Memory Layout Optimisation on Abstract Syntax Trees

Impact on Utilisation Speed During Type Checking and Code Generation Phases

Iannis de Zwart
Supervisors: Soham Chakraborty, Dennis Sprokholt
EEMCS, Delft University of Technology, The Netherlands

June 22, 2024

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Name of the student: Iannis de Zwart
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Burcu Ozkan

Abstract

In the field of software engineering, the speed of compilation plays a crucial role in enhancing development productivity. This thesis investigates the impact of optimising the memory layout of Abstract Syntax Trees (ASTs) on the performance of the type checking and code generation phases in the compilation process of strictly-typed procedural programming languages. Existing compilers often employ a traditional Object-Oriented (OO) approach to AST construction, leading to inefficiencies such as high memory overhead and suboptimal cache usage. We applied Data-Oriented Design (DOD) principles to restructure ASTs, aiming to enhance data locality and reduce memory access times.

The study investigates various AST memory layouts, including a transition from a naive OO model to a Struct-of-Arrays (SoA) model. We evaluated the effect of these memory layouts on compiler performance using a set of benchmarks based on real-world code. We executed the benchmarks on diverse hardware platforms, measuring key performance metrics such as type checking time, code generation time, and cache miss rates.

The results demonstrate that adopting an SoA model for ASTs significantly reduces the duration of the type checking and code generation phases, with improvements varying across different hardware architectures. We provide empirical evidence supporting the application of DOD principles and specifically SoA to ASTs for performance improvements. By highlighting and addressing the performance bottlenecks associated with traditional AST layouts, we contribute to the broader goal of advancing compiler design and optimisation.

1 Introduction

A compiler operates through multiple phases that work together to transform source code into executable programs [1]. In the scope of this research, we consider a simplified phase model for a compiler designed for a strictly-typed programming language:

1. **Parsing**: In this first phase, the compiler creates an Abstract Syntax Tree (AST) from source code, capturing its semantic structure.

2. **Type-checking**: In this subsequent phase, the compiler determines the type of each syntactic element using the AST. It also verifies correctness based on the programmer-provided type declarations, and enriches the AST with additional type information to support the final phase.

3. **Code-generation**: In this final phase, the compiler produces platform-dependent executable code from the enriched AST.

Optimising the AST memory layout is imperative for achieving efficient compilation, given the central role of the AST data structure in the compilation process. A notable practical advance in this area includes the memory layout restructuring of the AST in the Zig compiler, which led to a performance improvement of over 30% during the type checking and IR generation phases [2]. Andrew Kelley, the author of the Zig compiler, drew inspiration from Richard Fabian's Data-Oriented Design book [3] and applied these principles to the AST layout in the Zig compiler. Despite these practical achievements, scientific research in the field of AST memory layout optimisation remains very limited, resulting in a knowledge gap where compiler architects often rely on trial-and-error to determine effective ways to store AST data.

We aim to highlight the significance of optimising AST memory layouts and provide guidance to compiler architects in designing memory-efficient ASTs. By exploring various AST memory layout alternatives and their influences on compilation speed, this study will offer quantitative, empirical, and reproducible insights for future advancements in the field.

Our primary research question is: **How does the application of Data-Oriented Design principles on Abstract Syntax Trees affect the speed of the type checking and code generation phases for compilation of procedural programming languages?**

2 Background

In modern computers, a significant discrepancy exists between the access times of the CPU data cache and main memory. Consequently, minimising fetches from main memory is crucial, as these can cause severe stalls in the CPU's fetch-decode-execute cycle. [4]

High information density within cache lines during execution is essential to ensure that each fetch operation maximises the data used by the program [5]. In the scope of this research, we define information density as the percentage of bytes in cache lines that are actually used by an operation performed on that data. A higher information density generally ensures superior performance, since it minimises memory access operations. In *Data-Oriented Design* (DOD), the programmer is mindful of the layout of data, and how it is accessed and transformed by operations. The goal is to enhance information density, ensuring optimal pipelining of multi-data operations. [3]

A naive AST might be implemented using an idiomatic Object-Oriented approach, where each node of the tree is dynamically allocated and stores all data related to the syntactic element, along with pointers to its children [6]. This approach contrasts sharply with DOD principles because of the following reasons:

- Dynamic allocation of individual nodes results in memory overhead due to the memory allocation algorithm [7], decreasing information density. Examples of such overheads are heap block headers, interblock padding, and heap fragmentation [8] [9].

- The arbitrary ordering of the dynamically allocated nodes can lead to a higher number of memory fetches, because access patterns are not guaranteed. It can also result in an elevated level of cache misses compared to traversing the AST via pointers to children, since dereferencing is frequent. [10]
- Some data in an AST node is redundant in specific compilation phases. Additionally, some data is only necessary in edge cases and cold execution paths. Both of these conditions attribute to sub-optimal information density.
- Nodes may contain fields of different sizes, necessitating padding between fields to maintain alignment, further decreasing information density [9].

To address these issues, the Zig compiler developers transitioned their AST implementation from a naive idiomatic Object-Oriented tree (see Figure 1) to a Data-Oriented *Struct-of-Arrays* (SoA) model (see Figure 2). This change significantly improved information density and cache efficiency of their AST traversal.

