

On-Stack Replacement for Program Generators and Source-to-Source Compilers

Grégory M. Essertel
gesserte@purdue.edu
Purdue University
USA

Ruby Y. Tahboub
rtahboub@purdue.edu
Purdue University
USA

Tiark Rompf
tiark@purdue.edu
Purdue University
USA

Abstract

On-stack replacement (OSR) describes the ability to replace currently executing code with a different version, either a more optimized one (tiered execution) or a more general one (deoptimization to undo speculative optimization). While OSR is a key component in all modern VMs for languages like Java or JavaScript, OSR has only recently been studied as a more abstract program transformation, independent of language VMs. Still, previous work has only considered OSR in the context of low-level execution models based on stack frames, labels, and jumps.

With the goal of making OSR more broadly applicable, this paper presents a surprisingly simple pattern for implementing OSR in source-to-source compilers or explicit program generators that target languages with structured control flow (loops and conditionals). We evaluate our approach through experiments demonstrating both tiered execution and speculative optimization, based on representative code patterns in the context of a state-of-the-art in-memory database system that compiles SQL queries to C at runtime. We further show that casting OSR as a high-level transformation enables new speculative optimization patterns beyond what is commonly implemented in language VMs.

CCS Concepts: • Software and its engineering → Compilers.

Keywords: OSR, on-stack replacement, metaprogramming, code generation

ACM Reference Format:

Grégory M. Essertel, Ruby Y. Tahboub, and Tiark Rompf. 2021. On-Stack Replacement for Program Generators and Source-to-Source Compilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3486609.3487207>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

GPCE '21, October 17–18, 2021, Chicago, IL, USA

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9112-2/21/10.

<https://doi.org/10.1145/3486609.3487207>

1 Introduction

The idea of on-stack replacement (OSR) is to replace code while it is running with a different version. The two main motivations are *tiered execution* and *speculative optimization*.

Tiered Execution. OSR was pioneered in just-in-time (JIT) compilers, concretely in the SELF VM [23] in the early 1990s. Various forms of dynamic compilation were known before. JIT compilers of the zeroth generation compiled whole programs during loading, or individual methods the first time they were called. For large programs, this leads to high overheads in compilation time. Many methods are used only rarely, so compilation does not pay off. First-generation JIT compilers improved on this model by splitting execution into two tiers: code starts out running in an interpreter, which counts invocations per method and triggers compilation only after a threshold of n calls. While this refined model is effective in focusing compiling efforts on “hot” methods, it can only switch from interpreted mode to compiled mode on method calls or returns but not *in the middle of* long-running methods, i.e., methods that contain loops. As an extreme case, a program consisting of a single main method that runs a billion loop iterations will never be able to profit from compilation. With OSR, however, one can count loop iterations, trigger background compilation once the *loop* becomes hot, and switch from interpreted to compiled code *within a running loop*, while the method is still executing.

Taking a more general view, code can be executed using a range of interpreters or compilers that make different trade-offs in the spectrum of compilation time vs. running time, i.e., optimization effort vs optimization quality. A typical set up is a low-level interpreter for fast startup, then a simple compiler, then a compiler with more aggressive optimizations.

Speculative Optimization and Deoptimization. A compiler can more aggressively optimize code if it is allowed to make optimistic assumptions, for example, that a certain guard condition is always *true*. However, if any such assumption is violated at runtime, there needs to be a fail-safe mechanism that allows the system to *deoptimize*. The premise for this technique is that deoptimization cases are infrequent enough so that the incurred overhead is outweighed by the performance gained on the fast path. As a concrete example, Java JIT compilers typically make speculative decisions based on the currently loaded class hierarchy. In particular, if a method defined in a certain class is never overridden in

a subclass, all calls to this method can be *devirtualized* since the precise call target is known. Based on the known call target, the compiler can further decide to inline the method. However, if a class that is newly loaded *does* override the method, this violates the original speculative assumption and all the optimizations that were based on this assumption need to be rolled back. In this example, all running instances of the optimized code have to be aborted since continuing under wrong assumptions about the class hierarchy would be incorrect. In other cases, there is a bit more flexibility when to deoptimize. For example, type feedback with polymorphic inline caches (PIC) [20] caches a limited number of call targets per call site. PIC misses do not necessarily mean that execution of compiled code needs to abort immediately, but deoptimization and recompilation are typically triggered after a certain threshold of misses.

Today, all cutting-edge VMs and JIT compilers (HotSpot, Graal for Java; SpiderMonkey, V8, JSC for JavaScript) support both forms of OSR, tiered execution and speculative optimization with deoptimization. Following recent related work [12, 26], we use the term “OSR” symmetrically to refer to both optimizing and deoptimizing transitions; older work does not recognize deoptimization as a form of OSR.

Source-to-Source Compilers and Program Generators.

Source-to-source compilation is a widely used approach for implementing domain-specific languages, and explicit program generators are often used to implement high-performance libraries because they can provide specialized code generation facilities in ways that are out of reach for fully automatic compilers (JIT or otherwise). While in many cases code is generated and compiled offline and then available for future use, there are also important use-cases where code is generated and compiled on the fly, and then run once and discarded. This means that the compilation process is part of the runtime of the service, much like zeroth generation JIT compilers. Hence it seems natural to look into techniques from the JIT compiler and VM space to improve performance.

As a key motivating use case for this paper (Section 3), we consider main-memory data processing frameworks such as Spark SQL [4], and state-of-the-art query compilers based on generative programming techniques [14, 35, 42], which are essentially source-to-source compilers from SQL to C. The embedded code generators in such systems emit C source code for debuggability, portability, and to benefit from the best compiler for a given hardware platform (e.g., Intel’s ICC for x86 processors). As we demonstrate in this paper, tiered execution can improve performance for workloads that execute many complex, but quick queries—a scenario that has been identified as a key challenge by the database community [24]—through a simpler but faster compiler (Section 4). In addition, we show that speculative code generation can help the downstream C compiler generate more efficient executables on specialized code paths, e.g., using vectorization

(Section 5). What is needed are OSR techniques adapted to this setting of explicit code generation, that are generic and easy to use but allow the generation of efficient code.

Liberating OSR from Low-Level VMs. How can we bring the benefits of OSR to this setting? That is the question we address in this paper! The first and obvious idea would be to turn systems like main-memory databases into full-blown VMs. But often that is not practical. First, implementing all the necessary infrastructure, including a bytecode language and facilities like a high-performance low-level interpreter to deoptimize into, would require a huge engineering effort in an area that is not the main purpose of the system, and require deep expertise in areas unfamiliar to database developers. Second, generating structured source code is often important, for optimization (no irreducible control flow) and for debuggability. In addition, there are cases where a given platform dictates a certain target language. For example, such external constraints may require generating Java, JavaScript, or C source for interoperability.

We follow recent work, in particular D’Elia and Demetrescu [12], in viewing OSR as a general way to transfer execution between related program versions, articulated in their vision to “pave the road to unprecedented applications [of OSR] that stretch beyond VMs”. Or, in the words of Flückiger et al. [17]: “Speculative optimization gives rise to a large and multi-dimensional design space that lies mostly unexplored”. By making speculative optimization meta-programmable and integrating it with corresponding toolkits, in our case LMS (Lightweight Modular Staging) [36], we give programmers a way to explore new uses of speculative optimization without needing to hack on a complex low-level VM.

Key Idea: Duplicate, Optimize, Trampoline. The key idea is to interrupt a loop at the granularity of one or a handful of loop iterations, switch to a different compiled implementation of the same loop and resume at the same iteration count. We achieve this as follows:

1. Duplicate loops into multiple identical copies; new copies may be created and JIT-compiled on demand at runtime.
2. Optimize each version independently; typically, with different objectives, e.g., fast turnaround vs. aggressive optimization, or with a certain speculation pattern in mind.
3. Allow each version to exit and trampoline into another version; to preserve semantics, switching must happen between loop iterations, not in the middle of an iteration.

