Step in compilation Identification and classification

Problem:

Now, to produce software we use a lot of tools in the compile chain. Not only one compiler.

A software is a result of many translations at different times.

Goal:

Identify and classify this different times.

Example: FFTW

DSL for FFT function generation

Library generation :

Decomposition & description allow to generate variant FFTW.

During installation:

Performance evaluation of each variant

At runtime:

Function selection by planer

Example: HPBCG

DSL for including compilette in your code

```
#cpu cell
typedef int (*pifi)(int);
pifi multiplyFunc;

pifi multiplyCompile(int multiplyValue)
{
  insn *code;

  posix_memalign(&code, 1024, 16);
  printf("Code generation for multiply value %d\n", multiplyValue);
  #[
       .org    code
       mpyi    $3, $3, (multiplyValue)
       bi $lr
  ]#;
  printf("Code generated\n");
  return (pifi)code;
}
```

Example: HPBCG

External processing:

- Parse assembler chunk in the code and assemble it in binary chunk.

Compilation:

- create code generator

Runtime :

- instantiate chunk of binary by putting correct constant value in the compilette.
- Call it

Analysis of some different tool:

FFTW : Fast Fourier Transform generator
SPIRAL: Code generation for DSP
Rathaxes : Code generation for driver
Mesa : 3d library
Gcc : C/C++ (and more) compiler
Llvm : a modular backend for create compiler
Nanojit : a C++ library that allow to emits machine code
HPBCG : High Performance Binary code generator
VPU : Fast, architecture neutral dynamic code generator
Java : Compiler and Virtual machine.
.NET : Compiler and Virtual machine.
Cuda : language extension for GPU programming
OpenCL : language extension for GPU programming

Result : Different technics are used

ECG: External code generation
High level algorithm representation and decomposition allow to generate specialized part of program

ICG: Internal code generation
Use program information to generate specialized part of program

IT: Install Time
Copy program and dependencies into a specific machine

LT:Loading Time
Collect usable part of code and load program

SC: Static compilation
translate program into optimized machine code

IC: Iterative Compilation
Use previous running information to optimize code

JIT: Just-in-time compilation
runtime code translation of already compiled IR in machine code

DDS: Data driven specialization
Runtime selection of the fastest alogithm

Result analysis:

Heterogenous **tools** → different **goals**

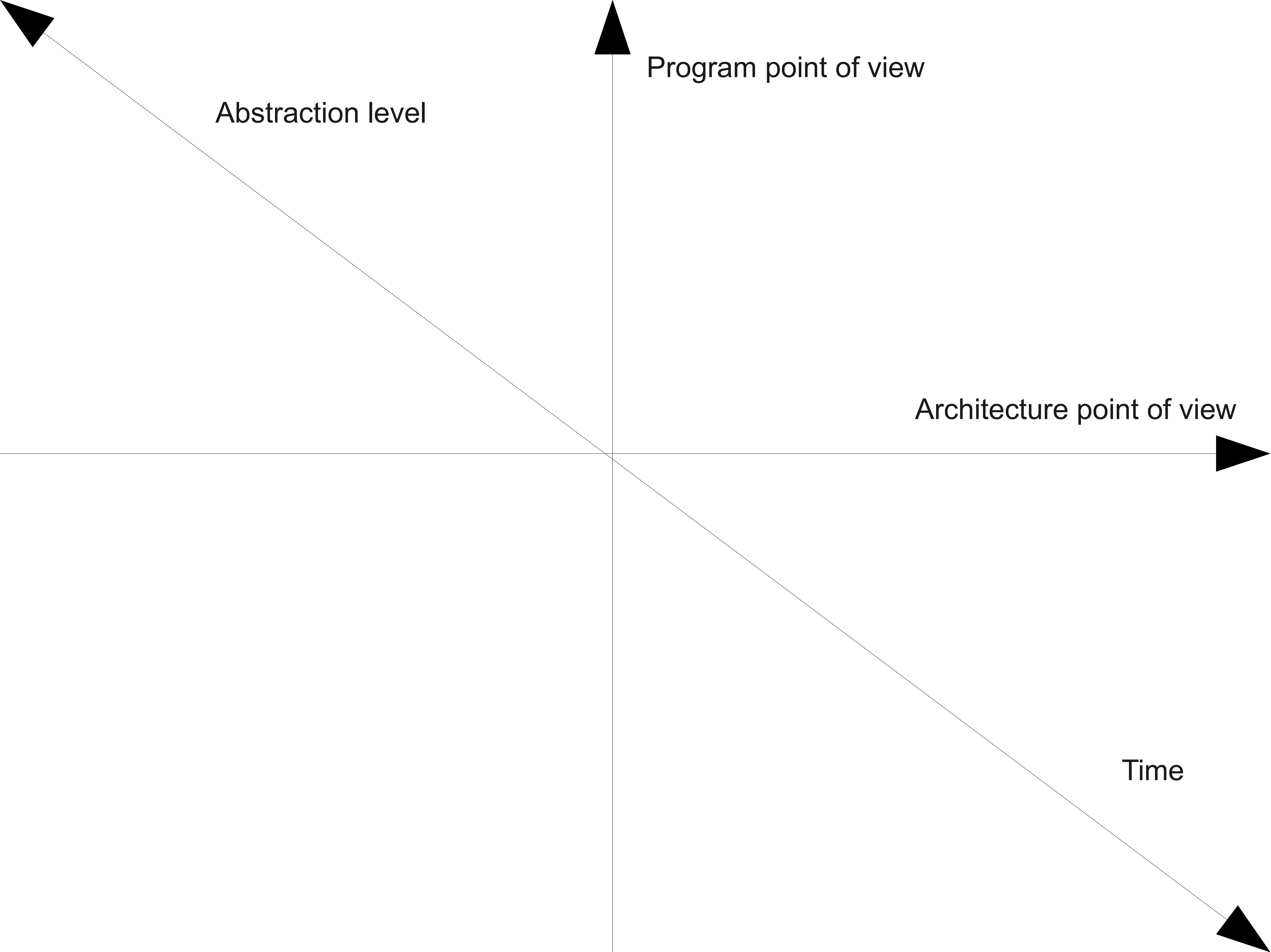
Give more or less **abstraction**
Give more or less **optimization**
Done **early** or **late** in compilation chain
Not the same concern: **model** or **data**

Different point of view:

Program

Architecture

So we need to take account of these parameter.


