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

Grégory Essertel, Ruby Tahboub, Tiark Rompf
Purdue University

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 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 [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 lead 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. To give a concrete example, Java JIT compilers typically make speculative decisions based on the current class hierarchy loaded. In particular, if a method defined in a certain class is never overriden 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) [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 is typically triggered after a certain threshold of misses is reached. 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.

Preprint, November 2018.

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 [11, 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.

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 zero 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 [3], and state-of-the-art query compilers based on generative programming techniques [13, 36]. 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 help reduce startup cost and execution time for small data sizes through a simpler but faster compiler, and 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.

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 into represents a huge engineering effort that is not the main purpose of the system. 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 dictated 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 [11], 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 generative programming toolkits, in particular LMS (Lightweight Modular Staging) [33], we give programmers a way to explore new uses of speculative optimization without needing to hack on a complex and low-level VM.

1. Input:

```
var res = 0
for (i <- 0 until n) {
  res += a(i) * b(i)
}
```

2. Desugar:

```
var res = 0; var i = 0
while (i < n) {
  res += a(i) * b(i)
  i += 1
}
```

3. OSR Transformation:

```
var res = 0; var i = 0
var loop = loop0
def loop0() = {
  while (loop == loop0)
    if (i >= n)
      return DONE
    res += a(i) * b(i)
    i += 1
  }
  return NOT_DONE
}
def loop1() = {
  ...
}
while(loop() != DONE) {}
```

4. With lambda lifting:

```
def loop0(loop: Var[Function], i: Var[Int], res: Var[Float], a: Array[Float],
          b: Array[Float], n: Rep[Int]): Rep[Boolean]; // version 0
def loop1(loop: Var[Function], i: Var[Int], res: Var[Float], a: Array[Float],
          b: Array[Float], n: Rep[Int]): Rep[Boolean]; // version 1
def main() {
  ...
  var loop = loop0 // XXX lift as well?
  while (loop(loop, i, res, a, b, n) != DONE) {}
}
```

Figure 1. Example: dot product in pseudocode. There are many possible variations, e.g., using function pointers directly vs dispatch codes. The (optional) lambda lifting step enables separate compilation, in particular using different compilers (fast vs slow, different optimization levels).

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.

Contributions In practical terms, this paper shows how program generators and source-to-source compilers can emit OSR patterns which enables 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. 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.

Intellectually, this paper propose an extremely simple model of OSR that may be useful for future formal study. In particular, we belive 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 how further program transformations deal with OSR primitives (there are none left). Downstream optimizations just need to preserve semantics, as usual.

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.

The paper is structured as follows:

- Technical description (Section 2).
- Tiered execution experiments (Section 3).
- Speculative optimization experiments (Section 4).
- Related work (Section 5).

2 Technical Description

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. In the case of tiered execution for loops, we can let the framework decide which loop should be converted to an OSR region based on some heuristic. For example, a possible strategy would be to convert all top-level loops; another strategy is to develop a heuristic based on the content of the loop and its position within the program. However, it can be interesting to let the programmer have some control of this process. As the loop interface does not change at all, it is not too cumbersome for the programmer to annotate the loops that should be transformed, and even pair those annotations with some background logic. For example, if nested OSR regions are not supported, the framework would ignore such annotations and generate naive loops instead. Figure 2 illustrates the API interface for the `while` loop.

In the case of speculative optimization, there exists a wider range of possibilities. Low-level speculative optimization can

```
def whileOSR(cond: ⇒ Rep[Boolean])(body: ⇒ Rep[Unit]): Rep[Unit] = {
  if (validOsr) { // Verify that OSR can/should be used.
    createOSRRegion(cond)(body)
  } else while (cond) body
}
```

API interface for `while` loop

```
var i = 0
whileOSR (i < 100) {
  ...
  i += 1
}
```

(a) High level code

```
int loaded = 0;
function_type* loaded_osr1 = NULL;
int i = 0;
if (loaded || osr1(&i, ...) != DONE)
  loaded_osr1(&i, ...)
```

(b) Generated code

Figure 2. Interface for tiered compilation of a while loop. The code is generated assuming there are exactly two different compilers (e.g. tcc and gcc).

