilog syntax [17].

The DPI-C interface bridges SV and C. Its API maps SV calls to SQLite operations [20], enabling database use without knowledge of C APIs or SQL. The interface also converts data types between SV and C automatically [17].

SQLite is a lightweight and reliable database engine. It runs without a separate server and does not require database administration. The database file is compatible with all major operating systems [20], allowing results to be shared across different hosts.

This architecture offers a clear separation of duties across three layers. The SV layer handles test stimulus and checking. The DPI-C interface manages data translation and API calls. The database engine layer provides on-disk storage and transaction control. The use of standard SV DPI-C keeps the solution independent of any one simulator. It works with all commercial simulators and with open-source tools such as Verilator [18][15].

B. DPI-C Interface Layer

The DPI-C interface layer implements the translation mechanism between SV and SQLite. The interface design focuses on minimizing overhead while maintaining data integrity across the language boundary.

The API organizes functions into several categories based on the direction of data flow and the type of operation. TABLE I Summarizes the primary DPI-C functions organized by category and data flow direction. The database *Connection Management* controls the lifecycle of database sessions. The *Write Operation* transfers data from SV into the database. The *Read Operation* retrieves data from the database and sends it back to SV. *Transaction Control* functions manage properties across multiple operations [19].

TABLE I
DPI-C API FUNCTIONS ORGANIZED BY CATEGORY

Category	Function Name	Purpose	Data Flow Direction
Connection Management	sqlite_dpi_open_database()	Establish database connection	SV → C → SQLite
	sqlite_dpi_close_database()	Close database connection	SV → C → SQLite
Write Operations	sqlite_dpi_insert_row()	Insert data into table	SV → C → SQLite
	sqlite_dpi_execute_query()	Execute SQL statement	SV → C → SQLite
Read Operations	sqlite_dpi_get_row()	Retrieve complete row	SQLite → C → SV
	sqlite_dpi_get_cell_value()	Retrieve single cell value	SQLite → C → SV
	sqlite_dpi_get_rowid_by_column_value()	Find row ID by value	SQLite → C → SV
Transaction Control	sqlite_dpi_begin_transaction()	Start transaction	SV → C → SQLite
	sqlite_dpi_commit_transaction()	Commit changes	SV → C → SQLite
	sqlite_dpi_rollback_transaction()	Revert changes	SV → C → SQLite

The DPI-C layer implements automatic type mapping between SV data types and SQLite-compatible formats. This conversion process occurs transparently at the language boundary, ensuring that data maintains semantic equivalence as it flows between layers. TABLE II shows the type mapping relationships that govern data conversion. Each SV type maps to a specific C type at the DPI-C boundary. This boundary then maps to an appropriate SQLite storage type.

The table represents only basic data mapping. More complex data mapping is beyond the scope of the SVDB framework. Custom mapping solutions can be constructed from basic transaction types. Refer to the SVDB [15] example directories for implementation details.

TABLE II
TYPE MAPPING ACROSS SYSTEMVERILOG, DPI-C, AND SQLITE

SV Type	DPI-C Type	SQLite Type	Data Flow Notes
Int	int	INTEGER	Direct mapping, 32-bit signed
String	const char*	TEXT	Null-terminated C string
bit[N:0]	unsigned char*	BLOB	Binary data, requires size parameter
Chandle	void*	N/A	Pointer for DB handle
Byte	char	INTEGER	8-bit signed value
Shortint	short	INTEGER	16-bit signed value
Longint	long long	INTEGER	64-bit signed value
Real	double	REAL	IEEE 754 double precision

C. Forward Data Flow

Figure 2 shows the forward data flow path. It begins when the DUT generates signal transitions or register changes. UVM monitors capture these transitions and construct transaction objects. The monitors send transactions to subscribers through TLM analysis ports using standard UVM communication mechanisms.

Transaction subscribers receive the transaction objects and extract relevant data fields. The subscriber formats this data into an appropriate data type suitable for passing to the DPI-C function as input.

The subscriber invokes the appropriate DPI-C function to store data in the database. For example, for transaction logging, it calls sqlite_dpi_insert_row() with parameters including the database handle, table name, column specification, and value specification. The function call crosses the DPI-C boundary from SV into C.

Type conversion happens automatically at the boundary, as shown in TABLE II. The C implementation parses the string SQL query parameters to extract individual column names and values. The C layer constructs SQL statements using SQLite's API (see Figure 2 DPI-C and SQLite interaction and SQLite docs for details [20]).