Program point of view
Abstraction level
Architecture point of view
Time

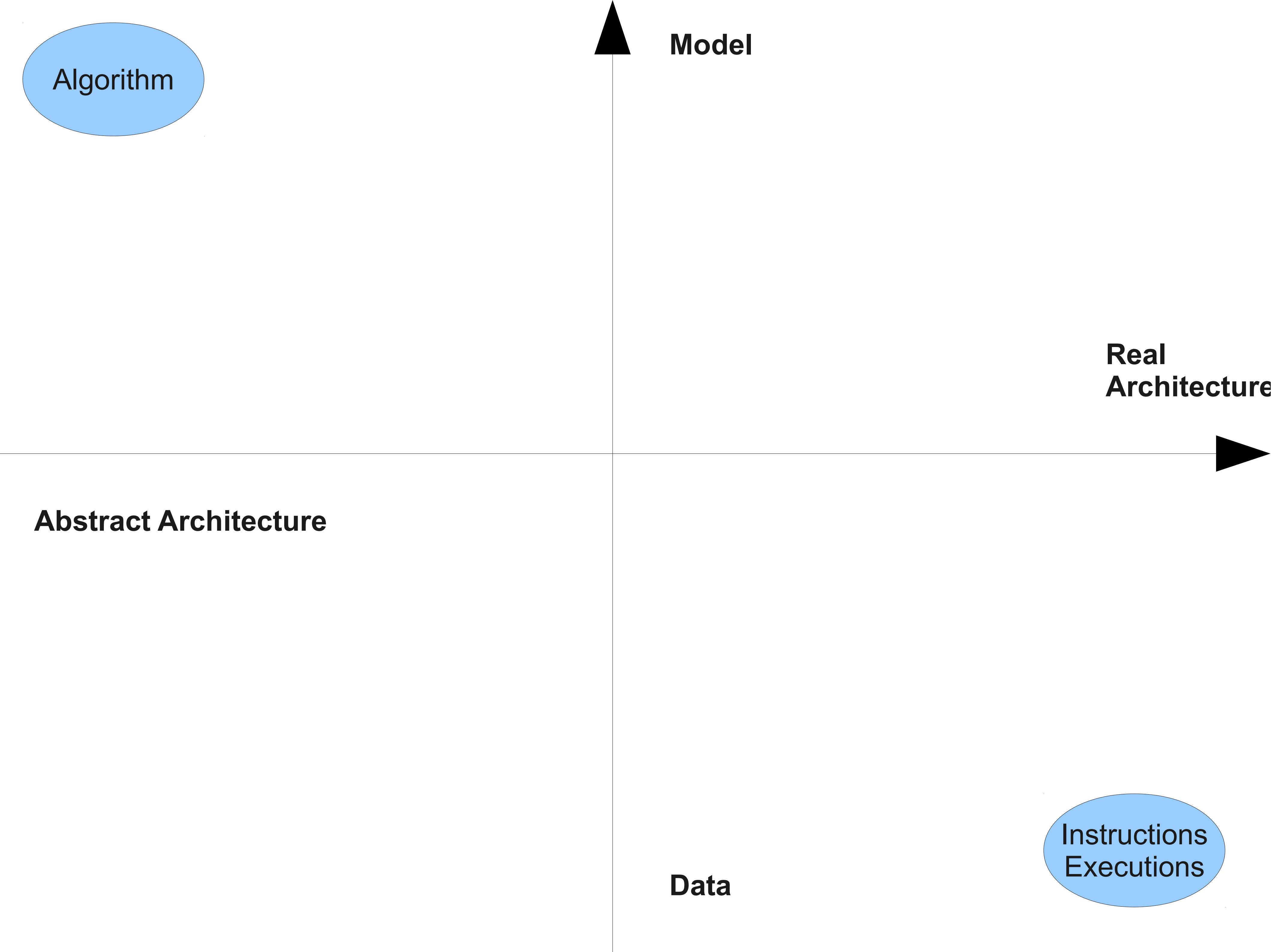
Emergence of an organisation:

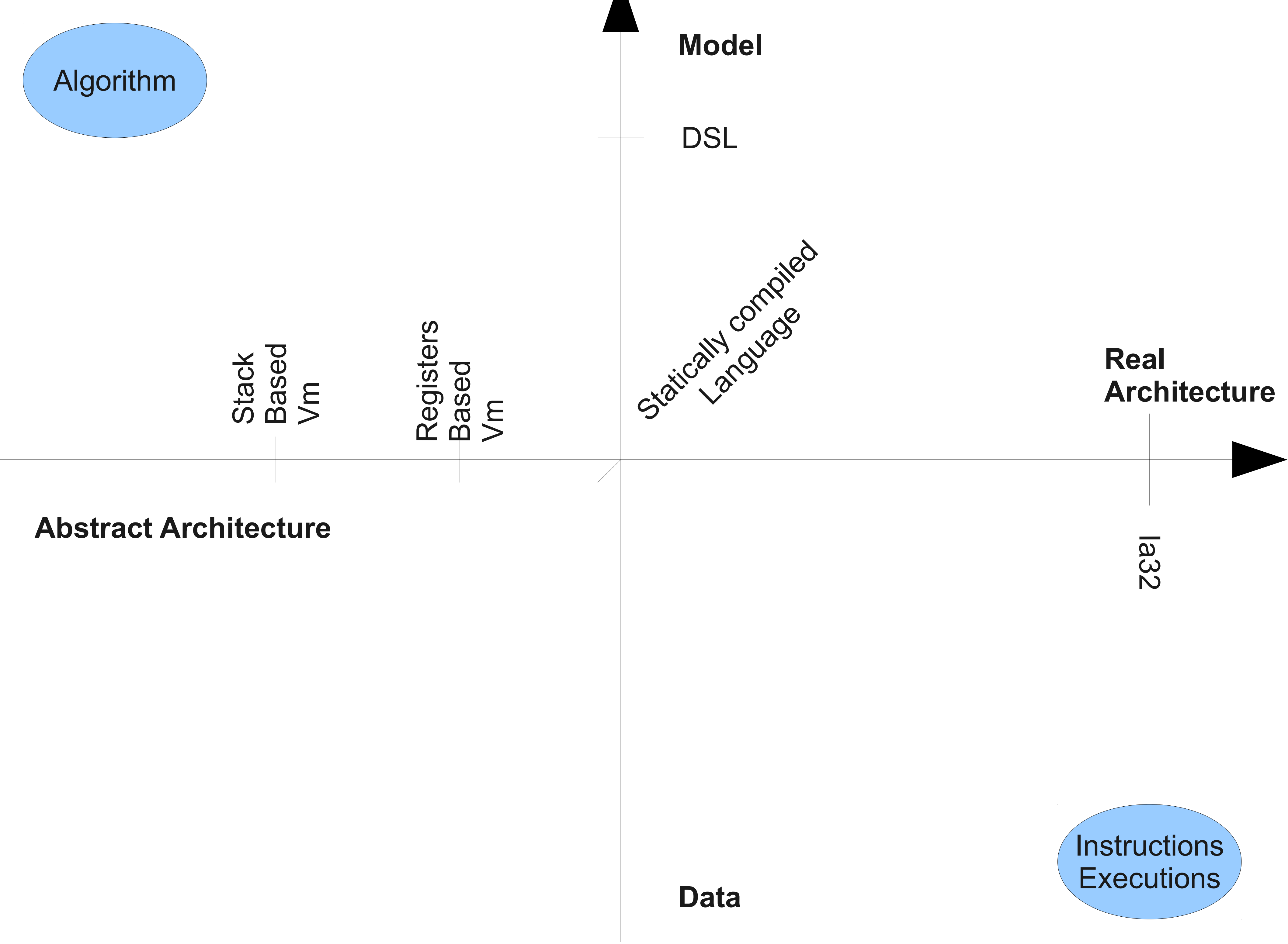
2 MAJOR OPOSITION :

