Interleaving with Coroutines: A Practical Approach for Robust Index Joins

Georgios Psaropoulos[⋆†] Thomas Legler[†] Norman May[†] Anastasia Ailamaki[⋆‡]

[⋆]EPFL, Lausanne, Switzerland
{first-name.last-name}@epfl.ch

[†]SAP SE, Walldorf, Germany
{first-name.last-name}@sap.com

[‡]RAW Labs SA

ABSTRACT

Index join performance is determined by the efficiency of the lookup operation on the involved index. Although database indexes are highly optimized to leverage processor caches, main memory accesses inevitably increase lookup runtime when the index outsizes the last-level cache; hence, index join performance drops. Still, robust index join performance becomes possible with *instruction stream interleaving*: given a group of lookups, we can hide cache misses in one lookup with instructions from other lookups by switching among their respective instruction streams upon a cache miss.

In this paper, we propose interleaving with coroutines for any type of index join. We showcase our proposal on SAP HANA by implementing binary search and CSB$^+$-tree traversal for an instance of index join related to dictionary compression. Coroutine implementations not only perform similarly to prior interleaving techniques, but also resemble the original code closely, while supporting both interleaved and non-interleaved execution. Thus, we claim that coroutines make interleaving practical for use in real DBMS codebases.

PVLDB Reference Format:
Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB*, 11(2): xxxx-yyyy, 2017.
DOI: https://doi.org/10.14778/3149193.3149202

1. INTRODUCTION

When choosing the physical operator for an equi-join between two relations, A and B, a query optimizer checks if either has an index on the join attribute. Such an indexed relation, e.g., A, can be used for an index join, which scans B, looking up A's index to retrieve the matching records.

In main memory column stores that employ dictionary encoding [9, 17, 19, 26, 27], we encounter relations that are always indexed: the dictionaries. Each dictionary holds the mapping between a value and its encoding, and every query for a list of values requires a sequence of lookups on the dictionary. In this paper, we consider these lookups a case

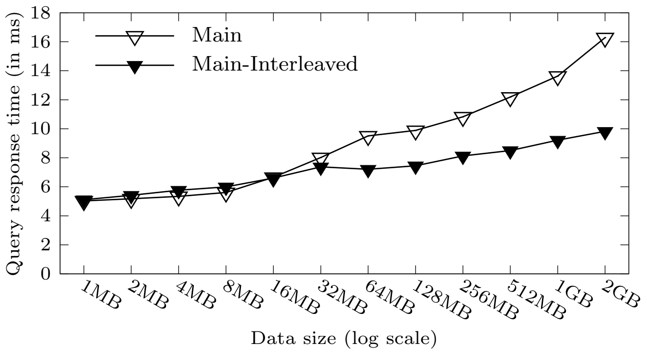


Figure 1: Response time of an IN-predicate query with 10K INTEGER values. Main memory accesses hinder sequential execution when the dictionary is larger than the cache (25 MB); interleaved execution is affected much less.

of an index join that we use to propose a practical technique that significantly enhances index join performance by hiding the cost of main memory accesses.

Like all index lookups, dictionary lookups become disproportionally expensive when the dictionary outsizes the last level cache of the processor. Figure 1 illustrates this disproportionality in the runtime of a query with an IN predicate, executed on the SAP HANA column store [9]. For the size range 1MB–2GB, we observe a significant runtime increase when the dictionary outgrows the last level cache (25MB). The increase is caused by main memory accesses (details in Section 2), a known problem for index joins [32] and main memory database systems in general [5, 20].

Traditional tenets for dealing with main memory accesses prescribe the elimination of excessive indirection and non-sequential memory access patterns. Given that dictionary implementations are cache-conscious data structures, we can assume any effected main memory accesses to be essential and thus unavoidable. Still, we can hide the latency of main memory accesses by providing the processor with enough independent instructions to execute while fetching data. In our case, lookups are independent from each other, allowing us to execute them concurrently in a time-sharing fashion: we can interleave the instruction streams of several lookups so that, when a cache miss occurs in one instruction stream, execution continues in another. Hence, the processor keeps executing instructions without having to wait for data.

Prior works propose two forms of such *instruction stream interleaving (ISI)*: static, like *group prefetching (GP)* [6] and *software pipelined prefetching (SPP)* [6], and dynamic, like the state-of-the-art *asynchronous memory access chaining (AMAC)* [15]. Static interleaving has negligible overhead for instruction streams with identical control flow, whereas dynamic interleaving efficiently supports a wider range of use cases, allowing instruction streams to diverge, e.g., with early returns. Both approaches require to rewrite code either as a group or a pipeline in the static case, or a state machine in the dynamic case (see Section 3). The interleaved code ends up in the database codebase alongside the original, non-interleaved implementation, implying functionality duplication, as well as additional testing and maintenance costs. As a result, developers can be reluctant to use interleaved execution in production.

In this work, we propose interleaving with coroutines, i.e., functions that can suspend their execution and resume at a later point. Coroutines inherently support interleaved execution, while they can also run in non-interleaved mode. A technical specification [3] for the popular C++ language introduces coroutine support at the language level: the programmer simply inserts suspension statements into the code, and the compiler automatically handles the state that has to be maintained between suspension and resumption (see Section 4). With this soon-to-be-standard language support, we implement interleaving with comparable performance to the prior proposals, supporting both interleaved and non-interleaved execution through a single implementation that closely resembles the original code (see Section 5).

Concretely, we make the following contributions:

- A technique for instruction stream interleaving based on coroutines. We exhibit our technique in index joins, where we use sorted arrays and CSB$^+$-trees as indexes, describing how it can be applied to any type of pointer-based index structure.
- A comparison with *group prefetching (GP)* and *asynchronous memory access chaining (AMAC)*. Since coroutines are a dynamic interleaving approach, they are equivalent to AMAC in terms of applicable use cases and performance without the need for an explicit state machine. Coroutine code closely resembles the original implementation and can be used in both interleaved and non-interleaved execution, relying on the compiler for state management and optimization.
- An optimized implementation for IN-predicate queries with predictable runtime, proportional to the dictionary size. This implementation is part of a prototype based on SAP HANA, and targets both sorted and non-sorted dictionaries for INTEGER columns.

Demonstrating how coroutines minimize the effort of implementing *ISI*, we believe our contributions provide a strong argument for interleaved execution in real DBMS codebases.

2. BACKGROUND & RELATED WORK

In this section, we describe how column stores use dictionary encoding, explain why dictionary lookups performed in bulk are instances of index joins, and establish IN-predicate queries against dictionary encoded columns as the running example for this work. Further, we quantify the negative effect of main memory accesses on dictionary lookups when the dictionary outsizes the last level cache, and justify why eliminating all main memory accesses is unrealistic for large dictionaries regardless of their implementation, thus motivating instruction stream interleaving.

2.1 Dictionaries in Column Stores

Dictionary encoding is a common compression method in main-memory column stores, e.g. [9, 17, 19, 26, 27]. It maps the value domain of one or more columns to a contiguous integer range [7, 9, 10, 12, 18, 20]. This mapping replaces column values with unique integer *codes* and is stored in a separate data structure, the *dictionary*, which supports two access methods:

- `extract` returns the *value* for a *code*.
- `locate` returns the *code* for a *value* that exists in the dictionary, or a special *code* that denotes absence.

The resulting vector of codes and the dictionary constitute the encoded column representation. The code vector is usually smaller than the original column, reflecting the smaller representation of codes, whereas the dictionary size is determined by the value domain, which can comprise from few to billions of distinct values as encountered by database vendors in customer workloads [25].

In this work, we use the column store of SAP HANA, which has two parts for each column: the read-optimized *Main*, against which the query in Figure 1 was run, and the update-friendly *Delta*. A *Main* dictionary is a sorted array of the domain values, and the array positions correspond to codes, similarly to [9, 17, 27]. Hence, `extract` is a simple array lookup, whereas `locate` is a binary search on the array contents for the appropriate array position. On the other hand, *Delta* dictionaries are implemented as unsorted arrays indexed by a cache-conscious B$^+$-tree (CSB$^+$-tree) [28]; `extract` is again an array lookup, but `locate` is now an index lookup on the CSB$^+$-tree.