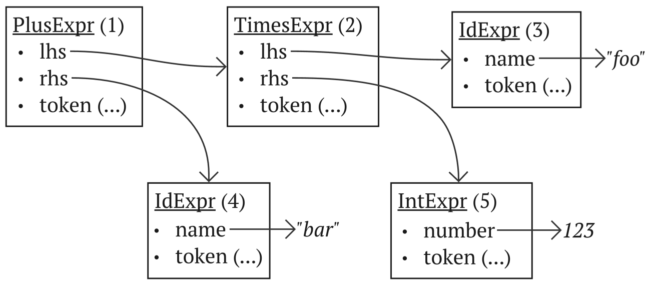
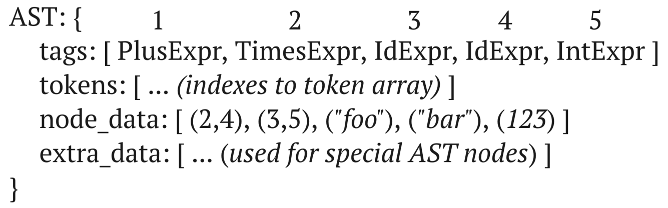


Figure 1: Naive Object-Oriented AST. (Simplified). The expression that is represented by this AST is:
`((foo * 123) + bar)`.



```
AST: {      1            2           3       4       5
   tags: [ PlusExpr, TimesExpr, IdExpr, IdExpr, IntExpr ]
   tokens: [ ... (indexes to token array) ]
   node_data: [ (2,4), (3,5), ("foo"), ("bar"), (123) ]
   extra_data: [ ... (used for special AST nodes) ]
}
```

Figure 2: The same AST as in Figure 1, implemented using a Data-Oriented Struct-of-Arrays model. (Simplified).

Using a Struct-of-Arrays model effectively addresses padding and memory overhead issues associated with memory allocation, as the arrays are stored in contiguous memory blocks. Additionally, it reduces inefficiencies caused by imbalances between frequently accessed ("hot") and infrequently accessed ("cold") struct fields [5]. By accessing cold fields only on demand, they are kept separate from hot data, thereby improving overall information density. By placing all data within single arrays, the SoA model introduces a well-defined data ordering, countering the arbitrary nature of dynamic memory allocation. However, to further enhance performance, optimising the order of data within these arrays to reflect access patterns can yield additional performance gains. [3]

In this research, we introduce and implement such an SoA model for ASTs. In our model, each AST node has its own index. The *tags* array stores the types of each AST node, the *tokens* array stores line and column references to the source code of each AST node, and the *node_ data* array stores fixed-sized data which is interpreted differently depending on the type of AST node. The *extra_ data* array stores extra node data on demand, if the *node_ data* object itself is not big enough. (E.g. an AST node for a function call may need to store a large number of arguments).

3 Methodology

To conduct a quantitative analysis and address the research question, we utilised a benchmarking approach, involving different compiler implementations with various Abstract Syntax Tree (AST) layouts. With this approach, we measured the impact of AST layout modifications on the performance of the type-checking and code generation phases during compilation.

Our methodology comprises the following steps:

1. **Selection of benchmark input**: We collected sample C programs from open-source repositories. C is a procedural, compiled, systems-level programming language, allowing for a consistent and representative dataset of code for the research. The full dataset can be found in Section 3.1.
2. **Transpilation to Tea language**: The C programs were run through the C preprocessor [11]. Then, they were converted to Tea language[1] code using a transpiler application[2] we specifically developed for this research. We chose the Tea language as the target language for the following reasons:
   - Tea reveals similarity to C as a procedural language with direct memory access. This similarity facilitates a relatively lossless transpilation process, as most C language constructs are supported in Tea. Consequently, Tea serves as an effective proxy language for benchmarking real-world C code.
   - The Tea compiler program is notably less complex than traditional C compilers, comprising

[1]`https://github.com/iannisdezwart/tea`
[2]`https://github.com/iannisdezwart/c-to-tea-transpiler`

a compact codebase of just 7.6k lines. This compact size ensured we could complete the research in a timely manner, as it enabled easier modifications and implementation of AST layout optimisations. Additionally, the simplicity of the codebase made alternative memory layout implementations more readable, aiding compiler architects in understanding the code impact of specific implementations. This understanding helps them evaluate the feasibility of adopting similar layout modifications in compilers they are working on.

3. **Designing alternative AST layouts**: We designed and implemented various AST layouts based on DOD principles to optimise memory accesses and data efficiency. The layout alternatives are explained in detail in Section 3.2.

4. **Performance evaluation**: We compiled the Tea code on a set of benchmarking machines. (See Section 3.3). We selected various machines to account for variations in instruction set architecture, CPU, memory timings, and other OS and hardware-level optimisations that could influence performance outcomes.

5. **Data collection and analysis**: We gathered various performance metrics (as detailed in Section 3.4) during the benchmarking process. We subsequently analysed the collected data to identify and establish relationships between different AST layouts and their impact on performance. All data was systematically collected using a data repository[3] we specifically designed for this research, and subsequently visualised in generated graphs.

