Contextual Dispatch for Function Specialization

OLIVIER FLÜCKIGER, Northeastern University
GUIDO CHARI, ASAPP INC
MING-HO YEE, Northeastern University
JAN JEČMEN, Czech Technical University
JAKOB HAIN, Northeastern University
JAN VITEK, Northeastern & Czech Technical University

In order to generate efficient code, dynamic language compilers often need information, such as dynamic types, not readily available in the program source. Leveraging a mixture of static and dynamic information, these compilers speculate on the missing information. Within one compilation unit, they specialize the generated code to the previously observed behaviors, betting that past is prologue. When speculation fails, the execution must jump back to unoptimized code. In this paper, we propose an approach to further the specialization, by disentangling classes of behaviors into separate optimization units. With contextual dispatch, functions are versioned and each version is compiled under different assumptions. When a function is invoked, the implementation dispatches to a version optimized under assumptions matching the dynamic context of the call. As a proof-of-concept, we describe a compiler for the R language which uses this approach. We evaluate contextual dispatch on a set of benchmarks and compare it to traditional speculation with deoptimization techniques. Our implementation is, on average, 1.7× faster than the GNU R reference implementation, and contextual dispatch improves the performance of 18 out of 46 programs in our benchmark suite.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: virtual machine, optimizing compiler, specialization, splitting, speculation

ACM Reference Format:
Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article xxx (November 2020), 23 pages. https://doi.org/10.1145/xxx

1 INTRODUCTION

Just-in-time compilers are omnipresent in todays technology stacks and the performance of the code they generate is central to the growing adoption of dynamic languages. That performance is increasingly dependent on sophisticated on-line optimizations that specialize programs according to observed behaviors, identify likely invariants, such as the types of the arguments of a given function, and generate code that leverages those invariants.

In our experience, to achieve performance for dynamic languages, a compiler needs information about the calling context of a function. This can be information about the type and shape of the function's arguments, the potential side-effects of called functions, or other predicates about program state that hold when the function is invoked. We have observed that classical compiler

Authors' addresses: Olivier Flückiger, Northeastern University; Guido Chari, ASAPP INC; Ming-Ho Yee, Northeastern University; Jan Ječmen, Czech Technical University; Jakob Hain, Northeastern University; Jan Vitek, Northeastern & Czech Technical University.

2475-1421/2020/11-ARTxxx
https://doi.org/10.1145/xxx

optimizations such as speculation and inlining work well together to expose that contextual information to the optimizer. Inlining allows to optimize the body of a function together with its arguments and speculation is needed to enable inlining. The drawbacks of this approach are that inlining grows the size of compilation units, and speculation may fail causing the entire compilation unit to be discarded and execution to proceed in unoptimized code.

In this paper, we explore an approach to structure a just-in-time compiler to better leverage information available at run time. Our starting point is a compiler for a dynamic language geared to perform run-time specialization: it compiles functions under assumptions, guesses about potential invariants, and deoptimizes those functions if any of their assumptions fails to hold. In addition, our baseline also performs optimizations such as dead code elimination, loop unrolling, and function inlining. Low-level code transformations are outsourced to a highly optimizing back-end compiler. Our goal is to extend this baseline compiler with a new technique that provides contextual information by specializing functions for different calling contexts. For every call, the best version given the current state of the system is invoked.

The inspiration for our work comes from customized compilation, pioneered by Chambers and Ungar [1989], an optimization that systematically specializes functions to the dynamic type of their arguments. We extend this approach by specializing functions to arbitrary contexts and dynamically selecting optimized versions of a specialized function depending on the run-time state of the program. We refer to the proposed approach as *contextual dispatch*. As such, we define a *context* to be a predicate on program state chosen such that there exists an efficiently computable partial order between contexts and a distinguished maximal element. A *version* of a function is an instance of that function compiled under the assumption that a given context holds at entry. To leverage versions, for function calls the compiler emits a *dispatch* sequence that computes the call site context and invokes a version of the target function that most closely matches the calling context. The unoptimized version of the function is associated to the maximal context and is the default version that will be called when no other applies.

As an illustration, consider Listing 1 written in R. The semantics of R is complex: functions can be invoked with optionally named arguments that can be reordered and omitted. Furthermore, arguments are lazy and their evaluation (when the value is needed) can modify any value, including function definitions. In the above example, the `max` function is expected to return the largest of its first two parameters, mindful of the presence of missing values (denoted `NA` in R). The third, optional, parameter is used to decide whether to print a warning in case a missing value is observed. If `max` is passed a single argument, it behaves as the identity function. Since R is a vectorized language, the arguments of `max` can be vectors of any of the base numeric types of the language. Consequently, compiling this function for all possible calling contexts is likely to yield inefficient code.

```
max <- function(a, b=a,
                  warning=FALSE) {
  if (warning &&
      any(is.na(c(a,b))))
    warn("NA Value")
  if (a < b) b else a
}

max(x) + max(y,0)
```

Listing 1. `max` function

Contextual dispatch is motivated by the observation that, for any execution of a program, there are only a limited, and often small, number of different calling contexts for any given function. For example, if `max(y,0)` and `max(x)` are the only calls to `max`, then we may generate two versions of that function: one optimized for a single argument and the other for two. Further specialization can happen on the type and shape of the first argument, this may either be a scalar or a vector of one of the numeric types. This shows that part of a context can be determined statically, *e.g.*, the number of arguments, but other elements are only known at run time, *e.g.*, the type and shape of arguments. Contextual dispatch thus, in general, requires run-time selection of an applicable call target.

Contributions. This paper presents contextual dispatch, a novel optimization for just-in-time compilers. To evaluate the benefits of this optimization, we provide a proof-of-concept implementation in the Ř research virtual machine [Flückiger et al. 2019]. Ř supports speculation and inlining; we add contextual dispatch to improve performance by enabling specialization to multiple contexts. We quantify the benefits of our approach with two experiments. First, we establish baseline performance of Ř with contextual dispatch by comparing it to the GNU R bytecode interpreter [Tierney 2019] and the FastR just-in-time compiler [Stadler et al. 2016]. GNU R is the reference implementation of the R language with limited optimization opportunities, while FastR is built leveraging a high-performance optimizing compiler running on top of a JVM. This first experiment shows that Ř has competitive performance. It is faster than GNU R (0.7–46×, average 1.7×) and slower than FastR (0.1–6×, average 0.58×). Unlike FastR, Ř's performance is never significantly worse than GNU R. The second experiment focuses on understanding the impact of contextual dispatch on performance. For this experiment we compare against Ř without contextual dispatch, but still performing inlining and speculative optimizations. The results of this experiment show that contextual dispatch can deliver improvements of up to 25% with negligible regressions.

We consider R an interesting host to study compiler optimizations because of the challenges it presents to language implementers. However, contextual dispatch is not specific to R. We believe that the approach carries over to other dynamic languages such as JavaScript or Python. Moreover, we emphasize that contextual dispatch is not a replacement for other optimization techniques; instead, it is synergistic.

Availability. Our work was done as an extension to an open-source virtual machine, available at ř-vm.net. Source code along with experimental data and containers to reproduce our results are publicly available [Flückiger et al. 2020].

2 BACKGROUND

In dynamic languages such as JavaScript, Python, or R, the source code often lacks the information a compiler needs to generate efficient code. This is due to language features such as polymorphism, reflective capabilities, and late binding, among others. Just-in-time compilers have a crucial degree of freedom: they can gather information about the program in profiling mode and generate code specialized to the program's actual behavior, rather than code that handles semantically possible situations. As new behaviors are encountered, the compiler adapts the generated code to handle them as well.

2.1 Related Work

Inlining. This powerful optimization has been used in static languages for over forty years [Scheifler 1977]. Replacing a call to a function with its body has several benefits: it exposes the calling context thus allowing the compiler to optimize the inlined function, it enables optimizations in the caller, and it removes the function call overhead. The limitations of inlining are related to code growth: compilation time increases and cache locality may be negatively impacted. In dynamic languages, function calls are usually expensive, so inlining is particularly beneficial.

Speculation. Most modern just-in-time compilers rely on speculative compilation to generate code for a subset of the possible behaviors of a function. For instance, Java compilers speculatively devirtualize methods that are not overridden [Paleczny et al. 2001]. Speculative compilation implies support for deoptimization when the speculation premises fail [Hölzle et al. 1992]. The technique can be applied to the level of speculating on a single execution trace [Gal et al. 2009]. The drawback of speculation is that it does not scale well with very dynamic behavior, as the speculation applies indiscriminately. For instance, all contextual specializations presented in our work already exist as speculations in Ř— contextual dispatch additionally allows us to disentangle different calling contexts, which in turn also leads to fewer behaviors and thus narrower speculations within the different versions. Another drawback of speculation is that deoptimization is costly, as the compiler needs to add and maintain safe-points which inhibit some optimizations.