Sorted or not, a dictionary array can be considered a relation $D(code, value)$ that is indexed on both attributes: codes are encoded as array indices, whereas values can be retrieved through binary search or index lookup, respectively in the sorted and the unsorted case; here, we focus on the *value* index. Since a sequence of values is also a relation $S(value)$, every value lookup from a column involves a join $S \bowtie D$, which is performed as an index join when $|S| << |D|$. Such joins dominate the IN-predicate queries that we discuss next.

2.2 IN Predicates and Their Performance

In this paper, we study the problem of random memory accesses in queries with IN predicates [23], yet our analysis and the coroutine-based technique we propose apply to any index join that involves pointer-based data structures like hash tables or B$^+$-trees, or algorithms involving chains of non-sequential memory accesses like binary search.

IN predicates are commonly used in an ETL process to extract interesting items from a table. An IN predicate is encountered in the WHERE clause of a query, introducing a subquery or an explicit list of values that the referenced column must match. Listing 1 showcases an IN predicate from Q8 of TPC-DS [4], which extracts all zip codes from the *customer_address* table that belong in a specified list of 400 predicate values.

```
SELECT substr(ca_zip,1,5) ca_zip
FROM customer_address
WHERE substr(ca_zip,1,5)
   IN ('24128', ..., '35576')
```

Listing 1: IN predicate excerpt from TPC-DS Q8.

When IN-predicate queries are executed on dictionary-encoded data, the predicate values need to be encoded before the corresponding rows can be searched in the code vector. This encoding comprises a sequence of `locate` operations, which can be viewed as an index join, as we described in the previous subsection. Ideally, the runtime of an IN-predicate query would depend only on the computation involved, resembling the *Interleaved* data series of Figure 1 rather than the measured *Sequential* one. This behavior is not observed only in Main; Delta has a similar problem, as illustrated in Figure 8.

To identify what causes the significant runtime increase for large dictionaries of both Main and Delta, we profile the query execution for the smallest and largest dictionary sizes, i.e., 1 MB and 2 GB. The resulting list of hotspots identifies dictionary lookups (`locate`) as the main execution component, as shown in Table 1.

Table 1: Execution details of `locate`.

	Main		Delta	
	1 MB	2 GB	1 MB	2 GB
Runtime %	21.4	65.7	34.3	78.8
Cycles per Instruction	0.9	6.3	0.7	4.2

In the 1 MB case, `locate` contributes just 21.4%(34.3%) of the total execution time for Main(Delta), but this contribution surges to 65.7%(78.8%) for the 2 GB case. These surges can be attributed to the 7×(6×) increase in the *cycles per instruction (CPI)* ratio between the 1 MB and the 2 GB cases. To explain the CPI difference, we investigate microarchitectural behavior applying the Top-Down Microarchitectural Analysis Method(TMAM) for identifying performance bottlenecks in out-of-order cores [11]. Below we establish the terminology used in the rest of the paper.

Top-down Microarchitecture Analysis. This method uses a simplified model for the instruction pipeline of an out-of-order core, consisting of two parts:
Front-end: Fetches program instructions and decodes them into one or more micro-ops (μops), which are then fed to the Back-end—up to four μops per cycle in Intel architectures.
Back-end: Monitors when the operands of a μop become available and executes it on an available execution unit. μops that complete execution *retire* (again up to four μops per cycle) after writing their results to registers/memory.

Furthermore, TMAM uses *pipeline slots* to abstract the hardware resources necessary to execute one μop and assumes there are four available slots per cycle and per core. In each cycle, a pipeline slot is either filled with a μop, or remains empty (*stalled*) due to a stall caused by either the *Front-end* or the *Back-end*. The *Front-end* may not be able to provide a μop to the *Back-end* due to, e.g., instruction cache misses; whereas the *Back-end* can be unable to accept a μop from the *Front-end* due to data cache misses (*Memory*) or unavailable execution units (*Core*). In the absence of stalls, the slot can either retire (*Retirement*) or execute non-useful work due to *Bad Speculation*.

In Table 2, we present the pipeline slots of `locate`'s execution[1] divided into the aforementioned categories. In the 2 GB case, memory stalls account for 46.0% and 85.9% of the pipeline slots, respectively for Main and Delta, while they are relatively less prominent in the 1 MB case. The stalls occur from random accesses in the dictionary array for Main and to the index nodes for Delta. The 1 MB dictionary fits in the processor caches, so memory stalls are avoided with out-of-order execution. For the 2 GB dictionary, only the first few binary search iterations (Main) or tree levels (Delta) are expected to be in a warmed-up cache, since they are reused by many lookups, while the rest of the data requests incur main memory accesses. Each main memory access has a latency of 182 cycles [11] that cannot be hidden with out-of-order execution, hence the observed memory stalls.

[1]Retrieved from a profiling session of 60 seconds.

Table 2: Pipeline slot breakdown for `locate`.

	Main		Delta	
	1 MB	2 GB	1 MB	2 GB
Front-End	10.4%	3.5%	0.7%	0.2%
Bad speculation	43.3%	26.1%	0.0%	0.3%
Memory	2.8%	46.0%	30.8%	85.9%
Core	16.4%	20.5%	28.9%	7.3%
Retiring	27.0%	3.9%	40.0%	6.3%

```
1  function lookup(table, value){
2    size = table.size()
3    low = 0
4    while ⌊size/2⌋ > 0 do
5      half = ⌊size/2⌋
6      probe = low + half
7      v = table[probe]
8      if v < value then
9        low = probe
10     size −= half
11   return low
12 }
```

Listing 2: Binary search.

Furthermore, a significant fraction of the issued instructions in Main never retire, regardless of the dictionary size. These instructions are speculatively executed and then rolled back because of bad speculation, which is inherent to binary search: the search continues *with the same probability* in either the left or the right subarray from the current array position, depending on the result of the comparison between the value of the current position and the one we are looking for (line 8 in Listing 2). To avoid waiting for the result of the comparison, the processor predicts which of the two alternative control flow paths will be chosen and executes it speculatively. Given both alternatives have the same probability, the prediction is wrong 50% of the time, so the speculatively executed instructions have to be rolled back. Nevertheless, a binary search implementation can avoid speculation using a *conditional move*; the Delta uses such an implementation for the tree nodes, so no pipelines slots get wasted due to bad speculation. When the comparison operands reside in the cache, avoiding speculation is preferable since there is little latency to hide; still, in case the comparison needs data to be fetched from main memory, speculated execution is better, as we explain in Section 5.4.1.

Moreover, we believe bad speculation is also the main reason for the front-end stalls we observe in Main, given that these stalls are negligible in Delta, and do not appear in the non-speculative microbenchmarks we study in Section 5. Finally, the core fraction in both Main and Delta contains stalls due to data unavailable execution units.

Takeaway. Memory stalls due to data cache misses become the main source of inefficiency for dictionaries that do not fit in the cache.

2.3 Tackling Cache Misses

In the literature, we find many software techniques to deal with cache misses. Based on how they affect the number of cache misses and the incurred penalty, these techniques fall into the following three categories:

- **Eliminate cache misses** by increasing spatial and temporal locality. Locality is increased by a) eliminating indirection, designing cache-conscious data structures like the CSB$^+$-tree [28]; b) matching the data layout to the access pattern of the algorithm, i.e, store data that are accessed together in contiguous space; or c) reorganizing memory accesses to increase locality, e.g., with array [16] and tree blocking [14, 32]. In this work, we assume that the index has the best possible implementation and locality cannot be further increased without penalizing single lookups. Nonetheless, our proposal can be applied to any index structure.
- **Reduce the cache miss penalty** by scheduling independent instructions to execute after a load; this approach increases instruction-level parallelism and leads to more effective out-of-order execution. To reduce the main-memory access penalty, a non-blocking load has to be introduced early enough, allowing independent instructions to execute while fetching data. This is achieved through simple *prefetching* within an instruction stream, or exploiting *simultaneous multithreading* with helper threads that prefetch data [31]. In index lookups, however, one memory access depends on the previous one with few independent instructions in-between, so these techniques do not apply.
- **Hide the cache miss penalty** by overlapping memory accesses. The memory system can serve several memory requests in parallel (10 in current Intel CPUs) and exploiting this *memory-level parallelism* increases memory throughput. However, overlapping requires independent memory accesses, which do not exist in the access chain of an index lookup.

Takeaway. These approaches do not benefit individual index lookups. Next, we show how to hide the cache misses of a group of lookups with interleaved execution.

3. INTERLEAVED EXECUTION

In this section, we present the idea of instruction stream interleaving, how it applies to IN-predicate queries, and existing techniques that can be used to implement it.