MODEL ↔ DATA

ABSTRACT ARCHITECTURE ↔ REAL ARCHITECTURE

More generic with abstraction in our program we are,
Less known values we use.
More Specialized with known values in our program we are,
Less abstraction we use.


Algorithm
Model
Real
Architecture
Abstract Architecture
Instructions
Executions
Data


Model
Algorithm
DSL
Statically compiled Language
Stack Based Vm
Registers Based Vm
Real Architecture
Abstract Architecture
Ia32
Instructions Executions
Data

Emergence of an organisation:

Code generation centric: Mainly works on source code (or AST) guided by the model.

* External code generation: generation via external tool after processing a DSL.
* Internal code generation: features of language (preproc, metaprog) for a generation, use the semantics of the host language.

Package centric: Mainly on architecture concrete, works on the the system (type of OS and library), guided by the machine.

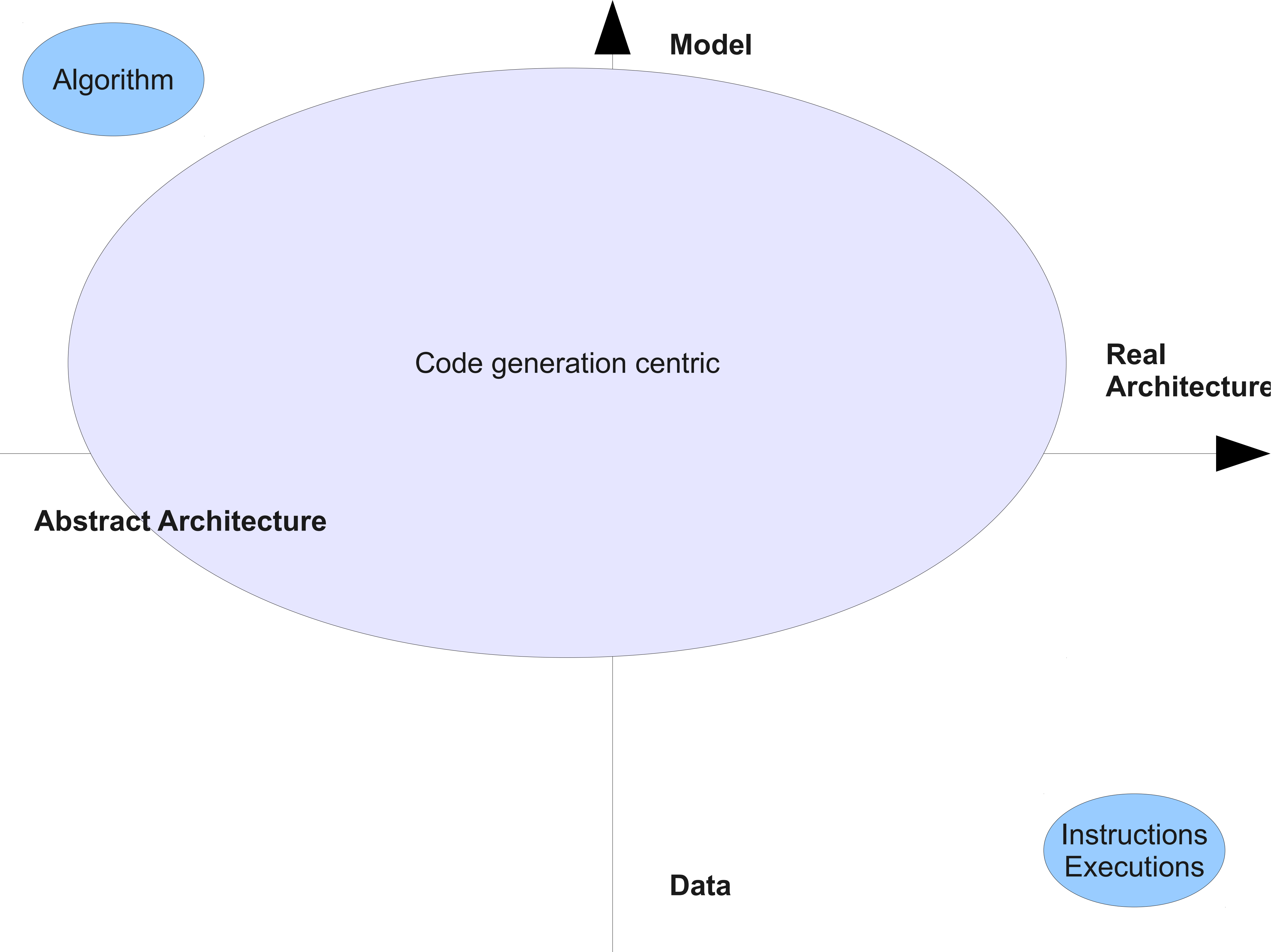
* Install time: action taken when copying the software in the system
* Loading time: action taken when loading the software into memory

Compiler centric: Cold optimization of the program guided by the source code.

