Cross-Language Component Testing: Performance and Interoperability Insights

Kai Erik Niermann

June 28, 2024

Outline

1. Introduction
   - Motivation
2. Research Questions
3. Study design
   - System Design
   - Approach
4. Methodology
   - rewriting
   - choice of apis
   - testing setup
   - mapping
   - improved static mapping
5. Discussion
6. Results

Introduction

Reasons for rewrites

- Second system effect
- New developments in libraries/languages
- Initial choices restrictive as application scales

Introduction

Reasons for rewrites

- Second system effect
- New developments in libraries/languages
- Initial choices restrictive as application scales

Benefits and drawbacks

- Overimplementation risk (Fred Brooks)
- Better understanding and new technologies

Introduction

Reasons for rewrites

- Second system effect
- New developments in libraries/languages
- Initial choices restrictive as application scales

Benefits and drawbacks

- Overimplementation risk (Fred Brooks)
- Better understanding and new technologies

Rewrites in software development

- Key component
- Thesis explores effective approaches

Motivation

Definition

- Using a different language for rewriting existing code

Motivation

Definition

- Using a different language for rewriting existing code

Benefits

- Addresses fundamental limitations of the original language

Motivation

Definition

- Using a different language for rewriting existing code

Benefits

- Addresses fundamental limitations of the original language

Challenges

- Effectiveness of new language
- Large system size
- Unforeseen incompatibilities

Motivation

Definition

- Using a different language for rewriting existing code

Benefits

- Addresses fundamental limitations of the original language

Challenges

- Effectiveness of new language
- Large system size
- Unforeseen incompatibilities

Testing Approach

- Test a critical component first
- Minimal additional code

Motivation

Definition

- Using a different language for rewriting existing code

Benefits

- Addresses fundamental limitations of the original language

Challenges

- Effectiveness of new language
- Large system size
- Unforeseen incompatibilities

Testing Approach

- Test a critical component first
- Minimal additional code

Strategies

- Wrap component in testing framework
- Use language interop APIs

Motivation

Definition

- Using a different language for rewriting existing code

Benefits

- Addresses fundamental limitations of the original language

Challenges

- Effectiveness of new language
- Large system size
- Unforeseen incompatibilities

Testing Approach

- Test a critical component first
- Minimal additional code

Strategies

- Wrap component in testing framework
- Use language interop APIs

Objective

- Determine if the component works better in another language

Research Questions

Main Question

How can we effectively test the performance of a system component rewritten in another programming language?

Research Questions

Main Question

How can we effectively test the performance of a system component rewritten in another programming language?

Ease of use

How do language APIs differ in their approaches to mapping objects and types, are some harder to work with than others?

Research Questions

Main Question

How can we effectively test the performance of a system component rewritten in another programming language?

Ease of use

How do language APIs differ in their approaches to mapping objects and types, are some harder to work with than others?

Performance

How do performance attributes effect the application of interop APIs in performance dependent code?

System design

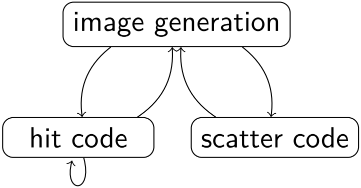


Figure: Ray tracer overview

System design

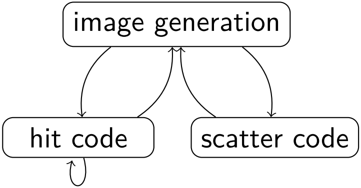


Figure: Ray tracer overview

- Complex enough system to demonstrate the idea in an applied setting.

System design

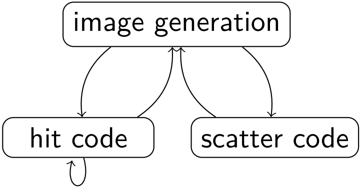


Figure: Ray tracer overview

- Complex enough system to demonstrate the idea in an applied setting.
- Distinct components that can be isolated for testing.

System design

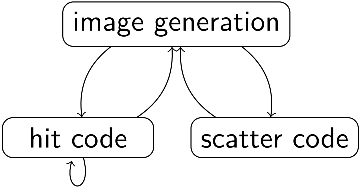


Figure: Ray tracer overview

- Complex enough system to demonstrate the idea in an applied setting.
- Distinct components that can be isolated for testing.
- Requires high performance to function effectively, making it ideal for assessing performance needs.

Study Approach: Q&A Format

How do we isolate the most performance impacting component?

- Profile our application for runtime and memory.
- Use the profiling data to make a judgement on the most performance impacting component.

Study Approach: Q&A Format

How do we isolate the most performance impacting component?

- Profile our application for runtime and memory.
- Use the profiling data to make a judgement on the most performance impacting component.

What languages do we choose for rewriting?

- Choose languages with embedding APIs for Julia.
- Opt for Python and C++ due to their differing fundamental properties and available interface libraries.

Study Approach: Q&A Format

How do we isolate the most performance impacting component?

- Profile our application for runtime and memory.
- Use the profiling data to make a judgement on the most performance impacting component.

What languages do we choose for rewriting?

