Demonstrating Self-Learning Algorithm Adaptivity in a Hardware-Oblivious Database Engine

Max Heimel
Technische Universität Berlin
max.heimel@tu-berlin.de

Filip Haase
Technische Universität Berlin
filip.haase@campus.tu-berlin.de

Martin Meinke
Universität Tübingen
martin.meinke@student.uni-tuebingen.de

Sebastian Breß
University of Magdeburg
sebastian.bress@ovgu.de

Michael Saecker
Parstream GmbH
michael.saecker@parstream.com

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

The increasingly heterogeneous modern hardware landscape is forcing database vendors to rethink basic design decisions: With more and more architectures to support, the traditional approach of building on hand-tuned operators might simply become too cost- and labor-intensive.

With this problem in mind, we introduced the notion of a hardware-oblivious database engine, which avoids device-specific optimizations and targets multiple different hardware architectures from a single code-base. We demonstrated the feasibility of this concept through Ocelot, a prototypical hardware-oblivious database that uses OpenCL to provide operators that can run on multiple architectures.

In this demonstration, we show how we modified Ocelot to support self-learning algorithm adaptivity: The ability to automatically learn which algorithms are optimal for a given operation on a given hardware architecture. We present how to specify operators that can be executed by multiple algorithms, provide details about the underlying learning and decision routines, and demonstrate how our system picks the optimal algorithm when running on systems with multiple devices, such as CPUs and graphics cards.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Design, Performance

Keywords

Modern Hardware Architectures, Self-Adaptivity, Hardware-Oblivious Data Processing

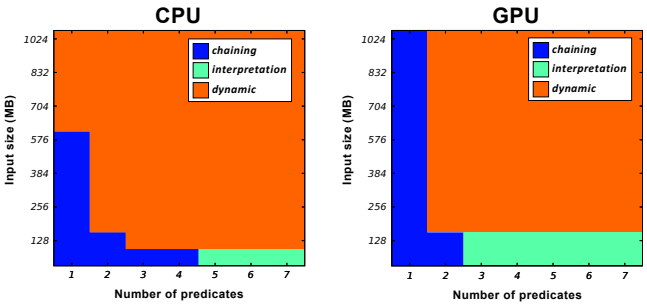

Figure 1: Maps displaying the optimal algorithm choice for a selection operation with multiple predicates dependent on input size and number of predicates for a CPU and a GPU.

1. INTRODUCTION & MOTIVATION

The hardware landscape is getting increasingly diverse, which forces modern database engines to target a wide variety of architectures. Today, a single machine can already contain several different parallel processors, such as multi-core CPUs or graphics processing units (GPUs), and hardware heterogeneity is expected to keep growing in the future [2]. Providing efficient data processing under these circumstances usually requires database vendors to implement several specialized, manually tuned database operators for each targeted architecture – a very challenging and resource-intensive task.

We introduced the notion of a hardware-oblivious database engine as a way to cope with this diversity in prior work [4]. The general idea is to minimize development and maintenance efforts by avoiding hand-tuned implementations and relying on hardware abstraction to generate device-specific operators at runtime. We contributed an initial proof-of-concept by implementing *Ocelot*[1], a prototypical hardware-oblivious database engine. Ocelot provides a set of hardware-oblivious drop-in replacements for the query operators of the in-memory column-store MonetDB [1]. We could demonstrate that this approach offers competitive performance across such diverse architectures as CPUs and GPUs from a single codebase.

[1]The Ocelot source code is available at: `goo.gl/GHeUv`.

10.5441/002/edbt.2014.57

However, the approach we chose for Ocelot is fairly limited: All targeted architectures are restricted and must use the same set of algorithms. This is problematic, as different architectures typically tend to prefer different algorithms for the same operation. For instance, sorting on a CPU is typically done using a variant of quick-sort, while GPUs rely on a combination of sorting networks, merge sort, and radix sort to provide good performance [8]. Forcing all architectures to use the exact same algorithms therefore negatively impacts performance.

Figure 1 illustrates this problem based on measurements we did for a multi-predicat eselection operation. The plot shows that – for both CPU and GPU –, the optimal algorithm choice[2] is highly dependent on both the number of predicates and the input relation size. Furthermore, the decision boundaries between the algorithms are usually non-trivial and vary greatly between two architectures. Based on our experience, boundaries even differ between devices from the same architecture (e.g., when comparing graphics cards from NVIDIA and AMD): In order to make optimal decisions, the system therefore needs a distinct set of cost models for each supported architecture!

