Run-time Data Analysis to Drive Compiler Optimizations*

Sebastian Kloibhofer
Johannes Kepler University
Linz, Austria
sebastian.kloibhofer@jku.at

Abstract

Throughout program execution, types may stabilize, variables may become constant, and code sections may turn out to be redundant—all information that is used by just-in-time (JIT) compilers to achieve peak performance. Yet, since JIT compilation is done on demand for individual code parts, global observations cannot be made. Moreover, global data analysis is an inherently expensive process, that collects information over large data sets. Thus, it is infeasible in dynamic compilers. With this project, we propose integrating data analysis into a dynamic runtime to speed up big data applications. The goal is to use the detailed run-time information for speculative compiler optimizations based on the shape and complexion of the data to improve performance.

Keywords: Dynamic compilation, Compiler optimization, Data analysis, Program optimization

1 Motivation

The rise of *big data* inevitably brings new challenges for applications and hardware in terms of query complexity, processing capabilities, and database design [1, 2]. With more and faster main memory being available [3], in-memory data processing is more accessible than ever, to rid data-centric applications of I/O bottlenecks and to use the different cache layers efficiently [4, 5]. Those factors shift the burden of dealing with large datasets to programming languages and the underlying runtimes. Even more so, it is the developers' responsibility to ensure that data queries are efficient.

Consider the query in Fig. 1. Assuming complete knowledge about the processed data, we could apply a number of performance improvements: We could reorder the conditions based on their *selectivities*, adapt the loop for predication [6], vectorization [7], and better cache utilization [8], or even apply prior transformations to the data (cf. Section 3). Developers, however, often do not have perfect information about the data. Hence, even recognizing any potential for hand-crafted optimizations is a non-trivial task.

The Graal Compiler [9, 10]—the dynamic JIT compiler of the polyglot Java platform GraalVM [11]—optimizes programs using run-time information. The Truffle framework[12] additionally allows guest-language integration via abstract syntax tree (AST) interpretation. To explore data analysis in the context of program optimization, we propose to embed a data analysis framework into GraalVM. Truffle already utilizes profiling information to boost performance of the otherwise "slow" interpreter via partial evaluation, but there is no comprehensive support for other forms of analyses yet.

While the collection of run-time metrics via Truffle is covered in a concurrently developing joint project and is not part of this particular work, we want to adapt the compilation process based on such a framework. By extracting data-specific information from the AST, we want to perform compiler optimizations on data-heavy applications.

```
for (row in db)
  if (row.x == 0 || row.y < 125) accm += row.z
```

Figure 1. A loop ("query") to process data in a program

2 Problem

Data analysis encompasses a variety of techniques, applicable to a wide range of use cases. Databases accumulate data statistics to optimize queries [4]. Taint tracking tries to detect the abuse of untrusted data to prevent exploits [13]. Offline analyses of large data sets result in statistics and visualizations [14, 15]. Even compilers perform data flow analysis to improve application performance [16, 17].

Using data to optimize programs is not a novel idea and has been applied in runtime systems to specialize operations for observed input types, to determine high-level vectorization targets, or to reduce the memory footprint [18, 19]. Particularly dynamic languages are a favored target, due to their limited static information [20]. Also, *polyhedral models*[1] are frequently applied in data science, speeding up image processing or deep learning via hardware-dependent optimizations [21]. Nevertheless, many of those approaches are limited to specific computational patterns and loop structures or offer APIs for optimized data handling [8, 21]. Zhang et al. [22] use Truffle to optimize query plans. However, they require using *pragmas* to explicitly highlight target loops—again shifting responsibility to the developer.

Large-scale object-level data analysis within a JIT compiler is often impossible due to its compile-time overhead as well as its scope, mostly limited to individual compilation units. Therefore, we argue that a combined approach—analysis

*This research project is partially funded by Oracle Labs.