We deal with the general case of code with independent instruction streams, some or all of which exhibit memory access patterns that hardware prefetchers cannot predict. Our objective is to overlap memory accesses from one instruction stream with computation from others, keeping the instruction pipeline filled instead of incurring memory stalls. We interleave the execution of several instruction streams, switching to a different instruction stream each time a load stalls on a cache miss. Therefore, we call this approach *instruction stream interleaving (ISI)*.

Figure 2 illustrates *ISI* through an example with three binary searches on a sorted array with eight elements. IS A, B, and C are instruction streams that correspond to each binary search. For simplicity, we assume all array accesses are cache misses and all other memory operations hit in the cache. Each instruction stream accesses the array three times, splitting the instruction stream into four computa-

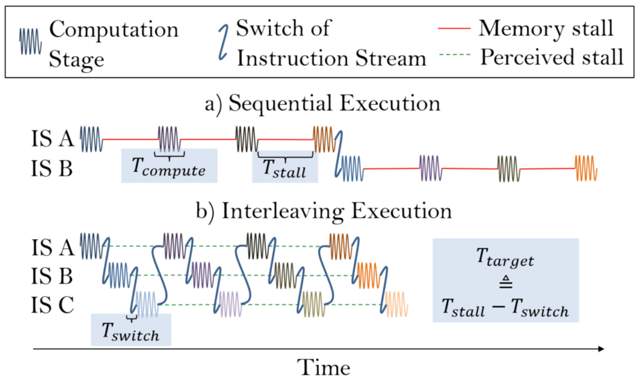


Figure 2: Sequential vs interleaved execution.

tion stages of duration $T_{compute}$. With sequential execution, the three instruction streams run one after the other (we omit the third instruction stream due to lack of space), and there is a mechanism (e.g., a loop) that, when IS A finishes, switches to IS B and then to IS C. Every array access leads to a cache miss and a corresponding T_{stall}. With interleaving, execution switches from one instruction stream to another at each memory access, incurring an overhead T_{switch} (per instruction stream) that overlaps with T_{stall} and leaves $T_{target} = T_{stall} - T_{switch}$ stalls. In this case, when IS A accesses the array during its first stage, execution switches to the first stage of IS B, then to the first stage of IS C, and then back to IS A for the second stage.

We generalize this example to model a group of G instruction streams, where each instruction stream i has distinct $T_{i,compute}$, $T_{i,switch}$, and $T_{i,target}$ parameters. $T_{i,target}$ is removed *iff* $T_{i,target} \leq \sum_{j \in [0..G)}^{j \neq i} (T_{j,compute} + T_{j,switch})$. In case of identical model parameters across the instruction streams, we can drop the indices and get:

$$G \geq \frac{T_{target}}{T_{compute} + T_{switch}} + 1 \tag{1}$$

Inequality 1 estimates the optimal G, i.e., the minimum group size for which stalls are eliminated. Interleaving more instruction streams does not further improve performance since there are no stalls; to the contrary, performance may deteriorate due to cache conflicts—see Section 5.4.5.

Considering the total execution of an instruction stream group, *ISI* belongs in the second category of the taxonomy in Section 2.3. Nevertheless, it becomes also a member of the third category when memory stalls from different instruction streams overlap, e.g., the example in Figure 2.

Implementing *ISI*. In principle, we can implement *ISI* using any cooperative multitasking technique. However, interleaved execution makes sense only when $T_{stall} >> T_{switch}$, i.e., the mechanism employed to switch instruction streams requires significantly less cycles than the penalty of the corresponding cache miss. Furthermore, an effective switching mechanism should not introduce more cache misses.

OS threads are a natural candidate to encode instruction streams and can be synchronized to cooperatively multitask; the granularity we consider, however, rules out such implementations: preemptive multithreading imposes non-negligible synchronization overhead, context switching involves system calls and takes several thousand cycles to complete, while switching among full-blown thread stacks likely thrashes the cache and the TLB. Hence, the techniques we present below do not depend on OS multithreading; instead, they eliminate stalls, increasing the efficiency of thread execution. Given an amount of work, interleaving techniques reduce the necessary execution cycles in both single- and multi-threaded execution.

Existing techniques. In the literature, we find prefetching techniques that implement restricted forms of instruction stream interleaving. Chen et al. [6] proposed to exploit instruction stream parallelism across subsequent tuples in hash joins by manually applying well-known loop transformations that a general-purpose compiler cannot consider due to lack of dependency information. They proposed *group prefetching (GP)* and *software-pipelined prefetching (SPP)*, two techniques that transform a fixed chain of N memory accesses inside a loop into sequences of N+1 computation stages separated by prefetches. *GP* executes each stage for the whole group of instruction streams in the loop before moving to the next stage; whereas *SPP* executes a different instruction stream at each stage in a pipeline fashion. Both techniques interleave instruction streams, although they target ones with a fixed number of stages.

To apply *GP* on the dictionary lookups of IN predicates, we decompose the loop of a binary search (lines 4–11 in Listing 2) into a prefetch and a load stage (lines 9–13 and 14–20 in Listing 3). The number of times these two stages are repeated depends on the `table` size, therefore the vanilla *GP* proposal does not apply. Nevertheless, the idea behind *GP* is not inherently restricted to a fixed number of stages: in cases like the dictionary lookups of IN predicates, the stage sequence is the same for all instruction streams, enabling us to use a variation of *GP*[2] in the implementation of Listing 3.

```
1  struct state { value; low; }
2
3  procedure bulk_lookup_gp(
4    group_size, table, table_size, input
5  ){
6    foreach value_group in input do
7      size = table_size
8      init search_group
9      while ⌊size / 2⌋ > 0 do
10       half = ⌊size / 2⌋
11       foreach state in search_group do
12         probe = state.low + half
13         prefetch table[probe]
14       for i = 1 to group_size do
15         state = search_group[i]
16         probe = state.low + half
17         v = table[probe]
18         if v <= value_group[i] then
19           state.low = probe
20       size −= half
21     foreach state in search_group do
22       store state.low
23 }
```

Listing 3: Binary search with *GP*.

The loop is shared among all instruction streams in a group, reducing the state variables that have to be maintained for each instruction stream, as well as the executed instructions. However, sharing the loop means the instruction streams are coupled and execute the same instruction sequence.

Kocberber et al. [15] considered this coupling as a limitation that complicates cases where each instruction stream follows a different control flow. To decouple the progress of different instruction streams, they proposed the state-of-the-art *asynchronous memory access chaining (AMAC)*, a technique that encodes traversals of pointer-intensive data structures as finite state machines. The traversal code is manually rewritten to resemble a state machine, enabling each instruction stream in a group of traversals to progress independently from others, based only on its current state.

```
1  enum stage { A, B, C }
2  struct state {
3    value; low; probe; size; stage
4  }
5
6  struct circular_buffer {
7    ... //members
8    function load_next_state() { ... }
9  }
10
11 procedure bulk_lookup_amac(
12   group_size, table, table_size, input
13 ) {
14   init b_f //circular_buffer of group_size
15   not_done = group_size
16   while not_done > 0 do
17     state = b_f.load_next_state()
18     switch (state.stage){
19     case A:  //Initialization
20       if index < input_size then
21         state.low = 0
22         state.value = input[index++]
23         state.size = table_size
24         state.stage = B
25       else
26         state.stage = Done
27         not_done = not_done − 1
28       break
29     case B:  //Prefetch
30       if ⌊state.size / 2⌋ > 0 then
31         half = ⌊state.size / 2⌋
32         state.probe = state.low + half
33         prefetch table[state.probe]
34         state.size −= half
35         state.stage = C
36       else
37         //Output result
38         state.stage = A
39       break
40     case C: //Access
41       v = table[state.probe]
42       if v <= state.value then
43         state.low = state.probe
44       else
45         state.stage = B
46       break
47     }
48     store state in b_f
49 }
```

Listing 4: Binary search with *AMAC*.

In Listing 4, we illustrate lookups on a sorted dictionary interleaved with *AMAC*. The state machine code is a switch statement (line 18–47) with one case for each stage. The state of each instruction stream is stored in a buffer (line 14) and retrieved in a round-robin fashion. The state machine examines the current stage of the instruction stream and decides how to proceed. This way, instruction streams can progress independently—but at the cost of an implementation that has little resemblance to the original code.

Takeaway. Table 3 summarizes the properties of the *ISI* implementation techniques we study in this paper.

