Automatically Transforming Arrays to Columnar Storage at Run Time[*]

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

Lukas Makor[†]
Johannes Kepler University
Linz, Austria
lukas.makor@jku.at

David Leopoldseder
Oracle Labs
Austria
david.leopoldseder@oracle.com

Daniele Bonetta
Oracle Labs
Netherlands
daniele.bonetta@oracle.com

Lukas Stadler
Oracle Labs
Austria
lukas.stadler@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Picking the right data structure for the right job is one of the key challenges for every developer. However, especially in the realm of object-oriented programming, the memory layout of data structures is often still suboptimal for certain data access patterns, due to objects being scattered across the heap. Therefore, this work presents an approach for the automated transformation of arrays of objects into a contiguous format (called *columnar arrays*). At run time, we identify suitable arrays, perform the transformation and use a dynamic compiler to gain performance improvements. In the evaluation, we show that our approach can improve the performance of certain queries over large, uniform arrays.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic compilers**; **Runtime environments**; Interpreters; • **Information systems** → *Column based storage.*

KEYWORDS

Columnar Storage, Array Storage, Program optimization, Dynamic Language, Dynamic Compilation

ACM Reference Format:
Sebastian Kloibhofer, Lukas Makor, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. 2022. Automatically Transforming Arrays to Columnar Storage at Run Time. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3546918.3560805

1 MOTIVATION

In object-oriented programs, we tend to store data as objects and collect groups of such objects in arrays. While arrays are common and easy to use for developers, the scattered layout of the array elements (the referenced objects may be placed randomly across the heap) has disadvantages for certain computational patterns: If we access individual object properties in a loop (as in Listing 1), no caching can take place across iterations since arbitrary memory positions have to be accessed [2]. The memory layout of such a data structure is shown on the left-hand side in Fig. 1.

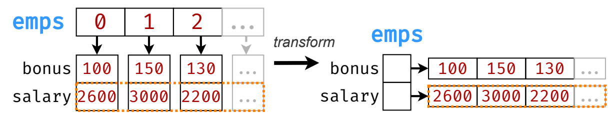

Figure 1: Array of objects vs. columnar array memory layout

Columnar arrays are one solution to this problem: In a columnar array, the elements' property values are grouped in contiguous memory regions (right-hand side of Fig. 1), such that an object in the array at position i has its property values located at the i-th positions in the resulting arrays ($bonus[i]$, $salary[i]$). Listing 2 shows an optimized version of Listing 1 that uses a columnar array. Accessing the array in the loop now causes the property values of adjacent elements to be cached, thus improving performance.

```
let total = 0
for (let i = 0; i < emps.length; i++)
  total += emps[i].salary
```

Listing 1: Salary aggregation over an array of employees

```
let total = 0
const salary = emps.salary
for (let i = 0; i < emps.length; i++)
  total += salary[i] // replaces "emps[i].salary"
```

Listing 2: Salary aggregation with a columnar array

While columnar arrays are often used in databases [1, 4, 16], research has also highlighted their benefits for traditional applications. Mattis et al. [9] published a columnar array API for Python, where the JIT compiler subsequently optimizes accesses. Pivarski et al. [13] as well as Homann and Laenen [6], respectively, developed a similar approach for C++—the former performing optimizations on the Abstract Syntax Tree (AST) level and the latter optimizing particle simulations. We detect applicable arrays at run time and automatically transform them into columnar arrays, thus requiring no code adaptation by the developer. We gain performance benefits by performing custom compiler optimizations on columnar array accesses during JIT compilation.

[*]This research project is partially funded by Oracle Labs.

