X: A Programming Language That Makes Optimisation Explicit

William Kirkman-Moller

https://github.com/x-lang-project/x-lang

Abstract

X lets programmers request result reuse, parallel execution and columnar storage through `memoise`, `parallel` and `layout`. Its compiler preserves record invariants across layout changes and accounts for the shared cache state introduced into pure functions. Memoised functions may update local variables and execute on parallel workers; the compiler tracks shared cache access separately from those local effects. The compiler is written in Rust and targets LLVM; companion Lean models describe cache behaviour and record updates. Experiments compare the transformations separately and in combination, using equivalent C implementations as baselines. Memoisation and parallelism speed up binomial option valuation by a median factor of 5.20 over serial, uncached C compiled with Clang at O3 on a batch of 4,096 positions with repeated inputs. Columnar storage speeds up serial payoff queries by a median factor of 4.38 at about four million positions. CRC-32 validation, which LLVM already makes cheap, slows down with caching and parallel dispatch.

1 Introduction

A programmer may know that a function repeatedly receives the same inputs, that independent checks can run concurrently, or that a loop mainly reads one field of each record. Acting on this knowledge can require a cache, a worker loop or a different representation of the data. X expresses these choices through declarations that leave the computation intact. The programmer requests result reuse with `memoise`, concurrent execution with `parallel`, and record or columnar storage with `layout`.

Combining these choices must preserve the program's effect restrictions, ownership rules and invariant checks. A function whose result depends only on its inputs can acquire a shared mutable cache. Splitting records into separate field arrays must preserve which record each field belongs to and when its invariant is checked. Parallel workers must finish before the data they read is released. X checks effects, ownership and calls, then carries the information needed to enforce these requirements into generated code. A function may update its own temporary variables while remaining suitable for memoisation; generated cache access is tracked separately from those source-level effects.

We evaluate X on CRC-32 validation, binomial option valuation and portfolio payoff queries, with matched C implementations for each. Valuation can reuse expensive results and distribute independent work; payoff queries scan one column of a large portfolio; and CRC validation leaves little work after LLVM optimisation.

Halide separates algorithms from schedules and supports parallelism, storage order and memoisation [1, 6]; Terra generates specialised code through staging [2]. X applies these choices to imperative programs with local mutation, borrowed fields and record invariants.

2 Language design

The portfolio application in Section 7 values each position with a 64-step binomial calculation. Repeated spot prices can reuse the same answer, and positions can be checked independently. The valuation function is declared `memoise pure fn price(spot: i64) -> i64`; its body updates a local array while computing the price. Listing 1 shows how the declarations control storage and execution in this application.

```
#[layout(AoS)]
struct Position {
    spot: i64, expected: i64, sequence: i64,
    m0: i64, m1: i64, m2: i64, m3: i64, m4: i64,
    invariant { self.spot >= 0 && self.expected >= 0
                && self.sequence >= 0 }
}

pure fn check(index: i32, positions: &[Position; 4096]) -> i32 {
    let result: i64 = price(positions[index].spot);
    invariant { result == positions[index].expected }
    invariant { positions[index].sequence == (index as i64) }
    return 0;
}

fn validate(positions: &[Position; 4096], count: i32) -> i32 {
    parallel for row in 0..count {
        check(row, positions);
    }
    return 0;
}
```

Listing 1: Valuation application excerpt. The body of `price` and input loading are omitted.

Removing `memoise` from `price` or `parallel` from the batch loop preserves the valuation and worker bodies. The compiler supplies the cache and dispatch. Changing `AoS` to `SoA` changes how the same record fields are stored. In the measured valuation workload, memoisation and parallel execution improve performance, while columnar storage gives no consistent gain. The payoff queries in Section 7.3 read just the spot-price column of these records and benefit from the layout change.

Table 1 describes X's ten semantic constructs. We study memoisation, layout and parallel execution, including the purity, ownership and invariant checks needed to combine them.

