Surgical Precision JIT Compilers

Tiark Rompf [‡*] Arvind K. Sujeeth[†] Kevin J. Brown[†] HyoukJoong Lee[†] Hassan Chafi[‡†]
Kunle Olukotun[†]

[‡]Oracle Labs: {first.last}@oracle.com [*]EPFL: {first.last}@epfl.ch
[†]Stanford University: {asujeeth, kjbrown, hyouklee, kunle}@stanford.edu

Abstract

Just-in-time (JIT) compilation of running programs provides more optimization opportunities than offline compilation. Modern JIT compilers, such as those in virtual machines like Oracle's HotSpot for Java or Google's V8 for JavaScript, rely on dynamic profiling as their key mechanism to guide optimizations. While these JIT compilers offer good average performance, their behavior is a black box and the achieved performance is highly unpredictable.

In this paper, we propose to turn JIT compilation into a precision tool by adding two essential and generic metaprogramming facilities: First, allow programs to invoke JIT compilation explicitly. This enables controlled specialization of arbitrary code at runtime, in the style of partial evaluation. It also enables the JIT compiler to report warnings and errors to the program when it is unable to compile a code path in the demanded way. Second, allow the JIT compiler to call back into the program to perform compile-time computation. This lets the program itself define the translation strategy for certain constructs on the fly and gives rise to a powerful JIT macro facility that enables "smart" libraries to supply domain-specific compiler optimizations or safety checks.

We present Lancet, a JIT compiler framework for Java bytecode that enables such a tight, two-way integration with the running program. Lancet itself was derived from a high-level Java bytecode interpreter: staging the interpreter using LMS (Lightweight Modular Staging) produced a simple bytecode compiler. Adding abstract interpretation turned the simple compiler into an optimizing compiler. This fact provides compelling evidence for the scalability of the staged-interpreter approach to compiler construction.

In the case of Lancet, JIT macros also provide a natural interface to existing LMS-based toolchains such as the Delite parallelism and DSL framework, which can now serve as accelerator macros for arbitrary JVM bytecode.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Run-time environments; D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming

General Terms Design, Languages, Performance

Keywords JIT Compilation, Staging, Program Generation

PLDI '14, June 9 – 11 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594316

```
class Record(fields: Array[String], schema: Array[String]) {
  def apply(key: String) = fields(schema indexOf key)
  def foreach(f: (String,String) => Unit) =
    for ((k,i) <- schema.zipWithIndex) f(k,fields(i))
}
def processCSV(file: String)(yld: Record => Unit) = {
  val lines = FileReader(file)
  val schema = lines.next.split(",")
  while (lines.hasNext) {
    val fields = lines.next().split(",")
    val rec = new Record(fields,schema)
    yld(rec)
  }
}
```

Figure 1. Example: Processing CSV files.

1. Introduction

Just-in-time (JIT) compilation of running programs presents more optimization opportunities than offline compilation. Modern JIT compilers, such as those found in virtual machines like Oracle's HotSpot for Java[38] or Google's V8 for JavaScript [19] base their optimization decisions on dynamic properties of the program which would not be available to a static compiler. To give one key example, most modern VMs keep track of the call targets of virtual method calls, which enables the JIT compiler to perform inlining across virtual calls if the call site is monomorphic, i.e. it only resolves to a single target, or polymorphic for a small, fixed number. Such an inlining decision is speculative; at a later time in the program's execution, the method call might actually resolve to a different target, which was not encountered before. In this case, the caller needs to be *deoptimized*. The compiled code, which was optimized using too optimistic assumptions, is thrown away and recompiled to take the new information into account. These and similar approaches go back to the seminal work of Urs Hölzle and others on the Self [10] and StrongTalk [7] VMs.

A Motivating Example However, there is a lot more runtime information that could be exploited, but is not readily accessible via simple profiling alone. Let us consider an example (in Scala): we would like to implement a library function to read CSV files. A CSV file contains tabular data, where the first line defines the schema, i.e. the names of the columns. We would like to iterate over all the rows in a file and access the data fields by name (sample data on the right):

```
processCSV("data.txt") { record =>
  if (record("Flag") == "yes")
    println(record("Name"))
}
```

```
Name, Value, Flag
A, 7, no
B, 2, yes
...
```

Moreover, we would like to process files of arbitrary schema and be able to iterate over the fields of each record as (key,value) pairs:

```
processCSV("data.txt") { record =>
  for ((k,v) <- record) print(k + ": " + v); println
}
```

A straightforward implementation of processCSV is shown in Figure 1. The runtime information we wish to exploit is the schema array: we can easily see that schema is never modified after reading the first line of the file. As we know both the shape of the data that follows and the code that performs the processing, we could specialize our processing logic accordingly. In our second snippet above, we would not need to execute a nested loop over the record fields but could directly execute

```
while (lines.hasNext) {
  val fields = lines.next().split(",")
  print("Name: "  + fields(0))
  print("Value: " + fields(1))
  print("Flag: "  + fields(2))
  println
}
```

and get away without the overhead of a name-to-column mapping and a record abstraction altogether.

Alas, no existing JIT compiler will perform this kind of specialization reliably. Why? There are just too many things it would need to guess. First, it would need to guess the right chunk to compile. It must pick the loop after schema initialization, but not the full method. Then it must prove that schema is actually a constant, which is difficult because it is stored in Record objects that are arguments to a method call, yld(rec), which needs to be inlined. Finally, multiple threads may read different files at the same time, and we want to obtain a specialized compiled code path *per invocation*. Specialization per *program point* – even temporal specialization with deoptimization and recompilation – would not work if multiple versions need to be active at the same time.

Roadblocks for Deterministic Performance The word "reliably" in the preceding paragraph is indeed our main concern. While JIT compilers provide good average performance for a large set of programs, their behavior is in general a black box and in many cases, the achieved performance is highly unpredictable. The programmer has no influence on when and how JIT compilation takes place. Key optimization decisions are often based on magic threshold values; the HotSpot Server compiler, for example, imposes a 35 byte method size limit for early inlining. Thus, small changes in the program source can drastically affect its runtime behavior. Since many optimization decisions rely on dynamic profile data, performance can vary significantly even between runs of the same program [17]. Profile data may be outdated or inaccurate after program transformations like inlining, if it is gathered before. To keep the runtime overhead tractable, profiling is usually based on single call sites, a scheme easily confused by higher order control flow [12]. More generally, the set of dynamic properties used for optimizations is naturally limited to those that can be easily detected and monitored. Another key source of non-determinism on managed runtimes are GC pauses. While JIT compilers generally try to reduce heap allocations by mapping non-escaping objects to registers, no guarantees can be given that a certain piece of code will not produce garbage, although this would be highly desirable for latency critical applications. Finally, like many offline compilers, JIT compilers do not usually provide extension points for domain-specific optimizations or facilities to specialize generic code in a user-defined way. Previous work has shown order of magnitude speedups for such facilities [41].

All this means that despite the impressive advances of JIT compiler technology, for programmers who seek top performance or operate under constraints that require deterministic execution characteristics, the standard advice is still to avoid managed language runtimes and JIT compilation and just write their programs in C.

JIT Compilation as Metaprogramming The goal of the present work is to remedy this situation by turning JIT compilation into a precision tool. A key observation is that JIT compilers share essential aspects with partial evaluators and staged meta-programming systems: A JIT compiler does not operate on closed programs but in the presence of *static data*, live objects on the heap, and *static code*, pieces of the program which can be executed right away, either because they are already compiled or because the VM also includes an interpreter. What sets current JIT compilers apart is their fully automatic nature, unobservable to the running program, and their support for dynamic deoptimization and recompilation.

```
// JIT compile function f (explicit compilation)
def compile[T,U](f: T => U): T => U
// evaluate f at JIT-compile time (compile-time execution)
def freeze[T](f: => T): T
// unroll subsequent loops on xs
def unroll[T](xs: Iterable[T]): Iterable[T]
```

Figure 2. JIT API required for CSV example.

```
class Record(fields: Array[String], schema: Array[String]) {
  def apply(key: String) = fields(freeze(schema indexOf key))
  def foreach(f: (String,String) => Unit) =
    for ((k,i) <- unroll(freeze(schema.zipWithIndex))) f(k,fields(i))
}
def processCSV(file: String) = {
  val lines = FileReader(file)
  val schema = lines.next.split(",")
  compile { yld: (Record => Unit) =>
    while (lines.hasNext) { ... yld(rec) }
  }
}
```

Figure 3. CSV example with explicit JIT calls.

