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

GRÉGORY ESSERTEL, RUBY TAHBOUB, TIARK ROMPF
PURDUE UNIVERSITY

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 show that casting OSR as a metaprogramming technique enables new speculative optimization patterns beyond what is commonly implemented in language VMs.

1 INTRODUCTION
The idea of on-stack replacement (OSR) is to replace currently executing code on the fly with a different version. There are two main motivations why one would want to do that: tiered execution and speculative optimization.

Tiered Execution. OSR was pioneered in just-in-time (JIT) compilers, concretely in the SELF VM [Hölzle and Ungar 1994] 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. 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 in the middle of the running loop, while the method is still running.

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 some optimistic assumptions. However, if assumptions are violated, a fail-safe mechanism needs to be used that allows the system to deoptimize. The overall 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 newly loaded class 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 particular 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) [Hölzle et al. 1991] 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. It is also interesting to note that PIC schemes benefit from processor-level branch prediction, which can be seen as a low-level speculative optimization approach implemented in silicon.

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 [D’Elia and Demetrescu 2018; Lameed and Hendren 2013], 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.

Generative Programming. Generative programming is often used to implement specialized code generation facilities in ways that are out of reach for automatic JIT compilers. 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 ran once and discarded. This means that the compilation process is part of the runtime of the service, much like zeroth generation JIT compilers. Therefore 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, we consider main-memory data processing frameworks such as Spark SQL [Armbrust et al. 2015], and state-of-the-art query compilers based on generative programming techniques [Essertel et al. 2018; Tahboub et al. 2018]. The embedded code generators in such systems often 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 key challenge by the database community [Kohn et al. 2018] — through a simpler but faster compiler. In addition, speculative code generation can help the downstream C compiler generate more efficient executables on specialized code paths (e.g., using vectorization). 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 practicable. First, implementing all the necessary infrastructure, including a bytecode language and facilities like a high-performance low-level interpreter to deoptimize, represents a huge engineering effort on an aspect that is not the
main purpose of the system, and requires 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 or JavaScript source for interoperability.

We follow recent work, in particular D’Elia and Demetrescu [2018], 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. [2018]: “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 generative programming toolkits, in particular LMS (Lightweight Modular Staging) [Rompf and Odersky 2012], we give programmers a way to explore new uses of speculative optimization without needing to hack on a complex and low-level VM.

**Key Ideas.** Our approach is based on two key ideas. First, we view OSR as a program transformation reminiscent of a data-dependent variant of loop unswitching. We interrupt a loop at the granularity of one or a handful of loop iterations, switch to a different compiled loop implementation and resume at the same iteration count.

The second idea is to treat loops as top-level compilation units. To enable separate compilation for either tiered execution or for speculation with dynamically generated variants, loops, or loop nests, have to become their own units of compilation. 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. Using the semantics of functions in the target language allows the framework to guarantee correctness without worrying about difficult low-level details such as saving and restoring registers. Furthermore, each compilation unit will have fewer data paths than the original loop, which lead to a better compiled code. It is also important to note that intraprocedural optimizations can be applied before the transformation.

This paper makes the following contributions:

- **Intellectually,** this paper proposes an extremely simple model of OSR that may be useful for future formal study. In particular, we believe that our model provides a simpler correctness story than previous formal models. We do not need to represent OSR primitives in an IR and instead translate away the OSR behavior into a high-level structured, AST-like, program representation. Hence, we do not need to be concerned about how further program transformations deal with OSR primitives (there are none left). Downstream optimizations just need to preserve the semantics, as usual. We introduce the key ideas by way of examples in Section 2.

- **In practical terms,** this paper shows how program generators and source-to-source compilers can emit OSR patterns which enable them to profit from tiered execution and speculative optimization in addition to standard code specialization. In addition, our approach allows the OSR runtime system to be embedded within the code. Thus, we demonstrate that we can add OSR non-intrusively to a program, without having a JIT setup. Compilation relies only on any of the available ahead-of-time compilers for the desired target language, and require minimal library support. We describe our implementation as part of the LMS metaprogramming framework [Rompf and Odersky 2012] in Section 3.

- **As a case study,** we add OSR capabilities to a state-of-the-art SQL to C compiler (Section 4) and evaluate its performance. We demonstrate that 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 challenge by the database community [Kohn et al. 2018]. We also show that the speculative optimization can have important 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. We discuss the tiered execution experiments in Section 5, and the speculative optimization experiments in Section 6.

Section 7 surveys related work; Section 8 concludes.

2 A SIMPLE SOURCE-TO-SOURCE MODEL OF OSR

Let us consider the following simple Scala program, which computes the dot product of two vectors, given as Float arrays x and y along with their size n:

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

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 wanted to cross-compile this piece of Scala code to C? Let us first desugar the code into a plain while loop:

```scala
var res = 0.0f
var i = 0
while (i < n) {
    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 lifted into a separate function `loop1`. In addition, there is another function `loop2`, which on the surface is equivalent to `loop1`. But the assumption here is that `loop2` is perhaps more optimized than `loop1`, or otherwise compiled in a different way. While ignoring for now how the runtime compiles and loads the different pieces of code (there is a mutable variable `loop` that holds a pointer to the currently execution loop body, and at one point `loop2` is going to be assigned to variable `loop` based on the result of invoking a function `oracle`), we can see that this code is equivalent to the original loop, but may switch between semantically equivalent loop implementations at any time during the execution of the loop.