The execution returns a status code indicating success or failure. The C implementation checks this status and converts it to a DPI-C return value. The return value propagates back across the DPI-C boundary to SV. The SVDB checks the return status and implements error handling logic.

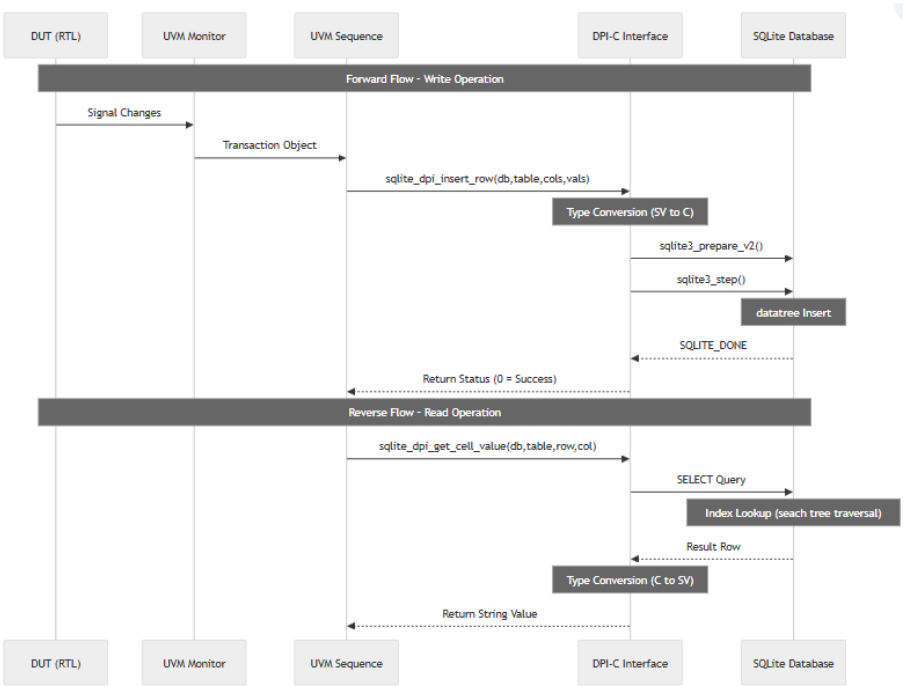

Figure 2: Forward data flow (Write Operations)

The DPI-C layer groups transactions into a single batch for completion. (see Figure 3). Batching reduces disk synchronization overhead by consolidating multiple row inserts into a single operation.

For SQLite write operations, a transaction begins with *sqlite_dpi_begin_transaction()*. All writes execute until *commit* is called. The database engine flushes data to disk and returns a status code through DPI-C for error handling.

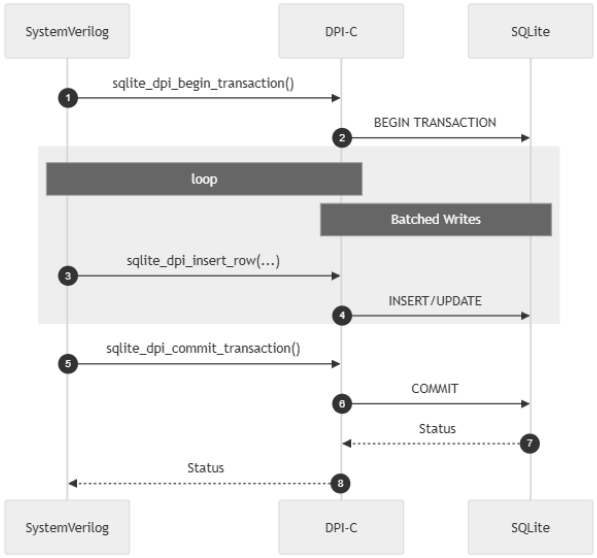

Figure 3: Forward data flow (Write Batch Mode)

D. Reverse Data Flow (Read Operation)

The reverse data flow path supports database queries and data retrieval. UVM components invoke DPI-C query functions to retrieve data from external storage. Figure 4 shows, read operations begin when the SV code calls a query function such as sqlite_dpi_get_cell_value(). The function parameters specify the target data location (table name, row identifier, and column name). The call crosses the DPI-C boundary into C.