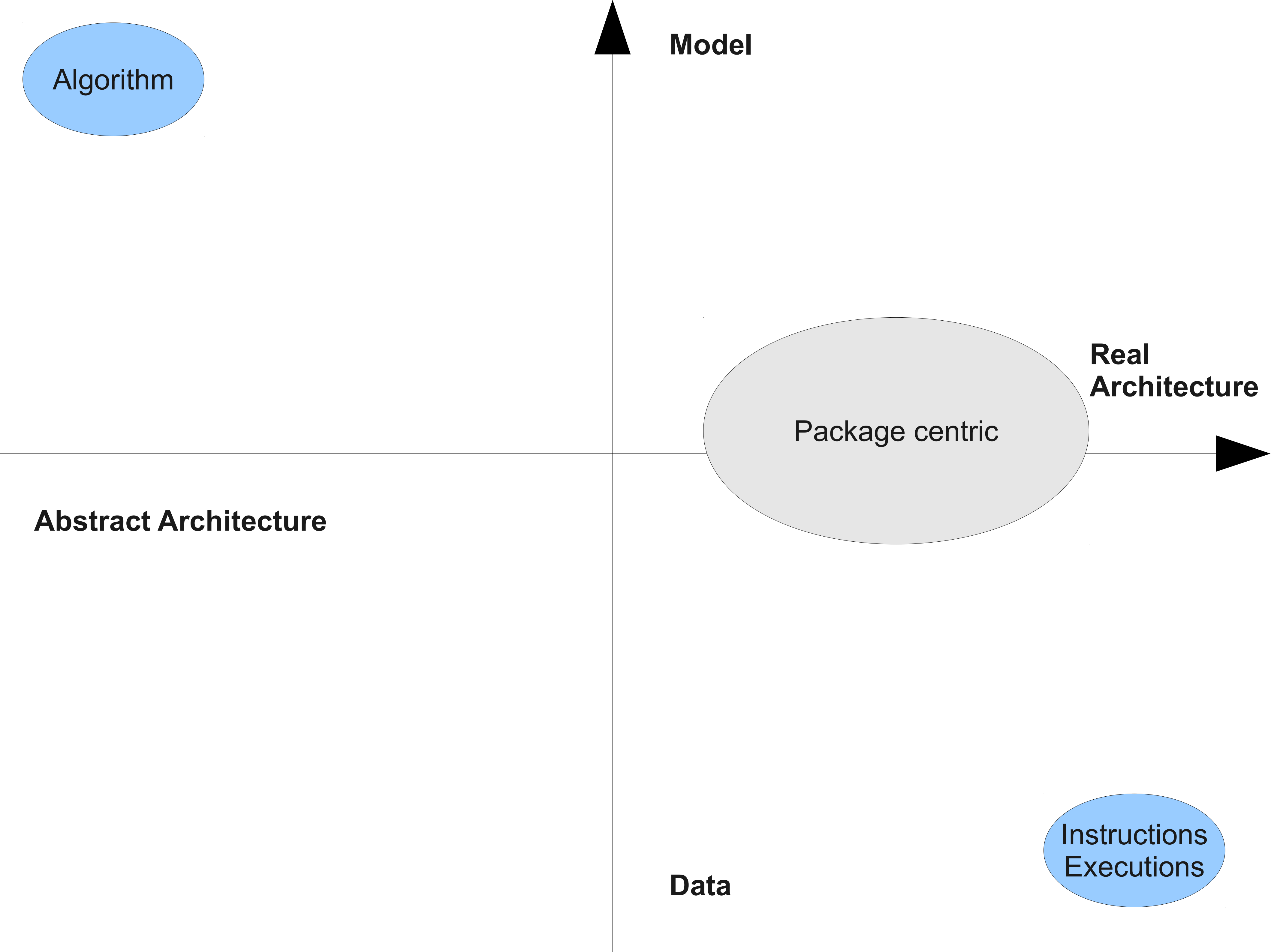
* Static compilation: Classical compilation
* Iterative compilation: compilation with consideration of the preceding run.

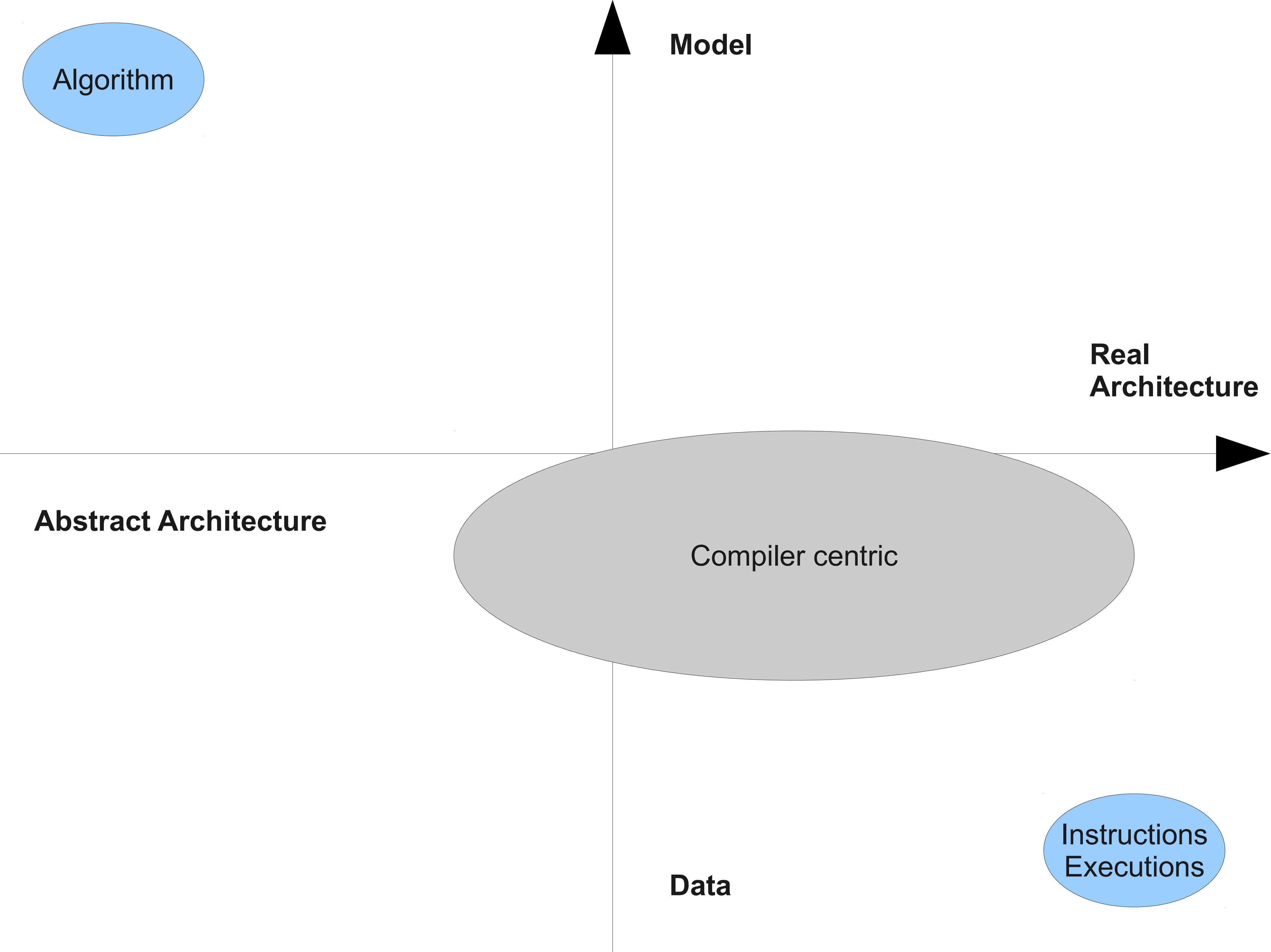
Runtime centric: Live optimization algorithm guided by the data and the concrete machine.