- Choose languages with embedding APIs for Julia.
- Opt for Python and C++ due to their differing fundamental properties and available interface libraries.

How do we test the rewritten components?

- Use the language specific benchmarking tools to test components in isolation.
- Use Julia's benchmarking tools to test compontents & overhead

Rewriting

- Most performance impact: hit functions, crucial for ray tracing efficiency.

Rewriting

- Most performance impact: hit functions, crucial for ray tracing efficiency.
- Efficiency concern: If target languages perform poorly with hit functions, rewriting the entire ray tracer may not be feasible.

Rewriting

- Most performance impact: hit functions, crucial for ray tracing efficiency.
- Efficiency concern: If target languages perform poorly with hit functions, rewriting the entire ray tracer may not be feasible.
- Rewriting components: Trivial task due to similar syntax across languages.

C++

```cpp
bool hit(const aabb& box, const ray& r, const interval& ray_t);
```

Python

```python
def hit(bbox: aabb, r: ray, ray_t: interval[float]) -> bool:
```

Julia

```julia
function hit!(bbox::aabb, r::ray, ray_t::interval)::Bool
```

Choosing an Interop API

Choosing the right embedding API to attach hit functions to the ray tracer implementation was crucial.

Choosing an Interop API

Choosing the right embedding API to attach hit functions to the ray tracer implementation was crucial.

Python

- Python/JuliaCall API:
    - Builds on previous implementations.
    - Supports extensive language conversions.

Choosing an Interop API

Choosing the right embedding API to attach hit functions to the ray tracer implementation was crucial.

Python

- Python/JuliaCall API:
  - Builds on previous implementations.
  - Supports extensive language conversions.

C++

- Options:
  - CxxWrap: Feature-rich but less user-friendly.
  - (**choice**) Jluna: Newer, user-friendly, lacks some features

Choosing an Interop API

Choosing the right embedding API to attach hit functions to the ray tracer implementation was crucial.

Python

- Python/JuliaCall API:
  - Builds on previous implementations.
  - Supports extensive language conversions.

C++

- Options:
  - CxxWrap: Feature-rich but less user-friendly.
  - (**choice**) Jluna: Newer, user-friendly, lacks some features

Considerations

1. Choose an API that minimizes additional code.
2. Prefer an API that simplifies component attachment.

Choosing an Interop API

Choosing the right embedding API to attach hit functions to the ray tracer implementation was crucial.

Python

- Python/JuliaCall API:
  - Builds on previous implementations.
  - Supports extensive language conversions.

C++

- Options:
  - CxxWrap: Feature-rich but less user-friendly.
  - (**choice**) Jluna: Newer, user-friendly, lacks some features

Considerations

1. Choose an API that minimizes additional code.
2. Prefer an API that simplifies component attachment.

Languages supporting reflection reduce additional code, simplifying rewriting and testing.

Testing setup

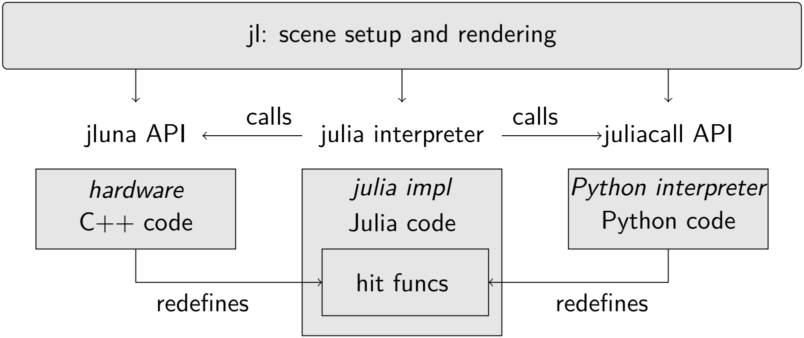


Figure: Testing setup for component isolation

Mapping Objects to Statically Typed Languages

Static Mapping

- User describes interface for types (e.g., class/struct).
- API maps Julia type to corresponding C++ type.
- User-defined reflection system parses low-level types.

Mapping Objects to Statically Typed Languages

Static Mapping

- User describes interface for types (e.g., class/struct).
- API maps Julia type to corresponding C++ type.
- User-defined reflection system parses low-level types.

Getters and Setters:

- Define functions get and set.
- get : $S \rightarrow V =$ object property x
- set : $S, V \rightarrow S =$ modified property x

Mapping Objects to Statically Typed Languages

Static Mapping

- User describes interface for types (e.g., class/struct).
- API maps Julia type to corresponding C++ type.
- User-defined reflection system parses low-level types.

Getters and Setters:

- Define functions get and set.
- get : $S \to V =$ object property x
- set : $S, V \to S =$ modified property x

Note: Similar to functional programming lens.

Challenges with Getter/Setter Pairs

Limitation

Problems occur when the setter has an abstract type.

Problem:

- Julia uses strings to represent types.
- Need to convert these strings back to C++ types.

Solution:

- Create a mechanism to map strings to types and compare them to find the right derived type

Optimizing Setter Functions

Key issue with trivial approach

- Requires providing all derived types to each getter/setter pair.
- Leads to annoying repeated user defined code and potential errors.

