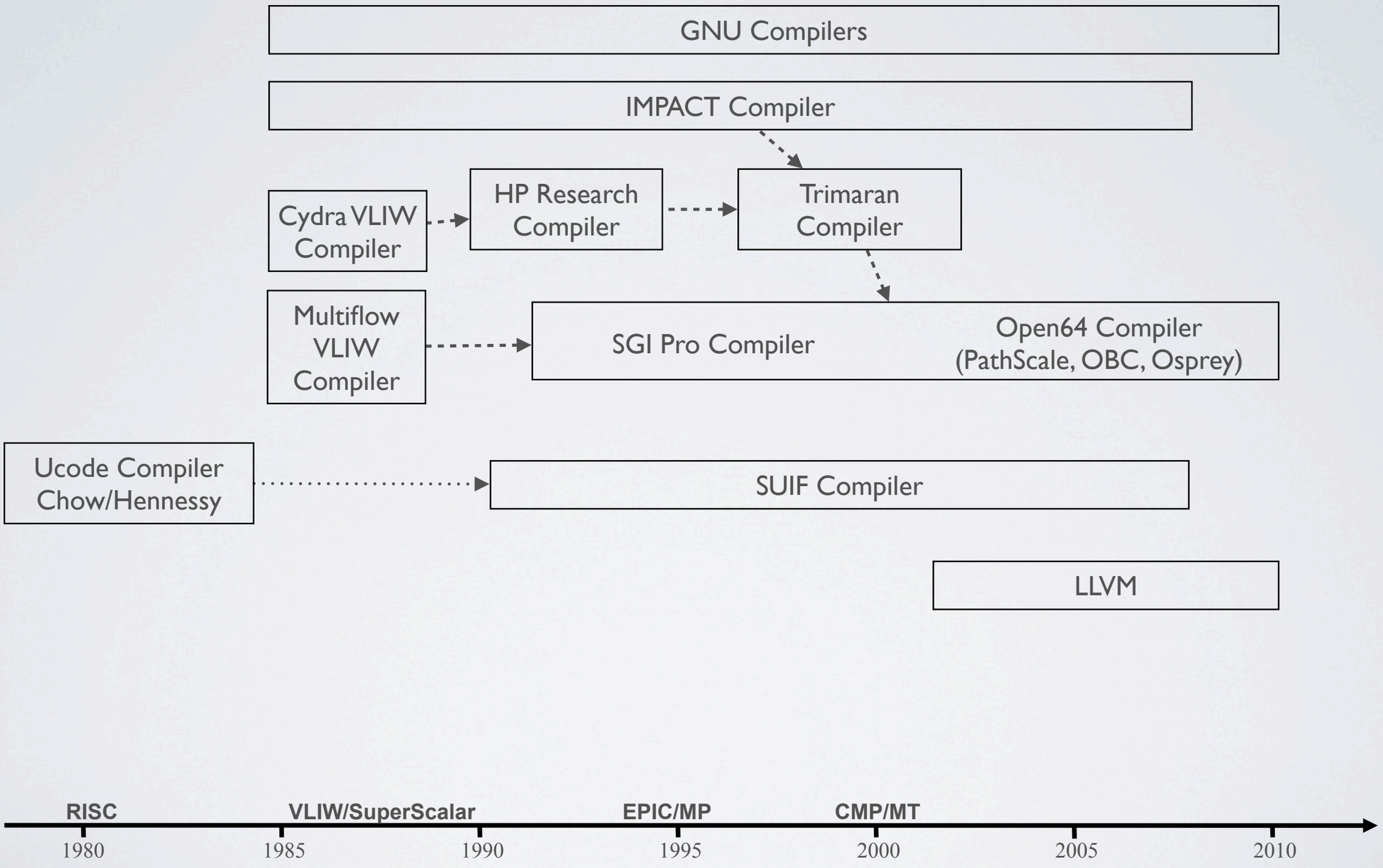
A SURVEY OF PACE AND LLVM

CPEG 621

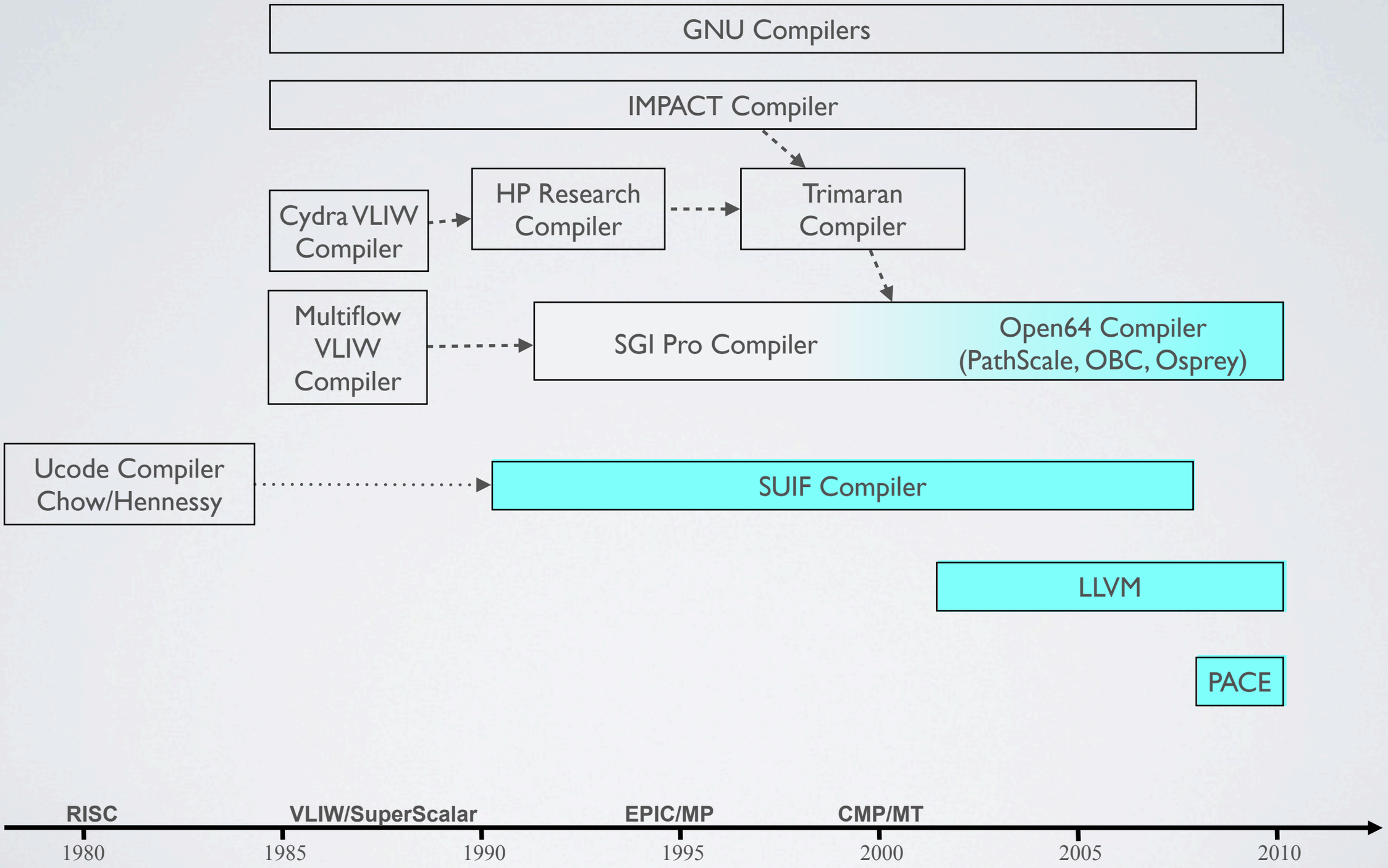
Kirk Kelsey — ET International

