PerfExpert

Jim Browne, Ashay Rane and Leo Fialho

TACC Summer Supercomputing Institute

Agenda

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Overview: why PerfExpert?

Problem: HPC systems operate far below peak

- Chip/node architectural complexity is growing rapidly
- Performance optimization for these chips requires deep knowledge of architectures, code patterns, compilers, etc.

Performance optimization tools

- Powerful in the hands of experts
- Require detailed performance and system expertise
- HPC application developers are domain experts, not computer gurus

Result: Many HPC programmers do not use these tools

(seriously)

Goal for PerfExpert: democratize optimization!

Subgoals:

- Make use of the tool as simple as possible
- Start with only chip/node level optimization
- Make it adaptable across multiple architectures
- Design for extension to communication and I/O performance

How to accomplish?

- Formulate the performance optimization task as a workflow of subtasks
- Leverage the state-of-the-art: build on the best available tools for the subtasks to minimize the effort and cost of development
- Automate the entire workflow

Introduction

The four stages of automatic performance optimization:

- Measurement and attribution (1)
- Analysis, diagnosis and identification of bottlenecks (2)
- Selection of effective optimizations (3)
- Implementation of optimizations (4)

Use of State-of-the-Art:

- HPCToolkit/Intel VTune, **MACPO** based on ROSE (1)
- **PerfExpert Team** (2 and 3)
- **PerfExpert Team** based on ROSE, PIPS, Bison and Flex (4)

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Introduction

Uniqueness of PerfExpert:

- Nearly complete optimization first three stages of optimization for chip/node level
- Framework for implementing optimizations is complete and several optimizations are completed
- Integrates code segment focused and data structure based measurements (**MACPO**)
- Workflow will apply to communication and I/O optimization as well

Introduction

Unique properties of MACPO (integrated into PerfExpert):

- Multicore resolved traces
- Code segment local measurement
- Data structure specific traces
- Order of magnitude lower overhead of measurement
- More accurate (associative) cache models
- Strides by data structure and code segment
- Architecture "independent" metrics

What PerfExpert can provide to you?

Performance report:

- Identification of bottlenecks by relevance
- Performance analysis based on performance metrics
- Recommendations for optimization

There are three possible outputs:

- Performance report only
- List of recommendations
- Fully automated code transformation

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
================================================================================
ratio to total instrns       % 0.........25...........50.........75........100
   - floating point      :    100 ********************************************
   - data accesses       :     25 ************
* GFLOPS (% max)         :     12 ******
   - packed              :      0 *
   - scalar              :     12 ******
--------------------------------------------------------------------------------
performance assessment     LCPI good.....okay.....fair.....poor.....bad....
* overall                :    3.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
upper bound estimates
* data accesses          :    9.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits            :    0.9 >>>>>>>>>>>>>>>>>
   - L2d hits            :    1.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
   - L2d misses          :    6.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses   :    0.1 >
   - L1i hits            :    0.0 >
   - L2i hits            :    0.0 >
   - L2i misses          :    0.1 >
* data TLB               :    4.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction TLB        :    0.0 >
* branch instructions    :    0.1 >>
   - correctly predicted :    0.1 >>
   - mispredicted        :    0.0 >
* floating-point instr   :    5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - fast FP instr       :    5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - slow FP instr       :    0.0 >
```

List of Recommendations

```
#---------------------------------------------
# Recommendations for mm.c:8
#---------------------------------------------
# This is a possible recommendation for this code segment
#
Description:  change the order of loops
Reason:  this optimization may improve the memory access
pattern and make it more cache and TLB friendly
Pattern Recognizers:  c_loop2 f_loop2
Code example:
loop i {
  loop j {...}
}
=====> loop j {
  loop i {...}
}
```

Fully Automated Code Transformation

Before:

```
void compute() {
 register int i, j, k;




 for (i = 0; i < 1000; i++)


  for (j = 0; j < 1000; j++)

   for (k = 0; k < 1000; k++)
    c[i][j] += (a[i][k] * b[k][j]);
}
```

After:

```
void compute() {
 register int i, j, k;
 //PIPS generated variable
 register int jp, kp;
 /* PERFEXPERT: start work here */
 /* PERFEXPERT: grandparent loop */
 loop_6:
 for (i = 0; i <= 999; i++)
  /* PERFEXPERT: parent loop */
  loop_7:
  for(jp = 0; jp <= 999; jp += 1)
   /* PERFEXPERT: bottleneck */
   for(kp = 0; kp <= 999; kp += 1)
   c[i][kp] += a[i][jp]*b[jp][kp];
}
```

Agenda

Basic Usage of PerfExpert

Making PerfExpert Available