Construct	Role in the language
`pure`	Constrain effects while permitting local mutation.
`memoise`	Generate a cache for a pure function.
`static`	Specialise a function for compile-time arguments.
`become`	Require a direct tail transfer with a matching ABI.
`multi`	Resolve overloaded calls from complete parameter types.
`throws`	Declare an error channel that callers must handle.
`unique`	Enforce exclusive access through a reference.
`invariant`	Check predicates at statements, construction, and mutation.
`layout`	Select record or columnar storage for record arrays.
`parallel`	Dispatch range iterations to pure, nonthrowing workers.

Table 1: X's ten semantic constructs.

2.1 Purity and result reuse

Listing 2 computes the sum of squares below a bound using a loop and a local accumulator.

```
memoise pure fn sum_squares(limit: i32) -> i32 {
    let mut i: i32 = 0;
    let mut total: i32 = 0;
    while i < limit {
        total = total + i * i;
        i = i + 1;
    }
    return total;
}
```

Listing 2: A memoised sum of squares.

The `pure` declaration permits local mutation and rejects I/O, mutable global reads, and writes to caller-owned storage. Both `i` and `total` belong to the invocation, so their updates preserve purity. Calls from a pure body must also be pure.

The `memoise` declaration adds a cache of argument–result pairs. Two consecutive calls to `sum_squares(8)` produce 140. After the first computation, the second can reuse its result. Removing `memoise` leaves the function body and its callers intact.

Memoisation requires scalar arguments and results and a pure, nonthrowing body. Each memoised function has a cache with a default capacity of 64 entries. A hash selects a slot, and a bitwise comparison of every encoded argument determines whether the stored key matches. A collision produces a miss; after evaluation, the new key and result replace the entry. Floating-point arguments retain their bit representation, so positive and negative zero have distinct keys.

2.2 Access and representation

The `unique` qualifier declares exclusive access through a reference. A parameter declared `unique value: &mut i32` receives a borrow written `&unique mut value` at the call site. The borrow checker prevents competing access during the borrow and treats the unique reference as move-only. A function accepting this mutable reference cannot be declared `pure`, since the parameter permits mutation of caller-owned storage.

The annotation `#[layout(SoA)]` selects columnar storage for fixed arrays of a record type. An array of structures (AoS) groups each element's fields together. A structure of arrays (SoA) groups the values of each field. For records with fields `low` and `high`, the representations are

$$\begin{aligned} \text{AoS} &: [(0,2),\ (20,30)], \\ \text{SoA} &: (\text{low} = [0,20],\ \text{high} = [2,30]). \end{aligned}$$

The expression `ranges[i].low` accesses the same logical field under either layout. Under SoA it indexes the lower-bound array. Reading a whole element gathers its fields; replacing one snapshots the new record and writes its fields to the separate arrays.

Listing 3 combines columnar storage with an interval invariant. The compiler inserts checks after construction and relevant mutations under either layout.

The replacement computes both new bounds from the existing interval, stores $(10, 12)$ across the two field arrays, and then checks the invariant. Separate assignments in the order `low = low + 10`, then `high = high + 10`, would first produce $(10, 2)$ and fail at that assignment.

Reading `ranges[0]` into `saved` gathers an independent record value. The second array element remains $(20, 30)$. Changing the annotation to `layout(AoS)` gives the same values and invariant checks with record storage.

```
#[layout(SoA)]
struct Interval {
    low: i32,
    high: i32,
    invariant { self.low <= self.high }
}

fn main() -> i32 {
    let mut ranges: [Interval; 2] = [
        Interval { low: 0, high: 2 },
        Interval { low: 20, high: 30 }
    ];
    ranges[0] = Interval {
        low: ranges[0].low + 10,
        high: ranges[0].high + 10
    };
    invariant { ranges[1].low == 20 && ranges[1].high == 30 }
    let saved: Interval = ranges[0];
    invariant { saved.low == 10 && saved.high == 12 }
    return 0;
}
```

Listing 3: Translating a complete interval in columnar storage.

