Lib Metamorphosis: A Performance Analysis Framework for Exchanging Data Structures in Performance Sensitive Applications

Michael D. Shah
Khoury College of Computer Sciences
Northeastern University
Boston, United States
mikeshah@northeastern.edu

***Abstract*—When software does not meet performance requirements, difficult decisions are made to change central data structures which may be costly financially and increase development time. In addition, monitoring how these data structures are used, and trying to understand performance implications of any change may prevent any evolution of the original infrastructure. Thus, *radical revisions* to software may be avoided due to the barriers of time and engineering complexity costs.**

Our solution to helping developers make infrastructure changes to improve performance is to provide a refactoring tool where developers may swap data structures. Our tool preserves correctness by utilizing the software's test suite and also measures performance automatically of the swapped data structure. We believe there is need for such a tool to help encourage more radical revisions and experimentation in large software projects to improve performance.

Our frameworks success will be evaluated based on preserving the correctness of the software within a developer created test suite while providing performance information based on modified data structures.

***Index Terms*—Performance, Instrumentation, Data Structures, Algorithms**

I. Introduction

Most popular compilers (e.g. clang, gnu gcc) are distributed with standard libraries for each respective language they support. These foundational data structures and algorithms equip programmers with a quick way to begin building software. The standard libraries provided (e.g. libc++) are optimized in a way that performance is good enough for a selection of general purpose use cases, and great lengths are often taken to build these libraries. However, a programmer *choosing* a data structure may not fully understand the trade offs when using a data structure other than "it worked correctly" for a given task. Data structures that have been chosen early in the development of a project, are now embedded into the core of the software, and programmers may be very hesitant to refactor them. As software evolves, the original data structures chosen may need to change or become more specialized to meet runtime performance requirements.

Developers may commit a great deal of time by building data structures or otherwise searching for a free or paid implementation that they speculate will improve performance. As an example, a custom implementation of a `string` may need to be created if profiling the standard libraries shows performance problems as the data structure scales. Once a developer finds another implementation, they must manually swap the implementation of a `std::string` (e.g. in C++) with their custom data implementation. This manual 'swap' even for a simple data structure has a cost in developer time, and developers would want to preemptively know if this exchange of data structure indeed results in an increase in performance. Kim et al. surveyed challenges of refactoring at Microsoft in a large survey, reporting developer hesitation to refactor given issues of: What if new bugs occur, what if corner cases are missed, or how long will the refactoring take [6]?

This paper presents our preliminary approach and design for a tool that swaps data structures automatically in large projects. The impact of this tool is to encourage development of more aggressive refactoring tools, with an emphasis on measuring the performance gain of a specific refactoring.

II. Related Work

The difficulty in choosing data structure collections has previously been investigated by Shacham et al. in a tool called Chameleon [10]. Chameleon attempts to assist developers in languages like C# and Java which are distributed with large prebuilt collections of data structures. This shows that choosing a correct data structure becomes a problem for programmers especially when considering performance.

De Wael et al have proposed just-in-time data structures which allow programmers to create a swap rule for when to use a data structure during run-time [2]. The argument presented is that finding the 'right' data structure is not enough, but a developer must find the right sequence of data representations for a variety of use cases. The swap rules specified allow data structures to change their representation (e.g. utilizing a sparse matrix when data is sparse). We believe this work is closer to our proposed approach. While monitoring which data structure to use has some cost, domains like graphics and games have more fixed resources, providing developers potential cues on the data representation at compile-time.

```c
typedef struct queue{
    int* data;
    int front,back, capacity,size;
}queue_t;

void enqueue(queue_t* q, int data){
    q->data[q->front] = data;
    q->front++; // Additional queue code ...
}
```

Listing 1. "A portion of a simple queue data structure written in C"

```
; Data structure implementation
%struct.queue = type { i32*, i32, i32, i32, i32 }

; Function definition with type information
define void @enqueue(%struct.queue* %q, i32 %data) #0 {
    ; ...
    ; Body omitted
    ; ...
}
```

Listing 2. "The generated LLVM Intermediate Representation of struct queue"

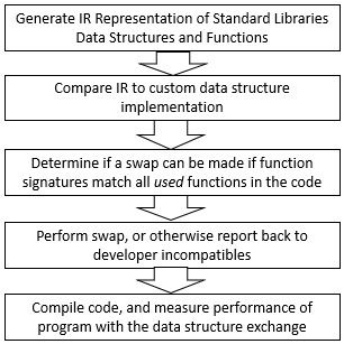

Fig. 1. The pipeline for exchanging data structures using LLVM's IR

Demsky and Rinard present a system that can enforce data structure consistency as well as automatically repair any violations regarding how data structures execute [3]. Our proposed work does not suggest repairing errors as a contribution, but does emphasize that the same unit tests should pass when any new data structure is swapped.

