Reusing Just-in-Time Compiled Code

MEETESH KALPESH MEHTA, IIT Mandi, India
SEBASTIÁN KRYNSKI, Czech Technical University in Prague, Czechia
HUGO MUSSO GUALANDI, Czech Technical University in Prague, Czechia
MANAS THAKUR, IIT Bombay, India
JAN VITEK, Northeastern University, USA

Most code is executed more than once. If not entire programs then libraries remain unchanged from one run to the next. Just-in-time compilers expend considerable effort gathering insights about code they compiled many times, and often end up generating the same binary over and over again. We explore how to reuse compiled code across runs of different programs to reduce warm-up costs of dynamic languages. We propose to use *speculative contextual dispatch* to select versions of functions from an *off-line curated code repository*. That repository is a persistent database of previously compiled functions indexed by the context under which they were compiled. The repository is curated to remove redundant code and to optimize dispatch. We assess practicality by extending Ř, a compiler for the R language, and evaluating its performance. Our results suggest that the approach improves warmup times while preserving peak performance.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Dynamic analysis.*

Additional Key Words and Phrases: Specialization, Code reuse, JIT compilation.

ACM Reference Format:
Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 263 (October 2023), 22 pages. https://doi.org/10.1145/3622839

1 INTRODUCTION

Just-in-time compilers are used to improve the performance of programs written in dynamic languages. They are effective for code that, statically, appears to have a large variety of behaviors but, in practice, only exercises a small subset of those behaviors in any given run. For example, consider a function that operates on strings and numbers. By delaying code generation long enough to observe how the function is used the compiler may speculate about its future invocations and generate specialized code. Just-in-time compiler architectures vary from language to language. Even within a single language there is a wealth of design choices. For concreteness, we illustrate the discussion with Ř, our experimental platform in this paper. Fig. 1 has a simplified overview of its architecture. Source code is translated to lightly optimized bytecode called RIR. This bytecode is instrumented with instructions for recording various forms of feedback, such as the type of variables, the identity of functions, and the direction of branches. Once a function has feedback information, it is scheduled for compilation. The compiler specializes the function and performs high-level optimizations that include inlining and unboxing in a hardware-independent representation called

Authors' addresses: Meetesh Kalpesh Mehta, s20012@students.iitmandi.ac.in, IIT Mandi, India; Sebastián Krynski, skrynski@gmail.com, Czech Technical University in Prague, Czechia; Hugo Musso Gualandi, hugo.gualandi@fit.cvut.cz, Czech Technical University in Prague, Czechia; Manas Thakur, manas@cse.iitb.ac.in, IIT Bombay, India; Jan Vitek, j.vitek@neu.edu, Northeastern University, USA.

2475-1421/2023/10-ART263
https://doi.org/10.1145/3622839

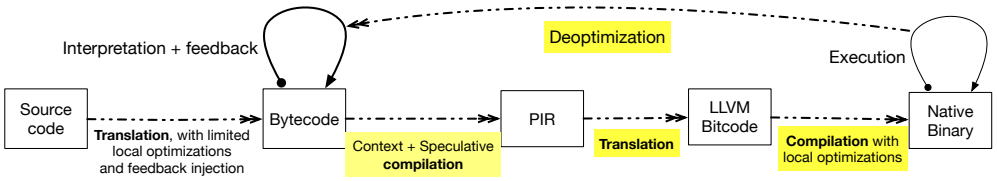

Fig. 1. Ř compiler pipeline

PIR, an SSA-like intermediate language, that is then translated into LLVM bitcode and handed off to the LLVM compiler infrastructure for native code generation. If speculation fails, deoptimization takes the execution back to the interpreter.

We address the duplicated effort involved in the yellow-shaded steps. While most programs use the same libraries in similar ways, Ř compiles all library functions from scratch, and likely the same deoptimizations and re-compilations. This repeated work impacts startup time. While our approach is designed for R and Ř, it is our belief it generalizes to other dynamic languages. We speculate on broader applicability and its limitations in Section 9. We leave the discussion of related work to the next section. A summary of our core challenge is, given two requests to compile function f, $compile(f) \rightarrow V_1 \ldots compile(f) \rightarrow V_2$, when is it correct to reuse the first binary V_1 instead of V_2?

Our approach. We extend a Ř with *speculative contextual dispatch* and an *off-line curated code repository* in order to decrease the frequency of compilation requests. Our approach is as follows. Each time a function is scheduled for compilation ($compile(f) \rightarrow V$), we compile the function and store the pair $\langle C, V \rangle$, where C encodes all compiler assumptions, in the repository. A single function may be associated with many such pairs. Subsequent calls to f query the repository for an applicable compiled code fragment V. To bound repository size, an off-line curation process deduplicates $\langle C, V \rangle$ and $\langle C', V' \rangle$ if the context C' is entailed by C and the code is similar.

In practice. The devil lies in the engineering details. Our implementation must answer a number of practical questions. *What is a context?* We extend previous work by Flückiger et al. [2020] to include the interpreter's speculative feedback information. *How to dispatch on contexts?* We perform a two-level dispatch that uses both the information available at the call site, and an inline cache of previously observed feedback. *What to store in the repository?* We store a portable intermediate representation and generate object files off-line. *How to curate the repository?* We remove redundant entries by comparing contexts and then tag each entry with representative contexts.

Expectations. What should we expect from the resulting system? We should be able to run all R programs as we have not restricted the semantics of the language, with a decrease in the number of compilation requests. Given that R programs can generate code on the fly, it is to be expected that the compiler will be called from time to time. Furthermore, Ř has some limitations that prevent compilation of a number of idioms, so interpretation overheads cannot be eliminated entirely. We expect to see a reduction in startup times as we will not have to perform the most expensive steps of compilation. We expect to be able to reuse code across different programs using the same libraries. Peak performance should be unchanged, provided the dispatch costs are not prohibitive. We expect the code repository to be large but not unreasonable in size.

Results. Our empirical results suggest that the approach presented can reduce Ř compile-times by 9× while retaining peak performance. We also observed that by re-using previously compiled versions of functions, it is possible to reduce the impact of phase change behavior in programs. Finally, curation reduces the size of code repositories by over 60%.

2 RELATED WORK

The R language. R is a rather dynamic language with challenging semantics for compiler writers [Flückiger et al. 2019]. It has a lazy functional core in which values are modified by copy-on-write and arguments are evaluated only when used. But that core is burdened with various ill-defined reflective APIs and semantic loopholes due to the presence of low-level C code with access to the interpreter internals. The Ř approach to context dispatch is described in [Flückiger et al. 2020]; our work extends that code base directly. Previous work did not include speculative information in contexts, we add those here. In more recent work, the Ř team proposed an approach that avoids deoptimization [Flückiger et al. 2022]; we do not use that version of the system, but believe integration with the approach we describe here is possible without a significant difference to our results. The reference implementation of R is referred to as *GNU R*. It includes a bytecode compiler with a small number of carefully tuned optimizations [Tierney 2019]. Other implementation approaches have been explored, from type-specializing interpreters [Kalibera et al. 2014; Wang et al. 2014], self-specializing interpreter running on top of just-in-time compiler [Stadler et al. 2016; Würthinger et al. 2013], and trace-based compilers [Talbot et al. 2012].

Ahead-of-time compilers. It is tempting to get rid of the just-in-time compiler altogether and compile R in a more traditional way. Wimmer et al. [2019] and Serrano [2021] have investigated how to achieve this in the context of Java and JavaScript, respectively. While the results are impressive, the dynamism of R adds many challenges that discouraged us from following that path. The combination of reflection, dynamic binding, and laziness is particularly vexing in this respect.