[1] a mathematical framework for optimizing operations such as nested loops

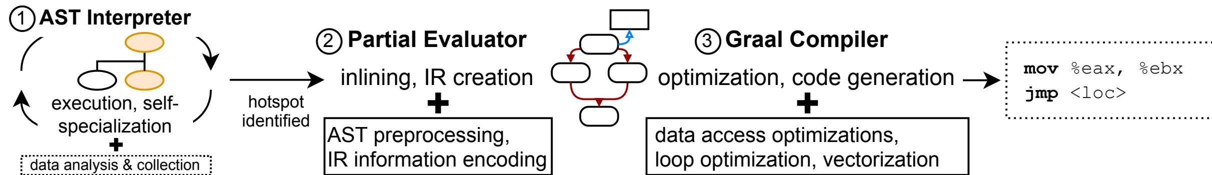

Figure 2. The proposed addition to the Graal compilation pipeline: Object-level analysis in the *AST interpreter*, preprocessing and encoding of analysis information into the IR at *partial evaluation*, and optimization of patterns and loops during *compilation*

without user interaction during interpretation and platform-specific optimizations during compilation—could counter the drawbacks of traditional data analysis and reveal new optimization potential. To the best of our knowledge, there is no research proposing a similar framework yet.

3 Approach

With this work, we focus on the compile-time dynamics that arise when using data analysis. We want to collect data-level information such as types and structures, accesses (memory reads, writes), value distributions, or query patterns (e.g. loops that filter/aggregate/transform data sets). While common compilers focus on structural optimizations such as loop unrolling [23] or auto-vectorization [7], we want to gather information about the data itself to enable further optimizations. Fig. 2 depicts the overall process:

1) Collection of run-time metrics is implemented in a joint project. There, we extend Truffle to enable object-level analysis without additional user input and accumulate metrics such as selectivity, access counters, and similar meta-data.

2) Truffle's partial evaluator optimizes the AST and inlines most operations [11], transforming it into the intermediate representation *Graal IR* [16]. Hence, it acts as the primary interface between the interpreter and the compiler. As the AST represents control flow, but the collected information is data-based, we are limited to pattern detection and evaluation of *stable* information, e.g., uniformly observed data types or accumulated query metrics (selectivity, costs).

3) Most optimization work is done in the compiler. At this point, query patterns have been determined and corresponding metrics should be available, subsequently allowing us to aggressively speculate on the shape of the data when observing large, homogeneous data structures. In filter queries, we can reorder conditions to prioritize highly selective predicates. On previously transformed data structures, we can optimize accesses to remove any overhead and can use hardware-specific instructions. This includes extensive vectorization, to perform simultaneous computations on multiple data in linear memory. We can restructure nested loops to improve cache performance and further boost vectorization. If any of our assumptions are invalidated, we furthermore have to be able to return to the interpreter and to revert optimizations.

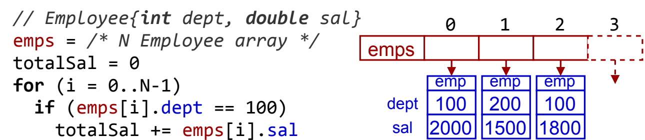

Figure 3. Querying an array of structures

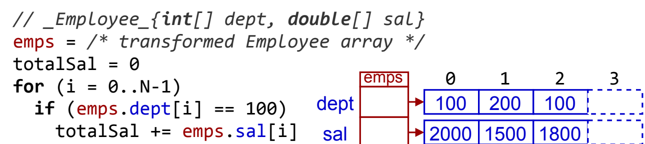

Figure 4. Querying a structure of arrays

Prototype: Storage Transformation. For initial experiments, we implemented a prototype in *SimpleLanguage* [24]. SimpleLanguage is a Truffle research language that exhibits typical characteristics of purely dynamic, object-oriented languages, while also addressing most technologies and components within the Truffle framework. This prototype handles patterns such as depicted in Fig. 3, where a loop aggregates the *salaries* of an array of `Employee` objects for a specific *department*. We can optimize this by transforming the underlying storage representation from an *array of structures (AoS)* to a *structure of arrays (SoA)* [25]. Therefore, we provide an array implementation that tracks accesses to stored objects. When observing a large number of read accesses, an AoS transforms itself by generating *property arrays* from the individual objects' properties. Subsequent access to the array is then rewired, effectively resulting in a query and memory layout as in Fig. 4. The benefits of this representation stem from the linear arrangement of the properties: Whereas the original representation involves object field accesses (neither in linear memory, nor cache-efficient), looping over object properties now effectively means iterating over arrays, thus ensuring better cache locality and enabling vectorization.