We consider only the latter essential and argue that JIT compilers should also come with reflective high-level APIs and extensibility hooks instead of being black-box systems. In particular, we propose to augment JIT compilers with two intuitive and general metaprogramming facilities, upon which further functionality can be built:

- Explicit compilation: if programs can invoke JIT compilation explicitly, code can be specialized with respect to preexisting objects on the heap, in the style of partial evaluation [28]. It also allows the JIT compiler to report warnings and errors to the program when it is unable to compile a code path in the demanded way, enabling stronger contracts about generated code than automatic background compilation.
- Compile-time computation: if the JIT compiler can call back into the running program, it can let the program itself define the meaning and translation strategy for certain constructs on the fly and based on context, in the style of multi-stage programming [48]. This gives rise to a JIT macro facility that makes the JIT API extensible and enables "smart" libraries to supply domain-specific compiler optimizations or safety checks.

Going back to our CSV example, we show that we need a JIT API to provide just three high-level methods (Figure 2), which we use in key places to achieve the desired specialization (Figure 3). The result of processCSV is now guaranteed to be a JIT compiled function that has all computation on schema evaluated at JIT-compile time, due to explicit compilation via compile and compile-time execution via freeze and unroll. However, compilation might fail with an exception if the argument of freeze cannot be evaluated during compilation. We argue that this is OK, and even desirable: instead of running suboptimal code, we want to obtain a guarantee that certain optimizations are performed and give the programmer the opportunity to react if not. Once again we stress that we do not propose to replace existing automatic JIT compilation but to add an additional facility for use cases where deterministic performance is important.

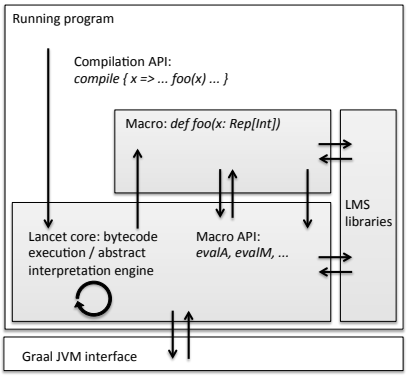

Figure 4. Lancet system overview.

1.1 Contributions

The rest of this paper demonstrates the viability and usefulness of the proposed approach. We present Lancet, a JIT compiler framework for Java bytecode. Lancet interfaces with the Oracle HotSpot JVM through hooks developed for Graal [37], a JIT compiler written in Java that grew out of the Maxine meta-circular JVM [54]. Figure 4 shows an overview of the system.

Deriving Realistic Optimizing Compilers from Interpreters Implementing optimizing JIT compilers that need to interface with the running program is by far no trivial undertaking. We demonstrate that such realistic compilers can be derived systematically from simple interpreters using pre-existing metaprogramming techniques (Section 2).

- We developed Lancet by taking a bytecode interpreter from the Graal project. We translated this interpreter to Scala and turned it into a bytecode compiler by adding staging, using the LMS (Lightweight Modular Staging) framework [40], a library-based program generation approach. This fact provides compelling evidence that the staged-interpreter approach can scale to compilers for complex non-toy languages like JVM bytecode (Section 2.1).
- We show how adding abstract interpretation to this staged interpreter turned the simple compiler into an optimizing compiler. This demonstrates that realistic optimizing compilers that perform aggressive constant propagation through object fields, specialization, etc. can be derived systematically from interpreters (Section 2.2).
- We demonstrate further that the staged interpreter approach offers a natural extension point to extend the core compiler functionality via JIT macros. In the case of Lancet, JIT macros are implemented using LMS as well and can redefine arbitrary method calls with custom code generation. Moreover, Lancet JIT macros can not only perform front-end desugaring but are integrated with the optimizations of the core compiler (Section 2.3).

Putting Surgical JIT Facilities to Use We demonstrate how JIT macros and explicit compilation can be used to define rich higher level APIs and support a number of important and challenging use-cases (Section 3):

- Program specialization (Section 3.1). Controlled specialization is key to remove abstraction overhead from generic code, and obtaining predictable high performance. We demonstrate the effect of runtime specialization on the CSV processing example from Section 1 and show that it outperforms statically compiled C code. In addition, we discuss controlled inlining, loop unrolling, as well as code caching and on-demand compilation.
- Speculative optimization (Section 3.2). Sometimes up-front specialization is not possible because the program behavior may change right in the middle of a compiled piece of code. This is where speculative optimization and deoptimization comes in. We show a custom speculation strategy for handling big-integer overflows. We also discuss search trees that speculate on a mostly constant tree structure but support infrequent structural changes through deoptimization.
- Just-in-time program analysis (Section 3.3). Garbage collection may be the biggest source of nondeterminism in managed runtimes. Explicit compilation can provide guarantees that generated code does not allocate data on the heap, and hence no garbage needs to be collected later. We can also implement more general *just-in-time* program analyses. JIT analysis applies static analysis algorithms at runtime and profits from the same increased precision as JIT compilation, without the overhead of dynamic analysis based on instrumented code.
- Smart libraries and DSLs (Section 3.4). Embedded compiled DSLs that expose an LMS-based staged programming interface to the end-user have been shown to yield orders of magnitude speedups over regular Scala code [41]. Rich toolchains have been built around LMS and the Delite DSL framework [4, 8, 33, 42] that provide optimizations beyond the capabilities of general purpose compilers, including parallelization and execution on heterogeneous devices (multi-core CPUs, GPUs and clusters). We demonstrate that we can hook up these toolchains as backends with Lancet to build smart libraries that perform on par with explicitly compiled DSLs. We also show that we can build accelerator macros retroactively for existing libraries such as collections. Even though accelerator macros are implemented in Scala, they can be used from other JVM languages.
- Cross compilation (Section 3.5). Targeting LMS as a back-end means we can also cross-compile byte code to other targets, for example CUDA, JavaScript, or SQL. In the case of SQL for language-embedded queries, it is possible to avoid common problems of heterogeneous environments like duplicate execution of queries and query avalanches by taking the context of the query into account during translation.

2. Deriving Realistic Compilers from Interpreters

It has long been known that specializing an interpreter to a program yields a compiled program (the first Futamura projection [16]), and that staging an interpreter, which enables specialization for arbitrary programs, effectively turns an interpreter into a compiler [1]. Here we give a slightly different intuition.

2.1 Interpreter + Staging = Compiler

We start with a simple toy language that contains just arithmetic operations, variables, and while loops. The syntax and denotational semantics are given in Figure 5. From the denotational semantics, it is easy to read off an interpreter, taking the liberty to implement object language while loops using while loops of the host language:

```
type Store = Map[String,Int]
type Val   = Int
def eval(e: Exp)(st: Store): Val = e match {
  case Const(c)    => c
  case Var(x)      => st(x)
  case Plus(e1,e2) => eval(e1)(st) + eval(e2)(st)
}
def exec(s: Stm)(st: Store): Store = s match {
  case Assign(x,e)  => st + (x -> eval(e)(st))
  case While(e,s)   =>
    var st1 = st; while (eval(e)(st1) != 0) st1 = exec(s)(st1); st1
}
```

However, we can also read the semantics as a recipe for translation that describes how to map object language terms to meta

language terms. In the case of denotational semantics, the meta-language is mathematics. The key enabling property is compositionality: the semantics of each term are described in terms of the semantics of its proper subterms. Thus, there is a clear phase distinction between object and meta language and translation can proceed via structural recursion. The key idea of turning interpreters into compilers via staging is to treat an interpreter itself, not a more abstract semantics specification, as a recipe for translation, by replacing the domain of values and its associated operations with a domain of program expressions that are constructed and composed by the staged interpreter.

In Scala, we use the LMS (Lightweight Modular Staging) framework [40], which provides a type constructor `Rep[T]` to distinguish staged expressions: whereas an expression of type `T` (where `T` could be for example `Int` or `String`) is evaluated right away, an expression of type `Rep[T]` denotes a piece of generated code that computes a value of type `T` when executed later. Staged `Rep[T]` expressions come with all operations of type `T`, but will represent these operations as code. For example, the staged addition on `Rep[Int]` values is defined as follows:

```
def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int] = reflect(s"$x+$y")
```

We turn our simple interpreter from above into a compiler by changing the types of values and stores:

```
type Store = Rep[Map[String,Int]]
type Val   = Rep[Int]
```

In this simple case, we do not need to change anything else, assuming that staged versions of all the operators used in the interpreter are provided. Given a program, and a staged expression that evaluates to a store, our interpreter now produces code that performs the computation of the given program. We have thus turned our interpreter into a compiler!

