Cross-Language Component Testing: Performance and Interoperability Insights

Kai Erik Niermann

June 25, 2024

Outline

Introduction

- Establishing terminology
- **Rewrite**:
  - Writing new code
  - Same or new language
  - Implements existing features
- **Refactoring**:
  - Rearrange/minimize existing code
- 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
- **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 test the performance of a system component rewritten in another programming language?

Role of Language APIs

How do language APIs differ in their approaches to mapping objects and types, are some more restrictive than others?

Performance and Real world applications

How does the overhead and restrictions on optimization techniques effect the application of interop APIs in performance dependent production code?

system design

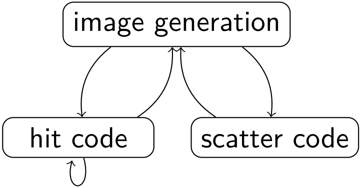


Figure: Ray tracer overview

- 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.

Testing 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.
- 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 Embedding 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

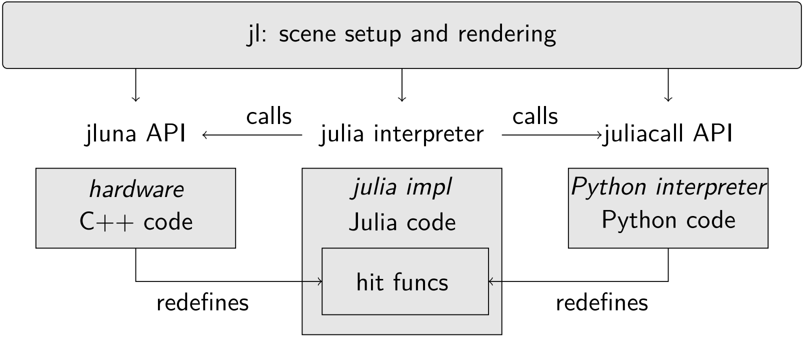


Figure: Testing setup for component isolation

dynamic mapping

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(x) and put(x).
- Property x represented as a string.
- $\text{get} : S \to V =$ object property x
- $\text{set} : S, V \to S =$ modified property x

Note: Similar to functional programming lens.

Challenges with Getter/Setter Pairs

Limitation

Issues arise when source S has an abstract type.

Notation

- $[S : T]$, $[V : T]$ represent source and value types.
- set : $[S : T_B], [V : T_{bi}] \rightarrow S_m$

Problem: Julia uses strings for type representation:

- set : $[S : T_B], [V : \text{str}(T_{bi})] \rightarrow S_m$
- Need to map strings to C++ types.

Solution: Create a mapping mechanism

- string to type mapping *st* and type comparison *f*:

$$st : \text{str}(T_i) \rightarrow \text{typeinf}(T_i) \quad f : st, \text{typeinf}(T_p) \rightarrow \text{bool}$$

- Use compile-time code generation with variadic templates.

Achieve correspondence using fold expressions on variadic templates.

Optimizing Setter Functions

An issue with the current approach:

- Requires providing all derived types to each getter/setter pair.
- Leads to significant overhead and potential errors.

Preferable solution:

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

Improved Design with Varadic Templates

We can improve upon this design by utilizing varadic templates for types, getters, and setters to create a compile-time mapping:

```
def initalize_type[T, attr..., derived...]() -> void:
    fold => (attr,
        mapping.insert(attr::name,
            (
                // getter
                [instance: T]() { instance[T](attr::get(instance)) }
                // setter
                [instance: T, jl_val: jlval, jl_type_string: str] {
                    fold(derived,
                        if (f(st(jl_type_string), typeinfo(derived))):
                            attr::set(instance, instance[derived](jl_val))
                            matched = true
                        )
                    if (!matched):
                        attr::set(instance, instance[T](jl_val))
                }
            )
        )
    )
```

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.
- 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 introspection

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

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 with C++ speed.
- Requires Julia data copying due to Julia C API limitations.

Main takeaways

- Metaprogramming approaches effective for object reflection
- We can map to static languages with minimal additional boilerplate
- Mostly straightforward testing and integration across languages

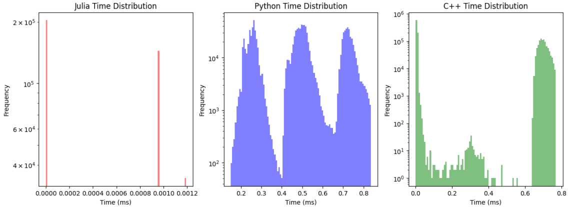
Baseline Performance analysis

Timing method

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

Results

- Julia:
  - Many calls near 0ms (no hit)
  - Some calls around 0.0010ms (ray intersection and computation)
- Python and C++:
  - Similar call time distribution (measured from Julia)
  - Overhead will be clearer in overhead-exclusive graphs



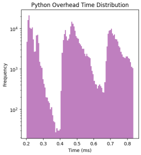
Isolated Performance - RQ3

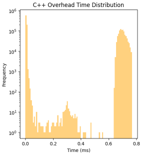
Results

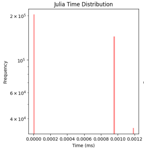
- C++ outperforms Julia and Python in component runtime.
- Substantial overhead from Julia API (dynamic dispatch, type inference).
- Julia performs well with type stability, hindered by dynamic calls.

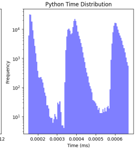
Conclusion

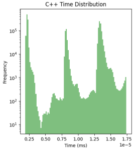
- High-performance interop between different languages is challenging.
- Effective on a smaller scale, but less viable for very different languages.











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 mainly due to language incompatibilities.
- Mitigation possible with languages using similar execution methods.
- APIs work well for testing components.
- Careful consideration needed for using these APIs in performance-dependent production code.