

Profiling Dataflow Systems on Multiple Abstraction Levels

Alexander Beischl
beischl@in.tum.de
Technical University of Munich

Timo Kersten
kersten@in.tum.de
Technical University of Munich

Maximilian Bandle
bandle@in.tum.de
Technical University of Munich

Jana Giceva
jana.giceva@in.tum.de
Technical University of Munich

Thomas Neumann
neumann@in.tum.de
Technical University of Munich

Abstract

Dataflow graphs are a popular abstraction for describing computation, used in many systems for high-level optimization. For execution, dataflow graphs are lowered and optimized through layers of program representations down to machine instructions. Unfortunately, performance profiling such systems is cumbersome, as today’s profilers present results merely at instruction and function granularity. This obfuscates the connection between profiles and high-level constructs, such as operators and pipelines, making interpretation of profiles an exercise in puzzling and deduction.

In this paper, we show how to profile compiling dataflow systems at higher abstraction levels. Our approach tracks the code generation process and aggregates profiling data to any abstraction level. This bridges the semantic gap to match the engineer’s current information need and even creates a comprehensible way to report timing information within profiling data. We have evaluated this approach within our compiling DBMS Umbra, showing that the approach is generally applicable for compiling dataflow systems and can be implemented with high accuracy and reasonable overhead.

CCS Concepts: • Software and its engineering → Data flow architectures.

Keywords: profiling, dataflow systems, query compilation

ACM Reference Format:

Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling Dataflow Systems on Multiple Abstraction Levels. In *Sixteenth European Conference on Computer Systems (EuroSys ’21)*, April 26–29, 2021, Online, United Kingdom. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3447786.3456254>

1 Introduction

Dataflow graphs are a powerful abstraction for a variety of applications and workloads: from more traditional systems like databases and compilers to more widely adopted computing frameworks for big-data [29, 32, 51], graph- and stream-processing [7, 44], and machine- or deep-learning [2, 40]. It allows developers to express the data dependencies between various tasks on a high abstraction level and map computations to (pipelines of) operators [43].

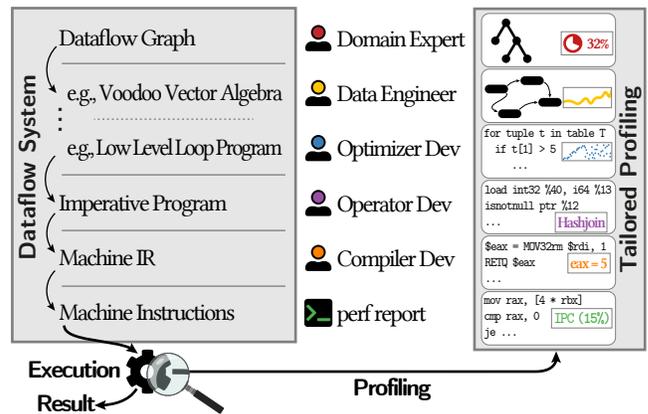


Figure 1. Layered organization of compiling dataflow systems on the left and profiling results of our novel Tailored Profiling approach on the right.

Using expressive abstraction layers allows the system stack to absorb the complexity of generating efficient code and mapping it onto the available hardware resources, as opposed to burdening the developer. In fact, compiling and code-generating dataflows are what many believe to be the only way to address the increasing heterogeneity of the underlying computing resources and allow domain-expert developers to focus on the important task at hand using a Domain Specific Language (DSL) at an abstraction level they are most comfortable and productive at, without having to worry about low-level details [8, 15, 23, 39, 42]. The key to this success is that the background-process involves progressive layering of optimization steps for dataflow graphs that generate lower-level intermediate representations (cf. Figure 1), which eventually lead to a high-performant and efficient binary program.

While this has many advantages, with each optimization layer/step we lose semantic knowledge about the (higher-level abstraction) dataflow so that some critical tasks, like debugging and performance profiling, become intractable. Most profiling tools used today primarily operate on a much lower level and report metrics on an assembly instruction- or function- granularity [3, 18, 25, 50]. While for systems experts the task to map information provided by these pro-

filers to self-written low-level code is rarely an issue, the problem becomes less trivial when someone needs to read performance profiling for machine-generated code and interpret it in terms of higher-level abstractions — especially since existing software systems are quite complex and involve many components that interact during the dataflow computation’s execution.

In this paper, we present how to performance debug and profile compiling dataflow systems with *Tailored Profiling* — in a way that brings value to any user working on a selected abstraction level. To achieve understandable profiling we analyze the state of the art to identify the reason behind the big semantic gap between the original dataflow graph and its subsequent transformations into lower abstraction levels (cf. Section 3). We then list key requirements a dataflow performance profiler should meet and present our high-level design in Section 4.

Inspired by how debug tools enrich the generated code with meta-data [11, 20], we propose extending the compilation steps to also annotate the generated code with meta-data, stored in a Tagging Dictionary that can be used to map the profiling results back to the desired abstraction level. To disambiguate samples on shared code locations to their respective caller, we introduce Register Tagging, a novel, lightweight alternative to call-stack sampling. This enables us to post-process the data and present it at a granularity that brings the best insights to the developer. The simplicity of our solution makes it applicable to any system that lowers the dataflow graph to generate Machine IR or native instructions for hardware platforms that support profiling with sampling (e.g., the CPU) [5, 8, 19, 29, 34, 46], provided that they run on a single (machine) node.

In Section 5, we detail the steps needed to build such a profiler with our prototype, integrated as part of our high-performance compiling DBMS *Umbra* [34]. As appropriate profiling is already challenging, the focus of the prototype was on single-machine multi-threaded CPU computations — leaving both distribution on multiple nodes and running on heterogeneous hardware targets (accelerators) as directions for future work. We discuss the benefits of our approach in the context of a few compelling use-cases and show that we can achieve good accuracy with moderate 38% overhead in Section 6. Eventually, we conclude, discuss benefits and limitations of our approach, and outline future work in Section 8.

2 Background

2.1 Code Generation

Dataflow systems express their computation on data with dataflow graphs, which are used for high-level logical optimization. The system then automatically restructures the graph to minimize the execution time. Dataflow graphs can either process the input data through their operators, e.g., by interpreting the generic operator code according to the

input data and dataflow configuration, or generate machine code just in time for each dataflow graph, thus removing any interpretation overhead.

Most systems organize machine code generation in a layered approach with multiple intermediate representations (IRs). Successive lowerings from dataflow graph to machine instructions allows for different optimization strategies to be applied to the corresponding layers that reorder and restructure the program to get better performance [8, 15, 23, 39, 51], as shown on the left of Figure 1. The topmost graph layer is translated into more concrete intermediate representations, which vary widely depending on the actual system. For instance, Voodoo [42] proposes a vector algebra to reason about data partitioning, instruction level, and thread parallelism, while TVM uses low-level loop programs to reason about control flow while still abstracting from a concrete hardware implementation [8]. Such IR levels are usually followed by imperative program representations that target specific hardware instructions. A particular effect of these optimizations is that when optimizations move code they often intertwine instructions from different operators — an effect commonly referred to as operator fusion.

2.2 Profiling Tools

To analyze the performance characteristics of complex computer systems and find tuning opportunities, developers rely on profiling tools [1, 14, 18, 25]. These tools output the system’s performance profile for a given workload and show the utilization of various micro-architectural hardware features. To do this, profilers use the processor’s Performance Monitoring Units (PMUs) to collect samples of selected hardware events (e.g., stalled CPU cycles, cache-misses, memory accesses, etc.) and map them to the assembly instructions that triggered them. To make the output more user-friendly, the profilers will often generate a performance report on a source line or function granularity.

Recently, Intel introduced the Processor Event-Based Sampling mode (PEBS) [17], where the processor itself records and writes samples into a dedicated in-memory buffer without raising an interrupt. This significantly improves the precision of the samples and reduces the overhead, as the kernel is only involved when the buffer is full. In such cases, the interrupt handler writes out the samples to memory and clears the buffer for further sampling. In default mode, PEBS just records the instruction pointer (IP) of the executed instruction at the sampling time-point, but one can also configure it to record the full call-stack.

3 Profiling Dataflow Systems

The challenge of profiling dataflow systems is very often an artifact of the complex compilation and optimization process that undergoes a series of transformation steps. Many of these optimization steps are designed by developers with different fields of expertise, so any information that identifies

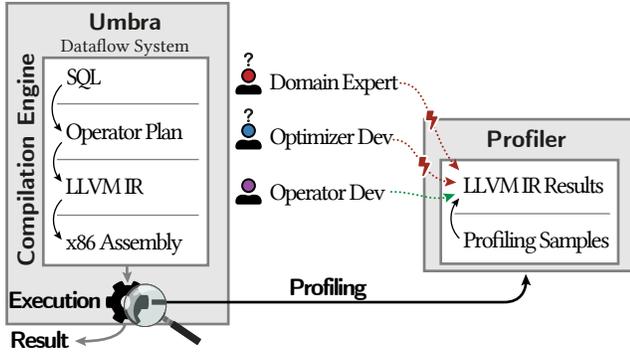


Figure 2. Layers of intermediate representation for the Umbra dataflow system. With today’s profilers, developers with expertise on different layers must all use profiling reports on the lowest IR level.

hotspots and bottlenecks in the system (e.g., where the time is spent, how operations interact, how efficiently operations use underlying resources, etc.) would be of great use.

