The Cost of Native Calls in GraalVM's SQLite Runtime

Anonymous

Abstract

SQLite compiled to LLVM bitcode can run under Sulong and repeatedly cross GraalVM's native function interface to acquire and release glibc mutexes. We isolate the cost of this boundary on Linux x86-64. An opt-in JVM overlay recognizes two resolved pthread entry points, a fixed ABI signature, and a previously converted native pointer. It invokes the same functions through a typed JNI bridge with guest `errno` transfer and retains the generic path for other calls. The targeted Tier 2 compilation drops three array allocation sites and shrinks from 1,229 to 722 bytes. In six fresh-process blocks, the mutex-pair probe used 147.37 ns per pair for libffi, 35.92 ns for the published Panama backend bound to the same two guarded calls, and 18.23 ns for typed JNI. Six paired SQLite runs gave a median warmed CPU-time reduction of 63.5% versus libffi and 38.2% versus Panama. Typed JNI used a median 1.47× the warmed lookup CPU time of stock SQLite compiled with Clang. We release the source, per-process measurements, generated code, and a differential contract harness.

1 Introduction

GraalVM can run SQLite through Sulong, a Truffle language implementation that consumes LLVM bitcode [1]. The guest executes SQLite's existing query engine and uses the native function interface (NFI) to call host libraries. SQLite's serialized mode protects a connection and its derived objects with mutexes [2]. Even a single application thread using that mode executes locks and unlocks on each lookup. The host mutex operations in such a workload are typically uncontended. This creates a setting where the cost of crossing into native code can be larger than the cost of the native operation.

We ask whether a narrowly specialized downcall can materially change the cost of a warmed application workload while preserving the tested behavior of the original boundary. The generated Tier 2 code for `sqlite3_mutex_enter` retains three argument-preparation arrays on the libffi path. A fixed-signature JNI adapter avoids these allocations while using the same mutex implementation. Holding the guest module and overlay JAR fixed, switching the bridge on reduces the cost of serialized point lookups.

A planner that selects between database-backed tools can estimate initialization and compilation costs separately from the cost of repeated requests in a warm process. In this experiment, Panama had a lower median warm elapsed time than libffi and a higher median whole-process time. The two regimes can therefore lead to different backend choices. This study measures the database runtime. Agent policies and task outcomes are outside its experimental scope.

2 Design and verification

Generic downcall. The pinned libffi backend prepares a primitive byte array and arrays for possible argument patch metadata. In the captured mutex call, the argument is already a numeric native address and no object patch is required. Earlier Sulong argument vectors were virtualized in the captured graph, while the arrays at the NFI boundary remained. A diagnostic of the lookup workload counted six lock calls and six unlock calls per additional serialized lookup. That diagnostic motivates isolating the cost of the crossings in a separate lock-loop probe.

Guard and bridge. The test is evaluated after symbol resolution and argument conversion and before generic buffer preparation. It requires the fixed, nonvariadic NFI signature `(POINTER):SINT32`, a boxed `Long` argument, and a resolved address equal to one of the two corresponding symbols obtained from the loaded glibc. A disabled overlay or guard failure proceeds along the original libffi path. In particular a different provider can supply a function with the same signature. Its address keeps it on that path. The argument reachability fence and the normal JVM-to-JNI transition remain in use.

Let g be the resolved function address, p the unboxed pointer bits and e the guest thread's `errno` before the call. On the admitted ABI, the generic serializer writes exactly p into the pointer argument slot. The bridge writes e to libc's thread-local `errno`, invokes g with p as a `pthread_mutex_t *`, and reads the resulting e'. It encodes the return value r and e' into a JNI result word

$$w = 2^{32}\,\mathrm{u32}(e') + \mathrm{u32}(r).$$

The Java adapter restores guest `errno` from the high half and returns the signed 32-bit value from the low half. The equivalence argument for this guard uses the same function, pointer bits, thread-local input, return bits, and mutex side effects in both paths. It assumes the pinned Linux x86-64/glibc ABI. Other signatures, pointer representations, and providers continue through generic dispatch.

Table 1: Initial steady-state probe results. Medians of three fresh processes per arm for the mutex-pair probe and two per arm for serialized lookups.

Profile	Mutex pair (ns CPU)	Lookup window (ms CPU)
Native C	7.70	46.41
JVM, libffi	144.10	191.60
JVM, typed JNI	18.43	68.01

Targeted tests. The differential harness checks ownership and error returns on error-checking mutexes, guest `errno` sentinels, a native-to-guest callback, and an alternate provider with the same signatures and distinct observable return and `errno` behavior. Interpreter and forced Tier 2 runs each record 4,000 specialized glibc invocations and zero invocations of the bridge for the alternate provider. The counting build is separate from the timing build. The lookup contract matches native results and VM step counts on the tested cases and records Tier 2 installation. These checks establish the targeted behavior under the tested settings.