October 27th 2010

A MAP OF MODERN COMPILER PLATFORMS



A MAP OF MODERN COMPILER PLATFORMS



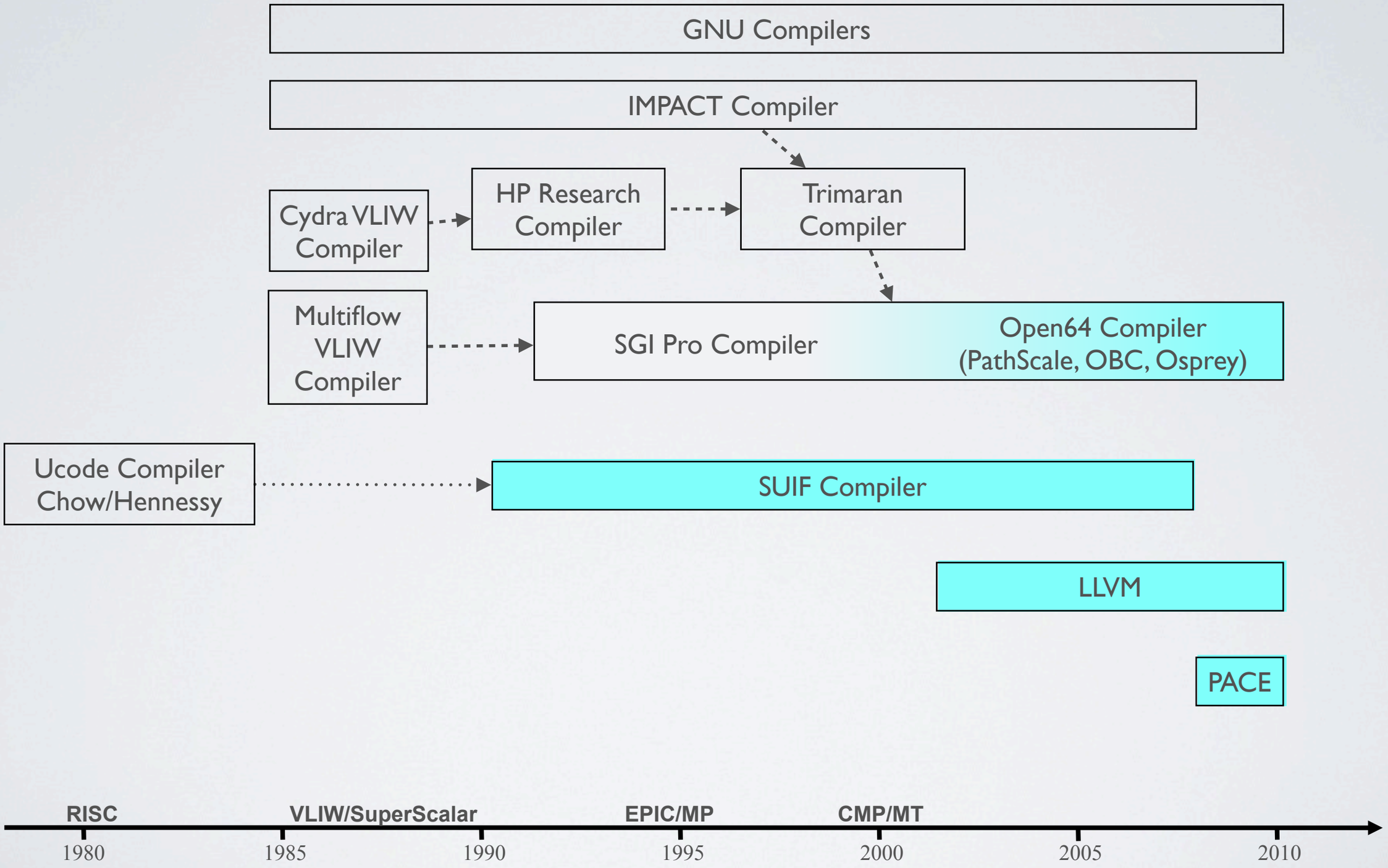
OUTLINE

- Open 64
- National Compiler Infrastructure Project
- Architecture-Aware Compiler Environment
- Low-Level Virtual Machine