6. **Formulation of recommendations**: Following the analysis, we drew conclusions regarding the effectiveness of the different AST layouts on compilation performance. These insights provide a foundation for offering recommendations to compiler architects, guiding them towards the optimal implementation of ASTs in their projects.

3.1 Benchmark dataset

The complete benchmark dataset[4], consisting of Tea programs, is presented in Table 1. These programs were generated by transpiling the corresponding C code.

3.2 AST layout alternatives

We implemented and benchmarked the following AST layout implementations, each building upon the implementation preceding it in the list:

[3]https://github.com/iannisdezwart/rp-data

[4]https://github.com/iannisdezwart/c-to-tea-transpiler/tree/main/transpiled_code

Program	Source code size
chibicc_parse.tea	126 kB
chibicc_combined.tea	229 kB
gzip.tea	150 kB
zlib.tea	221 kB

Table 1: Transpiled Tea programs that were benchmarked, and their sizes.

- **No optimisations (original Tea compiler)**[5]: The original Tea compiler features a naive Object-Oriented AST structure consisting of 24 nodes (Listed in Appendix A) for different language constructs.

- **Compact tokens**[6]: During the tokenisation phase of the original Tea compiler, tokens are generated with token value fields that are only relevant during parsing and become unnecessary during the subsequent type checking and code generation phases. This optimisation introduces a `CompactToken`, which is used during the type checking and code-generation phases. This `CompactToken` excludes the aforementioned token value fields, leaving only the line and column numbers. This results in a size reduction of the base `ASTNode` struct from 128 to 88 bytes.

- **Class IDs**[7]: In the original Tea compiler, class names are stored as strings and accessed through string keys. To reduce memory usage, this optimisation introduces a mapping between class names and their IDs, eliminating the need for storing them as strings. This optimisation reduced the base `ASTNode` further to 72 bytes.

- **Compact types**[8]: In the original Tea compiler, built-in types and user-defined class types were handled separately, requiring distinct fields in the `Type` struct. This optimisation introduces IDs for all built-in types and introduces a dynamic ID creation scheme for user-defined types. Additionally, we extracted a cold dynamically allocated array containing pointer depth information from the `Type` struct and stored into an external unified memory pool. With this optimisation, the size of the `Type` struct was reduced from 48 bytes to just 16 bytes, thereby reducing the `ASTNode` struct further to a mere 40 bytes.

- **Struct-of-Arrays**[9]: This optimisation replaces the original Tea compiler's Object-Oriented AST struc-

[5]https://github.com/iannisdezwart/tea/tree/main

[6]https://github.com/iannisdezwart/tea/tree/compact-token

[7]https://github.com/iannisdezwart/tea/tree/class-ids

[8]https://github.com/iannisdezwart/tea/tree/compact-types

[9]https://github.com/iannisdezwart/tea/tree/struct-of-arrays

ture with a Struct-of-Arrays model. The new AST implementation utilises several arrays, including:

- `tags`: Maps each AST node to a 1-byte enum value, indicating the node type.
- `node_data`: Maps each AST node with an 8-byte union data structure, interpreted differently based on the node's type.
- `tokens`: Maps each AST node to an 8-byte structure containing its line and column numbers. This cold array is only used for error reporting.
- `extra_data`: A memory pool containing additional information for specific nodes beyond the 8 bytes in `node_data`, referenced from entries in `node_data`.
- `types`: Maps each AST node to a 12-byte type information structure, constructed during the type-checking phase, and utilised during the code-generation phase.

3.3 Benchmark machines

The full list of benchmark machines is shown in Table 2.

Processor	ISA	Platform	Compiler
Apple M2 Pro 12-core	ARM	MacOS 14.4.1	Clang 18.1.5
ARM Cortex-X1	ARM	Android 14	Clang 18.1.7
Intel Xeon E5-2630	x86	Ubuntu 20.04	GCC 9.3.0
AMD Ryzen 7-7745HX	x86	Ubuntu 24.04	GCC 13.2.0
Intel Core i7-8650U	x86	Ubuntu 24.04	GCC 14.1.1
Intel Core i7-10750H	x86	Fedora 40	GCC 11.4.0

Table 2: Machines the benchmark was run on.

3.4 Performance metrics

We used the following tools to collect performance metrics for analysis:

- **Hardware clock**: We used the C++ `std::chrono` library [12] to measure high resolution time deltas within the Tea compiler. We measured the following time performance results:
  - **Type check time (mean & standard deviation)**: Samples of duration of the the type-check phase.
  - **Code gen time (mean & standard deviation)**: Samples of the duration of the code-gen phase.
- **Linux `pidstat`** [13]: A command-line utility used for monitoring Linux kernel tasks. `pidstat` was used to collect memory usage results.
- **Valgrind** [14]: A Linux programming tool for memory debugging and profiling. Valgrind was used to collect the following cache efficiency data:
  - **IR references**: instruction reads.
  - **IR1 miss rate**: instruction read level 1 cache misses.
  - **D references**: references to memory.
  - **D1 miss rate**: data level 1 cache misses.
  - **LLd miss rate**: last level data cache misses.