Caching compilation artifacts. One of the earliest works on optimizing dynamic languages was for SELF [Holzle 1994]. The authors rely on runtime profiling and type-feedback information to improve performance. They discuss the idea of using a code cache at runtime that keeps compiled code in memory, and uses a least recently used policy to evict code. Our work extends this general idea and moves the code cache outside of the runtime, making it reusable across different programs and executions. Reddi et al. [2007] reused compiled code with a persistent cache. Their work focuses on reducing cold start times and leaving the task of compiling specialized code to the runtime. Two Java compilers, Quicksilver [Joisha et al. 2001] and ShareJIT [Xu et al. 2018], proposed reusing binaries. They faced issues of cache invalidation when a method was redefined. In our case, invalidation is performed in combination with context dispatch and deoptimization. These systems did not deal with multiple versions of functions, nor did they have an explicit notion of context. Zhuykov and Sharygin [2017] proposed a JavaScript compiler that cached its intermediate representation. Their choice was pragmatic: patching native binaries is hard. We also cache intermediate representation but then post-process and compile a custom binary offline which is then easily patched at runtime, thus avoiding the extra last-step compilation. An alternate approach is to cache the feedback information instead of compiled code. Although this does not save compilation time, it is easy and can decrease the time spent in the interpreter as shown by Arnold et al. [2005]. The authors also addressed the question of what to do when different runs produce conflicting profiles. They chose to approximate safely, with suboptimal peak performance when programs were trained on bimodal inputs. Our approach does not merge profiles since we keep multiple versions and dispatch on contexts. On the other hand, our dispatch costs are higher. Ottoni and Liu [2021] followed a similar idea in the context of PHP. They focused on a use case where many servers run exactly the same program, so merging profiles was not as big an issue. One recent work that tries to reuse previously compiled code is JITServer [Khrabrov et al. 2022]. The authors offload the task of compilation to a remote server that also caches the compilation results. They maintain one generic version per function, which improves cold start times at the cost of degraded peak performance.

3 SYSTEM OVERVIEW

We refer to our experimental implementation as Ř+, to differentiate from baseline Ř. The baseline is a just-in-time compiler that features function specialization and speculation techniques to aggressively optimize R code. Function specialization is implemented using contextual dispatch [Flückiger et al. 2020]. Contextual dispatch separates classes of function behaviors based on their call site properties and runtime argument information. Additionally, it uses runtime feedback to predict future behavior and optimize for the same. In other words, call site predicates dictate which version of a function to pick, while guards inside the compiled code's body ensure that the speculative assumptions hold.

Ř+ extends Ř to persist and reuse compiled code across different executions. The changes to the existing code base are mostly in the compiler. In addition, we wrote an execution recorder and a loader. We added speculative contexts and introduce the concept of speculative contextual dispatch. Furthermore, we wrote a code repository along with support for off-line curation and optimized compilation. Fig. 2 illustrates use cases.

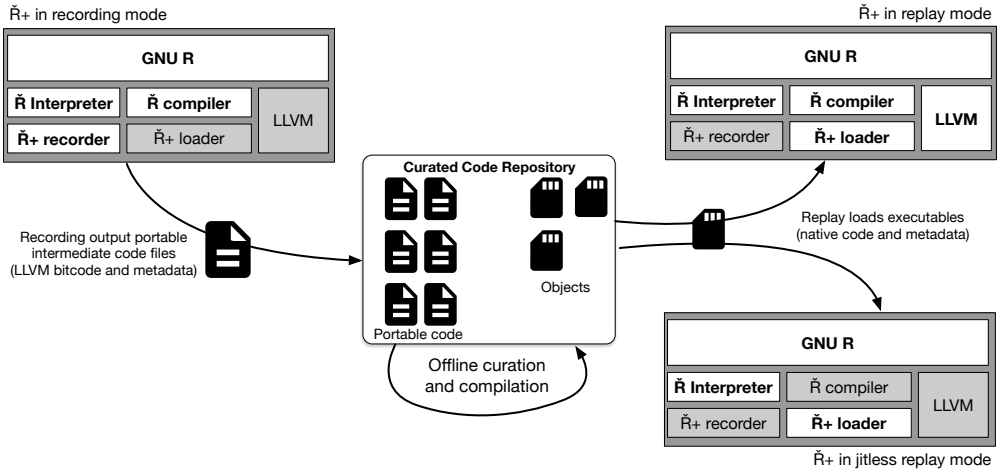

Fig. 2. Usage overview

There are three main use cases for Ř+ (with some additional variations):

- **Recording:** During recording mode, execution is mostly in the interpreter; each time a compilation is triggered, we let Ř do its work all the way until LLVM bitcode, then we create a portable intermediate code file with metadata and bitcode and send it to the repository.
- **Replay:** In this mode, when Ř encounters a function, we query the repository; if a match is found for the current context, we load and patch the native file and install a dedicated dispatcher for that function. When the repository does not have a matching function or when we deoptimize, the Ř compiler is triggered.
- **Jitless Replay:** Same as replay, except the compiler is turned off and we default to the interpreter if needed.

When recording, one can leave the compiler enabled, thus benefiting from the compiled code. One can also combine replay and recording modes, though we have not explored that configuration. One last option would be to package together relevant chunks of the repository with libraries, so that when downloading a library through a package manager, users would already have commonly used compiled code for it.

The repository combines portable intermediate code files produced by one or more recording runs. It performs an off-line curation pass that gets rid of redundant files. The remaining intermediate code is then compiled to native code using the LLVM compiler at the highest optimization level. Once code generation is complete, the repository can also determine the structure of the dispatch code for each function. While the intermediate code files are hardware independent, the curated repository has code specialized to the current hardware platform. This means that recording can be done anywhere, and combined into a single highly-optimized repository.

4 A MOTIVATING EXAMPLE

To illustrate the workings of Ř+, let us take an example in a simplified language where variables hold either integers or strings and addition is polymorphic. Fig. 3 shows a function that increments a global variable and prints the result. The function succeeds for any combination of types as all operations are generic. Fig. 4 is a client script that invokes the function and sets the global variable. Fig. 5 gives pseudo-code for one compiled version of the function.

```
function f(a) {
  r1 = global
  r2 = addAny(a,r1)
  r3 = printAny(r2)
}
```

Fig. 3. Sample function

```
global = 42
f(1)
f(2)
global = "c"
f("a")
f("b")
```

Fig. 4. Client script

```
V1 (int):
  r1 = global
  if !(r1 is int) deopt()
  r3 = addAny
  if r3 != &12435 deopt()
  r2 = addInt(a,r1)
  printInt(r2)
```

Fig. 5. A compiled version

The compiled code is specialized for an integer argument. The compiler also speculated that the type of the global variable is a number. The code first checks if that speculation is correct by testing the type of the global. If this is not, it triggers deoptimization and falls back to interpreter. It then checks that the addition function was not redefined by comparing the function's memory address to a cached value. If this second guard succeeds, a type-specific versions of addition and printing can be called. For brevity, we left out the address comparison for `printAny`.

Fig. 6 shows three additional versions. The first (V2) is compiled under a context that accepts strings and numbers, and speculates that the global is a string. The second (V3) expects strings in the argument and speculates that the global is an integer. The last one (V4) makes no assumptions.

```
V2 (string|int):
  r1 = global
  if !(r1 is string) deopt()
  r4 = toString(a)
  r3 = add
  if r3 != &12435 deopt()
  r2 = addString(r4,r1)
  printString(r2)
```

```
V3 (string):
  r1 = global
  if !(r1 is int) deopt()
  r4 = toString(r1)
  r3 = add
  if r3 != &12435 deopt()
  r2 = addString(a,r4)
  printString(r2)
```

```
V4 (string|int):
  r1 = global
  r2 = addAny(a,r1)
  printAny(r2)
```

Fig. 6. Additional compiled versions of Fig. 3

There is a partial ordering between versions in terms of applicability. Let us write $(T)[T]$ to denote the combination of argument and global variable types, a version is *applicable for any particular pair* if the predicate on the argument holds. The predicate on the global variable does not have to hold, the function is allowed to trigger deoptimization when the guard fails. For instance, V1 is only applicable to `(int)[int]`, V2 is applicable to `(int)[string]` and `(string)[string]`, and V3 to `(string)[int]`. Lastly, V4 is applicable to all contexts. An implementation is free to pick any applicable version, the difference between applicable versions is their non-functional characteristics, i.e. speed and memory consumption.