However, with the current tools this task is not trivial. Even in the simple(r) case, where the dataflow runs on a single machine, does not rely on I/O for data exchange or synchronization, and only uses the CPU (does not offload computation to accelerators), the problem of mapping the low-level profiling detail to higher-level components and abstraction levels is a challenge. To understand the problem better, we make the following observations:

Profilers work on low-level IR. They operate on the executable and its libraries. As a result, the profiles they generate only aggregate the recorded events on assembly-level or source-line / function call granularity. While this is useful information for a low-level systems engineer, like the Operator Developer working with compilers and code generation, the data is too raw for anyone working with higher-level constructs and concepts (cf. Figure 2). They will have to reverse-engineer through multiple layers of code generation to find where these instructions originate from, a process that can easily become involved, ineffective, and error-prone.

Profiling reports overall statistics for an event. Profilers often fail to leverage the time dimension recorded along with the collected samples. This data would be useful, not only for performance tuning pipelines where multiple operations can be active at the same time, but also when provisioning resources to different operators at runtime (e.g., for streaming dataflow engines [27]).

Memory tracing is costly and done by another tool. Knowing the set of addresses accessed during program execution can be very valuable to developers. For instance, which data structure was accessed when most of the cache-misses are recorded and by which operator can help a developer choose a more suitable data structure, or be more careful with data partitioning among the executing threads.

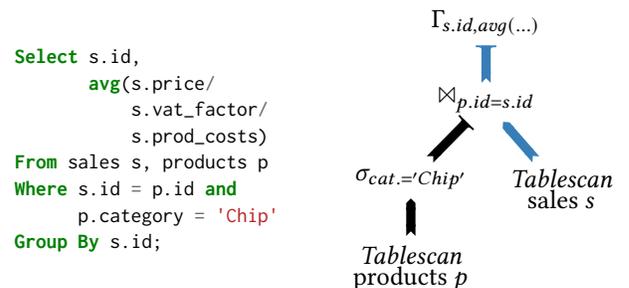
Typically, memory tracing is done on a system level, which comes with a big performance overhead, making it impractical, and in a format that maps the frequency of access requests to memory addresses, making it too raw for anyone working on higher abstraction levels.

There is a lack of a holistic solution. All of the above-identified limitations of existing profilers are because they operate completely decoupled from the rest of the compilation and optimization process (Figure 2). In fact, the whole focus during the lowering and optimization process is on generating highly optimized code. As a side-effect, we lose track of the higher-level components. For instance, in the step of lowering a database query plan to LLVM-IR, the code generator produces low-level loops that fuse multiple operators together, thereby losing the abstraction concept of operators per-se and the dependency between them. As a result, profilers cannot re-establish the link because the boundaries of the higher-level components are often blurred in the IR of lower levels, which is also why profiling dataflows on multiple abstraction levels becomes such a puzzle for anyone.

3.1 Profiling Example

To make things more clear, let us walk through an example that highlights the different steps needed to identify a potential bottleneck in our DBMS Umbra that generates machine code to achieve maximum in-memory processing speed.

As many other dataflow systems, Umbra progressively lowers each user’s request (i.e., query) through a series of optimization steps. The query in Figure 3a, for example, is first parsed and then internally represented as the dataflow graph in Figure 3b. The dataflow graph is then lowered to an imperative program (i.e., into LLVM IR, the intermediate representation of the LLVM optimizing compilation framework [22]). LLVM then lowers the IR program down to executable machine code.



(a) Example query in SQL (b) Dataflow graph for the query.

```

1  for each tuple  $t_1$  in sales s
2      if  $t_1$  has match in  $\bowtie_{p.id=s.id} [t_1.id]$ 
3          store  $t_1$  in hashtable of  $\Gamma_{s.id}$ 
    
```

(c) Pseudo-code for the execution of the blue pipeline of Figure 3b.

Figure 3. Example query with corresponding dataflow graph and generated code.

```

1 | loopTuples:
2 | 0% | %localTid = phi [%1, %loopBlocks %2, %contScan]
3 | 0.1% | %3 = getelementptr int8 %state, i64 320
4 | 0.1% | %4 = getelementptr int8 %3, i64 262144
5 | 2.2% | %5 = load int32 %4, %localTid
6 | 2.3% | %7 = crc32 i64 5961697176435608501, %5
7 | 1.5% | %8 = crc32 i64 2231409791114444147, %5
8 | 1.2% | %9 = rotr i64 %8, 32
9 | 2.3% | %10 = xor i64 %7, %9
10 | 2.2% | %11 = mul i64 %10, 2685821657736338717
11 | 1.2% | %12 = shr %11, 16
12 | 2.4% | %13 = getelementptr int8 %5, i64 %12
13 | 32.1% | %14 = load int32 %40, i64 %13
14 | 0.2% | %15 = isnonnull ptr %12
15 | 0.3% | condbr %15 %loopHashChain %nextTuple
16 | loopHashChain:
17 | 0.1% | %hashEntry = phi [%12, %loopTuples %99, %contProbe]
18 | 0.2% | %16 = getelementptr int8 %hashEntry, i64 16
19 | 1.1% | %17 = load int32 %16
20 | 0.3% | %18 = cmpeq i32 %5, %17
21 | 0.2% | condbr %18 %else %contProbe
22 | else:
23 | 0.5% | %19 = getelementptr int8 %0, i64 786432
24 | 2.2% | %20 = load int32 %19, %localTid
25 | 9.8% | ; ... // load values %22, %24, %26
26 | 9.5% | %27 = sdiv i32 %22, %24
27 | 9.6% | %28 = sdiv i32 %27, %26
28 | 2.9% | %30 = crc32 i64 5961697176435608501, %20
29 | 2.4% | %31 = crc32 i64 2231409791114444147, %20
30 | 1.3% | %32 = rotr i64 %31, 32
31 | 1.4% | %33 = xor i64 %30, %32
32 | 2.3% | %34 = mul i64 %33, 2685821657736338717
33 | 1.7% | %35 = and i64 %34, 1023
34 | 1.9% | ; ... // find entry
35 | 2.2% | store int32 %20, %37
36 | 0.2% | %38 = getelementptr int8 %37, %4
37 | 2.1% | store int32 %28, %38
38 | ...

```

Listing 1. Performance profile of the actually generated program in LLVM IR for the blue pipeline of Figure 3b.

Before discussing performance profiles of the generated code, let us briefly inspect the structure of the generated code. The operators of Figure 3b marked in blue form a pipeline of operators that directly pass tuples to each other during execution. Conceptually, the system generates the pseudo-code of Figure 3c, where the scan operator loops over the tuples of the input table (Line 1), passes each tuple to the join operator (Line 2), which in case of a match forwards the tuple to the aggregation operator (Line 3). In reality, however, the system produces the detailed LLVM IR shown in Listing 1.

Now, when profiling the example query, the profiler will report the results on line- or function-level of the IR program as shown in Listing 1. Each line is annotated with the number of collected samples the profiler attributes to the corresponding source line. This approximates the execution cost of each instruction. Observe how this profile view is rather low-level. At first glance, it is apparent that a significant amount of time is spent on the load instruction in Line 13. However, it takes quite some time and expertise to realize

that this instruction implements the directory lookup of the chaining hash table used in the join operator. Further, it is easy to miss that in total an even higher number of samples (50%) belong to the aggregation operator, whose samples are spread out over Lines 23–37. In short, the initial impulse to focus on improving the join operator would miss the fact that the aggregation operator is the main bottleneck.

Unfortunately, a report of samples on a function level — as most profilers offer — does not remedy the situation either. Operator fusion tightly couples operators of the whole pipeline into a single function, leaving the function aggregation level too coarse to obtain any useful insights. Due to the coupled operators, we cannot apply evident approaches such as generating each operator instance’s code in a separate source file or emitting instructions to update timers on entry and exit from operators to derive high-level profiling information. Additionally, neither the function nor the source-level view lend themselves to visualize a time dimension.

4 Abstraction Appropriate Profiling

As shown in the previous section, today’s profilers present reports mainly on the lowest abstraction level. This covers only a fraction of the information needs of the different experts involved in building dataflow systems. Here, we present our profiling approach that caters to everyone involved.

We list the desired features in Section 4.1, propose a solution in Section 4.2, and present its advantages in Section 4.3.