2.3 Parallel execution

A `parallel for` loop executes range iterations on pure, nonthrowing workers and waits for completion. Each worker receives an `i32` index, and the loop discards its return value. Bounds and shareable capture expressions are evaluated once before dispatch. Captured values are placed in an environment passed to each worker.

For example, a pure worker can square an index and check the result against a local table. A loop over `0..8` invokes that worker at indices 0 through 7 in an unspecified order. The checks proceed independently because the worker cannot write shared state.

A worker may also call a memoised function. Its source body remains pure, but the generated code shares a mutable cache with other workers. The compiler records these cache accesses separately from source effects and propagates them through the call graph. A pure wrapper whose callee uses a cache therefore cannot receive LLVM `memory(none)`.

2.4 Supporting call mechanisms

A `static` parameter supplies a compile-time literal. The compiler substitutes it into a shared specialised body and removes it from the runtime signature. A `become` statement requires a direct tail transfer with a matching runtime ABI and emits LLVM `musttail`. Borrow analysis rejects arguments that refer into the discarded caller frame. In a mutual-recursion test, specialising a three-parameter function for one static argument gives the same two-parameter runtime signature as its caller; 100,000 transfers return 700,000 at O0 and O3 without accumulating frames.

Rust also provides explicit tail calls through its experimental `become` construct [7]. X uses the same syntax and checks that specialisation preserves the required runtime signature and reference lifetimes.

Calls to `multi` functions resolve to a unique implementation from complete argument types. A `throws` function exposes a tagged error channel that its callers handle or propagate. Before lowering the call, the compiler checks the resolved callee's effects and return convention.

3 Compiler implementation

The compiler is written in Rust and generates native code through LLVM. Type checking resolves calls and specialisations, and borrow analysis tracks moves, loans, and reference origins.

The typed intermediate representation records resolved callees and storage projections. Projections retain their enclosing records, allowing code generation to insert the applicable invariant checks after writes through nested fields. Tail transfers retain the checked call information needed to emit `musttail`.

For each memoised function, the compiler records the cache capacity, key encoding, indexing rule, and synchronisation requirements. Each generated cache contains validity flags, keys, results and a lock. Lookup and replacement hold the lock; on a miss, evaluation proceeds outside it before the result is stored under the lock. Recursive calls can use the same cache, and simultaneous misses may compute the same result before either stores it.

LLVM memory attributes are assigned after the generated cache effects have been propagated through calls. Unique-reference parameters receive `noalias` only when the compiler checks exclude raw-pointer paths that could alias the referent.

Parallel ranges use a linked OpenMP runtime. Lowering evaluates the bounds and captures into temporaries before dispatch. For captured workers, code generation builds an environment and a wrapper that loads its fields before calling the worker. The dispatch call completes before that environment goes out of scope.

We ran the sum-of-squares, interval, parallel-check and tail-transfer examples on Windows x86-64 at O0 and O3. The generated O0 code contains a 64-entry cache for `sum_squares` and `musttail` calls in the specialised mutual-recursion test. The interval program produces $(10, 12)$ and $(20, 30)$ under both AoS and SoA, and the parallel program completes all eight checks. The compiler rejects the mutable-reference parameter on a pure increment function and the caller-local reference in a tail transfer.

3.1 Semantic models

For a total pure function f and cache C, the cache invariant is $C(i) = \text{some}(k, v) \implies v = f(k)$ for every occupied slot i. A hit compares the complete key; a miss computes $f(k)$ and may publish it after another worker has replaced the slot. The companion Lean model `ConcurrentMemoise` treats lookup and publication as atomic operations, with pure computation between them. For every admitted schedule, `every_completed_call_correct` proves that each response equals $f(k)$ for its original input, and `completed_cache_valid` proves that the cache invariant is preserved. The key must contain every observable input. Simultaneous misses can repeat work without changing the answers.

Parallel reads also depend on storage lifetime and record ownership. Read-only parallel iteration evaluates bounds and captures once and keeps captured storage live until the workers join. Layout translation retains each projection's logical record owner alongside its physical field address, so a write checks the same invariant at the same mutation boundary. Workers can therefore share the generated cache while reading the source records under their original access restrictions.