Ř compiles versions on the fly, alternating between running code in an interpreter that records information about each program and running compiled code intended to run efficiently. Let's consider a hypothetical run of such a compiler with a compilation threshold set to one, i.e. each method is run once in the interpreter before being compiled. Fig. 7 has an example script that calls the above function in various contexts.

step	1	2	3	4	5	6	7
client	`global=42`	`f(1)`	`f(2)`	`f("a")`	`f("b")`	`global="d"`	`f("d")`
exec		B	O	B	O		O/B
comp/deopt			C		C		D
record		`(I)[I]`		`(S)[I]`			`(S)[I|S]`

Fig. 7. Sample just-in-time compiler run

The *exec* row indicates if the function was executed in the **B**ytecode interpreter or as an **O**bject file. The *comp/deopt* row indicates when **C**ompilation or **D**eoptimization occurs. The *record* row shows what was recorded by the interpreter for that call. For our purposes, we assume the interpreter records types of arguments and local variables. So, at step 2, the call to `f(1)` records that both argument and global were integers. At step 3, the compiler generates V1 using the recorded context.

Ř performs contextual dispatch. At step 4, we have a context `(string)`, but we only have V1 compiled under context `(int)`. As there is no applicable method, we return to the interpreter and execute the method to gather more information. At Step 5, Ř compiles `f` under the context `(string)[int]`, generating V3.

At step 6, the global is changed to a string. In the final call to `f`, context dispatch finds V3. Executing V3 leads to a deoptimization as the global does not have the expected type. The next compilation for a similar call will have `(string)[int|string]` as a context and could generate a generic version like V4.

The above example allows us to observe:

- For a given context, we retain only the most generic function. So, eventually, V4 will replace V2 for the context `(string|int)`. This suggests that performance may degrade over time.
- What we record about globals is 'sticky'. Once `global` has been observed having either a string or a number, that information is used for all future compilations. This avoids compilation-deoptimization loops, but it degrades code quality.
- Contextual dispatch may require multiple compilations for the same context to reach a steady state due to changes in the recorded argument information.

Improving on the motivating example. Our approach allows Ř+ to record multiple client scripts. Each time a function is compiled, the version along with its full context `(T)[T]` is added to the repository. Imagine that our motivating example was part of a widely used library, in that case, the repository would be populated with versions and contexts obtained by recording all client

programs that import the library. Fig. 8 shows the possible contents of a code repository for the function `f` above. The repository is a persistent database where function versions are indexed by their entire context, i.e. both the type of argument and speculative information recorded by the interpreter.

full context	version
`(I)[I]`	V1
`(S)[I]`	V3
`(S|I)[S]`	V2
`(S|I)[S]`	V4
`(S|I)[S|I]`	V4

Fig. 8. Sample repository for the above function

Observe that the last two rows have different keys, but the same version. This can happen because some of the information provided by context does not unlock additional optimizations and thus the compiler ends up generating the same code. Consider a function that has an unused parameter, our approach could lead to generating versions for all the observed types of the unused argument at the different call sites, but the generated code of these versions would be identical. For this reason, the repository will be curated, and identical versions will be deduplicated.

The benefit of having a large number of versions is that it greatly increases our chances to locate the most efficient code for each function and context.

This example simplified our approach for didactic purposes, we now present the approach in its full generality and discuss engineering issues.

5 SPECULATIVE CONTEXTUAL DISPATCH

Speculative contextual dispatch is an implementation technique that picks one among multiple versions of a function compiled under different assumptions about their environment. For a given call to a function f compiled to versions $V_1 \dots V_n$, dispatch is *correct* if it picks a V_i that either executes successfully or deoptimizes, and is *efficient* if V_i is the best version according to some chosen performance metric. The difference with previous work is that we enhance context with speculative information. A *speculative context* is thus a pair C, F where C is the argument context following Flückiger et al. [2020], and F is the feedback information recorded by the interpreter for previous runs of f (e.g., `(Int)[String]` in the motivating example). Contexts are chosen so that there is an efficiently computable partial order such that $C_1, F_1 < C_2, F_2$ means that C_2 entails C_1 and F_2 entails F_1. Intuitively, think of contexts as predicates on program state. For instance, `(Int)[Int]` states that the function's argument and the global variable hold an integer. An example of ordering is `(Int)[Int] < (Int|String)[Int]`; the program state where the argument is either an integer or a string entails the one where it can only be an integer. The element $\top$ denotes the largest possible context which is also the least informative one.

The difference between the two components of a speculative context is that C is a predicate that is not allowed to be invalidated by program actions, whereas F may not hold for the whole function body. When emitting code for a function f under context C, F, a compiler may assume C to be true, but, any time it uses F, it needs to emit code to check that F holds. We thus model compilation as:

$$compile(f, C, F) = V, D$$

where D is the subset of F that occurs as a guard for deoptimization. Any call of f that dispatches to V is correct if C holds at the call site, it will succeed if D holds when checked in the body and safely

deoptimize otherwise. At a call site, a dispatcher can compute C by querying the arguments and can approximate F by accessing a cached value observed by the interpreter when it last executed f. Assume that there is a set of compiled versions $\langle f, C_1, D_1, V_1\rangle \ldots \langle f, C_n, D_n, V_n\rangle$. For correctness, the dispatcher only needs to pick C_i such that $C < C_i$. In practice, there are likely to be multiple versions that are correct. For efficiency, one should pick the most optimized version that will execute without deoptimizing. We do not aim for an optimal choice. A reasonable heuristic is to pick one of the smallest Cs such that $F < D$. We assume the presence of a $\top$ context that leads to execution in the interpreter.

5.1 Example

Consider a variation of our running example with two global variables. Fig. 9 and Fig. 10 show, respectively, the function definition and a client script. In this case, the function would run once in the interpreter with the following context `(int)[int,string]`. The compiler may choose to generate `V1` where it only speculates on the first global variable being an integer as shown in Fig. 11.

```
function g(a) {
  r1 = global1
  r2 = global2
  r3 = addAny(a,r1)
  printAny(r2)
  printAny(r3)
}
```

Fig. 9. Sample function

```
global1 = 42
global2 = "hi ma"
g(1)
g(2)
g(3)
```

Fig. 10. Client script

```
V1 (int):
  r1 = global1
  if !(r1 is int) deopt()
  r2 = global2
  r3 = addAny
  if r3 != &12435 deopt()
  r4 = addInt(a,r1)
  printAny(r2)
  printInt(r4)
```

Fig. 11. A compiled version

Here, the second call to `g` compiles to `V1` and the last call performs contextual dispatch. At that point, inspecting the type of the argument gives us C = `(int)` and F = `[int,string]` as observed by the interpreter when it evaluated `g(1)`. As the compiler only used part of the feedback, the D of `V1` is `[int, ⊤]`. Dispatch will pick the compiled method as the Cs are the same and $F < D$.

5.2 Implementation

Our implementation extends the Ř dispatcher. Contexts (Cs) include more information than just types. They also have strictness annotations, argument ordering information and bits to denote missing arguments. We refer to the aforementioned paper for a detailed description. When computing the call-site context, the compiler can leverage any statically known information to reduce the run-time work required. The actual dispatch is performed by a function call to a dispatcher that has access to a pointer to the GNU R function object which holds an array of compiled versions.