Scaling To a Realistic Language The example language in the previous section was decidedly simple and we are unaware of any "serious" compiler developed previously using this approach. However, this is precisely what we did to implement Lancet. We ported an existing bytecode interpreter from the Graal/Truffle project from Java to Scala and added staging annotations using LMS to turn the interpreter into a compiler.

The main architecture of the bytecode interpreter is shown in Figure 6. Figure 7 summarizes the delta of changes we applied to turn the interpreter into a compiler. The core data structure to store interpreter state is the `Frame` class, which contains an array of local objects and a pointer to a parent frame. Class `InterpreterFrame` extends `Frame` with a link to a JVM method and abstractions for mapping an operand stack onto the frame's local variable slots. The basic structure of the interpreter corresponds to that of a CESK machine [15], with the linked `InterpreterFrame` objects pointed to by variable `globalFrame` representing the control, environment, and continuation components of the interpreter's state. The store component is modeled directly by the JVM heap, which is accessed through the `Runtime` interface. The main interpreter routine `loop` fetches the current frame, executes bytecodes one by one via `executeInstruction`, and dispatches control transfers by calling exec with a suitably setup frame.

Syntax:

```
x, v                              identifiers, values
st: x -> v                        stores
e ::= v | x | e + e | ...         expressions
s ::= x = e | while (e) s | ...   statements
```

Semantics:

```
E[[ . ]]: e -> st -> v
S[[ . ]]: s -> st -> st

          E[[ c ]](st) = c
          E[[ x ]](st) = st(x)
    E[[ e1 + e2 ]](st) = E[[ e1 ]](st) + E[[ e2 ]](st)

      S[[ x = e ]](st) = st[ x -> E[[ e ]](st) ]
S[[ while (e) s ]](st) = f(st)
          where f(st') = if (E[[ e ]](st') != 0) f(S[[ s ]](st'))
                         else st'
```

Figure 5. Staged interpreter example: syntax and denotational semantics for a toy language.

To turn this interpreter into a compiler, only three aspects had to be changed (see Figure 7). First, the array that holds local data in class `Frame` changed its type from `Array[Object]` to `Array[Rep[Object]]`. This type denotes a present-stage array that contains staged objects. In other words, the stack layout and all associated operations, like pushing or popping operands, are resolved in a fully static way. This is a key difference to the previous section, where we had a global store that was fully dynamic, i.e. of type `Rep[Map[String,Int]]`. Most importantly, the static execution includes all the bytecode dispatch logic; only the actual stack contents and the low-level primitive or heap operations remain dynamic. Following this type change, the method signatures in classes `Frame`, `InterpreterFrame` and `Runtime` had to be changed to add `Rep[T]` types to expressions that will hold dynamic values. The second change was to introduce staged operations on primitives and for heap accesses (class `Staging`). The actual implementation differs from Figure 7 in that it creates IR nodes instead of strings.

The third change was to switch the main execution loop from an "execute next frame until done" model to one that may add multiple control flow branches to a worklist and compiles blocks on that worklist until no more work remains.

2.2 Compiler + Abstract Interpreter = Optimizer

While adding staging gave us a simple compiler with very low effort, it did not give us an *optimizing* compiler right away. How can we add compiler optimizations if the starting point is just an interpreter? The key idea is to combine the staged interpreter, which performs code generation, with an abstract interpreter, which performs program analysis.

We start by introducing a domain of abstract values alongside the staged values that represent generated, residual code. Our main goal is to perform aggressive constant propagation and specialization in the style of partial evaluation, so we distinguish primitive constants, static objects, partially static objects, and unknown dynamic values:

```
abstract class AbsVal[T]
case class Const[T](x: T) extends AbsVal[T]
case class Static[T](x: T) extends AbsVal[T]
case class Partial[T](f: Map[JavaField,Rep[Any]]) extends AbsVal[T]
case class Unknown[T]() extends AbsVal[T]
```

We further introduce a mapping from staged values to abstract values that allows us to attach abstract information (`AbsVal[T]`) to pieces of generated code (`Rep[T]`). This information is uniformly accessed through an `evalA` function:

```
def evalA[T](x: Rep[T]): AbsVal[T]
```

Function `evalA` can compute the abstract information in many different ways but provides a uniform interface that enables us to implement individual optimization rewrites in a localized fashion without knowledge about how the information is gathered.

To give an example, a simple constant folding rule for integer addition can be implemented like this:

```
override def infix_+(x: Rep[Int], y: Rep[Int]) =
(evalA(x), evalA(y)) match {
  case (Const(x),Const(y)) => liftConst(x+y)
  case _ => super.infix_+(x,y) }
```

If both operands are constants, we return a constant as the result of the plus operation (`liftConst`). If not, we perform a super call, which will create an appropriate IR node for the operation.

```
class Frame(numLocals: Int, parent: Frame) {
  val locals: Array[Object] = new Array[Object](numLocals)
  def setObject(index: Int, value: Object): Unit = locals(index) = value;
  def setInt(index: Int, value: Int): Unit       = locals(index) = value.asInstanceOf[AnyRef]
  def getObject(index: Int): Object              = locals(index)
  def getInt(index: Int): Int                    = locals(index).asInstanceOf[Int]
}
class InterpreterFrame(method: JavaMethod, parent: InterpreterFrame) extends Frame(method.getMaxLocals() + method.getMaxStackSize(), parent) {
  private var tos: Int = 0;
  def pushObject(value: Object): Unit = { tos += 1; setInt(tos, value) }
  def pushInt(value: Int): Unit       = { tos += 1; setInt(tos, value) }
  def popObject(): Object             = { val value = getObject(tos); tos -= 1; return value }
  def popInt(): Int                   = { val value = getInt(tos);    tos -= 1; return value }
}
class Runtime {
  def setFieldObject(value: Object, base: Object, field: JavaField): Unit = unsafe.putObject(resolveBase(base, field), resolveOffset(field), value)
  def setFieldInt(value: Int, base: Object, field: JavaField): Unit       = unsafe.putInt(resolveBase(base, field), resolveOffset(field), value)
  def getFieldObject(base: Object, field: JavaField): Object              = unsafe.getObject(resolveBase(base, field), resolveOffset(field))
  def getFieldInt(base: Object, field: JavaField): Int                    = unsafe.getInt(resolveBase(base, field), resolveOffset(field))
}
class Interpreter {
  def executeInstruction(frame: InterpreterFrame, bs: BytecodeStream) = bs.currentBC() match {
    case Bytecodes.IADD           =>  frame.pushInt(frame.popInt() + frame.popInt())
    case Bytecodes.LADD           =>  frame.pushLong(frame.popLong() + frame.popLong())
    case Bytecodes.FADD           =>  frame.pushFloat(frame.popFloat() + frame.popFloat())
    case Bytecodes.DADD           =>  frame.pushDouble(frame.popDouble() + frame.popDouble())
    case Bytecodes.GETFIELD       =>  getField(frame, nullCheck(frame.popObject()), resolveField(frame, bs.currentBC(), bs.readCPI())
    case Bytecodes.PUTFIELD       =>  putField(frame, bs.readCPI());
    case Bytecodes.INVOKEVIRTUAL  =>  exec(invokeVirtual(frame, bs.readCPI()))
    case Bytecodes.INVOKESPECIAL  =>  ecec(invokeSpecial(frame, bs.readCPI()))
  }
  def getField(frame: InterpreterFrame, base: Rep[Object], field: JavaField): Unit = field.getKind() match {
    case Kind.Object  => frame.pushObject(runtimeInterface.getFieldObject(base, field));
    case Kind.Int     => frame.pushInt(runtimeInterface.getFieldInt(base, field));
  }
  def invokeVirtual(frame: InterpreterFrame, cpi: Char): Unit = {
    // resolve call target and alloc caller frame with parent=frame
  }
  var globalFrame: InterpreterFrame
  def exec(frame: InterpreterFrame) { globalFrame = frame }
  def loop() = while (globalFrame != null) { /* exec instructions in current frame */ }
}
```

Figure 6. Core of the Java bytecode interpreter ported from the Graal/Truffle project (with minor modifications).

```
class Frame(...) {
  val locals: Array[Rep[Object]] = new Array[Rep[Object]](...)
  def setObject(index: Int, value: Rep[Object]): Unit = ...
  def getObject(index: Int): Rep[Object]              = ...
}
class InterpreterFrame(...) {
  def pushObject(value: Rep[Object]): Unit = ...
  def popObject(): Rep[Object]             = ...
}
class Runtime {
  def setFieldObject(value: Rep[Object], base: Rep[Object], field: JavaField): Unit = ...
  def getFieldObject(base: Rep[Object], field: JavaField): Rep[Object]             = ...
}
class Staging {
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]                                    = reflect[Int](s"$x + $y")
  def infix_*(x: Rep[Int], y: Rep[Int]): Rep[Int]                                    = reflect[Int](s"$x * $y")
  object unsafe {
    def putObject(base: Rep[Object], offset: Rep[Long], value: Rep[Object]): Rep[Unit] = reflect[Unit]  (s"unsafe.putObject($base, $offset, $value")
    def getObject(base: Rep[Object], offset: Rep[Long]): Rep[Object]                   = reflect[Object](s"unsafe.getObject($base, $offset")
  }
}
class Interpreter {
  var worklist: Set[InterpreterFrame]
  def exec(frame: InterpreterFrame) { worklist += frame }
  def loop() = while (worklist.nonEmpty) { /* exec instructions in current frame */ }
}
```

Figure 7. Turning the interpreter into a (simple) compiler by adding staging annotations (Rep[T] types) in key places. Method bodies remain mostly unchanged; only primitive operations require explicit code generation.