While transformation of the data takes place in the interpreter, our part of the project is to optimize the access to this data. Hence, we ensure that for any array access strictly either the fully optimized or the unoptimized version is compiled, with a state change (storage transformation) resulting in deoptimization and recompilation. This allows us to amortize the analysis overhead, e.g., by preventing bounds checks or `null` checks for the property arrays. In Fig. 4, this also means that we can extract loop-invariant parts (e.g. loading the base addresses of `emps.sal`, `emps.dept`). We perform

most of this work using compiler intrinsics as well as experimental compiler phases, detecting accesses into transformed structures within Graal IR by leveraging the known types and structures used in the transformed representations.

Evaluation Methodology. Work on this project is currently driven by improving performance on hand-crafted SimpleLanguage microbenchmarks. These focus on operations on large in-memory datasets of similar objects (akin to in-memory query processing). Figure 5 contains early results of this evaluation and shows the relative speedups of our implementation (cf. Section 3) in microbenchmarks over two query types. We measure the performance with and without compiler optimizations and for each also show the speedup without initial transformation overhead. This suggests that we are able to amortize the analysis costs via compiler optimizations. In the course of this project, we also aim at evaluating our approach with other data-centric open-source benchmarks running on Truffle languages.

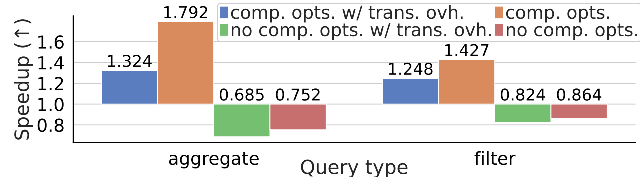

Figure 5. SimpleLanguage microbenchmark results

4 Conclusion

Our project aims at leveraging analysis of large datasets in a dynamic compiler. By shifting the analysis part into the interpreter, we want to foster, simplify and speed up compiler optimizations, primarily focusing on data-heavy operations and queries. Our main goal is to achieve performance improvements for in-memory data processing applications—especially in dynamic languages—by speculating on the properties of data structures and queries, subsequently utilizing vectorization, loop transformations, and other platform-specific optimizations. Using Truffle, we aim to extend this process to polyglot applications, performing analysis and optimizations across language boundaries.

References

[1] X. Jin et al. 2015. Significance and challenges of big data research. *Big Data Research*. Visions on Big Data, 59–64. ISSN: 2214-5796. DOI: 10.1016/j.bdr.2015.01.006.

[2] A. Abelló. 2015. Big data design. In (DOLAP '15). ACM, New York, NY, USA, 35–38. ISBN: 978-1-4503-3785-4. DOI: 10.1145/2811222.2811235.

[3] S. F. Oliveira, K. Fürlinger, and D. Kranzlmüller. 2012. Trends in computation, communication and storage and the consequences for data-intensive science. In IEEE, 572–579. DOI: 10.1109/HPCC.2012.83.

[4] D. Das et al. 2015. Query optimization in oracle 12c database in-memory. *Proc. VLDB Endow.*, 1770–1781. ISSN: 2150-8097. DOI: 10.14778/2824032.2824074.

[5] G. Graefe et al. 2014. In-memory performance for big data. *Proc. VLDB Endow.*, 37–48. ISSN: 2150-8097. DOI: 10.14778/2735461.2735465.

[6] J. R. Allen et al. 1983. Conversion of control dependence to data dependence. In (POPL '83). ACM, New York, NY, USA, 177–189. ISBN: 978-0-89791-090-3. DOI: 10.1145/567067.567085.