[2]We have not yet investigated how to form a pipeline with variable size, so we do not provide a *SPP* implementation.

Table 3: Properties of interleaving techniques.

Interleaving Technique	IS Coupling	IS Switch Overhead	Added Code Complexity
GP	Yes	Very Low	High
AMAC	No	Low	Very High
Coroutines	No	Low	Very Low

GP adds minimum overhead owing to execution coupling, whereas *AMAC* supports more use cases by allowing each instruction stream to proceed independently. Nevertheless, both *GP* and *AMAC* require intrusive code changes that obfuscate the original control flow, incurring high costs for development, testing, and maintenance. These costs make the two techniques impractical to use in a large codebase.

In the next section, we present an interleaving technique that requires minimal non-intrusive code changes.

4. INTERLEAVING WITH COROUTINES

A coroutine is a control abstraction that extends subroutines [24]. A subroutine starts its execution upon *invocation* by a caller and can only run to *completion* where the control is returned to the caller. The coroutine construct augments this lifetime with *suspension* and *resumption*: a coroutine can suspend its execution and return control before its completion; the suspended coroutine can be resumed at a later point, continuing its execution from the suspension point onward. To resume a coroutine, one has to use the coroutine handle that is returned to the caller at the first suspension.

Although coroutines were introduced in 1963 [8], mainstream programming languages did not support them until recently, except for restricted generator constructs in languages like C# [1] and Python [2]. The advantages of coroutines gave rise to library solutions, e.g., Boost.ASIO and Boost.Coroutine[3], which rely on tricks like Duff's device[4], or OS support like fibers on Windows [30] and `ucontext_t` on POSIX systems [13]. Asynchronous programming and its rise in popularity brought coroutines to the spotlight as a general control abstraction for expressing asynchronous operations without callbacks or state machines. Languages like C#, Python, Scala and Javascript have adopted coroutine-like `await` constructs; C++ has a technical specification for coroutines as a language feature [3], which at the time of writing is supported by the Visual C++ and the Clang compiler. Naturally, database implementations have also picked up coroutines to simplify asynchronous I/O [29], but, to the best of our knowledge, not to hide cache misses.

Implementing *ISI*. Coroutines can yield control in the middle of their execution and be later resumed. This ability makes them candidates for implementing *ISI*, as already remarked by Kocberber et al. [15]. An efficient implementation needs a) a suspension/resumption mechanism that consumes a few tens of cycles at most, and b) a space footprint that does not thrash the cache. Our interleaving with coroutines proposal satisfies these requirements by using the *stackless coroutines* as specified for C++ in [3].

Coroutines as state machines. A stackless coroutine[5] is compiled into assembly code that resembles a state machine. In a sequence of transformation steps, the compiler splits the body of a coroutine into distinct stages that are separated by the suspension/resumption points. These stages correspond to the state machine stages that a programmer derives manually for AMAC; in the coroutine case, however, the compiler performs the transformation, taking also care of preserving the necessary state across suspension/resumption. The compiler identifies which variable to preserve and stores them on the process heap, in a dedicated coroutine frame that is analogous to each entry in the state buffer of AMAC. Beside these variables, the coroutine frame contains also the resume address and some register values; these are stored during suspension and restored upon resumption, adding an total overhead equivalent to two function calls.

Binary search as a coroutine. In Listing 5, we demonstrate with binary search how to transform an index lookup to support interleaved execution. The presented pseudocode introduces the `coroutine` keyword to denote the difference to the ordinary `procedure` and `function`. Moreover, the suspension and return statements hint to the actual code.

```
1  coroutine lookup(
2    table, table_size, value, interleave
3  ){
4    size = table_size
5    low = 0
6    while ⌊size / 2⌋ > 0 do
7      half = ⌊size / 2⌋
8      probe = low + half
9      if interleave == true then
10       prefetch table[probe]
11       co_await suspend_always()
12     v = table[probe]
13     if v < value then
14       low = probe
15     size -= half
16   co_return low
17 }
```

Listing 5: Binary search coroutine.

Calling `lookup` creates a coroutine instance and returns a handle object with the following API: a `resume` method for resuming execution; an `isDone` method that returns `true` if the coroutine completed its execution, and `false` otherwise; a `getResult` method to retrieve the result after completion.

Lines 4–16 are the code of the original sequential implementation augmented with a prefetch (line 10) and a suspension statement (line 11) before the memory access that causes the cache miss (line 12). The added code is wrapped in an `if` statement combining both sequential and interleaved execution (depending on the value of `interleave`) in a single implementation. The actual C++ code uses template metaprogramming with `interleave` as a *template* parameter to ensure the conditional statement is evaluated at compile time, therefore generating the optimal code for each case. Finally, instead of a normal `return` statement, the coroutine code uses `co_return` in line 16 to return the result.

CSB$^+$-tree lookup as a coroutine. In Listing 6, we depict the coroutine implementation for a CSB$^+$-tree lookup that adheres to the original proposal of Rao et al. [28]. For simplicity, we assume a cached root node; for all other tree levels, we prefetch all cache lines for each touched node and suspend (lines 10 to 12). Note that, for the binary search within nodes, we use the coroutine of Listing 5 without suspension; the node prefetch brings the `keyList` to the cache, so the binary search causes no cache misses. Moreover, a

[3] `http://www.boost.org/doc/libs`

[4] `https://en.wikipedia.org/wiki/Duff%27s_device`

[5] As opposed to stackfull coroutines, see [24] for details.

leaf node differs from an inner node since the result of the binary search is used to fetch the searched value from the `valueList` instead of a child node; this value is the result returned in line 16.

```
 1 coroutine tree_lookup(
 2   tree, tree_height, value, interleave
 3 ){
 4   node = tree->root
 5   while node->level > 0 do
 6     i_l = node->keyList; i_n = node->nKeys
 7     handle = lookup(i_l, i_n, value, false)
 8     i_c = node->firstChild
 9     node = i_c + handle.getResult()
10     if interleave then
11       prefetch node
12       co_await syspend_always()
13   l_kl = node->keyList; l_n = node->nKeys
14   handle = lookup(l_kl, l_n, value, false)
15   l_vl = node->valueList
16   co_return l_vl[handle.getResult()]
17 }
```

Listing 6: CSB$^+$-tree lookup coroutine.

Sequential and Interleaved Execution. The `lookup` described above can be executed with or without suspension, depending on the *scheduler*, i.e., the code implementing the execution policy for the lookup sequence.

```
 1 procedure runSequential(
 2   index, index_size, values, results
 3 ){
 4   foreach value in values do
 5     handle = lookup(index,
 6         index_size, value, false)
 7     result = handle.getResult()
 8     store result to results
 9 }
10
11 procedure runInterleaved(
12   index, index_size, values, results
13 ){
14   for i = 0 to group_size − 1 do
15     value = values[i]
16     handles[i] = lookup(index,
17         index_size, value, true)
18   not_done = group_size
19   i = group_size
20   while not_done > 0 do
21     foreach handle in handles do
22       if not handle.isDone() then
23         handle.resume()
24       else
25         result = handle.getResult()
26         store result to results
27         if i < values.size() then
28           handle = lookup(index,
29               index_size, values[i], true)
30           i = i + 1
31         else
32           not_done = not_done − 1
33 }
```

Listing 7: Sequential and interleaved schedulers.

In Listing 7, we present two schedulers:

- The `runSequential` scheduler performs the lookups one after the other (lines 4–8). The coroutines are called with `interleaved=false`, so they do not suspend. The only difference to a normal lookup function is that we retrieve the result through the handle.
- The `runInterleaved` scheduler initializes a group of coroutines, specifying `interleaved=true`, and maintains a buffer of coroutine `handles` (lines 15–17). Since `lookup` execution now suspends, the `while` loop over the buffer resumes unfinished lookups (line 23), or retrieves the results from the finished lookups (lines 25–26) and starts new ones (lines 27–29).

Either of the two schedulers can be selected depending on the probability of cache misses in a lookup and amount of lookup parallelism present; in cases like the node search in Listing 6, or when there is no other work to interleave with, sequential execution is better as it incurs no overhead.

Finally, since the schedulers are agnostic to the coroutine implementation, they can be used with any index lookup.