```
$ module load papi perfexpert
```

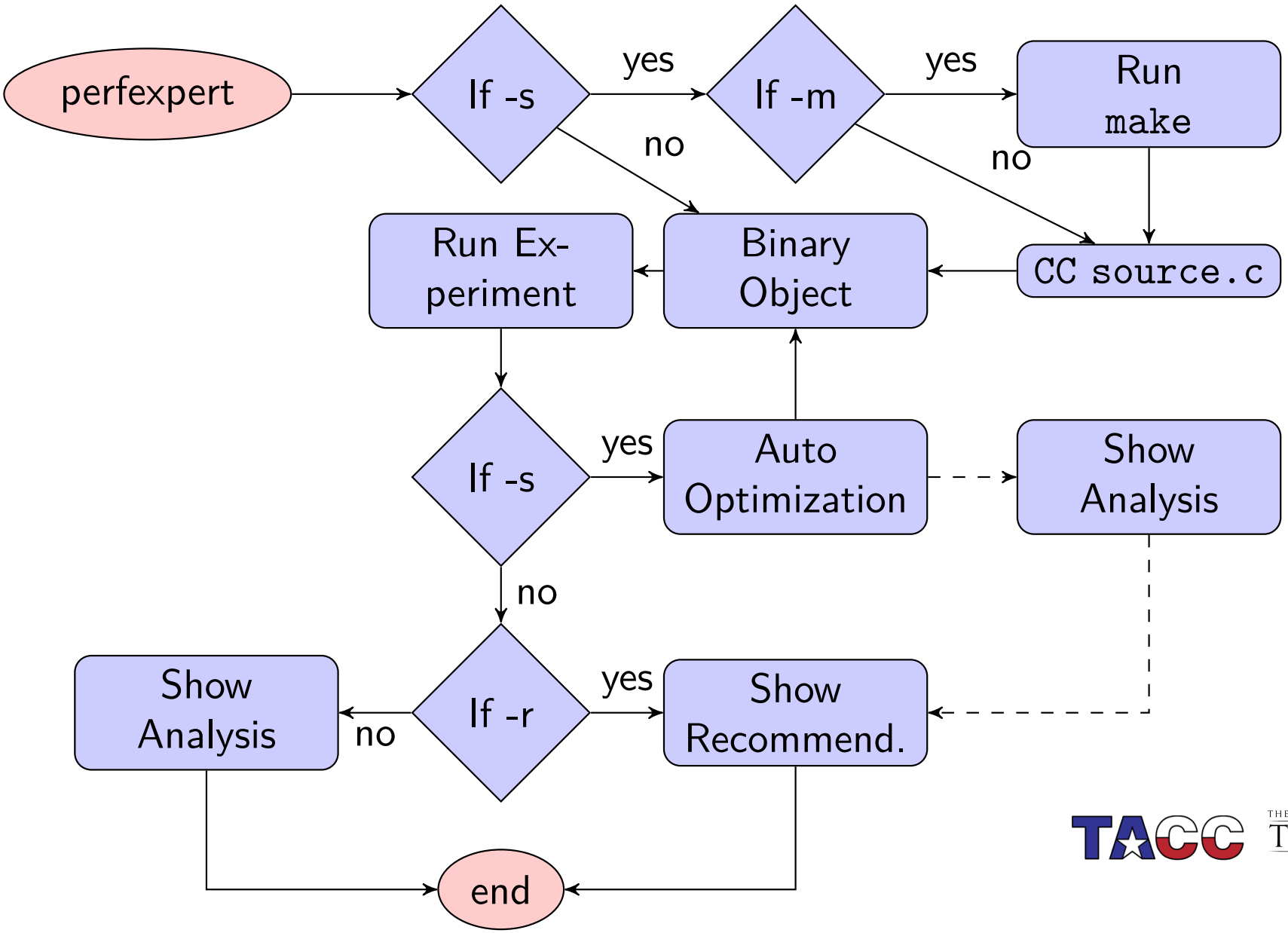
Execution Options

```
Usage:  perfexpert [-ghmvq] [-w DIR] [-s FILE] [-r COUNT]
                   program_executable [program_arguments]

-s    Use FILE as the source code
-m    Use 'make' to compile source code
-q    Disable verbose mode
-w    Use DIR as temporary directory
-g    Do not remove the temporary directory
-r    Use COUNT as the number of recommendation to show

Use CC, CFLAGS and LDFLAGS to select compiler and
compilation/linkage flags
```

Basic Usage of PerfExpert



Basic Usage of PerfExpert

In Other Words...

- No source code, no automatic optimization
- No source code, choose between analysis or recommendation
- Source code, enable automatic optimization
- Source code, choose the compilation method (-m) and options (CC, CFLAGS and LDFLAGS)
- Source code, show analysis and recommendation after all the possible automatic optimizations have been applied

Examples:

```
$ perfexpert my_program param1 param2
$ perfexpert -r 5 my_program param1 param2
$ perfexpert -s my_program.c my_program param1 param2
$ perfexpert -m -s my_program.c my_program param1 param2
```

Understanding PerfExpert Analysis

On the The Analysis Report...

- The more "expensive" comes first
- Tells user where the slow code sections are as well as why they perform poorly
- Every function or loop which takes more than 1% of the execution time is analyzed (default value)
- Yes, we rely on performance metrics (but not only and not the raw ones)
- No, we do not rely on hardware specs
- If you are not using properly the node PerfExpert may conclude everything is fine (use a representative workload)

Metrics used by PerfExpert

Source Code

- Language (C, C++, Fortran)
- File name and line number
- Type (loop or function)
- Function name and "deepness"
- Representativeness (percentage of execution time)

Execution Performance

- Raw data (PAPI)
- LCPI: local cycles per instruction (PerfExpert Analyzer)

Metrics used by PerfExpert

Architecture Characteristics

- Memory access latency: L1, L2, L3 and main memory (based on micro-benchmarks)
- Memory hierarchy, topology and size (based on hwlock)
- Branch latency and missed branch latency (based on micro-benchmarks)
- Float-point operation latency (based on micro-benchmarks)
- Micro-architecture (in progress)

Metrics used by PerfExpert

Data Access Performance (from MACPO)

- Access strides and the frequency of occurrence (*)
- Presence or absence of cache thrashing and the frequency (*)
- Estimated cost (cycles) per access (*)
- NUMA misses (*)
- Reuse factors for data caches (*)
- Stream count

(*) *per variable*

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
================================================================================
ratio to total instrns       % 0.........25...........50.........75........100
   - floating point      :    100 ********************************************
   - data accesses       :     25 ************
```

Interpretation

- What percentage of the total instructions were computational (floating-point instructions)
- What percentage were instructions that accessed data
- So, whether optimizing the program for either data accesses or floating-point instructions would have a significant impact on the total running time of the program?

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
* GFLOPS (% max)          :    12 ******
   - packed               :     0 *
   - scalar               :    12 ******
...
```

Interpretation

- GFLOPs rating, which is the number of floating-point operations executed per second
- This metric is displayed as a percentage of the maximum possible GFLOP value for that particular machine
- It is rare for real-world programs to match even 50% of the maximum value

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
performance assessment     LCPI good.....okay.....fair.....poor.....bad....
* overall                :   3.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
...
```

Interpretation

- LCPI values: is the ratio of cycles spent in the code segment for a specific category, divided by the total number of instructions in the code segment
- The overall value is the ratio of the total cycles taken by the code segment to the total instructions executed in the code segment
- Generally, a value of 0.5 or lower for the CPI is considered to be good

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
* data accesses          :   9.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits            :   0.9 >>>>>>>>>>>>>>>>>
   - L2d hits            :   1.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
   - L2d misses          :   6.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses   :   0.1 >
   - L1i hits            :   0.0 >
   - L2i hits            :   0.0 >
   - L2i misses          :   0.1 >
...
```

Interpretation

- LCPI arising from accesses to memory for program variables
- LCPI arising from memory accesses for code (functions and loops)
- Shows different levels of memory (L1, L2, etc.)

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
* data TLB               :   4.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction TLB        :   0.0 >
* branch instructions    :   0.1 >>
   - correctly predicted :   0.1 >>
   - mispredicted        :   0.0 >
* floating-point instr   :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - fast FP instr       :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - slow FP instr       :   0.0 >
```