A slightly more complicated case is short-cutting object field reads:

```
override def getFieldObject(base: Rep[Object], field: JavaField]) =
evalA(base) match {
  case Static(x) if field.isFinal => staticRuntime.getFieldObject(x,field)
  case Partial(fields) if fields.contains(field) => fields(field)
  case _ => super.getFieldObject(base, field)
}
```

Here, we rely on abstract store information. We maintain an abstract store as a global data structure to reason statically about heap objects (we elide the details). Method `evalA` may return a static object that was created outside of the code to be compiled. For a `final`, i.e. immutable, field, we just read and return its value. But `evalA` may also return a `Partial` object, which consists of a list of abstract fields. `Partial` objects denote either objects allocated in the compiled code or static objects that have been modified by generated code. If we have an entry for the field we are looking for, we return that, and if not, we again perform a super call to create an IR node for the `getFieldObject` operation.

Finally, to make our analysis and optimizations flow-sensitive, we need to modify the main execution loop to perform fixpoint iteration and compute least upper bounds on abstract stores at control flow joins. For the sake of explanation, we show the behavior as pseudocode for while loops in the context of the toy language from above instead of unstructured control flow as we have in bytecode:

```
type Store:    Rep[Map[String,Int]]    // residual code
type AbsStore: Map[String,AbsVal[Int]] // abstract info
def exec(s: Stm)(stg: Store, sta: AbsStore) = s match {
  case While(e,s) =>
    def loop(sta1: AbsStore) = {
      val (stg2,sta2) = exec(s)(stg,st1)
      if (sta2 != sta1) // not converged
        loop(sta1 lub sta2)
      else (stg2,sta2)  // converged
    }
    val (stg2,sta2) = loop(sta)
    reflect(s"while (${ eval(e) }) $stg2)
}
```

In short, we iterate until the abstract store at loop entry has converged to a fixpoint and return the last generated expression for the loop body. Using this straightforward combination of a staged interpreter with an abstract interpreter, we perform dataflow analysis interleaved with transformation, and are able to maintain optimistic

assumptions around loops in the style of [34]. This approach can also be viewed as a (very simple) kind of supercompilation [51], as we essentially use fixpoint iteration to generalize program fragments directly, instead of first generalizing values in an abstract dataflow domain and then transforming the program afterwards. Overall, we follow closely the LMS-based extensible compiler architecture that was proposed in previous work [41], with adaptations as required by the comparatively lower-level nature of Java bytecode.

2.3 JIT Macros as Extension Points

Since the core of Lancet is implemented using LMS, it can be easily extended by other pieces of multi-stage code, even if they are defined by the running program itself. In fact, Lancet provides explicit hooks to register callbacks – JIT macros – that are called whenever a certain method call occurs in bytecode under compilation.

To support these possibly user-defined macros, the essential mechanism is to override the `invokeVirtual` method and look for methods that have associated callbacks registered. If such a method is found, we dispatch to the callback instead of compiling the method body and whatever staged code fragment the callback returns will replace the method call. Since macros are called from the main fixpoint transformation and dataflow analysis loop, they will always see up-to-date abstract information and may be invoked multiple times as abstract values are refined.

Using this mechanism, we can for example implement method `freeze` from Section 1, which evaluates its argument at JIT-compile time. The user-facing method is declared with the signature of the identity function:

```
object LancetLib {
  def freeze[A](x: => A): A
}
```

The corresponding macro is defined as follows, with the same name and signature but adding `Rep` types:

```
object LancetMacros {
  def freeze[A](f:Rep[()=>A]): Rep[A] = liftConst(evalM(f)())
}
```

It is installed like this:

```
Lancet.install(classOf[LancetLib.this],LancetMacros)
```

Macros can easily interface with the compiler internals, for example to inspect their arguments or the dynamic scope of their invocation. Lancet provides a number of handy support functions. The core of `freeze` is a call to `evalM` (evaluate materialized), which tries to evaluate a `Rep[T]` value back to `T`. Its implementation first determines the precise class of the expression, allocates an uninitialized object of that class and then recursively materializes all fields of the object:

```
def evalM[A](x: Rep[A]): A = evalA(x) match {
  case Static(x)   => x
  case Partial(fs) =>
    val obj = staticRuntime.allocateInstance(evalClass(x))
    fs.foreach(f=>staticRuntime.putObject(obj,f.offset,evalM(f.value)))
    obj
}
```

If any of these steps fails, an error is raised. In the case of `freeze`, the argument closure is materialized and called directly to produce a constant result. In the same way, any other user-level method can serve as a marker or high-level API to JIT functionality. We will see more examples below.

3. Putting Surgical JIT Facilities to Use

3.1 Program Specialization

Controlled specialization is key to remove abstraction overhead from generic code and obtaining predictable high performance, as motivated already by the CSV processing example in Section 1. Before going into further implementation details, we present performance numbers for this application in Figure 1. We compare versions of the code in plain Scala, C++ and Scala specialized with Lancet reading input files from 23 to 92 MB with 20 columns, out of which 10 are accessed by name. The Lancet compiled code handily beats the plain Scala and C++ versions by a factor of 2. These measurements include file IO as well as expensive library calls like `String.split`. It is worth noting that C++ and Lancet performance is almost unaffected by the input data size, whereas HotSpot compiled Scala code is slower than C++ for small sizes and faster for large inputs.

	Input size: 23	46	69	92 MB
CSV Reading				
C++	1.00	0.98	0.99	0.99
Scala Library	0.92	0.91	1.16	1.25
Scala Lancet	2.19	2.20	2.91	2.65

Table 1. Effects of JIT specialization on the CSV processing app from Section 1. Speedup numbers are relative to hand written C++ and normalized for input size.

Controlled Inlining In automatic JIT compilers, inlining decisions are a major source of non-determinism. For example, higher-order methods like `foreach` pose difficulties for call-site based profiling:

```
def foreach(f: A => Unit) = {
  var it = this.iterator
  while (it.hasNext) f(it.next()) }
```

In many cases, profiling will identify the call site of `f` inside `foreach` as hot, but since there may be many invocations of `foreach` in a program, the same number of different call-targets for the closure `f` exist [12]. The exact JIT behavior depends on many factors and is hard to predict. What we really want instead is to inline `foreach` itself at all of its callers to relate each particular closure allocation with its invocation.

Lancet takes a different approach and will always try to inline non-recursive functions, unless instructed otherwise. Of course this is not always the right choice, and therefore inlining can be controlled by high-level directives `inlineAlways`, `inlineNonRec` (default, for non-recursive) and `inlineNever`, all of which are implemented as JIT macros. For example:

```
inlineAlways {  xs.foreach(i => foo(i))  }
```

Here, all methods calls, including recursive ones, will be inlined in the dynamic scope of `foreach`, `foo` and its callees unless superseded by another, closer, directive.

In addition to attaching directives to a full dynamic scope, we can use higher-order directives `atScope` and `inScope`, which are again implemented as JIT macros, to trigger directives only once a method that matches a certain pattern is entered. For example:

```
inlineAlways {
  atScope("^java.io.")(inlineNever) {
    // no IO methods will be inlined
  }}
```

In this case, all method calls in the dynamic scope will be inlined, until a method on an object from the `java.io` package is hit. At this point inlining stops and a method call in emitted. Using `inScope` instead of `atScope` would trigger one level down, inside the first call to a matching method. Compared to explicit inline annotations on particular methods, the scope driven approach is more expressive because decisions can be controlled in a non-local and compositional way.