4 Results

We produced results across three main categories:

- **Time performance results**: Each machine in the benchmarking set compiled the dataset of transpiled C code using every optimisation candidate, facilitated by a dedicated benchmarking script[10]. Detailed results are presented in Section 4.1.
- **Memory usage results**: We measured memory consumption by different AST layouts for each optimisation candidate. Memory usage was measured using a benchmarking script[11], which recorded measurements before the parsing phase and after the type-checking phase using `pidstat`. The difference of the measurements was taken as the total AST size. Detailed results are presented in Section 4.2.
- **Cache efficiency results**: We evaluated cache utilisation efficiency for each optimisation candidate using a benchmarking script[12] that runs Valgrind for detailed analysis. Detailed results are presented in Section 4.3

4.1 Time performance results

4.1.1 Type-checking phase

We measured the mean duration of the type-checking phase across all AST optimisation candidates. We then compared the results to the baseline duration without any optimisations.

[10] `https://github.com/iannisdezwart/tea/blob/main/bench.sh`

[11] `https://github.com/iannisdezwart/tea/blob/main/bench-mem.sh`

[12] `https://github.com/iannisdezwart/tea/blob/main/bench-cache.sh`

Figure 3 presents the mean speedup across all platforms and programs in the benchmark set for each optimisation candidate, including the standard deviation. Figure 4 presents the same results as Figure 3, grouping them by the different machines in the benchmark set. Figure 5 presents the same results as Figure 3, grouping them by the different programs run in the benchmark set.

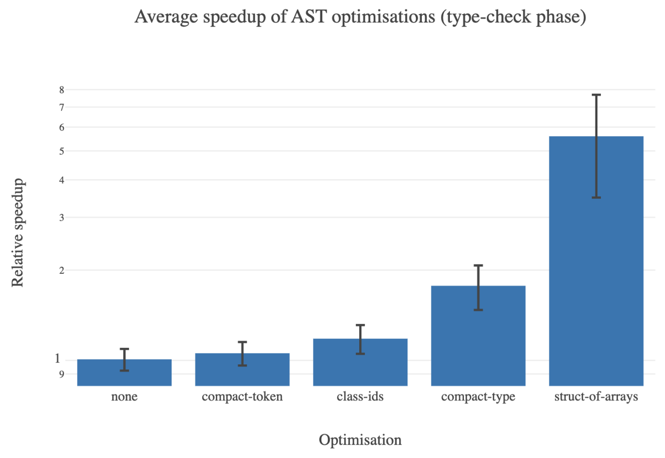


Figure 3: Relative average speedup obtained in type-checking phase by applying different optimisations. Logarithmic y-axis.

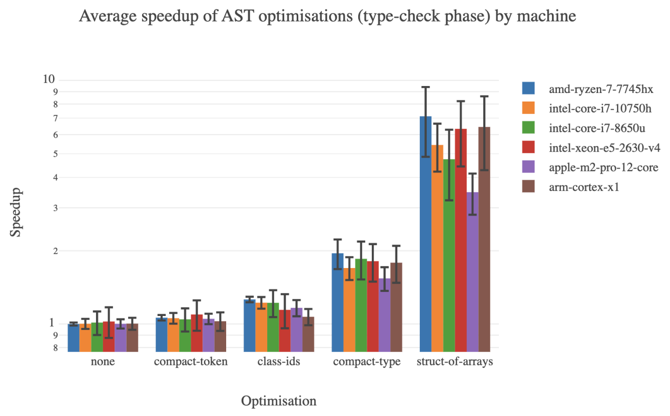


Figure 4: Relative average speedup obtained in type-checking phase by applying different optimisations, grouped by machine. Logarithmic y-axis.

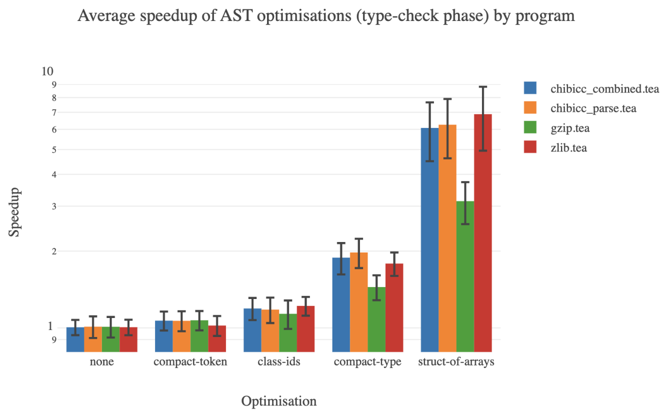


Figure 5: Relative average speedup obtained in type-checking phase by applying different optimisations, grouped by program. Logarithmic y-axis.

4.1.2 Code-generation phase

We measured the mean duration of the code-generation phase across all AST optimisation candidates. We then compared the results were to the baseline duration without any optimisations.