4.1 Requirements from an Ideal Profiler

A profiler should report results at a *granularity familiar to the reader* of the report. Specifically, the report should be in terms they already use while interacting with the system. Such terms could be operators from the dataflow graph or vectors, loops, etc., from lower optimization layers.

While these terms can be quite high-level, the profiler should *not hide details* due to aggregation. Information that is available in profiling samples, e.g., timestamps, accessed memory addresses, etc., should be presented to the reader.

Beyond the right format, a profiling report should also accurately reflect the behavior of the executed computation. This means, first, *association* of samples with high-level components must be *correct*. Second, the sampling *frequency* must be *high* enough to not miss any behavior, e.g., due to aliasing effects, where frequent short running components are not recorded sufficiently. Third, the performance *overhead* of sampling should be *low*, so that the behavior of the profiled process can be observed undisturbed.

In the next section, we present a profiler that meets these demands. Our solution relies on hardware profiling support to supply accurate, low-overhead samples with instruction pointers and timestamps and requires that each low-level component can be mapped to its next-higher abstraction level component (cf. Section 4.2.5).

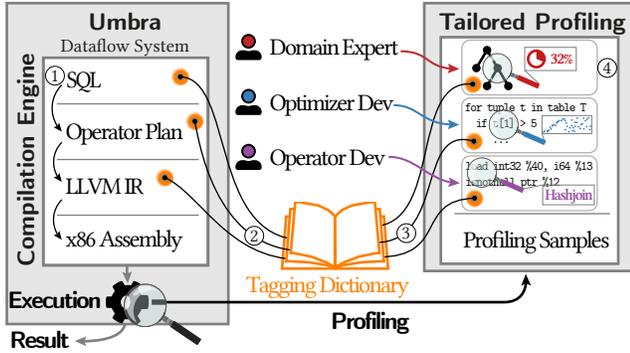


Figure 4. Tailored Profiling requires small extensions to collect a Tagging Dictionary during code generation and enable Register Tagging. With this, it can generate high-level performance reports for all parties involved.

4.2 Tailored Profiling

With Tailored Profiling, we bridge the semantic gap between the low-level results traditional profilers produce and the developers’ need for reports on higher abstraction levels. Tailored Profiling supports all requirements listed in Section 4.1 and requires no conceptual changes of the dataflow system.

4.2.1 Solution Overview. Tailored Profiling solves traditional profilers’ shortcomings by tracking lineage of the low-level Machine IR code generation across the many compilation steps to enable linking profiling samples to higher abstraction levels. Our approach, shown in Figure 4, achieves this by ① tracing the links between components of the different abstraction levels throughout the lowering process and ② storing the links for each lowering step in Tagging Dictionary logs. After profiling, ③ a post-processing phase uses the Tagging Dictionary to annotate the collected profiling samples with abstraction information, e.g., operators, and ④ produce a profile meeting the needs of the selected developers depicted in Figure 4.

This solution works for dataflow systems that undergo multiple lowering steps to generate code from a dataflow graph.¹ To do that, the system uses a single code generator to lower the dataflow graph to Machine IR and then compiles Machine IR to native instructions with a second code generator as we do in Umbra. Otherwise, we can perform all lowering steps down to machine instructions within a single code generator.

4.2.2 Tagging Dictionary. The Tagging Dictionary is populated during the lowering of the dataflow graph at compile time and consists of multiple logs (e.g., hash tables), one for each lowering step as illustrated in Figure 5. Each log is filled during its respective lowering phase ①, and contains an entry for each lower-level component that links it to the corresponding component on the next-higher level ②. During the first lowering step in Figure 5, *Log A* is populated and links each source location² of the imperative program to its operator, while *Log B* is filled during the second lowering step. The logs store entries as key-value pairs, called *links*, with the lower-level component as key and the higher-level counterpart as value. Thus, the post-processing phase can map native samples bottom-up to the required abstraction level(s) using the Tagging Dictionary ③ to provide profiling results on different levels ④. To capture the links, the system’s compilation engine keeps track of the currently lowered (active) component of the lowered level using an Abstraction Tracker (cf. Section 4.2.4) and adds an entry to the Tagging Dictionary’s corresponding log whenever a lower-level component is created.

¹Our highest-covered abstraction level is the dataflow graph, which most dataflow systems already use to reason about query execution. Thus, it is commonly known by domain experts and developers, and suited to explain the procedural execution of DSL queries.

²Source locations refer to the imperative program code for which the system emits Machine IR instructions (cf. Figure 3c and Listing 1).

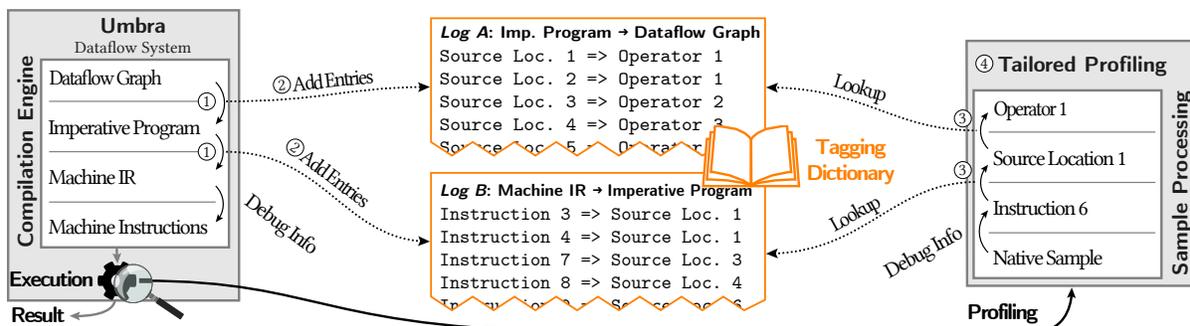


Figure 5. Tailored Profiling applies the Tagging Dictionary to report the profiling results on higher abstraction levels. The Tagging Dictionary is populated during query compilation. *Log A* links the source locations (Figure 3c) to their operators (Figure 3b), while *Log B* links Machine IR instructions (Listing 1) to their source locations. After execution, the profiler uses the Tagging Dictionary to map the native samples to higher abstraction levels. The circled numbers match the numbers in Figure 4.

4.2.3 Lowering Machine IR to Machine Instructions.

Tailored Profiling links the abstraction levels from the dataflow graph down to Machine IR with one Tagging Dictionary log for each lowering step, as shown in Figure 5. Most systems use a compiler for the last lowering step from Machine IR to machine instructions that already provides debug information (e.g., DWARF format), which we also use in our Tailored Profiling’s prototype in Umbra to link these two levels. In cases where the dataflow system itself performs the last compilation step, one needs to add an additional Tagging Dictionary log that links the machine instructions to Machine IR.

4.2.4 Abstraction Trackers. During each lowering step, Tailored Profiling uses an Abstraction Tracker to monitor which higher-level component is currently lowered to the next-lower level. The Abstraction Tracker is an auxiliary structure, e.g., a pointer or stack, storing the currently lowered higher-level component. For our running example from Figure 5, the Abstraction Tracker of the first lowering step points to the active operator. Thus, whenever the compilation engine creates a lower-level component in the lowering step, it can determine its higher-level counterpart by checking the Abstraction Tracker and storing the link in the Tagging Dictionary’s log. In Figure 5, Tailored Profiling uses two Abstraction Trackers: one to track the active operator during the first lowering step and a second one to track the active source location when lowering the imperative program to machine instructions.

4.2.5 Challenges with Shared Source Locations. The Tagging Dictionary implicitly makes the assumption that every lower-level component is generated by exactly one higher-level component, i.e., every source location in the generated code belongs to exactly one operator in the dataflow graph. Hence, we can map every profiling sample to one source location, and thus, to exactly one operator.

While the assumption is true for most of the generated code, it is still possible that two operators share a source location. This happens, for example, in Umbra’s join operator. It calls a pre-compiled `insert` function to add entries to a hash table. Two instances of the join operator will then share all source locations of the pre-compiled function. Yet, any given profiling sample must be attributed to only one of the two operators. Thus, we need to disambiguate shared source locations. This can either be achieved with *call-stack sampling* or our novel *Register Tagging* approach, both of which we discuss next.

Call-Stack Sampling. The default approach on how a profiler can disambiguate shared source locations is using call-stack sampling that records the entire call-stack with each sample. Having the call-stack stored in the sample can then help us identify the higher-level component for each function that executes the shared source location. Thus, when

```

1  ...
2  prevValue = setTag(op1); // set op1 as currently active
3  insert(); // call shared code location
4  setTag(prevValue); // reset to previously active op
5  ...

```

Listing 2. Register Tagging uses a processor register to trace the component that calls the shared code location. The register is reserved for exclusive use by Register Tagging.

Tailored Profiling encounters a sample with shared source locations, it traverses the call-stack to identify the active components for each ambiguous abstraction level. Then, it links the sample to all other abstraction levels with the Tagging Dictionary and the disambiguated components. The major drawback of this approach is its cost. It suffers either from high performance overhead or is limited to a low sampling frequency (cf. Section 6). The positive aspects are that it can be applied without any alteration of the generated code and when hardware support for Register Tagging is not available.