Our extension adds an inline cache that has a pointer to an extended dispatcher object. To minimize lookups of feedback vectors (Fs), which can be sizable and include hundreds of entries, particularly for large functions that belong to different code objects due to inlined functions, we cache the last version dispatched to and invalidate the cache when we deoptimize or F changes. Another optimization is to only look at the entries in F that are actually used by the target function (i.e. D). Fig. 12 shows the C++ code of the extended dispatcher.

```
1  Function*
2  Dispatcher::dispatch() {
3    if (cache != NULL) return cache;
4    for (int i = length() - 1; i >= 0; i--) {
5      auto f = getFunction(i);
6      if (f->abled() && f->matchD())
7        return cache = f;
8    }
9    return cache = getFallback();
10 }
```

Fig. 12. Pseudo-code for the speculative dispatcher

6 CURATED CODE REPOSITORY

The Ř+ code repository gathers and curates optimized code for re-use. Fig. 13 gives an overview of the recording architecture. Each time a function is compiled, the recorder will perform a number of tasks to store this compiled code in a portable format. Code sent to the repository is deduplicated and further optimized. The recorder takes the optimized LLVM bitcode generated by Ř as input and performs three operations: patching, serialization, and linearization. These operations generate a portable file consisting of code and metadata which is saved to disk. Curation is executed off-line, it is a stage that reads in the saved code, filters out redundant versions and then compiles the bitcode to native using LLVM at its highest optimization level.

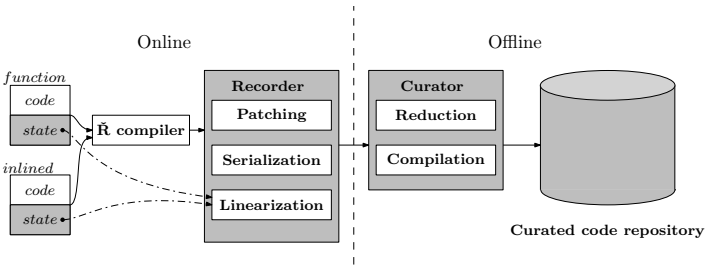

Fig. 13. Overview off-line curation of code repository

6.1 Recording

To store compiled code in a portable format requires removing any dependencies to concrete surroundings of the code. Such dependencies include things like memory addresses of other functions and of literal constants. This is done by the patching step that finds these dependencies and replaces them with external symbols and references to a serialized object pool. In Ř, calls to known functions are replaced with optimized call sequences. To reuse such codes requires recording information in a metadata file, this includes the function name, its signature and call-site context. The recorder additionally identifies deoptimization points and stores them in the metadata.

The disk representation of the data produced by the recorder consists of three files:

- **.bc**: LLVM bitcode output by Ř.
- **.pool**: the object pool with constant data referenced by the function.
- **.meta**: the metadata needed for dispatch.

The metadata contains the following:

- Fid: a unique identifier for the function.
- C: the context under which the function was compiled.
- F: speculative context (including that inlined functions).
- D: deoptimization points (points in the code where the speculation may fail).
- deps: Fid s of function inlined in this one.
- sigs: arguments to load from stack, optimized call-convention metadata, ordering of code fragments and other function flags.

The Fid is a 64-bit hash of the function body and its enclosing namespace. For anonymous and inner functions, we hash the parent function with the offset of the inner function in the parent's code. In case of a collision, we blacklist the involved functions; they are not included in the repository and will be executed without the benefits of our Ř+. For technical reasons, some inner functions are compiled before their enclosing function. They are also black listed.

The context C is inherited from previous work. The speculative context F contains feedback information. As Ř may have inlined other functions in this one, and may have added speculative code into inlined code. Consider some function f_1 compiled to $V1$ which inlines functions f_2 and f_3, we write $[f_1^d, f_2^d, f_3^d]$; where f_1^d is the feedback of the main function and the other two belong to the inlined ones. In the implementation, this is a vector of vectors, where each inner vector contains a f_x^d corresponding to its respective Fid_x.

The deps are the set of inlined function Fids in the compiled code. This list is maintained separately for convenience.

The sigs are compilation attributes of a version needed by Ř. This includes general information such as the number of arguments it expects on the stack, and the argument-reordering information that is needed if a call from the compiled code ends up in the GNU R interpreter. A compiled code object is composed of smaller fragments that hold things such as promises. These are expected to appear in a certain order inside the compiled code object; we store this ordering information explicitly in the metadata.

For our recording stage, we chose to store the compiled functions in the form of LLVM bitcode, which is the last IR Ř generates before they are turned into native executables. This offers some advantages. Firstly, patching bitcodes for reuse is easier than doing the same with machine code. Secondly, it makes the repository portable across architectures.

6.1.1 Patching. In Ř, compiled code can contain different kinds of references to its enclosing environment. To make the code reusable, we patch it as follows:

- **Globals**: addresses of global variables become lookups via a table of global addresses.
- **Indexed References**: references to the the various constant pools of GNU R are turned into table lookups.
- **Builtins**: calls to built-in functions become lookups in a function linkage table.
- **R Code**: references to other functions become lookups using the target function's Fid.
- **R Language Objects**: references to source code objects with mutable attributes are turned into indirect accesses; we ensure that any sharing is preserved.
- **R Objects**: references to R objects such as strings and vectors become lookups into the serialized object pool.
- **Deoptimization points**: references to a struct with a program counter and instructions for rebuilding stack frames are patched to a Fid-relative offset in the serialized object pool.

This step produces LLVM IR code that is free from any memory references, but contains lookups that need to be backpatched.

6.1.2 Serialization. The serialization stage creates an object pool with all the values referenced from the compiled code. General R objects and deoptimization metadata are directly added to this pool and the references in the LLVM bitcode are patched to reflect the index of the serialized values. We avoid serializing R code directly, instead, we patch references to code objects to make them Fid-relative. Once all the references have been patched, the pool is serialized to the disk (as **.pool* file). We also run an O2 pass over the patched LLVM bitcode and serialize it to disk as bitcode **.bc* file.

6.1.3 Linearization. When the Ř interpreter records speculative feedback, this feedback is stored as R objects within the bytecode of the function. To store this information in the repository, we traverse the basic blocks of the function depth-first and store feedback into a vector. Fig. 14 illustrates that this information is more than just types of variables, there are also branches taken and functions called.

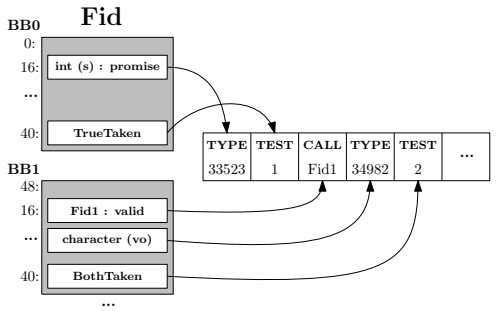

Fig. 14. Feedback information linearization

In Ř, the feedback for types and branches are, fixed size, 32-bit integers. Hence, this feedback can be treated as numeric values which simplifies the task of storing and comparing them. Function feedback has the addresses of the last three observed callees. If more are observed, the slot is marked as invalid and is not used by Ř. We resolve addresses to their corresponding Fid and store these instead. Each entry is tagged to indicate the kind of feedback stored within it. Linearization is performed for all the functions being considered, including inlined functions.

Once all the relevant data has been collected, the recorder generates a metadata file. This file contains the Fid, call-site context, speculative context (i.e. linearized function states), deoptimization information, a list of inlined functions, and the function signature.

6.2 Curation

Before using a repository for replay, we remove redundant versions of functions, as well as prepare the dispatchers, and compile LLVM bitcode to object code. This can be done off-line; during this process one can safely use the previous version of the repository.

The main reason for curation is to limit the size of the repository. Ideally, one would keep only the versions of a function that provide significant performance improvements. We do not have a principled way to decide which these are. Instead, we take a simple criterion that works well enough in practice. Given two versions, $\langle \mathsf{Fid}, C, D, V \rangle$ and $\langle \mathsf{Fid}, C', D', V' \rangle$, we delete the first version if $C' < C$ and $V = V'$. In other words, if the bitcode is the same, we pick the version that is applicable to more contexts. We experimented with more aggressive heuristics, but they resulted in discarded versions being compiled again and inserted back into the repository, leading to compilation-deprecation-compilation loops.