The C layer constructs statements based on the parameters. For cell-level queries, the statement retrieves a single column value from a specific row identified by the primary key. For detailed information and implementation guidance, refer to the online documentation available in the SVDB Gateway GitHub repository [19]. This targeted access minimizes data transfer compared to full table scans. Statement preparation follows the same process as write operations. The C implementation calls sqlite3_prepare_v2() to compile the SELECT statement. SQLite's query optimizer analyzes the statement and generates an execution plan. Query execution proceeds through sqlite3_step(). SQLite navigates the data tree structure to locate the requested data. Result extraction occurs after successful execution.

Then, type conversion transforms the SQLite result to a DPI-C compatible format. For string results, the C layer constructs a null-terminated character array. For numeric results, it performs type conversion to match the expected return type. The result value returns across the DPI-C boundary to SV. The calling code receives the return value and continues processing.

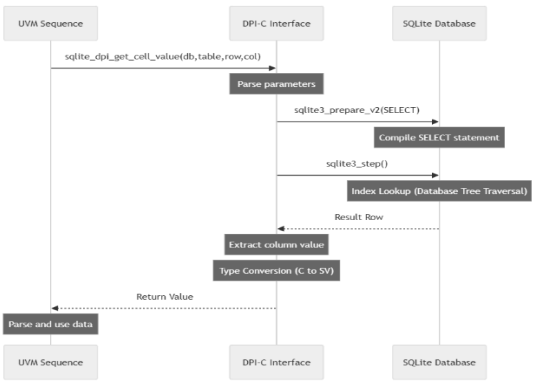

Figure 4 Reverse data flow (Read Operation)

E. SVDB Performance and Optimization Strategies

The architecture's performance characteristics derive from the interaction of all three layers and the optimization strategies applied at each level. Several optimization strategies enhance data flow performance. These optimizations reduce overhead at each layer while maintaining data integrity and consistency [20][22]. For example, transaction batching groups multiple database operations within a single SQLite transaction. It reduces expensive operations that synchronize data to disk. Batching improves throughput from hundreds of operations to tens of thousands per second. Index optimization accelerates query operations. For large databases, indexed queries can be up to 100 times faster than unindexed queries. SQLite architecture enables concurrent read access during write operations. Multiple verification threads can query the database while another thread performs inserts.

Measurements show specific benefits in key areas relevant to billion-gate verification. The database connection overhead is spread out over the duration of the simulation run. Opening a database connection requires milliseconds, but this cost occurs only once at simulation startup. The connection remains open throughout the test, enabling thousands of operations to share the initialization cost. The latency of the insert operation depends on the transaction mode. Individual inserts with immediate commit take approximately 10 milliseconds due to disk synchronization. Batched inserts within transactions reduce per-operation latency to tens of microseconds, achieving throughput of 50,000 to 100,000 inserts per second [23][24]. Query operation latency depends on indexing and result set size. Indexed queries on primary keys complete in microseconds. Unindexed table scans scale linearly with table size. For a table with one million rows, indexed lookup takes microseconds while a full scan takes seconds [20][22].

Memory overhead is minimal compared to simulation. The database engine maintains a cache of recently accessed pages, typically sized at a few megabytes. Total CPU memory usage remains constant regardless of the database size. As the design size increases, the database size grows linearly. Query performance remains constant because SQLite's built-in indexing management maintains efficient data retrieval regardless of data volume.

III. APPLICATION LAYER EXAMPLE (DATABASE-DRIVEN UVM REGISTER ABSTRACTION LAYER)

The UVM RAL provides essential capabilities for register verification in complex SoCs. However, traditional implementations face significant challenges in scaling billion-gate designs. This section examines these limitations and presents a database-driven dynamic RAL approach that leverages SQLite for runtime register management while preserving UVM compatibility. The following subsection details the implementation of this concept. It is based directly on the example provided in the SVDB Gateway git repository[15].

A. Traditional RAL Limitations

Traditional RAL relies on static generation of whole register models. This approach works for smaller designs but creates inefficiencies as register counts exceed tens of thousands. Static register model generation creates bottlenecks by instantiating all registers at ones. In complex SoCs, "we can easily have tens of thousands of register fields in the model," where each field, register, and block requires dedicated classes. This approach follows standard UVM guidelines but results in "a significant load and build-time performance penalty" because the model must handle the full hierarchy regardless of test needs. As a result, verification environments load unused registers, reducing overall efficiency [11].