Interpretation

- Data TLB: provides an approximate measure of penalty arising from strides in accesses or regularity of accesses
- Instruction TLB: reflects penalty from fetching instructions due to irregular accesses
- Branch instructions: counts penalty from jumps (i.e. if statements, loop conditions, etc.)
- Floating-point instructions: counts LCPI from executing computational (floating-point) instructions

A Simple Example

Automatic Optimization of a C Code

```
$ perfexpert -s code.c code
```

Optimization Steps

- One full optimization cycle
- Runs out of automatic optimizations during the second cycle
- Shows the analysis report as well as recommendations
- Execution time: from 88.856 seconds to 6.967 seconds
- There is (still) room for improvement

Comparing Codes

Before:

```
void compute() {
 register int i, j, k;




 for (i = 0; i < 1000; i++)


  for (j = 0; j < 1000; j++)

   for (k = 0; k < 1000; k++)
    c[i][j] += (a[i][k] * b[k][j]);
}
```

After:

```
void compute() {
 register int i, j, k;
 //PIPS generated variable
 register int jp, kp;
 /* PERFEXPERT: start work here */
 /* PERFEXPERT: grandparent loop */
 loop_6:
 for (i = 0; i <= 999; i++)
  /* PERFEXPERT: parent loop */
  loop_7:
  for(jp = 0; jp <= 999; jp += 1)
   /* PERFEXPERT: bottleneck */
   for(kp = 0; kp <= 999; kp += 1)
   c[i][kp] += a[i][jp]*b[jp][kp];
}
```

Comparing Reports

Before: 88.856 sec

```
ratio to total instrns          %
   - floating point      :    100
   - data accesses       :     25
* GFLOPS (% max)         :     13
   - packed              :      0
   - scalar              :     13
-------------------------------
performance assessment       LCPI
* overall                 :    3.7
* data accesses           :   40.6
   - L1d hits            :    2.3
   - L2d hits            :    4.9
   - L2d misses          :   33.4
* instruction accesses    :    0.1
...
* data TLB                :    4.5
* instruction TLB         :    0.0
* branch instructions     :    0.1
...
* floating-point instr    :    5.7
   - fast FP instr       :    5.7
   - slow FP instr       :    0.0
```

After: 6.967 sec (12x faster)

```
ratio to total instrns          %
   - floating point      :    100
   - data accesses       :     29
* GFLOPS (% max)         :     29
   - packed              :     17
   - scalar              :     12
-------------------------------
performance assessment       LCPI
* overall                 :    0.7
* data accesses           :   10.5
   - L1d hits            :    2.6
   - L2d hits            :    0.9
   - L2d misses          :    7.0
* instruction accesses    :    0.0
...
* data TLB                :    0.0
* instruction TLB         :    0.0
* branch instructions     :    0.1
...
* floating-point instr    :    1.7
   - fast FP instr       :    1.7
   - slow FP instr       :    0.0
```

Exploring the Temporary Directory

Automatic Optimization of a C Code

- Each optimization cycle has it's own subdirectory containing:
  - Source code directory
  - Debug file and intermediary file for every optimization step (5)
  - Analysis report
  - Directory containing the code fragments identified as bottleneck
  - Directory containing the optimized source code
- Workflow log file

We Are Ready To Help You

- The group of people developing PerfExpert is ready to help you!
- There are several other folks at TACC who also use PerfExpert and will be glad to help users get started
- Do not be shy, send us an email, knock our doors, we are here for that!

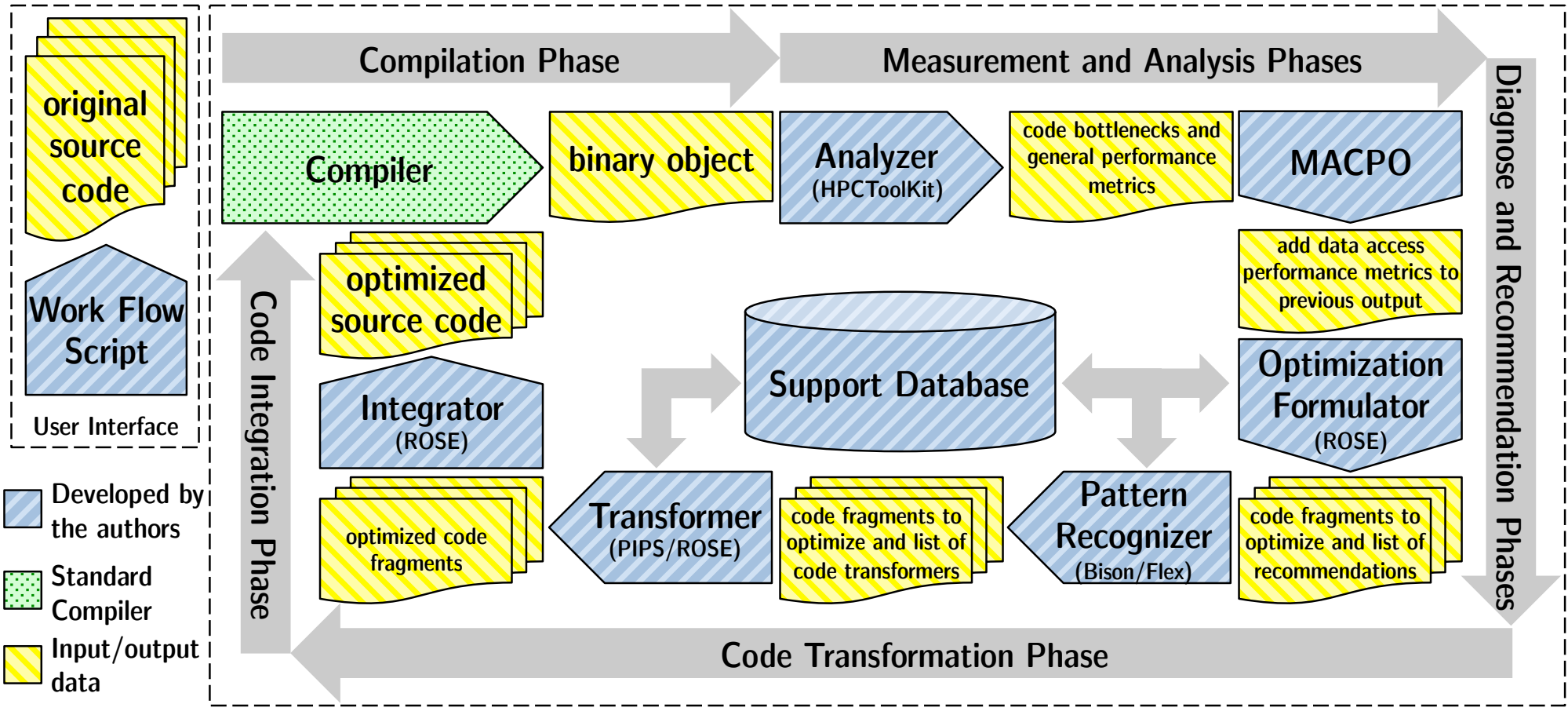
  — http://www.tacc.utexas.edu/perfexpert

