Surgical Precision JIT Compilers

Tiark Rompf et al. (PLDI 2014)

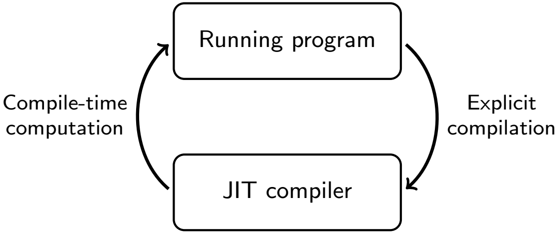
Presented by: Ming-Ho Yee

February 23, 2015

Introduction

JIT compilation allows many optimizations, but the process is a black box and often unpredictable.

Goal: Turn JIT compilation into a "precision tool."



Result: Lancet, JIT compiler framework for Java bytecode.

Outline

Deriving Realistic Optimizing Compilers from Interpreters

- Interpreter + Staging = Compiler
- Compiler + Abstract Interpreter = Optimizer
- JIT Macros as Extension Points

Putting Surgical JIT Facilities to Use

- Program Specialization
- Speculative Optimization
- Just-In-Time Program Analysis
- Smart Libraries and DSLs

Interpreter + Staging = Compiler

Staging: Delaying computation of expressions by generating code.

Lightweight Modular Staging (LMS), a Scala framework

- Expressions of type `T`
- Expressions of type `Rep[T]`

Example: Specializing a regular expression matcher.

```
def matcher(pattern: String, text: Rep[String]) = ...
matcher("abc*", input)
```

Interpreter + Staging = Compiler

A simple interpreter:

```
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(s1, st) + eval(e2, st)
}
```

Staging the interpreter:

```
type Store = Rep[Map[String, Int]]
type Val   = Rep[Int]
def eval(e: Exp, st: Store): Val = ... // unchanged
```

To implement Lancet, the authors took the bytecode interpreter from the Graal project, ported it to Scala, and then staged it with LMS.

Compiler + Abstract Interpreter = Optimizer

Idea: Combine staged interpreter (code generator) with abstract interpreter (program analyzer).

- Introduce abstract values (`AbsVal[T]`)
- Introduce mapping (`evalA`) from `Rep[T]` to `AbsVal[T]`

Example: Constant folding.

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

JIT Macros as Extension Points

Extensions for Lancet are implemented by registering callbacks (macros).

Lancet can then call these user-defined macros, which have access to the compiler internals.

Example: `freeze` evaluates its argument at JIT-compile time.

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

// Macro definition
object LancetMacros {
  def freeze[A](f: Rep[() => A]): Rep[A] = ...
}
```

Program Specialization

Controlled inlining

Inlining can be a source of nondeterminism in automatic JITs.
Lancet provides directives to control inlining:

- `inlineAlways`, `inlineNonRec`, `inlineNever`
- `atScope`, `inScope`

Example:

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

Program Specialization

Code caching and on-demand compilation

Consider specializing calc(x: Int, y: Int) for given values of x.

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

Further ways to extend calcJIT:

- Implementing a custom cache eviction policy
- Specializing only for "hot" values of x
- Adding background compilation
- Generalizing it for any two-argument function

Speculative Optimization

JIT macros allow us to convey speculation directives to Lancet.

```
// The condition is likely to succeed.
// Warn if profiling suggests otherwise.
if (likely(cond)) { ... } else { ... }

// Assume the condition always succeeds and compile the true branch.
// If it fails, switch to interpreted mode.
if (speculate(cond)) { ... } else { ... }

// Assume the condition changes rarely.
// If it fails, recompile the code.
if (stable(cond)) { ... } else { ... }
```

Speculative Optimization

Implementing deoptimization

The primitive `slowpath` (`fastpath`) triggers a switch to interpreted (freshly-compiled) mode at the current point of execution.

```
// Assume the condition always succeeds and compile the true branch.
// If it fails, switch to interpreted mode.
def speculate(x: Boolean) =
  if (x) true else { slowpath(); false }
```

Essentially, `slowpath` and `fastpath` are doing on-stack-replacement.

Speculative Optimization

Exploiting stable structure in trees or graphs

- Consider implementing a dictionary with a search tree
- Typically, reads dominate writes, so the tree structure is fairly stable
- Compile (specialize) the lookup code for a given instance
- Invalidate and recompile the lookup code as needed

Just-In-Time Program Analysis

Controlling allocation and garbage collection

GC is yet another source of nondeterminism. However, the JIT compiler controls all memory allocation.

```
checkNoAlloc {
  // Compiler error if heap allocation cannot be replaced by local fields
}
```

If no error is raised, no heap allocation occurs, so no GC is needed.

Active Libraries and Embedded DSLs

Lancet allows ordinary Scala code to use other, existing LMS-based frameworks as backends.

Example: Delite framework for developing parallel domain-specific languages.

Define the DSL using Delite operators. Then Delite will generate optimized code for the target language (e.g. Scala, C++, CUDA).

Goal: Use Lancet and Delite to improve the performance of ordinary Scala code.

Active Libraries and Embedded DSLs

Building active libraries

OptiML is a parallel DSL for machine learning, built on top of Delite.

Performance speedups of an OptiML application, using:

- Pure Scala version of OptiML
- Pure Scala version of OptiML with Lancet macros that invoke Delite methods
- Stand-alone Delite

Cores:	1	2	4	8
***k*-Means Clustering**				
Scala library	1.00	1.37	1.50	1.83
Lancet-Delite	4.92	8.82	17.10	24.00
Delite	5.17	10.03	19.81	24.78

We get performance competitive with compiled DSLs, but the readability of ordinary Scala code.

Active Libraries and Embedded DSLs

Accelerating existing libraries

Now we use Lancet and Delite to transparently optimize existing Java bytecode programs.

```
def nameScore(names: Array[String]) = {
  val scores = names.zipWithIndex map { case (a, i) =>
    val score = a.map(c => c - 64).reduce(_+_)
    ((i + 1) * score).toLong
  }
  scores.reduce(_+_)
}
```

We implement macros for `zipWithIndex`, map, and reduce which call Delite operators.

Cores:	1	2	4	8
Name Score				
Scala library	1.00	1.71	3.08	4.36
Lancet-Delite	1.92	3.15	6.54	9.67

Conclusion

Lancet, a JIT compiler framework, allows the running program to control the compilation process.

Lancet and the program can call into each other, enabling:

- Program specialization
- Speculative optimization
- Just-in-time program analysis
- Smart libraries