For large models, "each field, register and hierarchical block defined by classes, we often have considerably more SV classes in the register model than the rest of the verification environment". This volume causes "a significant load

and build-time performance penalty" compared to environments without the full model. In practice, using it for backdoor loading of thousands of registers can significantly increase compilation and runtime [10][11].

The factory mechanism alone introduces substantial costs, as "the factory's role in dealing with the large number of classes" creates additional overhead for 10,000-plus fields. Static models retain all register objects in memory throughout the simulation, even if most remain unused. As documented in industry forums, "simulation performance is badly affected by uvm_reg env", with measurable increases in process memory consumption. This becomes critical for SoCs where register models dominate the class count, straining simulator resources [10][11].

B. Database-Driven Dynamic RAL Implementation

The database-driven approach moves register data to external SQLite storage. This enables dynamic loading that aligns with actual test demands, improving both performance and maintainability.

The lazy-loading model delays object creation until they are accessed. Rather than building the full register hierarchy at once, the register model configures them on demand during runtime. SQLite stores register definitions using a schema that captures addresses, fields, and attributes. The SVDB Gateway's DPI-C interface provides targeted access to retrieve specific data from the external SQLite database.

Memory footprint optimization techniques improve efficiency by avoiding unnecessary components. Coverage and factory registration are performed only for accessed registers, eliminating unnecessary overhead associated with full instantiation. For large models, this approach reduces compilation and runtime by removing static model overhead. In practice, "in a test, we may be needing only 10-20% of those registers depending on the feature it is required to cover" [12], allowing runtime memory usage to scale with actual register access patterns rather than full model instantiation.

C. UVM Integration Methodology

Figure 5 illustrates how the dynamic register model transforms a generic register template into specific register instances. During the uvm_build_phase, the environment opens the SQLite database and passes the handle to the dynamic model.

For each register, the test sequence calls the configure_register() function to retrieve specific register attributes and apply them to the generic svdb_dynamic_reg object. This transforms the generic template into a specific register. At this point, the dynamic register has become a specific register instance, allowing it to serve any register described in the database without requiring recompilation. During the run phase, test sequences issue standard RAL calls, such as read() and write(). These calls pass through the standard UVM adapter, agent, and predictor without any special handling for dynamic registers. The result is a drop-in path that loads definitions at runtime yet uses the same UVM flow as a static model.

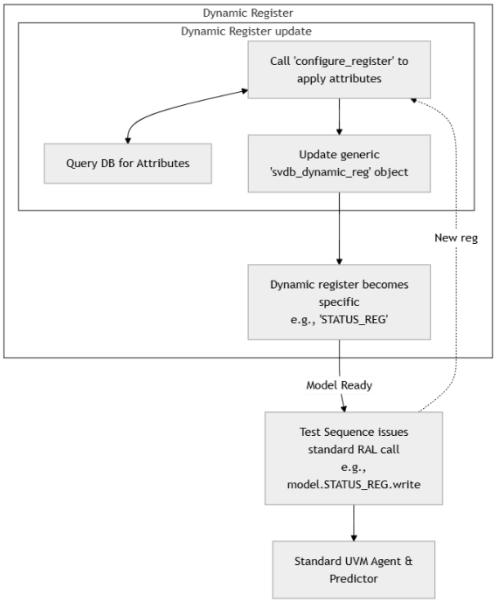

Figure 5. Dynamic Register

D. Discussion and Integration in Verification Flows

Standard UVM RAL provides robust verification for typical use cases but lacks flexibility for nonstandard applications. Conventional flow requires all configurations to reside in memory during simulation, which limits adaptability to evolving register maps or dynamic runtime environments. The SVDB Gateway addresses these

limitations by introducing a runtime configuration layer. Registers are loaded on demand through DPI-C calls to an external SQLite database, rather than through static compilation. SVDB hooks override key uvm_reg methods, enabling the model to adhere to standard RAL application programming interfaces (APIs) while supporting dynamic data retrieval. Flexibility comes with the benefit of easy fallback. With a mode selector, simulation can switch off database use and revert to a standard static RAL, needing minimal code changes. This ensures existing flows remain undisturbed and hybrid regressions are possible.