Customization. Chambers and Ungar [1989] describe customized compilation as the compilation of several copies of a function, each customized for one receiver type, so that the type of the receiver is bound at compile time. Method dispatch on the receiver type ensures that the correct version is invoked. This idea of keeping several customized versions of a function is generalized in the Jalapeño VM, which specializes methods to the types and values of arguments, static fields, and object fields [Whaley 1999]. Some specialization is enabled by static analysis, some by dynamic checks. The Julia compiler specializes functions on all argument types and uses multimethod dispatch to ensure the proper version is invoked [Bezanson et al. 2018]. As customization may lead to code bloat, Dean et al. [1995] proposes to limit overspecialization by specializing to sets of types. Hosking et al. [1990] argues for customized compilation of persistent programs to specialize code based on assumptions about the residency of their arguments. **?** present dynamic specialization to parametrized types in the intermediate representation of the .NET virtual machine; similarly **?**, or using user-guidance **?** for Java. **?** specialize on arbitrary values for JavaScript functions with a singular calling context and **?** introduce a type-based specialization which combines dynamic and static information. Liu et al. [2019] proposes to specialize methods under a dynamic thread-escaping analysis to have lock-free versions of methods for thread-local receivers. Different granularities for customization have been studied; one notable design point is the basic block versioning technique of Chevalier-Boisvert and Feeley [2015], each basic block is specialized and a jump between basic blocks consists of selecting a specialized target depending on the types of local variables. For ahead-of-time compilers, **?** initially proposed to clone methods, for instance to support inter-procedural constant propagation. Many similar context-sensitive optimization approaches follow, for instance by **?**. **?** use a static technique to partition instances and calling contexts, such that more specialized methods can be compiled and then dynamically invoked. **?** present dynamic specialization to concrete arguments using dynamic binary rewriting. **?** introduce a state-machine based technique to reduce the number of clones, when applying context-sensitive optimizations in the presence of longer call strings. Overall, keeping customized copies of the same function is a well-known optimization technique. Contextual dispatch shows how to select a target customization for each call at run-time. Existing approaches perform the selection by, either, piggybacking onto existing dispatching mechanisms in the VM, by implementing an ad-hoc and fragmented approach, or by statically splitting at each call site.

Splitting. Chambers and Ungar [1989] describe the SELF compiler as predicting types that are statically unknown but likely, and inserting run-time type tests to verify predictions. Whereas customization duplicates methods, splitting duplicates call-sites. Leveraging splitting and speculation a compiler can increase inlining opportunities. For instance splitting on the receiver type and tail-duplication allow SELF to statically resolve repeated calls to the same receivers. Splitting is a common optimization in ahead-of-time compilers, for instance LLVM [**?**] has a pass to split call-sites to specialize for non-null arguments. In a dynamic language splitting can be thought of as the frozen version of a polymorphic inline cache [**?**]. Both, inline caches and splitting, are orthogonal to contextual dispatch. Ř uses (external) caches for the targets of contextual dispatch and we could use splitting to split call-sites for statically specializing to the most commonly observed contexts.

Applicability to other languages. Contextual dispatch builds on and extends techniques used in languages such as JavaScript, SELF, and Java. While the details of the implementation, in particular the choice of contexts, presented in this paper are tailored to R's calling conventions, the idea of dispatching on information available at call sites carries over broadly. In SELF and Julia, the existing dispatch mechanism was used to dispatch on receiver and argument types. For these languages, one could imagine extending the dispatch machinery. For languages without a built-in dispatching mechanism, adding contextual dispatch can be as simple as compiling a trampoline with straightforward case analysis, and using that as a dispatcher. We expect that most context-sensitive optimizations can be implemented using contextual dispatch.

2.2 Compiling R

The R language [R Core Team 2019] is an imperative language for statistical computing with vectorized operations, copy-on-write of shared data, a call-by-need evaluation strategy, multiple dispatch, first-class closures, and reflective access to the call stack. In this section, we focus on the features that are relevant to this paper. Previous work related to speeding up R includes the GNU R optimizing bytecode compiler [Tierney 2019], the Purdue FastR specializing interpreter [Kalibera et al. 2014], the Oracle FastR compiler [Stadler et al. 2016] and the Riposte tracing compiler [Talbot et al. 2012]. Of these systems, only GNU R and Oracle's FastR are maintained as of this writing.

Obstacles. The list of challenges for optimizing R is too long to detail. We restrict the presentation to seven headaches, which we address with contextual dispatch in the subsequent sections.

(1) <u>Out of order</u>: A function can be called with a named list of arguments, thus the call `add(y=1,x=2)` is valid, even if arguments `x` and `y` are out of order. *Impact:* To deal with this, GNU R reorders its linked list of arguments on every call.

(2) <u>Missing</u>: A function can be called with fewer arguments than it defines parameters. For example, if function `add(x, y)` is called with one argument, `add(1)`, it will have a trailing missing argument. While the calls `add(,2)` and `add(y=2)` have an explicitly missing argument for `x`. These calls are all valid. *Impact:* If the missing parameters have default values, those will be inserted. Otherwise, the implementation must report an error at the point a missing parameter is accessed.

(3) <u>Overflow</u>: A function can be called with more arguments than it defines parameters. *Impact:* The call sequence must include a check and report an error.

(4) <u>Promises</u>: Any argument to a function may be a thunk (promise in R jargon) that will be evaluated on first access. Promises may contain arbitrary side-effecting operations. *Impact:* A compiler must not perform optimizations that depend on program state that may be affected by promises.

(5) <u>Reflection</u>: Any expression may perform reflective operations such as accessing the local variables of any function on the call stack. *Impact:* The combination of promises and reflection requires implementations to be able to provide a first-class representation of environments.

(6) <u>Vectors</u>: The most widely used data types in R are vectorized. Scalar values are vectors of length one. *Impact:* Unless it can prove otherwise, the implementation must assume that values are boxed and operations are vectorized.

(7) <u>Objects</u>: Any value, even an integer constant, can have attributes. Attributes tag values with key-value pairs which are used, among other things, to implement object dispatch. *Impact:* The implementation must check if values have a class attribute, and, if so, dispatch operations to the methods defined to handle them.

It is noteworthy that none of the above obstacles can be definitely ruled out at compile time. Even with the help of static program analysis, these properties depend on the program state at the point a function is called. To illustrate this, consider the number of arguments passed to a function. The following code calls `add()` twice, once with a statically known number of arguments and the second time with the result of expanding the *varargs* parameter:

```
g <- function(...) add(1,3) + add(...)
```

The triple dots expand at run time to the list of arguments passed into `g`. Thus, to know the number of arguments of `add` requires knowing the number of arguments of `g`. The following are all legal invocations:

```
g(); g(1); g(,1); g(1,2,3); g(b=1, a=2); g(..., a=1);
```

We conclude with a reassuring code snippet:

```
good <- function(arg) { ugly <- 1; arg; ugly }
bad <- function(x) rm(li=x, envir=sys.frame(-1))
good(bad("ugly"))
```

The `good` defines a local variable named `ugly` and, between that variable's definition and access, evaluates `arg` which leads to a `bad` call. The `bad` reflectively deletes the `ugly`. Thus, the `good`'s final act will be to look for the `ugly`, first in its local scope, then at the top-level. This example showcases R's expressive power, and hints at implementation challenges.

Ř. Ř is a just-in-time compiler that plugs into the GNU R environment and is fully compliant with the language's semantics. It passes all GNU R regression tests as well as those of recommended packages with only minor modifications.[1] Ř follows R's native API which exposes a large part of the language run-time to user-defined code. It is also binary compatible in terms of data structure layout, even though this is costly as GNU R's implementation of environments is not efficient. The hooks required to load Ř are small enough to be easily ported to newer releases. Compliance is checked automatically at each commit.

Ř adopts a multi-tier execution strategy. Source code is translated to an intermediate representation named RIR which can be interpreted. RIR is translated to a static single assignment intermediate representation called PIR [Flückiger et al. 2019]. Optimizations such as global value numbering, dead store and load removal, hoisting, escape analysis, and inlining are all performed on PIR code. Finally, native code is generated by a LLVM-backend. It is noteworthy that many of the Ř optimizations are also provided by LLVM. However, in PIR they can be be applied at a higher level. For instance, function inlining is involved due to first-class environments. There are also R specific optimizations, such as scope resolution, which lowers local variables in first-class environments to PIR registers; promise inlining; or optimizations for eager functions.

Ř relies on speculative optimizations. Profiling information from previous runs is used to speculate on types, on shapes (scalars vs. vectors), on the absence of attributes, and so on. Ř also performs speculative dead code elimination, speculative escape analysis to avoid materializing environments, and speculative inlining of closures and builtins. Speculation is orthogonal to contextual dispatch; every version of a function can have its own additional speculative optimizations. The design of speculative optimizations is based on work by Flückiger et al. [2018]. A novel feature is that PIR allows scheduling of arbitrary instructions, such as allocations, only on deoptimization. When speculation fails, a deoptimization mechanism transfers execution back to the RIR interpreter.