Loop Unrolling with Dynamic Scope Loop unrolling is a key transformation in performance oriented code. For example, we may want to implement an unrolling loop combinator `ntimes`:

```
ntimes(3) { i => println(i) }
```

The desired effect is to produce the following code:

```
println(0); println(1); println(2)
```

When implementing a suitable macro for ntimes, we can use evalM to materialize the constant trip count. We could also use evalM to materialize the closure that defines the loop body, but here, full materialization is actually not desired because the loop body may well refer to non-static data. Therefore, we use a different operator, funR, which turns a staged function Rep[A=>B] into a function on staged values Rep[A]=>Rep[B]. Invoking the result of funR will inline the function body, substituting the formal parameter with the argument passed. Here, we use funR to obtain an unfoldable representation of the closure that defines the loop body. Putting things together, we arrive at the following macro definition:

```
def ntimes(n: Rep[Int])(f: Rep[Int=>Unit]): Rep[Unit] = {
  val body = funR(f)
  for (i <- 0 until evalM(n))  // staging-time for-loop
    body(liftConst(i))
}
```

However, we can go a step further. Making unrolling decisions separately for each loop limits modularity. We may want to put a loop inside a function,

```
def loopy(x: Int) = ntimes(x) { i => println(i) }
```

which is called elsewhere, and we would like to make an unrolling decision at the call site:

```
unrollTopLevel { loopy(7) }
```

Similar to the inlining directives, we can implement this pattern using dynamic scope, relying on the fact that Lancet will inline loopy at the point where it is used. We define the following macros:

```
var shouldInline = false
def unrollTopLevel(f:Rep[()=>Unit]) = {
  val save = shouldInline; shouldInline = true
  try funR(f)() finally shouldInline = save
}
def ntimes(n: Rep[Int])(f: Rep[Int=>Unit]): Rep[Unit] = {
  // unroll only if shouldInline is true,
  // and set shouldInline to false when unfolding f
}
```

As we can see, dynamically scoped variables enable communication between nested macro invocations.

Instead of using unrollTopLevel directly at the call site of loopy we can use atScope or inScope to control unrolling from further up the chain:

```
atScope("loopy")(unrollTopLevel) { ... }
```

Now, for any invocation of loopy within the dynamic scope of the block, the enclosed loop will be unrolled. We believe that this ability of attaching directives to a dynamic scope instead of individual methods or loops provides an advantage in expressiveness that is hard to overstate.

Code Caching and On-Demand Compilation Specializing code on demand is a truly "just in time" use-case. Let us consider the general case of a two-argument function calc(x:Int,y:Int). We would like to implement a variant of calc that creates specialized versions for particular values of x on demand and on the fly, without exposing this fact to the user of the function.

We implement a code cache, and a function calcJIT with the same interface as calc:

```
val cache = new WeakHashMap[Int,Int=>Int]
def calcJIT(x:Int,y:Int)=
  cache.getOrElseUpdate(x, compile(z => calc(x,z)))(y)
```

If we find a specialized version for a given x we will use that. Otherwise, a new version is compiled and cached for later use.

Instead of relying on some VM-internal black-box behavior that is not accessible to the program and might even change in the next release, we have complete control over how code is compiled and re-used. For example, we could easily extend our cache with a custom eviction policy. In any event, this implementation of calcJIT guarantees that it will execute a code path in which x is a compile-time constant.

In some cases, this guarantee might be our primary interest, but in other cases it might not. For example, we might care more about amortizing compilation cost. This is a different concern, so no single black-box solution is sufficient. With a programmable JIT, we can easily accommodate. In our new version, calcHOT, we choose not to specialize calc eagerly for all values of x, but only after a certain value becomes "hot". To achieve this, we add an additional data structure to store profiling information:

```
val profile = new HashMap[Int,Int] withDefault 0
def calcHOT(x: Int, y: Int) = if (profile(x) < threshold) {
  profile(x) += 1; calc(x,y)
} else calcJIT(x,y)
```

With just a little bit more effort we could add background compilation by submitting the actual compilation as a task to a worker thread.

Of course implementing code caches and profiling afresh for individual methods is not very scalable in terms of effort, but working in a high-level language, we can easily generalize and provide combinators like makeJIT:

```
def makeJIT[A,B,C](f:(A,B)=>C) = {
  val cache = new WeakHashMap[A,(A,B)=>C]
  (x,y) => cache.getOrElseUpdate(x, compile(z => f(x,z)))(y)
}
val calcJIT = makeJIT(calc)
```

The implementation of makeHOT for the profiling case is analogous.

3.2 Speculative Optimization

In addition to straight compilation and program specialization, we want to put the dynamic VM capabilities to good use, namely speculative optimization, and the facility to discard optimized code on the fly if it turns out that compilation assumptions were too optimistic (deoptimization).

Using JIT macros, which provide access to the internal compiler state, we can define a range of high-level operators to convey speculation directives. To start with a simple example, method likely may tell the compiler to assume that a test will likely succeed:

```
if (likely(cond)) { ... } else { ... }
```

The VM could come to the same conclusion by profiling the actual execution and recording the branches taken, but even in this case, likely can serve as a contract, and cause the VM to signal a warning if profiling suggests that the test is actually not likely. To give the compiler even tougher guidelines, we introduce method speculate:

```
if (speculate(cond)) { ... } else { ... }
```

This will instruct the compiler to assume that the test will always succeed and optimize the method accordingly, replacing the conditional by its then branch. Of course Lancet cannot be certain that this assumption always holds so it must insert a guard that, if the check fails, takes a slow path and drops into interpreter mode for this particular execution. A variant of this approach is stable:

```
if (stable(cond)) { ... } else { ... }
```

The assumption here is that the condition may change permanently but infrequently. If the guard fails, the method is recompiled on the fly with the new value instead of dropping into interpreter mode.

Implementing Deoptimization / OSR / Continuations Internally, we can express operations like speculate and stable using lower-level primitives slowpath and fastpath: explicit operations to interpret and compile the current continuation, i.e. to switch to interpreted or freshly-compiled mode at the current point of program execution. Method speculate uses slowpath like this:

```
def speculate(x: Boolean) =
  if (x) true else { slowpath(); false }
```

Similarly, we can implement stable using fastpath and freeze:

```
def stable(x: => Boolean) = {
  val c == freeze(x)
  if (x == c) c else { fastpath(); x }
}
```

Of course we can apply the same technique to implement variants that operate on types other than `Boolean` or perform more complex check patterns.

In VM parlance, `slowpath` and `fastpath` perform on-stack-replacement (OSR), in PL parlance, they discard the current continuation and replace it with an interpreted or freshly compiled version of it. We implement on-stack-replacement using the operator `shiftR`, which provides a macro-level variant of the `shift` delimited control operator [14] with type `(Rep[A=>B]=>Rep[B])=>Rep[B]`. Invoking `shiftR` will pass the current continuation, up to the nearest enclosing `resetR` delimiter, to the argument closure. Internally, the continuation is represented by a linked chain of `InterpreterFrame` objects.

Implementing a user-facing `shift` operator as a macro is straightforward. We just unfold `f` with the continuation as its argument:

```
def shift[A,B](f: Rep[(A=>B)=>B]): Rep[B] = shiftR { k => funR(f)(k) }
```

User-facing `shift` and `reset` control operators have no direct relation to speculative optimization but go beyond the functionality that is usually available on JVMs. Delimited continuations can be used to implement all kinds of advanced control structures like coroutines, generators or asynchronous callbacks.

To implement `slowpath` and `fastpath`, we first convert the continuation to a function that is either interpreted or compiled and then we invoke it:

```
def slowpath(): Rep[Unit] = shiftR { k => interpret(k)() }
def fastpath(): Rep[Unit] = shiftR { k => compile(k)() }
```

The process for both versions is similar: We traverse the linked `InterpreterFrame` objects, and for each frame we emit code to reconstruct a corresponding frame object at runtime. Naturally, all local data in the versions created at runtime will be constants. After the code that reconstructs the frame state, we emit a call to invoke an interpreter or compiler instance with the newly created state. By using `shiftR`, we have already aborted the current continuations and thus achieve the desired effect of on-stack-replacement (OSR) using a few high-level combinators. It is worth pointing out that we can in principle compile or interpret arbitrary chains of frames, as long as return types match. These chains do not have to be created by actual call chains resulting from bytecode. This gives us considerable freedom for deoptimization because we just need to create *some* valid target we can "deoptimize into".