Figure 6 presents the mean speedup across all platforms and programs in the benchmark set for each optimisation candidate, including the standard deviation. Figure 7 presents the same results as Figure 6, grouping them by the different machines in the benchmark set. Figure 8 presents the same results as Figure 6, grouping them by the different programs run in the benchmark set.

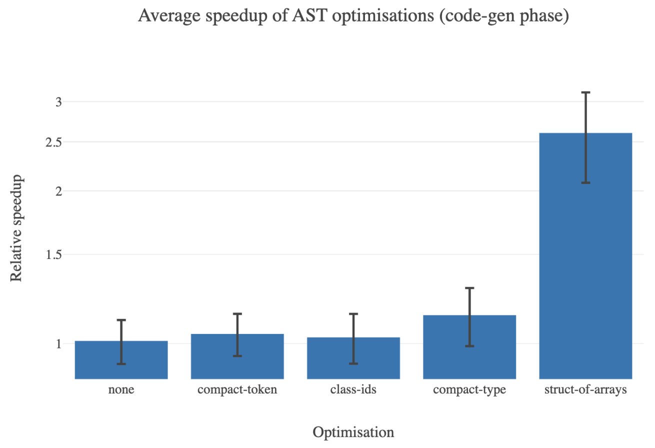


Figure 6: Relative average speedup obtained in code-generation phase by applying different optimisations. Logarithmic y-axis.

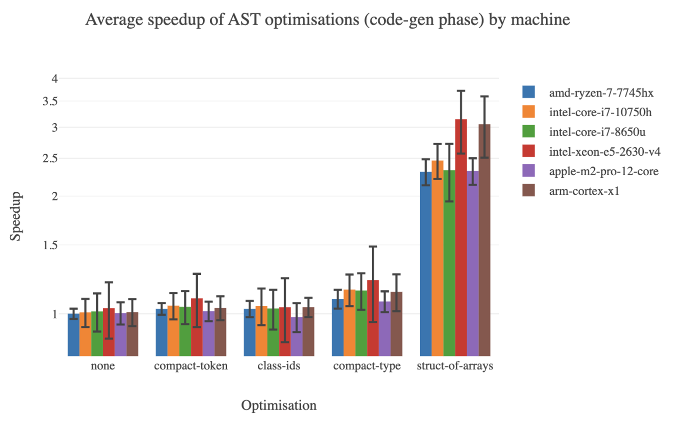


Figure 7: Relative average speedup obtained in code-generation phase by applying different optimisations, grouped by machine. Logarithmic y-axis.

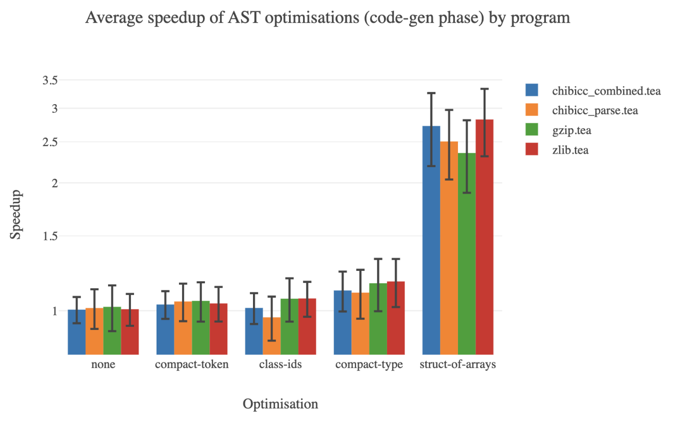


Figure 8: Relative average speedup obtained in code-generation phase by applying different optimisations, grouped by program. Logarithmic y-axis.

4.2 Memory usage results

Figure 9 presents the average size of the AST by program, *including* the information generated during the type-checking phase.

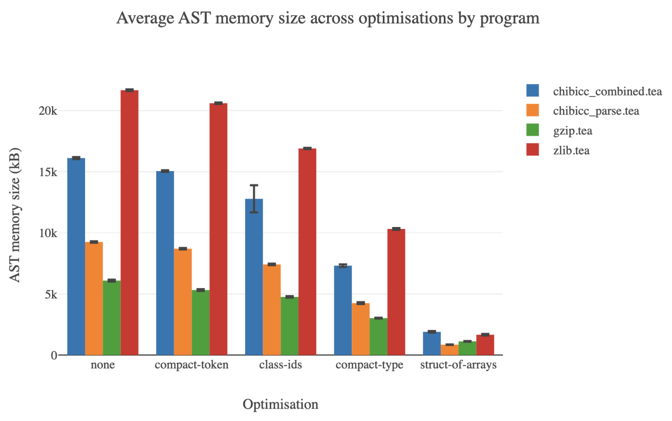


Figure 9: Average AST size including type information for each optimisation, grouped by program.

4.3 Cache efficiency results

Figure 10 presents the cache miss rates by program for each optimisation candidate. The data was gathered using Valgrind.