Duplicating loops in this way suggests a view of OSR as a program transformation, akin to a data-dependent form of loop unswitching. To enable separate and dynamic compilation, loops, or loop nests, have to become their own compilation units. This can be achieved elegantly using a form of lambda lifting, i.e., representing the loop body as a function and making the function top-level by turning all the free variables of the loop into parameters of the extracted function.

Translating away OSR abstractions early in the compilation chain means that we do not need to worry about saving and restoring registers at the right moment or preserving OSR primitives across optimizations, as is the case for low-level implementations of OSR in compiler back-ends. At the same time, it is important to note that standard intraprocedural optimizations can be applied to any loop before the OSR transformation.

This early translation also leads to an appealingly simple correctness story. OSR variants of a loop are derived from semantically equivalent copies, and then optimized *independently* using a specific objective. Speculative optimization patterns can be implemented using guard checks and fall-back to the default variant, as long as the speculation logic preserves the core contract of only switching *between* loop iterations, i.e., either fully executing an iteration or not at all. Making guard checks and switching logic explicit as *regular program instructions* guarantees that later compiler passes can remain unmodified, and that they remain free to apply optimizations independently on each loop variant. So as long as those downstream OSR-oblivious optimizations are correct, and each speculation pattern is correct in isolation, the entire OSR pattern is correct.

In summary, this paper makes the following contributions:

- Intellectually, this paper proposes an extremely simple model of OSR that improves our conceptual understanding of the technique by exposing its essence. In practical terms, this paper shows how program generators and source-to-source compilers can emit OSR patterns enabling them to profit from tiered execution and speculative optimization without a complex VM (Section 2).
- As a case study, we add OSR capabilities to a state-of-the-art SQL to C compiler, a setting representative of many high-performance code generation scenarios (Section 3).
- We demonstrate that tiered execution based on our design successfully reduces end-to-end query execution time for workloads with complex but short-running queries, an important use case that has been recently posed as a challenge by the database community [24] (Section 4).
- We further show that speculative optimization can have significant benefits in a setting like SQL, providing speculation capabilities beyond what current JIT compilers have to offer, in particular enabling speculative vectorization based on dynamic monitoring of the selectivity of filter predicates, but also handling variable-size data types and inlining of data structures such as indexes (Section 5).

Section 6 surveys related work; Section 7 concludes.

2 A Simple Source-to-Source Model of OSR

Our OSR transformation can be applied either manually by a programmer, automatically by a compiler, or in a metaprogramming setting via macros or staging. As a running example, consider the following Scala program, which computes

the dot product of two vectors, given as `Float` arrays `x` and `y` along with their size `n`, with the little twist that only values in `x` greater than 100 are considered:

```
var res = 0.0f
for (i <- 0 until n) {
  if (x(i) > 100) res += x(i) * y(i)
}
println(res)
```

2.1 Advanced Speculation Patterns

By default, Scala compiles to JVM bytecode, and the JVM's JIT compiler will naturally support a variety of OSR techniques to optimize this code at runtime. But what if we want more? As our first example, let's consider a domain-specific speculation pattern not implemented on standard JVMs: switching between a vectorized and a non-vectorized version of a loop, based on runtime profiling. This is a simplified version of the case study in Section 5.3. Taking some notational liberties, a SIMD-enabled vectorized version can be written as:

```
@simd for (i <- 0 until n) {
  res += (x(i) > 100) & (x(i) * y(i))
}
```

While vectorization is *usually* beneficial, it will lead to *more* work when `x(i) <= 100` for most `i`. But this selectivity factor is only known at runtime! The OSR solution is to speculatively start executing a vectorized version, monitor if the condition really is true most of the time, and when it is not, switch to a scalar version of the loop. The scalar version can perform the same kind of monitoring and switch back to vectorized execution. To achieve this, we first desugar the starter code into a plain while loop:

```
var res = 0.0f; var i = 0
while (i < n) {
  if (x(i) > 100) res += x(i) * y(i)
  i += 1
}
println(res)
```

Now we can transform it into an equivalent OSR-enabled program following the ideas described in the introduction. The `while` loop is vectorized and lifted into a separate function `loop1`. In addition, there is another function `loop2`, which implements the default version:

```
var res = 0.0f; var i = 0; var loop = loop1
def loop1() = {
  @simd while (loop == loop1 && i < n) {
    res += (x(i) > 100) & (x(i) * y(i))
    i += 1
    if (oracle()) loop = loop2
  }
  if (i >= n) DONE else NOT_DONE
}
def loop2() = {
  while (loop == loop2 && i < n) {
    if (x(i) > 100)
      res += x(i) * y(i)
    i += 1
    if (oracle()) loop = loop1
  }
  if (i >= n) DONE else NOT_DONE
}
while (loop() != DONE) {}
println(res)
```

Switching between the loop variants is achieved by assigning the desired continuation function to the variable `loop`, whenever our profiling logic, represented here as function `oracle()`, tells us that it's time to switch. This will exit the currently execution function and cause the other one to pick up at the next loop iteration.

2.2 Uncooperative Environments

It is important to note that this pattern is language-agnostic and does not at all rely on any existing JIT-compilation support or language runtime. What if we wanted to cross-compile this piece of Scala code to C? There may be a multitude of valid reasons for this, including interoperability with other native code, execution on certain closed platforms, etc. And while it is not excruciatingly complicated to map large subsets of Scala to C, we would not be able to benefit from OSR or other dynamic compilation techniques using standard techniques.

To support such settings, all we need to do is perform another lowering step and apply lambda lifting to close off and un-nest each of the loop functions. A direct mapping to top-level functions in C code is now straightforward. Importantly, each loop function can be placed in a separate file and compiled independently of the others:

```
// lambda-lifted versions, 'Ref' denotes a mutable reference cell
def loop1(loop: Ref[Func], i: Ref[Int], res: Ref[Float],
  n: Int, x: Array[Float], y: Array[Float]): Int = { ... }
def loop2(loop: Ref[Func], i: Ref[Int], res: Ref[Float],
  n: Int, x: Array[Float], y: Array[Float]): Int = { ... }
// main program
def main() {
  ...
  val res = Ref(0.0); val i = Ref(0); val loop = Ref(loop1)
  while (loop(loop, i, res, n, x, y) != DONE) {}
  println(res.read)
}
```

2.3 Dynamic Compilation and Loading

Moreover, new alternatives can be generated and loaded at runtime, as long as they follow the same interface. As we will show throughout the paper, based on this simple pattern, we can realize a variety of practically relevant OSR patterns, including lazy tiered compilation (recompile on-demand using more optimizing compilers) and various forms of speculative optimization with deoptimization. A basic variant of lazy tiered execution can be achieved as follows (disregarding vectorization):

```
def loop1(loop: Ref[Func], i: Ref[Int], res: Ref[Float],
  n: Int, x: Array[Float], y: Array[Float]): Int = {
  while (loop == loop1 && i < n) {
    if (x(i) > 100) res += x(i) * y(i)
    i += 1
    if (threshold())
      compileAsyncWithAggressiveOpts(sourceOfLoop)
        .onReady(loopc => loop = loopc)
  }
  if (i >= n) DONE else NOT_DONE
}
```

When `threshold()` indicates that the loop has become hot, an asynchronous compilation of the source code of the loop

is triggered, and, once complete, execution switches over to the freshly compiled version. While the example is shown in Scala, exactly the same can be done in C, using a standard ahead-of-time compiler such as CLANG or GCC as JIT, and loading the compiled code as dynamic library.

2.4 The Generic OSR Transformation

Based on these examples, we can describe a generic OSR template in a slightly more formal way. We again consider a suitable subset of Scala as our object language and focus on the representation of while loops. In addition, we introduce an oracle operator `select`. Given a sequence of conditions $\langle c_i \rangle_n \in \text{Exp}^n$ (where `Exp` is the syntactic category of program expressions) and a sequence of statements $\langle t_i \rangle_n \in \text{Stm}^n$, the term `select`($\langle c_i \rangle_n$){ $\langle t_i \rangle_n$ } executes a statement t_i for some i for which c_i is true. In addition, `select` returns the result of evaluating statement t_i .