Another strength of the SVDB flow is its clean separation of specification from simulation. IP-XACT register descriptions are converted into a SQLite database using the provided Python script. The simulation only needs to select an appropriate database at runtime, enabling rapid testing of new register maps or silicon revisions without recompilation. This approach supports version control, rapid regression cycles, and concurrent testing of multiple chip configurations from a single testbench. For detailed implementation, refer to the SVDB Gateway GitHub repository [15].

E. RAL SVDB Performance

Testing on a PCIe-based AI ASIC design with approximately 70,000 registers achieved substantial improvements. Register field memory reduced from 141.47 MB to 7.99 MB (94% reduction). Compilation time reduced from 37 to 22 minutes (39% reduction). Runtime reduced from 6.4 to 4.1 hours (36% reduction). Peak memory consumption reduced from 24.62 to 21.27 gigabytes (13% reduction). All measurements were collected on a system with 2 TB of RAM and two Intel Xeon Gold 6248R processors.

The case study demonstrates the practical viability of database-driven RAL for billion-gate designs. Register models that previously dominated testbench class counts and compilation overhead now represent minimal contributors to total verification time.

IV. USE CASES

Database-driven RTL simulation offloads verification data from the simulator memory to external storage, enabling separation of compute from storage. SV executes the design logic while SQLite manages data persistence through the DPI-C interface. This approach leverages database techniques such as indexing and transaction batching to improve specific verification workflows. The following sections explore potential applications of this approach.

A. UVM configuration management

Traditional UVM configuration mechanisms face scaling challenges in large testbenches. The *uvm_config_db* and *uvm_resource_db* use string-based lookups stored in simulator memory. As testbenches grow, memory overhead becomes more significant. The *config_db* maintains name-value pairs that require runtime string searching and hierarchical scope matching. Each parameter lookup consumes CPU cycles. Persistence is another weakness. The configuration exists only during simulation. After runs are complete, configurations cannot be compared across multiple tests or reproduced for specific parameter combinations. Audit trails are difficult to maintain.

SVDB could solve these problems by offloading configuration data to external SQLite databases. Database lookups via indexed queries replace string searching. Storing configuration in external databases enables easy comparison and preserves audit trails across multiple test runs. DPI-C calls integrate transparently with existing testbenches, requiring minimal infrastructure changes while providing massive scalability improvements.

B. Logging and triage

Logging adds overhead to simulation runtime. Complex testbenches with many monitors and scoreboards amplify this effect, and it becomes more significant in long-running tests. To improve performance, regression runs typically disable logging. This approach reduces observability, yet failed tests must still be rerun locally with logging enabled for debugging.

Post-simulation analysis presents additional challenges when dealing with text-based logs. A single transaction may be recorded across dozens of log entries from different monitors and checkers. These entries appear in temporal order but contain no explicit links to related information. Locating relationships between failures and related data requires manual analysis.

Structured databases would enable a fundamentally different approach. Events are stored in normalized tables with columns for event type, timestamp, transaction ID, source component, and payload. This structure persists throughout the simulation and is not discarded on test failure. SQL queries locate failing packets using any requested attributes. Results are generated rapidly. Multiple test runs store data in a single database. Trends across regression cycles become visible and comparable across all hierarchical levels at any point in time.

C. Coverage and Cross-Platform Challenges

Alternative database-driven approaches to functional coverage further demonstrate the viability of SQL-based verification infrastructure [26]. Verification relies on multiple coverage metrics and spans multiple platforms. Each platform generates isolated coverage artifacts. For example, simulation coverage generates UCDB files. Emulation data uses proprietary formats. C model coverage might come from gcov [25]. FPGA results appear in trace captures. No unified schema exists. Fragmented coverage across platforms makes coverage closure labor-intensive.

SVDB can address these problems by providing a common database schema for all coverage types and platforms. This approach reveals potential opportunities for process automation. Coverage events can be written to normalized tables regardless of source. RTL simulation coverage would write to the same tables as emulation results. C code coverage could integrate alongside hardware functional coverage. SQL queries would access all data uniformly. Queries such as "Show unreached bins across simulation and emulation" would return results directly. This schema is vendor-agnostic. Simulators would map to identical table structures. Multiple regression runs could accumulate coverage in a single database. Coverage gaps would emerge with minimal manual interaction.

D. Regression and CI/CD

Thousands of regression tests generate artifacts, including pass/fail results, runtime metrics, coverage, and logs. These artifacts are not linked to the parameters or stimulus that produced them. Failure analysis relies on manual correlation across many files. Without those links, key questions remain unanswered: Which parameter values appear most often in failures? Do specific stimulus patterns correlate with defects?