To build the dispatchers, there are choices as the comparison can safely be approximated. One approach would be to compare the entire F available at the call-site with the ones used to compile the function. Given the size of contexts, some have thousands of entries, which may slow down dispatch. A previous version of this paper tried to heuristically pick the most important bits of F, but this led to pathologies. Instead, we identify which parts of F are referenced from the deoptimization points in V, this is how we build D. After the identification of D, we are left with a smaller, albeit not minuscule number of comparisons to be done at runtime. Luckily, our dispatcher has a cache for its fast case that avoids doing these comparisons most of the time.

6.3 Replay

To replay, Ř+ starts by loading metadata for all functions in the repository and builds a dictionary of versions indexed by the Fid of functions. Whenever we are about to compile a function, we check if its Fid appears in the dictionary. If a version exists, Ř+ tries to load it and all function versions that it depends on as recorded in the metadata. The loading process thus recursively loads all the transitively needed versions. Before a version is loaded into memory, Ř+ deserializes its object pool so that code can be backpatched. However, we delay patching the binary until the first call to the function, this ensures that we do not end up spending time patching binaries that are not needed. When loading is completed, we construct a dispatcher for that version as shown in Fig. 15. This new dispatcher is added into the context dispatcher inherited from Ř.

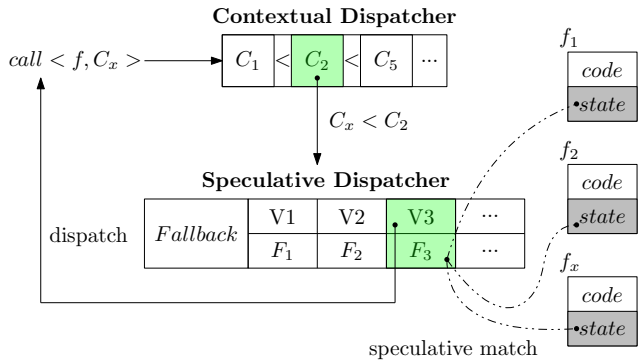

Fig. 15. Overview of a dispatcher

The dispatcher consists of a list of (i) pointers to versions, (ii) pointers to feedback information, and (iii) a fallback function. The list can grow if new versions are loaded due to calls or dependencies. The fallback function is initially empty. It can store newly compiled code if a new run-time context is observed for the function.

We now describe the main operations performed on dispatchers:

(1) *Create.* A new dispatcher is added to a slot of the legacy Ř dispatch table. If the slot already contains a pointer to code; then we replace that pointer with the new dispatcher and set the fallback to the old code, otherwise we set the fallback to the interpreter.
(2) *Insert.* Add newly linked deserialized binaries into the function list.
(3) *Dispatch.* Iterate over the function list and dispatch to the first matched binary that is enabled.
(4) *Deoptimization.* In case a version deoptimizes, the corresponding entry in the dispatch table is disabled.

7 BENEFITS OF THE PROPOSED APPROACH

Ř+ offers various advantages over a traditional just-in-time compiler in various real-world scenarios.

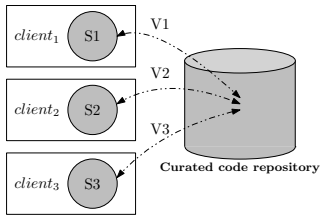

Fig. 16. Specialized dispatch

Consider Fig. 16, three different clients use the same function in different contexts. Think of a statistics library that is used by different research communities to analyze different kinds of data. Here S1, S2 and S3 are the different runtime feedback information for each use of the function. Our speculative dispatcher can pick optimized versions based on the use case of each individual client. In case a, previously unseen, specialization is compiled, Ř+ will add it to the cache. As time goes on, specializations gathered from different use-cases will be gathered in the repository and, after curation, the ones that lead to better performance will be available for future clients.

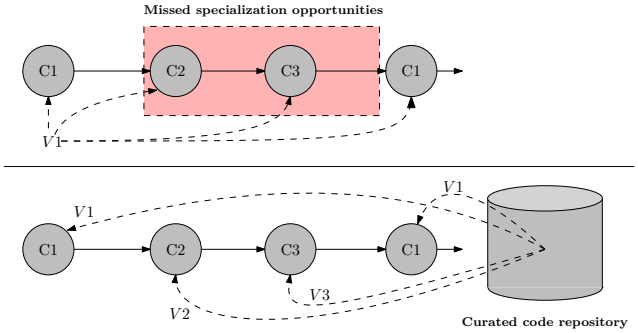

Fig. 17. Missed specialization opportunities due to early generalization

A just-in-time compiler has to limit the time it spends compiling. In Fig. 17, a function is called in different contexts. The runtime compiles it under a rather generic context C1 first. Then context C2 and C3 will not be specialized for, as C1 is applicable. For the compiler, using V1 compared to plain interpretation is good enough, it does not know if C2 or C3 will bring benefits sufficient to amortize compilation costs. This means the system may end up stuck running a sub-optimal version. Our approach has the benefit that different programs may call a library function in different contexts, so over time, more contexts can end up in the repository. Furthermore, compilation costs can be amortized over many executions.

At a glance, one may think that Ř+ will perform no better than a long-running instance of Ř. Let us consider a frequently used function in a long-running virtual machine executing several client programs, shown in Fig. 18. The first client program uses the function and compiles it under S1 to V1. Later, the second client uses the same function in a new context. The old compilation deoptimizes. A more general version, V2, is generated to handle both contexts. As we keep adding clients, the function state mutates and the subsequent compilations become even more generic

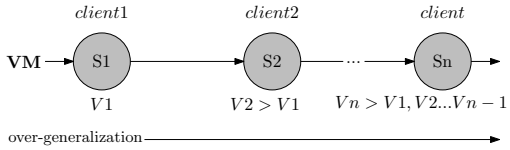

Fig. 18. Over generalization in a long-running Ř instance

and very likely less performant. This means that if we run the first client again, it may not run as fast as it ran the first time around. Generalization helps in preventing deoptimization loops at the cost of peak performance. Our approach retains all versions and thus increases the chances a more optimized version is available.

8 EMPIRICAL EVALUATION

The goals of this section are to evaluate the benefits and costs of reusing previously compiled code. In particular, we want to show that we can significantly reduce warmup time without impacting peak performance. We evaluate this by comparing the performance profiles of programs with and without our scheme, focusing particularly on the time spent on just-in-time compilation. We aim to validate the following questions:

Q1. Does Ř+ reduce warmup times?
Q2. Does curation remove redundant binaries?
Q3. Does Ř+ preserve peak performance?
Q4. How effective is Ř+ in handling deoptimization and over-generalization?

8.1 Experimental Setup

For our experiments, we use benchmarks bundled with Ř and three real-world programs. **RMarkdown** is a document processing application that imports many R libraries for plotting and data analysis [Love et al. 2013]. It is not computationally intensive but it does stress the compiler by loading substantial amounts of code. **Raytracer** is a computational benchmark with significant phase changes [Morgan 2008]. Lastly, we use an R library from the CRAN repository, **Recommenderlab** [Hahsler 2022], for which we run its example code.

Our experiments are run on a dedicated benchmark machine, with all background tasks disabled. The machine features an Intel i7-6700K CPU, stepping 3, microcode 0xea with 4 cores and 8 threads, 32 GB of RAM and Ubuntu 18.04 on a 4.15.0-151 Linux kernel. Experiments are built as Ubuntu 20.04.1 based containers, and executed on the Docker runtime 20.10.7, build f0df350.

8.2 Warmup and Peak Performance

How are warmup times (Q1) and peak performance (Q3) affected by Ř+? Fig. 19 shows the wall-clock time (in seconds) of 20 iterations of 15 benchmark programs for Ř+ with a pre-built repository and Ř. Ř+ is run with the just-in-time compiler enabled as a backup. Higher is worse.

Each benchmark uses a dedicated code repository built by running the benchmark five times while in record/replay mode. This is a best-case scenario for our approach as we run a program exactly in the same environment as the one used for recording.