  — fialho@utexas.edu

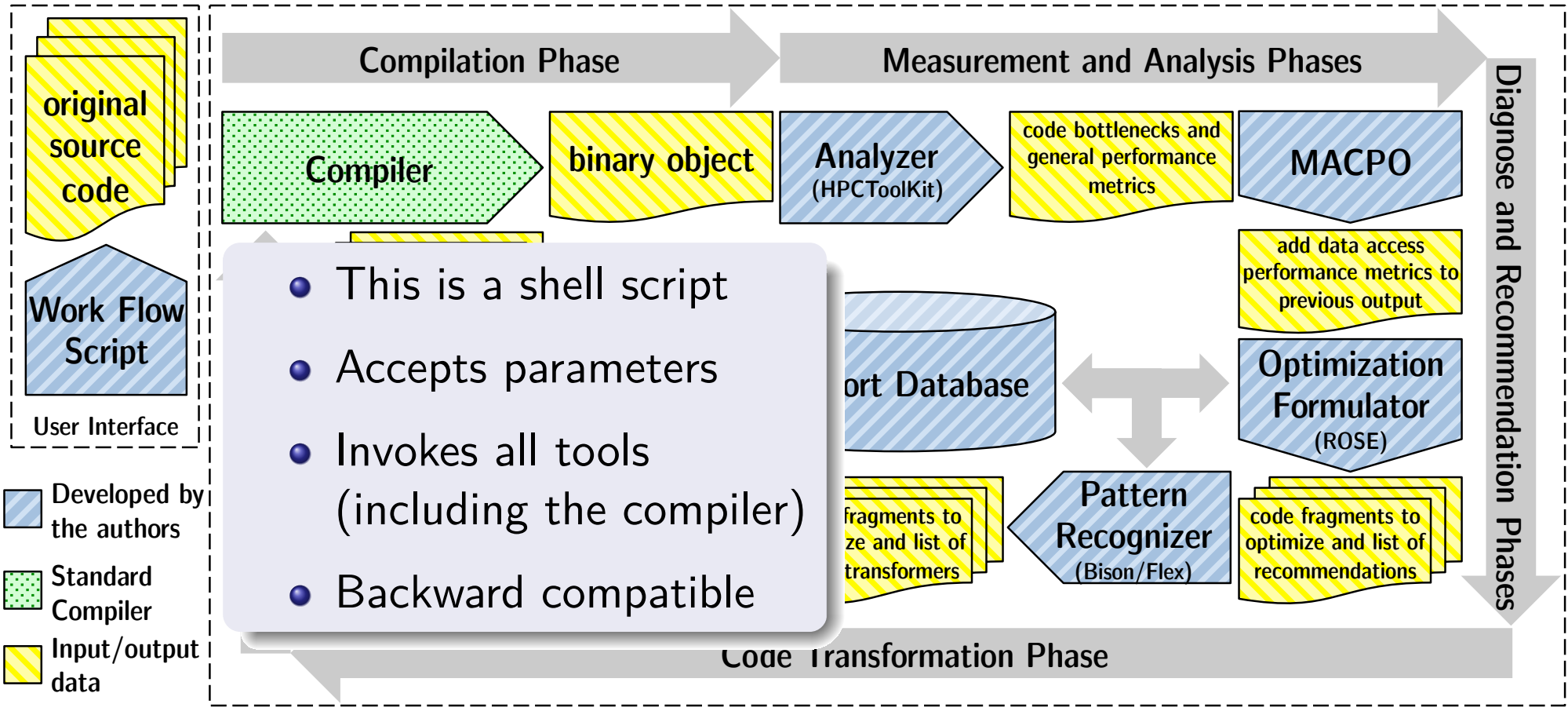
  — Office 1.526
- We will also be happy to help you install PerfExpert on your system

Agenda

How PerfExpert does that: The Big Picture



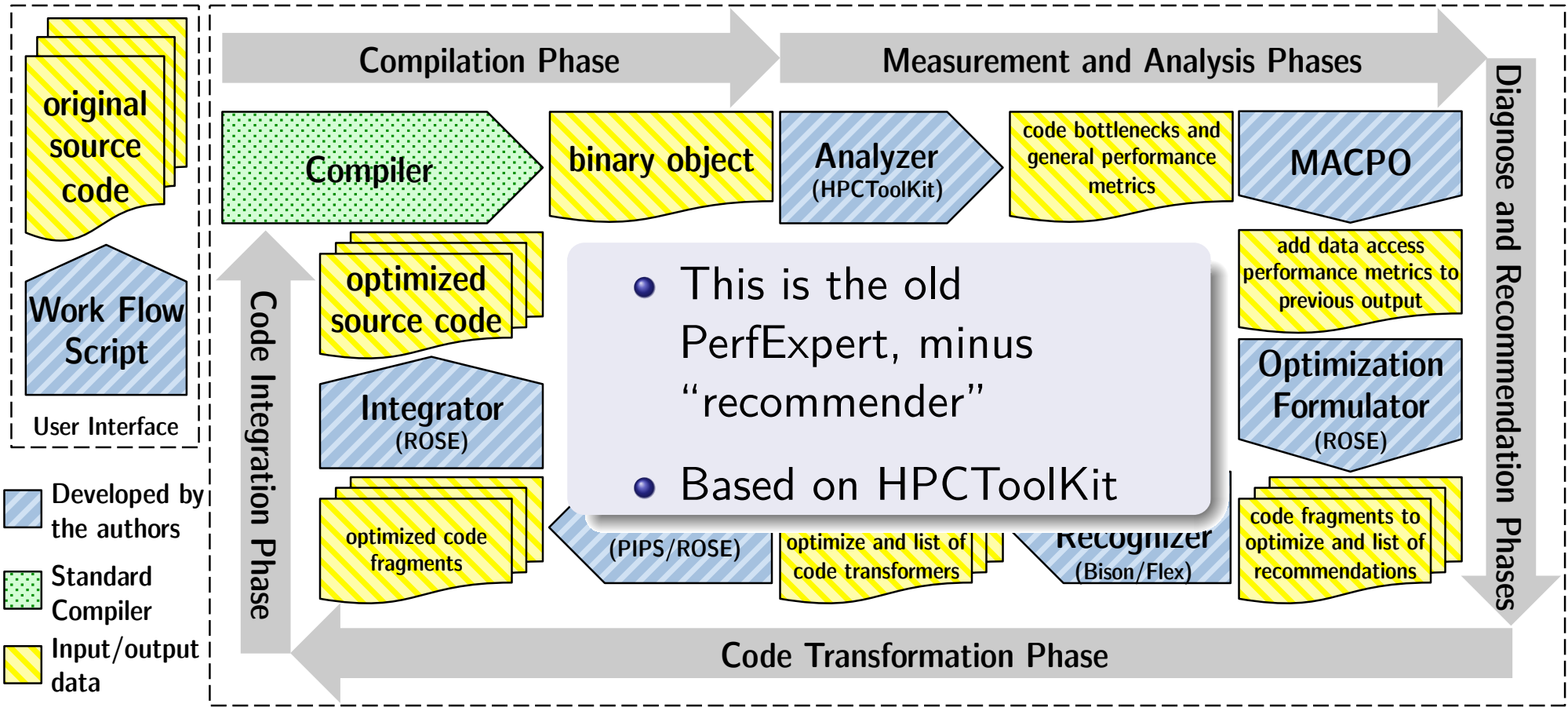
How PerfExpert does that: Work Flow Script