```scala
var res = 0.0f
var i = 0
var loop = loop1
def loop1() = {
    while (loop == loop1) {
        if (i >= n) return DONE
        res += x(i) * y(i)
        i += 1
        // or e.g. dynamic compilation for tiered execution:
        if (oracle()) loop = loop2
        // if (threshold()) compileAsync(loop2).onReady(loopc => loop = loopc)
    }
    NOT_DONE
}

def loop2() = {
    while (loop == loop2) {
        if (i >= n) return DONE
        ... // same code as loop1, but compiled differently
    }
}
```
With that, the generic OSR transformation is:

$$\text{NOT\_DONE}$$

while (loop() != DONE) {}
println(res)

In a further lowering step, we can perform lambda lifting to close off and un-nest each of loop functions. A direct mapping to top-level functions in C code is now straightforward. Importantly, each loop function can even be placed in a separate file and compiled independently of the others:

```scala
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 = ...

def main() {
...  
  val i = Ref(0);  val res = Ref(0.0)
  val loop = Rep(loop1)
  while (loop(loop, i, res, n, x, y) != DONE) {}
  println(res)
}
```

As we will show, based on this simple pattern, we can realize a variety of practically relevant OSR patterns, including lazy tiered compilation (recompile on-demand using a more optimization compilers) and various forms of speculative optimization with deoptimization.

A 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 `<c_i>_{n∈\text{Exp}^n}` (where `Exp` is the syntactic category of program expressions) and a sequence of statements `<t_i>_{n∈\text{Exp}^n}`, `select(<c_i>_{n})` executes `t_i` for an `i` such that `c_i` is true. In addition, `select` returns the result of the evaluation of the expression `t_i`.

We propose a generic OSR transformation \[ \llbracket \text{while} \, (c) \rrbracket \llbracket \langle f_i \rangle \rrbracket _{n} \rightarrow \langle f_i \rangle _{n} \in (\text{Exp} \times \text{Exp} \mapsto \text{Exp})^n, \text{meeting the following conditions:} \]

- For all `i`, the statement \( f_i(e) \) is equivalent to \( e \) under the condition that \( c_i \) is true.
- At any point in the program, there is a `i` such that \( c_i \) is true.

With that, the generic OSR transformation is:

\[
\llbracket \text{while} \, (c) \rrbracket \llbracket \langle c_i, f_i \rangle \rrbracket _{n} = \llbracket \text{while} \, (c) \rrbracket \llbracket \langle c_i \rangle \rrbracket _{n} \rightarrow \llbracket f_i(e) \rrbracket _{n} \\
= \llbracket \text{while} \, (c) \rrbracket \llbracket \langle c_i \rangle \rrbracket _{n} \rightarrow \llbracket \text{if} \, (c) \rrbracket \llbracket f_i(e) \rrbracket _{n} \rightarrow \text{DONE} \rrbracket \llbracket \text{NOT\_DONE} \rrbracket \rightarrow \text{DONE} \rrbracket \\
= \text{def} \, \text{loop1}() = \{ \llbracket \text{while} \, (c) \rrbracket \llbracket \langle c_i \rangle \rrbracket _{n} \rightarrow \llbracket \text{if} \, (c) \rrbracket \llbracket f_i(e) \rrbracket _{n} \rightarrow \text{DONE} \rrbracket \llbracket \text{NOT\_DONE} \rrbracket \rightarrow \text{DONE} \rrbracket \\
... \\
= \text{while} \, (\text{select} <c_i>_{n}) \rightarrow \llbracket \text{loop1}() \rrbracket _{n} \rightarrow \text{DONE} \rrbracket 
\]

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 `<c_i, f_i>_{n}` has different characteristics.

We can tie the formalism back to our example: \( f_i \) is the identity function, \( c_1 = \text{loop} = \text{loop}_1, f_2(e) = e_2, \text{and } c_2 = \text{loop} = \text{loop}_2 \). The only unknown left is how the loop variable is assigned to \text{loop}_2. For tiered execution, the transformation \( f_i \) is simply the identity function. The \( c_i \) is evaluated to true if the compilation process has terminated for \text{loop}_1. For speculation, the transformations \( f_i \) can be different in arbitrary ways. In our evaluation (Section 6), 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 that \( f_{i+1}(e) \), 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 be 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
transformation is more fitted for a kind of data pattern, and the OSR pattern allows the code to
adapt to the best possible version.

Limitations. We focus on structured loop nests only and do not consider arbitrary recursive
functions. In the setting of generative programming 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 in many cases, code can be fruitfully
replaced on a per-method boundary and new invocations will pick it up.

3 METAPROGRAMMING FACILITIES FOR OSR

The high-level transformations introduced in Section 2 can be realized in any suitable way, including
directly through the use of a syntactic rewriting framework. In practice, an attractive way to build
a (simple) compiler is by programatically specializing an interpreter using a form of multi-stage
programming (staging) [Taha and Sheard 2000]. The fact that specializing an interpreter to a given
input program yields a compiled version of this program is known as the first Futamura projection
[Futamura 1971].

In the remainder of this paper, we will use Scala as the metaprogramming language, and LMS
(Lightweight Modular Staging) [Rompf and Odersky 2012] as our multi-stage programming frame-
work. In LMS, all normal Scala expressions (of type Int, String, or in general T) are executed during
code generation time, while expressions that need to be generated are of type Rep[T]. LMS provides
the infrastructure for all Rep primitive types, such as Rep[Int], Rep[Float], Rep[Array[T]], etc., and
all common operations as well as control flow operators such as if, while.

Throughout the rest of the paper, Scala code examples represent the API that our system provides,
all normal Scala expressions (of type Int, String, or in general T) are executed during
code generation time, while expressions that need to be generated are of type Rep[T]. LMS provides
the infrastructure for all Rep primitive types, such as Rep[Int], Rep[Float], Rep[Array[T]], etc., and
all common operations as well as control flow operators such as if, while.

3.1 Tiered Execution

High Level Interface. As mentioned previously, we aim at providing programmers an easy interface
for OSR. In the case of tiered execution, different compilers are used to compile the same piece of
code. As a result, the code generator does not require additional information: the programmer must
simply mark the loop as an OSR loop. Using LMS and Scala, the interface for programmers to emit
a staged while loop is given by the following function (note that => signifies a by-name parameter
in Scala):