[1]Two error messages were changed, and a 1 second increase to a timeout was needed to account for higher compile times.

3 CONTEXTUAL DISPATCH

With contextual dispatch we provide the means to keep several differently specialized function versions and then dynamically select a good candidate, given the dynamic context at the call-site. To gain some intuition, let us revisit the example of Listing 1 and contrast speculation, inlining and contextual dispatch. Figure 1 shows *idealized* compilation of that code. On the left we observe the two call-sites to the `max` function. In the first case, since both callers omit the `warning` parameter, a compiler could speculatively optimize it away, by leaving an `assume` to catch calls that pass the third argument. However any unrelated call-site in the program could invalidate this assumption and undo the speculative optimization for all callers simultaneously. Second, inlining allows us to specialize the `max` function for each call site independently. In R, inlining is generally unsound as functions can be redefined by reflection. Therefore the assumption of a static target for the inlined call-site is a speculative optimization. Inlining increases code size, and, is often disallowed if the called method is large. As a further drawback the compiler must specialize the function for each call-site anew and is limited to specialize on statically known information. For instance in `max(deserialize(readline()), 1)`, the argument type is dynamic and inlining does not allow us to specialize for it.

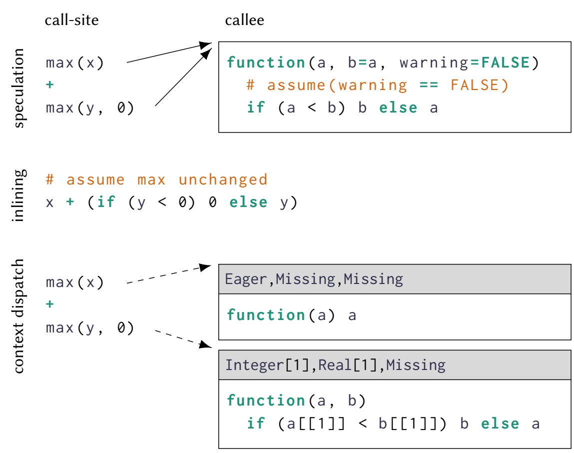

Fig. 1. Speculation, Inlining and Contextual dispatch

In contrast, as depicted in the last example in Figure 1, contextual dispatch allows the compiler to create two additional versions of the target function, one for each calling context. At run-time the dispatch mechanism compares the information available at the call-site with the required contexts of each available version and dynamically dispatches to one of them. Unlike inlining, there is no limit to the target function size, the specialization is bounded by the number of different contexts. Here we assume that type of `x` and `y` and the fact that they are scalar values, can not be inferred from the source code, but can be checked at run-time. Contextual dispatch is then realized by first approximating a current context. For instance if `x` is a scalar integer, then at the call-site `max(x)` a current context of `Integer[1]` could be inferred. Given this current context, a target version with a compatible context is invoked. In our example the context `Eager,Missing,Missing` is chosen. If no compatible specialized version is available, then we dispatch to the original version of the function, therefor contextual dispatch always succeeds. Compatibility is expressed in therms of ordering: contexts form a partial order, such that any smaller context logically entails the bigger one. In other words, a function version can be invoked if its context is bigger than the current one. In our implementation a context is an actual datastructure with an efficient representation and comparison. For each call-site the compiler infers a static approximation for the upper bound of the current context. Additional dynamic check further concretize and populate the approximation. This dynamically inferred current context has two uses. First, it serves as the lower bound when searching for a target version of a particular function. The target might not be unique as we will discuss later. Secondly, if no good approximation is found, then it serves as the assumption context to compile a fresh version to be added to the function.

Contextual dispatch shares some similarities with splitting, as depicted in ??. Similarly specialized copies of functions are created, for instance here `max2` is a copy of `max` which takes two arguments. Additionally, if there are multiple static candidates, then those are disambiguated at runtime by rewriting the call-site into a fixed sequence of branches. In this example we test for the length and in case of 1 call the copy `max2s`, specialized to receiving two scalar arguments. However, the specialization happens at compile-time and cannot be extended without recompilation. All those four techniques can be easily combined. For instance the performance of inlining can be improved by inlining an already optimized version using a static approximation of contextual dispatch. Or statically known candidates of likely contexts can be used statically by splitting on contexts.

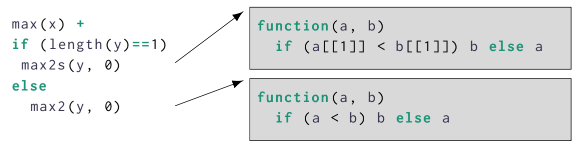

Fig. 2. Splitting

The following section provides a precise definition of contexts and contextual dispatch. Then, we present a more detailed account on the performance trade-offs of contextual dispatch. The actual instance of contextual dispatch as implemented in Ř is detailed in the later section 4.

3.1 Definitions

We envision a number of possible implementations of contextual dispatch. The following provides a general framework for the approach and defines key concepts.

Context. Contexts C are predicates over program states with an efficiently computable partial order $C_1 < C_2$ *iff* $C_1 \implies C_2$, *i.e.*, C_1 entails C_2. Let $\top$ be the context that is always true; it follows that $C < \top$ for all contexts C.

Current Context. A context is called current with respect to a state S if $C(S)$ holds.

Version. $\langle C, V \rangle$ is called a version, where V is code optimized under the assumption that C holds at entry. A function is a set of versions including $\langle \top, V_u \rangle$, where V_u is compiled without assumptions.

Dispatch. To invoke a function F in state S, the implementation chooses any current context C' (with respect to S) and a version $\langle C, V \rangle \in F$ such that $C' < C$ and transfers control to V.
The above definitions imply a notion of correctness for an implementation.

Theorem. *Dispatching to version $\langle C, V \rangle \in F$ from a state S and a current context C' implies $C(S)$.*

This follows immediately from the definition of the order relation. It means that dispatch transfers control to a version of the function compiled with assumptions that hold at entry.

We might want to require contexts be closed under conjunction. While not necessary, the benefits are that a unique smallest current context exists and the intersection of two current contexts is a more precise current context.

The above definitions may not necessarily lead to performance improvements; indeed, an implementation may choose to systematically dispatch to $\langle \top, V_u \rangle$. This is a correct choice as the version is larger than any current context but it also provides no benefits.

Heuristics. It is reasonable for an implementation to compute a smallest current context as that context captures the most information about the program state at the call site. On the other hand, increased precision might be costly, and thus an approximation may be warranted. An implementation may also compute the current context by combining static and dynamic information. For instance, one may be able to determine statically that $C \equiv$ `type(arg0)==int` holds, perhaps because that argument appears as a constant, whereas $C' \equiv$ `type(arg1)==string` must be established by a run-time check. Given $C \wedge C'$ exists, it is a more precise current context.

Similarly, dispatch can select any version that is larger than the current context, but typically, one would prefer the smallest version larger than the current context, as it is optimized with the most information. But this choice can be ambiguous, as illustrated in Figure 2. There are four versions of the binary function F: V_u can always be invoked, V_{SS} assumes two strings as arguments, $V_{I?}$ assumes the first argument is an integer, and $V_{?I}$ assumes the second is an integer. Given F is invoked with two integers, *i.e.*, in state S, the current context $C_{I?} \wedge C_{?I} = C_{II}$ is not available as a version to invoke. The implementation can dispatch to either $C_{I?}$ or $C_{?I}$; however, neither is smaller than the other. Alternatively, the implementation can compile a fresh version $\langle C_{II}, V_{II} \rangle$ to invoke.

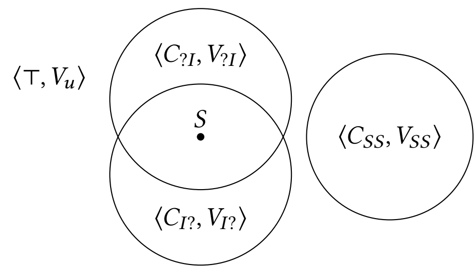

Fig. 3. Versions and current program state

The efficiency of dispatch depends on the cost of computing the current context and its order. Contexts can be arbitrarily complex, *e.g.*, "the first argument does not diverge" is a valid context. Given a call site `f(while(a){})`, we can establish this context using the conjunction of "if `a==FALSE` then `while(a){}` does not diverge" and "`a==FALSE`." The former is static, but the later is dynamic.

An implementation must decide when to add (or remove) versions. Each dispatch where the current context is strictly smaller than the context of the version dispatched to is potentially sub-optimal. The implementation can rely on heuristics for when to compile a new version that more closely matches the current context.

The compiler may replace contextual dispatch with a direct call to a version under a static current context. This has the benefit of removing the dispatch overhead and enabling inlining. The drawback is that the static context might be larger than the dynamic one and it forces the implementation to commit to a version early.