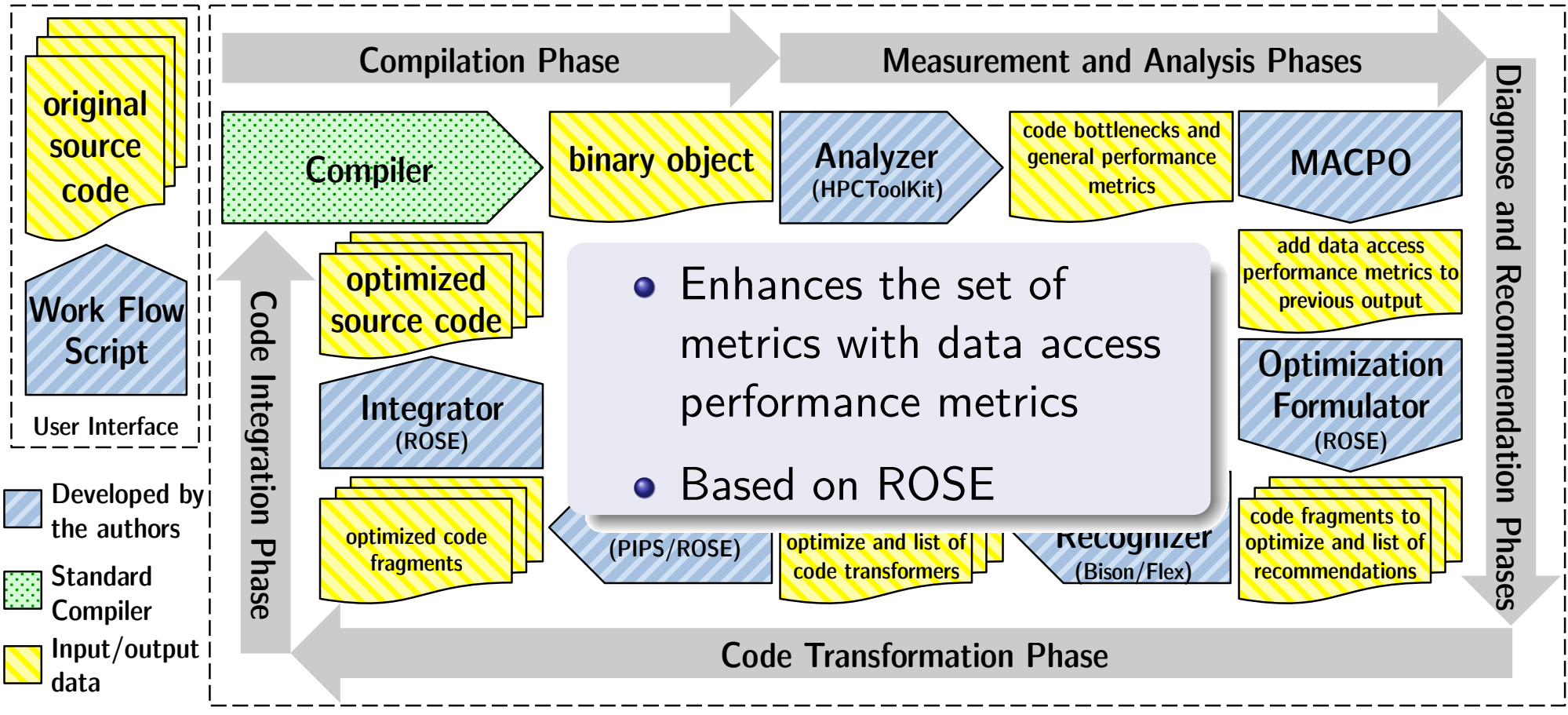
- This is a shell script
- Accepts parameters
- Invokes all tools (including the compiler)
- Backward compatible