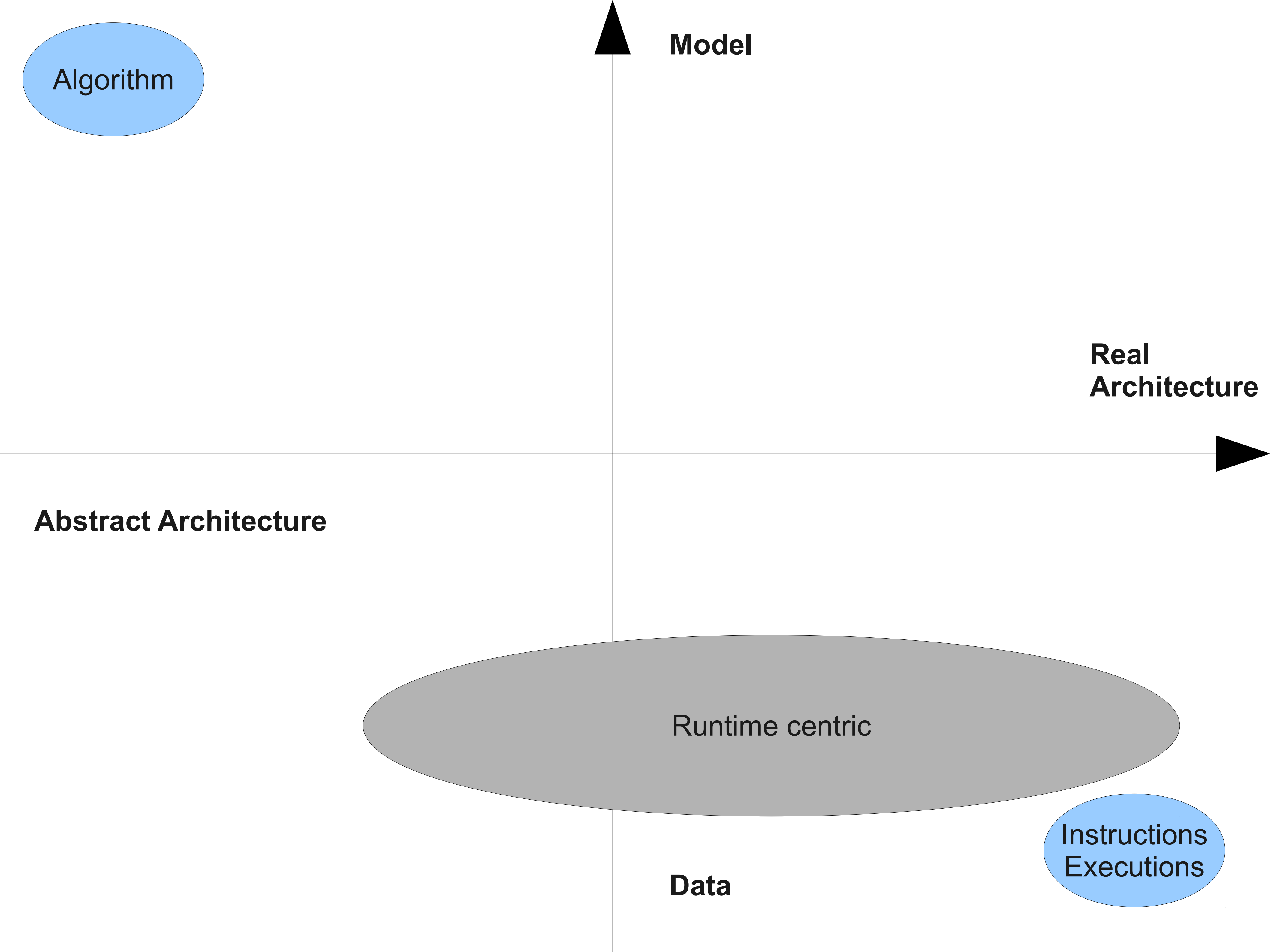
* JIT: Compilation and dynamic specialization based on the data to be processed (hotspot compilettes). Huge runtime overhead.
* Data-driven Specialization: selecting among a set of functions the most appropriate in relation to input data. Little or no runtime overhead.