3 Evaluation

Platform and controls. The initial measurements use Linux 6.18.57, x86-64, glibc 2.41, GraalVM Community JDK 24.0.2+11.1, Sulong 24.2.2, SQLite 3.54.1 and Clang 18.1.3. The native baseline runs stock SQLite compiled with Clang `-O2`. Both JVM arms use the same guest LLVM module and overlay JAR. An opt-in runtime property selects the typed bridge. The lookup module uses the coalesced guest stack in both JVM arms. The native stock baseline supplies a reference for the residual performance gap. The bridge's causal comparison is between the two JVM arms.

Warm observations. The lookup panel uses 2,048 rows, seeded requests, 12,000 warm-up lookups, and ten windows of 150,000 lookups in each process. The runner rotates the profile order, checks results and checksums, and pins these initial JVM runs to CPU 6. Guest compilation is synchronous with a last-tier threshold of 3,000. The timing statistic retains the last five windows. For each retained Graal window both the lookup driver and the region must begin with Tier 2 code installed. The trace must report zero compilation, invalidation, deoptimization or failure events within the window. We summarize each process by the median of the retained windows and summarize a profile by the median across process medians. The process is the unit of replication [3]. Wall time and calling-thread CPU time answer separate questions. We also record process elapsed time including JVM and application startup, compilation, warm-up, and all timed windows.

Table 2: Six randomized blocks of the recursive mutex-pair probe. Process elapsed time includes setup and compilation.

Profile	CPU (ns/pair)	Wall (ns/pair)	Process (s)
Native C	7.73	7.72	0.009
JVM, libffi	147.37	221.21	3.97
JVM, Panama	35.92	68.13	4.34
JVM, typed JNI	18.23	17.86	4.04

Table 3: Tier 2 `sqlite3_mutex_enter` compilation. Sites are final-graph occurrences of `new_array_or_null`, a marker for array allocation.

Measure	libffi	Typed JNI
Installed code bytes	1,229	722
Final Graal IR nodes	378	266
Array allocation sites	3	0

Mutex-pair probe. A separate loop repeatedly locks and unlocks one real recursive pthread mutex and checks both return values. It measures windows of 100,000 pairs. Table 1 shows a 7.82× improvement for the typed bridge relative to the generic NFI in this probe. Both timings include the loop and return-value checks. The probe supplies a scale for the individual crossing with no SQLite page or VM dispatch in the measured loop.

Mutex-probe replication. Six blocks repeat the probe with one fresh process per arm and a seeded random profile order within each block. The JVM arms share the same guest bitcode. Runs are pinned to the same logical CPU, and each retained window starts with the probe at Tier 2 and records no compiler events. Table 2 shows median calling-thread CPU, elapsed time over the retained windows, and whole-process time. The CPU interval includes the adjacent wall clock reads. All six paired CPU comparisons order native C, typed JNI, Panama, and libffi from fastest to slowest. Comparing arm medians, Panama is 4.10× as fast as libffi and typed JNI is 1.97× as fast as Panama. The median within-block speedups are 4.09× and 1.85×, respectively. The individual process times include long whole-run observations of 74.16 s for Panama and 68.54 s for typed JNI. Those observations remain in the recorded results.

The median paired CPU-time ratio for typed JNI over Panama is 0.542, with an order-statistic interval of $[0.268, 0.954]$. For six independent block ratios from a continuous distribution, the interval from the smallest to the largest has $1 - 2(1/2)^6 = 96.875\%$ coverage for the population median. A separate six-block check compared the pristine libffi JAR to both overlays loaded with their optimizations disabled. The arm medians were 140.59 and 142.33 ns per pair, respectively. The paired ratio of disabled-overlay to pristine time had median 1.006 and interval $[0.817, 1.022]$, which includes parity.

Table 4: Six-block serialized lookup replication. Warm columns are per 150,000 lookups. Each cell is a median across independent process summaries for its arm.

Profile	Warm CPU (ms)	Warm wall (ms)	Process (s)
Native stock	46.69	46.88	0.489
Native matched LLVM	59.34	59.34	0.629
JVM, libffi	183.71	186.55	50.875
JVM, Panama	111.27	113.35	78.659
JVM, typed JNI	68.74	69.47	50.686

Generated code. Table 3 compares independently captured Tier 2 versions of `sqlite3_mutex_enter`. The generic graph calls the libffi primitive path. The typed version calls the JNI entry with the resolved symbol, mutex pointer, and guest error state. Disassembling the C bridge shows the thread-local `errno` write, an indirect call to the original pthread address, and the result packing. Together these traces establish that the changed code is used and identify the allocations the specialization removes. A diagnostic configuration collects graphs and bridge counts. Timing uses the uninstrumented library with graph dumping disabled.

