Dynamic Compilation of DSLs

Vojin Jovanovic and Martin Odersky

EPFL, Switzerland
`firstname.lastname@epfl.ch`

Domain-specific language (DSL) compilers use *domain knowledge* to perform *domain-specific optimizations* that can yield several orders of magnitude speedups [4]. These optimizations, however, often require knowledge of values known only at program runtime. For example, in matrix-chain multiplication, knowing matrix sizes allows choosing the optimal multiplication order [2, Ch. 15.2] and in relational algebra knowing relation sizes is necessary for choosing the right join order [6]. Consider the example of matrix-chain multiplication:

```
val (m1, m2, m3) = ... // matrices of unknown size
m1 * m2 * m3
```

In this program, without knowing the matrix sizes, the DSL compiler can not determine the optimal order of multiplications. There are two possible orders `(m1*m2)*m3` with an estimated cost `c1` and `m1*(m2*m3)` with an estimated cost `c2` where:

```
c1 = m1.rows*m1.columns*m2.columns+m1.rows*m2.columns*m3.rows
c2 = m2.rows*m2.columns*m3.columns+m1.rows*m2.rows*m3.columns
```

Ideally we would change the multiplication order at runtime only when the condition `c1 > c2` changes. For this task *dynamic compilation* [1] seems ideal.

Yet, dynamic compilation systems—such as DyC [3] and JIT compilers—have shortcomings. They use runtime information primarily for specialization. In these systems profiling tracks *stability* of values in the user program. Then, *recompilation guards* and *code caches* are based on checking equality of current values and previously stable values.

To perform domain-specific optimizations we must check stability, introduce guards, and code caches, based on the computation specified in the DSL optimizer—outside the user program. Ideally, the DSL optimizer should be agnostic of the fact that input values are collected at runtime. In the example stability is only required for the condition `c1 > c2`, while the values `c1` and `c2` themselves are allowed to be unstable. Finally, recompilation guards and code caches would recompile and reclaim code based on the same condition.

An exception to existing dynamic compilation systems are Truffle [7] and Lancet [5]. They allow creation of user defined recompilation guards. However, with Truffle, language designers do not have the full view of the program, and thus, can not perform global optimizations (e.g., matrix-chain multiplication optimization). Further, recompilation guards must be manually introduced and the code in the DSL optimizer must be modified to specially handle decisions based on runtime values.

We propose a dynamic compilation system aimed for domain specific languages where:

– DSL authors declaratively, at the definition site, state the values that are of interest for dynamic compilation (e.g., array and matrix sizes, vector and matrix sparsity). These values can regularly be used for making compilation decisions throughout the DSL compilation pipeline.
– The instrumented DSL compiler transparently reifies all computations on the runtime values that will affect compilation decisions. In our example, the compiler reifies and stores all computations on runtime values in the unmodified dynamic programming algorithm [2] for determining the optimal multiplication order (i.e., `c1 > c2`).
– Recompilation guards are introduced automatically based on the stored DSL compilation process. In the example the recompilation guard would be `c1 > c2`.
– Code caches are automatically managed and addressed with outcomes of the DSL compilation decisions instead of stable values from user programs. In the example the code cache would have two entries addressed with a single boolean value computed with `c1 > c2`.

The goal of this talk is to foster discussion on the new approach to dynamic compilation with focus on different policies for automatic introduction of recompilation guards: *i)* heuristic, *ii)* DSL author specified, and *iii)* based on domain knowledge.

References

1. Joel Auslander, Matthai Philipose, Craig Chambers, Susan J Eggers, and Brian N Bershad. Fast, effective dynamic compilation. In *International Conference on Programming Language Design and Implementation (PLDI)*, 1996.
2. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.
3. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1), 2000.
4. Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanović, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Symposium on Principles of Programming Languages (POPL)*, 2013.
5. Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Kunle Olukotun, and Martin Odersky. Project Lancet: Surgical precision JIT compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013.
6. P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *International conference on Management of data (SIGMOD)*, pages 23–34, 1979.
7. Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013.