Performance considerations. The described way of interleaving with coroutines relies on an optimizing compiler to generate assembly that a) in interleaved execution, recycles coroutine frames from completed lookups for subsequent coroutine calls (lines 25–29 in Listing 7), and b) in sequential execution, allocates no coroutine frame, since it is not necessary for non-suspending code (lines 5–6 in Listing 7). These optimizations avoid unnecessary overhead by eliding unnecessary frame allocations. At the time of writing, the Visual C++ compiler we use does not perform these optimizations, so we apply them manually in separate implementations for sequential and interleaved execution. As compiler support matures, manual optimization separately for sequential and interleaved execution will become unnecessary.

Takeaway. To implement interleaving with coroutines means essentially to add suspension statements to sequential code at each point where there will probably be a cache miss. Furthermore, the same coroutine implementation supports both sequential and interleaved execution, depending on the scheduler we use. As we show next, coroutines make interleaved execution significantly easier to adopt compared to *GP* and *AMAC*, while offering similar performance.

5. EXPERIMENTAL EVALUATION

In this section, we demonstrate that *interleaving with coroutines* is easier to code and maintain, and performs similarly to the other two *ISI* techniques studied in this work, i.e., *GP* and *AMAC*. First, we compare the three techniques in interleaved binary searches, highlighting the minimal code overhead and the few modifications of interleaving with coroutines. Second, we demonstrate the advantages of interleaving over sequential execution for binary searches over `int` and `string` arrays. Third, we explain the performance gains with a thorough microarchitectural analysis of the `int` case, where we also show how to estimate the best *group size*—the number of concurrent instruction streams. Finally, we implement our coroutine-based technique in both the Main and the Delta of SAP HANA, enabling IN-predicate query execution with robust response times.

5.1 Methodology

Microbenchmarks. We study five binary search implementations: two for sequential execution and three for interleaved. The sequential ones are `std::lower_bound` from the C++ standard library (abbreviated as `std`), and `Baseline`, which is similar to Listing 2 and uses a conditional move to avoid speculative execution. Based on `Baseline`, we implement the three *ISI* techniques. For each implementation, we can configure the `group size`, i.e., how many lookups run interleaved at any given point in time. Finally, GP and AMAC resemble the pseudocode in Listings 3 and 4 respectively, whereas CORO is a modified version of Listing 5 that avoids memory allocations by using the same coroutine frame for subsequent binary searches.

Table 4: Architectural parameters.

Processor	**Intel Xeon 2660v3 [11]**
Architecture	Haswell
Technology	22nm @ 2.6GHz
# Cores	10 (Hyperthreading disabled[6])
Core Type	4-wide OoO
L1 I/D (per core)	32 KB/32 KB, 8-way associative
# Line Fill Buffers	10
L2 Cache	256 KB, 8-way associative
LLC Cache	25 MB
DTLB	64 entries, 4-way associative
STLB	1024 entries, 8-way associative

Experimental Setup. The workstation used in our experiments is listed in Table 4. It features two Intel Xeon 2660 v3 processors with 10 cores per socket and runs Windows 10 1511. For our measurements, we have pinned our microbenchmarks on one core and migrated all other processes to the other socket in order to minimize performance variability due to uncontrollable thread migrations between sockets and external interference from other processes.

To use coroutines, we compile our code with the Microsoft Visual C++ (MSVC) v14.1 compiler[7]. To prefetch data, we use the instruction PREFETCHNTA (through the compiler intrinsic `_mm_prefetch(ptr, _MM_HINT_NTA)`). The compilation flags used are: `/Ox /arch:AVX2 /await`. Finally, we use Intel VTune Amplifier XE 2017 to profile execution and observe microarchitectural behavior.

5.2 Code Complexity and Maintainability

By comparing the implementation methodologies of the three *ISI* techniques (and the corresponding examples in Listings 3, 4, and 5), we intuitively see that interleaving with coroutines is easier to implement and maintain than the prior techniques. To validate this intuition, we calculate the lines of code (LoC) that are different between each of the *ISI* implementations and the original sequential code (*Diff-to-Original*), as well as the total LoC one has to maintain per lookup algorithm, e.g., binary search, to support both sequential and interleaved execution (*Total Code Footprint*). The first metric hints to the implementation complexity, while the second one to maintainability; for both metrics, lower values are better.

In Table 5, we present the two metrics for GP and AMAC, as well as CORO for both the proposed unified implementation (CORO-U) and the separate implementations (CORO-S). CORO-U requires the least modifications/additions (6 LoC) to the original code to be implemented, while it has the smallest code footprint (16 LoC) thanks to the unified codepath; all other implementations have separate codepaths for each mode of execution and, thus, have two implementations for the same lookup algorithm. Nonetheless, both CORO variants have significantly less code than GP and AMAC.

[6]To simplify the interpretation of measurements [21].

[7]Clang did not have a stable support at the time we performed the experiments, hence the use of MSVC.

Table 5: Implementation complexity and code footprint of *ISI* techniques. The two CORO variants differ the least from the original code (11 LoC) and require the least code to support both sequential and interleaved execution.

Technique	GP	AMAC	CORO-U	CORO-S
Interleaved	24	67	15	18
└ Diff-to-original	18	64	6	9
Total Code Footprint	35	78	16	29

Why not use essential or cyclomatic complexity? Contrary to the two LoC metrics we use above, standard metrics like essential and cyclomatic complexity [22] reflect code properties that are not useful in determining which technique is easier to implement and maintain. Essential complexity assesses how structured the code is, examining the entry and exit points of each control flow structure used in the code; depending on their state, coroutines are entered and exited at different points, which means they have high essential complexity although they are arguably easy to understand. Moreover, cyclomatic complexity is a property of the control flow graph, which is almost identical for AMAC and CORO since they are both state machines with the same states; consequently, AMAC and CORO have similar cyclomatic complexity, despite the little resemblance between them, in analogy to how a `switch` statement and the equivalent sequence of `if...else`s have the same cyclomatic complexity. For these reasons, we do not consider these two metrics in our comparison.

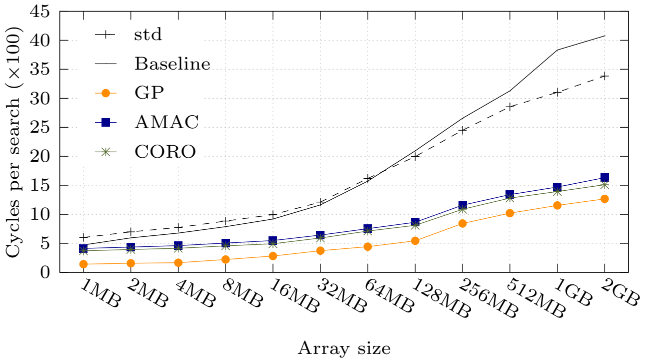
Takeaway. Interleaving with coroutines has two key properties: a) the lookup logic is kept separate from the execution policy, enabling a single codepath to be configured for sequential or interleaved execution; b) the coroutine code is the sequential code extended with a prefetch and a suspension statement per switch point. Thanks to these properties, coroutines incur significantly lower development and maintenance costs compared to prior *ISI* techniques.

5.3 Sequential vs Interleaved Execution

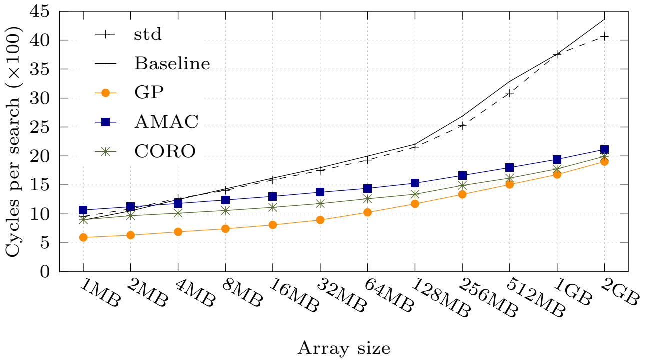
We evaluate the five aforementioned implementations on sorted arrays whose size ranges from 1 MB to 2 GB. We generate the array values using the array indices: for integer arrays, the values are the corresponding array indices, whereas for string arrays we convert the index to a string of 15 characters, suffixing characters as necessary. Further, the list of lookup values is a subset of the array values, generated using `std::uniform_int_distribution` and `std::mt19937` with seed `0`.

Figure 3 depicts the performance per binary search with lookup lists of 10K values. We report the average runtime of 100 executions, and, for ISI implementations, the depicted measurements correspond to the best group size configuration (see Subsection 5.4.5). Since the instructions executed in a binary search are a logarithmic function of the array size, the horizontal axis has a logarithmic scale.