be performed: here, the same code within a loop is compiled using different compiler configurations. For example, the code in Figure 3a is a simple loop with a body that can be vectorized using the Single Instruction Multiple Data (SIMD) instructions, but within a conditional. This means that while the code generated can perform four floating points operations in parallel, some may be executed but then discarded if the condition is false. The vectorized code will always have the same performance no matter the selectivity 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, non-vectorized code would perform much better. Indeed, if the selectivity is 0, the branch predictor would perform very well and no useless computations will be executed. We propose a generic interface in Figure 3 that allows the programmer to create a loop to be compiled with different configurations and a heuristic to swap between them.

```
def lowLevelSpeculativeCompilation(flags: Configuration, swap: ⇒ Rep[Boolean])
   (cond: ⇒ Rep[Boolean])(body: ⇒ Rep[Unit]): Rep[Unit]
def vectorize(swap: ⇒ Rep[Boolean]) =
   lowLevelSpeculativeCompilation(Flag("tree-vectorize"), swap) _

var miss = 0
var it = 0
val arr: Rep[Array[Float]] = ...
vectorize(miss.toFloat / it > CUTOFF)
  (it < n) {
  if (arr[it] > 0.0f) {
    // vectorializable code
    ...
  } else {
    miss += 1
  }
  it += 1
}
```

(a) High-level code

```
int miss = 0;
int it = 0;
int next = INTERVAL;
while (it < n) {
  if ((float)miss/it > CUTOFF) {
     // Compiled with -fno-tree-vectorize
     miss += nonVectLoop(it, arr, next);
  } else {
     // Compiled with -ftree-vectorize
     miss += vectLoop(it, arr, next);
  }
  it += INTERVAL;
  next += INTERVAL;
}
```

(b) Generated code

Figure 3. Interface for low-level speculative optimizations

The code generator can also perform high-level speculations. In that setting, two different pieces of code are generated for the same task: one which is based on an optimistic assumption that allows the generation of very efficient code,

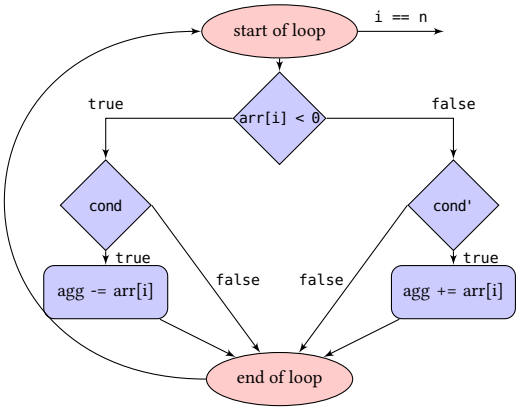


Figure 4. Complex control flow for loop vectorization. Multiple nested branches prevent the compiler from vectorizing the computation.

and one that is generic. In the situation where the assumption is violated, the program will recover, and fall back to the generic code.

For the next paragraphs, we will use the following example to illustrate the different technical details. 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 will help us to highlight the small details that need to be considered.

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

Why does speculation work so well? 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 4, the condition `arr[i] < 0` can be speculated to be always `false`. Thus, the residual program is now potentially vectorizable, and in the event the speculation is correct, the performance would be greatly improved. We provide an examination of this situation in our evaluation (see Figure 12).

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 `C` different compilers or compiler configurations. We also assume that the compilation time using the i^{th} configuration is lower than compilation time of the $i + 1^{th}$ configuration.

Most languages have support for dynamic loading through library support: the running program needs to be notified of a newly available code. 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 will periodically check 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. We present the architectures of these two ideas in Figure 5. Theoretically, the background thread option wastes less 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: polling at each loop iteration would have a high overhead. The solution is to check only after `X` iterations; however, the less optimized code may keep running while a new faster version is available. In that case, the performance gain is a trade-off between the polling overhead and the waste of using less optimal code.

This 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 slow and fast compilers can start the compilation process at the same time: the code generated by the slow compiler will be executed until the fast compiler terminate. In the case of speculative optimization, the compilation will be triggered based on given criteria, e.g. low-selectivity of a vectorializable for-if construct. This keeps the waste of resources if the compilation is not necessary.

We evaluate these differences in Section 3 and 4.

```
volatile int loaded[N][C];
void* loaded_osr[N][C];
void* load_shared(void* args) { // launched as a separate thread

  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);

    void* plugin = dlopen(path);
    loaded_osr[osr_reg][version] = dlsym(plugin, name)
    if (loaded_osr[osr_reg][version] == NULL) continue;

    loaded[osr_reg][version] = 1;
  }
  return NULL;
}


int osrX(...) {
  ...
  while (...) {
    if (loaded[..][..]) {
      ...
      return NOT_DONE;
    }
  }
  ...
  return DONE;
}
```