A possible approach to tackle this problem is to ship the database engine with multiple pre-defined sets of cost models – one for each supported architecture. While this approach would allow us to reuse the existing query optimizer infrastructure without any major changes, it would also limit the system to a vendor-selected set of architectures. This limitation directly contradicts our goal of building a "hardware-oblivious" database, i.e., one without any inherent reliance on a specific architecture. We instead suggest an alternative approach, that utilizes methods from Machine Learning to automatically learn cost models at runtime by observing the operators on the installed hardware. This way, the system can automatically adapt its engine to any given, previosly unknown architecture.

In this demonstration, we present how we modified Ocelot to provide these properties. We demonstrate our interface to specify multi-algorithm operations, and our framework[3] to learn the decision boundaries and pick the optimal algorithm for the current hardware and input parameters. By combining these techniques, we can provide self-learning algorithm-adaptivity, which is a very important step on the road towards a truly hardware-oblivious system.

2. SYSTEM OVERVIEW

In this section, we provide an overview of our system Ocelot, introduce the framework to specify multi-algorithm operators, and discuss the training and decision routines that are employed to learn the optimal algorithm choice.

2.1 Ocelot

The overall architecture of Ocelot is depicted in Figure 2. Primarily, Ocelot is designed as a drop-in replacement for MonetDB's execution engine, which allowed us to reuse several major components, including data layout, storage management, and the query optimizer.

The central part of Ocelot are the operators, which are implemented against the abstract parallel programming library OpenCL [9]. We decided to use OpenCL as our foundation, as it is an open standard that is supported across a wide variety of platforms from all major hardware vendors – including CPUs, GPUs, accelerators like Xeon Phi, and even FPGAs. Thereby we achieve efficient, yet highly portable code without the need for optimization by hand.

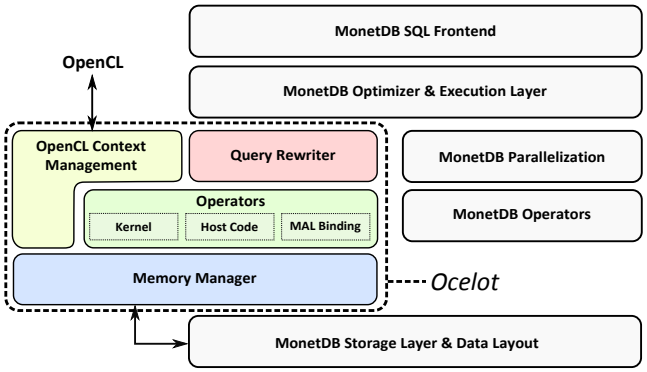

Figure 2: The architecture of Ocelot.

Each operator is advertised in MonetDB via a *MonetDB Assembly Language (MAL) binding*, which describes the interface and the location of the entry function. The entry function – also called the *operator host-code* – is performing management and maintenance services for the operator. This includes checking of input parameters, setting input and output resources, and scheduling the actual execution on the device. The actual work is performed by the so-called *kernels*, which are small programs that perform data-intensive operations on the device.

Besides the operators, Ocelot consists of infrastructure components that help to abstract from details of the hardware. An important example of such an infrastructure component is the Memory Manager, which is used by the operators to request resources and also manages device caches on devices with dedicated memory. Another important component is the Scheduler, which maintains wait lists of scheduled kernel executions to ensure that operations are only started once their inputs are ready, and also tracks and reports the runtime of finished operations.

2.2 Specifying Multi-Algorithm Operators

In order to specify a multi-algorithm operator in Ocelot, the developer has to provide the following three parts:

1. The actual *algorithms*: These are basically "traditional" Ocelot operators, consisting of host-code and kernels, as discussed in Section 2.1. All algorithms of the same operation have to share the same signature.

2. The *feature-provider*: This is a function that receives the input for an operator and has to extract meaningful training features from it. These features are used to learn the cost functions of the provided algorithms and to predict their costs. Possible features are for instance: Input cardinality, predicate selectivity, predicate complexity, and distribution skewness.