Serialized lookups. The initial paired calling-thread CPU medians were 196.71 versus 67.46 ms and 186.49 versus 68.56 ms, giving a 64.5% reduction in the median across processes. The native control median was 46.41 ms. The whole-process medians for the two JVM arms were 50.35 and 46.66 s, respectively, and the native median was 0.471 s. Consequently the warm-loop improvement is much larger than the observed whole-process gain. Six times the measured 125.67 ns pair difference corresponds to about 754 ns per lookup, compared with about 824 ns saved per lookup in the larger benchmark. This calculation establishes a scale only. The probe and SQL workload compile different loops, and the two measurements cannot attribute the full observed change to a per-call constant.

Five-arm lookup replication. The replication uses six fresh-process blocks on CPU 6. It keeps the original benchmarking protocol and adds a native control compiled from the same LLVM module as the JVM arms, and the guarded official Panama binding. All three JVM profiles use the same bitcode and load the same overlays. Only the selection properties differ. Each block uses a seeded random order (seed 2410). JNI ran before libffi in five of the six blocks. All 30 processes supplied five valid retained windows, and the native result checks matched. Whole-process times include the untimed setup and warm-up as well as compilation and every timed window. The first cold samples remain in the artifact.

Table 4 gives arm medians. For each block we also compute the ratio of the process CPU medians, then report the median and the 96.875% order-statistic interval as above. JNI over libffi is 0.365, $[0.352, 0.407]$, and JNI over Panama is 0.618, $[0.469, 0.648]$. Panama over libffi is 0.595, $[0.544, 0.759]$. JNI used less CPU than both JVM alternatives in every block. JNI over stock native is 1.469, $[1.368, 1.563]$, while JNI over matched LLVM native is 1.162, $[0.984, 1.229]$. The intervals are conditional on the measured host and the independent-block model. The order imbalance and single host limit extrapolation.

Table 5: Median process elapsed seconds across three fresh processes per `speedtest1` workload and profile. JVM runs use background compilation.

Suite	Native `-O2`	libffi	Panama	JNI
`main`	0.020	17.11	18.11	18.42
`cte`	0.026	28.89	30.85	24.98
`orm`	0.009	6.29	7.24	6.19

Upstream workloads. The initial `speedtest1` probe used one process per profile and workload with background compilation. The original and typed adapters matched the native verification counts, hashes, and case ordering on `main`, `cte`, and `orm`. The process elapsed times for the generic and typed arms were 19.08 and 14.73 s on `main`, 26.37 and 27.95 s on `cte`, and 5.53 and 7.27 s on `orm`. These exploratory observations include startup and compilation and have mixed directions.

A confirmation panel ran `speedtest1` at `size 1` with `verify` and `memdb`, using a common guest module across all three JVM profiles. The native `-O0` and `-O2` controls and three JVM profiles each ran three fresh processes on each of the three workloads in rotating order on CPUs 4–5. The 45 executions completed and matched native verification. The timings in Table 5 include preparation, startup, and compilation. In this panel the typed JNI median is higher than libffi on `main`, lower on `cte`, and close on `orm`. The per-process ranges for JNI and libffi overlap on all three workloads. For example, the JNI range for `cte` is 24.52–32.46 s and the libffi range is 27.38–30.65 s. The host was shared with the CPU 6 lookup panel. A separate overlapped pilot was stopped before the planned number of trials and is preserved without pooling its measurements into the confirmation panel.

4 Related work

Native call backends. Truffle documents the libffi backend and a Panama backend based on the Java Foreign Function and Memory API, with different supported NFI types [4, 5]. The primary contrast is within the pinned libffi configuration. It isolates the impact of replacing the specific argument serializer and call path in this workload. A backend comparison requires an explicit binding path and comparable signature and error-state behavior. An ordinary libffi call, an official NFI Panama call, and an independent Java FFM wrapper have different execution paths.

Matched Panama binding. In the pinned Sulong source, ordinary loaded native functions use a prefix-free NFI signature and the default backend. The alternate-backend mechanism in that source is used for exported closures. We add a comparison overlay at the direct binding site. It checks the ordinary signature and equality of the resolved native address to one of the same two glibc pthread entry points. It then requests `with panama (POINTER):SINT32` using the existing signature cache and the unmodified published Panama backend. Other native bindings, indirect calls, and callbacks retain their original path. The pinned Panama executor copies guest `errno` into the libc thread-local value before its FFM call and copies it back afterward. The comparison passed the native differential contract with error-checking mutexes, guest error-state sentinels, and the alternate-provider and callback cases. This makes the guarded binding a matched third arm for this ABI rather than a global replacement of the native backend.