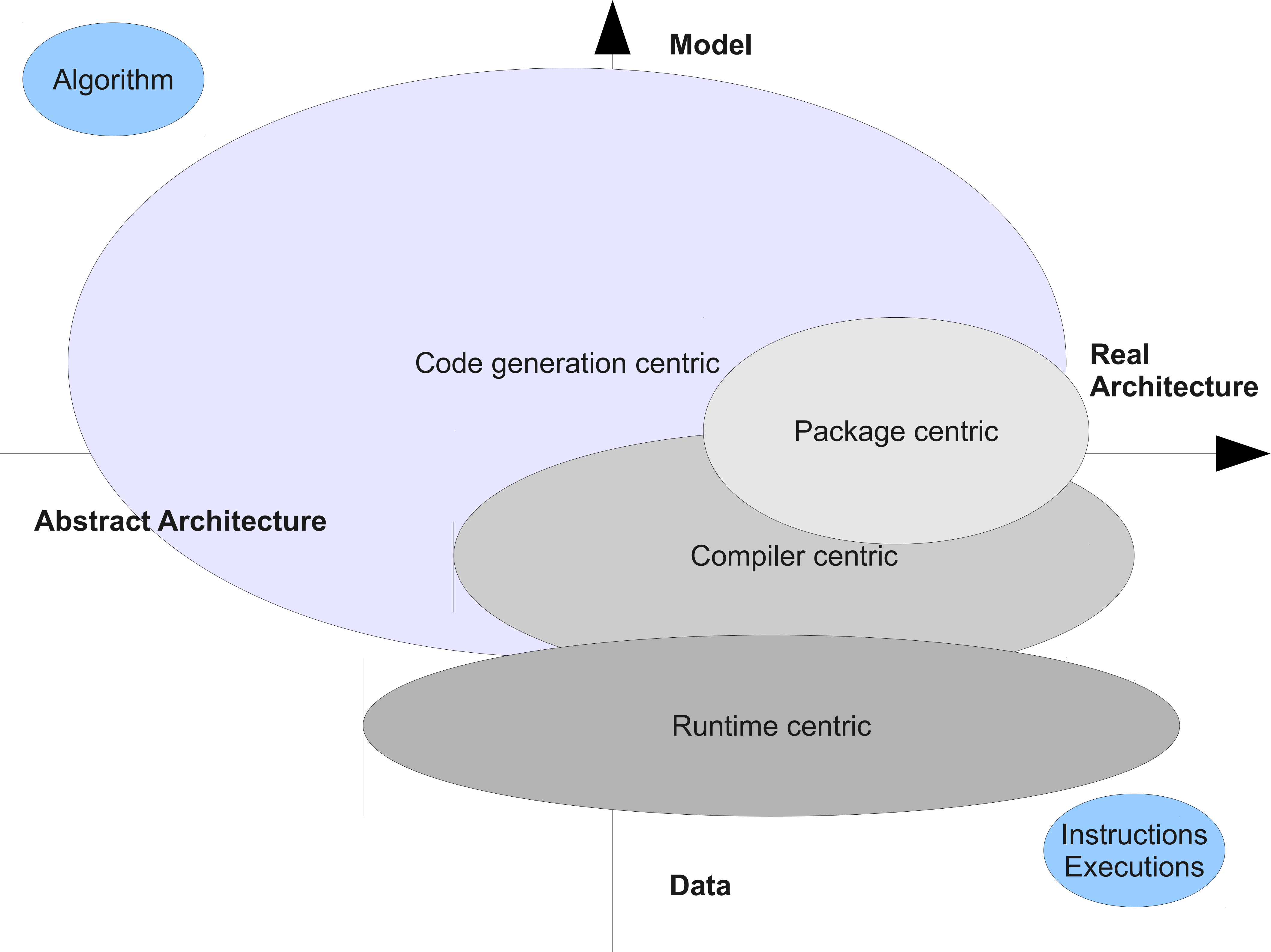


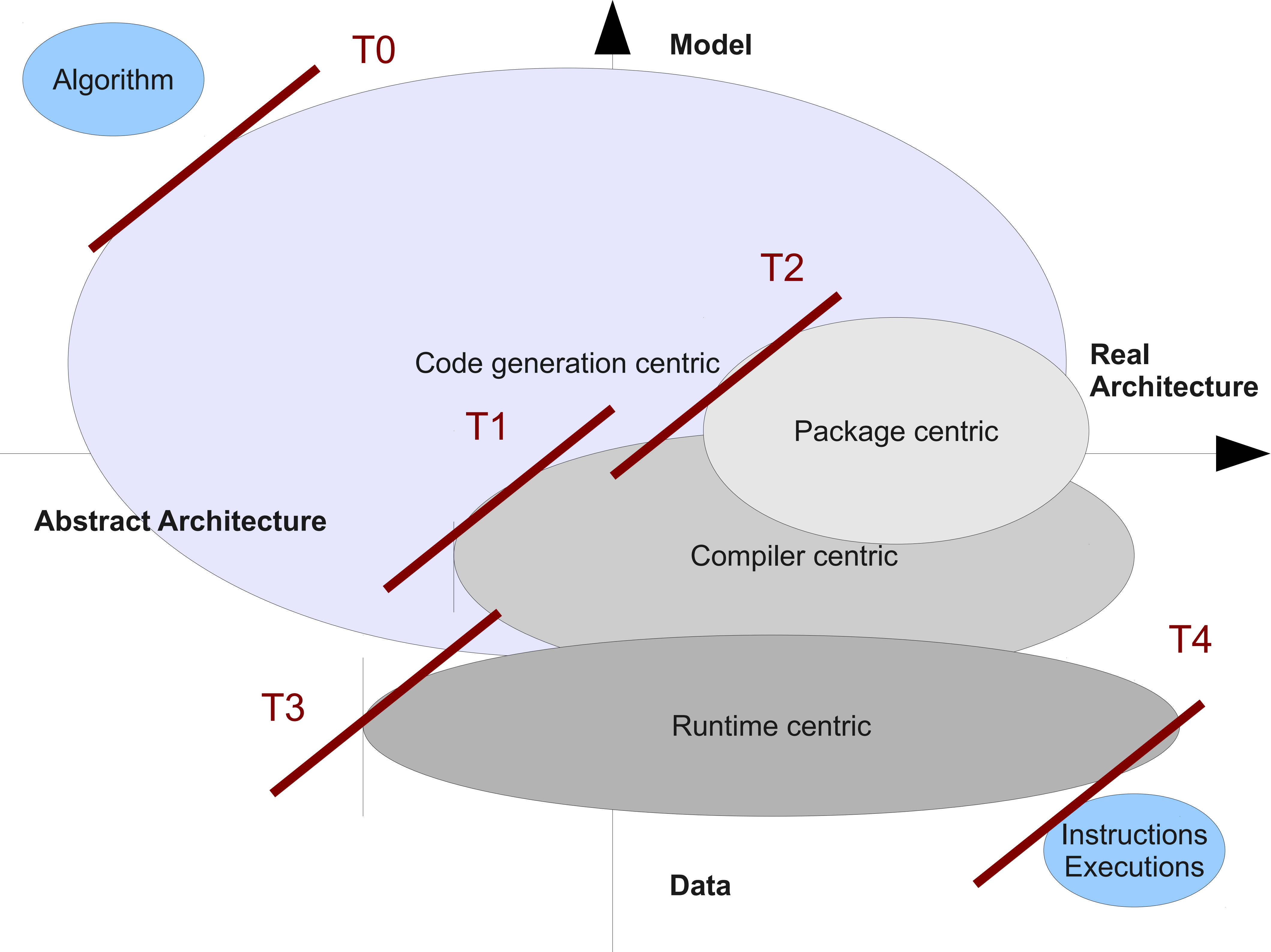
Algorithm
Model
Code generation centric
Real
Architecture
Abstract Architecture
Instructions
Executions
Data


Algorithm
Model
Real Architecture
Package centric
Abstract Architecture
Instructions Executions
Data


Model
Algorithm
Real
Architecture
Abstract Architecture
Compiler centric
Instructions
Executions
Data


Algorithm
Model
Real Architecture
Abstract Architecture
Runtime centric
Instructions Executions
Data


Model
Algorithm
Code generation centric
Real
Architecture
Package centric
Abstract Architecture
Compiler centric
Runtime centric
Instructions
Executions
Data



Algorithm
T0
Model
T2
Code generation centric
Real
Architecture
T1
Package centric
Abstract Architecture
Compiler centric
T4
T3
Runtime centric
Instructions
Executions
Data