[†]Both authors contributed equally to the paper

MPLR '22, September 14–15, 2022, Brussels, Belgium

ACM ISBN 978-1-4503-9696-7/22/09.
https://doi.org/10.1145/3546918.3560805

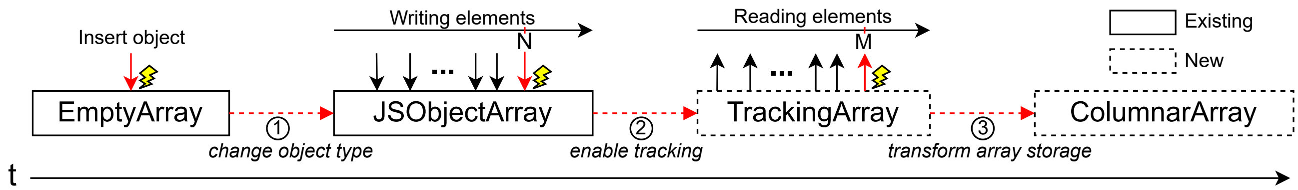

Figure 2: Integration of the new array storage strategies that enable tracking and storage transformation

2 APPROACH

In this work, we summarize our approach [8] that we implemented in the GraalVM JavaScript runtime [10]. This language implementation is based on the Truffle framework [5, 17, 18], which enables integration of guest languages [10–12, 14] via AST interpretation [19]. The GraalVM Compiler [3, 7, 15] enables us to implement custom compiler phases based on columnar array accesses.

As JavaScript arrays are highly dynamic (they may contain arbitrary elements, may have holes, and may not start at index 0), their representation in GraalVM JavaScript is described by an internal *storage strategy* that defines, how the array elements are laid out in memory. We leveraged this characteristic for the integration of our approach by developing additional storage strategies, which are depicted in Fig. 2.

After a new (empty) array is initialized, it is first assigned a built-in default strategy (*EmptyArray*). When new elements are inserted, a new strategy is assigned based on the element kinds. The built-in *JSObjectArray* strategy indicates an array of objects that starts at index 0 and ensures that the array does not have holes. We adapted this strategy to track the size of arrays and trigger a strategy change when the size exceeds a configurable threshold (50000 by default). The new *TrackingArray* strategy then performs more sophisticated (albeit costlier) tracking by counting the number of array read accesses. This separation allows us to minimize the tracking overhead for smaller arrays. After the read count exceeds a second configurable threshold (25000 by default), the array is automatically transformed to a columnar layout (*ColumnarArray* strategy). During transformation, we verify that all elements in the array have the same type. Then, we allocate arrays for each property of the original array elements—we denote them as *property arrays*—and fill them with the property values. This results in a memory layout similar to the one in Fig. 1 (right-hand side).

While the transformation and the new array strategies are integrated into the language implementation, performance benefits are actually gained in the compiled code via custom compiler phases that detect columnar arrays and their accesses. Due to our knowledge about the columnar data structure, we can remove a number of checks that the compiler would normally introduce. In the columnar layout, we furthermore skip loading the object from the array at compile time (`emps[i]`), as the property values are directly accessed from the property array (`total += salary[i]`). Additionally, loading the property array from the columnar array (`emps.salary`) is loop-invariant, hence most of the instructions are lifted from the loop to achieve performance improvements in each loop iteration. For the example in Listing 1, these optimizations result in a representation similar to Listing 2 after compilation.

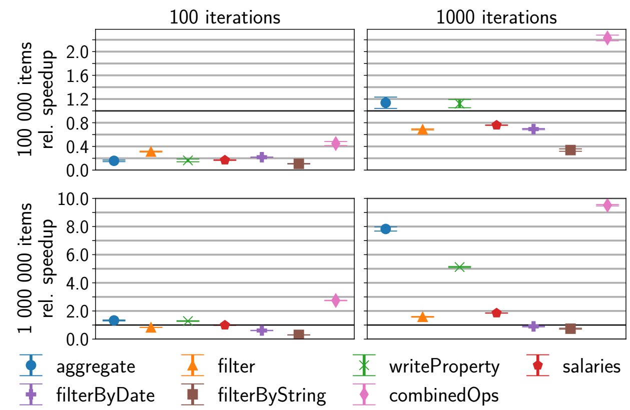

Figure 3: Microbenchmark throughput relative to baseline without storage transformation *(higher is better)*

3 EVALUATION

To show the capabilities of our approach we designed a number of JavaScript microbenchmarks that each implement a specific query on a large array [1]. As depicted in Fig. 3, we executed all microbenchmarks with different array sizes (100K and 1M items) and different numbers of traversals of the whole array (100 and 1000 iterations).