```scala
def ___while(cond: => Rep[Boolean])(body: => Rep[Unit]): Rep[Unit]
```

In exactly the same way, we can propose a very simple interface that emits OSR-enabled while
loops, requiring a minimal amount of changes from the programmer:

```scala
val validOSR: Boolean

// Provided by our framework
def tieredExecutionOSRRegion(cond: => Rep[Boolean], body: => Rep[Unit]): Rep[Unit] = {
  if (validOSR) // Verify that OSR can/should be used.
    tieredExecutionOSRRegion(cond, body)
  else
    while (cond) body // Invokes ___while internally
}
```

The Scala syntax allows the whileOSR loop to have almost the same syntax as the regular one,
which makes its usage virtually transparent to the programmer. The validOSR flag is of type Boolean;
this branch is therefore executed during the code generation phase. When the flag is true, the tieredExecutionOSRRegion function is invoked and generates the required code. Otherwise, it calls a regular LMS while, which generates a regular loop. It means that through a simple flag, the program is able to toggle OSR; while also having the possibility to use while explicitly when a loop should never be transformed. It is of course not always the case that loops will benefit from an OSR transformation. But based on this simple API, programmers can easily build more complex heuristics when necessary. For example, we can use a more complex heuristics than a simple flag by transforming the validOSR flag into a function that returns a Boolean. Possible heuristics could be to only transform the first loop, or only the top level ones, which would thus disable nested OSR loops. With this interface, we can bring OSR to any program by changing each while loop, into a whileOSR.

Generated Code. The code generator needs to generate the low-level details to support the OSR semantics. As discussed earlier, the loop needs to become its own compilation unit: a single function, potentially in a separate file. The dot product example we used in Section 2, would be generated as follows:

```c
typedef int (function_type*) (int*, float*, int,
   float*, float*);
function_type* loaded_osr = NULL;
int osr_region1(int* i_p, float* res_p, int n,
   float* x, float* y) {
   int i = *i_p;
   float res = *res_p;
   int eCode = NOT_DONE;
   while (loaded_osr == NULL) {
      if (!(i < n)) { eCode = DONE; break; }
      res += x[i] * y[i];
      i++;
   }
   *i_p = i;
   *res_p = res;
   return eCode;
}
int main() {
   float res = 0; int i = 0;
   ...
   loaded_osr = osr_region1;
   while (loaded_osr(&i, ...) != DONE);
   printf("%.f
", res)
}
```

3.2 Speculative Optimization

In the case of speculative optimization, there exists a wider range of possibilities than in tiered optimization. We distinguish between low-level and high-level speculative optimizations.

**Low-level Speculative Optimization** is compiling the same code with different compiler configurations to produce different binaries. All generated binaries produce the correct result; however, their performance may vary depending on the input data (e.g., one version may be more efficient on negative values). Pieces of code with a simple loop traversing an array contiguously aggregating the positive values are a perfect example to illustrate this:

```c
float* dist = ...;
float agg = 0;
for (int i = 0; i < N; ++i) {
   if (dist[i] > 0) agg += dist[i];
}
```

This code snippet can be vectorized using Single Instruction Multiple Data (SIMD) instructions. This means that while the code generated can perform four floating points operations in parallel (with SSE 128bits width registers, up to 8 or 16 with AVX2 and AVX-512), some may be executed, but then discarded if the condition does not hold. The vectorized code always has the same
type Loop = Rep[Boolean] => Rep[Unit] => Unit
def lowLevelSpeculation(flags: Seq[Configuration], swap: => Rep[Int])(loop: Loop => Unit): Unit

// Example
var i: Rep[Int] = 0
var hit: Rep[Int] = 0
val arr: Rep[Array[Float]] = ...
lowLevelSpeculation(Seq(Flag("tree-vectorize"), Flag("no-tree-vectorize"))), // configs for gcc -fXX
{ if (hit < CUTOFF* i) 1 else 0 }) // heuristic to choose which configuration to run
whileSpec => // custom while loop to use for the speculative work
whileSpec (i < n) {
  if (arr[i] > 0) { agg += arr[i]; hit += 1 }
  i += 1
}

Fig. 1. API interface for low level speculative loops on a simple example.

performance no matter the selectivity value of the condition. In general, this is very beneficial, as
the computation may complete in a quarter of the time. However, in the case of very low selectivity,
the non-vectorized code would perform much better. Indeed, at the limit, if the selectivity is 0%,
the branch predictor performs perfectly well and no useless computations are executed. However, at
selectivity 100%, vectorized code is exactly 4 times faster than the non-vectorized code. Therefore,
there exists a selectivity $S$ such that vectorized code is more efficient for selectivity greater than $S$,
and the opposite is true for the selectivities lower than $S$. This selectivity $S$ depends on the cost of
the operations: the simpler the code, the more it benefits from vectorization and the lower the $S$
value is. The ideal solution is to use the more suitable code based on the selectivity of the condition.
However, the selectivity is data-dependent and non-uniform within the array: for example, the
selectivity of the condition in our example for this array $[-1, -1, -1, 1, 1, 1]$ is 50%, but it is 0%
for the first half, and 100% for the second. This means that the correct version must be selected at
runtime. In order to provide an abstract interface for this situation, we need to be able to define
different configurations and the conditions in which each configuration should be used. We propose
a generic interface in Figure 1 that allows the programmer to create a loop to be compiled with
different configurations, as well as a heuristic to swap between them.

The different configurations are expressed through an abstract Configuration class that contains
all the information necessary for the compilation. The selection function swap returns the speculated
configuration that needs to be used for the following iterations. In our example, we use a single
variable for gathering the necessary information for the configuration selection, but the function
can be arbitrarily complex.

Similarly to the tiered execution situation, the whileSpec loop is transformed into a function.
However, in the case of low-level optimization, the generated code cannot check the swapping
condition at each iteration. Indeed, if it was the case, the loop could not be vectorized, as it would
add a data-dependent exit condition. The idea, then, is to split the workload into segments, and
check the switching condition after a full segment has been executed. Once the next configuration is
required, the compilation can be started and the execution will jump to it once it becomes available.
Figure 2 shows a sketch of the generated code.

High-level Speculative Optimization is the process of generating different pieces of code for the
same task: for each piece of code, some optimistic assumptions that allows the generation of more
efficient code are made. One piece of code (the last) is required to be generic, and work without
typedef void (loop*)(int, float*, int, int*);
volatile loop osr_region1_nonvect = NULL;

int hit = 0;
int i = 0;
int next = MIN(n, INTERVAL);
while (i < n) {
    int cond = hit < CUTOFF * i;
    if (cond) { // trigger compilation
        if (osr_region1_nonvect != NULL && cond) {
            // Compiled with -fno-tree-vectorize
            osr_region1_nonvect(i, arr, next, &hit);
        } else {
            // Compiled with -ftree-vectorize
            osr_region1(i, arr, next, &hit);
        }
    } else { // Compiled with -fno-tree-vectorize
        osr_region1(i, arr, next, &hit);
    }
    i = next;
    next = MIN(n, n + INTERVAL);
}

void osr_region1(int i, float* arr, int n, int* hit_p) {
    int hit = *hit_p;
    for (; i < n; ++i) {
        if (arr[i] > 0) { agg += arr[i]; hit += 1 }
    }
    *hit_p = hit;
}

Fig. 2. Code generated for the code example in Figure 1

Fig. 3. 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.

any assumptions. In the situation where the assumption is violated, the program must recover and fall back to the next available segment of code.

We next seek to explain why speculation is highly effective. If the control flow within a loop is more complex than a single loop, compilers have difficulty applying optimizations. However, a compiler could optimize the code if the different paths were within their own compilation units. For example, in the program represented by Figure 3, the condition $arr[i] < 0$ can be speculated to always evaluate to true. Thus, the residual program can now potentially be vectorized, and in the event the speculation is correct, the performance would be greatly improved. There are many situations in which the programmer may have good reason to speculate that a condition will likely evaluate to false, such as an error checking branch. While the code is required when such an error occurs, the additional code may prevent the compiler from applying some optimizations. Using our high-level speculation interface would provide the opportunity to fully optimize the useful parts of the code, without being blocked by edge cases.

We provide an examination of this situation in our evaluation in Section 6.
3.3 Implementation Details

In this section, we will use the following example to illustrate the different technical details by focusing on the tiered execution situation. While the program is not representative of a real workload, it is representative of a large class of loops. The condition and the body have side-effects, which help highlight the small details that must be considered.