We propose a generic OSR transform $\llbracket \cdot \rrbracket \setminus \langle (c_i, f_i) \rangle_n$, where $c_i \in \text{Exp}$ and $f_i \in (\text{Stm} \rightarrow \text{Stm})$ subject to the following conditions:

- For all i , the statement $f_i(e)$ is equivalent to e under the condition that c_i is true.
- On invocation of `select`, there exists an i so c_i is true.

Using the `select` oracle, we derive the generic OSR transformation as follows:

```
\llbracket while (c) e \rrbracket \setminus \langle (c_i, f_i) \rangle_n
= while (c) { select (< c_i >_n) { < f_i(e) >_n } }
= while (select (< c_i >_n) {
  < while (c_i && c) { f_i(e) }; if (!c) DONE else NOT_DONE >_n
}) != DONE {}
= def loop1() = {
  while (c1 && c) { f1(e) }
  if (!c) DONE else NOT_DONE
}
...
while (select (< c_i >_n) { < loop_i() >_n } != DONE) {}
```

The oracle `select` represents the runtime selection of the code that should be run. In addition, `select` hides the potential compilation and loading of different pieces of code. For each of the OSR situations described earlier, tiered execution and speculation, the transformation sequence $\langle (c_i, f_i) \rangle_n$ has different characteristics.

We can tie the formalism back to our examples. f_1 and f_2 are the identity function, $c_1 = \text{loop} == \text{loop}_1$, and $c_2 = \text{loop} == \text{loop}_2$. The only unknown left is how the loop variable is assigned to `loop2`. For tiered execution, the transformations f_i are simply the identity function. For any i , the c_i is evaluated to true if the compilation process has terminated for `loopi`. For speculation, the transformations f_i can differ in arbitrary ways. In our evaluation (Section 5), we will look at two different kinds of speculation. In the first situation, the different transformations make optimistic assumptions about the data handled by the program (e.g., that all values are positive). In this setting, $f_i(e)$ is more optimized than $f_{i+1}(e)$, and c_i is true as long as the assumptions made for

the f_i transformation are valid. But once the assumption is invalidated, it can never become valid again. In the second situation, the conditions can be invalidated and become true again later. The conditions c_i are evaluated, like a heuristic, based on data collected during the execution of the program. In this setting, each transformed code is more fitted for a certain kind of data pattern, and the OSR pattern allows the code to adapt to the best possible version.

2.5 Key Benefits

Given the high-level nature of the transformation, we do not need to represent OSR primitives in an IR. Instead we translate away the OSR behavior into a high-level structured, AST-like, program representation by extracting each loop body into a (potentially unbounded, and lazily JIT-generated) set of functions, following a mechanical pattern. Hence, we do not need to be concerned about how further program transformations deal with OSR primitives (a key difficulty in previous work), since there are none left! Downstream optimizations just need to be semantics-preserving with respect to individual functions, as is standard.

In practical terms, our approach allows the OSR runtime system to be embedded within the code. Thus, we can add OSR non-intrusively to a program, without an underlying VM. Compilation relies only on any of the available ahead-of-time compilers for the desired target language, and requires minimal library support. Thus, any existing ahead-of-time compiler can serve as JIT in this setting, and no bytecode language or interpreter is necessary to “deoptimize into”.

2.6 Limitations

In this paper, we focus on structured loop nests only and do not consider arbitrary recursive functions. In the setting of performance-oriented DSLs and explicit program generation, this is a very sensible choice, as performance-sensitive code tends to be dominated by such coarse-grained loop nests and fine-grained recursion is generally avoided for performance reasons. Moreover, dealing with loops within a function really is the core problem addressed by OSR. With fine-grained recursion, methods are entered and exited all the time so code can be fruitfully replaced on a per-method boundary and new invocations will pick it up. The only case not directly supported by our pattern as presented is *returning* into a different variant of a calling function. However, this functionality can be achieved by a calling convention that passes along DONE/NOT_DONE flags. Corresponding techniques have been successfully implemented in lightweight threading systems [37] using a more general form of trampolining.

3 Case study: Compiling SQL to C

We apply the ideas and tools presented in the preceding sections to a real-world case study, adding OSR to a state-of-the-art SQL to C compiler, LB2 [42]. The LB2 authors show how generative programming using LMS [36] can be used

```

// Pipeline 1
val lineitem = Stream[Record](...)
val hm = MultiMap("partkey", ...)

Print(
  Join(Seq("partkey"),
    Filter(Eq(Field("shipdate"),
      Const("1995-09-01")),
    Project(Seq("partkey",
      "shipdate", "quantity"),
    Scan(stream("lineitem"))),
    Project(Seq("type", "partkey"),
    Scan(stream("part"))))
  )
)
(a)

// Pipeline 2
val part = Stream[Record](...)

while (part.hasNext) {
  val rRec = part.next
  val prRec = rRec("type", "partkey")
  for (lRec <- hm(prRec("partkey")))
    (lRec ++ prRec).print
}
(c)

```

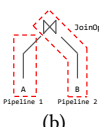


Figure 1. Shape of the generated code for a Join operator. (a) Query with a Join operator, (b) pipelines and loop structure of the code, (c) high level representation of the code generated.

to design a query compiler by implementing a simple staged interpreter for relational algebra. While compilation for SQL queries has been an active topic of research for a number of years, optimizing for fast startup and for workloads that execute many complex, but short running, queries has only recently been identified as an important challenge by the database community [24]. As we will show, the execution patterns of the SQL language makes it a perfect candidate for OSR compilation

Here is a simple query to scan a single table with a filter:

```
select user, likes from tweets where likes >= 1000
```

Knowing that the CSV format is “user name, # of likes, tweet content”, this query would be translated to C as follows:

```

int main() {
  char* tweets = ...; // open file and mmap it
  int fileLength = ...; int pos = 0;
  while (pos < fileLength) {
    int userL; char* user;
    pos += parseString(tweets + pos, &user, &userL) + 1;
    int Likes;
    pos += parseInt(tweets + pos, &likes) + 1;
    int tweetL; char* tweet;
    pos += parseString(tweets + pos, &tweet, &tweetL) + 1;
    if (likes >= 1000)
      printf("%.s, %d\n", userL, user, likes);
  }
}

```

The code consists of a single while loop that iterates over the tweet table. This pattern is a key characteristic of code generated for SQL queries. Even in complex queries which include aggregates or joins, the code is composed of top-level while loops scanning through collections (e.g., tables or data-structures). These loops are called “operator pipelines” [27]; operators such as aggregate or join are called “pipeline breakers,” as they must materialize the tuple of a pipeline and store it in a temporary data-structure, and then produce

their result within another pipeline. Figure 1 (a)-(c) shows the overall shape of the code with a Join operator. This pattern of code is a perfect example where transforming the pipeline data into OSR regions would be beneficial.

3.1 The Need to Go Beyond Generic VMs

In implementing OSR for this style of code, it is instructive to first generate code that is executed on a virtual machine which already possesses the necessary JIT mechanisms. We use the example in Figure 1 and generate code to execute the query in Scala. The generation process specializes the data structures (e.g MultiMap) to a very efficient low-level implementation on arrays. The `lineitem` table contains 60k records (a CSV file of ~ 7MB) and the `part` table 2k records (a CSV file of ~ 253 kB). The generated Scala code is compiled and run in 963ms.

If we instead generate C code that does not use any OSR constructs, the code is compiled and run in only 260ms (see Figure 2). This demonstrates that the JVM JIT compilation was unable to bring the Scala code to the same performance as C. The reasons for this difference are manifold but the general conclusion is that despite powerful JIT compilers, there are still material benefits in generating code in a low-level language at runtime. With the results of this paper, we can benefit from OSR in addition.