(a) Background thread

```
int osrX(...) {
  int test = SWITCH_THRESHOLD
  ...
  while (...) {
    if (--test == 0) {
      if (poll(...)) {
        ...
        plugin = dlopen("...");
        loop = dlsym(plugin, "...")
        return NOT_DONE;
      }
      test = SWITCH_THRESHOLD;
    }
  }
  ...
  return DONE;
}
```

(b) Polling

Figure 5. Different strategies to test if the next region is available and load it

```c
int osr1(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] || osr1(&i, n) != DONE) {
  loaded_osr[0](&i, n);
}
// output
0, 1, 2, 3, 5, 6, 7, ...
```

(a) Incorrect

```c
int osr1(int* i_p, int n) {
  int i = *i_p;
  while (!loaded[0]) {
    if (i++ >= n) {
      *i_p = i;
      return DONE;
    }
    printf("%d\n", i-1);
  }
  *i_p = i;
  return NOT_DONE;
}

// main
int i = 0;
if (loaded[0] || osr1(&i, n) != DONE) {
  loaded_osr[0](&i, n);
}
// output
0, 1, 2, 3, 4, 5, 6, ...
```

(b) Correct

Figure 6. Illustration on the importance of the check order (OSR swap arises after printing the 3)

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 need to be written, they will 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 6). 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, thus 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.

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 6.

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 noticeable effect, or they need to be rolled back. Whereas it is not possible to define a generic model that would be correct in all situation, a windows slicing pattern can be used. At the begin 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 will then be able to execute the loop from the save state. This pattern does not work without additional precaution for IO operations. It is also interesting to note that in 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.

```c
int main() {
  // Initialization.
  ...
  // First osr region.
  int it1 = 0;
  if (loaded[0][0] || osr1(&it1, ...) != DONE) {
    loaded_osr[0](&it1, ...);
  }
  ...
  // Last osr region.
  int itN = 0;
  if (loaded[N-1][0] || osrN(&itN, ...) != DONE) {
    loaded_osr[N-1](&itN, ...);
  }
}
```

Figure 7. OSR program for reducing start-up cost.

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 fast path may be done, but as the slow path did not reach the OSR region, the swap may never happen. In order to maximize performance, it is important that the initialization step be made of code that is optimally optimized by the slow path compiler. For example, it can be made of function calls to libraries which are pre-compiled with high optimization settings. Figure 7 is a representative program that can benefit from OSR tier execution transformation.

Nested loops In line with the previous paragraph, the case of nested loops can lead to unwanted overhead. Assume that an outer loop is transformed into an OSR region, and the inner loop is taking more time that 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 examples show that it may not always be the most beneficial to transform the outer loop in an OSR region.

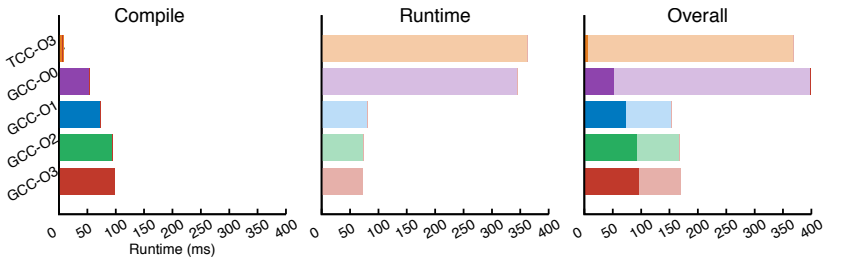
On a technical point of view however, a nested loop does not make the transformation more complex or invalid.

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where  l_shipdate >= date '1994-01-01'
 and l_shipdate < date '1995-01-01'
 and l_discount between 0.06 - 0.01 and 0.06 + 0.01
 and l_quantity < 24
```

(a)