```c
var i = 0;
while (i++ < n) {
    printf("%d\n", i-1);
}
```

**Dynamic Code Loading.** The implementation of OSR requires some runtime support to be embedded within the generated code in order to load the code which 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. In the case of 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 lay on 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 to have a polling option. 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 it is necessary to use an auxiliary thread that is in charge of the loading step. We present the architectures of these two ideas in Figure 4. Theoretically, the background thread option wastes fewer resources as it is mainly 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 can 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. In the case of 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 5 and 6.

**Compensation Code.** At each boundary of the OSR regions, some extra code needs to be inserted to handle transitions. It is important to structure the 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 aliasing issues, the compiler may not be able to apply all optimizations. The idea 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 (See Figure 5). In addition, it may be necessary to give more information to the compiler on the aliasing and the alignment of the pointers, as they are known by the code
typedef int (osr_regionX_t)(...);
void* loaded_osr_region[M];
volatile int prev_version[M] = {-1, ..., -1};

// function launched as a separate thread
void* load_shared(void* args) {
    for (;;) {
        // Wait for a region to be compiled.
        int osr_reg, version;
        char path[40], name[20];
        wait_until_ready(&osr_reg, &version, path, name);

        if (version <= prev_version[osr_reg])
            continue;

        void* plugin = dlopen(path);
        loaded_osr_region[osr_reg] = dlsym(plugin, name);
        prev_version[osr_reg] = version;
    }
    return NULL;
}

int osr_regionX(int cur_version, ...) {
    ...
    while (true) {
        if (--test == 0) {
            if (poll_osr_regionX(&version, path) && version > cur_version) {
                // Save state
                void* plugin = dlopen(path);
                loaded_osr_regionX = dlsym(plugin, "osr_regionX")
                return NOT_DONE;
            }
            test = SWITCH_THRESHOLD;
        }
    }
    ...
}

(b) Polling

Fig. 4. Different strategies to test if a newest version is available and load it for tiered execution. (a) is using a background thread that blocks until the compilation process notify it. It then loads the newly compiled version. (b) is using a polling method and check every SWITCH_THRESHOLD if a newer version is available, if yes it loads it and return.

generator. Generating code with __restricted__ or aligned(X) annotations may be required. This pattern works perfectly for 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; thus requiring a task-specific compensation code.

Correctness. The OSR transformations must preserve the semantics of the original loop. 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, exiting the loop after executing it and starting a new loop would lead to an incorrect transformation. With this constraint, we ensure that a loop iteration (condition and body) execute completely, or not at all. We show how the checking order can lead to invalid code transformation in Figure 5.
int osr_region1(int* i_p, int n) {
    int i = *i_p;
    while (i++ < n) {
        if (!loaded[0]) {
            *i_p = i;
            return NOT_DONE;
        }
        printf("%d, ", i-1);
    }
    *i_p = i;
    return DONE;
}

// main
int i = 0;
if (loaded[0] || osr_region1(&i, n) != DONE) {
}
// output if swap arise after iteration i = 3
0, 1, 2, 3, 5, 6, 7, ..., n
// output if swap arise after iteration i = n - 1
0, ..., n - 1, n + 1, ... // miss stop condition

(a) Incorrect

int osr_region1(int* i_p, int n) {
    int i = *i_p;
    while (!loaded[0]) {
        if (i++ >= n) {
            *i_p = i;
            return DONE;
        }
        printf("%d
", i-1);
    }
    *i_p = i;
    return NOT_DONE;
}

// main
int i = 0;
if (loaded[0] || osr_region1(&i, n) != DONE) {
    loaded_osr[0](&i, n);
}
// output if swap arise after iteration i = 3
0, 1, 2, 3, 4, 5, 6, 7, ..., n
// output if swap arise after iteration i = n - 1
0, ..., n - 1, n + 1, ... // miss stop condition

(b) Correct

Fig. 5. Illustration on the importance of the check order. If the loop condition is carried out before the OSR check, the condition would be executed an extra time when the following OSR region start. It therefore break the language semantics. However doing the OSR check first ensures the correctness.

For speculative optimizations, the condition to switch to different code can be arbitrarily complex, and depends on the kind of speculation that is made. Such an OSR transformation would be correct if and only if upon an aborted iteration, the program can not have had observable effects; otherwise they need to be rolled back. Whereas it is not possible to define a generic model that would be correct in all situation, a windowed slicing pattern can be used. At the beginning of the loop, the state of the program is saved. Once part of the loop has been executed, the program can check for errors. If there are any, the previous state can be restored, and the OSR region can exit; the next region can then execute the loop from the saved state. This pattern does not work without additional precaution for I/O operations. It is also interesting to note that in a multi-threaded program, restoring the state may not be enough, as another thread may already have observed the modifications that have then been rolled back.

Initialization. In order to be as efficient as possible, OSR code needs to jump to the best available code as soon as possible. However, there are some situations where it does not happen. For example, if the code has a lengthy initialization, the compilation of the optimal version may be done. However, as the slower code has not yet reached the OSR region, the swap will arise very late, thus negating the advantage of using OSR altogether. In order to maximize performance, it is important that the initialization step is made of code that is optimally compiled by the fast compiler. For example, it can be made 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 will be 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 transform the inner loop into an OSR region. This example shows that it may not always be the most beneficial to transform the outer loop in an OSR region.
On a technical point of view, however, a nested loop does not make the transformation more complex or invalid. The impact on the performance, however, needs to be evaluated. Therefore, leaving the decision up to the programmer is the most logical sense in our situation. Lameed and Hendren [2013] discuss the impact of different approaches.

4 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 [Tahboub et al. 2018]. 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 important challenge by the database community [Kohn et al. 2018]. As we will show, the execution patterns of the SQL language makes it a perfect candidate for OSR compilation.

We show key parts of a simple SQL compiler in Figure 6. This code is taken from Rompf and Amin [2015] and the complete version, including various join operators and optimized datastructures, is implemented in less than 500 lines using Scala and LMS.

```
abstract class Op
case class Scan(stream: Stream) extends Op
case class Print(par: Op) extends Op
case class Project(out: Schema, par: Op) extends Op
case class Filter(pred: Predicate, par: Op) extends Op
case class Join(key: Schema, left: Op, right: Op) extends Op

def exec(op: Op)(cb: Record => Unit) = op match {
  case Scan(stream) =>
    while (stream.hasNext) {
      cb(stream.next)
    }
  case Print(par) => exec(parent) { rec =>
    rec.print
  }
  case Filter(pred, par) => exec(par) { rec =>
    if (execPredicate(pred)(rec)) cb(rec)
  }
  case Join(key, left, right) =>
    val hm = new MultiMap(key, schemaOut(left))
    exec(left) { rec =>
      hm += (rec(key), rec)
    }
    exec(right) { rRec =>
      for (lRec <- hm(rRec(key))) {
        cb(lRec ++ rRec)
      }
    }
}
```

Fig. 6. Implementation of relational query engine DSL with the corresponding staged interpreter (exec).

The LB2 query compiler [Tahboub et al. 2018] that forms the basis of our experiments is essentially a scaled-up version of this code. The LB2 authors show how generative programming using LMS can be used to design a query compiler by implementing a simple staged interpreter. Interpreters have the advantage of being relatively simple, and can easily be understood by programmers with different backgrounds, even those who are not compiler experts. For example, in Figure 6, the function hasNext returns a Rep[Boolean]; the while statement in the Scan case will therefore be generated. As such, our interpreter will actually become a code generator.

Here is a simple query to scan a single table:

```
select * from tweets
Print(Scan(stream("tweets")))
```

Following the methodology of the LB2 design (and 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 = ...;
```
```c
int pos = 0;
while (pos < fileLength) {
    int userLength; char* user;
    pos += parseString(tweets + pos, &user, &userLength) + 1;
    int nbLikes;
    pos += parseInt(tweets + pos, &likes) + 1;
    int tweetLength; char* tweet;
    pos += parseString(tweets + pos, &tweet, &tweetLength) + 1;
    printf("%.*s, %d, %.*s
", userLength, user, likes, tweetLength, tweet);
}
```

Going further, if we add a filtering operation to only display the name and the number of likes:

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

The query may be expressed as

```
Print(Project(Seq("user", "likes"), Filter(Geq(Field("likes"), Const(1000)), Scan(stream("tweets")))))
```

in the language of relational query operators. For this query, the code generator produces nearly the same code, but rather than printing the whole tuple, it generates code that prints only the expected values, with a simple conditional statement around it:

```
if (likes >= 1000)
    printf("%.*s, %d
", userLength, user, likes);
```

Here, we can see that the code is comprised of a single while loop that goes through 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” [Neumann 2011]; 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 8 (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. In our work, an OSR region is a loop that will potentially have multiple instructions at runtime, as presented above.