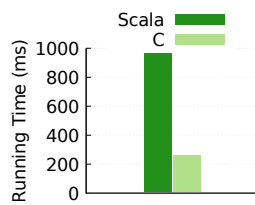


Figure 2. Runtime of the query in Figure 1 (a) with different target languages.

3.2 Implementation Considerations

Dynamic Code Loading. OSR requires some runtime support to be embedded within the generated code to load code that has been dynamically compiled. In the general setting, we assume that there are N OSR regions, and they can be compiled using M different compilers or compiler configurations. With tiered execution, we also assume that the $(i+1)^{th}$ configuration is *better* than the i^{th} configuration. Therefore, the runtime can prioritize the highest configuration available at any given time by loading the newest available code.

Most languages have support for dynamic loading through library support; therefore the challenges lie in notifying the running program of a newly available code or starting the compilation of a required piece of code.

For the first challenge, there are several possibilities that can be considered; we examine two of them here. The first is polling: in this option, the worker thread periodically checks if a new version is available. A second possibility is to block until the next version is ready, in which case an auxiliary thread is necessary for the loading step. Theoretically, the background thread option wastes fewer resources as it will mainly be waiting on a blocking event while the worker thread only has to check a single condition at each iteration.

The polling version, however, must pay the (usually more expensive) cost of polling. If this was done at each loop iteration, the overhead would be too high, with little benefit. Our solution is to check only after X iterations; however, the less optimized code may keep running while a new, faster version is available if X is too large. In that case, the performance gain is a trade-off between the polling overhead and the waste of using less optimal code.

For the second challenge, our design gives the flexibility to compile the code when it is the most efficient, without adding extra overhead in the computation thread. In the case of tiered execution for fast startup, the fast and slow compilers start the compilation process at the same time: the code generated by the fast compiler will be executed until the slow compiler terminates and the highest optimized code is available. For speculative optimization, the compilation needs to be triggered based on given criteria, e.g., low-selectivity of a `for-if` construct vectorizing the code would be beneficial. This avoids the waste of resources if the compilation is not necessary. We evaluate these differences in Section 4 and 5.

Transition and Compensation Code. At the boundary of each OSR region, some book-keeping is necessary to handle transitions. It is important to structure this code so that the downstream compiler can still fully optimize the code. If some variables are mutated within the loop, they need to be passed as pointers. Because of potential aliasing, the compiler may not be able to apply all optimizations. One solution is to dereference the pointer once when entering the region, save the value into a local variable, execute the region using those variables, and finally assign the current value back to the pointer when exiting the region. In addition, we can provide more information to the compiler on the aliasing and the alignment of the pointers, as they are known by the code generator. Generating code with `__restricted__` or `aligned(X)` annotations tends to improve loop-tiered execution and low-level speculative optimization.

In the case of high-level speculative optimizations, the data layout between regions may be arbitrarily different; the compensation code would potentially have to transform the already computed data. For example, when using a hashmap, one can speculate that the keys will only be an integer from 0 to 10, and thus use an array instead of a generic hashmap. If the assumption fails, the next OSR region may use a generic hashmap; when transitioning, all keys already inserted in the array must be correctly inserted in the hashmap, requiring task-specific compensation code.

Correctness. The OSR transformations must preserve the semantics of the loop and guarantee that switching only happens *between* loop iterations. For tiered execution, we can ensure this if we always test the switching condition at the beginning of the loop *before* the loop conditions. Indeed, if the loop condition has side-effects, it can only be executed once or it will lead to an incorrect transformation. With this

constraint, we ensure that a loop iteration (condition and body) executes completely, or not at all.

For speculative optimizations, the condition to switch to different code can be arbitrarily complex, and depends on the desired speculation pattern. Such a pattern is correct if and only if upon an aborted iteration, the program cannot have had observable effects; any such effects need to be rolled back, or implemented using a transactional commitment approach to ensure atomicity. How exactly this is achieved is the responsibility of the individual speculation pattern. Importantly, each speculation pattern only needs to ensure that its own specific abort logic behaves atomically, independent of any other component.

Initialization. For maximal efficiency, OSR code needs to jump to the best available code as soon as possible. However, in some situations the swap arises long after the new code is available. For example, if the code has a lengthy initialization, the compilation of the optimal version may already be done, but, as the code has not yet reached the OSR region, the swap will happen only later, thus negating the advantage of using OSR. In order to maximize performance, it is important that the initialization consists of high-quality code compiled by the fast compiler. For example, a good init sequence would consist of function calls to libraries which are precompiled with high optimization settings.

Nested Loops. In line with the previous paragraph, the case of nested loops can lead to unwanted overhead. Assuming that an outer loop is transformed into an OSR region, and the inner loop is taking more time than the compilation process, there is a long period between the end of the compilation and the starting of the execution of the fast code. In that situation, it may be preferable to take the inner loop or the entire loop nest as the OSR region. This example shows that it may not always be the most beneficial choice to transform the outer loop in an OSR region. From a technical point of view, however, a nested loop does not make the transformation more complex or invalid. The performance impact, however, needs to be evaluated case-by-case. Lameed and Hendren [26] discuss the impact of different approaches.

4 Tiered Compilation Experiments

All experiments are conducted on a single NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu 16.04.4 LTS. We use Scala 2.11, GCC 5.4, Clang 6.0 and TCC 0.9.26. We use the TPC-H benchmark [43] with scale factor (SF) 0.1, 0.3, 1.0 (~100MB, 300MB, and 1GB of csv files). For all our experiments, we report the median of 5 runs unless stated otherwise.

4.1 Tiered Compilation for SQL Queries

In this experiment, we evaluate tiered compilation in the context of compiled query evaluation. The LB2 query compiler [42] uses generative programming to compile SQL queries

into optimized C. The back-end of a typical query compiler consists of a small number of structured operators that emit evaluation code in the form of tight, long-running loops that process data and perform various computations (e.g., computing an aggregate over grouped data). The execution path of a compiled SQL query consists of data structure initialization, data loading, and evaluation. In practice, SQL queries in TPC-H take 100-400ms to compile using GCC with optimization flag -O3. This time is acceptable when processing large datasets, but for small-size workloads, compilation time becomes a rather considerable overhead that may defeat the purpose of compiling SQL queries to low-level code [24]. How, then, do we improve query compilation time?

A first idea is to tune the compilation optimization level without negatively impacting run time. Thus, we first study the effect of varying the optimization levels on compilation time for GCC (we also evaluate Clang in Section 4.4). We pick the simplest query, TPC-H Q6, a simple filter-aggregate, with scale factor SF0.1 (the table size is around 71MB). The hand-written Q6 is essentially a loop with an if condition that iterates over the table, filters records, and computes a single aggregate operation (i.e., the sum of a simple computation on each data record). Figure 3 shows the time to compile, run, and the end-to-end execution time for Q6 using different optimization levels in GCC. At first glance, we observe that using lower optimization flags improves compilation time by 20-40ms in GCC. Furthermore, the run times of -O3, -O2, and -O1 is nearly identical. Hence, for this basic query, GCC-O1 achieves the best compilation time and end-to-end execution time. However, using the lowest level -O0 significantly slows down run time by 5×. How can the OSR pattern discussed earlier improve the performance of small-size queries?

While using a lower optimization flag *does* improve compilation time, 60-70ms is still perceived as large for small-size queries. In our example, it is as much as the run time itself. An alternative approach would be to use a less-optimizing, faster compiler to implement the OSR pattern. The Tiny C Compiler (TCC) [31] is a fast, lightweight compiler that trades performance for speed. For instance, compiling Q6 in TCC takes 7-8ms. The key idea is to compile and launch the query using TCC until the slow compiler finishes its work, after which OSR switches execution to the fast compiled code. We evaluate this in Section 4.3.