III. Our Tool and Experimental Methodology

The fundamental question we are trying to solve is: *"Can we automatically exchange data structures in large projects and improve runtime performance?"* Our approach, preliminary design and experimental methodology follows.

A. Basic Approach - LLVM Compiler Infrastructure

The preliminary tool we are are building uses LLVM, a compiler framework for building compilers and supporting tooling infrastructure [7]. Code that is compiled using the Clang frontend (for C-family languages like C and C++) is transformed into an intermediate representation (IR) which can be manipulated. Code listing 1 is an example queue data structure in the C-language and code listing 2 is the IR of the same data structure generated from LLVM. The struct and the associated enqueue function roughly preserves the structure and fully preserves the type information of the C queue code. At the IR level we can then exchange data structures as illustrated in Figure 1. Because we are modifying only the intermediate representation, none of the original source code changes, and we may add instrumention at the IR level to measure performance.

B. Preliminary Design

LLVM instrumentation provides an alternative technique to interpositioning. Interpositioning also allows the replacement of functions at compile-time, link-time, or run-time. However, we have two potential advantages.

1) We can embed performance measurement without using wrapper functions which *may* introduce additional overhead. This includes at different granularities such as the function, basic block, or instruction level.
2) We can statically analyze the IR for metrics like allocations, branches, or call graph size, understanding data structure swap potential.

C. Experimental Methodology

In order to measure the effectiveness of our system we plan on using several large open source projects that are performance sensitive including: Blender3D, Quake 2 RTX, and Pixar OpenSubdiv [5], [8], [9]. Each benchmark will have a lightweight instrumentation layer, injecting timers to the start and end of any data structure functions that are exchanged.

IV. Discussion and Open Questions

We believe our system will work for addressing issues of performance while still maintaining correctness. The measurement of correctness relies on projects having quality and correct unit tests. Thus correctness is captured by Dijkstra's quote *program testing can be used very effectively to show the presence of bugs but never to show their absence* [4]. A data structure in our system thus cannot be swapped unless it at the least passes the same unit tests for a program.

A second potential flaw with our system is if the software is engineered with an explicit focus on resilience to errors as opposed run-time performance. However, we speculate a user may use our infrastructure to intentionally inject buggy data structures which is best explored in other works such as by Coman [1]. Our worked first focuses on applications in the graphics and games domain where performance is the key evaluation metric.

V. Conclusion

In this paper, we have presented our preliminary design for a LLVM-based system to swap out data structures and measure performance of the change. Our tool enforces correctness based on developer written unit tests, and we hope will encourage more aggressive revisions of software to improve performance.

REFERENCES

[1] I. D. Coman, A. Sillitti, and G. Succi. An initial characterization of bug-injecting development sessions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 129–130, New York, NY, USA, 2018. ACM.

[2] M. De Wael. Just-in-time data structures: Towards declarative swap rules. In *Proceedings of the 13th International Workshop on Dynamic Analysis*, WODA 2015, pages 33–34, New York, NY, USA, 2015. ACM.

[3] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 78–95, New York, NY, USA, 2003. ACM.

[4] E. Dijkstra. On the reliability of programs. Jun 2005.

[5] B. Foundation. Blender foundation - blender.org. https://www.blender.org/, 2019.

[6] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM.

[7] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] NVIDIA. Nvidias implementation of rtx ray-tracing in quake ii. https://github.com/NVIDIA/Q2RTX, 2019.

[9] PIXAR. Opensubdiv - pixar graphics technologies. http://graphics.pixar.com/opensubdiv/docs/intro.html, 2019.

[10] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.