The difference between sequential and interleaved execution is clear for both int and string arrays. `std` and `Baseline` incur an important runtime increase for arrays larger than 16 MB. These arrays outsize the last level cache (25 MB), so binary search incurs main memory accesses that manifest as stall cycles, as we explained in Section 2.2. As a result, runtime diverges significantly from the logarithmic function

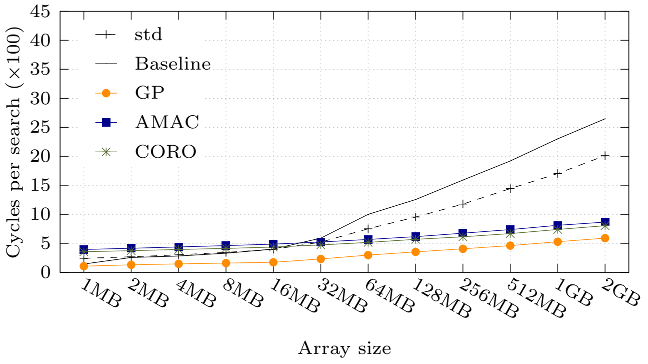


(a) Integer array.

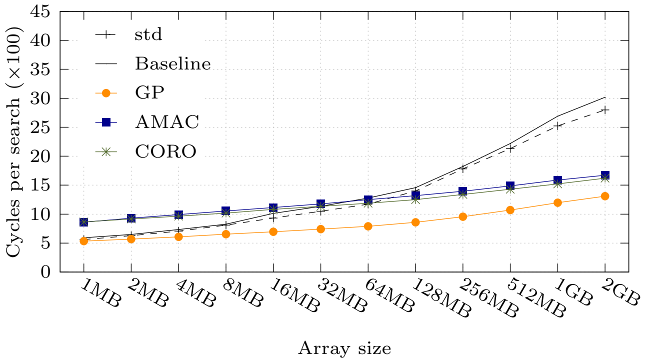


(b) String array.

Figure 3: Binary searches over sorted array. Interleaving increases runtime robustness. `CORO` performs similarly to `AMAC`, while the difference to `GP` is smaller for the string case.



(a) Integer array.



(b) String array.

Figure 4: Binary searches over sorted array with sorted lookup values. Sorting increases temporal locality, but does not eliminate compulsory cache misses.

we described above. Contrary to this behavior, runtime increases are less significant for `GP`, `AMAC` and `CORO`.

Focusing on arrays larger than the last level cache, the three `ISI` implementations behave similarly. `GP` constantly has the lowest runtime, in the range 2.7–3.7× and 1.8–2.2× respectively for integer and string values. `CORO` and `AMAC` follow with decent speedups, in the ranges 2.0–2.4× and 1.8–2.3× for integers, and in the ranges 1.4–2.1× and 1.2–1.9× for strings. We should note that, thanks to compiler optimizations, `CORO` performs slightly better that `AMAC`, whose data alignment and layout we have carefully optimized.

Finally, as array size increases, we observe a smoother increase of the interleaved execution for strings than for integers. This observation reflects the computationally heavier string comparisons, which de-emphasize cache misses. In Section 5.4, we focus on the integer case, identifying how runtime behavior changes for different array sizes.

Increasing locality with sorting. Sorting small lists is a cheap operation, and thus a valid preprocessing step. In this case, the lookup values are sorted before starting the binary searches. Figure 4 depicts the corresponding measurements for integers (strings). `std` and `Baseline` are up to 2.6×(1.8×) and 2.4×(1.8×) faster, owing to increased temporal locality: since subsequent lookups access monotonically increasing positions in the array, the values access in one lookup are likely to be cache hits in later lookups. This additional temporal locality benefits also `GP`, `AMAC` and `CORO` up to 2.2×(1.4×), 1.9×(1.3×) and 1.9×(1.3×) respectively. Still, sorting does not affect spatial locality: if the lookup values are not close to each other, which is likely for arrays much larger than the lookup lists, there will still be compulsory cache misses to hide.

Takeaway. Interleaved execution is more robust to array size increases compared to sequential execution. `CORO` has slightly better performance than the functionally equivalent `AMAC`, while `GP` performs best thanks to the minimal overhead of static interleaving. Furthermore, sorting the lookup values increases temporal locality between subsequent lookups, but does not eliminate compulsory cache misses.

5.4 Microarchitectural Analysis

To understand the effect of interleaved execution, we perform a microarchitectural analysis of our binary search implementations. We study them for int arrays and unsorted lookup values, analyzing them with TMAM (described in Section 2.2). Furthermore, we leverage the same analysis to determine the best group size for each implementation.

5.4.1 Where does the time go?

In Figure 5, we depict the execution time breakdown of a binary search as the array size increases, with the best group size for each technique, i.e., 10 for `GP`, and 6 for `AMAC` and `CORO` (we describe how to determine these values in Section 5.4.5). We calculate the execution cycles spent on front-end, memory or resource stalls, wasted due to bad speculation, or retired normally (as specified by *TMAM*) by multiplying the respective percentages reported by VTune with the measured cycles per search.

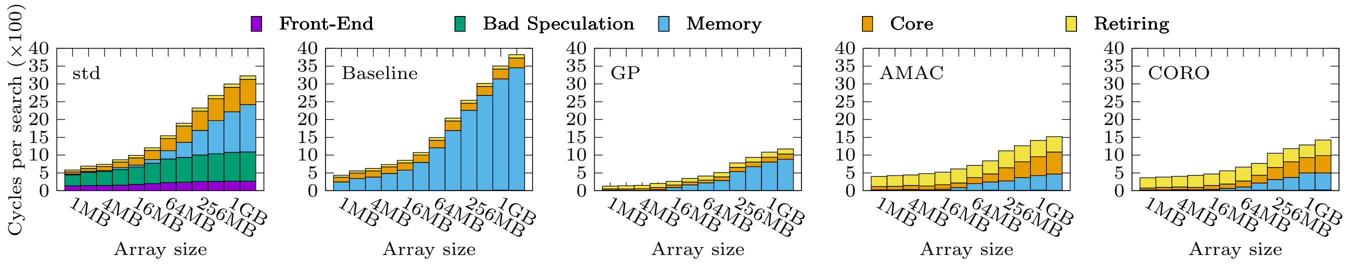


Figure 5: Execution time breakdown of binary search. Interleaved execution reduces memory stalls significantly.

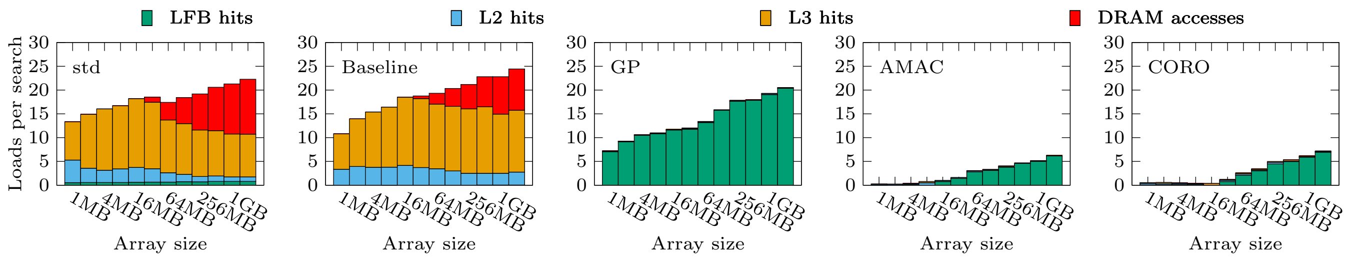


Figure 6: Breakdown of L1D misses. Interleaved execution hides the latency of data cache misses.

Owing to the small instruction footprint of our implementations, the front-end and bad speculation components are negligible in all implementations except for `std`, which is penalized by bad speculation as explained in Section 2.2. Notably, however, `std` runs faster than `Baseline` for arrays larger than 16 MB; this means that speculation, even if it is bad half the time, is better that waiting for the data to be fetched from the main memory.

Compared to `std` and `Baseline`, memory stalls are reduced in `GP`, `AMAC` and `CORO`. They are negligible until 4 MB, and they start to dominate `GP` execution from 32 MB; in `AMAC` and `GP` memory stalls are even fewer, but come with more resource stalls and normally retiring cycles, as a result of their instruction overhead which is larger than `GP`.