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

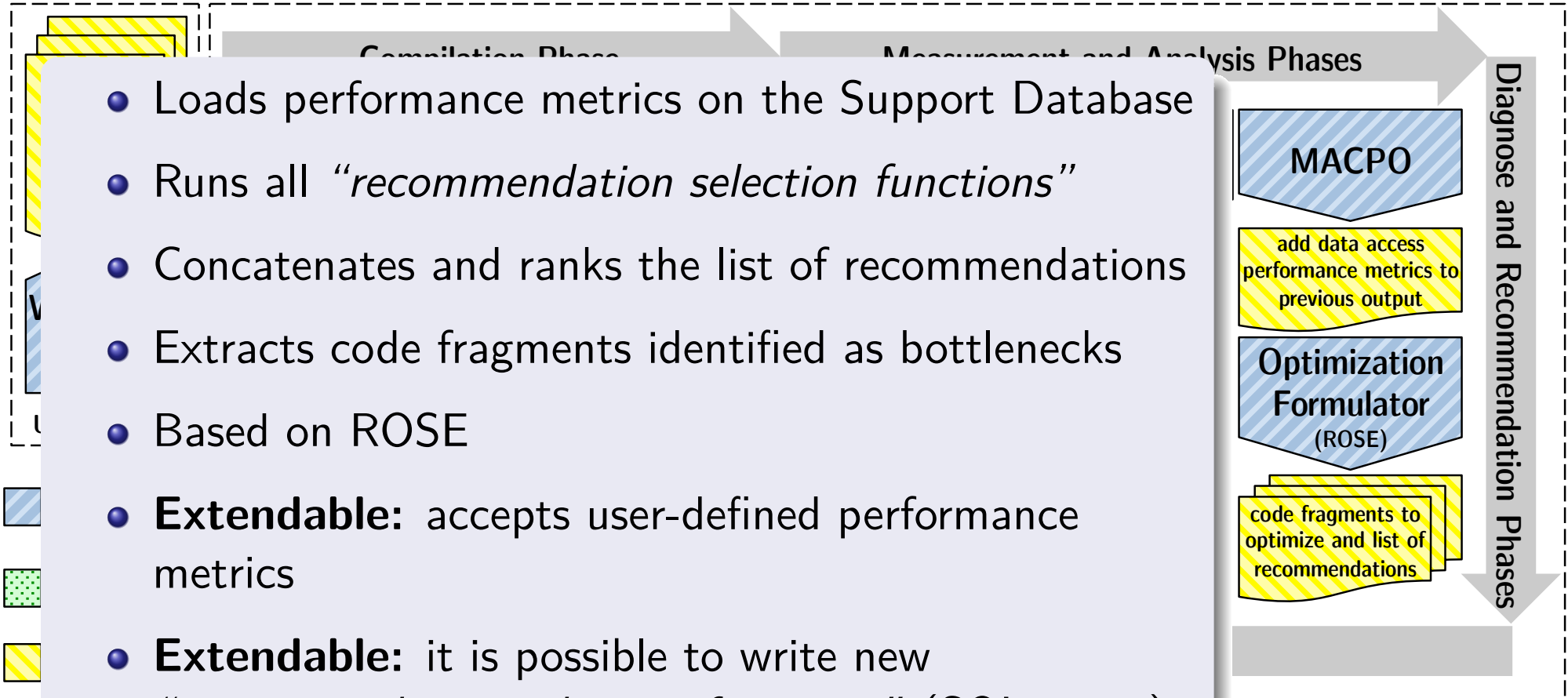
How PerfExpert does that: Analyzer



How PerfExpert does that: MACPO



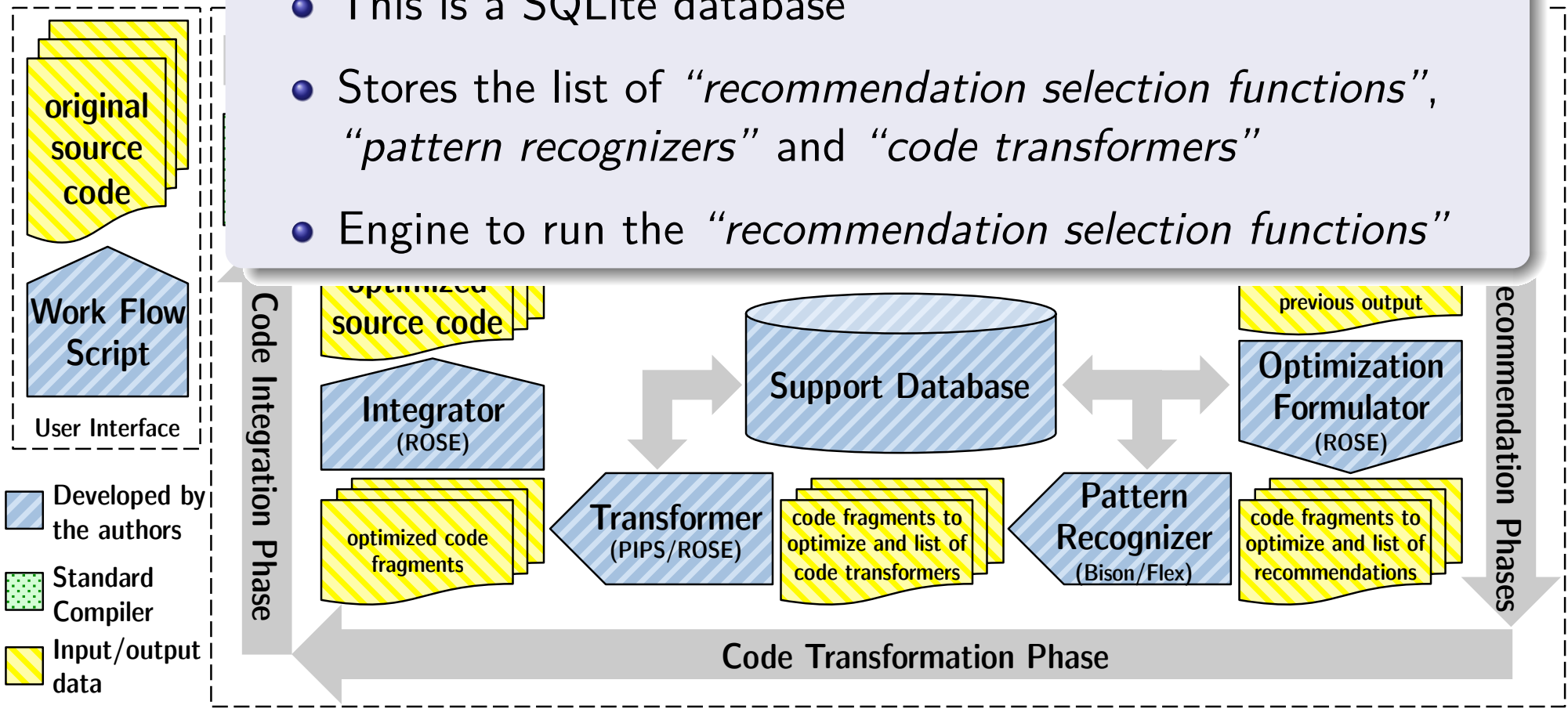
How PerfExpert does that: Optimization Formulator



- Loads performance metrics on the Support Database
- Runs all *"recommendation selection functions"*
- Concatenates and ranks the list of recommendations
- Extracts code fragments identified as bottlenecks
- Based on ROSE
- **Extendable:** accepts user-defined performance metrics
- **Extendable:** it is possible to write new *"recommendation selection functions"* (SQL query)