```
// data structures initialization and data loading elided ...
for (i = 0; i < size; i++) {
  double l_quantity = l_quantity_col[i];
  double l_extendedprice = l_extendedprice_col[i];
  double l_discount = l_discount_col[i];
  long l_shipdate = l_shipdate_col[i];
  if (l_shipdate >= 19940101L && l_shipdate < 19950101L &&
    l_discount >= 0.05 && l_discount <= 0.07 &&
    l_quantity < 24.0) {
    revenue += l_extendedprice * l_discount;
  }
} ...
```

(b)



TCC-O3	GCC-O0	GCC-O1	GCC-O2	GCC-O3	GCC-O3	GCC-O2	GCC-O1	GCC-O0	TCC-O3	GCC-O1	GCC-O2	GCC-O3	TCC-O3	GCC-O0
10.6	1.4	1.0	0.8	0.8	1.10	1.08	1.0	0.23	0.22	1.0	0.92	0.91	0.42	0.39

(c)

Figure 8. 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.

3 Tiered Compilation Experiments

In this section, we evaluate the performance of on-stack replacement patterns on tiered compilation. The workload targeted in our experiments is based on compiled query processing. In spirit, compiling SQL queries to low-level code is a compiler design and code generation problem. For instance, Spark [3] uses a form of generative programming to generate Java code where it relies on the JVM for JIT compilation and OSR at runtime. Unsurprisingly, generating Java code is slow due to JVM overhead, whereas generating C code is significantly faster as demonstrated in Flare [14]. In such situations, OSR has the potential to provide dramatic speedups. Moreover, the common pattern of query evaluation is long-running loops with computations that can profit from the OSR pattern as presented. We use the TPC-H [37] 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 [37] benchmark with scale factor SF0.1, SF0.3, SF1 and SF10.

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.

3.1 Tiered Compilation for SQL Queries

In this experiment, we evaluate tiered compilation in the context of compiled query evaluation. The LB2 query compiler [36] 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 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 defeats the purpose of compiling SQL queries to low-level code. In fact, a fast interpreter is sufficient in such cases [24]. How, then, to improve query compilation time?

A first idea is to tune the compilation optimization flag 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 3.4). We pick the simplest query, TPC-H Q6, illustrated in Figure 8a, with scale factor SF0.1 (the lineitem table size is approximately 71MB). The hand-written Q6 shown in Figure 8b 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 8c 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 4.6×. 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. An alternative approach would be to use a less-optimized, faster compiler to implement the OSR pattern. The Tiny C Compiler (TCC) [29] 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.

Figure 9a shows the execution time of Q6 using TCC as 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 compile time, a part of TCC runtime, and GCC runtime. The OSR execution path achieves a 15% speedup over the default path (i.e., using only GCC). Moreover, the OSR bars in Figure 11a appear as though executing the fast target began before the compilation completed. As discussed earlier, the OSR code is structured as code regions and compiled using -fPIC and -shared flags, which result in slightly shorter compilation time.

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 the higher levels spend more time in performing optimizations. The linking time is high as well (the same amount of time TCC spends in compilation).