Overall, Ř+ reduces warmup time and preserves peak performance. This can be observed by looking at the wall-clock time of the first iterations of each program. Only few compilations are triggered, and replay is able to link the binaries on demand. In Ř+, one interpreter execution is forced so that minimal feedback is collected for later dispatch. Varying on the code structure, some programs will get linked to binaries earlier than others. In more detail:

(1) *Zero warmup*: In several of the programs there is no observable warmup. After an initial interpreter run, they get quickly linked to the stored binaries. This happens already at iteration 1. (binarytrees, bounce, knucleotide, nbody, pidigits, regetdna).
(2) *One iteration warmup*: Some programs might have one main function which features a time-intensive loop. These will get linked to their binaries only after the first interpreter iteration. This shows up as an initial slow iteration. (convultion, fannkuchredux, fasta, fastredux, flexcust, mandelbrot, reversecomplement, spectralnorm, volcano).
(3) *No recompilation pauses*: The occasional late-stage compilation pauses present in Ř in many of the programs are completely avoided in Ř+.
(4) *Faster peak performance*: volcano has Ř+ running significantly faster than Ř at peak. This is meaningful as this is one of the most realistic of the set of benchmarks. The explanation is that, during the training phase, we have compiled a better-optimized version under a context that was not reached by Ř.
(5) *Almost zero compilation*: Only in one case, flexclust, we see two compilations in Ř+. These correspond to contexts that might have not emerged in the training phase.
(6) *Peak performance is preserved*: All the benchmarks show that the peak of Ř is preserved Ř+.

Table. 1 gives detailed statistics about the benchmark runs. The **Peak** column shows peak performance of the two systems. Peak performance is mostly stable, with a worst-case slowdown of 7% for pidigits and a 9% improvement for volcano.

For each benchmark, Table. 1 column **JIT overhead** tabulates to time spent in the compiler (both Ř and LLVM) in milliseconds. The Ř+ times are clearly much smaller, on average the difference in compile times is 9x in favor of Ř+, but not zero. To explain this, columns **#compiles** and **#deopt compiles** indicate, respectively, how many times the compiler was triggered. The first column indicates that the compilation was initiated due to missing contexts and due to deoptimization. On the other hand, deopt compilations emerge when the runtime is stuck interpreting a hot loop, in an effort to exit quickly, the runtime temporarily compiles just a part of that function. Both the columns show that Ř+ does not compile in most cases, while deopt does trigger the occasional compilation. One benchmark stands apart, in flexclust, Ř has to deopt compile 28 times. This is

Table 1. Statistics

Benchmark	JIT overhead (ms)		# compiles		# deopt compiles		Peak (ms)		Offline Curation	
	Ř	Ř+	Ř	Ř+	Ř	Ř+	Ř	Ř+	Time (ms)	Reduc. (%)
binarytrees	5049	13	24	0	1	0	78	79	2120	2
bounce	1265	3	11	0	1	0	133	136	640	0
convolution	797	76	8	0	2	1	12	13	460	0
fannkuchredux	1237	163	8	0	2	1	291	291	550	0
fasta	1966	122	15	0	3	1	422	441	840	0
fastaredux	3015	234	16	0	4	1	431	441	1100	0
flexclust	29701	11073	119	2	28	27	11830	11993	9327	20
knucleotide	6856	84	42	0	3	1	460	429	2365	13
mandelbrot	1328	3	17	0	2	0	62	58	745	0
nbody	7274	19	19	0	2	1	130	133	1805	9
pidigits	50871	85	76	0	3	0	64	69	27807	22
regexdna	1822	5	17	0	1	0	1359	1355	965	0
reversecomplement	1879	223	14	0	2	1	333	339	820	0
spectralnorm	1684	138	12	0	3	2	42	42	925	0
volcano	8588	695	23	0	3	1	22438	20602	2760	5

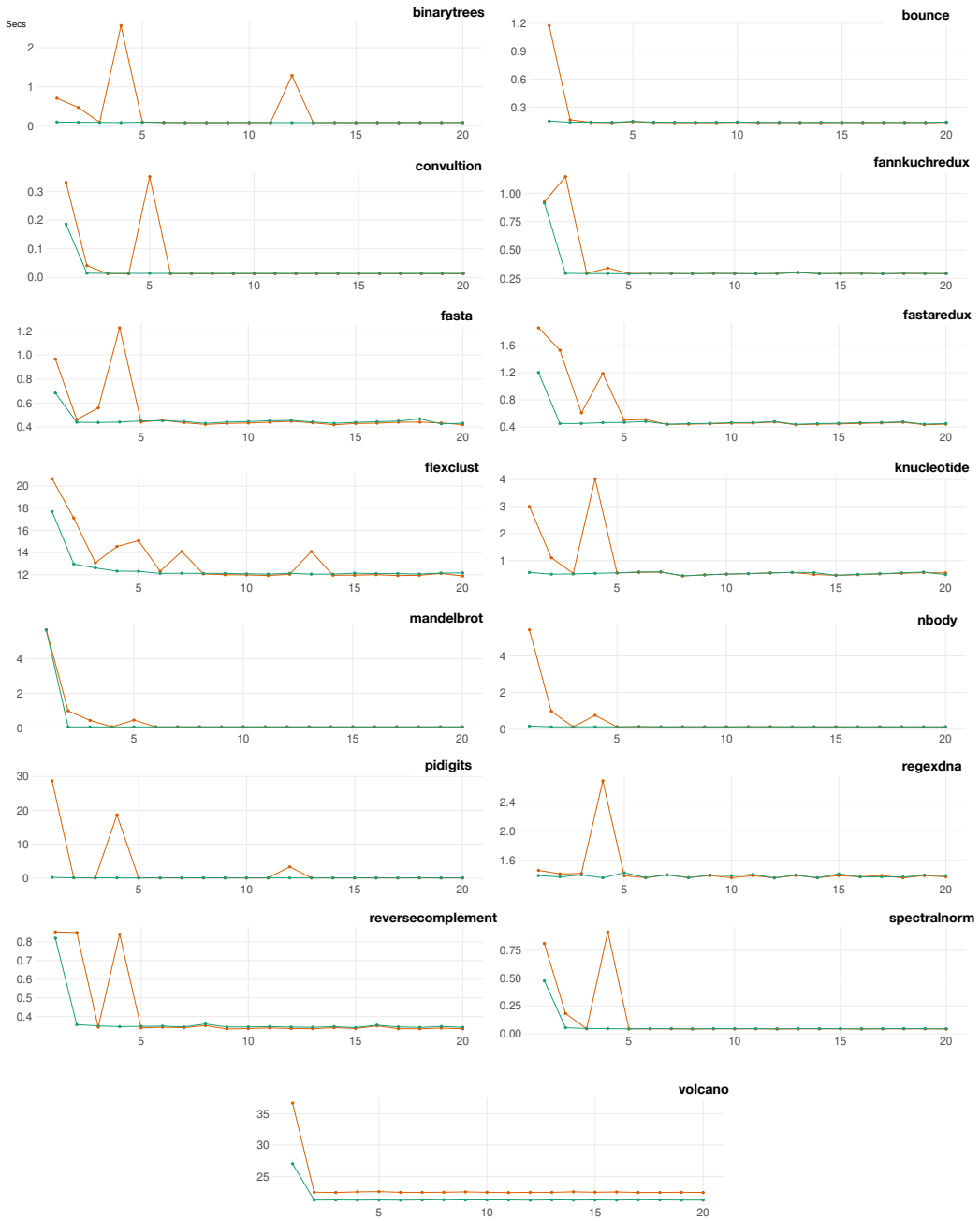

Fig. 19. Iteration times for Ř+ (green) and Ř (red)

due to a limitation in Ř, where we are repeatedly forced to deopt compile a part of a function that cannot be fully compiled due to a lack of support. We inherit that blemish in Ř+. Table. 1 also gives the time spent processing the repository and the percentage reduction in a number of object files due to curation. For these benchmarks, the savings are modest because they tend to have few redundant contexts.