To make matters concrete, we give two examples of contextual dispatch:

(1) <u>Customized Compilation</u>: This technique introduced in SELF specializes methods to concrete receiver types, by duplicating them down the delegation tree. The technique can be understood as an instance of contextual dispatch. The contexts are type tests of the method receiver $C_A \equiv \text{typeof}(\mathit{self}) == A$. The order of contexts is defined as $C_A < C_B$ *iff* $A <: B$. It follows that if the receiver is of class A, and A is a subtype of B, dispatch can invoke a version compiled for B. In the Julia language, this strategy is extended to the types of all arguments.

(2) <u>Global Assumptions</u>: Contexts can capture predicates about the values of global variables, *e.g.*, $C \equiv \mathit{debug} == \text{true}$ or $C' \equiv \mathit{display} == \text{headless}$. If we allow such contexts to be combined, we get a natural order from $C \wedge C' < C$, *i.e.*, a smaller context is one that tests more variables. The smallest current context is the conjunction of all current singleton contexts. An interesting application is shown by Liu et al. [2019], where a dynamic analysis detects thread-local objects. The property is then used to dispatch to versions of their methods that elide locks.

3.2 Detailed Example

To illustrate the trade-offs when specializing functions, consider the map-reduce kernel written in R of Figure 3. The `reduce` function takes a vector or list `x` and iteratively applies `map`. The `map` function has two optional arguments, `op` which defaults to `"m"`, and `b`, which defaults to `1` when `op` is `"m"`. `map` is called twice from `reduce`: the first call passes a single argument and the second call passes two arguments. The type of the result depends on the type of vector `x` and the argument `y`. As a driver, we invoke `reduce` ten times with a vector of a million integers, once with a list of tuples, and again ten times with an integer vector. This example exposes the impact of polymorphism on the performance. Figure 3 (right) illustrates the execution time of each of the twenty measurements in seconds (smaller is better).

```
map <- function(a,
                b = if(op=="m") 1,
                op= "m") {
           if (op=="m")   a * b
   else if (op=="a")   a + b
   else error("unsupported")
}
reduce <- function(x, y=3, res=0) {
  for (i in x)
    res <- res + map(i) + map(i, y)
  res
}
for (i in 1:10)
  system.time(reduce(1L:1000000L))
reduce(list(c(1L,1L), c(2L,2L)))
for (i in 1:10)
  system.time(reduce(1L:1000000L))
```

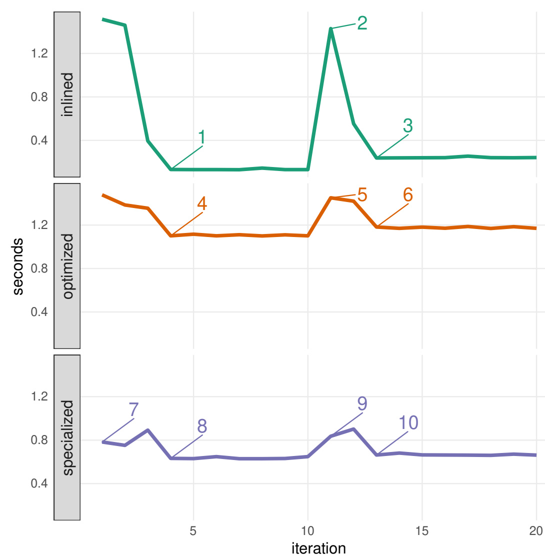

Fig. 4. Optimization strategies

The red line describes the results with inlining and speculation enabled. In this case, `map` is inlined twice. The point marked with (1) shows optimal performance after the compiler has finished generating code. However, the call to `reduce` with a list of tuples leads to deoptimization and recompilation (2). Performance stabilizes again (3), but it does not return to the optimal, as the code remains compiled with support for both integers and tuples. The green line shows the results with inlining of the `map` function manually disabled. After performance stabilizes (4), the performance gain is small. This can be attributed to the high cost of function calls in R. Again, we observe deoptimization (5) and re-optimization (6). The curve mirrors inlining, but with smaller speedups. Finally, the blue line exposes the results when we enable contextual dispatch (without inlining). The first iteration (7) is fast because `reduce` can benefit from the compiled version of `map` earlier, thanks to contextual dispatch of calls. Performance improves further when the `reduce` function is optimized (8). We see a compilation event at (9). Finally, we return to the previous level of performance (10), in contrast to the two previous experiments, where the deoptimization impacted peak performance. The reason is that the version of `map` used to process integer arguments is not polluted by information about the tuples, since they are handled by a different version.

Like inlining, contextual dispatch allows to specialize a function to its calling context. Unlike inlining, the specialized function can be shared across multiple call sites. While speculation needs deoptimization to undo wrong assumptions, contextual dispatch does not. Contextual dispatch applies at call boundaries, while speculation can happen anywhere in a function. Finally, let us repeat that these mechanisms are not mutually exclusive: the implementation of Ř supports all of them and we look forward to studying potential synergies in future work.

4 CONTEXTUAL DISPATCH IN Ř

Below we detail the implementation of contextual dispatch for the R. Given the complexity of function calls in R, our design focuses on properties that can optimize the function call sequence and allow the compiler to generate better code within the function.

4.1 Contexts

The goal of contextual dispatch is to drive optimizations. Accordingly, we design contexts in Ř mainly driven by the seven headaches for optimizing R introduced in section 2.2. Contexts are represented by the `Context` structure presented in Listing 2, which consists of two bit vectors (`argFlags` and `flags`) and a byte (`missing`). The whole structure fits within 64 bits, with two bytes (`unused`) reserved for future use. The `EnumSet` class is a set whose values are chosen from an enumeration.

```
enum class ArgAssumption {
  Arg0Eager,      ..., Arg7Eager,      Arg0NotObj,      ..., Arg7NotObj,
  Arg0SimpleInt, ..., Arg7SimpleInt,  Arg0SimpleReal, ..., Arg7SimpleReal,
};
enum class Assumption {
  NoExplicitlyMissingArgs, CorrectArgOrder, NotTooManyArgs, NoReflectiveArg,
};
struct Context {
  EnumSet<ArgAssumption, uint32_t> argFlags;
  EnumSet<Assumption, uint8_t> flags;
  uint8_t missing = 0;
  int16_t unused = 0;
};
```

Listing 2. Context data structure

More specifically, `argFlags` is the conjunction of argument predicates (`ArgAssumption`) for the first eight arguments of a function. For each argument position `N < 8`, we store if the argument has already been evaluated (`ArgNEager`), if the argument is not an object, *i.e.*, it does not have a **`class`** attribute (`ArgNNotObj`), if the value is a scalar integer with no attributes (`ArgNSimpleInt`), and if the value is a scalar double with no attributes (`ArgNSimpleReal`). Any subsequent arguments will not be specialized for. The limit is informed by data obtained by Morandat et al. [2012], suggesting that the majority of frequently called functions have no more than three arguments and that most arguments are passed by position.

The `flags` field is a set of `Assumption` values that summarize information about the whole invocation. The majority of the predicates are related to argument matching. In R, the process of determining which actual argument matches with formal parameters is surprisingly complex. The GNU R interpreter does this by performing three traversals of a linked list for each function call. The Ř compiler tries to do this at compile time, but some of the gnarly corners of R get in the way. For this reason, contexts encode information about the order of arguments at the call site. Thus `flags` has a predicate, `NoExplicitlyMissingArgs`, to assert whether any of the arguments is *explicitly* missing. This matches in three cases: when an argument is explicitly omitted (`add(,2)`), when an argument is skipped by matching (`add(y=2)`), and when a call site has more missing arguments than expected in the compiled code. `CorrectArgOrder` holds if the arguments are passed

```
bool Context::smaller(const Context& other) const {
  // argdiff positive = "more than expected", negative = "less than"
  int argdiff = (int)other.missing - (int)missing;
  if (argdiff > 0 && other.flags.contains(Assumption::NotTooManyArgs))
    return false;
  if (argdiff < 0 && other.flags.contains(Assumption::NoExplicitlyMissingArgs))
    return false;
  return flags.includes(other.flags) &&
         typeFlags.includes(other.typeFlags);
}
```

Listing 3. Implementation of Context ordering

in the order expected by the callee. `NotTooManyArgs` holds if the number of arguments passed is less than or equal to the number of parameters of the called function. `NoReflectiveArg` holds if none of the arguments invoke reflective functions. Finally, missing arguments that occur at the end of an argument list are treated specially; `missing` records the number of *trailing* missing arguments (up to 255).

4.2 Ordering

Recall that contexts have a computable partial order, which is used to determine if a function version can be invoked at a particular program state. For example, let C' be the current context of program state S, F be a function invoked at S, and $\langle C, V \rangle$ be a version in F. Then the implementation can dispatch to $\langle C, V \rangle$ if $C' < C$.