In looking to implement OSR for this style of code, it is instructive to first generate code that is executed on a virtual machine which already possesses all the JIT mechanisms required. We use the example found in Figure 8 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 7). This demonstrates that Scala’s JIT compilation was unable to bring the Scala code to the same performance as that of the C code. In part, this is due to the fact that the I/O operations in C are much more efficient than those in the Scala runtime. The C code can directly mmap the input file and parse the csv, whereas the Scala code needs to use a slow Stream interface. For those reasons, we want to be able to generate C code, as well as benefit from OSR.
On-Stack Replacement for Program Generators

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

(a)

(b)

Fig. 8. Shape of the generated code for a Join operator. (b) Example of a query with a Join operator, (b) highlight the pipelines and loop structure of the code, (c) is the high level representation of the code generated. The Stream and MultiMap classes are abstractions that are specialized when the code is generated.

In the generative programming setting, the programmer can use high-level constructs for generating OSR regions, without having to handle tedious low-level technical details. We examine three different kinds of OSR regions that are available in our framework, and discuss the technical details for each: tiered execution loops, low-level speculation, and high-level speculation.

5 TIERED COMPIILATION EXPERIMENTS

In this section, we evaluate the performance of OSR patterns on tiered compilation. The workload targeted in our experiments is based on compiled query processing (Section 4). In such situations, OSR has the potential to provide dramatic speedups for fast startup and workloads that are comprised of complex but short-running queries. We use the TPC-H benchmark [The Transaction Processing Council 2018] benchmark that focuses on the performance of practical analytical queries.

Experimental Setup. 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.

Dataset. We use the standard TPC-H benchmark with scale factor SF0.1, SF0.3, SF1 and SF10 (approximately 100MB, 300MB, 1GB and 10GB of csv files respectively).

Experimental Methodology. For all our experiments, the timing is based on the gettimeofday function call. We report the median of 5 runs unless stated otherwise.

5.1 Tiered Compilation for SQL Queries

In this experiment, we evaluate tiered compilation in the context of compiled query evaluation. The LB2 query compiler [Tahboub et al. 2018] 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 on 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 the highest 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 [Kohn et al. 2018]. How, then, to improve query compilation time?

```sql
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate >= '1994-01-01'
and l_shipdate < '1995-01-01'
and l_discount between 0.05 and 0.07
and l_quantity < 24
```

A first idea is to tune the compilation optimization level without negatively impacting runtime. Thus, we first study the effect of varying the optimization flag levels on compilation time for GCC (we also evaluate Clang in Section 5.4). We pick the simplest query, TPC-H Q6, illustrated in Figure 9a, with scale factor SF0.1 (the lineitem table size is approximately 71MB). The hand-written Q6 shown in Figure 9b is essentially a loop with an if condition that iterates over the lineitem table, filters records, and computes a single aggregate operation (i.e., the sum of a simple computation on each data record). Figure 9c 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 approximately 20-40ms in GCC. Furthermore, the runtimes 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 runtime 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 a large compilation time for small-size queries, in our example, it is as much as the runtime itself.

Fig. 9. TPC-H Q6 in (a) SQL and (b) handwritten C, (c) the compile, runtime and end-to-end execution of Q6. Speedups in the table are relative to GCC -O1.

A first idea is to tune the compilation optimization level without negatively impacting runtime. Thus, we first study the effect of varying the optimization flag levels on compilation time for GCC (we also evaluate Clang in Section 5.4). We pick the simplest query, TPC-H Q6, illustrated in Figure 9a, with scale factor SF0.1 (the lineitem table size is approximately 71MB). The hand-written Q6 shown in Figure 9b is essentially a loop with an if condition that iterates over the lineitem table, filters records, and computes a single aggregate operation (i.e., the sum of a simple computation on each data record). Figure 9c 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 approximately 20-40ms in GCC. Furthermore, the runtimes 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 runtime 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 a large compilation time for small-size queries, in our example, it is as much as the runtime itself.
An alternative approach would be to use a less-optimized, faster compiler to implement the OSR pattern. The Tiny C Compiler (TCC) [Poletto et al. 1999] is a fast, lightweight compiler that trades performance for speed. For instance, compiling Q6 in TCC takes approximately 7ms. 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 5.3.

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

```
gcc -O3 -time tpch6.c
# cc1 0.06 0.01
# as 0.00 0.00
# collect2 0.01 0.00

gcc -O1 -time tpch6.c
# cc1 0.05 0.00
# as 0.00 0.00
# collect2 0.01 0.00

gcc -O0 -time tpch6.c
# cc1 0.02 0.01
# as 0.00 0.00
# collect2 0.01 0.00
```

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

5.2 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 data records. Integrating OSR in query compilers requires stopping a running loop in order to switch from slowly compiled code to one which has been compiled quickly. In this experiment, we evaluate the two switching mechanisms discussed in Section 2. Recall, the first approach uses a polling mechanism. We implemented the 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 to be much less expensive than a polling operation.

Table 1 illustrates the impact of using a background thread and various switching thresholds (1, 100, 1000, 10000 and 100000 on OSR runtime in Q6 SF1). We observe that checking the availability of the fast code at each iteration incurs approximately 20ms overhead in runtime 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.