Takeaway. Interleaving significantly reduces the memory stalls that dominate sequential binary search execution.

5.4.2 *How does interleaving reduce memory stalls?*

Memory stalls occur when a load instruction fetches data from an address that is not in the *L1D* cache. In this case, the *Line Fill Buffers (LFB)* are checked to see if there is a memory request for the same cacheline. If not, a new memory request is created, an empty *LFB* is allocated to track the status of the request, and the request itself gets forwarded to the *L2* cache. If the requested address is not in the *L2*, the request is next forwarded to the *L3* cache (also called last level cache, *LLC*), which is shared among the cores in a socket for the Intel processor used in our experiment. Finally, if the address is not in the *L3*, the request goes to the memory controller and subsequently to the main memory (*DRAM*). Depending on the level where the requested address is found, we categorize a load as a L1D hit, a LFB hit, a L2 hit, a L3 hit, or a DRAM access.

In Figure 6, we depict a breakdown of the load instructions per implementation and array size, based on the memory hierarchy level in which they hit and omitting L1D hits as they do not cause lengthy memory stalls. We generally observe that, with interleaved execution, most L1D misses are LFB hits. The reason for this behavior is the use of prefetch instructions by the interleaving techniques: each prefetch that misses in L1D creates a memory request allocating an LFB; the corresponding load either finds the data in L1D in case enough instructions are executed between the prefetch and the load, or finds the allocated LFB otherwise. The instructions `GP` injects between a prefetch and the corresponding load are not enough to effectively hide L1D misses, despite using the best group size (see Section 5.4.5 for an explanation); still, they reduce the average miss latency, leading to the observed runtime decreases. Contrary to `GP`, `AMAC` and `CORO` eliminate most L1D misses for arrays up to 32 MB; for larger arrays, the effected L1D misses seem to be caused by address translation, as we describe in Section 5.4.3.

Takeaway. Interleaved execution introduces enough instructions between a prefetch and the corresponding load, decreasing the average memory latency of load instructions.

5.4.3 *How does address translation affect execution?*

In Section 5.3, we note that runtime increases smoothly for string arrays. However, in the measurements for integer arrays, we observe *runtime jumps* when increasing the array size from 4 MB to 8 MB, from 16 MB to 32 MB, and with every increase beyond 128 MB. Since the memory load analysis of Section 5.4.2 cannot explain these runtime jumps, we monitor and analyze the address translation behavior.

Profiling shows most loads hit in the DTLB, the first-level translation look-aside buffer for data addresses. However, DTLB misses can hit in the STLB, the second-level TLB for both code and data addresses; or perform a page walk to find the address mapping in the page tables. In the latter case, the appropriate page tables can be located in any level of the memory hierarchy—we denote the page walks that hit L1D, L2, L3 and DRAM as PW-L1, PW-L2, PW-L3 and PW-DRAM respectively.

The aforementioned runtime jumps correspond to parameters related to address translation. The first runtime jump from 4 MB to 8 MB matches the STLB size, and our profiling results show PW-L1 hits for larger arrays, while the second one, from 16 MB to 32 MB, corresponds to PW-L2 hits. Since the latencies of L1D and L2 are partially hidden by out-of-order execution, the two first jumps are small. However, the PW-L3 hits that cause the third jump cannot be hidden, so increasing the array size beyond 128 MB incurs the most evident runtime increases.

We should note that interleaving works thanks to prefetch instructions. A prefetch does not block the pipeline in case of an L1D miss, and thereby allows subsequent instructions in the instruction stream to execute. Yet, the pipeline is blocked until the prefetched virtual address is translated to a physical one, possibly involving long page walks.

Takeaway. Larger array sizes imply higher address translation latency that cannot be hidden with interleaving.

5.4.4 *Why does* `GP` *perform best?*

The performance difference between the three instruction stream interleaving techniques can be explained by their respective instruction overhead: Compared to `Baseline`, from which they are derived, `GP`, `AMAC` and `CORO` execute 1.8×, 4.4× and 5.4× more instructions. These instruction overheads, also reflected as more retiring cycles in Figure 5, correspond to the overhead of switching among instruction streams, which mainly consists of managing state.

Many binary searches on the same array is a best case scenario for group prefetching: all instruction streams within a group execute the same code for the same number of iterations. The instruction streams share the binary search loop, reducing the number of instructions executed and the number of state variables that have to be tracked per instruction stream. As we describe in Listing 3, the tracked state variables include only the searched `value` and the current `low`, whereas `probe` is inexpensively recomputed. In addition to these variables, the non-coupling `AMAC` and `CORO` have to maintain the loop state separately per instruction stream, which means they execute more load and store instructions when switching between instruction streams.

Takeaway. Contrary to `AMAC` and `CORO`, `GP` shares computation among instruction streams and maintains less state per instruction stream. In other words, `GP` executes less instructions than `AMAC` and `CORO`, and thus performs best.

5.4.5 *How to choose the group size?*

As already mentioned, the results we present correspond by default to the best group size configurations for each implementation. Given that all lookups execute the same instructions, we can estimate the best group sizes by applying the interleaving model of Section 3 to the profiling measurements. From `Baseline`, we map memory stalls to T_{stall} and all other cycles to $T_{compute}$. Further, we compute T_{switch} as the difference in retiring cycles between `Baseline` and each of the three interleaved implementations for group size 1. We apply these parameters to Inequality 1 for a 256 MB int array, yielding $G_{GP} \geq 12$ and $G_{AMAC} = G_{CORO} \geq 6$.

To verify these results, we run our microbenchmarks with all possible combinations of array and group sizes. In Figure 7, we depict our runtime measurements for a 256 MB int array as a function of the group size, which ranges between 1 and 12 concurrent binary searches (performance varies little for larger group sizes). We observe that the best group sizes are 10 for `GP`, and 5–6 for `AMAC` and `CORO`. For `GP`, $G_{estimated}$ differs from $G_{observed}$ due to a hardware bottleneck: current Intel architectures have 10 LFBs (see

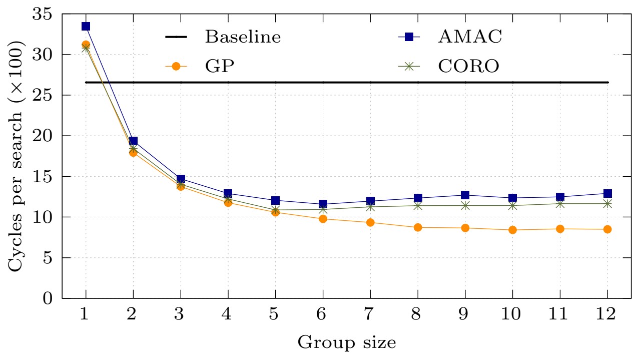


Figure 7: The effect of group size on runtime (for 256 MB int array). Best group sizes: 10 for `GP`, 5–6 for `AMAC`, `CORO`.

Section 5.4.2), limiting the number of outstanding memory accesses and, thus, the benefit of interleaving. However, 10 LFBs suffice for `AMAC` and `CORO`, corroborating our estimates.

We should note that interleaved execution with group size 1 makes no sense: `GP`, `AMAC` and `CORO` are slower than `Baseline` due to the overhead of the switching mechanism. The non-negligible overhead emphasizes the need for implementations that switch only in case of interleaved execution; otherwise, the switching mechanism should be bypassed.

Finally, similar observations to the ones above can be made for the other array sizes and for string arrays (we omit these measurements due to space limitations). Varying the array size affects the number of cache misses and not the $T_{compute}$ nor the T_{stall} per cache miss, whereas $T_{compute}$ for comparing strings with 15 characters seems to not differ significantly from integer comparison.

Takeaway. Knowing per instruction stream the available computation, the memory stalls, and the switch overhead, we can assess the effect of an interleaving technique on a group of lookups. Since the above parameters are similar for all lookups, Inequality 1 provides a reasonable estimate of the best group size, as long as the hardware supports the necessary memory-level parallelism.

5.5 IN-Predicate Queries on SAP HANA

Beside our microbenchmark evaluation, we apply our coroutine proposal in the codebase of SAP HANA. We interleave the execution of dictionary lookups in both Main (binary search) and Delta(CSB$^+$-tree lookup). The Main implementation is straightforward, but the Delta one differs from the CSB$^+$-tree described in Section 4: leaf nodes contain codes instead of values, so comparisons against the values of a leaf node incur accesses to the dictionary array to retrieve the actual values, adding an extra suspension point on the access to the dictionary array.