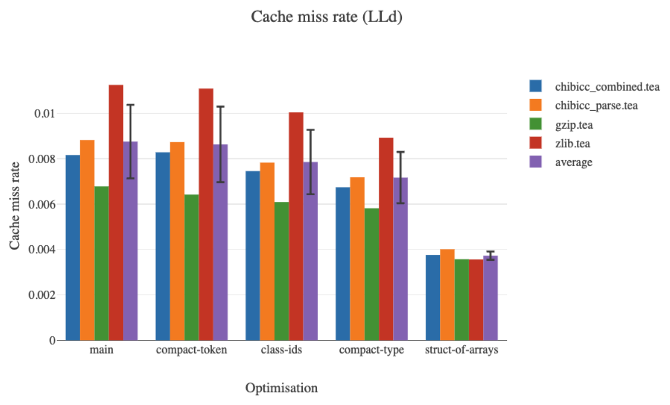


Figure 10: Cache miss rates for each optimisation, grouped by program.

5 Analysis and Discussion

5.1 Type-checking phase performance improvements

The `struct-of-arrays` optimisation significantly enhanced performance during the type-checking phase. Across all platforms, the SoA optimisation yielded an average speedup of 5.6x compared to the original Tea compiler. The performance benefits varied between 3.5x and 7.1x depending on the specific machine, and between 3.1x and 6.9x depending on the test program.

The `compact-type` optimisation also showed notable improvements, with an average speedup of 1.8x. The reduction from a 48-byte to a 16-byte `Type` struct minimised memory writes, and the unified array for pointer depths eliminated many dynamic allocations. This enhanced the performance substantially.

In contrast, the `class-ids` and `compact-token` optimisations provided more modest performance gains, averaging 1.06x and 1.18x speedups, respectively.

5.2 Code-generation phase performance improvements

During the code-generation phase, the `struct-of-arrays` optimisation continued to demonstrate significant performance gains, averaging a 2.6x speedup. Across different machines, this improvement ranged between 2.3x and 3.1x, while across the programs, it ranged between 2.4x and 2.8x.

Conversely, the other optimisations showed minimal performance gains, with some combination even performing slightly worse than the baseline. `compact-type` achieved a modest 1.14x speedup on average.

5.3 Overall performance improvements

From the results, it is evident that the type-checking phase responded more favourably to AST restructurings compared to the code-generation phase. This disparity can be attributed to the nature of inputs and outputs for both phases:

- **Type-checking phase**:
  - Input: parsed AST (controlled by experiment)
  - Output: type information (controlled by experiment)
- **Code-generation phase**
  - Input: parsed AST & type information (controlled by experiment)
  - Output: Tea bytecode (not controlled by experiment)

The optimisations we applied only controlled the data layout of ASTs and types. While our optimisations affected both memory reads and writes in the type-checking phase, they only affected memory reads in the code-generation phase. Despite this limitation, the `struct-of-arrays` optimisation still yielded a promising 2.6x speedup, indicating substantial potential for performance improvements in code generation through AST memory layout optimisation.

5.4 Discrepancies between machines and optimisations

The performance impact of different AST layout optimisations varied significantly across the different machines. Key observations include:

- **Type-checking phase**:

  Across different optimisations in the type-checking phase, the AMD Ryzen 7-7745HX demonstrated the highest average speedup, while the Apple M2 Pro 12-core showed the least benefit. This disparity can be attributed, in part, to differences in memory latency characteristics between these processors. AMD Ryzen CPUs typically exhibit higher memory latency compared to other CPUs with similar performance benchmarks[13]. Conversely, Apple-made CPUs are known for lower memory latency[14], which can enhance performance in scenarios with high cache misses. We would expect a machine with higher memory latency to perform worse at the baseline benchmark, thereby inflating the apparent optimisation effectiveness in the lower cache miss scenarios of the optimisations. Therefore, higher memory latency of the AMD Ryzen 7-7745HX machine aligns well with the higher observed speedup during type-checking.

  Additionally, the less significant performance increase on the Apple M2 Pro 12-core machine could be explained by its modern data memory-dependent prefetcher, which speculatively prefetches memory addresses to improve the latency of referencing operations [15]. These factors may collectively contribute to why the performance difference between no optimisations and the `struct-of-arrays` optimisation is comparatively smaller on the Apple M2 Pro 12-core than on other machines.

- **Code-generation phase** In the code-generation phase, the Intel Xeon E5-2630 v4 and ARM Cortex-X1 machines presented the most significant benefits from the `struct-of-arrays` optimisation, while other machines showed relatively similar speedups across different optimisations. The reasons for this specific performance variation require further investigation.

5.5 Discrepancies between programs and optimisations

The `gzip.tea` program showed the least benefit from the optimisations in terms of time performance across both compilation phases. Specifically, it achieved speedups of 3.1x in the type-checking phase and 2.4x in the code-generation phase with the `struct-of-arrays` (SoA) optimisation compared to the baseline (see Figures 5 and 8). In contrast, the `zlib.tea` program demonstrated the most significant improvements, with speedups of 6.9x and 2.8x in the type-checking and code-generation phases, respectively.