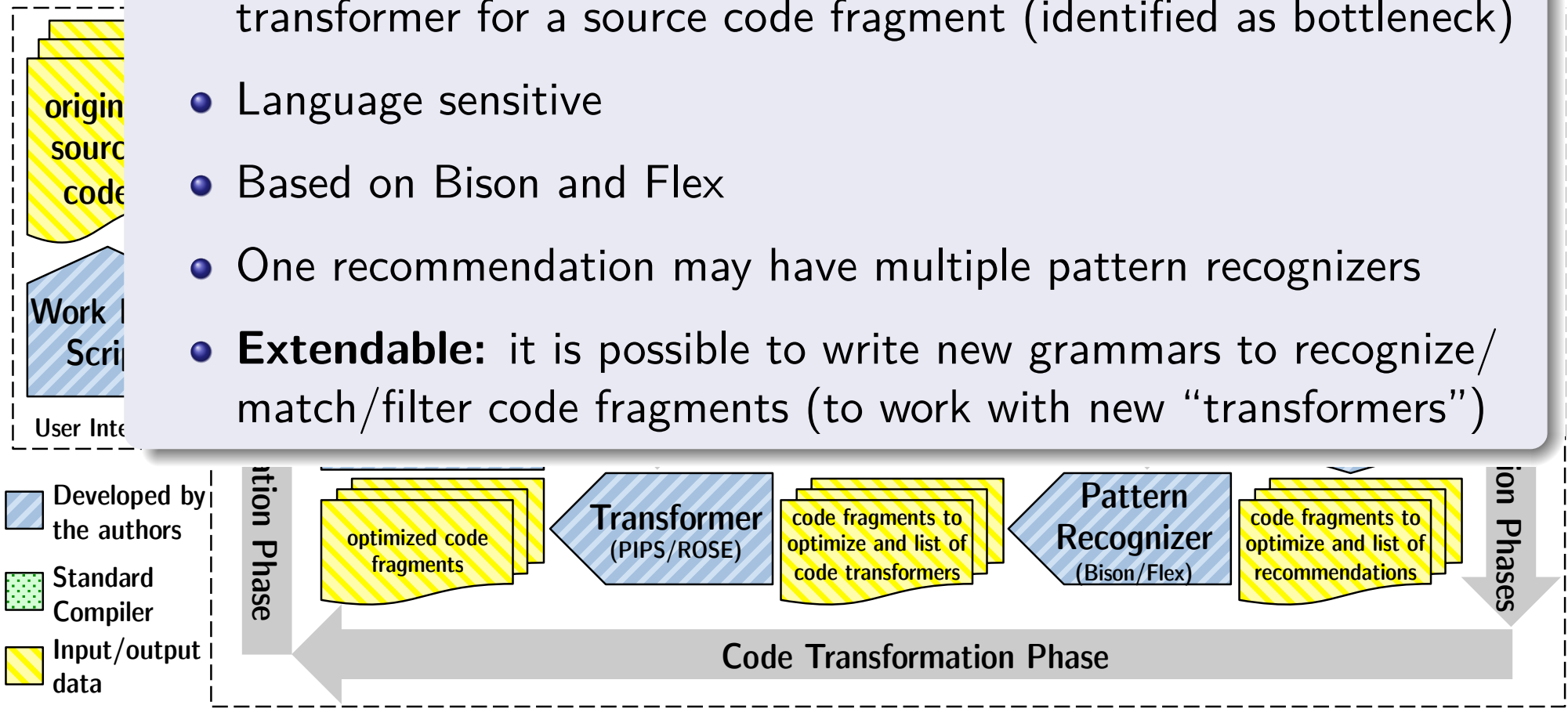
How PerfExpert does that: Support Database

- This is a SQLite database
- Stores the list of *"recommendation selection functions"*, *"pattern recognizers"* and *"code transformers"*
- Engine to run the *"recommendation selection functions"*



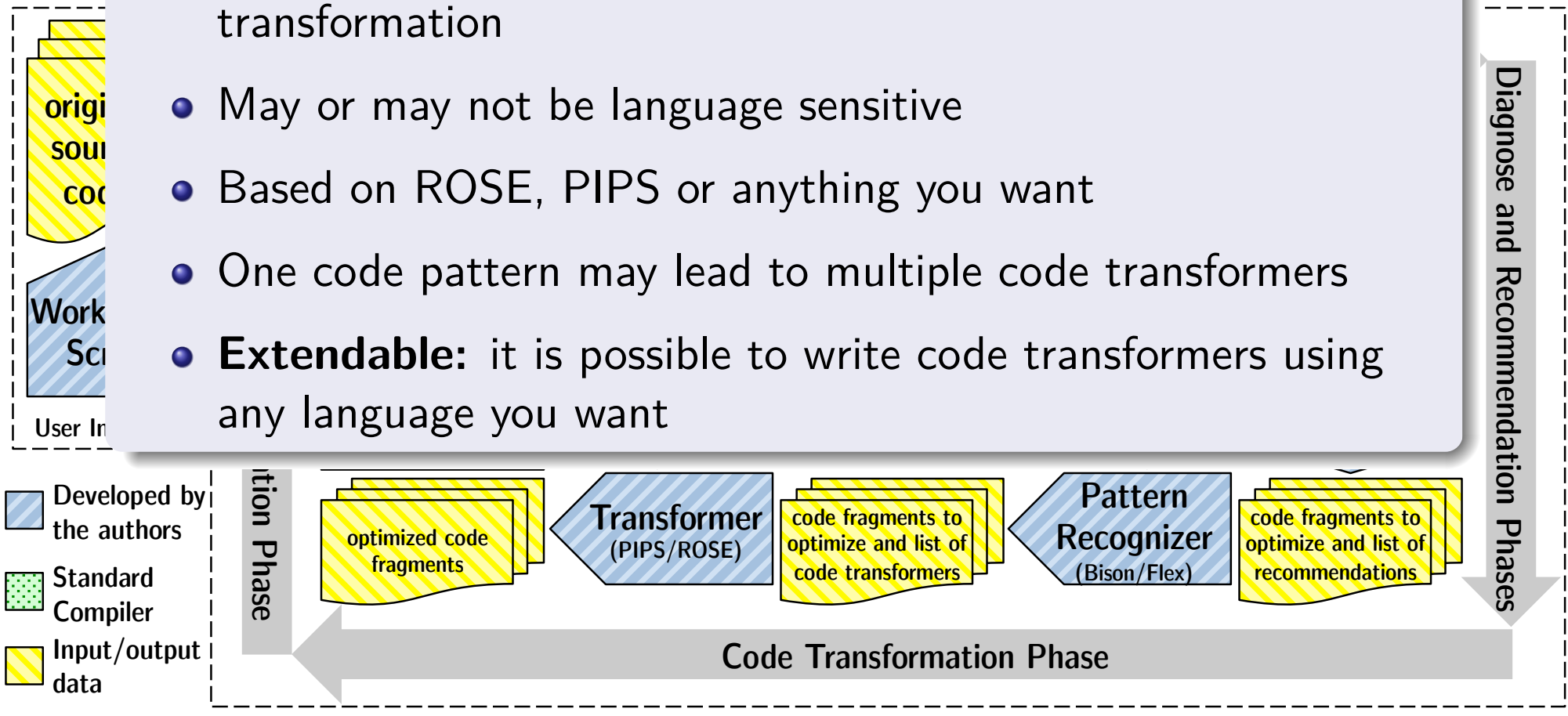
How PerfExpert does that: Pattern Recognizer