SVDB provides the opportunity to store regression data in open-source, simulator-agnostic queryable databases. SQL queries could identify tests that fail multiple times across recent runs and reveal which parameter values correlate with failures. Trend analysis becomes straightforward, and configuration dependencies become visible automatically. Results across different hierarchical levels could be correlated efficiently. Block-level findings would link to subsystem-level data and system-level outcomes. Queries could reveal how defects propagate from blocks through subsystems to the complete system. This visibility would enable test direction optimization. Verification efforts could focus on levels where defect detection is highest. Root cause analysis would become possible across the entire hierarchy.

With comprehensive historical data, regression processes would become intelligent and adaptive. Predictive analysis would identify tests likely to fail under specific conditions and skip them to conserve resources. Long-running passing tests would run less frequently, while high-defect-detection tests would run early in cycles. Initial regression passes would run critical tests. If failures appear, expanded test suites would activate automatically. Configuration-aware selection would focus verification efforts on high-value tests. Resource allocation would be optimized based on coverage contributions. CI/CD pipelines would gain data-driven decision capability and improve efficiency as data accumulates.

V. USE CASES PERFORMANCE EVALUATION

This section presents a quantitative analysis of performance improvements achieved by applying database-driven approaches to key verification use cases. The evaluation covers measured results from the SVDB Gateway implementation and estimated savings from industry sources.

A. DPI-C and UVM Overhead

Database-driven verification introduces latency at the SystemVerilog-to-C language boundary. DPI-C basic operations incur 0.25 to 0.35 milliseconds. Complex operations require 0.4 to 0.7 milliseconds [13][14]. SQLite read and write operations have different performance profiles. Indexed read operations complete in 0.1 to 0.2 milliseconds by navigating database structures [20][22]. However, unindexed reads slow down with table size, so indexing is essential. Write operations require an average of 0.3 to 0.5 milliseconds [20][22].

Combined DPI-C plus SQLite overhead totals 0.4 to 0.7 milliseconds for read operations and 0.4 to 1.2 milliseconds for write operations. Cumulative latency depends on operation frequency and workload mix. Unoptimized individual write operations with immediate disk synchronization incur a 10-millisecond overhead per operation. Transaction batching improves throughput to 50,000-100,000 operations per second [23][24]. Index optimization provides 100x performance improvement for read queries. [20][22].

B. UVM Config_db and resource_db

The UVM configuration database exhibits severe performance degradation with wildcard-based scope matching. Empirical measurements show that 5,000 configuration set/get operations using wildcards require approximately 200 seconds [27], while the same operations complete in 2-3 seconds with an external database approach. For 10,000 operations, the UVM configuration database requires more than 6 minutes [27], compared to 2-3 seconds with external storage.

The root cause is algorithmic: the traditional config_db implementation iterates through all entries and builds regular expressions for wildcard matching, resulting in O(n x regex_cost) complexity. Large designs with 20,000-30,000 configuration calls during the UVM build phase experience delays of 20-30 minutes [27].

Offloading configuration data to external SQLite databases eliminates regular expression (regex) overhead, achieving measurable improvements across four categories.

Data Volume: Configuration storage is reduced by 79-80% through compression, saving 1.04-5.2 MB per simulation instance and 1-5 GB per 1,000-run regression suite.

Compilation: The build phase time improves 98-99% in the configuration database portion, reducing from 20-30 minutes to 12-18 seconds. Overall build time decreases from 2-4 hours to 15-30 minutes. Database generation from an IP-XACT source represents a one-time preprocessing cost. For 10 million entries, this conservatively requires 10 minutes. This cost is recovered within the first 2-3 regression runs [27].

Memory: The in-process memory footprint decreases by 40-60%, from 5-8 MB to 2-4 MB per simulation, resulting in 1-4 GB of system-wide savings across 1,000 parallel simulations.

Runtime: Configuration operation speed improves 144-160X, reducing simulation overhead from 2,000 seconds to 125 seconds for 50,000 operations. This delivers 5-15% overall simulation time improvement for configuration-heavy testbenches [27].

For billion-gate AI ASIC designs with 20,000-30,000 configuration parameters, cumulative savings reach 240-400 CPU hours per 100-test regression cycle, representing a 20-25% total improvement in build and simulation time.