The `WideCheckedLayout` development covers records with any finite number of fields and any Boolean record predicate. Its theorem `mutation_handoff_calls_correct` connects mutation and layout translation to memoised jobs. Once a successful mutation finishes and releases its loan, gathering the column store supplies the same job keys as the logical record store. Every completed response then equals the uncached result, and the cache remains valid under the request and publication schedules of `ConcurrentMemoise`.

4 Composing layout, memoisation and parallelism

To test the transformations together, we compile programs that update records and then share a read-only view with workers. Each program constructs 129 records containing a key, three expected values and a padding field; its invariant requires nonnegative keys and expected values. Runtime input selects one of three seeds. After updating the keys, each worker evaluates an integer recurrence for 32 steps and compares its result with a value computed in Python. A second check verifies the record's logical row through its padding field.

We vary AoS or SoA storage, ordinary or memoised computation, and serial or parallel iteration at O0 and O3. Two input distributions exercise reuse and replacement: one repeats eight keys, while the other uses 129 distinct keys, necessarily placing distinct keys in a shared slot of the 64-entry cache. Four passes reuse the same cache within each process. Removing memoisation, switching layout or removing `parallel` leaves the worker body unchanged.

The resulting 32 binaries run with each input seed. Parallel variants request one or four OpenMP threads; serial variants use one. Successful executions complete every check and print a marker. To test failure handling, we corrupt one expected result in the first, middle or last row. Those executions must stop with an illegal-instruction trap, ensuring that the checks remain active even though the parallel loop discards its workers' return values.

All 576 executions had the expected outcome, with 144 successful runs and 432 traps at the corrupted check. OpenMP affinity reports confirmed four distinct worker threads in all 48 successful runs that requested four threads. Inspection of the generated modules confirmed parallel dispatch, invariant traps and memo-cache locking.

5 Comparative performance

Table 2 summarises the application results and their baselines. Memoisation and parallelism help when repeated calculations are expensive; columnar storage helps when a scan reads one field of a large record array. CRC validation provides a case where the generated cache and dispatch cost more than the calculation they replace.

Choice	Workload	Baseline	Result
Memoisation and four workers	Option valuation: 4,096 positions, eight repeated inputs	Serial, uncached C at O3	5.20× faster
Same transformation choices	The same valuation batch	Handwritten C cache and parallel loop	X: 5.588 ms C: 4.066 ms
Columnar storage	Payoff queries: 4,194,304 positions, 64 strikes	Serial X with record storage	4.38× faster
Memoisation	CRC-32 validation across both layouts and batch sizes	Serial, uncached X	1.40–2.65× slower
Four workers	The same CRC-32 workloads, without memoisation	Serial, uncached X	3.35–5.67× slower

Table 2: Application results. Factors are medians of paired runtime ratios; times are median batch latencies. The two valuation rows distinguish the gain over serial C from the comparison with C implementing the same transformations.

With the same cache policy and parallel loop, handwritten C has lower latency than X in this configuration (Table 2).

We first isolate memoisation and layout, changing only `memoise` or the AoS/SoA annotation while retaining the arithmetic and invariants. Sections 6 and 7 then examine the application workloads.

5.1 Workloads and measurement

The memoisation workloads repeatedly apply an integer recurrence. Each call starts from its key and performs either 256 steps (expensive) or one step (cheap), using $v' = (1664525v + 1013904223) \bmod 1000003$. The expensive reuse workload makes two million calls cycling through eight keys. A second makes 200,000 calls cycling through 4,096 keys, exceeding the generated 64-entry cache. The cheap workload makes twenty million calls through eight keys. The function uses an invocation-local accumulator in every variant.

The layout workloads contain 1,024 records of eight i64 fields and perform 65,536 passes. One increments a single field; the other replaces each complete record with its eight fields incremented. Both retain the record invariant $0 \leq f_0 \leq 10^9$. Initialisation and final checksum collection are included in the measured region. The array occupies 64 KiB of field data, while its selected column occupies 8 KiB.