We evaluate the two implementations in the execution of IN-predicate queries with 1K, 5K, 10K, and 50K predicate values over INTEGER arrays with distinct values and dictionaries whose size ranges in 1 MB–2 GB. Figure 8 depicts the query response time for the 10K case. In the other cases, lookups account for a smaller part of total execution and the benefit of interleaving is less evident.

For dictionaries larger than 16 MB, interleaving reduces the Main runtime for dictionary sizes larger than the cache, from 9% at 32 MB to 40% at 2 GB, corroborating the microbenchmark results. On the other hand, the Delta runtime is

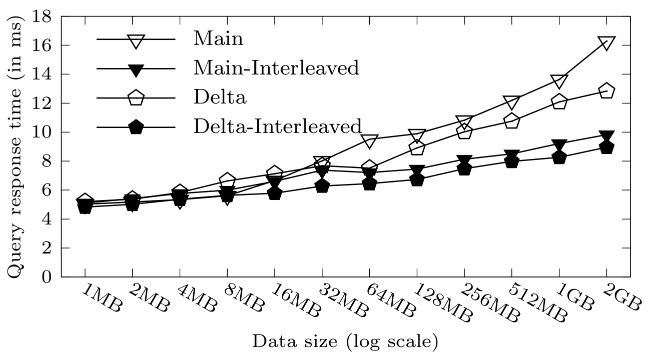


Figure 8: IN-predicate queries with 10K INTEGER values run on SAP HANA. Interleaving with coroutines increases performance in both Main and Delta.

reduced for all dictionary sizes, from 10% at 1 MB to 30% at 2 GB; this can be explained by the memory stalls Delta exhibits in the 1 MB case, as we see in Section 2.2.

Takeaway. For both Main and Delta, interleaved execution hides the cache misses of dictionary lookups as the dictionary grows larger that the last level cache. As a result, IN-predicate queries become robust to dictionary size.

6. DISCUSSION & FUTURE WORK

In this work, we have proposed a practical way to the latency of unavoidable memory access in index joins. Here, we discuss other potential applications, hardware features that can increase the efficiency of the suspension/resumption mechanism, as well as the interplay of interleaving with address translation and NUMA effects.

Other targets for interleaving. Having demonstrated interleaving with coroutines on the distinct codepaths of binary search and CSB$^+$-tree traversal, we believe our technique applies to the lookup methods of any pointer-based index. A hash-table with bucket lists is such an index, so the probe phases of hash joins that use it are straighforward candidates for our technique; moreover, since Kocberber et al. [15] demonstrate AMAC also on the build phase, and our technique is equivalent to AMAC, interleaving with coroutines applies also to important hash-join operators. In fact, our technique can be employed in any data- or task-parallel operation with a memory stall problem, like sort operators, or operations related to the state management of the lock and the transaction manager. However, if the amount of computation and stalls varies among instruction streams, we have to consider the parameters of each instruction stream separately, so we cannot use Inequality 1.

Moreover, coroutines allow instruction streams to progress asynchronously, so, in principle even different operations on multiple data-structures can be interleaved, from simple lookups to whole transactional queries. In the latter case, instruction stream latency might pose a restriction—unlike in a join—while instruction cache misses can increase front-end stalls. Consequently, the potential of interleaving requires further study.

Hardware support for interleaving. In this work, we switch instruction streams at every load that may cause cache miss, assuming the switch eliminates more memory stalls than adds instruction overhead. However, we could conditionally switch instructions streams with hardware support in the form of an instruction tells if a memory address is cached; with such an instruction, we could avoid suspension when the data is cached and unnecessary overhead.

Moreover, most of the instruction overhead comes from storing/restoring the state of each instruction stream. With a processor that supports as many hardware contexts as the instruction streams in a group, an instruction stream switch would be instant, as it would not require to swap working sets in the registers. This way, interleaving with coroutines that are aware of this support could be fast as group prefetching.

Interleaving and TLB misses. To circumvent the lack of TLB locality in binary search over large arrays, we can introduce a B$^+$-tree index with page-sized nodes on top of the sorted array. Lookups on this structure traverse the tree nodes, performing binary searches within each of them. Each binary search involves memory accesses within a single page, so the corresponding address translations hit in the TLB most of the time, contrary to original scheme without the B$^+$-tree, where the binary search thrashes the TLB incurring expensive page walks. We could also use large or huge pages, but this alternative requires special privileges, manual configuration, or dedicated system calls, requirements inappropriate for general-purpose systems. Nevertheless, both alternatives can be combined with interleaving.

Interleaving and NUMA effects. In our experiments, we ensure the microbenchmarks allocate memory and run on the same socket, to avoid process migration across sockets and remote memory accesses. However, the idea of interleaved execution applies also to cases with remote memory accesses; interleaving could be even more beneficial, assuming there is enough work to hide the increased memory latency. We plan on analyzing interleaved execution in a NUMA context to determine the related bottlenecks.

7. CONCLUSION

Instruction stream interleaving is an effective method to eliminate memory stalls in index lookups and improve index join performance. However, prior techniques required significant effort to develop, test and maintain a special implementation for interleaved execution, alongside the original code, an effort that could prohibit the adoption of interleaved execution in a real database system.

In this work, we proposed interleaving with coroutines, a compiler-based approach for lookup implementations that can be executed both sequentially and interleaved. Leveraging language support, our approach is easy to implement for any kind of index lookup, which we have demonstrated with binary search and CSB$^+$-tree lookups. Finally, interleaving with coroutines exhibits similar performance to carefully optimized implementations of prior interleaving proposals, so we consider our proposal the first practical interleaving technique for real codebases.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, as well as Ismail Oukid, Eleni Tzirita-Zacharatou, Lucas Lersch, Frank Tetzel, Manos Karpathiotakis, Iraklis Psaroudakis, Stefan Noll, Georgios Chatzopoulos, and Mirjana Pavlovic for their valuable feedback on the paper. This work was supported by SAP SE, Walldorf, Germany.

9. REFERENCES

[1] C# Reference. `https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx` [Online; accessed 14-August-2017].

[2] Generators. Python Wiki. `https://wiki.python.org/moin/Generators` [Online; accessed 14-August-2017].

[3] Programming Languages – C++ Extensions for Coroutines. Proposed Draft Technical Specification ISO/IEC DTS 22277 (E). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf` [Online; accessed 14-August-2017].

[4] Transaction Processing Performance Council. TPC-DS Benchmark Version 2.3.0. `http://www.tpc.org/tpcds/` [Online; accessed 14-August-2017].

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: where does time go? In *Proc. VLDB*, pages 266–277, 1999.

[6] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), 2007.

[7] M. Colgan. Oracle Database In-Memory. Technical report, Oracle Corporation, 2015. `http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html`.

[8] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[9] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[10] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, June 2016.

[12] A. Kemper and T. Neumann. HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*, pages 195–206, 2011.

[13] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.*

[14] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. SIGMOD*, pages 339–350, 2010.

[15] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *PVLDB*, 9(4):252–263, 2015.

[16] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. ASPLOS*, 1991.

[17] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data Blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. *Proc. SIGMOD*, 2016.

[18] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL Server column stores. In *Proc. SIGMOD*, pages 1159–1168, 2013.

[19] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server column store indexes. *Proc. SIGMOD*, pages 1177–1184, 2011.

[20] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[21] J. Marusarz. Understanding how general exploration works in Intel VTune Amplifier XE, 2015. `https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe` [Online; accessed 14-August-2017].

[22] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[23] J. Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features.* Elsevier Science Inc., 2002.

[24] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, Feb. 2009.

[25] I. Müller, C. Ratsch, and F. Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *Proc. EDBT*, pages 283–294, 2014.

[26] M. Poess and D. Potapov. Data compression in Oracle. *Proc. VLDB*, pages 937–947, 2003.

[27] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: so much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[28] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *Proc. ACM SIGMOD*, pages 475–486, 2000.

[29] RethinkDB Team. Improving a large C++ project with coroutines, 2010. `https://www.rethinkdb.com/blog/improving-a-large-c-project-with-coroutines/` [Online; accessed 14-August-2016].

[30] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6th edition, 2012.

[31] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proc. VLDB*, pages 49–60, 2005.

[32] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proc. VLDB*, pages 405–416, 2003.