3. An *operator description*: This is a small code fragment that ties together the other pieces. At compile

[2]For further details on the algorithmic variants that were used for this illustration, please refer to Section 3.
[3]The learning framework is based in large parts on the Hype library by Bress et al. [3].

Listing 1: Operator description code for a multi-algorithm selection operator in Ocelot.

```
1 BEGIN_OPERATOR(selection)
2   ALGORITHM(ocl_select_chain, "chaining");
3   ALGORITHM(ocl_select_interp, "interpretation");
4   ALGORITHM(ocl_select_dynamic, "dynamic");
5   FEATURE_PROVIDER(ocl_select_features);
6 END_OPERATOR(selection)
```

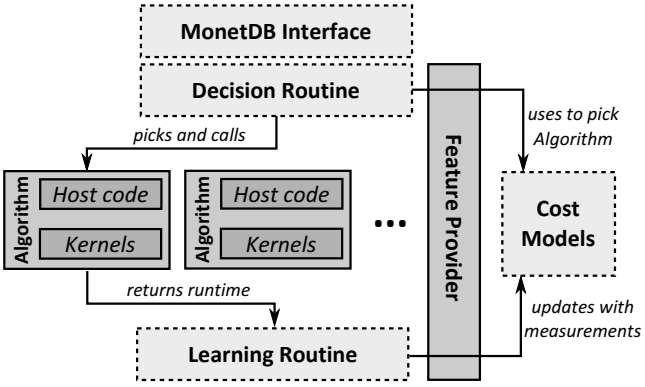

Figure 3: Overview over the generated operator framework used to pick an algorithm. Boxes with solid lines are provided by the developer, while boxes with dotted lines are generated.

time, Ocelot uses this information to generate a framework operator that implements the decision and training logic. An exemplary operator description can be seen in Listing 1.

From these pieces, Ocelot generates a framework operator that can adapt to the underlying hardware by choosing from the provided algorithm alternatives. Figure 3 illustrates the main components of this generated operator framework. The work-flow is as follows: When the operator is called, the generated decision routine picks one of the provided algorithms. In order to make this decision, expected runtime costs for all provided algorithms are computed using the learned cost models and the feature provider. After the chosen algorithm returns, the generated learning routine receives a new training point, consisting of the chosen algorithm's runtime and the feature vector. This data point is then used to update the algorithm's cost model. Since Ocelot can run on multiple devices simultaneously, we maintain multiple models per algorithm.

2.3 Learning & Decision Framework

In this subsection, we give a few more details about how our decision and learning logic works. Basically, learning and exploiting the cost functions of the provided algorithms is a variant of the so-called multi-armed bandit, which is a central problem from probability theory [6]: Imagine a gambler that is presented with a set of slot machines, each of which has a different, unknown pay-out rate. In order to maximize his winnings, the gambler has to quickly identify the machine with the highest pay-out rate. In our scenario, the slot machines correspond to the algorithms, and the pay-out rates to their unknown cost functions.

At the core of each strategy, in order to tackle the multi-armed bandit, is a trade-off between *exploration* and *exploitation*. We can either exploit our current knowledge by choosing the algorithm that we believe to be optimal at the moment, or we can explore by choosing a different algorithm and improve our knowledge about its cost function. If we keep exploring, we are guaranteed to eventually find the optimal algorithm, however, it might be very expensive, since we will frequently make "sub-optimal" choices. On the other hand, if we start exploiting too early, we might end up with a sub-optimal choice due to lacking knowledge.

Luckily, literature has shown that even basic heuristics yield surprisingly good results for this problem. One possible heuristic is the so-called ϵ-greedy strategy: Given a parameter $\epsilon \in [0, 1]$, this strategy picks the currently best choice with probability $(1 - \epsilon)$, and explores a random other choice with probability ϵ [10]. We decided to use this strategy in the slightly modified decaying ϵ-greedy variant. In this version, the value of ϵ starts comparably high and then decreases over time. This results in a phase with high exploration in the beginning of the training, and increasingly conservative behavior that exploits the collected knowledge as time goes on.

After an algorithm was chosen, our system transparently tracks its runtime through the OpenCL profiling mechanism. The collected runtime and the generated feature vector are then passed on as a training example to our learning system, which is based on the Hype library by Bress et. al. [3]. Internally, the learning system trains and continuously updates L2-regularized linear regression models for each combination of hardware and algorithm. In order to also be able to learn non-linear cost functions, our system automatically extends all feature vectors by including several non-linear combinations of the provided features.