3.2 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 writes a lock file as soon as the fast target becomes available. Furthermore, a switching threshold is configured to determine the frequency of checking for the lock file. The second approach launches a background thread that performs a blocking read (to avoid expensive polling) and loads the dynamic library when the fast target becomes available. After that, the same thread updates a volatile variable to signal readiness to the main processing thread. Checking a volatile variable is less expensive than the constant factor of switching.

Table 1 illustrates the impact of using background threads and various switching thresholds (1, 100, 1000, 10000 and 100000 on OSR runtime in Q6 SF1). We observe that checking for the fast code on each iteration incurs approximately 20ms overhead in runtime. Similarly, picking a large threshold adds overhead depending on which point in the loop the compiled target becomes ready. On the other hand, using a background thread is 25ms faster than the best threshold used in this experiment.

	thread	1	100	1000	10000	100000
OSR runtime (ms)	672	721	698	697	700	705
switched at (ms)	59	62	60	58	61	73
switch iteration	224265	67856	234500	234000	250000	300000

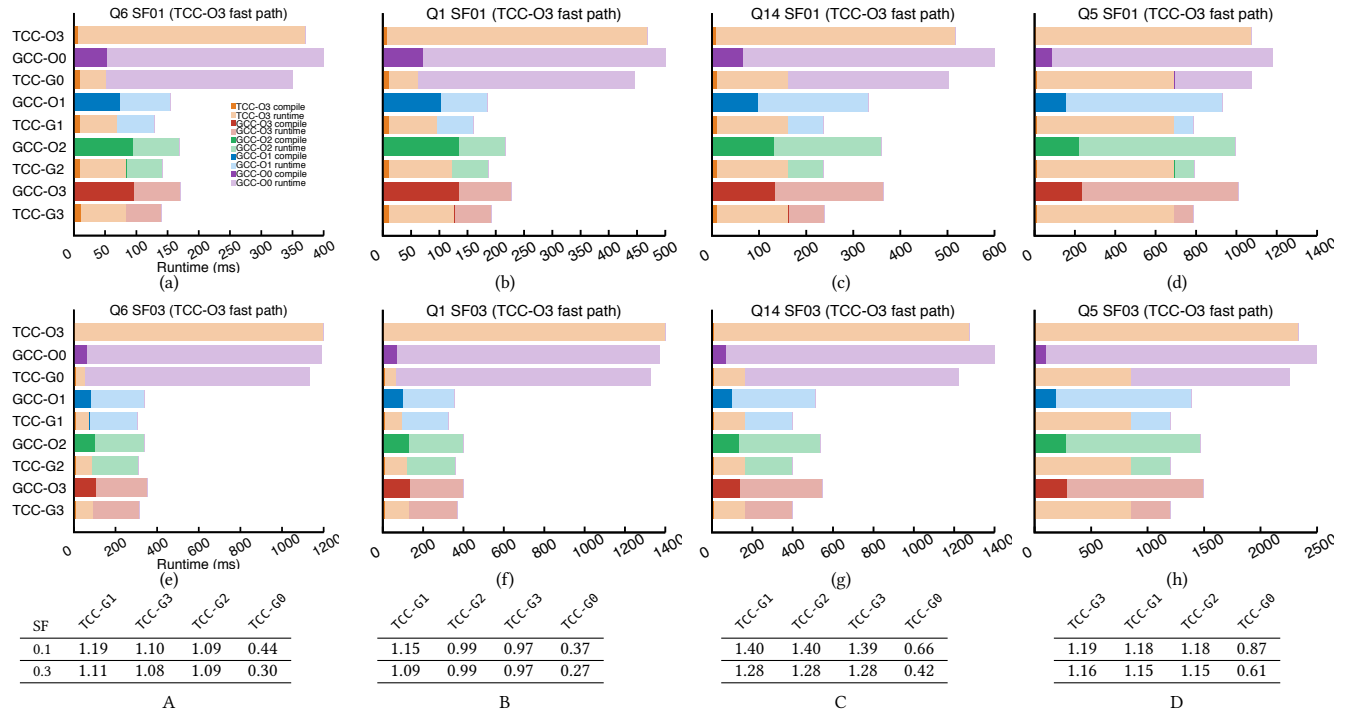
Table 1. The impact of various switching thresholds on OSR-runtime using Q6 SF1 (See Figure 5)

3.3 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 10a-b that shows TPC-H Q1 in SQL and pseudocode. At a high level, Q1 is an example of an aggregate operation that divides data into groups and computes the sum, average, etc., for each group. The pseudocode breaks down the Q1 implementation into three code regions as follows. The first region is a loop for inserting data into 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 11b-d shows the compile and runtime of executing three TPC-H queries that consist of multiple code regions. 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 as shown in 11e-h. However, increasing data size reduces the overall benefit of using OSR since the runtime dominates the end-to-end execution time.

On query by query, the OSR path in Q1 reduces end-to-end runtime by 20-30ms in comparison with GCC -O3, -O2 and -O1. TPC-H Q14 and Q5 are examples of join operations between two and five tables respectively. The pseudocode in Figure 10c gives a high level implementation of a hash join operator between two tables. Join operation consists of two loops. The first loop iterates over input data and inserts records into a multimap. The second loop probes the multimap and produces output. A key observations in Figure 11c-d, the OSR path appears as not switching directly once the fast target becomes available. This behavior is due to the longer initialization phase for complex queries. Initialization phase is always executed in TCC. There is no disadvantage for using TCC since initialization code is dominated by system call that cannot be further optimized.



SF	TCC-G1	TCC-G3	TCC-G2	TCC-G0
0.1	1.19	1.10	1.09	0.44
0.3	1.11	1.08	1.09	0.30

A

TCC-G1	TCC-G2	TCC-G3	TCC-G0
1.15	0.99	0.97	0.37
1.09	0.99	0.97	0.27

B

TCC-G1	TCC-G2	TCC-G3	TCC-G0
1.40	1.40	1.39	0.66
1.28	1.28	1.28	0.42

C

TCC-G3	TCC-G1	TCC-G2	TCC-G0
1.19	1.18	1.18	0.87
1.16	1.15	1.15	0.61

D

Figure 9. OSR execution with multiple code regions Q6, Q1, Q14 and Q5 using SF01 and SF03. Tables A-D lists relative speedup/slowdown of OSR execution over GCC -O1.

3.4 Shape of Code.

As discussed in Section 3.1, 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. A concrete example is evaluating nested expressions in TCC is faster than multiple expressions in ANF.