Finally, consider the following breakdown of GCC -O3 and GCC -O0 compile times:

```
gcc -O3 -time tpch6.c gcc -O1 -time tpch6.c gcc -O0 -time tpch6.c
# cc1 0.06 0.01      # cc1 0.05 0.00      # cc1 0.02 0.01
# as 0.00 0.00      # as 0.00 0.00      # as 0.00 0.00
# collect2 0.01 0.00 # collect2 0.01 0.00 # collect2 0.01 0.00
```

We observe that the higher optimization levels spend more time in the compilation phase. Also, the linking time is high (the same amount of time TCC spends in compilation). These observations encourage the use of the OSR pattern, leaving the slow compiler a chance to perform more optimizations without “wasting” time as code is already running.

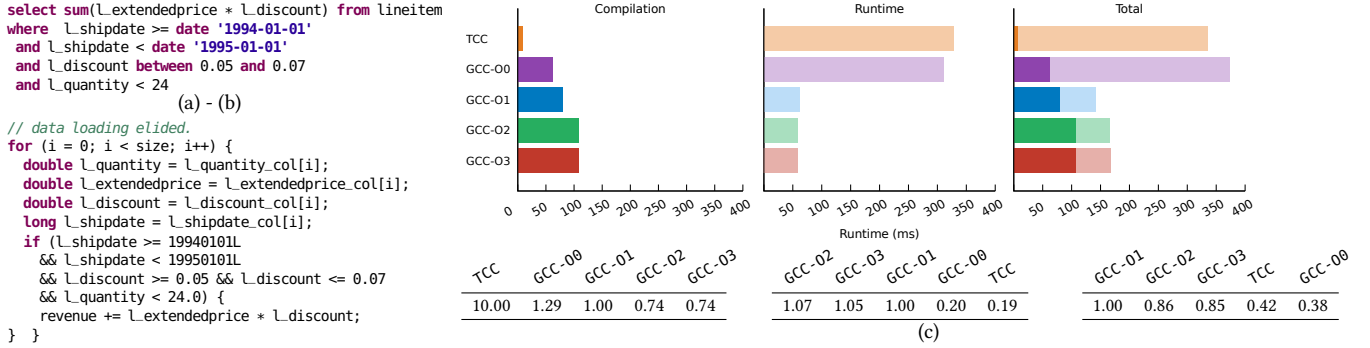


Figure 3. TPC-H Q6 in (a) SQL and (b) handwritten C, (c) execution time of Q6. Speedups in the table are relative to GCC-01.

4.2 Switching From Slow to Fast OSR Paths

Query evaluation is best described as performing a long-running loop where each iteration evaluates a number of records. Integrating OSR in query compilers requires stopping a running loop in order to switch from quickly compiled but slow code to code which has been compiled with more optimizations. In this experiment, we evaluate the two switching mechanisms discussed in Section 3.2. Recall, the first approach uses a polling mechanism. We implement polling as follows: the compiler process creates a lock file as soon as the new target becomes available. Furthermore, a switching threshold X is configured to determine the frequency at which the code checks for the lock file, for each epoch the code performs X iterations then probes the existence of the lock file. The second approach uses a background thread that blocks until it is notified by the compilation process. Once notified, it dynamically loads the dynamic library. After that, the loading thread updates a volatile variable to signal readiness to the main processing thread. Checking a volatile variable has the advantage of being much less expensive than a polling operation.

Table 1. The impact of various switching thresholds on OSR-run time using Q6 SF1.

threshold	thread	1	10^2	10^3	10^4	10^5
execution (ms)	672	721	698	697	700	705
switched at (ms)	59	62	60	58	61	73
switch iteration (x100)	2243	679	2345	2340	2500	3000

Table 1 illustrates the impact of using a background thread and various switching thresholds (1, 100, 1000, 10000 and 100000 on OSR run time in Q6 SF1). We observe that checking the availability of the fast code at each iteration incurs approximately 20ms overhead in run time compared to the other thresholds. Indeed, performing a check at each iteration uses precious computation time, thus when the switch happens the amount of useful computation that has been performed is lower. In our experiment, the code executes only 67k iterations with a threshold of 1 versus 240k for the others. Similarly, picking a large threshold potentially wastes

time depending on when the compiled target becomes ready. Indeed in the worse case, a more optimized code could be available at the beginning of an epoch just after the check thus it could be available for a full epoch without being used. This situation is exhibited by our experiment: if the threshold is 100 iterations, the OSR swap arises at iteration 234k, thus for a threshold of 100k the swap occurs at 300k and therefore the code spends more time than necessary in the less optimized code when the threshold is too high (73ms vs 62ms). On the other hand, using a background thread is 25ms faster than the best threshold used in this experiment, making it the best solution if multi-threading is supported.

4.3 Complex Code with Many OSR Regions

The OSR pattern is applicable on any long-running loop. For complex programs, each loop is processed as an independent code region where the main program coordinates running code regions. Consider the query TPC-H Q1. At a high level, Q1 is an example of an aggregate operation that divides data into groups and computes the sum, etc., for each group. The execution breaks down into three distinct code regions as follows. The first region is a loop for inserting data into a hash table. The loop in the second region traverses the hashmap, obtains the computed aggregates and performs sorting. The last region iterates over the sorted buffer and prints results.

Figure 4 shows the execution time of different TPC-H queries using TCC as the baseline, GCC with various compilation flags, and OSR where TCC and GCC are the fast and slow compilers, respectively. The OSR query time consists of TCC compilation time, a part of TCC run time, and GCC run time. We observe first, TCC compilation is very short (around 10ms), which allows starting execution early. Second, the OSR path reduces the end-to-end execution time by executing TCC at the beginning. Third, OSR preserves its expected behavior with increasing data size. However, increasing data size reduces the benefit of using OSR since the run time dominates the end-to-end execution time. It may seem surprising that in the OSR context, the code switch appears to happen *before* the GCC compilation terminates. This

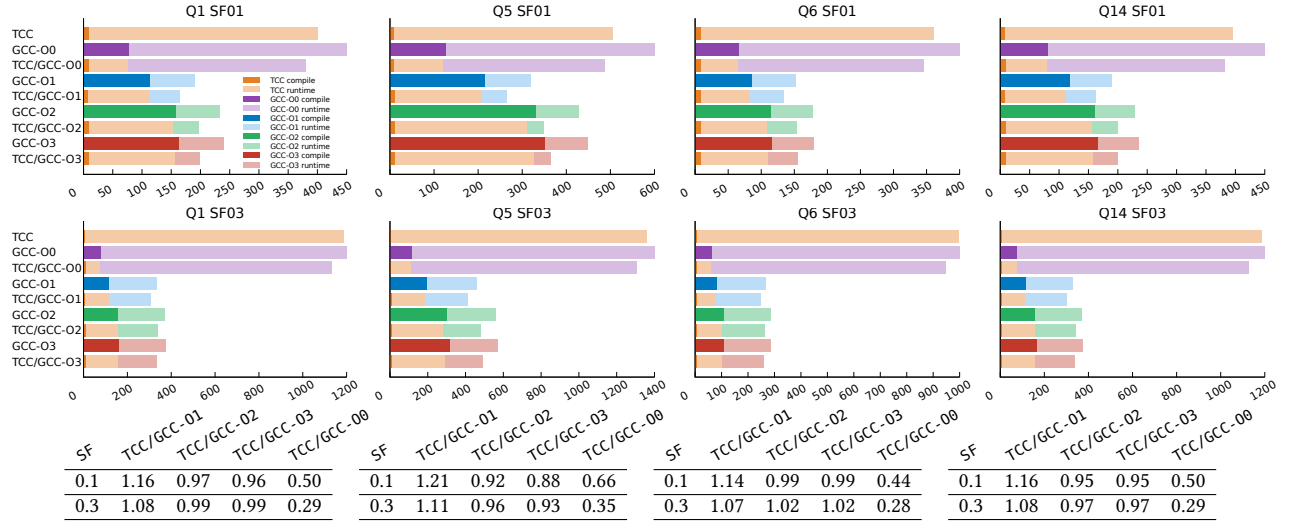


Figure 4. Tiered execution comparison for Q1, Q5, Q6, and Q14 with different configurations. For XXX/YYY, the run was an OSR execution compiled XXX then YYY. The tables list the relative speedup of OSR execution over the GCC -O1 configuration.

is due to the fact, that in the OSR context, GCC has less code to compile than in the non OSR context and it does not have to create a full executable but only a shared library. Thus the compilation is slightly faster – around 6% for TPC-H Q1.