In Ř, the order between contexts, $C' < C$, is defined mainly by set inclusion of both assumption sets. For trailing missing arguments, there are two cases that need to be considered. First, if C assumes `NotTooManyArgs`, then C' must have at least as many trailing missing arguments as C. Otherwise, this implies C' has more arguments than C expects, contradicting `NotTooManyArgs`. Second, if context C allows any argument to be missing, then it entails a context C' with fewer trailing missing arguments (*i.e.*, more arguments). The reason is that *missing arguments* can be passed as explicitly missing arguments, reified by an explicit marker value for missing arguments. If we invert that property, it means that a context with `NoExplicitlyMissingArgs` does not accept more trailing missing arguments.

Some example contexts with their order relation (increasing from left to right) are shown in Figure 4. Contexts with more flags are smaller, contexts with a greater `missing` value are smaller, and contexts with `NoExplicitlyMissingArgs` require the same number of missing arguments to

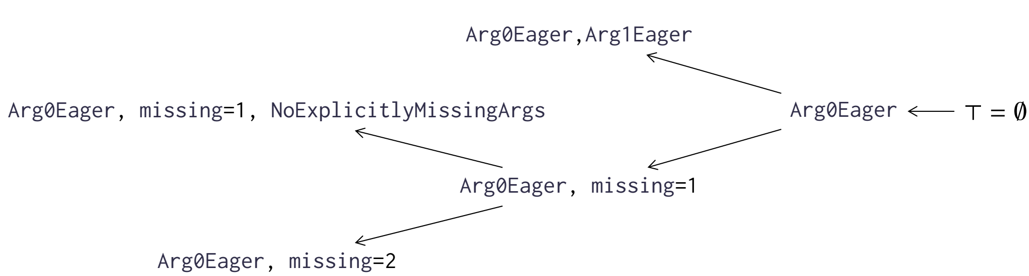

Fig. 5. An example for the order of some Contexts

be comparable. The comparison is implemented by the code of Listing 3. Excluding `mov` and `nop` instructions, the `smaller` comparison is compiled to fewer than 20 x86 instructions by GCC 8.4.

4.3 Evaluating Contexts

For every call the current context is needed, which is partially computed statically and completed dynamically. The Ř optimizer enables static approximation of many of the assumptions. For example, laziness and whether values might be objects are both represented in the type system of its IR. Therefore, those assumptions can sometimes be precomputed. Call sites with varargs passed typically resist static analysis. On the other hand, `NoExplicitlyMissingArgs`, `CorrectArgOrder`, and `NotTooManyArgs` are static assumptions for call sites without named arguments, or if the call target is known and the argument matching can be done statically. Similarly, the number of missing trailing arguments are only statically known if the call target is static.

The most interesting assumption in terms of its computation is `NoReflectiveArg`. Since the context has to be computed for every dispatch, the time budget is very tight. Therefore, this assumption is only set dynamically if all arguments are eager, which is a very conservative over-approximation. However, we perform a static analysis on the promises to detect benign ones, which do not perform reflection. This shows that even computationally heavy assumptions can be approximated by a combination of static and dynamic checks.

The static context is computed at compile time and added to every call instruction. At call time, a primitive function implemented in C++ supplements all assumptions which are not statically provided. This seems like a gross inefficiency—given the static context, the compiler could for each call site generate a specific and minimal check sequence. We plan to add this optimization in the future. So far we have observed the overhead of computing the context to be small compared with the rest of the call overhead.

4.4 Dispatch Operation

All the versions of the same function are kept in a *dispatch table* structure, a list of versions sorted by increasing contexts. Versions with smaller contexts (*i.e.*, with more assumptions) are found in the front. To that end we extend the partial order of contexts to a total order: if two contexts are not comparable then the order is defined by their bit patterns.

Listing 4 shows a pseudocode implementation of the dispatching mechanism. Dispatching is performed by a linear search for the first matching context (see Listing 3). The result of a dispatching operation can be cached, since given the same dispatch table and context, the result is deterministic. If the context of the dispatched version is strictly larger than the current context, it means that there is still an opportunity to further specialize. We rely on a counter based heuristic to trigger the optimizer. At the time of writing, dispatch tables are limited to 15 elements; to insert an entry into a full table, a random entry (except the first) is evicted.

4.5 Optimization Under Assumptions

Optimizations in Ř are performed on an intermediate representation called PIR. The version of PIR shown here is a simplification of the actual representation in the compiler. For an in-depth discussion of PIR, we refer to Flückiger et al. [2019]. Our work extends the model with explicit type annotations, a feature that is extensively used in the real system. The grammar of PIR is shown in **??**. Below, we briefly illustrate some optimizations relying on the contextual assumptions introduced in the previous sections. For the following examples, it is important to know that values in PIR (as in R) can be lazy. When a value is used in an eager operation, it needs to be evaluated first, by the `Force` instruction. PIR uses numbered registers to refer to the result of instructions. Those are

```
Version* dispatch(Context staticCtx, Cache* ic, DispatchTable* dt) {
  Context cc = computeCurrentContext(staticCtx);
  if (ic->dt == dt && ic->context == cc)
    return ic->target;  // Cache hit
  Version* res = dt->find([&](Version* v){ return cc.smaller(v->context); });
  if (res->context != cc && jitThresholdReached(res))
    res = optimize(dt, res, cc);   // Compile a better version
  updateCache(ic, dt, res, cc);
  return res;
}
```

Listing 4. Dispatching to function versions under the current context

not to be confused with source-level R variables, which must be stored in first-class environments. Environments are also first-class entities in PIR, represented by the MkEnv instruction.

```
   cls  %1 = LdFun (f, G)
int$~   %2 = LdConst [1]
   any  %3 = Call %0 (%1) G
```

As a simple example, consider the R expression f(1) that calls the function f with the constant 1. This translates to the PIR instructions on the right. The first instruction loads a function called f from the global environment. The second instruction loads the constant argument, which is a unitary vector [1]. This instruction has type integer and additionally the value is known to be scalar ($) and eager (~). The third instruction is the actual call. The static context for this call contains the Arg0SimpleInt and Arg0Eager flags. Assuming the call target is unknown, the result can be lazy and of any type.

```
  any  %0 = LdArg (0)
envir  %1 = MkEnv (x = %0 : G)
 any~  %2 = Force (%0) %1
             Return (%2)
```

R variables are stored in first-class environments. As an example, consider the body of the identity function, `function(x) x`, shown in PIR on the right. The first instruction LdArg loads the first argument. In general, the arguments can be passed in different orders, and the presence of varargs might further complicate matters. However, all functions optimized using PIR are compiled under the CorrectArgOrder assumption. This allows us to refer to arguments by their position, since it is now the caller's responsibility to reorder arguments as expected. The MkEnv instruction creates a first-class R environment to store variables. The name x is bound to the first argument and the global environment is the parent. The later means that this closure was defined at the top level. Finally, the argument, which is a promise, is evaluated to a value (indicated by the ~ annotation) by the Force instruction and then returned.

```
 any  %0 = LdArg (0)
any~  %2 = Force (%0)
            Return (%2)
```

It is worth noting that the Force instruction has a dependency on the environment. This is required since forcing promises can cause arbitrary effects, including reflective access to the local environment of the function. Under the NoReflectiveArg assumption, this dependency can be removed, because the assumption ensures that no argument can invoke a reflective function. Since this dependency was the only use of the local environment, it can be completely removed.

```
any~  %0 = LdArg (0)
            Return (%0)
```

The Force instruction is still effectful. However, if we can show that the input is eager, then the Force does nothing. Under the Arg0Eager assumption, we know the first argument is evaluated and therefore the Force instruction can be removed. In summary, three assumptions `CorrectArgOrder`, `NoReflectiveArg`, `Arg0Eager` were necessary to conclude that the function implements the identity function.