```
//nested expression
x = y + z * u;
```

```
//ANF form
x1 = z * u;
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 11a-b shows the compile- and runtime of Q6 using handwritten and generative programming. 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 2.17×-2.34× speedup, respectively.

Figure 11c 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, using OSR with GCC -O3, O2, O1, Clang O-3, -O2 resulted in approximately a 7%-22% speedup. Similarly, the speedup for generated OSR with GCC -O3, -O2, -O1 is between an 8%-18% speedup. On the other hand, using Clang with OSR resulted in 5%-32% slowdown in manual, and 10%-62% in generated. The cause of slowdown is attributed to the fact that the compiling time is a significant start-up cost (in comparison with the 7ms in TCC), and thus shortens the amount of runtime that can be performed until the slow compiler finishes its work. Therefore, GCC and Clang with the lowest optimization level -O0 are not good fast compiler choices for OSR in this situation. Finally, the key insight is frameworks like Lightweight Modular Staging will need to generate code that makes nested expressions, etc.

4 Speculative Optimization

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

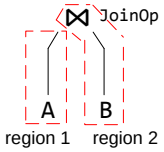
```
select l_returnflag, l_linestatus, sum(l_quantity), sum(l_extendedprice),
       sum(l_extendedprice * (1 - l_discount)),
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
       avg(l_quantity), avg(l_extendedprice), avg(l_discount), count(*)
from   lineitem
where l_shipdate <= date '1998-12-01' - interval '90' day
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

(a)

```
//Code regions in aggregate query TPC-H1

// Code region 1
- for each input record
    use the group by attributes to calculate hash value, insert or update hashmap

// Code Region 2
- for each group inside hash map
    read the computed aggregate result and insert it into a sorting buffer

- sort the result buffer based on order by attributes

// Code region 3
- for each value in sorting buffer
    print record as specified in the select clause
```

(b)

```
 // Code regions in Join operation

 // Code region 1
 - for each input record a in table A
    use join attribute(s) to calculate hash value,
    insert or update multimap

 // Code region 2
 - for each input record b in table B
     for each matching b in multimap
       concatenate a, b
       print
```



(c)

Figure 10. (a) TPC-H 1 in SQL (b) code regions in TPC-H 1 (c) code regions in join operation

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 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.

4.1 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).

4.2 Variable-Size Data

An example of variable-size data is the `mpz_t` datastructure of the GMP library [15]. 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.

We have three different programs that compute the sum of integers of arbitrary size (See Figure 12)

Figure 13 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^{31}, 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 shows 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 program (3) loop is vectorized, but program (2) 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 if necessary. The following region will have to rerun the last window. This is necessary because a loop with multiple exits can not yet be vectorized by compilers (icc, gcc, or clang), but it could potentially in the future and therefore improve this situation even further.

4.3 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, Query 1 of the TPC-H benchmark has only four different keys for the `group by` operation. For generality, it is implemented using a 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 key stores them in local variable 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

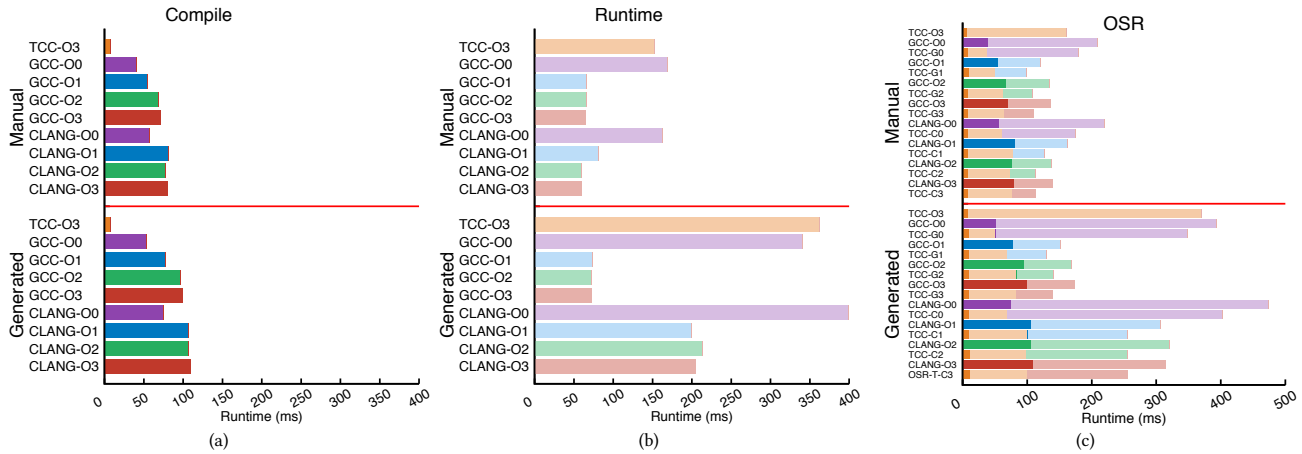


Table A: The relative speedup of manual over generated implementation of TPC-H Q6

CLANG-O2	TCC-O3	CLANG-O3	CLANG-O0	CLANG-O1	GCC-O0	GCC-O3	GCC-O1	GCC-O2
2.34	2.32	2.26	2.17	1.89	1.89	1.27	1.26	1.25

Table B: The relative speedup/ slowdown of manual TPC-H Q6 over manual GCC -O1

TCC-G1	TCC-G2	TCC-G3	TCC-C2	TCC-C3	GCC-O1	TCC-C1	GCC-O2	GCC-O3	CLANG-O2	CLANG-O3	TCC-O3	CLANG-O1	TCC-C0	TCC-G0	GCC-O0	CLANG-O0
1.22	1.12	1.1	1.08	1.07	1	0.95	0.9	0.89	0.88	0.87	0.76	0.74	0.7	0.68	0.58	0.55

Table C: The relative speedup/ slowdown of generated TPC-H Q6 over generated GCC -O1

TCC-G1	TCC-G3	TCC-G2	GCC-O1	GCC-O2	GCC-O3	TCC-C1	TCC-C2	TCC-C3	CLANG-O1	CLANG-O2	CLANG-O3	TCC-G0	TCC-O3	GCC-O0	TCC-C0	CLANG-O0
1.18	1.09	1.08	1	0.9	0.88	0.6	0.59	0.59	0.5	0.48	0.48	0.44	0.41	0.39	0.38	0.32

Figure 11. Compilation time and execution time of Q6, for the handwritten code and the generated one. We compare, different compiler and OSR configurations.

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

(1) The naive version using `mpz_t` for all number and for the computation.