Register Tagging. As an alternative, we propose a novel light-weight approach that we refer to as Register Tagging. The key idea is to disambiguate the shared source location by storing a tag in a machine register (tag register) that identifies the active component. During sampling, the profiler records the register values along with a profiling sample, as modern x86 processors have the ability to record machine registers in the samples. Tailored Profiling can then use the tag to disambiguate source locations for profiling samples containing a shared location.

Linking back to our example, where two joins share the function `insert`, just before the first join calls the common function, the Register Tagging would generate code that moves the tag for the first join into the tag register (shown in Listing 2). Note that on setting the tag in Line 2, we remember the previous value of the tag in order to reset the value after the function call (Line 4). Thereby, Register Tagging can also handle nested shared code locations. Register Tagging also instructs the compiler to not use the tag register for any other purposes to avoid overriding the value. Finally, when a profiling sample is taken from the `insert` function, the value of the tag register is also captured so that we can uniquely identify the caller and map the sample to all abstraction levels with the Tagging Dictionary.

The disadvantage of Register Tagging is that it relies on hardware profiling support to also capture register values, which is not possible for dataflow systems that run in managed runtime like JVM. A second disadvantage is its invasiveness with respect to the code generation engine, because it leaves the generator with a register less to work with. It is important to note the small amount of changes to the code generation process, compared to a significant reduction in the overhead compared to call-stack sampling without compromising accuracy.

So far, we described how Register Tagging works for only one abstraction level with shared source locations. However, it can also cover them on multiple abstraction levels. To do this, 1) one needs to reserve a machine register exclusively for each abstraction level with shared source locations and 2) propagate awareness of higher-level components with a shared source location through the progressive lowering steps. For example, when the system uses a dataflow operator with a pre-compiled function (shared source location), it has to pass this information through all progressive lowering steps down to code generation. The code generation then encloses the sections of Machine IR code descending from this pre-compiled function with Register Tagging to write the tag into the operator level’s tag register and disambiguate it at runtime. If one wants to optimize the number of reserved registers while keeping the performance overhead low, one can place tags of multiple levels into a single machine register, e.g., by splitting it into chunks of 8-bit or 16-bit instead of using an entire register per level or even choosing each level’s chunk size accordingly to its number of operators.

4.2.6 Generating Tailored Profiling Reports. Applying Tailored Profiling, the profiler aggregates samples at the abstraction level that meets the developer’s needs. Therefore, the profiler processes the samples and maps them to the needed higher abstraction levels in a bottom-up approach using the Tagging Dictionary, as illustrated in Figure 5.

To map a sample containing a machine instruction to the dataflow graph, the profiler proceeds as follows: At first, Tailored Profiling uses debug information to map the sample’s instruction to its Machine IR instruction, for this example Instruction 7. Then, the profiler looks up the entry of Instruction 7 in the Tagging Dictionary’s *Log B* to map it to its imperative program component, which is Source Loc. 3. Now, the profiler can look up the dataflow graph operator of Source Loc. 3 in *Log A* to map the sample to Operator 2. For samples containing instructions from shared source locations, the profiler first retrieves the active component either from Register Tagging or the call-stack sampling.

Tailored Profiling also supports iterative dataflow graphs, although the Tagging Dictionary cannot differ between iterations. Therefore, the post-processing phase uses the samples’ timestamps to detect iterations and distinguish between iterations.

4.2.7 Optimization Techniques. The compilation engine populates the Tagging Dictionary during the lowering phase and applies Register Tagging around shared code locations. In doing so, Tailored Profiling also covers common optimizations applied during the lowering phase and adapts accordingly. Here, we describe how it can handle almost any optimization (cf. Table 1) the compilation engine performs itself or the ones for which it can track the optimized instruction’s origin (if performed externally) and briefly cover the ones where it fails to capture the links.

Table 1. Tailored Profiling supports common optimization transformations when lowering to Machine IR. Umbra supports Tailored Profiling for all implemented optimizations (cf. Section 5.4).

Optimization	Tailored Profiling	Umbra
Operator fusion	☑	✓
Instruction fusing	☑	×
Code elimination	☑	✓
Constant folding	☑	✓
Common subexpression elimination	☑	✓
Loop unrolling & interleaving	☑	×
Polyhedral optimizations	☑	×
Dataflow graph operator fusion	☑	✓
Common abstraction for heterogeneous accelerators	☐	×

☑ supported ☐ not yet supported *by Tailored Profiling*;
 ✓ implemented × not implemented *in Umbra*

Supported optimizations. Tailored Profiling implicitly handles *operator fusion* by linking the components of different abstraction levels during the lowering steps. Thus, we can look up the code generating component of each Machine IR instruction in the Tagging Dictionary and subsequently map it back to the unfused operator using the logs.

The Tagging Dictionary covers Machine IR *instruction fusing* by updating its log entries accordingly. For instance, if we fuse Instruction 7 and Instruction 8 from Figure 5, then we remove both links from *Log B* and add a new link to Source Loc. 3 and Source Loc. 4 because the fused instruction belongs to both higher-level locations.

Code elimination does not require any changes, since the eliminated Machine IR instructions will not appear in the profiling samples; however, we can still remove them from the Tagging Dictionary.

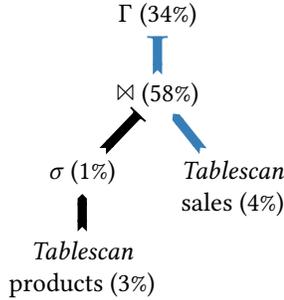
Constant folding is solely a compile-time operation; we just apply code elimination to remove the original instructions.

Common subexpression elimination is equivalent to shared source locations; thus, we handle it with Register Tagging.

For *loop unrolling & interleaving*, we trace each Machine IR instructions’ origin during the optimization and update the Tagging Dictionary *Log B* accordingly. Loop control flow instructions are attributed to the operators of all original control-flow instructions, identical to instruction fusion.

Polyhedral optimizations applied to vectorized execution can be handled similar to loop unrolling & interleaving. During the transformation we track which part of the transformed code, e.g., control flow structures and vector operations, belongs to which original operator(s) and update the Tagging Dictionary’s *Log B* accordingly.

If operators are fused at a higher abstraction level, i.e., *Dataflow graph operator fusion*, we track which parts of the fused operator correspond to which original operator. In



(a) Query plan from Figure 3b annotated with each operator’s costs.

```

1      loopTuples:(tablescan 2.4% hash join 45.7%)
2      ...
13 32.1% %14 = load int32 %40, i64 %13
14 0.2% %15 = isnonnull ptr %12
15 0.3% condbr %15 %loopHashChain %nextTuple
16      loopHashChain: (hash join 1.9%)
17 0.1% %hashEntry = phi [%12, %loopTuples...]
18 0.2% %16 = getelementptr int8 %hashEntry, ...
19 1.1% %17 = load int32 %16
20 0.3% %18 = cmpeq i32 %5, %17
21 0.2% condbr %18 %else %contProbe
22      else: (group by 50.0%)
23 0.5% %19 = getelementptr int8 %0, i64 786432
24 2.2% %20 = load int32 %19, %localTid
25 9.8% ; ... // load values %22, %24, %26
26 9.5% %27 = sdiv i32 %22, %24
27 9.6% %28 = sdiv i32 %27, %26
28      ...

```

(b) Excerpt of the performance profile from Listing 1 extended using the data from the Tagging Dictionary. Note, the percentages are based only on the samples of the blue pipeline.

Figure 6. Tailored Profiling provides the profiling reports on developers’ abstraction levels.

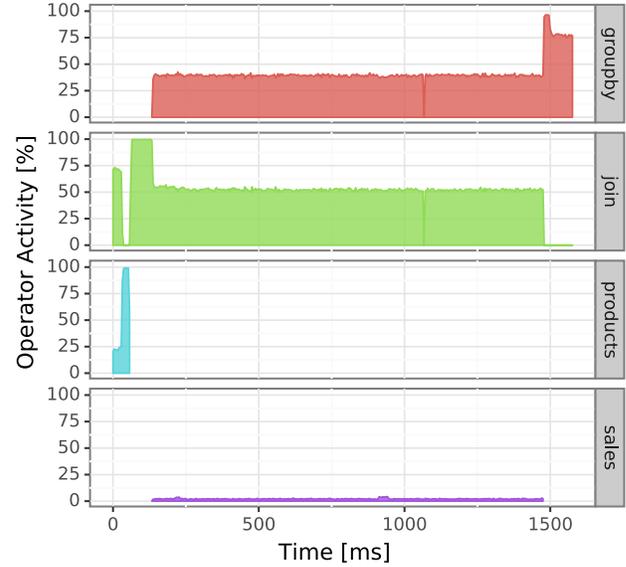
Umbra we implement this, e.g., for the groupjoin [31] by distinguishing between the group by and join sections inside the fused operator (cf. Section 5.4 for technical details).