We ran the benchmarks on Windows 11 with an AMD Ryzen 9 9950X3D and LLVM/Clang 22.1.8. X emitted object files at O0 and O3, which Clang linked with a timing helper. Windows performance counters measured execution inside the process, excluding startup and compilation. An external call supplies the input seed after timing starts, and the stop function consumes the final checksum, preventing the compiler from eliminating the computation.

Each of the 20 binaries ran in two warm-up rounds and nine measured rounds. We shuffled variant order with a fixed random seed and used the same input seed for each paired comparison. Every process started with a fresh cache. All 220 executions agreed with the Python reference calculation. Table 3 reports median O3 times and median paired runtime ratios at both optimisation levels; ratios above one favour the transformation. These sequential microbenchmarks retain the synchronisation in the generated cache.

Workload	Baseline ms	Selected ms	S_{O0}	S_{O3}
Expensive, 8 keys	1508.95	8.75	151.485	171.227
Expensive, 4096 keys	150.98	151.85	0.997	0.998
Cheap, 8 keys	16.10	87.10	0.495	0.185
Single-field updates	18.03	13.00	1.250	1.377
Whole-record updates	43.93	76.82	0.630	0.572

Table 3: Median O3 execution times and paired baseline/selected ratios. Baselines are plain functions for memoisation and AoS for layout; selected variants use memoisation or SoA. Each row compares equal workloads.

5.2 Benefits and costs

For expensive repeated calls, memoisation gives a median O3 paired speedup of 171.23, with measured pairs ranging from 169.03 to 175.41. Cycling through 4,096 keys gives a ratio near one, with pairs ranging from 0.978 to 1.007. The cheap recurrence instead runs about 5.41 times slower with memoisation; avoiding its arithmetic does not recover the cache's lookup and synchronisation cost.

SoA improves the single-field workload by a median factor of 1.38 at O3, with paired ratios from 1.372 to 1.401. Whole-record updates are about 1.75 times slower under SoA, with ratios from 0.486 to 0.594. The timings reflect both memory-access and instruction-selection effects.

6 CRC-32 validation

The first application kernel validates synthetic batches of records using CRC-32. Each record contains an identifier occupying eight bytes, an expected digest, a sequence number and five metadata fields. The worker calculates the identifier's CRC-32 and checks both the digest and logical row. We use the polynomial and byte order of the gzip checksum [9], with expected values computed by Python's `zlib.crc32`.

6.1 Matched implementations and runtime inputs

Both X and C use unsigned 64-bit identifiers and unsigned 32-bit CRC state, return values and cached results. They start with the same bitwise algorithm over eight little-endian bytes. The C implementation supplies all eight combinations of layout, memoisation and parallel execution manually. Its memoiser uses the same cache capacity of 64 slots, full-key comparison, hash finalizer and acquire/release byte lock as X, and evaluates misses outside the lock. Both implementations are compiled at O3.

Runtime files supply identifiers and expected digests from three datasets. Each batch contains 256 or 4,096 records, with eight repeating identifiers or distinct identifiers. A run performs 262,144 logical record checks, with cache hits reusing the digest. A different XOR mask changes the identifiers in each batch. The input generator adjusts expected checksums using CRC's affine relation and checks every masked identifier with zlib. Both programs obtain the next mask and checksum correction through external calls. These calls remain inside the outer loop after optimisation, so the compiler cannot replace the repeated batches with a single validation.

The timed interval includes mask acquisition, validation, cache operations and dispatch; file loading and record construction precede it. We shuffle variant order reproducibly within each round and use the same input seed across variants. Each of the 64 combinations of language, transformation choices and workload has one warm-up and seven measured samples. All 512 runs validate their inputs. Corrupting one digest in the first, middle or last row causes the expected illegal-instruction trap in all 48 controls. Separate affinity runs confirm four worker threads for each parallel binary; affinity logging is disabled during timing.

