Catwalk.jl: An adaptive dispatch optimizer

Krisztián Schäffer[1]

[1]independent researcher

ABSTRACT

`Catwalk.jl` is a JIT compiler implemented as a Julia library that generates optimized dispatch code based on statistical profiling. Unlike typical JIT compilers it requires some integration work from its users, allowing it to completely eliminate the need of complex deoptimization logic. It is able to compile new type-stabilized routes or reorder existing ones if the distribution of dispatched types changes during runtime and the customizable cost model predicts significant speedup compared to the best version that was previously compiled.
`Catwalk.jl` was designed for situations when both composability and run-time polymorphism is required, and some run-time compilation overhead is acceptable for speeding up dynamic dispatch in hot loops.
Proceedings of JuliaCon

Keywords

Julia, JIT, Dynamic Dispatch, Adaptive Optimization, Profiler

1. Introduction

Although in most cases we can eliminate run-time dispatch in Julia by using generics or staging, when not, it incurs significant overhead compared to single dispatched virtual method calls of other languages. Thus, polymorphic behavior in hot loops is problematic in Julia [4].
A variety of techniques and packages is known to alleviate this issue, but they are either restricted to collections [5] [3] or affect composability: Types injected from dependent packages cannot benefit from the speedup [1] [2].
`Catwalk.jl` uses a technique I call "iterated staging" to include all encountered types in the optimization, while interacting with the Julia compiler using only standard metaprogramming facilities: macros, `@generated` functions and generics.

2. Integration: Iterated staging

The user has to shape their hot loop to work in batches if they did not do it already e.g. for reporting. Batch size is typically between 100 and 1_000_000 iterations.
The user has to manually drive the optimization process, like:

```
optimizer = Catwalk.JIT()
for batch_num in 1:BATCH_COUNT
    Catwalk.step!(optimizer)
    single_batch(batch_num, Catwalk.ctx(optimizer))
end
```

`single_batch` or its downstream function that contains the dynamically dispatched call site must also be marked with the `@jit` macro that turns it into a generated function.

3. Compilation

This structure allows `Catwalk.jl` to recompile between batches if needed. The *type* of the "JIT context" returned by the `Catwalk.ctx()` call determines the code to be generated. The call `f(a, unstable, b)` will be rewritten to:

```
if unstable isa FIXTYPE1
    f(a, unstable, b) # Type-stabilized route
elseif unstable isa FIXTYPE2
    f(a, unstable, b) # Type-stabilized route
...
else
    f(a, unstable, b) # Fallback to dynamic dispatch
end
```

4. Optimization

Randomly selected batches get instrumented with profiling code, collecting the frequencies of dispatched types.
After profiled batches the optimizer generates the list of most frequent types, and estimates the cost of dispatch for the collected profile not only for this ideal list, but also for every previously compiled one. Then it finds the best previous compilation, and either activates it or decides to compile code from the ideal list if the cost simulation suggests significant speedup.
Every aspect of the optimization process is configurable and extendable, including the maximal number of stabilized types, the cost model, the profiler and the optimizer.

5. Conclusion and further work

Thanks to the "Just Ahead Of Time" compilation model of Julia, `Catwalk.jl` can adaptively generate highly optimal dispatch code, while working completely in "user space".
Compilation cost is effectively minimized by manual impact point selection and backtesting peviously compiled versions against newly collected profiles using a cost model of dispatch.
`Catwalk.jl` speeds up dynamic dispatch by a factor typically between 5 and 200. Total experienced speedup exceeds 30% in some real-life programs.
In the future I plan to improve compilation overhead by using more efficient type-encoding; automatically determine the maximal length of the fixtype list by calculating marginal returns; and experimenting with decision trees based on type hashes.

6. References

[1] Tim Holy. Union-splitting: what it is, and why you should care. *The Julia Language Blog*, 2018.

[2] jlapeyre@github John Lapeyre. Singledispatcharrays.jl. *Github*, 2020.

[3] melonedo@github. Singledispatcharrays.jl. *Github*, 2021.

[4] Ronneesley Moura Teles. Performance drawback with subtyping. *discourse.julialang.org*, 2020.

[5] tkoolen@github Twan Koolen. Typesortedcollections.jl. *Github*, 2017.