Unsupported Optimizations. Tailored Profiling can handle optimizations performed by the compilation engine itself. If an external compiler performs the optimizations, Tailored Profiling needs to track the lineage between input and optimized output instructions. For example, when relying on external compilers like LLVM for lowering Machine IR to machine instructions, Tailored Profiling has to rely on the provided debug information to track optimizations.

Although our prototype is implemented in Umbra, a CPU-only system, we are certain that the concept of Tailored Profiling can be extended to *common abstraction for heterogeneous accelerators*. We expect combining profiling results of different hardware types as well as covering I/O latencies to be the main challenges.

4.3 Benefits and Limitations of Tailored Profiling

We conclude Section 4 showing benefits and limitations of Tailored Profiling.

**Figure 7.** Tailored Profiling associates each sample with an operator and thus determines operator activity over the query runtime.

Benefits. To show our approach’s advantages and practical impact, let us revisit the example from Section 3.1.

For the domain expert, the profiler maps the samples to the dataflow graph, in this example the query plan, and aggregates them per operator as shown in Figure 6a. The domain expert can then inspect the annotated query plan to learn about the costs of each operator, derive decisions to reconfigure the database system, and fine-tune SQL queries.

For the optimizer developer, the operator plan is also a familiar abstraction. They can compare the profiling results of different query plans for the same query to evaluate the cardinality estimates of the optimizer and refine the query plan optimizations.

The operator developer – even though they are familiar with the low-level results of the IR program – still benefits from Tailored Profiling. The profiler enriches the profiling results, as shown in Figure 6b. It annotates each instruction with its operator and aggregates the costs of each operator on different granularities, e.g., on basic blocks and functions. Thereby, the costs of each operator are provided as a frame of reference to avoid missing expensive operations distributed across multiple instructions.

Aggregating to appropriate levels enables an additional, crosscutting feature. The components from each level provide an ideal base to visualize the performance profile over time. For example, the profiler can show operator activity over time, as shown in Figure 7. The operator developer can inspect this to learn about the interaction between operators and detect temporal hotspots. Then they can use the profiler to narrow down on the next lower abstraction level, i.e., limit the results to the time interval of the hotspot. With

visualization over time, developers can pinpoint bottlenecks that would otherwise be hidden in aggregation.

Limitations. However, Tailored Profiling’s capabilities are still limited by the hardware used. For example, when profiling Umbra’s execution with PEBS (CPU profiling), the CPU cannot record data while it is blocked, e.g., due to heavy disk I/O, network contention, or memory latencies.

Furthermore, if the dataflow system uses an external code generator, like LLVM, to compile Machine IR to machine instructions, the generator must provide meta-data to map the samples to Machine IR instructions. This can be achieved either with debug information like DWARF or by adding Tagging Dictionary support to the external code generator, which we discuss in Section 8.

5 Integration into Umbra

We implemented Tailored Profiling in the compiling dataflow system *Umbra* to demonstrate its feasibility and advantages. In this section, we discuss the implementation details of our prototype.

Umbra is a high-performance relational database system, which compiles queries with data-centric code generation based on the produce & consume model [30, 33, 34, 51]. Umbra’s query engine is implemented in C++ and lowers dataflow graphs from relational operators (8a) through pipelines of tasks (8b) and LLVM IR (8c) to machine instructions (8d). Thus, the engine runs queries by executing native instructions, which allows the profiler to directly use hardware features, such as PEBS, to collect samples.

5.1 Umbra’s Compilation Phase

Umbra’s query engine compiles the dataflow graph in three progressive lowering steps (Figure 8).

First, the engine splits the dataflow graph at its tuple materialization points to lower the relational operators to a *pipeline* abstraction [33] (or stages as used by [13]) and applies the *operator fusion* optimization. Figure 8b’s dataflow graph has to materialize at the join’s build, the group by’s materialize and aggregate, resulting in three pipelines. Each pipeline contains all *tasks* [13] of operators between the materialization points. The tasks of materializing operators can be split across multiple pipelines, e.g., the join’s build and probe task.

In the second lowering step, *code generation*, the query engine compiles each pipeline of tasks into tight-loops of LLVM IR instructions (operator fusion), illustrated as blocks in Figure 8c. For example, the pipeline containing the join’s probe is translated to a program similar to Listing 1.

Finally, in the third step, the query engine *compiles* the LLVM IR instructions to an executable of native instructions using the LLVM compiler framework [22] before executing it to process the query.

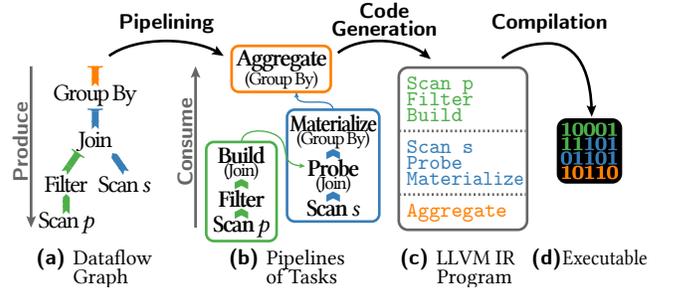


Figure 8. Umbra’s execution model compiles the dataflow graph in 3 progressive lowering steps to native instructions.

5.2 Populating the Tagging Dictionary

As introduced in Section 4.2, Tailored Profiling links the components of different abstraction levels with the Tagging Dictionary during the lowering phase. In the case of Umbra, we use two Tagging Dictionary logs. The first log links tasks to their operators during the first lowering step, while the second log links LLVM IR to tasks in the second step. Keeping track of the higher-level component (i.e., relational operators during the first lowering step or pipeline tasks during the second step) is done with two Abstraction Trackers and is integrated within the lowering steps. For the third step, compilation, Umbra uses debug information to link native instruction to LLVM IR.

Umbra’s code generation process is based on the produce & consume model. In produce & consume, each operator is responsible for generating the code that implements the operator’s functionality. When operators are composed into an operator tree, e.g., as shown in the dataflow graph of Figure 8a, they need to pass tuples among each other. This happens through the interface of produce and consume functions. An operator A can ask its input operator B to produce tuples (i.e., generate code that produces tuples) by calling B’s produce function. Operator B generates code that prepares a tuple and then passes that code to A’s consume function. Thus, the compilation engine traverses the operator tree in depth-first order, as operators use the produce function to delegate producing tuples down the tree until we reach a leaf node (e.g., scan operator), before traversing back up by invoking the corresponding consume functions.

Lowering Relational Operators to Tasks. The first lowering from operators to pipelines of tasks is done inside the operators’ produce function. In the produce function, each operator registers its task for the active pipeline and, in the case of materialization, starts the new pipeline and task. For example, the join operator consists of two tasks, Build and Probe. First, it has to build a hash table for the tuples of Scan *p*, and then it probes the hash table with each tuple of Scan *s* to join them. Thus, the join operator first registers the Probe task, then starts the left pipeline and registers the Build task.

When registering a task, Tailored Profiling checks the active operator with the Abstraction Tracker and adds a link for the task to the Tagging Dictionary log.

Lowering Tasks to LLVM IR. After reaching a leaf operator (e.g., Scan p), we initiate the second lowering step (code generation by calling the consume function). Starting with this leaf node, each operator now first executes its own consume function and afterwards calls its parent's consume. Inside the consume function, Umbra triggers the operator's registered task(s) that generate the LLVM IR code implementing the task's functionality. The generated code of all tasks then forms the LLVM IR program, as shown in Figure 8c.

When a task generates an LLVM IR instruction, Tailored Profiling checks the active task with the second Abstraction Tracker and adds a link to the second Tagging Dictionary log, linking the LLVM IR instruction to its task. After all the tasks have finished generating their code, the second lowering step is complete and the Tagging Dictionary is fully populated.

Lowering LLVM IR to Native Instructions. In the final step, Umbra compiles the generated LLVM IR program to an executable of native instructions with the LLVM compiler framework and utilizes the debug information generated by LLVM to link native instructions to their LLVM IR instructions.

Even though the described procedure seems to require many changes to the lowering phase, this is not the case. In Umbra, the produce, consume, task registration, task triggering, and instruction generation are all funnelled through a single code location, which we use both to update the Abstraction Trackers and to populate the Tagging Dictionary.

5.2.1 Abstraction Trackers. During the lowering process, Tailored Profiling always keeps track of both the currently active operator and the task with its two Abstraction Trackers. The active operator only changes when either produce or consume is called. More specifically, on entry of either function we update the operators' Abstraction Tracker to the called operator and reset to the previous operator on exit. A task is active and generates code after being triggered by the consume function. Thus, Tailored Profiling updates the tasks' Abstraction Tracker whenever a task is triggered and resets it when the task is done generating code.