- Acts as a "filter" trying to find (match) the right code transformer for a source code fragment (identified as bottleneck)
- Language sensitive
- Based on Bison and Flex
- One recommendation may have multiple pattern recognizers
- **Extendable:** it is possible to write new grammars to recognize/match/filter code fragments (to work with new "transformers")

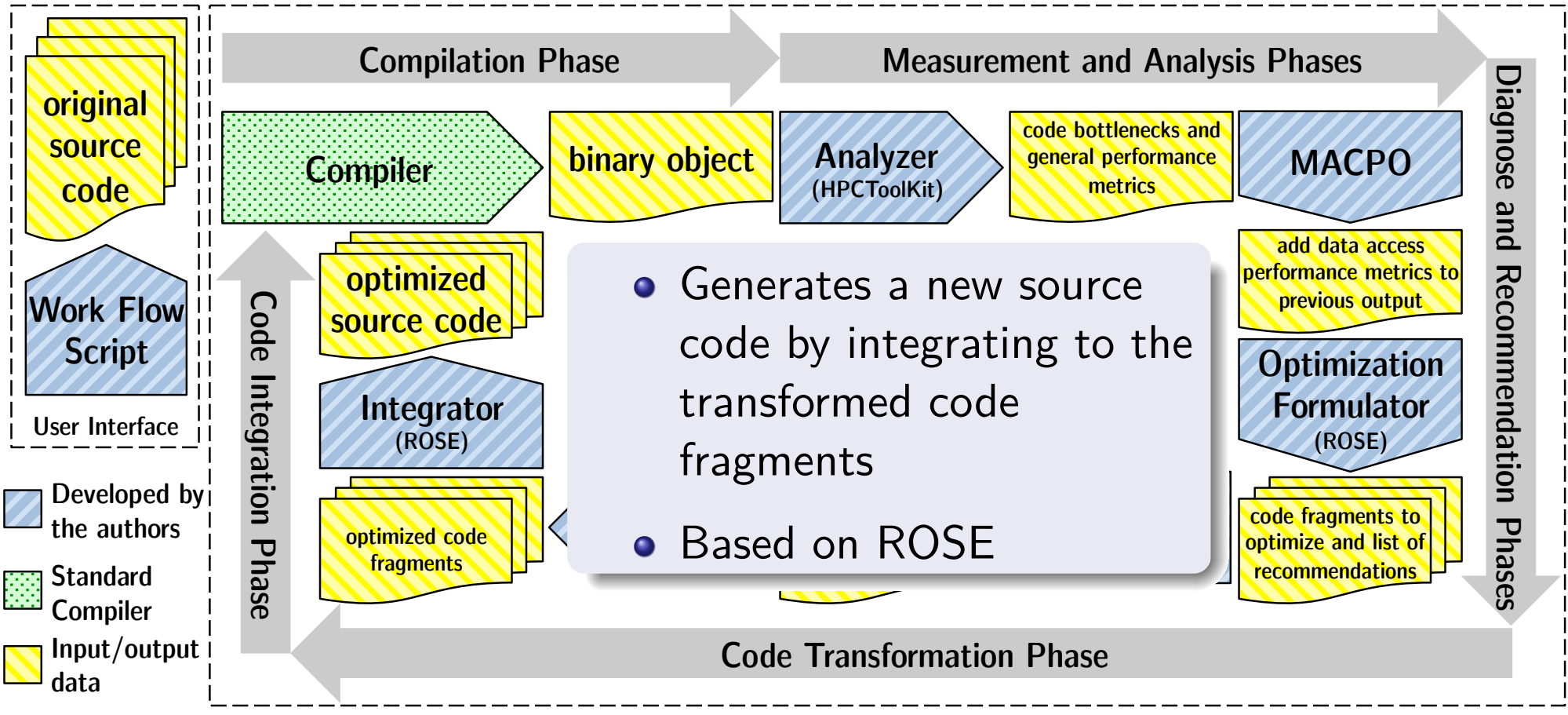


How PerfExpert does that: Transformer

- Implements the recommendation by applying source code transformation
- May or may not be language sensitive
- Based on ROSE, PIPS or anything you want
- One code pattern may lead to multiple code transformers
- **Extendable:** it is possible to write code transformers using any language you want



How PerfExpert does that: Integrator



How PerfExpert does that: Key Points

Why is this performance optimization "architecture" strong?

- Each piece of the tool chain can be updated/upgraded individually
- It is flexible: you can add new metrics as well as plug new tools to measure application performance
- **It is extendable**: new recommendations, transformations and strategies to select recommendations
- Multi-language, **multi-architecture**, open-source and built on top of well-established tools (HPCToolKit, ROSE, PIPS, etc.)
- Easy to use and lightweight!

Agenda

Conclusions

- This is the first end-to-end open-source performance optimization tool (as far as we know)
- It will become more and more powerful as new recommendations, transformations and features are added
- Different from (most of) the available performance optimization tools, there is no "big code" (to increase in complexity until it become unusable or too hard to maintain)

Next Steps

Major Goals

- Improve analysis based on the data access (in progress)
- Increase the number of recommendations and possible code transformations (continuously)
- New algorithms for recommendations selection (in progress)
- Add support to MIC architecture (in progress)
- Add support to MPI-related recommendations (medium term)
- Add support to MPI-related code transformations (long term)

Next Steps

Minor Goals

- Support "Makefile"-based source code/compilation tree (done!)
- Make the required packages installation process easier (done!)
- Add a test suite based on established benchmark codes (in progress)
- Easy-to-use interface to manipulate the support database (medium term)

Thank You

fialho@utexas.edu