For the runs with a scale factor of 0.1 (100MB), the best OSR execution (TCC with GCC -O1) achieves between 14 and 21% speedup over the GCC -O1 configuration. For a larger scale factor of 0.3 (300MB), the speedup is between 7 and 11%. This confirms that OSR will be beneficial, as long as the compilation time is non-negligible compared to the running time. The OSR path in Q1 reduces end-to-end run time by 20-30ms in comparison with GCC -O3, -O2 and -O1. TPC-H Q5 and Q14 are examples of join operations between five and two tables respectively. The pseudo-code in Figure 1 (d) gives a high-level implementation of a hash join operator between two tables. With this experiment, we see that even with a higher number of OSR regions in the generated code, the technique improves the run time.

4.4 Shape of Code

As discussed in Section 4.1, highly-optimizing compilers spend around two-thirds of the time in performing optimizations. Also, the linking time in GCC alone is around the same as TCC’s compilation time. However, the class of fast compilers (e.g., TCC, GCC-O0 and Clang-O0) perform a small set of optimizations to minimize compilation time at the expense of performance. For instance, less sophisticated compilers evaluate statements *individually*, whereas optimizing compilers process multiple statements together. For example, TCC generates more efficient code for nested expressions than for a cascade of expressions (such as ANF form).

In this experiment, we explore how the shape of code can help fast compilers to generate faster code. We manually implemented TPC-H Q6 using nested expressions and executed the query using TCC, GCC, and Clang. Figure 5a-b shows the compile and run time of Q6 using the handwritten code and the code generated by LB2. Table A summarizes the key outcome by listing the relative speedup of the manual code over the generated one. We observe that compilers with the slowest compiling times (TCC and Clang) benefited the most with 1.93×-1.87× speedup, respectively. Figure 5c shows the OSR execution using TCC as the fast compiler and various GCC and Clang configurations as slow compilers. Tables B and C summarize the speedup of manual and generated OSR execution paths over GCC -O1. For the manual case, we see that OSR configurations TCC/GCC-OX outperforms GCC-O1 by 12%, 6%, and 5% respectively. However, only TCC/GCC-O1 outperforms GCC-O1 (14%) in the generated setting as TCC is much less efficient in this situation. However, the pattern is conserved. Indeed, the OSR configuration always outperforms its corresponding non-OSR configuration.

While the running times of generated and handwritten code are very close for GCC and Clang, the compilation time changes a lot. This means that compilers manage to optimize the code and converge to the same version but need more time. GCC takes between 50-60% and Clang around 46% more compilation time when the code is generated. For lower optimization level or TCC, the compilation and the run time are increased by almost a 2× factor. The key insight is that it is possible, for code generators like LMS, to generate code that makes downstream compilation faster using particular constructs, e.g. nested expressions, etc.

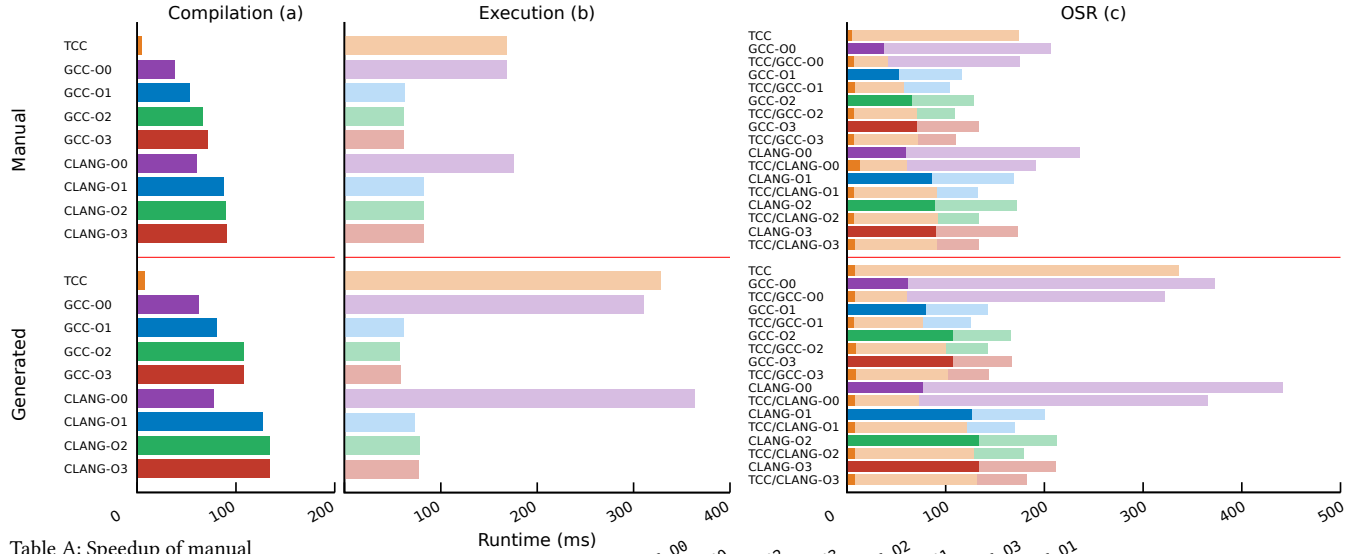


Table A: Speedup of manual over generated implementation of TPC-H Q6

Table B: Speedup of manual TPC-H Q6 over manual GCC-O1

Table C: Speedup of generated TPC-H Q6 over generated GCC-O1

	TCC	CLANG-00	GCC-00	GCC-01	GCC-02	GCC-03	CLANG-01	CLANG-02	CLANG-03
Manual	1.93	1.87	1.80	1.30	1.26	1.23	1.22	1.22	1.18
Generated	1.12	1.06	1.05	1.00	0.91	0.88	0.87	0.87	0.87

	TCC/GCC-01	TCC/GCC-02	TCC/GCC-03	GCC-01	GCC-02	TCC/CLANG-01	GCC-03	TCC/CLANG-02	TCC/CLANG-03	CLANG-01	CLANG-02	CLANG-03	TCC	TCC/GCC-00	TCC/CLANG-00	GCC-00	CLANG-00
Manual	1.12	1.06	1.05	1.00	0.91	0.88	0.87	0.87	0.87	0.69	0.67	0.67	0.67	0.66	0.61	0.56	0.49
Generated	1.14	1.00	0.99	0.99	0.86	0.85	0.84	0.79	0.78	0.71	0.67	0.67	0.44	0.42	0.39	0.38	0.32

Figure 5. Compilation time and execution time of Q6 on SF0.1, for handwritten code and generated code on different compiler configurations. XXX/YYY indicates the run was an OSR execution with first configuration XXX and second YYY.

5 Speculative Optimization Experiments

In this section, we evaluate the performance of speculative optimizations. We first look at high-level speculations where multiple code snippets are generated for the same task. We then look at low-level speculation where the same generated code is compiled using different options. In both cases, there are multiple OSR regions generated and the program generator adds the logic to swap between them efficiently.

In generic code with many different execution paths, compilers may have difficulty optimizing each individual path. Our hypothesis is that if we separate each path into its own compilation unit, the compiler will do a much better job. For example, autovectorization may be disabled because of complex control-flow, and singling out a single path may re-enable it (see Figure 6). In addition, we conjecture that it is possible to combine the different paths back together and thus optimize the original program. In the remainder of this section, we test these hypotheses on some targeted benchmarks and evaluate the possible benefits.