This disparity can be partially attributed to the composition of the `gzip.tea` code, which contains a lower proportion of function definition code compared to other pro-

[13]`https://cpu.userbenchmark.com/Compare/AMD-Ryzen-7-7745HX-vs-Intel-Core-i7-10750H/m2100010vsm1053158`

[14]`https://news.ycombinator.com/item?id=25420290`

grams in the benchmarking dataset. Function definition code demands more intensive operations during both the type-checking and code-generation phases, thus benefiting more from AST layout optimisations.

Additionally, examining the AST memory size improvements (see Figure 9), `gzip.tea` showed only a 6x reduction in AST memory footprint when comparing no optimisations to the SoA approach. In contrast, `zlib.tea` demonstrated a reduction of over 12x. Furthermore, the cache miss rate reduction for `gzip.tea` was only 1.9x, whereas `zlib.tea` experienced a 3.2x reduction. These results suggest that the effectiveness of the SoA implementation varied significantly between the two programs, with `zlib.tea` benefiting more substantially from the optimisations.

5.6 Development challenges in applying optimisations

Transitioning from a traditional OO model to a DOD-based SoA model required extensive refactoring of the AST-utilising code. The implementation of the SoA optimisation required a modification of 5.6k lines of code[15], whereas the implementations for the other optimisations only required several hundreds of lines of code to be modified. Additionally, the transition to SoA introduced additional complexity in terms of code maintenance and debugging. The complexity is largely due to the need for precise management of memory and access patterns to maintain the performance benefits. Despite the development effort, the resulting speed improvements were very substantial.

5.7 Comparison to existing research

The findings align with similar improvements observed in the Zig compiler, where transitioning to a DOD layout significantly enhanced performance by over 30% in both type-checking and code generation phases. Our findings, showing 5.6x and 2.6x speedups in the type-checking and code-generation phases, respectively, suggest even greater potential for performance improvements. However, it is essential to note that the original Tea compiler was in an unoptimised non-production-grade state, and efficient SoA implementation required additional researched optimisations to be applied. These reasons may have inflated the speedup results.

6 Conclusions and Future Work

We investigated the impact of memory layout optimisations on Abstract Syntax Trees (ASTs) and their influence on the performance of the type-checking and code-generation phases in the compilation process. Our primary research question focused on how applying Data-Oriented Design (DOD) principles to ASTs affects the speed of these critical compilation phases.

6.1 Key findings

- We conclusively demonstrated that restructuring ASTs using DOD principles can significantly enhance performance in both type-checking and code-generation phases of a compiler. Implementing a Struct-of-Arrays (SoA) model showed the most promising performance improvements, but the transition process from an Object-Oriented AST requires significant code rewriting.
- Performance gains from these optimisations varied across different platforms and architectures, yet no instances indicated a significant decline in performance. Overall, a consistent upwards trend was shown across all tested machines.

6.2 Contributions

The findings provide valuable insights and practical guidance for compiler architects aiming to construct or optimise compiler implementations for improved performance.

6.3 Future work

While we have laid the groundwork for AST memory layout optimisations, several opportunities for future research remain open:

- Exploring advanced layout strategies such as hierarchical Struct-of-Arrays or hybrid layout models to potentially achieve further performance improvements.
- Researching dynamic optimisation techniques that adapt AST layouts based on input or runtime profiling, thereby potentially optimising performance dynamically. This avenue is particularly interesting for JIT compilers.
- Extending the study to include other programming languages with actual production-grade compilers to gain comprehensive insights into memory layout optimisation across different domains and language classes.
- Validating theoretical benefits by implementing AST layout optimisation in real-world, widely used compilers, thereby identifying challenges and verifying performance gains in practical scenarios.

6.4 Recommendations

Based on our findings, the following recommendations are proposed for compiler architects:

[15] Obtained from running '`git diff --stat main Compiler`' in `https://github.com/iannisdezwart/tea/tree/struct-of-arrays`

- Adoption of an SoA layout for ASTs is strongly recommended to maximise cache utilisation and minimise memory access times, resulting in substantial performance improvements in both type-checking and code-generation phases.
- Implementing compact data structures in compilers using DOD principles can yield moderate performance gains by increasing information density and reducing memory overhead.
- Utilising memory pools for dynamically allocated objects can further enhance performance by minimising memory allocation overhead.
- Transitioning to DOD-based ASTs requires meticulous planning and significant refactoring efforts. Compiler architects should carefully evaluate the feasibility and potential performance benefits before starting such transitions, ensuring performance gains justify the development costs.

In conclusion, optimising Abstract Syntax Tree memory layouts through Data-Oriented Design principles offers promising compiler performance improvements. With this study, we provided a foundation for further research and development in this field.

7 Responsible Research

We adhere to principles of responsible conduct by addressing several key considerations.

7.1 Ethical Considerations

All software we used in the benchmark dataset and for the research is open-source. This ensures that we do not infringe on proprietary software or data. By using publicly available software and data, our findings can be independently verified and reproduced by other researchers.

7.2 Data integrity and management

Benchmark data has been carefully collected and recorded, with datasets clearly documented to ensure transparency (See Appendix B). This careful documentation validates the integrity of the data used in our research.

