An Overview of Hawk: A Hardware-Tailored Code Generator for the Heterogeneous Many Core Age

Sebastian Breß[1] Henning Funke[2] Steffen Zeuch[1] Tilmann Rabl[1] Volker Markl[1]

Abstract: Processor manufacturers build increasingly specialized processors to mitigate the effects of the power wall in order to deliver improved performance. Currently, database engines have to be manually optimized for each processor which is a costly and error prone process. In this paper, we provide a summary of our recent VLDB Journal publication, where we propose concepts to adapt to performance enhancements of modern processors and to exploit their capabilities automatically. Our key idea is to create processor-specific code variants and to learn a well-performing code variant for each processor. These code variants leverage various parallelization strategies and apply both generic and processor-specific code transformations. We observe that performance of code variants may diverge up to two orders of magnitude. Thus, we need to generate custom code for each processor for peak performance. Hawk automatically finds efficient code variants for CPUs, GPUs, and MICs.

1 Introduction

Over the last decade, the main memory capacity has reached the terabyte scale. Main memory databases exploit this trend in order to satisfy the ever-increasing performance demands. As a result, they store data primarily in main-memory to eliminate disk I/O as the main bottleneck. As a result, memory access and data processing have become the new performance bottlenecks for in-memory data management [Ma00].

Current designs of main-memory database systems assume that processors are homogeneous, i.e., with multiple identical processing cores. However, todays hardware vendors break with this paradigm in order to circumvent the fixed energy budget per chip [BC11]. This so-called *power wall* forces vendors to explore new processor designs to overcome the energy limitations [Es11]. Hardware vendors integrate heterogeneous processor cores on the same chip, e.g., combining CPU and GPU cores as in AMD's Accelerated Processing Units (APUs). Another trend is *specialization*: processors are optimized for certain tasks, which already have become commodity in the form of *Graphics Processing Units* (GPUs), *Multiple Integrated Cores* (MICs), or *Field-Programmable Gate Arrays* (FPGAs). These accelerators promise large performance improvements because of their additional computational power and memory bandwidth. Thus, from a processor design perspective, the *homogeneous many*

[1] TU Berlin and DFKI GmbH, Berlin {sebastian.bress,steffen.zeuch,tilmann.rabl,volker.markl}@dfki.de
[2] TU Dortmund, Dortmund, henning.funke@tu-dortmund.de

doi:10.18420/btw2019-ws-07

core age ends [BC11]. The upcoming *heterogeneous many core age* provides an opportunity for database systems to embrace processor heterogeneity for peak performance.

Our goal is to empower database systems to automatically generate efficient code for any processor without *any a priori* hardware knowledge, thus making database systems fit for the heterogeneous many-core age. To achieve this goal, we proposed Hawk [Br18], a novel hardware-tailored code generator, which produces variants of generated code. By executing code variants of a compiled query, Hawk adapts to a wide range of different processors without any manual tuning. Hawk achieves low compilation times and executes queries on a wide range of processors. In this paper, we provide a summary of our recent publication in the VLDB Journal [Br18]. Hawk's code is available as open source.[3]

2 Overview of Hawk

To provide an architectural overview, we describe Hawk's role in the process of executing an SQL query. The SQL parser translates queries into relational query plans. After that, the query optimizer rewrites the query plan by applying common optimizations to obtain a query execution plan. On the next layer, Hawk provides a code generation back-end to perform query compilation for efficient query execution. To this end, Hawk compiles query execution plans just-in-time into machine code of a target processor.

Hawk's key feature is the generation of efficient code for processors of different architectures. Our approach follows the principles of query compilation [Ne11] as opposed to vector-at-a-time processing [Bo05], because query compilation has the largest potential of applying processor-specific optimizations. Hawk uses a three-step compilation process: 1) query segmentation, 2) variant optimization, and 3) code generation (see Figure 1). In general, Hawk receives a query plan as input and outputs optimized code for the underlying processors. This process centers around *pipelines*, i.e., non-blocking data flows. In particular, all operations in a pipeline are fused into one operator. The individual steps are as follows.