Another problem we target with contextual dispatch is argument matching. Consider the following `function(a1, a2=a1) {a2}`, which has a default expression for its second argument. This function translates to the PIR on the right. As can be seen, the second argument must be explicitly checked against the missing marker value. The default argument implies that we must dynamically check the presence of the second argument and then evaluate either `a1` or `a2` at the correct location. Default arguments are, like normal arguments, evaluated by need.

```
BB0 :  any     %0 = LdArg (0)
       any     %1 = LdArg (1)
       envir   %2 = MkEnv (a0 = %0, a1 = %1  : G)
       lgl$∼   %3 = Eq (%1, missing)
                    Branch (%3, BB1, BB2)
BB1 :  any∼    %4 = Force (%0) %1
                    Return (%4)
BB2 :  any∼    %5 = Force (%1) %1
                    Return (%5)
```

Optimized under a context where the last trailing argument is missing, this test can be statically removed. With this optimization, basic block 2 is unreachable. Note that as in the simpler example before, under the additional `Arg0Eager` assumption, the `Force` instruction and the local environment can be statically elided and the closure does not need an R environment.

```
BB0 :  any     %0 = LdArg (0)
       any     %1 = LdArg (1)
       envir   %2 = MkEnv (a0 = %0, a1 = %1  : G)
       any∼    %4 = Force (%0) %1
                    Return (%4)
```

Almost all of these specializations can also be applied using speculative optimizations. For instance, the previous example could be speculatively optimized as follows: instead of contextual assumptions, the speculative assumptions that `a1` is missing and that `a0` is eager are explicitly tested. The `Assume` instruction guards assumptions and triggers deoptimization through the last checkpoint. Creation of the local environment is delayed and only happens in case of a deoptimization. As can be seen here, the need to materialize a local environment on deoptimization is a burden on the optimizer and additionally, this method does not allow us to specialize for different calling contexts separately. However, there are of course many instances where speculative optimizations are required, since the property of interest cannot be checked at dispatch time. For instance, the value of a global binding might change between function entry and the position where speculation is required.

```
BB0 :  any     %0 = LdArg (0)
       any     %1 = LdArg (1)
       cp      %4 = Checkpoint BB1
       lgl$∼   %5 = Eq (%1, missing)
       lgl$∼   %6 = Is⟨any∼⟩(%0)
                    Assume (%5, %6) %4
                    Return (%0)
BB1 :  envir   %2 = MkEnv (a0 = %0, a1 = %1  : G)
                    Deopt (baseline, %0, %1, %2)
```

Other optimizations. In addition to contextual dispatch, Ř supports inlining, speculative optimizations, optimizations specific to R such as scope resolution, and traditional optimizations including global value numbering, hoisting, and escape analysis. These techniques are not mutually exclusive, and in fact, they are complementary. In the next section, we conduct a baseline performance comparison, and then evaluate the performance contribution of contextual dispatch.

5 EMPIRICAL EVALUATION

In this section, we empirically validate the impact of contextual dispatch. Our approach is to first establish a baseline performance by comparing our system to two existing implementations, and then to drill down in the results and show the contributions of different assumptions of a context to performance.

5.1 Methodology

Non-determinism in processors, *e.g.*, due to frequency scaling or power management, combined with the adaptive nature of just-in-time compilation, make reporting experimental results challenging. We identify outliers by running the same performance experiment automatically on every commit and comparing histories. To deal with warmup phases of the virtual machine, *i.e.*, iterations of a benchmark during which compilation events dominate performance, we run each benchmark fifteen times in the same process and discard the first five iterations. To further mitigate the danger of incorrectly categorizing the warmup phase [Barrett et al. 2017], we plot individual measurements in the order of execution. The graphs in **??** visualize the warmup behavior.

An important question when comparing implementations is their compliance; partial implementations can get speedups by ignoring features that are difficult to optimize. The GNU R interpreter is the reference implementation, so it is compliant by definition. As of this writing, Ř is compliant with version 3.6.2 of GNU R, verified by running the full GNU R test suite and the tests of its recommended packages. FastR is not fully compliant with GNU R, but we believe it adheres to the R semantics in the benchmarks included in this paper.[2]

The selection of benchmarks is important. The suite used in this paper consists of 59 programs that range from micro-benchmarks, solutions to small algorithmic problems, and real-world code. Some programs are variants; they use different implementations to solve the same problem. We categorize the programs by their origin:

- $\underline{\mu}$ Code fragments known by the R community to be slow. While too small to draw conclusions from, their performance is easier to analyze than the larger benchmarks.
- <u>awf</u> We translated three benchmarks to R from Marr et al. [2016]: `Bounce`, a bouncing balls physics simulation; `Mandelbrot`, to compute the Mandelbrot set; and `Storage`, a program that creates trees.
- <u>sht</u> The Computer Language Benchmarks Game [Gouy 2020], ported to R by Kalibera et al. [2014]. The suite contains multiple versions of classic algorithms, written to explore different implementation styles. Most of the original programs had explicit loops, so the suite provides more idiomatic R versions that rely on vectorized operations.
- <u>re</u> `Flexclust` is a clustering algorithm from the flexclust package [Leisch 2006]. It exercises many features that are hard to optimize, such as generic methods, reflection, and **`lapply`**. `Convolution` consists of two nested loops updating a numerical matrix; it is an example of code that is typically rewritten in C for performance.

We ran experiments on a dedicated eight-core i7-6700K CPU, clocked at 4 GHz, stepping 3, μcode version 0xd6, with 32 GB of RAM and Ubuntu Bionic on Linux kernel version 4.15.0-88. GitLab runner version 12.9.0 executed each benchmark with a harness compatible with the ReBench [Marr 2018] framework in Docker version 19.04.8. For the baseline performance experiment, we used GNU R version 3.6.2, released in December 2019; FastR 3.6.1, part of GraalVM 19.3.1, released in January 2020; and Ř commit bc1933dd.

[2] In our experiments, we were unable to make FastR pass 5 out of 15 of the recommended packages in GNU R's test suite and we discovered a bug that could lead to integer overflow warnings being suppressed.

5.2 Baseline Performance Comparison

Studying the performance of GNU R, FastR, and Ř allows us to compare a lightly optimizing bytecode interpreter and two optimizing just-in-time compilers, where one also implements contextual dispatch. The systems feature different implementation strategies and trade-offs. This comparison is therefore mainly to show that Ř is competitive with a state of the art optimizing compiler, to highlight the significance of the results of the evaluation of contextual dispatch in the next section.

We start by explaining the format of our results. Figure 5 shows the `spectralnorm` benchmark as an example. The y-axis measures the speedup compared to the execution time of the GNU R interpreter (higher is better, scale is logarithmic). For each of Ř (left) and FastR (right), a box plot shows the results of the ten runs. Here, the box plot collapses to a single line as performance is stable in both systems. The individual observations are overlaid on top of the box plots as a time series. Warmup iterations are excluded from the box plot, but displayed on the graph as black crosses.

In this benchmark, Ř and FastR have comparable performance profiles. The first three warmup iterations are slow, about the speed of the GNU R interpreter, then performance is over 30× faster with no large outliers. Ř has a slower warmup, due to very simple heuristics for when a function is optimized.

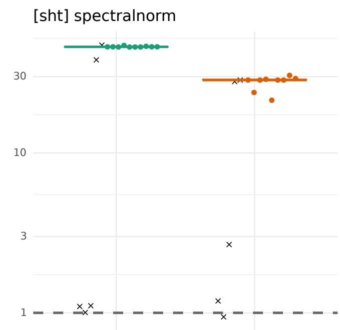

Fig. 6. Performance of Ř (left) and FastR (right), normalized to GNU R (dashed line)

Figure 6 shows results for 16 of the 59 benchmarks. This selection excludes the [μ] benchmarks and includes one variant for each program, other than `nbody` where we selected two variants. To avoid cluttering the graphs, warmup is not pictured. Table 1 summarizes the range of speedups per benchmark family. The [μ] benchmark results are best ignored as they include benchmarks that are mostly optimized away (*e.g.*, the maximum speedup for Ř relative to GNU R is 483949×). The full set of results and the warmup times are given in ??.

Ř can achieve a similar performance to FastR, but can also be significantly slower when the benchmark relies on features that are not optimized in Ř. It is noteworthy that Ř is slower than GNU R on `flexclust`: this is because the benchmark uses features that currently cause Ř to give up compiling some methods. Ř is also slightly slower on `knucleotide`, `nbody`, and `pidigits`. Ř's performance relative to GNU R ranges from a 0.7× slowdown to a 46× speedup, with the overall speedup being 1.7×.

While FastR can indeed be fast, it is worth noting that there is a large variance for peak performance in both directions, when compared to the GNU R interpreter. For instance `Storage` and `spectralnorm_math` are much slower than GNU R and `pidigits` and `binarytrees` have very large amounts of variability. Finally, we had to exclude `flexcust` and `regexdna` from our measurements since they ran orders of magnitude slower in FastR per iteration. Excluding those, Ř is between 0.1× slower and 6× faster than FastR, with an overall slowdown of 0.58×. We observe that it is difficult to meaningfully compare FastR to Ř, due to the large variability in the performance relative to GNU R. For instance two small programs from the [μ] family, `listWhile` and `listFor`, time out in FastR due to incremental growth of a vector, leading to a large garbage collection overhead. Benchmarks are run in the default configuration, however we verified that manually specifying JVM heap limits between 1G and 12G does not lead to fundamental differences in results. In GNU R and Ř it is not possible to configure a global heap limit.