5.3 Complex Code with Many OSR Regions

The OSR pattern is applicable on any long-running loop. Applying OSR on TPC-H Q6 is straightforward since it consists of a single loop or code region. For the case of complex programs, each loop is processed as an independent code region where the main program coordinates running code regions. Consider Figure 11 that shows TPC-H Q1 in SQL. At a high level, Q1 is an example of
Fig. 10. Tiered execution comparison for Q1, Q5, Q6, and Q14 with different compiler configurations. XXX/YYY indicates the run was an OSR execution with first configuration XXX and second YYY. Tables A-D lists the relative speedup of OSR execution over the GCC -O1 configuration.

an aggregate operation that divides data into groups and computes the sum, average, 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 10 shows the execution time of four 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 all TCC compilation time, a part of TCC runtime, and

<table>
<thead>
<tr>
<th>threshold</th>
<th>runtime (ms)</th>
<th>1</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>switched at (ms)</td>
<td>59</td>
<td>62</td>
<td>60</td>
<td>697</td>
<td>700</td>
<td>705</td>
</tr>
<tr>
<td>switch iteration</td>
<td>224265</td>
<td>67856</td>
<td>234500</td>
<td>234000</td>
<td>250000</td>
<td>300000</td>
</tr>
</tbody>
</table>

Table 1. The impact of various switching thresholds on OSR-runtime using Q6 SF1 (see Figure 4).
GCC runtime. We observe first, TCC compilation is very short (around 10ms), which allows starting execution early. Second, the OSR path is successful in reducing 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 overall benefit of using OSR since the runtime 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 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 runtime 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 8 (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 region in the generated code, the technique improves the runtime.

5.4 Shape of Code

As discussed in Section 5.1, the highly-optimized compilers spend around two-thirds of compilation time in performing optimizations. Also, the linking time in GCC alone is around the same as TCC’s total compilation time. On the other hand, the class of fast compilers (e.g., TCC, GCC -O0 and Clang -O0) perform only a small set of optimizations to minimize compilation time at the expense of performance. For instance, less sophisticated compilers evaluate statements individually, whereas optimized compilers process multiple statements together. For example, TCC generates more efficient code for nested expressions than for a cascade of expression (such as ANF form).

\[
\text{//nest expression} \quad x = y + z * u; \\
\text{//ANF form} \quad x1 = z * u; \quad x = y + x1;
\]

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 12a-b shows the compile- and runtime 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 12c shows the OSR execution using TCC as the fast compiler and various GCC and Clang configurations as slow compilers. For speedup computations, we pick GCC -O1 as the best default (non-OSR) path to measure performance. Tables B and C summarize the speedup or slowdown of manual and generated OSR execution paths over GCC -O1. For the manual case, we see that the OSR
configuration 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 OSR pattern is conserved. Indeed, the OSR configuration always outperforms its corresponding non-OSR configuration.

While the running times between generated and handwritten code are very close for GCC and Clang (with optimization), the compilation time actually changes a lot. This means that compilers manage to optimize the code and converge to the same version but need more time to do it. 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 time, as well as the runtime, is increased by almost a 2× factor.

The key insight is code generation frameworks like Lightweight Modular Staging will need to generate code that makes compilation faster, e.g. nested expressions, etc. if they want to benefit fully from OSR.

6 SPECULATIVE OPTIMIZATION EXPERIMENTS

In this section, we evaluate the performance of on-stack replacement patterns in various scenarios 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 optimization 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 paths, compilers may have difficulty optimizing each path correctly. Our hypothesis is that if we separate each path into its own compilation unit, the compiler will do a much better job for each path. For example, autovectorization may be ruled out because of complex control-flow, and singling out a single path may permit it. In addition, we assume that it is possible to combine the different paths back together and thus optimize the original program. In the following paragraph, we test this hypothesis on some microbenchmarks and evaluate the possible benefits.

6.1 Type Specialization
Dynamic language VMs are all about type specialization. A generic + operation could be used on integers, doubles, or even strings, depending on context. For program generators, this is not a typical use case. Since programmatic specialization is one of the prime applications of staging, one would typically try very hard to generate type-specialized code up front.

However, there are related applications that do occur in practice: for example, needing to support for variable-precisions within the same type (see below).

6.2 Variable-Size Data
An example of variable-size data is the mpz_t datastructure of the GMP library [et al. 2002]. The space used by the data is runtime dependent, and the performance is linked to its size. A programmer may want to be able to 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 (See Figure 13). Program (1) is storing the integer value into mpz_t, the program (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.

Figure 14 reports the average running time in milliseconds of twenty runs of these three programs, all of which operate on arrays of 5 million integers. We ran the experiment with inputs having different densities of numbers larger than $2^{63}$, 1 in 1 million, 10, 100, and 1000 in 1 million. The higher the density, the lower the index of the first large number 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 region 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. In order to make the vectorization possible, program (3) needs to be written in a particular manner. Instead of exiting as soon as the assumption is violated, the program computes the aggregate on a fixed window and sets a flag that the assumption is violated. After finishing the window, the program checks the flag and swaps the OSR region upon assumption violation. The following region has to rerun the last window. This is necessary because a loop with multiple exits cannot yet be vectorized by compilers (ICC, GCC, or Clang), but it could potentially in the future and therefore improve this situation even further.