	256 records/batch		4,096 records/batch	
Choice	8 repeated	Distinct	8 repeated	Distinct
AoS, plain, serial	0.899 / 0.896	0.906 / 0.903	0.899 / 0.899	0.901 / 0.904
AoS, plain, parallel	3.561 / 5.215	4.132 / 3.869	3.131 / 2.845	2.977 / 3.494
AoS, memo, serial	1.308 / 1.302	2.396 / 2.312	1.268 / 1.255	2.377 / 2.285
AoS, memo, parallel	14.646 / 14.077	24.233 / 25.616	12.376 / 14.139	23.317 / 26.993
SoA, plain, serial	0.901 / 0.903	0.889 / 0.907	0.895 / 0.900	0.900 / 0.894
SoA, plain, parallel	4.492 / 4.486	4.188 / 3.678	4.081 / 4.524	5.117 / 2.836
SoA, memo, serial	1.292 / 1.299	2.248 / 2.317	1.264 / 1.278	2.261 / 2.318
SoA, memo, parallel	13.961 / 13.265	24.021 / 27.136	13.775 / 12.854	23.974 / 25.737

Table 4: Median validation times in milliseconds, shown as X / C for each matched choice. Every run schedules 262,144 logical checks; parallel variants use four threads. Seven measured samples contribute to each median.

6.2 Results and interpretation

For ordinary serial validation, median paired X/C runtime ratios range from 0.987 to 1.010 across the two layouts and four workloads. LLVM replaces the bit-at-a-time CRC loop with a lookup table in both languages, making the compiled calculation much cheaper than the source loop suggests.

Memoisation makes X's serial variant 1.40–2.65 times slower. Parallel execution without memoisation is 3.35–5.67 times slower than the corresponding serial variant. The combined variants in Table 4 are slower still: cache sharing and dispatch cost more than the CRC calculation that remains after LLVM optimisation.

7 Binomial valuation over portfolio positions

The second application kernel prices European call options with a binomial model [10]. It uses synthetic portfolio positions, 64 time steps and integer cents. For terminal index j, the algorithm starts from the input spot price, applies $s \leftarrow \lfloor 102s/100 \rfloor$ j times and then $s \leftarrow \lfloor 98s/100 \rfloor$ $64 - j$ times, and stores $\max(s - 10000, 0)$. It repeatedly replaces adjacent terminal values with their integer average until one value remains. These fixed parameters and rounding rules define the benchmark calculation.

7.1 Implementations and experiment

X and C use the same dynamic program, updating an array of 65 values in place. Inputs, arithmetic, results and cached values use signed 64-bit integers, and all tested intermediates fit in that range. A Python implementation computes reference answers by recursively interpreting the tree. Each record contains the spot price, expected value, sequence number and five metadata fields. The worker calculates the value and checks both the result and logical row. All eight X variants share the same valuation and worker bodies, changing only the layout annotation and `memoise`/`parallel` choices. C supplies the corresponding columns, cache and OpenMP loop manually, using the same cache policy and synchronisation as in the CRC comparison.

Batches contain 256 or 4,096 positions, with eight repeated spot prices or distinct prices. Three seeds determine the prices. The datasets with repeated prices occupy seven, seven and eight distinct cache slots, respectively, so two include conflict misses. Each process evaluates its batch once with a fresh cache. Timing starts after input loading and record construction, and includes validation, cache operations, dispatch and initial OpenMP team startup. Each of the 64 combinations of language, transformation choices and workload runs in one warm-up process and nine measured processes, with three measurements per input seed and a reproducibly shuffled order.

All 640 timed executions produce the expected values and number of checks. All 48 tests with a corrupted expected value trap, and separate affinity runs confirm four worker threads in each parallel binary. The compiler, host and timer are the same as in the CRC-32 study. Table 5 reports every combination.