The Abstraction Trackers are implemented in Umbra as a stack, where the active operator and task are always on top of their stack. For example, to track the active operator, we push each operator onto the stack when accessing it with produce and remove it on the last visit with consume.

5.2.2 Tagging Dictionary. Umbra's Tagging Dictionary consists of two logs: one that links tasks to operators and one that links LLVM IR instructions to tasks. Both logs are populated during the respective lowering phase. Each log is

implemented as a hash table, with the lower level's components as keys and the higher level's as values. For instance, we use the unique LLVM IR variables (SSA-form) as keys to map LLVM IR instructions to tasks. At the end of the compilation phase we write all logs into a meta-data file, which is read by the post-processing phase to map the samples bottom-up to the abstraction levels' components.

5.3 Register Tagging

Umbra applies Register Tagging to attribute samples of shared source locations at LLVM IR level to their correct tasks. Therefore, the system guards each call to a shared source location with inline assembly instructions that execute the tagging.

Let us pick up the example from Listing 2 to show how it works. Umbra includes the `insert` into the generated code of a task by generating a function call instruction. Register Tagging is applied by adding inline assembly instructions implementing `setTag` before and after the call instruction. These inline assembly instructions extract the register's previous value and write the active task's tag into the register.

The system ensures only Register Tagging alters the used register by removing it from allocation in the compilers. Umbra itself is compiled with `gcc` and the system uses the LLVM compiler framework to lower the generated code from LLVM IR to native instructions. For `gcc`, the system reserves the register using the `-ffixed` flag and we have modified the LLVM compiler framework to exclude it as well. Only the inline assembly instructions of Register Tagging can therefore access the register.

5.4 Implemented Optimizations

Umbra applies *operator fusion* during the lowering steps and keeps track of the links between the abstraction levels using the Abstraction Trackers, Tagging Dictionary, and Register Tagging as described in Section 5.1.

Code elimination and *constant folding* do not require updating the Tagging Dictionary in Umbra. Both optimizations are applied at the LLVM IR level and eliminate the original instructions. Thus, the profiling samples will not contain the eliminated instructions and we never look up their entries in the Tagging Dictionary. *Common subexpression elimination* is handled identically to shared source locations. The compilation engine frames each LLVM IR call to the shared expression with Register Tags to determine the caller at runtime. Umbra applies *dataflow graph operator fusion* to combine suited operators into a more efficient physical operator, e.g., `group by` and `join` might be fused to a `groupjoin` [31]. Tailored Profiling supports this by tracing the original operators' sections within the `groupjoin` on task level, i.e., we update the Abstraction Tracker for tasks to `groupjoin-join` when entering the join sections and to `groupjoin-groupby` for

partitioning and aggregation. Thus, we can map LLVM IR instructions through the task level back to the original dataflow graph operators.

5.5 Precise Timestamps for Profiling Samples

Tailored Profiling requires profiling samples with a reliable timestamp to report results with a time dimension. Umbra therefore uses the Linux kernel’s perf API [26] to record profiling samples with PEBS.

However, the samples’ timestamps provided by the Linux kernel have a bug and therefore do not represent the sampling time point correctly, as we observed. Instead of the existing timestamp, we use the processor’s Timestamp Counter (TSC) [17]. The TSC has cycle-grained resolution and is already collected in PEBS samples of processors since Skylake, though currently dropped by the kernel during sample formatting. We therefore modified the Linux kernel with a workaround to include the TSC in the formatted samples and convert it to *ns* using a kernel module [38].

6 Evaluation

In this section we evaluate the advantages of Tailored Profiling as well as its accuracy and runtime overhead.

Tailored Profiling’s major feature is to produce profiling reports at the right abstraction level for the developer, which is hard to quantify and very subjective. Thus, instead of success metrics, we show the value of Tailored Profiling with use cases for different users. Afterwards, we evaluate its accuracy and the induced overhead in Sections 6.2 to 6.3.

Experimental Setup. We used the TPC-H benchmark [49] with a scale factor of 1 (dataset size 1 GB) for the use-cases, and scale factor 10 (dataset size 10 GB) to measure performance and accuracy. Umbra and Tailored Profiling support multi-socket and multicore execution. However, we executed all queries single-threaded with Umbra for experimental clarity, e.g., to avoid locking and other side-effects. The use-cases were conducted on a machine with an Intel Core i7-7700K running at 4.2 GHz (turbo boost of 4.5 GHz), 32 GB DRAM and Ubuntu 19.10. The performance experiments’ test machine had an Intel Core i9-9900X with 3.5 GHz (turbo boost of 4.4 GHz), 64 GB DRAM and Ubuntu 20.04. We used Linux perf version 5.2 [25] to profile with PEBS, disabled sample throttling and handed the samples to Tailored Profiling with `perf script`. To profile costs and operator activity, we used the `INST_RETIRED.PREC_DIST` event and recorded a sample every 5000 events. For memory access patterns, we used the `MEM_INST_RETIRED.ALL_LOADS` event and captured a sample all 1000 loads.

6.1 Use Cases

We begin the use cases with the domain expert and proceed with the optimizer developer and the operator developer.

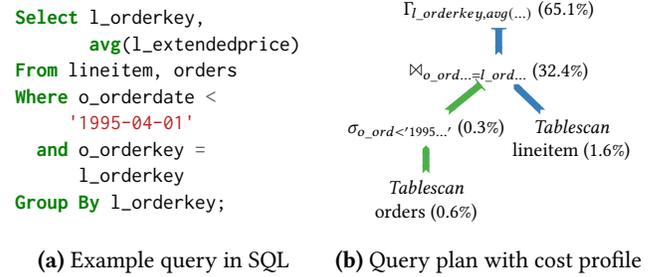


Figure 9. Tailored Profiling can aggregate samples up to query plan level — a concept database users are familiar with.

Domain Expert. In the first use-case, a user of Umbra investigates why the query from Figure 9a runs slower than expected.

At a familiar abstraction level, Tailored Profiling enables the user to view how much compute time each operator takes, as shown in Figure 9b. Here, they can quickly grasp the overall execution plan for the query. The report reveals that 65% of the runtime is spent in the aggregation operator and 32% in the join operator.

To speed up the query, the user can now make an informed decision on whether to, e.g., introduce index structures to reduce the cost of the join computation. Alternatively, they may decide to take computational shortcuts and add a sampling operator to reduce the number of tuples that reach the aggregation operator.

Note that most database systems have a feature that seemingly offers the same view. The `EXPLAIN ANALYZE` command counts how many tuples each operator processes and visualizes the statistics in an operator tree. However, even though the tuple count is a decent approximation, our sampling approach captures the actual time spent in each operator.

Optimizer Developer. As a second use-case, we inspect the work of an expert in Umbra’s optimizer. They investigate the performance of a query with the two alternative plans, as shown in Figure 10. Both plans have identical intermediate result sizes, so with the standard cost function the optimizer could choose either plan. Choosing the left one (Figure 10a) seems like a good option as the query plan first probes the smaller hash table (expecting fewer cache-misses) that will consequently reduce the number of tuples that also probe the (more expensive) larger hash table. Yet, this results in a slower runtime than the alternative.

As this is counter-intuitive, the developer wants to identify the cause and refine the cost function. The developer thus applies Tailored Profiling to inspect the operator activity over time in the probing pipeline (cf. Figure 11). The report confirms that the alternative plan is faster. Moreover, starting at 70 ms in the alternative plan, the join on orders becomes dominant while becoming negligible in the original plan. After this hint, further investigation reveals that lineitem

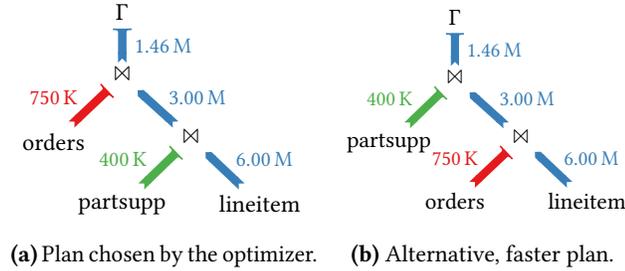


Figure 10. Alternative query plans for the optimizer developer’s use-case.

is scanned in the order of the join attribute. This leads to a situation where, first, the join on orders finds a match for all tuples and passes them to the next operator until 70 ms. Then, starting at 70 ms, the join on orders eliminates all tuples, so the hash table for partsupp is not probed at all, yielding an overall behavior that is easy to predict by branch predictors, which is especially beneficial for hardware with out-of-order execution capabilities. The optimizer developer can now decide whether to extend to cost function with such data-layout and hardware-specific properties.

Operator Developer. In the first use case, we have seen how a user of the database system can get a higher-level overview of the query’s performance (recall Figure 9). An operator developer, who is responsible for implementing efficient operators, needs a more detailed view of the internals. Very often, they are interested in the data access patterns, which can play a big role on the actual performance of the algorithm.