Query Segmentation. Hawk first segments query execution plans into pipelines using the produce/consume model [Ne11] (Step (1) in Figure 1). During this step, Hawk creates a *pipeline program* for each pipeline as the intermediate representation for a pipeline. A pipeline program consists of simple operations such as loop, filter, and hash probe and establishes the start point for optimization and target code generation.

Variant Optimizer. The initial pipeline program represents a hardware-oblivious blueprint as a starting point for processor-specific optimizations. Based on that, Hawk produces hardware-tailored code by applying modifications to the pipeline programs. A *modification* is a change to a pipeline program, which preserves its semantic but changes the

[3] `https://github.com/TU-Berlin-DIMA/Hawk-VLDBJ`

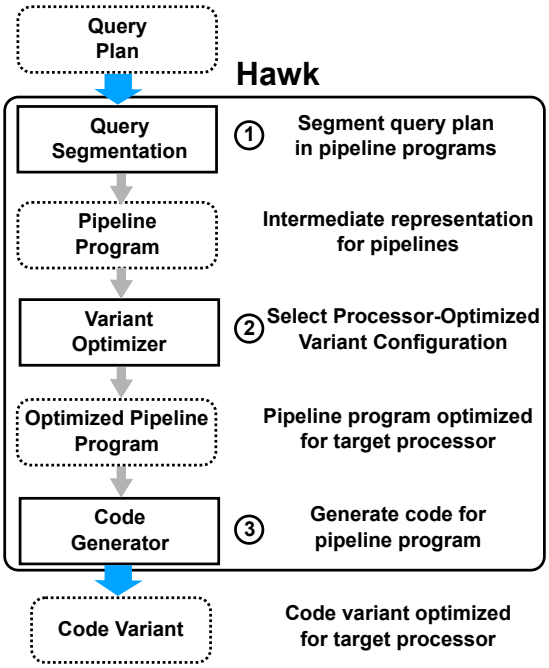


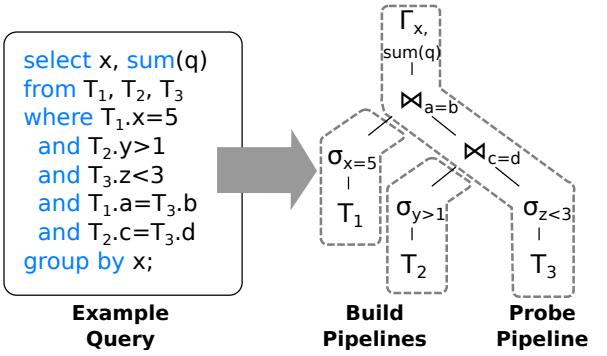
Fig. 1: Hawk's compilation process.

generated code (e.g., memory access pattern). A *variant configuration* captures all modifications of a pipeline program and thus provides a value for each supported modification. The set of all modifications defines the code generated by Hawk. The variant optimizer selects an efficient variant configuration for each pipeline program on a target processor (Step (2) in Figure 1). Note that Hawk automatically determines a variant configuration for each target processor without the need for manual tuning. In sum, Hawk applies the modifications specified in the variant configuration to the input pipeline program and returns an optimized pipeline program.

Code Generator. The code generator takes the optimized pipeline program as an input and produces the target code (Step (3)). We refer to the compilation result as *code variant*.

3 End-To-End Compilation Example

We exemplify the translation process of Hawk with the query illustrated in Figure 2. We also show the pipeline programs that are created during the translation. The query contains two hash joins, leading to a query plan with three pipeline programs. The pipeline programs describe three pipelines. The two build pipelines iterate over their input tables (T_1 and T_2), apply their filters, insert the matching keys into a hash table, and materialize their result. The probe pipeline program iterates over table T_3, applies it's filter, probes the hash tables, and performs the aggregation. Next, the variant optimizer selects a variant configuration, which describes the customized features for the translation. Then, the variant optimizer annotates each pipeline program according to the variant configuration. Hawk determines variant configurations by an offline-training workload of test queries using a structured experiment [Br18]. For simplicity and space restrictions, we continue our example for Build Pipeline 1 only.