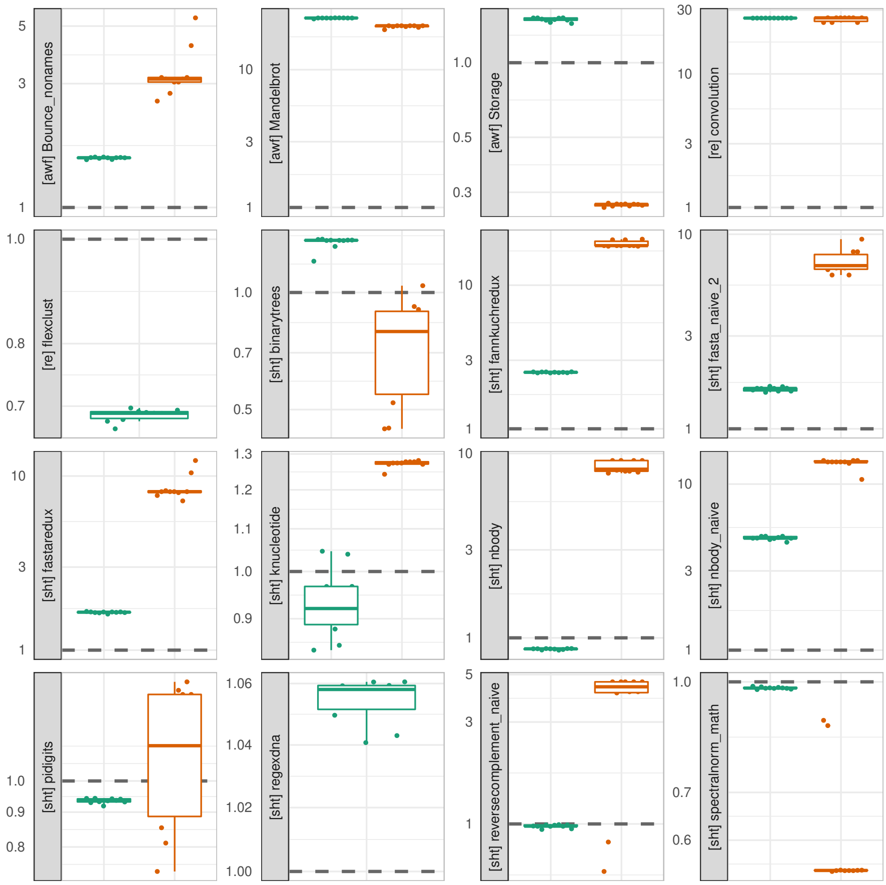

Fig. 7. Performance of Ř (left) and FastR (right), normalized to GNU R (dashed line)

Suite	min.	max.	geom. mean
[awf]	1.03	23.7	2.49
[re]	0.682	26.2	1.71
[sht]	0.756	46	1.58
overall	0.682	46	1.68
[μ]	0.791	483949	52.3

(a) Ř versus GNU R

Suite	min.	max.	geom. mean
[awf]	0.25	5.59	0.85
[re]	1.08	1.32	1.23
[sht]	0.108	4.2	0.52
overall	0.108	5.59	0.584
[μ]	0.0126	6286	4.58

(b) Ř versus FastR

Table 1. Speedups of Ř (excluding missing measurements and "overall" excluding [μ])

5.3 Performance of Contextual Dispatch

The previous section examined the performance of Ř with all of its optimizations enabled. How much of that performance is due to contextual dispatch? To answer that question, we disable individual assumptions that make up a context and study their impact on performance. Importantly, each and every assumption has an equivalent substitute speculative optimization in Ř's optimizer as described in section 4.5. Performance improvements in this section are therefore not due to additional speculative capabilities, but solely due to splitting into multiple versions and the specialization to multiple contexts, or due to reduced overhead from having less deoptimization points.

Unfortunately, it is not possible to turn off contextual dispatch altogether as it is an integral part of the Ř compiler. Each function starts with a dispatch table of size one, populated with the unoptimized version of the function. To achieve a modicum of performance, it is crucial to add at least one optimized version to the dispatch table. The unoptimized version cannot be removed as it is needed as a deoptimization target. What we can do is to disable some of the assumptions contained within a context. Thus, to evaluate the impact of contextual dispatch, we define seven, cumulative, optimization levels:

- L0 `NotTooManyArgs` and `CorrectOrder` are fundamental assumptions required by Ř;
- L1 `ArgNEager` for arguments that are evaluated promises;
- L2 `NoReflectiveArg` specifies that promises do not use reflection;
- L3 `ArgNNotObj` for arguments that do not have the **`class`** attribute;
- L4 `ArgNSimpleInt` or `ArgNSimpleReal` for arguments that are scalars of integers or doubles;
- L5 `missing` for a lower bound on missing arguments (from the end of argument list); and
- L6 `NoExplicitlyMissingArgs` to ensure that `missing` is the exact number of missing arguments.

For this experiment we, pick L0 as the baseline, as it is the optimization level with the fewest assumptions in the context. For each benchmark, we report results for each of L0 to L6, normalized to the median execution time of L0 (higher is better).

Figure 7 shows the results of the experiment for `spectralnorm`. Each level has its own box plot. The first box plot from the left is for L0 (and its median is set to one) and the last corresponds to L6. Dots show individual measurements. The blue line is the lower bound of the 95% confidence interval of a linear model. In other words, `spectralnorm` is predicted to improve at least 4.6% due to contextual dispatch. The largest changes in the emitted code can be seen in L2. The `NoReflectiveArg` assumption enables the optimizer to better reason about several functions. These optimizations are preconditions for the jump in L6, but yield fewer gains themselves. The improvement in L6 can be pinpointed to the builtin `double` function, with the signature **`function`**`(length=0L)`. The `NoExplicitlyMissingArgs` assumption allows us to exclude the default argument. The function is very small and is inlined early. However, statically approximated contextual dispatch allows the compiler to inline a version of the `double` function, which is already more optimized. This gives the optimizer a head start and leaves more optimization budget for the surrounding code.

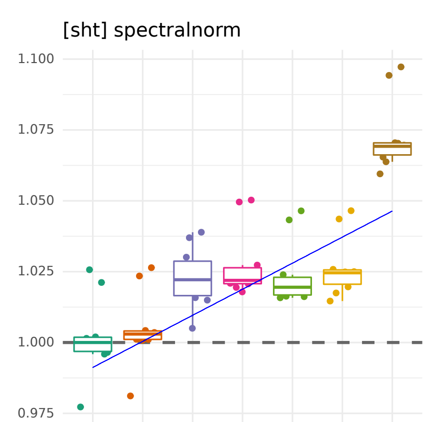

Fig. 8. Impact of optimization levels 0 to 6 (from left to right)

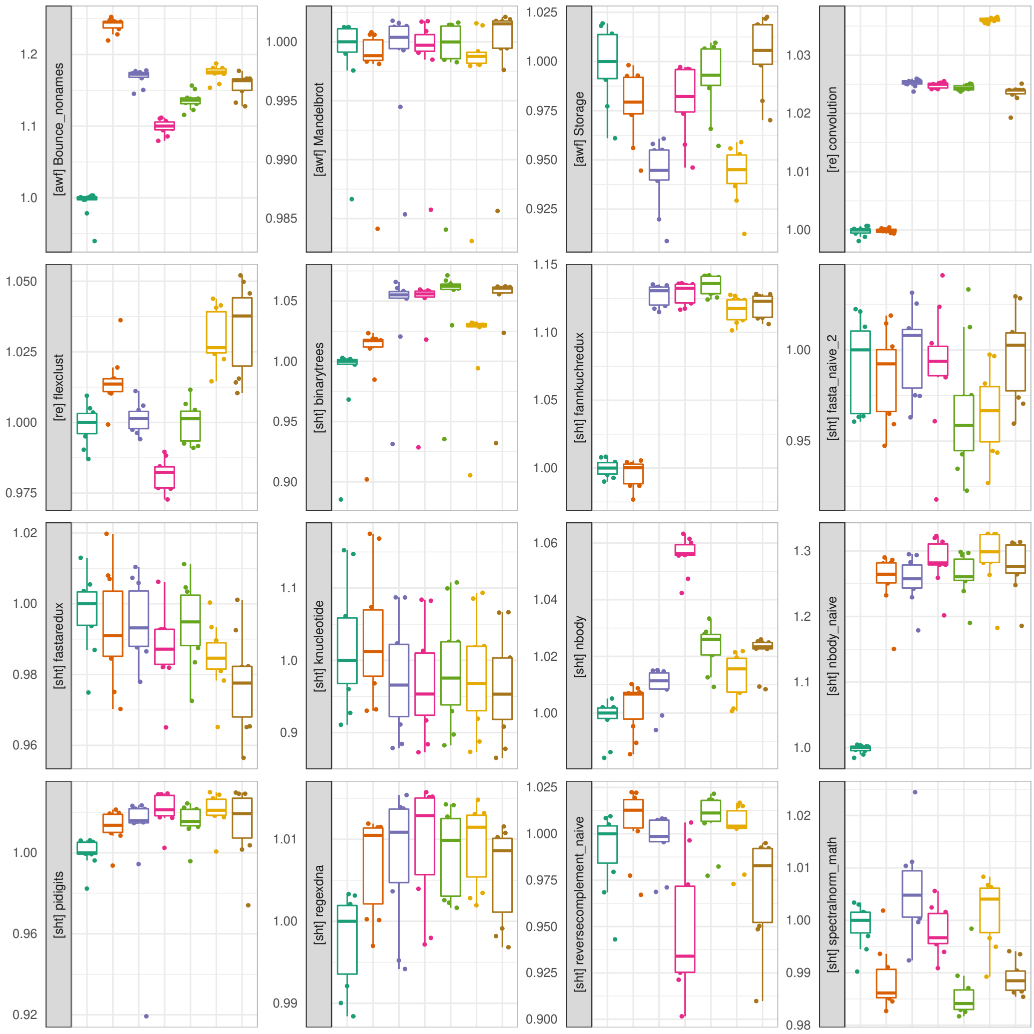

Fig. 9. Impact of optimization levels 0 to 6 (left to right)

Results. Figure 8 shows the performance impact of contextual dispatch on 16 of the 59 benchmarks. This is the same selection as in Figure 6, where we exclude the [μ] benchmarks and most variants. The complete results appear in **??**.