5.1 Variable-Size Data

An example of variable-size data is the multiple precision integer datastructure of the GMP library [15], `mpz_t`. The space used by an `mpz_t` is runtime-dependent, and the performance is linked to its size. A programmer may want to be able to

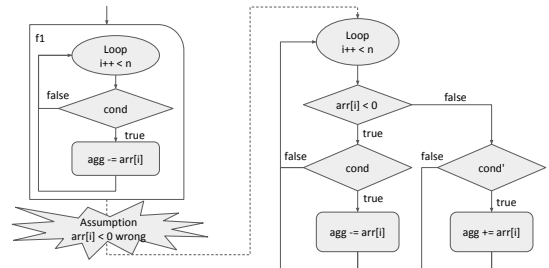


Figure 6. Complex control flow for loop vectorization. Multiple nested branches (right) prevent the compiler from vectorizing the computation. Extracting the hot path with a side exit (left) enables better optimization in general and especially vectorization of the hot path.

handle all possible scenarios in their program, however, using `mpz_t` when all data could fit into an `int` or a `long` will lead to serious performance penalties.

In this experiment, we look at three different programs that compute the sum of integers of arbitrary size: Program 1 is storing “all” values into `mpz_t`, programs 2 and 3 use a scheme where values between 0 and $2^{63} - 1$ are stored as a `long` and other values as `mpz_t`. Programs 2 and 3 assume that all values are stored as `long` and accumulate into a `long`, if the assumption is violated it continues by accumulating

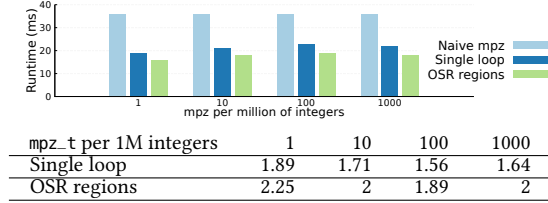


Figure 7. Comparison of the sum of an array of 5 millions integers. Average run time of 20 runs in microseconds (input array randomly generated). The table represent the speedup relative to the Naive `mpz_t` version.

in a `mpz_t`. Program 2 uses a single loop, whereas Program 3 implements the OSR logic we described in Section 2.

Figure 7 reports the average running time in milliseconds of twenty runs of these three programs, the input is an array of 5 million integers. We ran the experiment with different densities of numbers larger than 2^{63} : 1, 10, 100, and 1000 in 1 million. The higher the density, the lower the index of the first `mpz_t` will be, thus reducing the advantage of the speculation for programs 2 and 3. The experiments show that in this situation, our assumption was correct. The program with the OSR regions performed better than the single loop program. Using the flag that reports successful vectorization, we can confirm that in program 3 the loop is vectorized by GCC, but in program 2 it is not.

5.2 Inline Data Structures

Collections such as hashmaps are used to implement complex algorithms efficiently. They usually have a very good theoretical asymptotic performance; however, there are some specific cases where they are not optimal. For example, Q1 of the TPC-H benchmark has only four different keys for the group by operation using the standard TPC-H data. For generality, it is implemented using a hashmap. But in that context, hashmaps add more overhead than simply using four variables to store the different values. Based on that observation, we test some speculative high-level optimizations. We generated different code for the query: one that assumes there is going to be only 3 distinct keys, another 4, and another 5. For comparison, we also generated a program that is using the `GHashMap` from the `GLib` library. The code specialized for a given number of keys stores them in local variables instead of a more complex data-structure. Given that the number of keys is small, only a small number of comparisons is needed to find the correct variable to store the data. If the number of keys exceeds the speculated number, the program falls back to the generic implementation with the `GHashMap`. In Figure 8, we report the result of our experiment. We ran this program on a table that actually has 3, 4, 5, 6, or 25 distinct keys. We can see that when the assumption was correct (number of key speculated higher than the actual number of keys), the specialized code performs much better than the generic hashmap. But even more

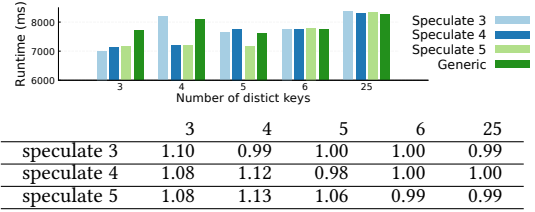


Figure 8. Running time of generated code with speculation on the number of distinct keys, and a generic hashmap implementation. The x axis represents the actual number of distinct keys. The table display the speed up of each configuration relative to the generic hashmap implementation.

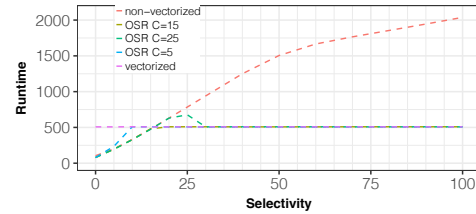


Figure 9. Comparison of vectorized, non-vectorized and OSR code (a basic map-filter-reduce loop). OSR exhibits the best behavior for any selectivity.

importantly, when the speculation is not valid, the code *does not perform worse* than the generic hashmap.

5.3 Predication vs. Branches

Modern processors support SIMD execution, and compilers need to evaluate the benefit of vectorization with some heuristic. However, the actual benefit is data-dependent. For example, consider how a for-loop computing an aggregate on an array based on some condition (e.g., a basic map-filter-reduce operation) is vectorized. Vector instructions use wide registers to compute multiple results at the same time. To handle the conditions, the processor creates a mask that *voids* the results computed that are not needed. This leads to a speed-up in most cases, but not always. Consider the situation where a SIMD instruction computes two values at the same time. If the computation of these values is significant and the majority of the time the values are discarded because the condition is false, the computation is pure overhead and the non-vectorized version may perform better.

In this experiment, we transform the loop into an OSR region and compile it once with the `-ftree-vectorized` flag and once without it. The loop also includes a counter that keeps track of the selectivity of the data: when reaching a given cutoff, the code switches between the two different compiled versions. In Figure 9, we present the result of the experiments. Empirically, we can see that the vectorized code and the non-vectorized code intersect when the selectivity is around 15%: using this for the cutoff value yields the best performance (the OSR cutoff 15 line cannot be seen as it is

under the non-vectorized line from 0 to 15 and under the vectorized line from 15 to 100). We can also see that different cutoffs switch too early or too late. Similar to the variable-size data experiment, the OSR region runs for a fixed window and checks the swapping condition when done; otherwise, it would prevent the vectorization of the loop. Different tradeoffs can be achieved by basing the swapping condition on the last window or the entire data processed so far.

In summary, both SIMD-vectorized and non-vectorized code can have highly suboptimal behavior based on the selectivity of a conditional. OSR allows us to dynamically switch between the two versions, and hence, achieve performance that is always within a small window around the optimal strategy for *any* selectivity.

6 Related Work

OSR. On-stack-replacement was first prototyped in SELF [23]. The VM was designed to combine interactivity and performance. The SELF code compiles only when needed, instead of performing a lengthy global compilation pass. The technique unwinds the stack and finds the best function to compile, then replaces all the lower stack parts with the stack of the optimized function. The SmallTalk 80 [13] system was implemented using many sophisticated techniques including polymorphic inline caching (PIC) and JIT compilation. In the case of the JIT compilation, the procedures' activation records had different representations for the interpreted code and for the native code: swapping between these representations is the same as swapping between two OSR regions in the speculative setting. Strongtalk [9] provides a type system for the untyped Smalltalk language. An OSR LLVM API is given in [11, 26], similar to our work but focused on a low-level approach within the LLVM IR more targeted toward VM implementation. OSR is also implemented in Hotspot [30] and V8 [18]. The work in [16] uses OSR to switch between garbage collection systems. Skip & Jump [44] presents an OSR API for a low-level virtual machine based on Swap-stack. Our work is different as it makes available OSR to the programmer explicitly, rather than within a runtime environment as an optimization of the language runtime.