Times:

T0-T1:

ECG: External code generation

T1-T2:

ICG:Internal code generation

SC: Static compilation

T2-T3:

:   IT: Install Time

LT: Loading Time

T3-T4:

JIT: Just-in Time

DDS: Data-driven Specialisation

T0-T4:

IC : Iterative compilation

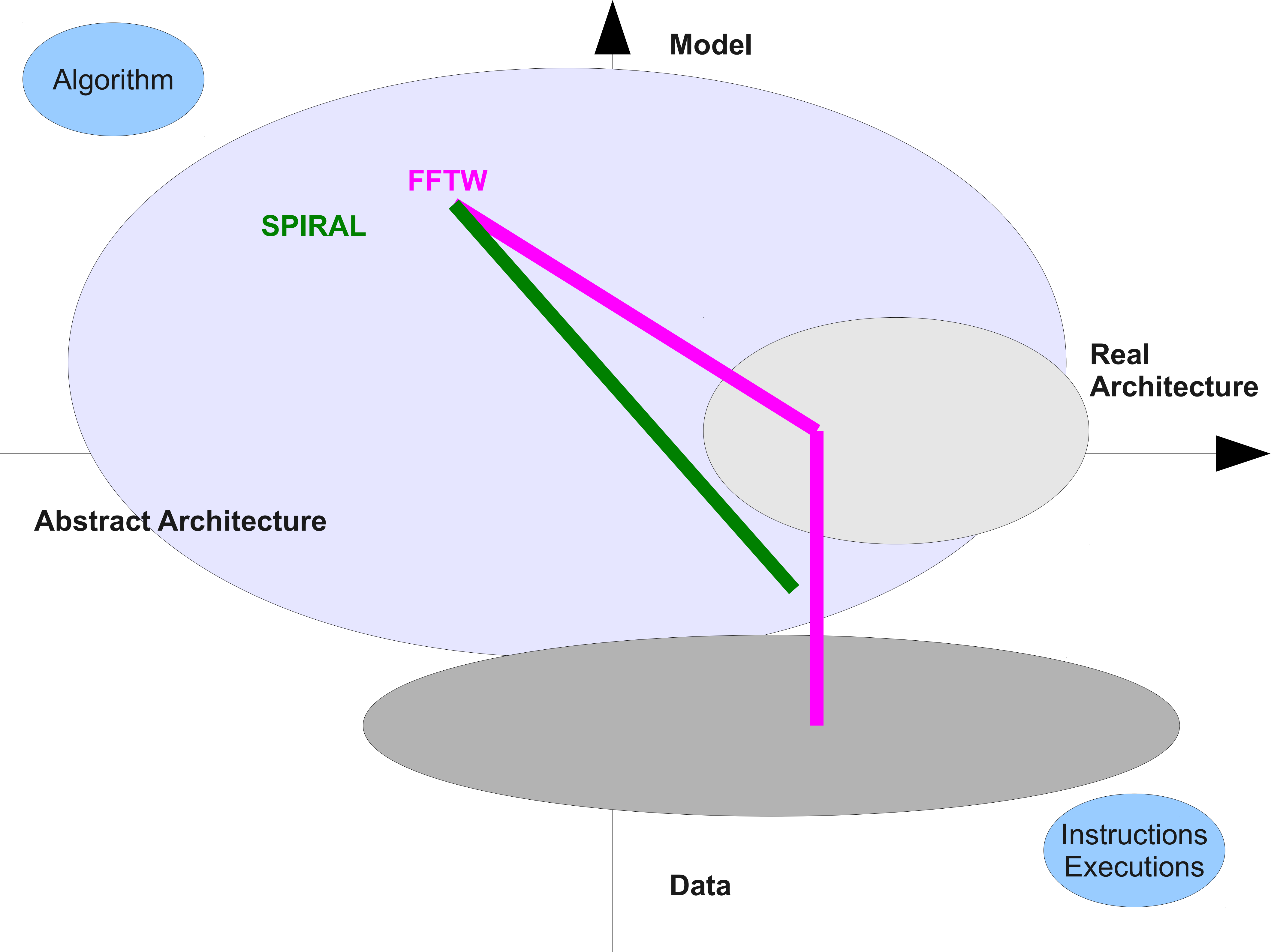
Some program life cycle thread

FFTW vs SPIRAL:

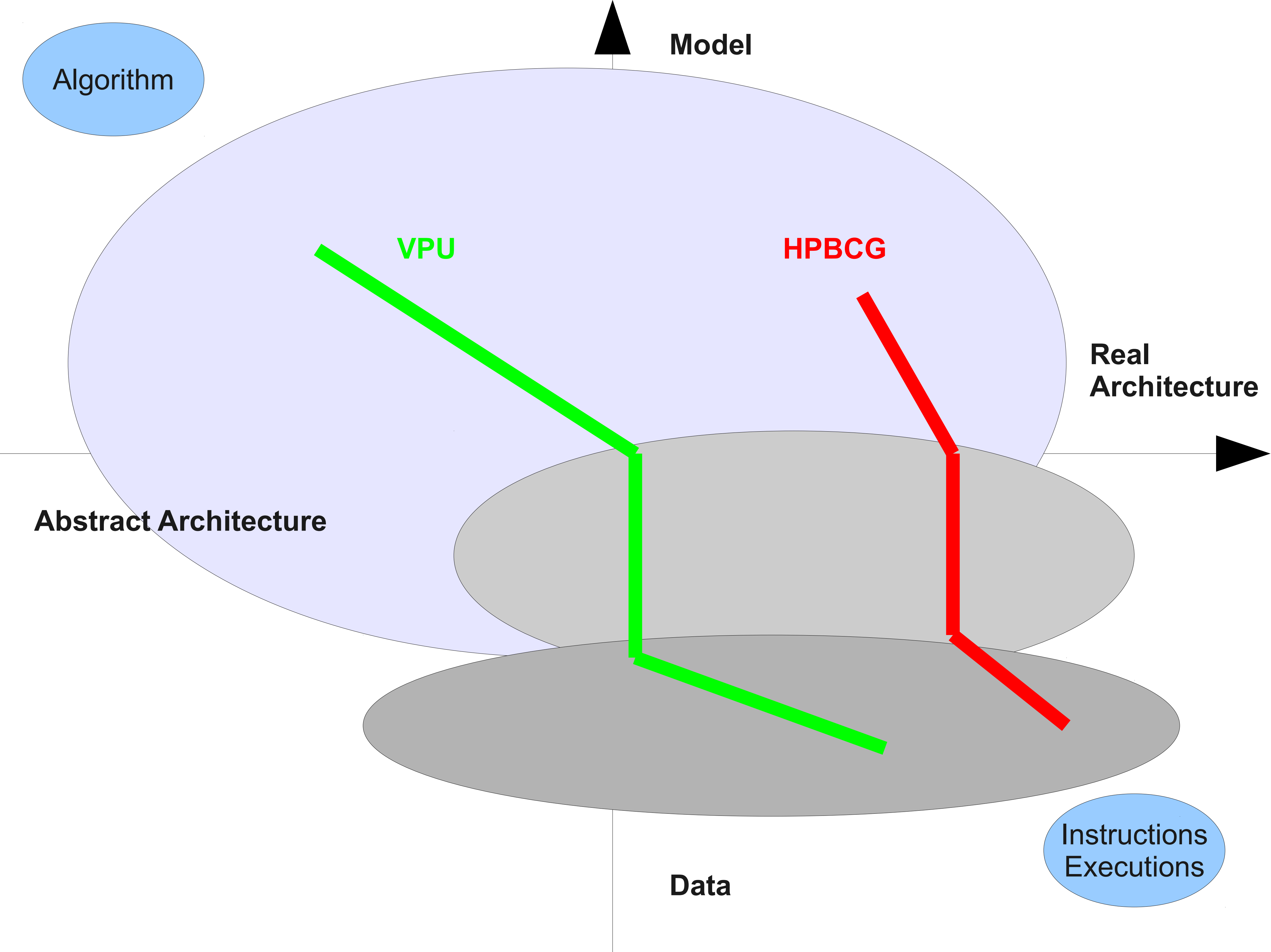
Both present a DSL
Both allow to describe mathematical abstraction