In general, we see a trend for higher levels to execute faster. The effects are sometimes fairly small; note that the each graph has a different scale. The outliers in `binarytrees` are caused by garbage collection. Some benchmarks have a large response on L1 or L2. The reason is that benchmarks are invoked with a workload constant as an argument, and the assumptions from those levels help with deducing that this argument is benign.

The aim of our experiment is to test if contextual dispatch significantly contributes to the overall performance of Ř. Often, optimizations do not benefit all programs uniformly, and can even degrade performance in some cases. We are therefore interested in the number of benchmarks which are significantly improved (or not worsened) over a certain threshold. We formulate the null hypothesis:

speedup	[awf][sht][re]	[μ]
-5%	44	11
0%	18	10
2%	11	8
5%	6	8
10%	4	7
20%	1	7

Table 2. Number of benchmarks significantly improved (¬H0 with $p = .05$) out of 46, and 13 ([μ])

H0 Contextual dispatch does not speed up the execution of a benchmark by more than N%.

We test H0 for various improvement thresholds, by fitting a linear model and testing its prediction for the lower bound of the 95% confidence interval at L6 (see the blue line in Figure 7). As can be seen in the summarized results from Table 2, we can conclude that contextual dispatch might slow down the execution of two benchmarks by more than 5%, improve 39% of benchmarks, and improves four benchmarks by more than 10%. Additionally, more than half of the benchmarks in [μ] see a speedup greater than 20%.

Discussion. The effects reported in this section can sometimes be subtle. The reasons are that Ř is already a fairly good optimizer without contextual dispatch. It employs a number of optimizations and speculative optimizations, which speculate on the same properties. We investigated the number of versions per function in `pidigits` and found them to range between 1 and 6. Many functions that belong to the benchmark harness or are inlined stay at 1 or 2 versions with few invocations. The functions with many versions concentrate on a few. A big hurdle for contextual dispatch in R is that it is not possible the check the types of lazy arguments at the time of the call. For instance, there is a user-provided `add` function that has 12 call sites with several different argument type combinations. However, Ř is not able to separate the types with contextual dispatch, because all call sites pass both arguments lazily. As predicted in section 3, this results in several deoptimizations and re-compilations, leading to a fairly generic version in the end. We see this as an exciting opportunity for future work, as it seems that contextual dispatch should be extended from properties that *definitely* hold at function entry to properties that are *likely* to hold at function entry. This would allow for multiple versions, each with different speculative optimizations, to be dispatched to depending on how likely a certain property is.

We investigated if garbage collection interferes with measurements. To that end, we triggered a manual garbage collection before each iteration of the experiment. Indeed, we observed slightly more significant results for the numbers reported in Table 2. To keep the methodology consistent with the previous section, where manually triggering a garbage collection would distort the results, we decided to keep the unaltered numbers.

We find the results presented in this section very encouraging. Additionally, and this is difficult to quantify, we believe that contextual dispatch has helped improve Ř in two important ways. First, there is a one-stop solution for specialization. This makes it easy to add new optimizations based around customizing functions, but we also use it extensively in the compiler itself. The compiler uses static contexts to keep different versions of functions in the same compilation run, to drive splitting and for more precise inter-procedural analysis. The second benefit is that contextual dispatch has helped to avoid having to implement each and every one of the painstakingly many corner cases of the R language. For instance, we can assume that arguments are passed to functions in stack order, and if for one caller our system does not manage to comply with this obligation, contextual dispatch automatically ensures that the baseline version without this assumption is invoked.

6 CONCLUSION

Just-in-time compilers optimize programs by specializing the code they generate to past program behavior. The combination of inlining and speculation is widely used in dynamic languages, as inlining enlarges the scope of a compilation unit and thus provides information about the context in which a function is called, and speculation enables inlining and allows to avoid uncommon branches.

This paper proposes a complementary technique, contextual dispatch, which allows the compiler to manage multiple specialized versions of a function, and to select the most appropriate version dynamically based on information available at the call site. The difference with inlining and speculation is that contextual dispatch allows a different version of a function to be chosen at each and every invocation of that function with modest dispatching overhead. Whereas inlining requires the compiler to commit to one particular version, and speculation requires the compiler to deoptimize the function each time a different version is needed.

The key design choice for an implementation of this approach is to pick contexts that have an efficiently computable partial ordering. We envision compilers for different languages defining their own specific contexts. The choice of context is also driven by the cost of deriving them from current program state, and the feasibility of approximation strategies.

Our implementation of contextual dispatch in Ř was evaluated on a benchmark suite composed of a number small algorithmic problems and a few real-world programs. Relative to the GNU R reference implementation, Ř with contextual dispatch achieves an average speedup of 1.7×, with the worst being a 0.7× slowdown and the best being a 46× speedup. Evaluating the contribution of contextual dispatch, we observed that it improves performance in 18 out of 46 programs in our benchmark suite, and in 10 out of 13 micro-benchmarks.

In future work we are considering extending the set of predicates that make up a context. To add richer properties and increase flexibility, we may have to change the dispatching technique. One idea would be to develop a library of simple building blocks, such as predicates, decision trees and numerical ordering; such that their combination still results in a context with efficient implementation and representation. The key challenge will be to control the cost of deriving contexts at run-time. For this we are considering improving our compiler's ability to evaluate contexts statically. Another direction comes from the observation that different contexts can lead to code that is almost identical, we will investigate how to prevent generating versions that do not substantially improve performance.

As for broader applicability, we believe contextual dispatch can be used even in typed languages to capture properties that are not included in the type system of the language. For instance, in Java one could imagine dispatching on the erased type of a generic data structure, on the length of an array, or on the fact that a reference is unique. Whether this will lead to benefits is an interesting research question.

ACKNOWLEDGMENTS

We greatly thank all the people contributing to Ř over the years. Special thanks to Peta Maj for starting the project and always satisfying our hunger for more CPU cores. We thank the anonymous reviewers, for their insightful comments and suggestions to improve this paper. This work has received funding from the Office of Naval Research (ONR) award 503353, the National Science Foundation awards 1544542 and 1618732, the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412.

REFERENCES

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3133876

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

Craig Chambers and David Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/73141.74831

Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.101

Jeffrey Dean, Craig Chambers, and David Grove. 1995. Selective Specialization for Object-Oriented Languages. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/223428.207119

Olivier Flückiger, Guido Chari, Jan Jecmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/3359619.3359744

Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. 2020. Artifact for "Contextual Dispatch for Function Specialization". https://doi.org/10.5281/zenodo.3973073

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158137

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1542476.1542528

Isaac Gouy. 2020. Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/143095.143114

Antony L Hosking, J Eliot, and B Moss. 1990. *Towards compile-time optimisations for persistence.* Technical Report.

Toms Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/2576195.2576205

Friedrich Leisch. 2006. A Toolbox for K-Centroids Cluster Analysis. *Computational Statistics and Data Analysis* 51, 2 (2006).

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2019. Accelerating Sequential Consistency for Java with Speculative Compilation. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314611

Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. https://doi.org/10.5281/zenodo.1311762 Version 1.0.

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/2989225.2989232

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-31057-7_6

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot Server Compiler. In *Symposium on Java Virtual Machine Research and Technology (JVM)*.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. https://www.R-project.org

Robert W. Scheifler. 1977. An Analysis of Inline Substitution for a Structured Programming Language. *Commun. ACM* 20, 9 (1977). https://doi.org/10.1145/359810.359830

Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/2989225.2989236

Justin Talbot, Zachary DeVito, and Pat Hanrahan. 2012. Riposte: A Trace-driven Compiler and Parallel VM for Vector Code in R. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1145/2370816.2370825

Luke Tierney. 2019. *A Byte Code Compiler for R.* www.stat.uiowa.edu/~luke/R/compiler/compiler.pdf
John Whaley. 1999. *Dynamic optimization through the use of automatic runtime specialization.* Ph.D. Dissertation. Stanford.