Safe and Efficient Numeric Overflow Speculative optimization is particularly useful for programs that do not have a clear stage distinction, i.e. where up-front specialization is infeasible because it is not clear when switching from one specialized version to another will be required. For example, we may want to switch from machine-size integers to `BigInteger` objects in case of an overflow. But overflows may happen anywhere, even inside a loop, so we cannot perform type-specialization ahead of time:

```
var prod = 1
while (i < n) {  prod = prod * i; i += 1  } // overflow for large n
```

This is a prime use case for speculative optimization and deoptimization: we assume optimistically that no overflow will occur and specialize the code for machine size integers. When an overflow does occur, we back out and drop into interpreted mode or recompile the code on the fly using on-stack-replacement. Here is our definition of overflow-safe integers:

```
abstract class SafeInt { def toBigInt: BigInteger }
case class Small(x: Int) extends SafeInt { ... }
case class Big(x: BigInteger) extends SafeInt { ... }
def SafeInt(x: Long) =
  if (x >= Int.MinValue && x <= Int.MaxValue) Small(x.toInt)
  else { slowpath(); Big(new BigInteger(x)) }
def infix_+(x: SafeInt, y: SafeInt): SafeInt = (x,y) match {
  case (Small(x),Small(y)) => SafeInt(x.toLong + y.toLong)
  case _ => slowpath(); Big(x.toBigInt.add(y.toBigInt))
}
```

The internal representation can switch from small to big integers, but the `BigInteger` case is treated as slow path. The key benefit of dropping into interpreted mode is that all compiled operations operate on `Int` values. Compiled code will never create `Big` objects, so all checks whether inputs are `Big` in subsequent operations can be eliminated. What remains are the overflow checks in `SafeInt`.

Exploiting Stable Structure in Trees or Graphs Speculative optimization can be useful in more complicated settings as well. Consider, for example, a dictionary implemented as a search tree. In many cases, reads dominate by orders of magnitude and updates to the key set are very rare. To increase performance, we can compile the lookup operation, i.e. map the search tree (represented as datatype) to branching code that implements the decision logic directly. Let us assume that we process some incoming data and want to increment a counter for a certain key. We can speculate that almost all key lookups will succeed, but we also need to handle the uncommon case where a key is not yet present in the dictionary. In this case we need to update the tree structure, and invalidate and recompile the lookup code.

There are several design decisions about the granularity of speculation. A fine-grained option is to define the child pointers in the tree as stable references, generalizing the `stable` operator from above to an `@stable` annotation for variables and fields:

```
class Tree(var key: Int, var value: Int,
           @stable var left: Tree, @stable var right: Tree)
```

This means that parts of the tree can be invalidated but the lookup code will only be recompiled once an invalid path is hit. There will also be guard checks for each node. Another option is to declare only the root pointer of the tree as stable and produce a new tree on each update. This reduces the number of guards but makes invalidation more coarse-grained.

Another interesting use case are observer networks of reactive dependencies. Again, structural updates are relatively infrequent and performance is dominated by processing on a stable version of the network.

3.3 Just-In-Time Program Analysis

Controlling Allocation / GC While we can control the behavior of JIT compilation, it can be argued that the memory subsystem and, in particular, garbage collection occurring at inopportune moments, are the biggest sources of nondeterminism in a VM. However, ultimately memory allocation is under the control of the JIT compiler as well. If a JIT compiler is able to bound the lifetime of objects, it can map object fields to local variables instead of allocating the object on the heap, which eliminates the need for garbage collection when the object is no longer used. Unfortunately, the current approaches based on simple escape analysis interact in intricate ways with inlining and other optimizations, which makes the outcome very unreliable again.

Making JIT compilation explicit, however, gives us an opportunity to relay warnings or error messages to the program. Combined with controlled inlining, dead code elimination and aggressive partial evaluation that is able to propagate static values through object fields, we can implement a directive `checkNoAlloc { ... }` that will produce a JIT compile error when it is unable to replace any heap allocation with local fields inside its dynamic scope. In addition, the code must not contain any deoptimization points or calls to external code not compiled with this directive. If compilation succeeds without error, we get a guarantee that the compiled code will never allocate objects on the heap and hence, no garbage will need to be collected later.

Taint Analysis / Secure Information Flow We can also implement more general *just-in-time* program analyses that are not directly related to optimizations. The idea of JIT analysis is to apply static analysis algorithms at runtime, in order to profit from the

```
// OptiML library implementation
class OptiMLCompanion {
  type DM = DenseMatrix[Double]; type DV = DenseVector[Double]
  def sum(start: Int, end: Int, size: Int)(block: Int => DV): DV = {
    val acc = new DenseVector[Double](size, true)
    for (i <- start until end) {
      acc += block(i)
    }
    acc
  }
  def sumRows(m: DM): DV = {
    sum(0, m.rows, m.cols) { i => m.row(i) }
  }
}
// OptiML accelerator macros
object OptiMLMacros extends ClassMacros {
  val targets = List(classOf[OptiMLCompanion])
  def sum(self: Rep[OptiMLCompanion], start: Rep[Int], end: Rep[Int],
    size: Rep[Int], block: Rep[Int => DV]): Rep[DV] = {
      val block1 = funR(block)
      new DeliteOpMapReduce[Int,DV] {
        def zero = DenseVector[Double].empty
        def reduce = (a,b) => a += b
        def map = x => block1(x)
        ...
      }
  }
}
```

Figure 8. OptiML library and macro implementation, using DeliteOpMapReduce.

same increased precision as JIT compilation, due to preexisting dynamic data which the analysis can treat as static. At the same time, JIT analysis is more efficient than dynamic analysis based on executing instrumented code.

One interesting use-case is taint analysis for secure information flow, with the goal of tracking what happens to user input. All user input is considered tainted. Anything possibly derived from tainted data is also considered tainted. We then want to check for a certain piece of code that it does not leak any tainted data, including or excluding basing branching decisions on tainted data which could e.g. lead to observable timing differences.

In order to implement this analysis, we need to extend the core Lancet implementation with an additional abstract interpretation lattice to model taint state, and operations to propagate taint information. We treat the result of all operations as tainted unless we can prove otherwise. The implementation is analogous to the internal analyses that are the basis for Lancet's constant propagation and partial evaluation engine. Moreover, taint analysis will be performed at the same time as the other analyses and can profit from increased precision due to constant propagation and other optimizations.

3.4 Active Libraries and Embedded DSLs

Lancet's macro facility enables us to hook into existing LMS-based toolchains like the Delite DSL framework [4, 8, 33, 42] directly from ordinary Scala libraries. In this section, we demonstrate that a library-only implementation of OptiML, a deeply embedded DSL for machine learning [46], can be combined with Lancet macros and Delite to construct an *active library* that performs as well as the original deeply embedded DSL. We also show that we can use Lancet and Delite to transparently accelerate a small existing general-purpose Scala program.

Building Active Libraries In this experiment, we implemented a scaled down version of OptiML as a pure Scala library. We used the library to implement two existing OptiML applications, *k*-means clustering and logistic regression. The library applications contain no `Rep` annotations, interoperate smoothly with IDEs, and are easy to debug, since they are regular Scala. We then separately implemented Lancet macros for the OptiML library that invoke existing deeply embedded OptiML DSL methods. Key parts of the implementation are shown in Figure 8.

	Cores: 1	2	4	8	GPU
K-Means Clustering					
Scala library	1.00	1.37	1.50	1.83	–
Lancet-Delite	4.92	8.82	17.10	24.00	50.75
Delite (Graal)	5.17	10.03	19.81	24.78	54.82
Delite (HotSpot)	5.11	9.90	18.84	22.40	54.26
C++	7.68	13.84	25.69	40.80	–
Logistic Regression					
Scala library	1.00	2.03	3.32	5.81	–
Lancet-Delite	7.79	16.16	28.99	32.69	49.79
Delite (Graal)	7.84	16.09	27.74	40.01	52.30
Delite (HotSpot)	13.02	24.77	35.27	42.88	51.05
Delite (manual opt)	23.92	45.30	86.30	133.41	–
C++	25.24	47.21	98.36	160.72	–
Name Score					
Scala library	1.00	1.71	3.08	4.36	–
Lancet-Delite	1.92	3.15	6.54	9.67	–

Table 2. Performance speedup of Lancet-Delite, stand-alone Delite, and C++ compared to pure Scala on *k*-means clustering and logistic regression. Transparently-optimized name score computation in Lancet compared to the original Scala library version.

We use Lancet to JIT compile the Scala application, which stages the application and executes the macros to construct Delite IR nodes. Finally, Delite generates and executes heterogeneous code (Scala and CUDA) for the staged block as usual. Table 2 compares the performance of this approach (Lancet-Delite) compared to the pure Scala library, stand-alone Delite (on both HotSpot 1.7 b17 and Graal revision 13489:780d53499ca2), and hand-optimized C++. The Scala library versions are run on Graal and parallelized using Scala Parallel Collections. The C++ versions are parallelized using OpenMP. For each experiment we timed the computational part of the application, ignoring initialization steps. We ran each application (with initialization) ten times without interruption and averaged the last five runs to smooth out fluctuations due to VM warmup, garbage collection and other variables.