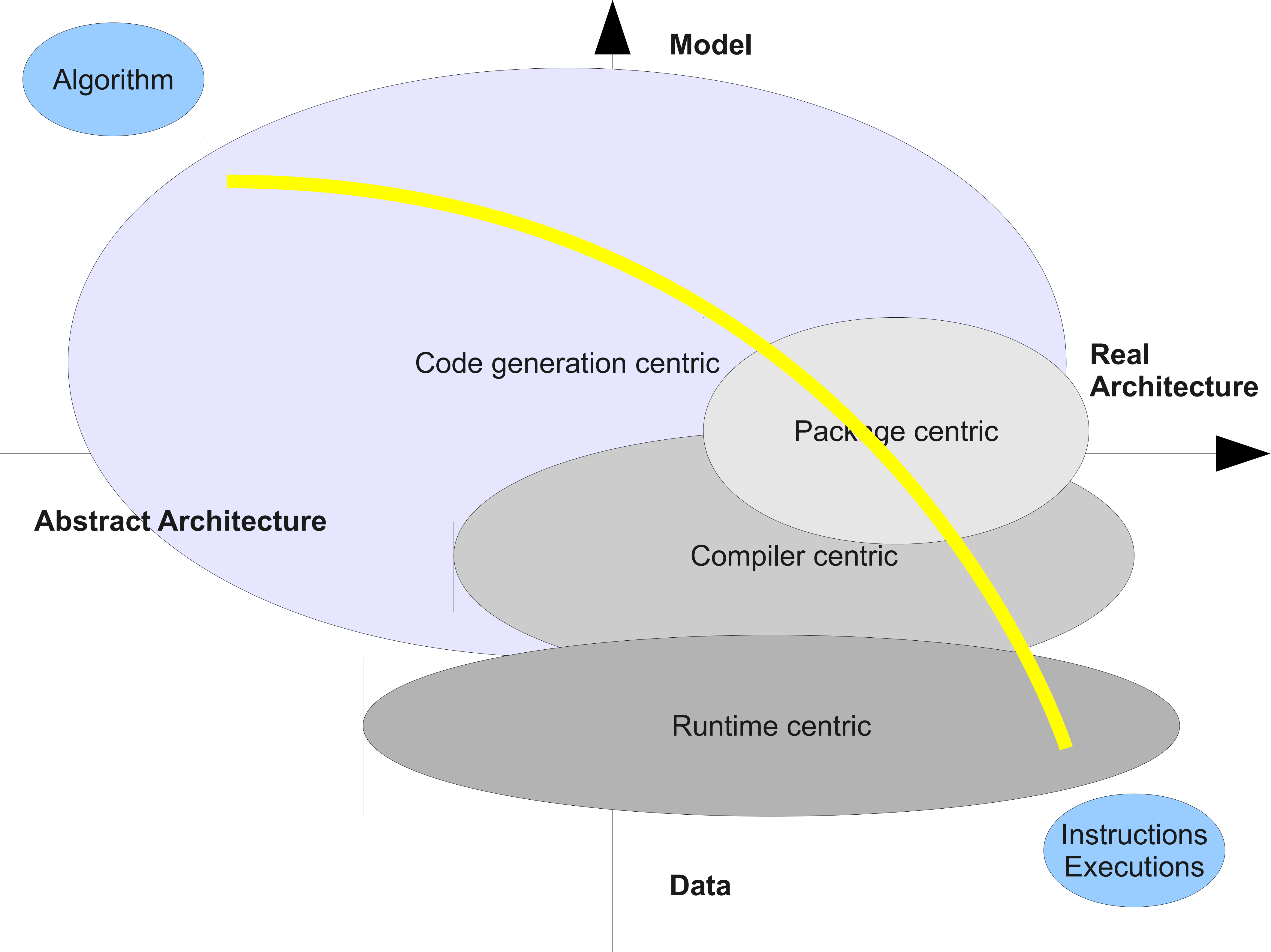
HPBCG vs VPU

VPU allow to emit via library virtual machine instruction
HPBCG allow to emit REAL instruction
Both generate some code before compilation
Both generate code in runtime



Model
Algorithm
FFTW
SPIRAL
Real
Architecture
Abstract Architecture
Instructions
Executions
Data


Model
Algorithm
VPU
HPBCG
Real Architecture
Abstract Architecture
Instructions Executions
Data



Model
Algorithm
Code generation centric
Real
Architecture
Package centric
Abstract Architecture
Compiler centric
Runtime centric
Instructions
Executions
Data

The Perfect Curve

Allow to crawl the different abstraction level throw algorithm to real instruction in all different time of execution.

What's about :

- Optimisation?
- multi-paradigm?

The Perfect Curve constraints:

Frontend agnostic

Modularity and flexibility

Mutli-architecture

Compilation and optimisation

Code emition handling (JIT and more)

The Perfect Curve constraints:

Frontend agnostic

Modularity and flexibility

Mutli-architecture

Compilation and optimisation

Code emition handling (JIT and more)

→ LLVM ?

Questions?