Optimizing Setter Functions

Key issue with trivial approach

- Requires providing all derived types to each getter/setter pair.
- Leads to annoying repeated user defined code and potential errors.

Preferable solution

- Define the set of derived types once.
- Allow all setters for attributes of a class to access this information.

Template Metafunctions for Object Properties

Template metafunctions are used to define object properties:

```
template <typename Ot, typename Ft, const char* name>
struct Property {
    static constexpr const char* get_name() { return name; }
    static std::function<Ft(Ot&)> getter;
    static std::function<void(Ot&, Ft)> setter;
};
```

Template Metafunctions for Object Properties

Template metafunctions are used to define object properties:

```
template <typename Ot, typename Ft, const char* name>
struct Property {
    static constexpr const char* get_name() { return name; }
    static std::function<Ft(Ot&)> getter;
    static std::function<void(Ot&, Ft)> setter;
};
```

Example property declaration:

```
Usertype<bvh_node>::initialize_type(
    tl<
        Property<bvh_node, Hittable*, #left>,
        Property<bvh_node, Hittable*, #right>,
        Property<bvh_node, aabb, #bbox>
    >(),
    tl<Triangle, Sphere, bvh_node>()
);
```

Limitations

Considerations

- Users must specify class properties via lambdas.
  - Macros can simplify this process but its not ideal.

Limitations

Considerations

- Users must specify class properties via lambdas.
  - Macros can simplify this process but its not ideal.
- All derived types must be passed to objects with base type attributes.
  - Tradeoff in library complexity and maintainability vs user defined types.

Limitations

Considerations

- Users must specify class properties via lambdas.
  - Macros can simplify this process but its not ideal.
- All derived types must be passed to objects with base type attributes.
  - Tradeoff in library complexity and maintainability vs user defined types.
- Still requires additional user defined type specifications to allow for reflection in the context of interop

Limitations

Considerations

- Users must specify class properties via lambdas.
    - Macros can simplify this process but its not ideal.
- All derived types must be passed to objects with base type attributes.
    - Tradeoff in library complexity and maintainability vs user defined types.
- Still requires additional user defined type specifications to allow for reflection in the context of interop

key challenge

Find some means of having the base type interface discern the correct derived type to instantiate.

Benefits

This system offers

- Simplified object deduction process
- Ability to map abstract objects
- Utilization of modern C++ principles for futureproofing
- Increased maintainability and extensibility
- Possiblity to adapt this when Reflection TS becomes available in C++26

Benefits

This system offers

- Simplified object deduction process
- Ability to map abstract objects
- Utilization of modern C++ principles for futureproofing
- Increased maintainability and extensibility
- Possiblity to adapt this when Reflection TS becomes available in C++26

Key Takeaway

It provides a more concise approach compared to traditional methods, enhancing ease of implementation and potential for future extensions.

Insights - RQ2

Observations

- Possible to implement a generic method for polymorphic object mapping.
- Offers JuliaCall API capabilities

Insights - RQ2

Observations

- Possible to implement a generic method for polymorphic object mapping.
- Offers JuliaCall API capabilities

Main takeaways

- We can map to static languages with minimal additional boilerplate
- Mostly straightforward testing and integration across languages

Baseline Performance analysis

Timing method

- Call to `bvh_hit` from Julia
- Execution time in target languages (C++ and Python)

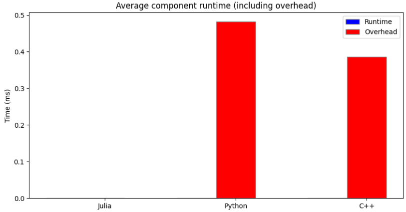
Baseline Performance analysis

Timing method

- Call to `bvh_hit` from Julia
- Execution time in target languages (C++ and Python)

Results

- Julia
  - Fastest average component execution time since no overhead
- Python and C++:
  - Similar average component execution times
  - Execution time primarily attributed to overhead



Isolated Performance - RQ3

Results

- C++ outperforms Julia and Python in component runtime.
- Substantial overhead from Julia API (dynamic dispatch, type inference).
- Julia performs well with type stability, hindered by cross-language calls.

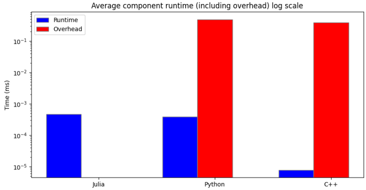
Isolated Performance - RQ3

Results

- C++ outperforms Julia and Python in component runtime.
- Substantial overhead from Julia API (dynamic dispatch, type inference).
- Julia performs well with type stability, hindered by cross-language calls.

Conclusion

- High-performance interop between different languages is challenging.
- Effective on a smaller scale, but less viable for very different languages.



Results and Takeaways

- Different language APIs have varied integration approaches.
- Adapting APIs to specific use cases is generally manageable depending on the language
- Metaprogramming is effective for implementing generic, extensible interop libraries.
- Performance overhead can depends alot on language compatability and paradigms
- APIs work well for benchmarking and testing components in isolation
- Careful consideration needed for using these APIs in performance-dependent production code.