[7] G. M. Duboscq. 2016. *Combining speculative optimizations with flexible scheduling of side-effects*. PhD thesis. Linz, April 2016.

[8] A. Simbürger et al. 2019. PolyJIT: polyhedral optimization just in time. *Int J Parallel Prog*, 874–906. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-018-0597-3.

[9] L. Stadler, T. Würthinger, and H. Mössenböck. 2014. Partial escape analysis and scalar replacement for java. In (CGO '14). ACM, Orlando, FL, USA, 165–174. ISBN: 978-1-4503-2670-4. DOI: 10.1145/2581122.2544157.

[10] D. Leopoldseder et al. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In ACM Press, Vienna, Austria, 126–137. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168811.

[11] T. Würthinger et al. 2013. One VM to rule them all. In (Onward! 2013). ACM, Indianapolis, Indiana, USA, 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.

[12] C. Wimmer and T. Würthinger. 2012. Truffle: a self-optimizing runtime system. In (SPLASH '12). ACM, Tucson, Arizona, USA, 13–14. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384723.

[13] J. Kreindl et al. 2020. Multi-language dynamic taint analysis in a polyglot virtual machine. In (MPLR 2020). ACM, New York, NY, USA, 15–29. ISBN: 978-1-4503-8853-5. DOI: 10.1145/3426182.3426184.

[14] S. Kandel et al. 2012. Enterprise data analysis and visualization: an interview study. *IEEE Transactions on Visualization and Computer Graphics*, 2917–2926. ISSN: 1941-0506. DOI: 10.1109/TVCG.2012.219.

[15] H. V. Jagadish et al. 2014. Big data and its technical challenges. *Commun. ACM*, 86–94. ISSN: 0001-0782. DOI: 10.1145/2611567.

[16] G. Duboscq et al. 2013. Graal IR: an extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. Shenzhen, China, 9.

[17] T. Gross and P. Steenkiste. 1990. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software: Practice and Experience*, 133–155. ISSN: 1097-024X. DOI: 10.1002/spe.4380200203.

[18] H. Wang, D. Padua, and P. Wu. 2015. Vectorization of apply to reduce interpretation overhead of r. *SIGPLAN Not.*, 400–415. ISSN: 0362-1340. DOI: 10.1145/2858965.2814273.

[19] J. Talbot, Z. DeVito, and P. Hanrahan. 2012. Riposte: a trace-driven compiler and parallel VM for vector code in r. In ACM Press, Minneapolis, Minnesota, USA, 43. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370825.

[20] H. Wang, P. Wu, and D. Padua. 2014. Optimizing r VM: allocation removal and path length reduction via interpreter-level specialization. In ACM, Orlando FL USA, 295–305. ISBN: 978-1-4503-2670-4. DOI: 10.1145/2544137.2544153.

[21] R. Baghdadi et al. 2019. Tiramisu: a polyhedral compiler for expressing fast and portable code. In (CGO 2019). IEEE Press, Washington, DC, USA, 193–205. ISBN: 978-1-72811-436-1.

[22] W. Zhang et al. 2021. Adaptive code generation for data-intensive analytics. *Proc. VLDB Endow.*, 929–942. ISSN: 2150-8097. DOI: 10.14778/3447689.3447697.

[23] J. W. Davidson and S. Jinturkar. 1996. Aggressive loop unrolling in a retargetable, optimizing compiler. In (Lecture Notes in Computer Science). T. Gyimóthy, editor. Springer, Berlin, Heidelberg, 59–73. ISBN: 978-3-540-49939-8. DOI: 10.1007/3-540-61053-7_53.

[24] Oracle. [n. d.] GraalVM - introduction to SimpleLanguage. GitHub. Retrieved 07/05/2021 from https://www.graalvm.org/graalvm-as-a-platform/implement-language/.

[25] Intel. 2010. A guide to vectorization with intel® c++ compilers. (2010).