8.3 Real-World Performance

The previous experiment addressed the performance of benchmarks. This experiment looks at a real-world application, RMarkdown. We can provide additional data points for Q(1, 2, 3). In this experiment, the repository is built with five runs of RMarkdown in record mode. This yields 3,140 versions which are curated down to 1,189 versions. Fig. 20 shows the wall-clock time for iterations of RMarkdown under several configurations: Ř with interpreter only, Ř with compiler, Ř+ with compiler, and Ř+ jitless. RMarkdown is an I/O intensive application with little computation in R. The interpreter is quite fast, a common scenario where we can only alleviate warmup costs that are not amortized later on. Ř incurs a massive 200 seconds pause and takes eight iterations to reach a steady state. Ř+ has better warmup times and reaches a steady state faster. Ř+ in jitless mode is almost matching the interpreter. The end-to-end time for all iterations has Ř take 405 seconds, Ř+ 150 seconds, Ř+ jitless 85 seconds and the interpreter 85 seconds.

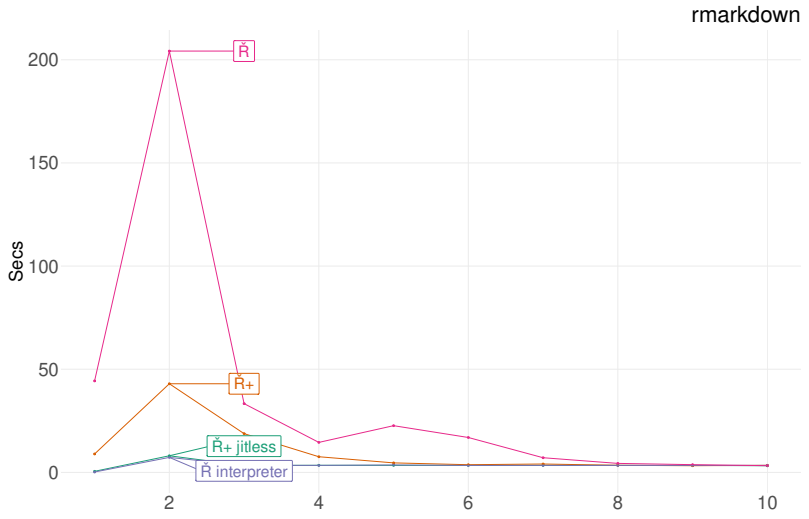

Fig. 20. Real-world performance

8.4 End-to-End Performance

Our previous experiments were best cases scenarios in which the code being run is identical to the code observed when recording. This experiment explores the impact of version skew.

Consider the case where one would like to test an R package. Each version of the package is likely to have only a few new functions, yet a just-in-time compiler will recompile everything for each test. With Ř, we have observed that some packages take tens of minutes to run through their test suite because tests need to run in a fresh virtual machine. This means the whole package and libraries is recompiled many times.

With Ř+, one would hope that once the first version of the package has been tested, the code repository will accumulate code for the unchanged parts of the package and only the new, or modified, code will be compiled when subsequent versions are checked out.

This experiment is set up to evaluate this use case. We took the Recommenderlab package from its Git repository at six different points over two years, thus obtaining different but related code bases. The code repository was built by recording the first checkout. All other versions of the package were run using that original repository. We expect that, as the code bases slowly diverges, the benefits of our approach may decrease. Fig. 21 shows the wall-clock time for one run of the test suite (red for Ř and green for Ř+).

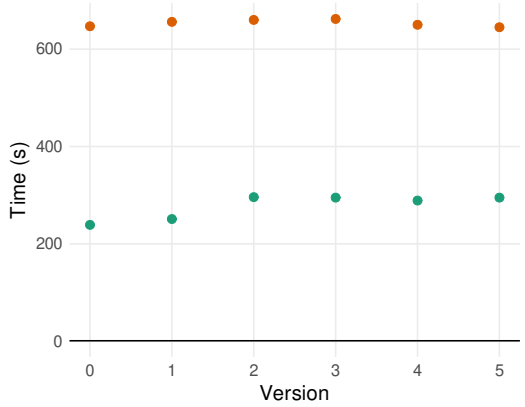

Fig. 21. Version skew (Ř in red, Ř+ with compiler in green)

The results are encouraging. While Ř+ still performs some compilation on every run, it reduces the overhead over Ř by almost a third. Over time, as the code of each release diverges, Ř+ spends slightly more time compiling. (Each iteration is based on the original repository, the overhead would be smaller if we updated the repository for each version.)

Broadly speaking, we expect end-to-end performance improvements in the following scenarios. (*i*) Large code bases where computationally intensive functions have multiple contexts. Furthermore, in case a just-in-time compiler does not have enough time to compile all the contexts, the missed specializations may be retrieved from our repository. (*ii*) For scripts composed of small functions only invoked a few times, the compiler hits a worst-case scenario where any time spent compiling is unlikely to be amortized, our approach can amortize compilation over multiple executions of programs.

8.5 Repository Construction

How should we construct a code repository? Ideally, we envision to gradually and incrementally add new versions from runs with different inputs. For libraries, one can populate the repository from client code. In our experiments, we have observed that new versions are found even when running the same program with the same input repeatedly. The reason for this is that Ř+ is sensitive to small changes in execution patterns. Each iteration of record/replay can cause the interpreter to observe different feedback and thus trigger different compilation.

Fig. 22 shows the iterations of record/replay. Between each iteration, we rebuild the repository so that its contents are used in the next iteration. The bars show the number of contexts that were added to the repository. This number includes the functions that were compiled for inlining.

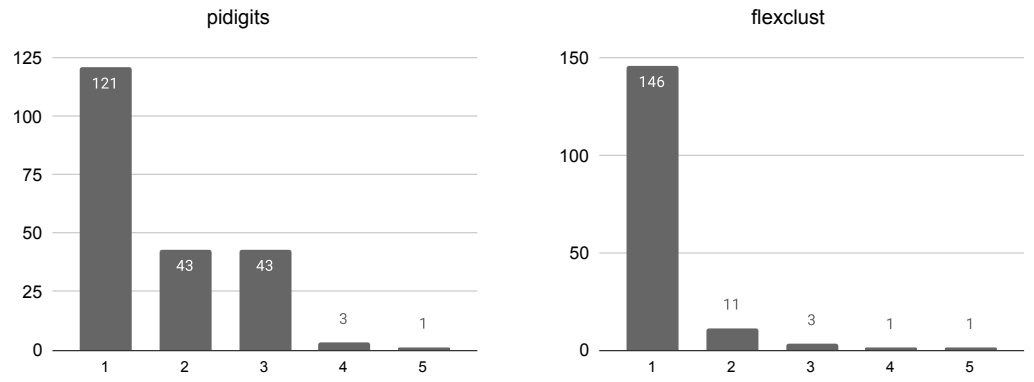

Fig. 22. New contexts under iterative feedback

We found the following reasons for the emergence of the new contexts after the first iteration:

(1) As the repository has optimized versions for the most popular contexts, the runtime which was previously dominated by interpretation is now spent in optimized code. This gives the less frequent, previously hidden contexts, a chance to emerge, leading to new compilations and new contexts.
(2) When we introduce compiled code early in the run, and the function state changes causing deoptimization, a new previously unseen state for a function is created. This new state when compiled, leads to a previously unseen context.

We picked five iterations as the default setting for the recording phase as we found that this was sufficient to collect the majority of the contexts. We also observed that for smaller programs where the number of function states is usually limited, one serializer run was enough to capture all the contexts.

8.6 Phase Change Behaviour

To explore how our scheme deals with deoptimization and over-generalization (Q4) we experiment with programs with phase changes. A phase change happens when a program, after repeatedly calling a function in a given context, shifts to a different context. This can be due to, for example, a change in the type of the data being processed.

In a traditional just-in-time compiler such as Ř, phase change triggers a deoptimization event followed by recompilation under a more general context, one that reflects the union of the new feedback with the previous one. Such recompilations cause slowdowns in the runtime and degrades the steady-state performance.