7.3 Reproducibility

To facilitate reproducibility, all tools and code we developed and used has been made available through public open-source repositories. The custom transpiler application we developed for the research, as well as the various AST layout optimisations and benchmarking scripts, are accessible online. The full listing of tools and code is presented in Appendix C. Although the exact machines used for benchmarks are privately owned and not publicly accessible, detailed instructions are provided to enable other researchers to replicate the experiments on their machines to verify the results.

7.4 Transparency

We provide a comprehensive description of our research methodology, including the selection of benchmark inputs, the design of alternative AST layouts, and the performance evaluation process. This ensures that other researchers can understand and scrutinise our research process.

References

[1] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools.* pearson Education, 2007.

[2] A. Kelley, "The zig programming language - 0.8.0 release notes."

[3] *Data-oriented design.* Stockport, England: Richard Fabian, Sept. 2018.

[4] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, no. 2007, p. 2007, 2007.

[5] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *International Symposium on Code Generation and Optimization (CGO'06)*, IEEE.

[6] D. Thain, "The abstract syntax tree," in *Introduction to compilers and language design*, ch. 6, Lulu.com, 2016.

[7] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 574–587, 2023.

[8] P.-H. Kamp, "Malloc (3) revisited," in *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, 1998.

[9] R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective.* Prentice Hall, 2011.

[10] D. N. Truong, F. Bodin, and A. Seznec, "Improving cache behavior of dynamically allocated data structures," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pp. 322–329, IEEE, 1998.

[11] R. M. Stallman and Z. Weinberg, "The c preprocessor," *Free Software Foundation*, vol. 16, 1987.

[12] B. Bäuchle, "std::chrono - typesafe time keeping in c++," 2014.

[13] S. Godard, "Sysstat/sysstat: Performance monitoring tools for linux."

[14] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[15] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security*, 2024.

A Original Tea compiler AST

The original Tea compiler consisted of 24 nodes, represented in the list below, in inheritance hierarchy. All nodes inherit from the base `ASTNode` object. The full source code of the AST implementation can be found here: `https://github.com/iannisdezwart/tea/tree/main/Compiler/ASTNodes`.

- `BreakStatement`
- `ClassDeclaration`
- `CodeBlock`
- `ContinueStatement`
- `ForStatement`
- `FunctionDeclaration`
- `IfStatement`
- `ReturnStatement`
- `SysCall`
- `TypeIdentifierPair`
- `TypeName`
- `VariableDeclaration`
- `WhileStatement`
- `ReadValue` (abstract)
  - `AssignmentExpression`
  - `BinaryOperation`
  - `CastExpression`
  - `FunctionCall`
  - `LiteralCharExpresssion`
  - `LiteralNumberExpresssion`
  - `LiteralStringExpresssion`
  - `WriteValue` (abstract)
    - `IdentifierExpression`
    - `MemberExpression`
    - `OffsetExpression`
    - `UnaryOperation`

B Full benchmark data listing

The full benchmark data consists of the following data points:

- **Time performance results**: 50 data points for each combination of machine, optimisation & program.
  - **Collected data points**:
    - Type-checking time (microseconds)
    - Code-generation time (microseconds)
    - Program
    - Machine
    - Optimisation
  - **Benchmark script**: `https://github.com/iannisdezwart/tea/blob/main/bench.sh`
  - **Data file**: `https://github.com/iannisdezwart/rp-data/blob/main/benchmark_data.csv`
- **Memory usage results**: 50 data points for each combination of optimisation & program.
  - **Collected data points**:
    - Optimisation
    - Program
    - VSZ delta parsing phase
    - RSS delta parsing phase
    - VSZ delta type-checking phase
    - RSS delta type-checking phase
    - VSZ delta code-generation phase
    - RSS delta code-generation phase
  - **Benchmark script**: `https://github.com/iannisdezwart/tea/blob/main/bench-mem.sh`
  - **Data file**: `https://github.com/iannisdezwart/rp-data/blob/main/memory-results.csv`

- **Cache efficiency results**: One data point per combination of optimisation & program. (Because Valgrind is relatively deterministic).
  - **Collected data points**:
    - Optimisation
    - Program
    - I Refs
    - I1 Misses
    - D Refs
    - D1 Misses
    - LLd Misses
    - LL Refs
    - LL Misses
  - **Benchmark script**: `https://github.com/iannisdezwart/tea/blob/main/bench-cache.sh`
  - **Data file**: `https://github.com/iannisdezwart/rp-data/blob/main/cache-results.csv`

C Full code listing

We used three GitHub projects in this research:

- **Tea compiler**: `https://github.com/iannisdezwart/tea`
  The Tea compiler is written in C++, and its lifetime preceded this research project by several years. We created a branch for each AST layout optimisation, so that each optimisation can be clearly viewed and compared.

- **C to Tea transpiler**: `https://github.com/iannisdezwart/c-to-tea-transpiler`
  The transpiler was written in Rust. The rationale for that decision was because the `lang_c` crate seemed to be a promising C parser with a well-documented high-level API.

- **Data repository**: `https://github.com/iannisdezwart/rp-data`
  Features a Node.JS Express API and a Postgres database running on Docker to facilitate the result processing. Uses Plotly.JS for graphing & visualisation. The final data is also stored in CSV format.