The active library implementation of OptiML is much simpler for end users and within a small margin of the statically compiled deeply embedded version. Starting from an ordinary Scala library, we are still able to generate optimized Scala and CUDA code. In contrast, running the library implementation without the Lancet macros results in between 5-13x slower CPU performance on *k*-means and 5-8x slower performance on logistic regression. The performance improvements with Lancet and Delite come mainly from compiling away data structures (the staged versions operate only on arrays) and fusing computationally heavy loops together, resulting in less traversals and intermediate data allocations. Delite's efficient static parallelization enables good scaling across thread counts and we are also able to generate and execute CUDA kernels for both applications, resulting in additional speedup. We ran the stand-alone Delite version using both HotSpot and Graal; Graal performed slightly better than HotSpot on *k*-means and substantially worse on logistic regression, although it catches up on 8 cores, probably due to short running times. This is an example of the difficulty in predicting and understanding performance from traditional black-box JIT compilers, even with two similar VMs.

To demonstrate an approximate upper-bound on achievable performance, we also show hand-optimized, hand-parallelized C++ performance. In these versions, we manually fuse together operations and aggressively re-use memory. As a result, the application code is much less readable than the Scala code that is the starting point for the other versions. In *k*-means clustering, Lancet-Delite is within 50% of the optimized C++ and outperforms it using

the GPU. However, the logistic regression C++ implementation is faster than the compiled Delite version which is in turn much faster than Scala library implementation. This is due to manual fusion of a key part of the logistic regression algorithm (reducing a gradient vector directly into a thread-local accumulator by computing and reducing each scalar result individually) which is not yet supported by Delite. However, we believe that it is possible to perform this transformation on the Delite IR and plan to do so in future work (it is simular in spirit to existing Delite fusion and loop interchange transformations).

Accelerating Existing Libraries The previous section showed how we can use Lancet to build active libraries that combine the productivity characteristics of ordinary libraries with the performance characteristics of compiled DSLs. We demonstrate next how we can use Lancet to transparently optimize existing libraries by adding JIT macros after-the-fact that convert library calls into optimized implementations. In particular, this enables us to now use Delite as a general-purpose accelerator for arbitrary existing Java bytecode programs, through appropriate macros. In this experiment, we accelerate the following small file processing program:

```
def totalScore(sortedNames: Array[String]) = {
  val scores = names.zipWithIndex map { case (a,i) =>
    val score = a.map(c => c-64).reduce(_+_)
    (i*score).toLong  }
  scores.reduce(_+_)
}
```

This simple program computes the "score" associated with a given name based on the ascii value of the characters in the name. To accelerate this program, we add Lancet macros to replace collection operations on Scala arrays with operations on `Rep[DeliteArray[_]]`, a Delite collection that supports high performance parallel operators. We implement macros for `Array` `zipWithIndex`, `map`, and `reduce` which call their corresponding `DeliteArray` methods, similar to the OptiML macros in Figure 8. Then, as in the active library case, we invoke `compile` on the `totalScore` method, which JIT compiles the snippet by staging it and then executes it with Delite. Table 2 shows the performance improvement obtained by using these simple macros compared to running the native Scala library with Scala Parallel Collections. Even on a simple application like this, Delite obtains approximately a 2x speedup over the library version, by performing two critical optimizations: unwrapping the array of tuples into two arrays (array-of-struct to struct-of-array conversion) and fusing the `map` with the `reduce`, eliminating a large intermediate data structure.

This experiment demonstrates that Lancet is a powerful tool for accelerating existing libraries with simple front-ends by mapping them to a common back-end toolchain like Delite. Previous work has shown that mapping computation to a shared, high-level intermediate representation is a good way to obtain high performance across multiple compiled, embedded DSLs used in the same program [47]. Lancet allows us to apply this principle not only to compiled embedded DSLs but also to active libraries by dynamically and selectively mapping portions of the libraries to the common back-end.

3.5 Cross-Compilation

Using LMS as a backend means that it is not difficult to generate code for other target languages. Previous work has shown JavaScript [31] and SQL generation [52]. We briefly discuss generating code for these targets, using Lancet as a bytecode decompilation front-end.

SQL / LINQ Language embedded query functionality, such as LINQ [35] on the .Net platform, is a popular way to access external data sources. A database query can look like a filter operation on an in-memory collection:

```
val data = table[Item]("t_item")
val res = data.filter(x => x.price > 0)
println(res.count)
```

Behind the scenes, the `filter` call will be translated to a `where` clause in a SQL query, which is shipped to the database. In LINQ and other systems, this is achieved through lifting of expression trees at (static) compile time. For simple cases like the one above this works fine, but more complicated cases come with certain issues.

In most systems, the following snippet will not work:

```
def p(x:Item) = ... // defined elsewhere
val data = table[Item]("t_item")
val res = data.filter(x => x.price > 0 && p(x))
println(res.count)
```

We are calling an externally defined function from inside the `filter` closure, but only the filter closure itself is lifted as expression tree. If we were using Lancet and lifting bytecode instead of static trees this would not be a problem because bytecode is available for all functions.

Another issue is that some operations like `sum` or `count` return scalar types, not queries:

```
val data = table[Item]("t_item")
val res = data.map(x => x.price)
println(res.count)
println(res.sum)
```

Most systems will execute this query twice. If the whole snippet were part of a larger block compiled with Lancet, queries could be analyzed with respect to their context and the query result reused. Similar transformation are performed internally by Delite.

Finally, nested queries often produce "query avalanches", executing a new query for each loop iteration:

```
val data = table[Item]("t_item")
val order = table[Order]("t_order")
for (x <- data) {
  println("orders for item "+x)
  val ordersForItem = order.filter(y => y.item == x)
  println(ordersForItem)
}
```

By looking at the context of the query, however, we can first create an index

```
val ordersForItemMap = order.groupBy(y => y.item)
```

and replace the nested `filter` call by an index lookup:

```
val ordersForItem = ordersForItemMap(x)
```

Now both `data` and `order` queries can be fetched in a single trip to the DB before traversing the order data. Orders are then stored in a hash table in the program. Avalanche avoiding transformations have been studied by [11, 22, 52, 53].

JavaScript To cross-compile to JavaScript, we first define a DOM interface as a Scala library containing only abstract classes. We will use trait `JS` as a marker interface for all DOM classes:

```
object Dom extends JS {
  trait Element extends JS
  trait Canvas extends JS {
    def getContext(key: String): Context
  }
  trait Context extends JS {
    def translate(x: Double, y: Double)
    def rotate(r: Double)
    def moveTo(x: Double, y: Double)
    def lineTo(x: Double, y: Double)
    ...
  }
  object document extends JS {
    def getElementById(id: String): Element = error("not implemented")
  }
}
```

Using this API, we can for example implement a small drawing applicaton that will print Koch snowflakes on a HTML canvas element (example taken from [31]):

```
def snowflake(c: Context, n: Int, x: Int, y: Int, len: Int) = {
  c.save(); c.translate(x,y); c.moveTo(0,0); leg(n)
  c.rotate(-120*deg); leg(n);
  c.rotate(-120*deg); leg(n);
  c.closePath(); c.restore();
  def leg(n: Int) { ... }
}
```

Defining a translation from DOM library calls to actual JavaScript code is achieved by a macro that looks for method invocations on objects inheriting from JS (some details elided):

```
def invokeMethod(parent: InterpreterFrame, m: ResolvedJavaMethod) = {
  if (classOf[JS].isAssignableFrom(m.getDeclaringClass)) {
    ... reflect[Object](""+receiver+
          "."+m.getName+"("+parameters.mkString(",")+")")  ...
  } else false
}
```

With this macro support in place, we are ready to emit JavaScript code that draws pretty pictures:

Of course only core functionality of a JavaScript cross-compiler is implemented; to build a workable web application toolkit in the style of GWT [18] that includes client-server interaction much additional work would be needed.

4. Related Work

JIT compilers have made significant progress over the last two decades, and are the method of choice for implementing many modern programming languages. Where compiler optimizations based on static analysis are largely considered intractable for dynamic languages, compiling pieces of the running program in the background with optimizations based on dynamic feedback gathered during program executing is an attractive choice. Combinations of dynamic type feedback and static type inference have also been shown to be beneficial [23].

JIT Compilers Many of the key technologies used by JIT compilers today have their roots in the Self [10] and StrongTalk [7] systems. Examples are polymorphic inline caches [24], deoptimization [25], run-time type feedback for dynamic calls [26], and adaptive optimization to focus compilation on performance critical methods [27].