Fig. 3 shows that our approach is currently not suitable for smaller arrays, due to the overhead introduced by the transformation process. As array sizes and array traversals increase, however, the transformation overhead is amortized and the performance of some benchmarks improves significantly. The benchmarks `aggregate`, `writeProperty`, and `combinedOps` benefit the most, with speedups of over 7x, 5x, and 9x, respectively. While the performance of `filter` and `salaries` is improved as well, benchmarks that use complex properties for filtering (`filterByDate`, `filterByString`) are not or negatively impacted.

4 CONCLUSION

In this work, we developed an approach for automated storage transformation in JavaScript that creates columnar arrays from arrays of objects. As a consequence, we can speed up accesses to these arrays. Hence, our approach is especially suited for processing object arrays with loops. An evaluation of our approach on a set of microbenchmarks shows that we can achieve significant speedups on bulk operations on large arrays, while suffering from the transformation overhead on smaller or more complex operations.

[1]Source code: https://github.com/lmakor-jku/data-intensive-js-benchmarks

REFERENCES

[1] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, Asilomar, CA, USA, 225–237.

[2] Ulrich Drepper. 2007. What Every Programmer Should Know about Memory. *Red Hat, Inc* 11 (2007), 2007.

[3] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. Shenzhen, China, 9.

[4] Amit Dwivedi, C. Lamba, and Shweta Shukla. 2012. Performance Analysis of Column Oriented Database Vs Row Oriented Database. *International Journal of Computer Applications* 50 (July 2012), 31–34. https://doi.org/10.5120/7841-1050

[5] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. Association for Computing Machinery, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[6] Holger Homann and Francois Laenen. 2018. SoAx: A Generic C++ Structure of Arrays for Handling Particles in HPC Codes. *Computer Physics Communications* 224 (March 2018), 325–332. https://doi.org/10.1016/j.cpc.2017.11.015

[7] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. ACM Press, Vienna, Austria, 126–137. https://doi.org/10.1145/3168811

[8] Lukas Makor, Sebastian Kloibhofer, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. 2022. Automatic Array Transformation to Columnar Storage at Run Time. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR 2022)*. Association for Computing Machinery, Brussels, Belgium. https://doi.org/10.1145/3546918.3546919

[9] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. 2015. Columnar Objects: Improving the Performance of Analytical Applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, Pittsburgh PA USA, 197–210. https://doi.org/10.1145/2814228.2814230

[10] Oracle. 2021. Graal.Js. https://github.com/graalvm/graaljs. (accessed 2020-09-09).

[11] Oracle. 2021. GraalPython. https://github.com/graalvm/graalpython. (accessed 2020-09-09).

[12] Oracle. 2021. TruffleRuby. https://github.com/oracle/truffleruby. (accessed 2020-09-09).

[13] Jim Pivarski, Peter Elmer, Brian Bockelman, and Zhe Zhang. 2017. Fast Access to Columnar, Hierarchically Nested Data via Code Transformation. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, Boston, MA, USA, 253–262. https://doi.org/10.1109/BigData.2017.8257933

[14] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. 2016. Sulong - Execution of LLVM-based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems - ICOOOLPS '16*. ACM Press, Rome, Italy, 1–4. https://doi.org/10.1145/3012408.3012416

[15] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, Orlando, FL, USA, 165–174. https://doi.org/10.1145/2581122.2544157

[16] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, Trondheim, Norway, 553–564.

[17] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, Tucson, Arizona, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[18] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, Indianapolis, Indiana, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[19] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587