7.2 Results

Across the eight ordinary serial comparisons, median paired C/X runtime ratios range from 0.995 to 1.010. For 4,096 positions with eight repeating inputs, memoised parallel X is 5.20 times faster than serial, uncached C at O3. The C baseline in this comparison uses neither caching nor parallelism. Comparing transformations within X, its AoS variant is 2.11 times faster with parallelism, 3.97 times faster with memoisation and 5.28 times faster with both,

Choice	256 positions		4,096 positions	
	8 repeated	Distinct	8 repeated	Distinct
AoS, plain, serial	1.818 / 1.835	1.821 / 1.837	29.098 / 29.059	29.139 / 29.090
AoS, plain, parallel	3.137 / 3.907	4.047 / 7.765	13.766 / 10.822	10.642 / 11.445
AoS, memo, serial	0.498 / 0.495	1.839 / 1.835	7.319 / 7.363	29.124 / 29.132
AoS, memo, parallel	5.217 / 4.156	4.398 / 4.124	5.588 / 4.066	10.794 / 10.167
SoA, plain, serial	1.822 / 1.829	1.828 / 1.836	29.123 / 29.114	29.242 / 29.135
SoA, plain, parallel	5.969 / 4.230	2.636 / 4.797	10.121 / 10.908	12.206 / 11.453
SoA, memo, serial	0.500 / 0.496	1.835 / 1.825	7.282 / 7.289	29.073 / 29.033
SoA, memo, parallel	2.964 / 3.630	4.079 / 3.014	6.461 / 4.343	11.102 / 11.195

Table 5: Median binomial-valuation batch latency in milliseconds, shown as X / C. Each median contains nine samples across three seeds. Parallel variants request four threads and include initial dispatch costs.

using median paired ratios against serial, uncached X. The combined ratios range from 3.53 to 13.03. The two seeds with cache collisions have serial memoised medians of 7.45 ms and 7.37 ms, compared with 0.075 ms for the seed without collisions.

With 4,096 distinct inputs, AoS parallelism gives a median paired speedup of 2.75, while serial memoisation has a ratio of 0.998 because there is no reuse. For 256 positions, every parallel variant has a higher median latency than its serial counterpart. These small batches do too little work to offset team startup and scheduling. SoA gives no consistent benefit for this computation, whose work is dominated by arithmetic. Payoff queries over the same records read one field and perform little arithmetic per position.

7.3 Payoff queries over the portfolio

The same position records also support aggregate exercise-payoff queries. For a strike k, a query computes $P(k) = \sum_i \max(s_i - k, 0)$ from the spot prices s_i. A batch evaluates 64 different strikes. Each query reads one field per position, so the cost of accessing records becomes more prominent than in binomial valuation.

We retain the eight-field record and the preceding study's input datasets. Larger portfolios repeat the instruments while assigning distinct position identifiers. Five sizes range from 4,096 to 4,194,304 positions. For input seed s and query index q, the strike is $9800 + 17s + 50q$ cents.

X and C use identical scans and integer arithmetic. Parallel variants distribute queries across four workers. Within X, each AoS/SoA pair differs only in the record's layout annotation. Record construction and input checks precede timing. The measured interval includes all queries, their result checks and parallel dispatch, including the first worker-team startup.

We use the same host and toolchain as the valuation experiment. Each configuration runs in one warm-up process and nine measured processes, with three observations per input seed and shuffled execution order. Expected answers come from a Python histogram calculation and are independently recomputed using sorted-value suffix sums. All query results agree; corrupting the expected answer for the first, middle or final query causes the corresponding failure in every variant. Table 6 shows the diverse-input distribution, whose original dataset has 4,096 spot prices.

At the largest size, SoA improves serial execution by a median factor of 4.38, with paired ratios from 4.31 to 4.50. Holding four threads fixed, changing layout gives a median factor of 4.61; these ratios range from 2.90 to 4.95. SoA with four threads is 13.50 times faster than serial AoS under the paired comparison. The dataset with eight repeated inputs gives similar layout gains of 4.38 serially and 4.34 with four threads. Matched C implementations give serial