SQLite specialization. SQPyte showed that a tracing interpreter can reduce the overhead of crossing between SQLite and a host language [6]. Our boundary lies farther down the stack, at Sulong's call from the SQLite guest to the host pthread library. LLVMSQLite compiles prepared statements and the VDBE program to native code through MLIR and LLVM [7]. This is complementary to replacing an individual native-call adapter while holding the compiled guest fixed.

5 Scope and reproducibility

The tested bridge targets two glibc functions on one ABI. The evidence concerns the observed SQLite access pattern and steady-state execution. The first lookup panel has two independent process pairs. Its windows give within-process measurements, and its small number of processes limits what can be inferred about other runs and machines. The public-suite coverage is incomplete. The contract harness tests the adapter, while the upstream cases bound supported runtime claims.

Four-worker diagnostic. In a separate replay of the unmodified four-worker `threadtest5` harness, the native build returned `OK`. The pinned guest returned `incorrect result` on both the JVM and the shipped AOT host, under both interpretation and JIT. A traced AOT-interpreter run with a two-second busy timeout had 27 missing rows after all 100 task claims committed. Six missing rows matched individual failed inserts that returned `SQLITE_BUSY`. The remaining 21 belonged to a task whose worker had already committed its claim and then exited after a busy schema-check prepare. At 30 seconds the traced AOT interpreter, AOT JIT and JVM interpreter each produced both expected 1,230-row tables with no reported SQL error. The traced JVM JIT reached the 90-second outer deadline with either busy timeout.

A deterministic two-connection test held a write lock after committing a task claim. Native C, the AOT interpreter and the JVM JIT each observed `SQLITE_BUSY` on the work insert, leaving one committed claim and zero inserted rows. Each inserted the row on an explicit retry. These observations explain the row deficit in the traced run through the harness's busy handling. The timeout change is a diagnostic intervention. Concurrent coverage and other timeout regimes remain open.

Earlier experiments used a standalone Graal AOT ELF with an embedded guest stack transformation and checked static branch replacements. Its resource and branch checks concern the binary's recorded byte ranges. The typed JNI speedup comes from the JVM experiments. The matched Native Image control build for the bridge stopped during static analysis under memory pressure. A separate coalescing panel found a 5.1% median change with two improvements in three pairs. The ELF branch-threading trials and broader workload timings gave mixed directions.

A follow-up stock rebuild with one compiler worker and a 2,200 MiB build heap was stopped at the 180-second deadline during static analysis. Its observed RSS maximum was approximately 2.36 GiB. This bounded timeout is distinct from the earlier memory-exhaustion exit.

A standalone Native Image smoke test compiled the production `MutexDowncall` class and pinned `NFIState` against the same JNI bridge. The quick-build configuration used two workers and a 512 MiB build heap. The command took 63.59 s with maximum child RSS of 872,340 KiB. Its output matched the JVM fixture for a real glibc `EDEADLK` result, error-state mutation and preservation, negative return packing, and the symbol guard. This small executable excludes Sulong and SQLite. The full SQLite performance results for the bridge come from the JVM.

Artifact availability. The accompanying reproduction files provide pinned runtime instructions, exact inputs and hashes for the JVM panels, per-process results, Tier 2 traces, contract logs, generated code, and replay scripts. Start with `REPRODUCE.md` for commands and dependencies. The supplement provides recorded evidence for the earlier whole-program ELF experiments. Reproducing those binary transformations requires the earlier prebuilt AOT distribution.

Agent assistance. A coding agent assisted with implementation, benchmark execution, diagnosis, analysis, and manuscript drafting.

6 Conclusion

Native-call marshalling can dominate short, frequent guest operations even when the underlying library executes native code. On the pinned Sulong JVM stack, a guarded path for pthread mutex calls removes three allocation sites while preserving the tested downcall contract. The measured reduction in warmed SQLite lookup cost identifies one source of the gap to compiled C. The startup results, mixed upstream timings, and unresolved compatibility cases bound that finding.

References

[1] GraalVM. LLVM Runtime. GraalVM documentation.

[2] SQLite Project. Using SQLite in multi-threaded applications. SQLite documentation.

[3] Tomas Kalibera and Richard E. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management*, pages 63–74, 2013. doi:10.1145/2464157.2464160.

[4] GraalVM. Truffle native function interface. GraalVM documentation.

[5] Oracle. Foreign function and memory API. Java SE 24 documentation. Java Core Libraries.

[6] Carl Friedrich Bolz, Darya Kurilova, and Laurence Tratt. Making an embedded DBMS JIT-friendly. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *LIPIcs*, pages 4:1–4:24, 2016. doi:10.4230/LIPIcs.ECOOP.2016.4.

[7] Thomas Kowalski. Compiling SQLite queries to native code with LLVM. Master's thesis, Imperial College London, 2020. Author's manuscript.