Tailored Profiling makes use of the hardware’s sampling support to also record the addresses with every memory access. With the Tagging Dictionary, the instruction that initiated the memory access can be associated with an operator, and as a result we can get an accurate memory access profile for each operator (Figure 12). The operator developer can inspect the memory profile and compare it to their expectations. In this example, the table scans on orders and lineitem show a linear data access pattern over time, which is ideal for hardware prefetchers, etc. The join and group by operators access memory in a more widespread fashion as a result of using hash tables in their implementation. This can be used as a starting point for further investigation, e.g., into a memory access profile with cache-miss information, or for considering alternative operator algorithms.

6.2 Runtime Overhead

Our approach to Tailored Profiling incurs three sources of runtime overhead.

First, while profiling, the hardware sampling mechanism stores samples in a memory buffer (PEBS buffer), which occasionally must be flushed by the operating system. Figure 13 shows how the sampling overhead increases with sampling

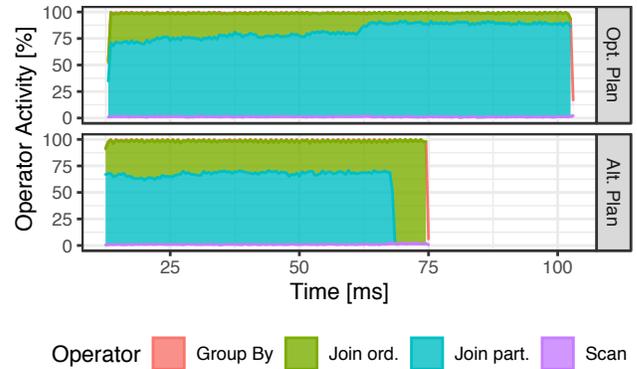


Figure 11. Operator activity over time for the plans of Figure 10.

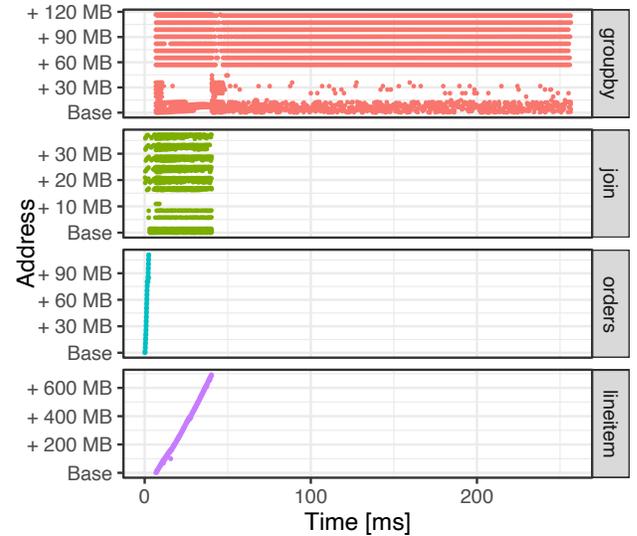


Figure 12. Profile of memory access patterns for the operators of Figure 9b. Each point denotes a sample with time (from query start) and accessed address (offset from lowest address the operator accesses).

frequency. At our default setting of taking a sample every 5000 cycles (0.7 MHz), the overhead is 35%. Note that this overhead is solely caused by PEBS recording the samples. The Tagging Dictionary is populated during compile time and thus does not incur runtime overhead.

Second, the amount of information included in the samples potentially increases the overhead. Figure 13 also shows the overhead for additionally sampling register values, as required for Register Tagging. When sampling every 5000 cycles the overhead grows to 38%, thus reserving a machine register, and writing the tags into it introduces 3% additional overhead. Call-stack sampling — the alternative to Register Tagging — incurs an overhead of 529%, constructing and recording the call-stack in each sample. In comparison, the overhead of Register Tagging is moderate.

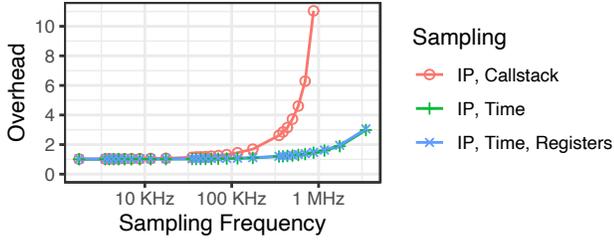


Figure 13. Performance overhead of the profiling approaches for TPC-H Query 16. The results for *IP, Time, Registers* incur on average 3% more overhead than *IP, Time*.

Third, reserving one register for Register Tagging slows down query execution, as the compiler generates worse code. Over all 22 TPC-H queries, we observed an overhead of 2.8%, on average.

Further overhead from profiling occurs in form of storage space required for the recorded samples and the Tagging Dictionary. Samples with IP, timestamp, and register values require 54 Bytes (when adding call-stack information 265 Bytes). Thus, at a sampling frequency of 0.7 MHz we need to store 77 MB per second. Each entry in the Tagging Dictionary is a triple of operator, task, and LLVM IR source line, which we represent with 24 Bytes. With one triple per LLVM IR instruction — of which there are on average ~ 1320 in a TPC-H query — the dictionary requires ~ 30 kB.

Overall, we observe that the induced overhead is rather low, as we never encountered any interference of profiling overhead with query execution and the performance profiles are very plausible.

6.3 Accuracy

To validate the correctness of Tailored Profiling’s reports, we check the accuracy of our approach and evaluate the accuracy of the samples recorded by PEBS.

To test the accuracy of the profiling reports, we profiled all 22 TPC-H queries with Tailored Profiling and report the amount of samples covered by the Tagging Dictionary’s mapping in Table 2. The experiment shows that our approach can attribute 98% of the samples to Umbra’s higher abstraction levels and the kernel (e.g., for memory allocations). Further investigation reveals that the remaining 2% belong to other system libraries, for which we did not apply Register Tagging.

An astute reader may have already observed that the Tailored Profiling can only attribute samples correctly when the sampled instruction pointer is accurate. We cross-checked the sampled instruction pointers with Register Tagging by applying the tagging not only for shared code locations but also for all instructions in generated code. Our test over all TPC-H queries yields no mismatches, thus, the instruction pointer matches the Register Tagging for all samples. Furthermore, we have evaluate the sample accuracy empirically

Table 2. Amount of samples attributed to Umbra by Tailored Profiling over all TPC-H queries.

Attribution	Amount of Samples
Umbra	98.0%
→ Operators	95.4%
→ Kernel Tasks	2.6%
No attribution	2.0%

Table 3. Lines of code of our prototype implementation of Tailored Profiling.

Component	Lines Added	Lines Before
Umbra Code Gen.	56	$\sim 22,000$
Tailored Profiling	1,686	0
→ Sample Processing	1,176	0
→ Visualization	510	0
Σ	1,742	$\sim 22,000$

by profiling the query execution for different profiling events. We cross-checked for three TPC-H queries (2, 16, 18) whether the instruction pointers in all samples occur at instructions that could plausibly cause the sampled event, e.g., samples for load-misses always point to loads and branch-misses contained either the branching instruction or the preceding compare causing the misprediction.

Finally, we have evaluate the accuracy of the sampled timestamps for Tailored Profiling’s time dimension. For this, we profiled the query execution taking a sample every 5000 cycles and check the TSCs of consecutive samples. In our experiment, the TSC values reflect the sampling distance (max. deviation ~ 40 cycles) and adapt accordingly when we vary it. Ultimately, Tailored Profiling’s timing information depends on the accuracy provided by the hardware. In our experience, TSC-based timestamps appear to provide a precise resolution reflecting the samples’ recording time.

Overall, our validation yields very small inaccuracies and validates the reliability of Tailored Profiling’s reports and time dimension.

6.4 Implementation Effort

Integrating our approach is lightweight and requires only small additions to the dataflow system, as shown in Table 3. Tailored Profiling leverages existing profilers to record samples and processes the profiling samples with the Tagging Dictionary to map them to higher abstraction levels.

Thus, we need to add the Tagging Dictionary mechanism and Register Tagging into the dataflow system and populate the Tagging Dictionary during the lowering process, as shown Figure 4. Integrating the dictionary into Umbra required only 44 lines of code, while the Abstraction Tracker needed 6 lines and Register Tagging has 6 lines. The main

implementation effort went into mapping the profiling samples to higher abstraction levels, followed by creating the visualizations of the developer tailored views. Modifying the kernel for samples with TSC timestamps needed just 1 line of code, and reserving a register in the LLVM compiler framework took only 2 lines.

Portability. Porting our approach to a different compiling dataflow system requires minor effort: adding the Tagging Dictionary mechanism and Abstraction Trackers into the system, creating a dictionary log for each lowering step, and depending on the runtime environment, either integrating Register Tagging or using call-stack sampling. The most critical part would be that the reports created by Tailored Profiling will need to be adapted to the system’s abstraction levels.