A similar approach to our self-adapating algorithm selection has been proposed by Răducanu et. al. to achieve "micro-adaptivity" in Vectorwise [7]. In this scenario, the authors considered the case of picking the optimal "flavor"[4] of an algorithm at runtime. Their system chops up the work into small pieces, and uses a modified ϵ-greedy algorithm to pick the optimal flavor for the current operation. This decision happens dynamically within the algorithm – hence the name "micro-adaptivity". Given that this technique works to automatically fine-tune single algorithm implementations, while our approach works across multiple algorithms, both methods could actually be used complementarily to improve performance.

3. DEMONSTRATION SETTING

In this section we give a quick overview over the scenario for our demonstration and our plan to present the most important aspects to the audience.

3.1 Scenario

For our demonstration, we decided to cover the trade-off between the basic query processing models of batch processing and dynamic code generation. While dynamic code generation usually yields better performing code [5], it also

[4]In this case, a flavor refers to a slightly different implementation of a given algorithm (e.g. unrolled loops or vectorization), or to identical implementations that were built using different compilers.

has higher fixed costs due to the expensive code generation, making it unsuited for fairly simple operations or for operations on small data sets. Batch processing on the other hand is often quite inefficient for complex operations, as it requires multiple passes over the data.

In order to keep things simple, we focus on a straightforward, but important case: The selection operation. We assume a scenario in which a high load of "complex"[5] predicates is evaluated against a base table. The operation should generate a bitmap that indicates for each tuple whether it qualifies the predicate or not. We specified our new selection operator based on the following three algorithms:

Chaining Each predicate is evaluated individually using a kernel that can efficiently evaluate range predicates, producing an intermediate bitmap. Afterwards, the intermediate bitmap is merged with the current result using a kernel that performs a bit-wise and operation.

Interpretation The complete predicate is encoded into an abstract syntax tree representation and shipped to a special kernel that can evaluate this representation on the given table.

Dynamic We use dynamic code generation to construct a custom kernel that evaluates the given predicate in a single pass over the data.

Each algorithm has its strengths and weaknesses: The *chaining* algorithm requires multiple passes over the data for complex predicates, however, it can use very efficient kernels. The *interpretation* algorithm requires only a single pass over the data, however, it also requires a fairly heavyweight kernel with several conditional statements – which can be inefficient on certain architectures, e.g., GPUs. Finally, the *dynamic* algorithm also requires only a single pass over the data, and features a very efficient kernel. However, the required code generation adds significant overhead before the kernel can even begin any work.

3.2 Demonstration Plan

At the beginning of the demonstration, we reset the internal bookkeeping of Ocelot to clear any prior knowledge about the three algorithms. We then launch a random workload of "complex" selection queries with varying number of predicates against multiple tables of different sizes. The system will immediately begin to learn the algorithm costs, converging to the optimal choice after a short burn-in period. After convergence, the user will be able to send custom selection queries to the system and observe which algorithm choice is made.

In order to give a more detailed insight into the decision process, we offer visual tools for inspecting the current cost functions, the decision boundaries, and the trend of the average runtime over the last few queries. In particular, we demonstrate how the system converges to a stable state after a short burn-in period, arriving at reasonable decision boundaries between the provided algorithms and lower average runtimes. We also demonstrate the effect of modifying the parameter ϵ, which governs the trade-off between exploitation and exploration: User can manually set a value for ϵ and restart the demonstration, to observe the difference in convergence behavior.

[5]Here, "complex" refers to a high number of disjoint conjunctive predicates that need to be evaluated.

In order to demonstrate the hardware-oblivious aspects of Ocelot, we run our scenario on at least two devices at the same time: The central processing unit and a graphics card. The viewer will be able to observe how the system learns decision boundaries for both devices individually, generating a similar illustration to the one shown in Figure 1.

4. REFERENCES

[1] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking The Memory Wall In MonetDB. *Communications of the ACM*, 51(12):77 – 85, December 2008.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[3] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic selection of processing units for coprocessing in databases. In *Advances in Databases and Information Systems*, pages 57–70. Springer, 2012.

[4] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[5] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[6] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[7] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, New York, NY, USA, 2013. ACM.

[8] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[9] The Khronos Group Inc. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, May 2011.

[10] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England, 1989.