JIT Compilers. Examples of modern JIT compilers include the Jalapeño VM [5] and its successor Jikes RVM [3]; the Oracle HotSpot VM [30] which improves performance through optimization of frequently executed application code; the Maxine metacircular research VM [45]; SPUR [6], a tracing JIT for C#; V8 [18]; Crankshaft [19]; and TurboFan [1]. Truffle [46] is built on top of Graal [29], and optimizes AST interpreters. The PyPy [8, 34] framework is written in Python and works on program traces instead of whole methods. The Mu micro VM [44] focuses on JIT compilation, concurrency, and garbage collection.

Optimization, Deoptimization, and Performance. Dynamic deoptimization was pioneered in the SELF VM to provide expected behavior with globally-optimized code. The

compiler inserts debugging information at interrupt points, while fully optimizing in between [22]. In essence, our OSR regions are similar to [22]. Debugging deoptimization in [21] deoptimizes dependent methods whenever a class loading invalidates inlining or other optimizations. PIC [20] extends the inline caching technique to process polymorphic call sites. The work in [17] deoptimizes code compiled under assumptions that are no longer valid. [7] present a generalized scheme to do exception-safe loop optimizations and exception-safe loop tiling.

Staging and Program Generation. Delite [2, 10, 40] is a general purpose compiler framework, that implements high performance DSLs, provides parallel patterns, and generates code for heterogeneous targets. LMS [36] is a library-based generative programming framework, which uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code. Code generation examples from relevant domains include Spiral [32] for digital signal processing (DSP) and Halide [33] for image processing. A subset of Spiral has been re-implemented using LMS [28], as has a compiler for a subset of MATLAB [38]. Generative programming has also been proposed as a viable approach to add SIMD intrinsics on managed language runtimes [39]. The work in [25] embeds a DSL that mimics JavaScript in Scala using LMS. Haskell is, of course, another popular host language for embedded DSLs [41].

7 Conclusions

In this paper, we have presented a surprisingly simple pattern for implementing OSR in source-to-source compilers or explicit program generators that target languages with structured control flow (loops and conditionals). We showed how on-stack-replacement provides the ability to replace currently executing code with a different version, either a more optimized one or a more general one, within a high level program. OSR has been a key component in all modern VMs for languages like Java or JavaScript for a long time, however it has only recently been studied as a more abstract program transformation, independent of language VMs. Our work extends the scope of OSR beyond the context of low-level execution models based on stack frames, labels, and jumps and makes it more broadly applicable.

We have evaluated key use cases and demonstrated attractive speedups for tiered compilation in the context of state-of-the-art in-memory database systems that compile SQL queries to C at runtime. We have further shown that casting OSR as a high-level transformation enables new speculative optimization patterns beyond what is commonly implemented in language VMs.

Acknowledgments

This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, and DOE award DE-SC0018050.

References

- [1] 2017. Launching Ignition and TurboFan. <https://v8.dev/blog/launching-ignition-and-turbofan>.
- [2] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An Embedded DSL for High Performance Big Data Processing (*BigData*). <http://infoscience.epfl.ch/record/181673/files/paper.pdf>.
- [3] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–418.
- [4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD*. ACM, 1383–1394.
- [5] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, Mary Beth Rosson and Doug Lea (Eds.). ACM, 47–65.
- [6] Michael Bebenita, Florian Brandner, Manuel Fähndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA*. ACM, 708–725.
- [7] Abhilash Bhandari and V. Krishna Nandivada. 2015. Loop Tiling in the Presence of Exceptions. In *ECOOP (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 124–148.
- [8] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy’s tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 18–25.
- [9] Gilad Bracha and David Griswold. 1993. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA*, Timlynn Babitsky and Jim Salmons (Eds.). ACM, 215–230.
- [10] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (Barcelona, Spain) (CGO 2016)*. ACM, New York, NY, USA, 194–205. <https://doi.org/10.1145/2854038.2854042>
- [11] Daniele Cono D’Elia and Camil Demetrescu. 2016. Flexible on-stack replacement in LLVM. In *CGO*. ACM, 250–260.
- [12] Daniele Cono D’Elia and Camil Demetrescu. 2018. On-stack Replacement, Distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (Philadelphia, PA, USA) (PLDI 2018)*. ACM, New York, NY, USA, 166–180. <https://doi.org/10.1145/3192366.3192396>
- [13] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *POPL*. ACM Press, 297–302.
- [14] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. USENIX Association, 799–815.
- [15] Torbjörn Granlund et al. 2002. GNU Multiple Precision Arithmetic Library 4.1.2. <http://swox.com/gmp/>.
- [16] Stephen J. Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompile with On-Stack Replacement. In *CGO*. IEEE Computer Society, 241–252.
- [17] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *PACMPL* 2, POPL (2018), 49:1–49:28.
- [18] Google. 2009. The V8 JavaScript VM. <https://developers.google.com/v8/intro>.
- [19] Google. 2010. A New Crankshaft for V8. <http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html>.
- [20] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP (Lecture Notes in Computer Science, Vol. 512)*, Pierre America (Ed.). Springer, 21–38.
- [21] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *PLDI*, Stuart I. Feldman and Richard L. Wexelblat (Eds.). ACM, 32–43.
- [22] Urs Hölzle and David Ungar. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Trans. Program. Lang. Syst.* 18, 4 (1996), 355–400.
- [23] Urs Hölzle and David M. Ungar. 1994. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In *OOPSLA*. ACM, 229–243.
- [24] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*.
- [25] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. 2012. JavaScript as an Embedded DSL. In *ECOOP*. 409–434.
- [26] Nurudeen Lameed and Laurie J. Hendren. 2013. A modular approach to on-stack replacement in LLVM. In *VEE*. ACM, 143–154.
- [27] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550. <http://www.vldb.org/pvldb/vol4/p539-neumann.pdf>
- [28] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *GPCE*. ACM, 125–134.
- [29] Oracle. 2012. OpenJDK: Graal project. <http://openjdk.java.net/projects/graal/>.
- [30] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX.
- [31] Massimiliano Poletto, Wilson C Hsieh, Dawson R Engler, and M Frans Kaashoek. 1999. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 2 (1999), 324–369.
- [32] M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (feb. 2005), 232–275. <https://doi.org/10.1109/JPROC.2004.840306>
- [33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. ACM, 519–530.
- [34] Armin Rigo and Samuele Pedroni. 2006. PyPy’s approach to virtual machine construction. In *OOPSLA Companion*, Peri L. Tarr and William R. Cook (Eds.). ACM, 944–953.
- [35] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *SNAPL (LIPIcs, Vol. 32)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 238–261.
- [36] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.
- [37] Sriram Srinivasan. 2010. *Kilim: a server framework with lightweight actors isolation types zero-copy messaging*. Ph. D. Dissertation. University of Cambridge, UK.
- [38] Alen Stojanov, Tiark Rompf, and Markus Püschel. 2019. A stage-polymorphic IR for compiling MATLAB-style dynamic tensor expressions. In *GPCE*. ACM, 34–47.

- [39] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD intrinsics on managed language runtimes. In *CGO*. ACM, 2–15.
- [40] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS* 13, 4s (2014), 134.
- [41] Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. 2014. Design exploration through code-generating DSLs. *Commun. ACM* 57, 6 (2014), 56–63.
- [42] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.
- [43] The Transaction Processing Council. 2018. TPC-H Version 2.15.0. <http://www.tpc.org/tpch/>
- [44] Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2018. Hop, Skip, & Jump: Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM. In *VEE*. ACM, 1–16.
- [45] Christian Wimmer, Michael Haupt, Michael L. Van de Vanter, Mick J. Jordan, Laurent Daynès, and Doug Simon. 2013. Maxine: An approachable virtual machine for, and in, java. *TACO* 9, 4 (2013), 30.
- [46] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *SPLASH*, Gary T. Leavens (Ed.). ACM, 13–14.