Fig. 23 compares Ř+ (with just-in-time compiler enabled) and Ř when running Raytracer. The graph is normalized to Ř and shows the speedup that Ř+ exhibits. In the third and fourth iterations (yellow) the speedup is gained by avoiding compilations that are now retrieved from the repository in Ř+, after which speedup goes back to baseline. At iteration 5 we introduce a phase change by changing the type of height map used by the algorithm, triggering a deoptimization event; Ř decides to recompile the new program, while Ř+ gets it from cache. Moreover, after the phase change, Ř+ manages to avoid overgeneralization and dispatches to a more specialized version already present in the repository, leading to a speedup throughout the new phase (green).

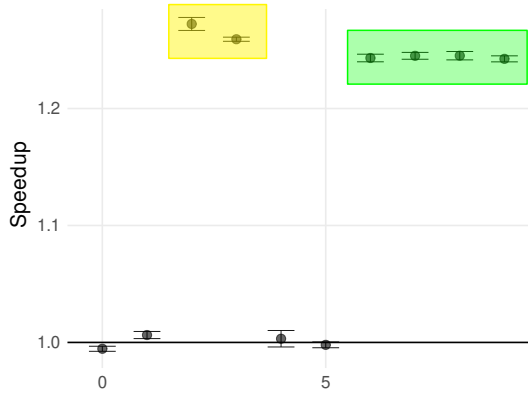

Fig. 23. Raytracer has a phase change at iteration 5

To show how Ř+ handles over-generalization during recompilation, consider the code snippet in Fig. 25. We call a function `f`, which in turn must look up the definitions of `g1` and `g2`. During the first 10 invocations of `f`, `g1` and `g2` are bound under context C1. In the 11th iteration, we redefine `g2`, replacing it with a polymorphic version. Finally, in the 21st iteration `g1` becomes polymorphic as well. In Ř these phase changes lead to degraded performance, as shown in Fig. 24. For Ř+, we start off from a repository already trained under the three different clients independently. In the replay run, speculative dispatch is able to separate the different call-site targets and select a more specialized binary for the new contexts at iterations 12 and 22, subsequently leading to highly performant runs that are not marred by overgeneralization. It is worth noting that, when dispatching to a *correct* speculative context there is always the chance of not choosing a good version; this might lead to deoptimization, however not ideal, this pathology is less severe than a recompilation in a just-in-time compiler.

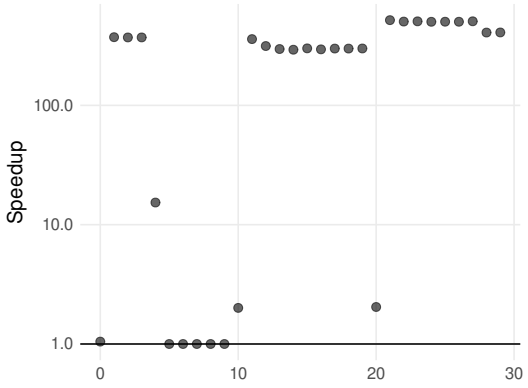

Fig. 24. Performance of Ř+ over Ř

```
f = function(a, b, r = 0) {
  for (i in 1:100000) { r = r + g1(a,1) + g2(1,b) }
  r
}
# C1
g1 = function(a,b) { bitwShiftL(a,b) }
g2 = g1
foo(60,2) # 10 times
# C2
g2 = function(a,b) { a=as.integer(a); bitwShiftL(a,b) }
f(60,2) # 10 times
# C3
g1 = function(a,b) { a=as.integer(a); bitwShiftR(a+5,b) }
g2 = function(a,b) { b=as.integer(b); bitwShiftR(a+6,b) }
f(60,2) # 10 times
```

Fig. 25. Code snippet

9 CONCLUSION

Just-in-time compilers optimize programs by learning from their history during execution. They observe the arguments at call sites, gather information about the types flowing into various expressions, remember call targets, and specialize a function by speculating on the feedback collected thus far. These compilers are capable of recovering from wrong assumptions by deoptimizing and recompiling to a more generic behavior. All this makes just-in-time compiled programs performant, at the cost of significant time spent in making these decisions and recompiling functions to suit their current context. This cost is often witnessed in the form of high warmup times that degrade the user experience and sudden slowdowns during execution.

This paper offers a novel view of just-in-time compilation. By reifying all assumptions made by the compiler into speculative contexts, it becomes possible to generate reusable units of compilation. Each function can be compiled into multiple versions, each specialized for a set of applicable contexts. Storing these versions into a single repository allows for additional off-line optimizations and deduplication of redundant versions. Our experimental results are encouraging in terms of the reduced warmup times.

While the prototype implementation explored here is for the R language, our approach may extend to other dynamic languages. The main issue that should be investigated is whether the cost of speculative contextual dispatch is acceptable in language with lighter-weight functions. In R, function calls are extremely expensive, and each function tends to do a lot of work. So the dispatching overhead can be amortized easily. It would be interesting to reproduce our experiments in the context of a language such as JavaScript or Python. Another question is how our approach would scale in a language with richer contexts, such as JavaScript, where they would need to describe user-defined classes.

10 DATA AVAILABILITY STATEMENT

Our code is open source and available online [Mehta et al. 2023].

ACKNOWLEDGMENTS

The authors thank the OOPSLA reviewers for their helpful comments. This work was supported by NSF grants CCF-1910850, CNS-1925644, and CCF-2139612, as well as the GAČR EXPRO grant 23-07580X (RiGID).

REFERENCES

Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1094811.1094835

Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/3359619.3359744

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 (2020). https://doi.org/10.1145/3428288

Olivier Flückiger, Jan Ječmen, Sebastián Krynski, and Jan Vitek. 2022. Deoptless: Speculation with Dispatched on-Stack Replacement and Specialized Continuations. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3519939.3523729

Michael Hahsler. 2022. Recommenderlab: An R framework for developing and testing recommendation algorithms. arXiv:2205.12371 [cs.IR]. (May 2022). https://doi.org/10.48550/ARXIV.2205.12371

Urs Holzle. 1994. *Adaptive optimization for self: Reconciling high performance with exploratory programming.* Ph. D. Dissertation. https://www.proquest.com/dissertations-theses/adaptive-optimization-self-reconciling-high/docview/304113365/se-2 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-23.

Pramod G. Joisha, Samuel P. Midkiff, Mauricio J. Serrano, and Manish Gupta. 2001. A Framework for Efficient Reuse of Binary Code in Java. In *Conference on Supercomputing (SC)*. https://doi.org/10.1145/377792.377902

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

Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. 2022. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 869–884. https://www.usenix.org/conference/atc22/presentation/khrabrov

Mike Love, Rafael Irizarry, Vince Carey, et al. 2013. Genomicsclass/labs: RMD source files for the Harvardx series ph525x. github.com/genomicsclass/labs.

Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. *Artifact of "Reusing Just-in-Time Compiled Code"*. https://doi.org/10.5281/zenodo.8330884

Tyler Morgan. 2008. Throwing Shade Ray Tracer. www.tylermw.com/throwing-shade/.

Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO51591.2021.9370314

Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. 2007. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, USA, 74–88. https://doi.org/10.1109/CGO.2007.29

Manuel Serrano. 2021. Of JavaScript AOT Compilation Performance. *Proc. ACM Program. Lang.* ICFP (2021). https://doi.org/10.1145/3473575

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

Haichuan Wang, Peng Wu, and David Padua. 2014. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. In *Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2581122.2544153

Christian Wimmer et al. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* OOPSLA (2019). https://doi.org/10.1145/3360610

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Symposium on New Ideas in Programming and Reflections on Software (Onward!)*. https://doi.org/10.1145/2509578.2509581

Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proc. ACM Program. Lang.* OOPSLA (2018). https://doi.org/10.1145/3276494

Roman Zhuykov and Eugeniy Sharygin. 2017. Ahead-of-time compilation of JavaScript programs. *Programming and Computer Software* 43, 1 (2017). https://doi.org/10.1134/S036176881701008X

Received 2023-04-14; accepted 2023-08-27