C. Transaction Logging

ASIC verification simulations generate substantial transaction volumes during execution. Traditional file-based logging approaches typically achieve approximately 60 operations per second. Optimized SQLite provides roughly 5,000 to 15,000 operations per second. This represents an 80x to 250x improvement factor over file-based methods.

Beyond direct performance gains, SQLite provides significant storage and query benefits. A single database file eliminates the metadata overhead associated with thousands of individual log entries. This consolidation reduces overall disk usage for multi-gigabyte simulation logs compared to fragmented file-based approaches. Indexed database queries significantly outperform sequential parsing of multiple text files.

D. Limitations and Mitigation

SQLite databases theoretically support files up to 281 terabytes. This size far exceeds typical ASIC verification workloads. However, practical limits arise from hardware and filesystem constraints, as well as performance degradation with very large databases. The SQLite documentation confirms that "usually SQLite will hit the maximum file size limit of the underlying filesystem or disk hardware long before it hits its own internal size limit" [28]. An optimal SQLite database for verification, representing approximately 10 to 50 GB of raw text logs, typically remains below 10 GB to ensure optimal performance.

Another limitation is that SQLite supports unlimited concurrent readers but only one writer at a time, serializing updates in multi-simulator environments. For larger or highly concurrent workloads, open-source client-server databases such as PostgreSQL [30] and MySQL[29] could be recommended alternatives. These systems offer robust scalability, advanced multi-version concurrency control, and distributed architectures. Choosing the right database depends on workload size, concurrency, and integration complexity. SQLite excels for embedded, single-process applications with moderate data volumes. Larger verification environments will benefit from client-server databases for superior throughput and scalability.

VI. CONCLUSION

We introduced the SVDB Gateway, an open-source SV DPI-C library for offloading UVM data to SQLite. The architecture is simulator-agnostic and minimalistic, requiring only standard DPI and SQLite. This data-centric approach addresses billion-gate bottlenecks by reducing memory footprint and improving simulation throughput.

Our case study on a UVM testbench demonstrates the practical benefits of this methodology. SVDB achieved 39% compile-time and 36% simulation-time speedups by offloading register accesses to the database. The solution significantly reduced peak memory usage and shortened testbench initialization time. These improvements demonstrate that database-driven approaches effectively reduce verification time for billion-gate designs.

SVDB Gateway unlocks advanced data management workflows by enabling cross-run regression analysis, automated CI/CD integration, and trend reporting. SQL queries provide flexible access to simulation data, supporting complex post-run analysis and pattern identification. However, this solution complements existing verification methodologies and addresses a specific subset of verification challenges. For designs where data storage and retrieval represent primary performance constraints, SVDB provides a cost-effective framework. For verification environments where stimulus generation, checking logic efficiency, or coverage closure represent the primary bottleneck, alternative approaches may be necessary.

REFERENCES

[1] IBM, "What are AI workloads?," IBM Think Topics, May 2025 https://www.ibm.com/think/topics/ai-workloads

[2] M. Musavi et al., "Communication Characterization of AI Workloads for Large-scale Multi-chiplet Accelerators," arXiv preprint arXiv:2410.22262, Oct. 2024 https://arxiv.org/html/2410.22262v2

[3] EdgeCortix, "AI Drives the Software-Defined Heterogeneous Computing Era," EdgeCortix Blog, Jun. 2023 https://www.edgecortix.com/en/blog/ai-drives-the-software-defined-heterogeneous-computing-era

[4] Y. Liu et al., "Review of chiplet-based design: system architecture and interconnection," Sci. China Inf. Sci., vol. 67, no. 10, Oct. 2024 http://scis.scichina.com/en/2024/200401.pdf

[5] N. Balaji and S. C. Pandian, "Optimization of ASIC Design Cycle Using Different Verification Methodologies," Int. J. Sci. Eng. Adv. Sci., vol. 1, no. 3, pp. 65-70, 2025 https://ijseas.com/volume1/v1i3/ijseas20150365.pdf

[6] Wilson Research Group and Siemens EDA, "IC/ASIC Functional Verification Trend Report-2024," Siemens EDA Verification Acad., Feb. 2025 https://verificationacademy.com/topics/planning-measurement-and-analysis/wrg-industry-data-and-trends/2024-siemens-eda-and-wilson-research-group-ic-asic-functional-verification-trend-report