Configuration Trade-Off. Depending on the dataflow system’s runtime environment and requirements, one can either rely on using call-stack sampling or Register Tagging. Some dataflow systems that run on managed runtimes (e.g., Spark on JVM) can primarily rely on call-stack sampling, while others can decide on the trade-off between profiling resolution and reserving machine registers.

To make that decision, we need to consider the number of lowering steps that the system employs without a unique mapping between the higher- and lower-level’s components. For each of those lowering steps, Register Tagging requires one exclusive register for disambiguation, which comes with a performance overhead. Thus, we need to make the trade-off between reserving more registers or switching to call-stack sampling.

7 Related Work

Tailored Profiling was inspired by how debug tools instrument executables with meta-data [11] to resolve native instructions to source code on a single level of abstraction [9, 47]. Li and Flatt extend this abstraction level to DSL terms suitable for the user with macros [24]. Debug information is also used by profilers to attribute profiling samples to executed code. Most research on profiling and work on profilers focuses on software that is compiled ahead of time [3, 18, 25, 50]. Consequently, they present profiles in terms of assembly, source lines, and function calls. Hotspot and vTune also offer an interactive view to zoom in on function-specific profiles or time intervals, also by selecting hardware events of interest. Furthermore, there are profilers built to analyze specific events (e.g., Intel’s PIN monitors memory bandwidth usage), while Noll et al. visualize memory access patterns [21, 28, 35].

Meanwhile, hardware vendors constantly improve the selection of events available for profiling, increase the accuracy, and reduce the overhead [16]. How they translate into practice is constantly being investigated [4, 10, 12, 36, 37].

Profiling (compiling) dataflow systems has always been a non-trivial task. Prior work includes manual analysis of

profile components to attribute samples to operators [35], replaying execution in a simulator [48], tracking memory allocations to map samples to data-structures [41], call-stack sampling within the Java virtual machine [45], or dynamic calling contexts to reduce overhead of call-stack sampling [6]. All of these approaches, however, fall short of providing a universal operator mapping that works for any abstraction level and can at the same time be sampled with low overhead and sufficient frequency to show behavior over time.

8 Conclusion

Despite having access to extensive hardware support, existing profilers are still unable to adequately present performance profiles tailored to the needs of everyone involved in building or using dataflow systems. In this paper, we have proposed Tailored Profiling which addresses this problem by providing reports on any abstraction levels the developer is comfortable using. Our approach is built on two novel contributions:

First, we introduced the Tagging Dictionary that tracks the links of high-level concepts and their generated low-level code (concepts), populated during the dataflow system’s compilation phases. The post-processing phase then combines the Tagging Dictionary with existing low-level meta-data (debug information) to map profiling samples to the dataflow system’s higher abstraction levels. Second, to disambiguate linking higher-level components having shared source locations, we have introduced Register Tagging as a light-weight alternative to call-stack sampling.

Our approach is applicable for dataflow systems running on a single (multi-socket, multicore) machine and provides reports for code executed on CPUs that support sampling-based profiling. Tailored Profiling can work with multiple code generators as long as they keep track of their lowering steps (with meta-data information populating the respective logs of the Tagging Dictionary) and make that information available to the post-processing phase. For example, our current prototype in Umbra already works with two sequential code generators. In the future, we envision our approach to be a fit to proposals of meta-compiler frameworks like MLIR [23].

Other venues for future work that are not yet in our system’s scope are adding support for profiling distributed systems as well as for dataflows that leverage heterogeneous compute resources (e.g., accelerators). This would require combining Tagging Dictionary logs from different sources and presenting them in an understandable and intuitive format.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

References

- [1] 2019. OProfile. <https://oprofile.sourceforge.io/>.
- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 265–283.
- [3] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummy, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
- [4] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *ROSS@HPDC 2017, Washington, DC, DC, USA, June 27 - 27, 2017*. ACM, 3:1–3:8.
- [5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. <https://doi.org/10.1145/2723372.2742797>
- [6] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. 2010. Bread-crumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 13–24. <https://doi.org/10.1145/1806596.1806599>
- [7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.
- [8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 578–594.
- [9] GDB developers. 2020. GDB: The GNU Project Debugger. <https://www.gnu.org/software/gdb/>
- [10] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. 2016. Reliable and efficient performance monitoring in linux. In *SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. IEEE Computer Society, 396–408.
- [11] Michael J. Eager. 2012. Introduction to the DWARF Debugging Format. <http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf>
- [12] Stéphane Eranian. 2019. Linux perf_events updates. Scalable Tools Workshop 19.
- [13] Panagiotis Garefalakis. 2020. *Supporting long-running applications in shared compute clusters*. Ph.D. Dissertation. Imperial College London.
- [14] Brendan D. Gregg. 2019. Flame Graphs. <http://www.brendangregg.com/flamegraphs.html>.
- [15] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS 2012, London, UK, March 3-7, 2012*. ACM, 349–362.
- [16] Intel. 2019. Intel 64 and IA-32 Architectures Optimization Reference Manual. <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>.
- [17] Intel. 2020. Intel 64 and IA-32 Architectures Software Developer Manuals. <https://software.intel.com/en-us/articles/intel-sdm>.
- [18] Intel. 2020. Intel VTune Profiler. <https://software.intel.com/en-us/vtune>.
- [19] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.)*. IEEE Computer Society, 195–206. <https://doi.org/10.1109/ICDE.2011.5767867>
- [20] Timo Kersten and Thomas Neumann. 2020. On another level: how to debug compiling query engines. In *Proceedings of the 8th International Workshop on Testing Database Systems, DBTest@SIGMOD 2020, Portland, Oregon, June 19, 2020*. ACM, 2:1–2:6.
- [21] Andi Kleen. 2020. pmu tools. <https://github.com/andikleen/pmu-tools>.
- [22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. 75–88.
- [23] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Sheisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore’s Law. *CoRR* (2020).
- [24] Xiangqi Li and Matthew Flatt. 2017. Debugging with domain-specific events via macros. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoit Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 91–102. <https://doi.org/10.1145/3136014.3136019>
- [25] Linux. 2020. Linux perf. <https://github.com/torvalds/linux/tree/master/tools/perf>.
- [26] Linux. 2020. perf_event_open(2). http://man7.org/linux/man-pages/man2/perf_event_open.2.html.
- [27] Xunyun Liu and Rajkumar Buyya. 2020. Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions. *ACM Comput. Surv.* 53, 3 (2020), 50:1–50:41.
- [28] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *SIGPLAN '05, Chicago, IL, USA, June 12-15, 2005*. ACM, 190–200.
- [29] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*.
- [30] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13.
- [31] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *Proc. VLDB Endow.* 4, 11 (2011), 843–851. <http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf>
- [32] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP 13, Farmington, PA, USA, November 3-6, 2013*. 439–455.
- [33] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
- [34] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.
- [35] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2020. Analyzing memory accesses with modern processors. In *DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. 1:1–1:9.

- [36] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. 2018. On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous Memory Management at Scale. In *MCHPC@SC 2018, Dallas, TX, USA, November 11, 2018*. ACM, 50–57.
- [37] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a Base of Trust with Performance Counters for Enterprise Workloads. In *USENIX ATC '15, July 8–10, Santa Clara, CA, USA*. USENIX Association, 541–548.
- [38] Trail of Bits. 2019. A tsc_freq_khz Driver for Everyone. https://github.com/trailofbits/tsc_freq_khz.
- [39] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *CIDR '17*. 45.
- [40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS 2019, 8–14 December 2019, Vancouver, BC, Canada*. 8024–8035.
- [41] Aleksey Pesterev, Nikolai Zeldovich, and Robert Tappan Morris. 2010. Locating cache performance bottlenecks using data profiling. In *EuroSys 2010, Paris, France, April 13–16, 2010*. ACM, 335–348.
- [42] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (2016), 1707–1718.
- [43] Malte Schwarzkopf. 2020. *The Remarkable Utility of Dataflow Computing*. <https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/>
- [44] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *SIGPLAN, PPOPP, Shenzhen, China, February 23–27, 2013*. 135–146.
- [45] Christian Stuart. 2020. *Profiling Compiled SQL Query Pipelines in Apache Spark*. Master's thesis. Universiteit van Amsterdam.
- [46] Google XLA team. 2017. XLA - TensorFlow, compiled. <https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html>.
- [47] LLDB Team. 2007. The LLDB Debugger. <https://lldb.lvm.org>
- [48] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in wonderland: where do cache misses come from in major OLTP components?. In *DaMoN 2013, New York, NY, USA, June 24, 2013*. ACM, 8.
- [49] Transaction Processing Performance Council (TPC). 1993–2018. *TPC BENCHMARKTM H (Decision Support) – Standard Specification Revision 2.18.0*.
- [50] Milian Wolff. 2020. Hotspot - the Linux perf GUI for performance analysis. <https://github.com/KDAB/hotspot>.
- [51] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.