	Serial			Four threads		
Positions	AoS ms	SoA ms	Speedup	AoS ms	SoA ms	Speedup
4,096	0.106	0.077	1.35	3.883	3.314	1.20
65,536	1.986	1.236	1.61	4.623	5.122	0.95
262,144	8.567	4.939	1.74	7.689	4.896	1.57
1,048,576	73.285	19.716	3.71	31.503	9.135	3.67
4,194,304	349.780	80.266	4.38	120.446	26.102	4.61

Table 6: X payoff-query batch times and paired AoS/SoA speedups for diverse inputs. Each batch evaluates 64 strikes. Times are medians; speedups are medians of paired ratios.

gains of 4.36–4.38 and parallel gains of 4.58–4.59 across the two distributions. Small query batches remain sensitive to worker startup, as the first rows of the table show.

The records occupy 256 MiB at the largest size, while the selected spot column occupies 32 MiB. In the optimised serial LLVM code, AoS accesses successive spots at 64-byte intervals using scalar loads; SoA uses contiguous two-lane integer vector loads. The layout change gives the compiler contiguous accesses that it can vectorise. Valuation gains from reuse and parallel work, while the payoff queries benefit from contiguous access to the spot prices.

8 Related work

Halide separates image-processing algorithms from schedules controlling parallelism, locality and recomputation [1]. Its API provides `parallel`, `memoize` and `reorder_storage`, including planar and interleaved storage [6]. X integrates these choices with imperative record mutation. A borrowed projection retains its record owner after a layout change, predicates are checked at mutation boundaries, and effect analysis accounts for the cache state introduced into source-pure functions.

Terra combines Lua metaprogramming with a low-level language for generating specialised code, including containers whose layout can change between AoS and SoA [2]. X exposes layout as a record annotation and preserves invariant checks through that change. Its static parameters direct specialisation, with runtime-signature checks for tail transfers. Rust's experimental `become` provides the tail-call syntax and documents its ABI and lifetime requirements [7].

Koka describes effects through row-polymorphic types and supports encapsulated local state [3]. X uses declared purity that permits local mutation, with a separate analysis of physical memory effects introduced by generated caches. Cyclone combines static analysis and runtime checks for memory representation [4]; X's checked record updates additionally retain the logical owner when storage is split into columns. The record predicates follow Design by Contract [5]. The companion X Proofs development gives executable and mechanised accounts of cache behaviour and owner-preserving layout translation [8].

Exo 2 lets users build scheduling libraries from trusted transformation primitives, with actions, inspection and references supporting reusable schedules [11]. X offers declarations for a fixed set of transformations and concentrates on their interaction with source effects, borrowed record fields and invariant checks.

9 Conclusion

X expresses result reuse, parallel execution and record layout as source declarations. The compiler carries record ownership through layout changes and tracks the effects of generated caches, allowing these choices to interact with checked mutation and pure calls. Companion Lean models establish the corresponding properties of record updates and cache behaviour.

Memoised parallel X is a median 5.20× faster than serial, uncached C at O3 for option valuation on 4,096 positions with repeated inputs. SoA gives a median 4.38× serial gain for payoff queries at about four million positions. Caching and dispatch instead slow CRC validation after LLVM has made its calculation cheap. X makes these choices available without requiring separate implementations of the computation.

References

[1] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, 2013.

[2] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. In *PLDI*, pages 105–116, 2013.

[3] Daan Leijen. Koka: Programming with row polymorphic effect types. *EPTCS*, 153:100–126, 2014.

[4] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, 2002.

[5] Bertrand Meyer. Applying "Design by Contract." *Computer*, 25(10):40–51, 1992.

[6] Halide contributors. Halide Func API: scheduling, memoisation and storage order. Online documentation, accessed 26 October 2026.

[7] The Rust Project. The `become` keyword and `explicit_tail_calls`. Online documentation, accessed 26 October 2026.

[8] William Kirkman-Moller. X Proofs: Checked Layouts and Memoised Parallel Calls. Companion mechanised development, 2026.

[9] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, 1996. Section 8 gives the CRC calculation.

[10] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, 1979.

[11] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. Exo 2: Growing a Scheduling Language. In *ASPLOS*, 2025.