```
long agg = 0L;
mpz_t bagg;
for (it = 0; it < length; it++) {
  int val = arr[it];
  if (val < 0) {
    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;
  }
}
```

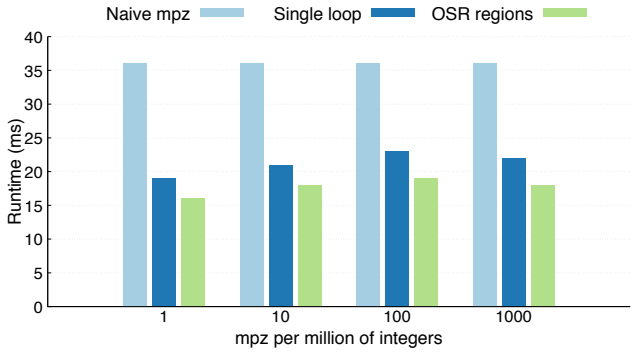
(2) Stores the value between 0 and $2^{31} - 1$ in a `int` and the other in `mpz_t`. It is a simple loop program that starts to assume that all values are stored as `int` and accumulate into a long, if the assumption is violated it continues by accumulating in a `mpz_t`.

```
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];
    fail |= val & TAG;
    agg += val;
  }
  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 >= 0)  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);
```

(3) is similar to (2), but instead of being a single loop it has two OSR regions.

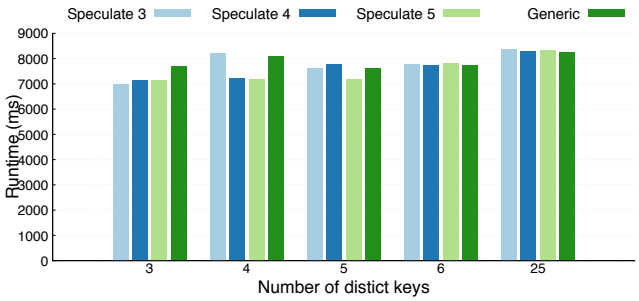
Figure 12. Different programs used for experiment on Variable-Size Data

speculated number, the program falls back to the generic implementation with the `GHashMap`. In Figure 14, 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 specialize 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.



mpz_t per 1M integers	1	10	100	1000
Single loop	1.89	1.71	1.56	1.64
OSR regions	2.25	2	1.89	2

Figure 13. 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). The table represent the speedup relative to the Naive `mpz_t` version.



	3	4	5	6	25
speculate 3	1.10	0.99	1.00	1.00	0.99
speculate 4	1.08	1.12	0.98	1.00	1.00
speculate 5	1.08	1.13	1.06	0.99	0.99

Figure 14. 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 raltively to the generic hashmap implementation.

4.4 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. However, it is actually difficult to create a safe point (see also [6]). 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.

4.5 Predication vs Branches

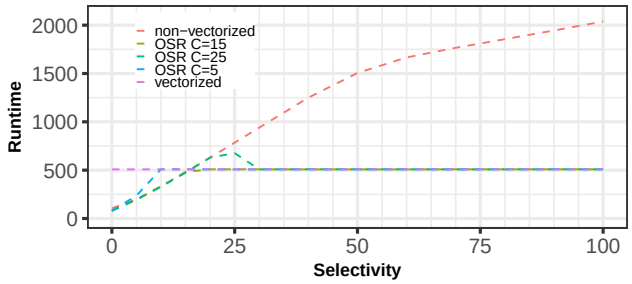


Figure 15. Comparison of vectorized, non-vectorized and OSR code. The OSR code exhibit the best behavior for any selectivity of the data.

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 3). 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 15, 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 can not 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 are switching too early or too late.

Similarly to the variable-size data experiment, the OSR region is running for a fixed window and check 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.

5 Related Work

OSR On-stack-replacement was first prototyped in SELF [23]. The SELF-93 VM was designed to combine interactivity and performance. The SELF code compiles only when needed, instead of performing a lengthly 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 [12] 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 representation is the same that swapping between two OSR regions in the speculative setting. Strongtalk [8] provides a type system for the untyped Smalltalk language. A OSR LLVM API is given in [10, 26]. OSR is also implemented in Hotspot [28] and V8 [18]. The work in [16] uses OSR to switch between garbage collection systems. The work in Skip & Jump [38] presents an OSR API for a low-level virtual machine based on Swapstack.