Modern JIT compilers for Java bytecode include the Server [38] and Client [32] compilers in the Oracle HotSpot VM, the Jalapeño compiler [3] in Jikes RVM [2], and the Graal [37, 55] compiler which evolved from the Maxine VM [54]. Google V8 [19, 20] is an advanced VM and JIT compiler system for JavaScript. The Truffle runtime system [56, 57], built on top of Graal, brings advanced dynamic optimizations to AST interpreters. PyPy [6, 39] is a JIT framework in Python that works on program traces instead of methods.

Staging, LMS, Delite Multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [48] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [9] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [40] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code. LMS draws inspiration from earlier work such as TaskGraph [5], a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [4, 8, 33, 41, 42, 47].

Partial evaluation for Java/JVM Partial evaluation [28] has been applied to Java and JVM code in a number of different contexts, for example JSpec/Tempo [43] the JSC Java Supercompiler [30] and Civet [44]. Despite their automatic nature, most of these systems also provide annotations to guide specialization decisions. DyC [21] is an annotation-directed specializer for C.

Further related work on partial evaluation adresses higher-order languages with state [50], partially static structures [36] and partially static operations [49]. Deriving compilers and virtual machines from interpreters has been studied in a formal context as functional derivations [1]. Related work on compiling via combinations of partial evaluation, staging and abstract interpretation includes [13, 29, 45].

5. Discussion

The main goals of this work are to make all aspects of JIT compilation programmable and to bring domain specific optimizations to libraries using JIT macros. The general programmer experience we seek to provide is "you get what you ask for". Thus, coming up with clever heuristics or automatisms was decidedly a non-goal. We also did not focus on compile speed, since compilation is explicitly invoked.

Putting a JIT compiler under the control of user code, and even (horrors!) executing arbitrary user code via macros may be a scary thought for security-minded people. To some of them, it may seem like we are deliberately throwing all of the VM's safety assurances out of the window. This is certainly a valid concern, but the same concern applies for example to the `sun.misc.unsafe` package, which is indispensable for performance programming but also allows unsafe reads and writes to arbitrary memory locations. Moreover, many performance critical applications run in closed environments without being faced with possibly adversarial user code. If we want to get more C programmers to adopt managed runtimes and JIT compilation, we cannot put security above all other concerns. Switching to a higher-level language and coding style will already improve the overall safety of the software produced. Finally, safety can be regained through regular access control mechanism that restrict the use of the JIT API to certain parts of the program or to certain libraries.

Some VM experts firmly believe that optimization decisions should only be based on data gathered by dynamic profiling and that programmers should not be allowed to interfere because they would likely get it wrong anyways. This idea that JIT compilers are always better at choosing profitable optimizations is not supported by evidence (see e.g. [41]). Many interesting program properties are not discernible by simple profiling alone and profile data may be imprecise or inaccurate. While inexperienced programmers will certainly get some of their optimization hints wrong, that does not imply that expert programmers are unable to make crucial use of such facilities. We believe that more research is needed into *useful* high-level optimization abstractions that capture programmer intent. Domain-specific languages and libraries are a promising avenue to reflect this high-level knowledge without requiring programmers to be explicit about details like, for example, which loop to unroll and how many times.

6. Conclusion

In this paper, we have presented Lancet, a JIT compiler framework that enables the running program to control all aspects of the JIT compilation process. We have demonstrated the usefulness of programmable JIT compilation on a number of use cases and we have presented performance results for runtime specialization that outperform C++ by 2x. Furthermore, we have demonstrated bytecode libraries with accelerator macros that achieve up to 52x speedup over plain Scala library implementations and are competitive with dedicated staged, compiled DSLs, within 20% in all cases.

Acknowledgments

The authors would like to thank the Graal/Truffle team for many insightful discussions. Christian Humer wrote the original Java bytecode interpreter from which we derived the core Lancet compiler. Tim Harris, Peter Thiemann and Nada Amin provided helpful and extensive comments on drafts at various stages, as did the anonymous OOPSLA'13, POPL'14 and PLDI'14 reviewers.

Parts of this work were funded by: DARPA Contract, SEEC: Specialized Extremely Efficient Computing, Contract # HR0011-11-C-0007; DARPA Contract-Air Force, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army contract AHPCRC W911NF-07-2-0027-1; NSF grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, NVIDIA, Huawei. Authors also acknowledge additional support from Oracle. "The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government."

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical report, BRICS, 2003.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. A. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. F. Mergen, J. E. B. Moss, T. A. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–418, 2005.

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeño jvm. In M. B. Rosson and D. Lea, editors, *OOPSLA*, pages 47–65. ACM, 2000.

[4] D.-B. Authors. Implicitly parallel distributed execution for domain-specific languages. Under submission, 2013.

[5] O. Beckmann, A. Houghton, M. R. Mellor, and P. H. J. Kelly. Runtime code generation in c++ as a foundation for domain-specific optimisation. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 291–306. Springer, 2003.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. 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*, pages 18–25. ACM, 2009.

[7] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In T. Babitsky and J. Salmons, editors, *OOPSLA*, pages 215–230. ACM, 1993.

[8] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[9] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *GPCE*, pages 57–76, 2003.

[10] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *Lisp and Symbolic Computation*, 4(3):243–281, 1991.

[11] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[12] C. Click. Fixing the inlining problem. `http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem`, 2011.

[13] C. Consel and S.-C. Khoo. Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.*, 15(3):463–493, 1993.

[14] O. Danvy and A. Filinski. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[15] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, 1987.

[16] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[17] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 57–76. ACM, 2007.

[18] Google. Goole Web Toolkit. `http://code.google.com/webtoolkit/`.

[19] Google. The V8 JavaScript VM, 2009. `https://developers.google.com/v8/intro`.

[20] Google. A new crankshaft for V8, 2010. `http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html`.

[21] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000.

[22] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe linq compilation. *PVLDB*, 3(1):162–172, 2010.

[23] U. Hölzle and O. Agesen. Dynamic versus static optimization techniques for object-oriented languages. *TAPOS*, 1(3):167–188, 1995.

[24] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 1991.

[25] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In S. I. Feldman and R. L. Wexelblat, editors, *PLDI*, pages 32–43. ACM, 1992.

[26] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In V. Sarkar, B. G. Ryder, and M. L. Soffa, editors, *PLDI*, pages 326–336. ACM, 1994.

[27] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-orieted languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[28] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[29] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In G. C. Buttazzo, editor, *EMSOFT*, pages 249–258. ACM, 2004.

[30] A. V. Klimov. A java supercompiler and its application to verification of cache-coherence protocols. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2009.

[31] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. Javascript as an embedded dsl. In *ECOOP*, pages 409–434, 2012.

[32] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *TACO*, 5(1), 2008.

[33] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[34] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002.

[35] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 706–706, New York, NY, USA, 2006. ACM.

[36] T. A. Mogensen. Partially static structures in a self-applicable partial evaluator. 1988.

[37] Oracle. OpenJDK: Graal project, 2012. `http://openjdk.java.net/projects/graal/`.

[38] M. Paleczny, C. A. Vick, and C. Click. The java hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[39] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In P. L. Tarr and W. R. Cook, editors, *OOPSLA Companion*, pages 944–953. ACM, 2006.

[40] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[41] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[42] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.

[43] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

[44] A. Shali and W. R. Cook. Hybrid partial evaluation. OOPSLA, pages 375–390, 2011.

[45] M. Sperber and P. Thiemann. Realistic compilation by partial evaluation. In *PLDI*, pages 206–214, 1996.

[46] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[47] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*, 2013.

[48] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[49] P. Thiemann. Partially static operations. In E. Albert and S.-C. Mu, editors, *PEPM*, pages 75–76. ACM, 2013.

[50] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state. Technical report, 1999.

[51] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[52] J. C. Vogt. Type Safe Integration of Query Languages into Scala. Diplomarbeit, RWTH Aachen, Germany, 2011.

[53] B. Wiedermann and W. R. Cook. Remote batch invocation for sql databases. In *DBPL*, 2011.

[54] C. Wimmer, M. Haupt, M. L. V. de Vanter, M. J. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *TACO*, 9(4):30, 2013.

[55] T. Würthinger. Extending the graal compiler to optimize libraries. In C. V. Lopes and K. Fisher, editors, *OOPSLA Companion*, pages 41–42. ACM, 2011.

[56] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *Onward!*, pages 187–204. ACM, 2013.

[57] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In A. Warth, editor, *DLS*, pages 73–82. ACM, 2012.