6.3 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
mpz_t agg;
mpz_init_set_ui(agg, 0);
for (it = 0; it < length ; ++it) {
    mpz_add(agg, agg, arr[it]);
}

 Throws the naive version using mpz_t for all number
and for the computation.

long agg = 0L;
mpz_t bagg;
int changed = 0;
for (it = 0; it < length; it++) {
    int val = arr[it];
    if (val & TAG) {
        if (changed) {
            mpz_add(bagg, bagg, storage[val ^ TAG]);
        } else {
            mpz_init_set_ui(bagg, agg);
            mpz_add(bagg, bagg, storage[val ^ TAG]);
            changed = 1;
        }
    } else {
        if (changed) mpz_add_ui(bagg, bagg, val);
        else agg += val;
    }
}

 Stores the value between 0 and $2^{63} - 1$ in a
int and the other in mpz_t. It is a simple loop
program that starts to assume that all values
are stored as long and accumulate into a long,
if the assumption is violated it continues by
accumulating in a mpz_t.

int add_spec(int* arr, int* it_p, long* agg_p, int interval) {
    long agg = *agg_p;
    int it = *it_p;
    int fail = 0;
    for (; it < interval; it++) {
        int val = arr[it];
        if (fail) return 1;
        *agg_p = agg;
        *it_p = it;
        return 0;
    }
}

int add(int* arr, mpz_t* storage,
        int* it_p, mpz_t agg,
        int length) {
    int it = *it_p;
    for (; it < length; i++) {
        int val = arr[it];
        if (val & TAG) mpz_add_ui(agg, agg, val);
        else mpz_add(agg, agg, storage[val ^ TAG]);
    }
}

int limit = 1000;
long agg = 0L;
mpz_t bagg;
while (it < length && !add_spec(arr, &it, &agg, limit)) {
    limit += 1000;
    mpz_init_set_ui(bagg, agg);
    if (it < length) {
        add(arr, storage, &it, bagg, length);
    }
}

Fig. 13. Different programs used for experiment on Variable-Size Data.

hashmap. But in that context, hashmaps add more overhead than simply using four variable 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 15, 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 importantly, when the speculation is not
valid, the code does not perform worse than the generic hashmap.

6.4 Loop Tiling
A famous optimization pattern in linear algebra algorithms is loop tiling. It is used to improve
locality and cache reuse. One could think that OSR could be used to speculate different tiling
strategies and allow the program to change on the fly in order to find the optimal tiling setting.
Fig. 14. Comparison of the sum of an array of 5 millions potentially large positive integers. Average runtime of 20 runs in microseconds (each run has a different input array randomly generated). The table represent the speedup relative to the Naive mpz version.

<table>
<thead>
<tr>
<th>mpz per 1M integers</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single loop</td>
<td>1.89</td>
<td>1.71</td>
<td>1.56</td>
<td>1.64</td>
</tr>
<tr>
<td>OSR regions</td>
<td>2.25</td>
<td>2</td>
<td>1.89</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 15. Runtime of generated 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 relatively to the generic hashmap implementation.

<table>
<thead>
<tr>
<th>Number of distinct keys</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>speculate 3</td>
<td>1.10</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>speculate 4</td>
<td>1.08</td>
<td>1.12</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>speculate 5</td>
<td>1.08</td>
<td>1.13</td>
<td>1.06</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

However, it is actually difficult to create a safe point (see also [Bhandari and Nandivada 2015]). In addition, knowing if or when it is beneficial to switch to a new configuration depends on timing and cache related statistics. This information is not easily accessible during compilation, and thus introduces further overhead in practice.

6.5 Predication vs Branches
Modern hardware supports SIMD, and compilers need to evaluate the benefit of vectorization with some heuristic. However, the benefit can actually be data-dependent. For example, a for-loop
computing an aggregate on an array based on some condition (e.g., a basic map-filter-reduce operation) can be vectorized (see Figure 1). The instructions emitted use wide registers to compute the result at the same time. In order to handle the conditions, the processor creates a mask that voids the results computed that are not needed. While in most cases it results in a sped-up computation, there are some limits. 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 an overhead and the non-vectorized version may be better. In this experiment, we propose to 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: upon a given cutoff, the code is going to switch between the two different assemblies coded. In Figure 16, 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.

Similarly to the variable-size data experiment, the OSR region is running for a fixed window and checks the swapping condition once it is done; otherwise, it would prevent the vectorization of the loop. There are different ways of computing the swapping condition: the decision can be made based on the last window, or on all the data that has been processed so far. Each of those strategies would have a worst-case scenario that would work for a swap at each window, while not being the optimal choice for the following one.

7 RELATED WORK

OSR. On-stack-replacement was first prototyped in SELF [Hölzle and Ungar 1994]. The SELF-93 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 [Deutsch and Schiffman 1984] 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 that swapping between two OSR regions in the speculative setting. Strongtalk [Bracha and Griswold 1993] provides a type system for the untyped Smalltalk language. An OSR LLVM API is given in [D’Elia and Demetrescu 2016; Lameed and Hendren 2013], 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 [Paleczny et al. 2001] and V8 [Google 2009]. The work in [Fink and Qian 2003] uses OSR to switch between garbage collection systems. The work in Skip & Jump [Wang et al. 2018] presents an OSR API for a low-level virtual machine based on Swapstack. Our work is different from those as it make available OSR to the programmer explicitly, rather than within a runtime environment as an optimization of the language runtime.


**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 [Hölzle and Ungar 1996]. In essence, our OSR regions implemented in this paper are similar to [Hölzle and Ungar 1996]. Debugging deoptimization in [Hölzle et al. 1992] deoptimizes dependent methods whenever a class loading invalidates inlining or other optimizations. PIC [Hölzle et al. 1991] extends the inline caching technique to process polymorphic call sites. The work in [Flückiger et al. 2018] deoptimizes code compiled under assumptions that are no longer valid. Bhandari and Nandivada [2015] present a generalized scheme to do exception-safe loop optimizations and exception-safe loop tiling.


### 8 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 metaprogramming technique enables new speculative optimization patterns beyond what is commonly implemented in language VMs.
REFERENCES


Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In Java Virtual Machine Research and Technology Symposium. USENIX.
Tiark Rompf and Nada Amin. 2015. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In ICFP.