JIT Compilers Examples of modern JIT compilers include the Jalapeño VM [4] that targets server applications; the Oracle HotSpot [28] VM which improves performance through optimization of frequently executed application code; Jikes RVM [2]; the metacircular research VM, Maxine [39]; SPUR [5], a tracing JIT for C#; Google's V8 [18]; Crankshaft [19]; and TurboFan [1]. Truffle [40] is built on top of Graal [27], and optimizes AST interpreters. The PyPy [7, 32] framework written in Python and works on program traces instead of methods. The Mu micro VM [38] 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 implemented in this paper 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. Bhandari et. al. [6] presents a generalized scheme to do exception-safe loop optimizations and exception-safe loop tiling.

Staging and Program Generation Delite [9, 34] is a general purpose compiler framework, implements high-performance DSLs, provides parallel patterns, and generates code for heterogeneous targets. LMS [33] is library-based generative programming and compiler framework. LMS 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 l [30] for digital signal processing (DSP) and Halide [31] for image processing Haskell language is used for embedded DSLs [35]. The work in [25] embed a JavaScript DSL in Scala using LMS.

6 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. 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

[1] 2017. Launching Ignition and TurboFan. https://v8.dev/blog/launching-ignition-and-turbofan.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Gilad Bracha and David Griswold. 1993. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA*, Timlynn Babitsky and Jim Salmons (Eds.). ACM, 215–230.

[9] 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 (CGO 2016)*. ACM, New York, NY, USA, 194–205. https://doi.org/10.1145/2854038.2854042

[10] Daniele Cono D'Elia and Camil Demetrescu. 2016. Flexible on-stack replacement in LLVM. In *CGO*. ACM, 250–260.

[11] 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 (PLDI 2018)*. ACM, New York, NY, USA, 166–180. https://doi.org/10.1145/3192366.3192396

[12] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *POPL*. ACM Press, 297–302.

[13] 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.

[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*. 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 Recompilation 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)*, Pierre America (Ed.), Vol. 512. 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-Orieted Languages. *ACM Trans. Program. Lang. Syst.* 18, 4 (1996), 355–400.

[23] Urs Hölzle and David M. Ungar. 1994. A Third-Generation SELF Implementation: Reconsiling 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] Oracle. 2012. OpenJDK: Graal project. http://openjdk.java.net/projects/graal/.

[28] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX.

[29] 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.

[30] M. Puschel, 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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. 2014. Design exploration through code-generating DSLs. *Commun. ACM* 57, 6 (2014), 56–63.

[36] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.

[37] The Transaction Processing Council. [n. d.]. TPC-H Version 2.15.0. http://www.tpc.org/tpch/

[38] 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.

[39] 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.

[40] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *SPLASH*, Gary T. Leavens (Ed.). ACM, 13–14.