[7] H. Foster, "How AI And Connected Workflows Will Close The Verification Bottleneck," SemiEngineering, Mar. 2025 https://semiengineering.com/how-ai-and-connected-workflows-will-close-the-verification-bottleneck

[8] H. Foster, "Breaking the Bottleneck: Overcoming the Verification Productivity Gap 2.0," Verification Acad., Jan. 2025 https://verificationacademy.com/topics/planning-measurement-and-analysis/breaking-the-bottleneck-overcoming-the-verification-productivity-gap

[9] P. Kumar, "The impact of AI and machine learning in chip design," World J. Adv. Res. Rev., vol. 17, no. 1, pp. 1-15, Apr. 2025 https://journalwjarr.com/sites/default/files/fulltext_pdf/WJARR-2025-1318.pdf

[10] A. Yehia, "Boosting simulation performance of UVM registers in high-performance systems," in DVCon Proc., 2024. https://dvcon-proceedings.org/wp-content/uploads/boosting-simulation-performance-of-uvm-registers-in-high-performance-systems.pdf

[11] M. Litterick, "Advanced UVM register modeling," in DVCon 2014 https://dvcon-proceedings.org/wp-content/uploads/advanced-uvm-register-modeling.pdf

[12] S.Akkem,"UVM Registers On Demand: Elimination of the Unnecessary," DVCon India Proc. 2015 https://www.scribd.com/document/326412518/UVM-Registers-On-Demand

[13] P. Goel et al., " C you on the faster side: Accelerating SV DPI based co-simulation," in DVCon Proc 2014. https://dvcon-proceedings.org/wp-content/uploads/c-you-on-the-faster-side-accelerating-sv-dpi-based-co-simulation.pdf

[14] D. Velickovic and K. Bozinovic, "Functional verification using C model: DPI-C vs static value tables," in DVCon Proc., 2024. https://dvcon-proceedings.org/wp-content/uploads/153-Functional-Verification-Using-C-Model-DPI-C-VS-Static-Value-Tables.pdf

[15] "SVDB Gateway: SQLite Database Gateway for SystemVerilog," GitHub Repository, 2025 https://github.com/xver/svdb_gateway

[16] IEEE Standard for Universal Verification Methodology Language Reference Manual, 1800.2-2020, Dec. 2020.

[17] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, 28 Feb. 2024

[18] "Verilator" 24 July 2025 In Wikipedia https://en.wikipedia.org/wiki/Verilator

[19] DataCamp, "What Are ACID Transactions? A Complete Guide," Feb. 2025 https://www.datacamp.com/blog/acid-transactions

[20] "SQLite" 22 Dec 2025 In https://sqlite.org/index.html

[21] PowerSync, "SQLite Optimizations For Ultra High-Performance," May 2023 https://www.powersync.com/blog/sqlite-optimizations-for-ultra-high-performance

[22] "Best practices for SQLite performance," Sep. 2025 https://developer.android.com/topic/performance/sqlite-performance-best-practices

[23] James Sinkala "Batches in SQLite," Feb. 2024 https://turso.tech/blog/batches-in-sqlite-838e0961

[24] Moldstud, "Optimize SQLite Batch Processing with Combined Queries," Apr. 2025 https://moldstud.com/articles/p-efficient-sqlite-batch-processing-combine-multiple-queries-for-optimal-performance

[25] GCC Project, "Gcov: a test coverage program," GNU Compiler Collection Online Documentation, Accessed Dec 22, 2025, https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html

[26] A. Efody, "SQL - born for functional coverage (1)," LinkedIn Pulse May 18,2018, https://www.linkedin.com/pulse/sql-born-functional-coverage-1-avidan-efody

[27] R. Edelman, "Avoiding configuration madness the easy way," Siemens EDA, DVCon Proc, 2023 https://dvcon-proceedings.org/wp-content/uploads/1118-Avoiding-Configuration-Madness-The-Easy-Way.pdf

[28] SQLite Development Team, "Implementation Limits For SQLite," SQLite.org, Oct. 27, 2025 https://sqlite.org/limits.html

[29] "MySQL" 22 Dec 2025 In Wikipedia https://simple.wikipedia.org/wiki/MySQL

[30] "PostgreSQL" 22 Dec 2025 In Wikipedia https://en.wikipedia.org/wiki/PostgreSQL