Build Pipeline 1	Build Pipeline 2	Probe Pipeline
LOOP(T_1, ..)	LOOP(T_2, ..)	LOOP(T_3, ..)
FILTER(x=5, ..)	FILTER(y>1, ..)	FILTER(z<3, ..)
HASH_PUT(a, ..)	HASH_PUT(b, ..)	HASH_PROBE(a=c, ..)
PROJECT(a, x, ..)	PROJECT(b, ..)	HASH_PROBE(b=d, ..)
		HASH_AGGREGATE(x, sum(q), ..)

Fig. 2: Segmentation of example query into pipeline programs.

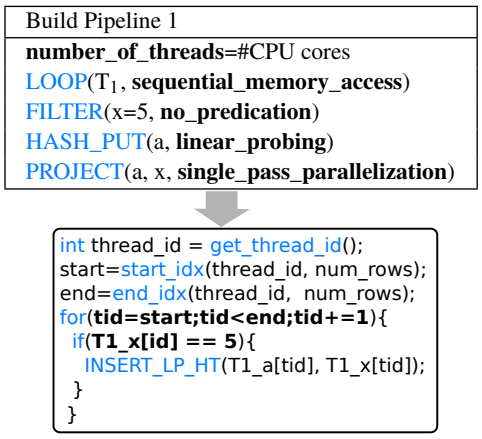


Fig. 3: Compiling an optimized pipeline program to target code.

Hawk supports several code modifications such as the memory access pattern (LOOP), branched predicate evaluation or software predication (FILTER), different hashing schemes (HASH_PUT), and parallelization strategies (PROJECT). We show a CPU-optimized pipeline program in Figure 3. It uses one thread per core, a sequential memory access pattern, a linear probing hash table, and single-pass parallelization. We illustrate the code generated by Hawk in Figure 3. On GPUs, we use a different parallelization approach called multi-pass to avoid high synchronization overhead between threads [Br18]. Finally, Hawk passes the code to the OpenCL compiler and executes the final kernel program.

4 Summary of Key Insights

In this paper, we provided an overview of Hawk, a hardware-tailored code generator that customizes code for a wide range of heterogeneous processors. Through hardware-tailored implementations, Hawk produces fast code *without manual tuning* for a specific processor. Our abstraction of pipeline programs allows us to flexibly produce code variants while keeping a clean interface and a maintainable code base. Code variants optimized for a particular processor can result in performance differences of up to two orders of magnitude on the same processor. Thus, it is crucial to optimize the database system for each processor.

Acknowledgments. This work was funded by EU project E2Data (780245), DFG Priority Program "Scalable Data Management for Future Hardware" (MA4662-5) and Collaborative Research Center SFB 876, project A2, and the German Ministry for Education and Research as BBDC I (01IS14013A) and BBDC II (01IS18025A).

References

[BC11] Borkar, Shekhar; Chien, Andrew: The future of microprocessors. Communications of the ACM, 54(5):67–77, 2011.

[Bo05] Boncz, Peter et al.: MonetDB/X100: Hyper-Pipelining Query Execution. In: CIDR. pp. 225–237, 2005.

[Br18] Breß, Sebastian et al.: Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. The VLDB Journal, Jul 2018.

[Es11] Esmaeilzadeh et al.: Dark Silicon and the End of Multicore Scaling. In: ISCA. ACM, pp. 365–376, 2011.

[Ma00] Manegold, Stefan et al.: Optimizing Database Architecture for the new Bottleneck: Memory Access. The VLDB Journal, 9(3):231–246, 2000.

[Ne11] Neumann, Thomas: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB, 4(9):539–550, 2011.