OPEN 64

Xiaomi An

A MAP OF MODERN COMPILER PLATFORMS



NATIONAL COMPILER INFRASTRUCTURE PROJECT

SUIF

- Stanford University Intermediate Format
  - also: Harvard, Rice, MIT, and the Portland Group
- Funded by DARPA and NSF
- Leveraged GCC and EDG

AACE

- Architecture Aware Compilation Environment
- Motivation:
  - processor design complexity continues to grow
  - performance tuning an application is directly related
  - compilers are the gatekeepers and bottlenecks

http://www.darpa.mil/tcto_aace.html

THE PACE PROJECT

- $16 million DARPA award
- Headed by researchers at Rice University
  The Ohio State University, Stanford University, Texas Instruments
- Focused on reducing compiler development time
  takes ~5 years to develop a compiler for a new architecture

http://pace.rice.edu/

PACE

OBJECTIVES

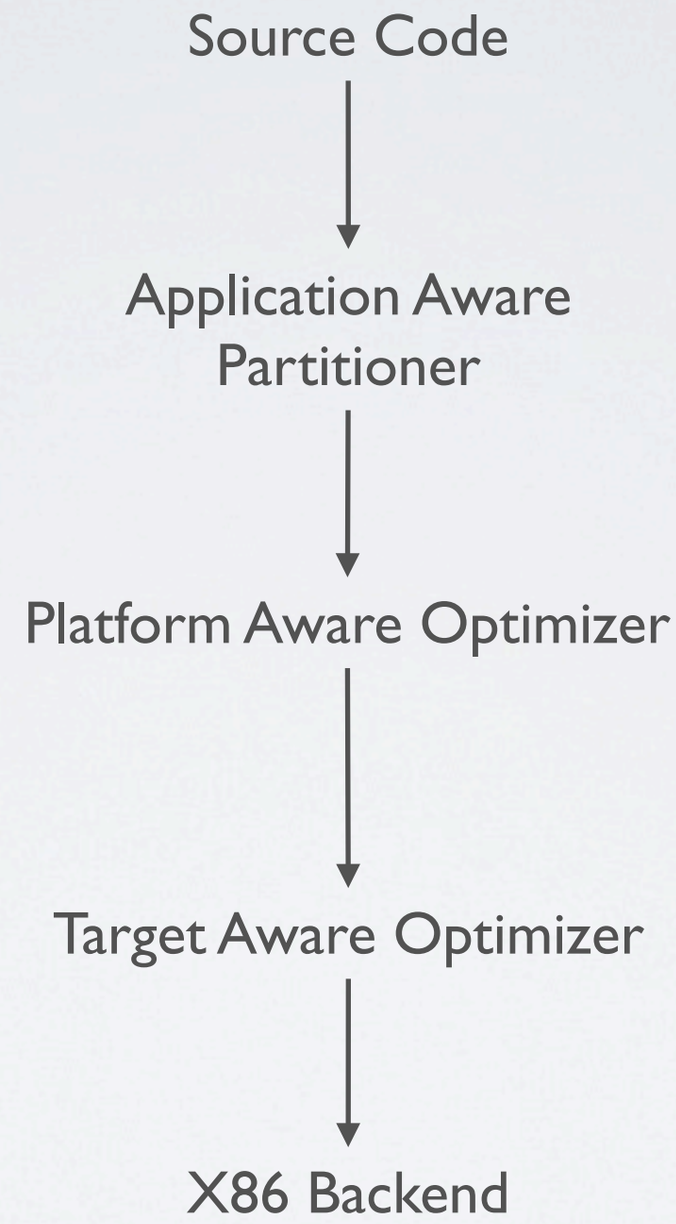
- Understand the compilation
  - the capabilities of the compiler
  - how the application is used
  - the attributes of the hardware
- Support multiple levels of transformation each based on different characteristics
- Use feedback from past compilation

PACE COMPILER

HIGH LEVEL DESIGN

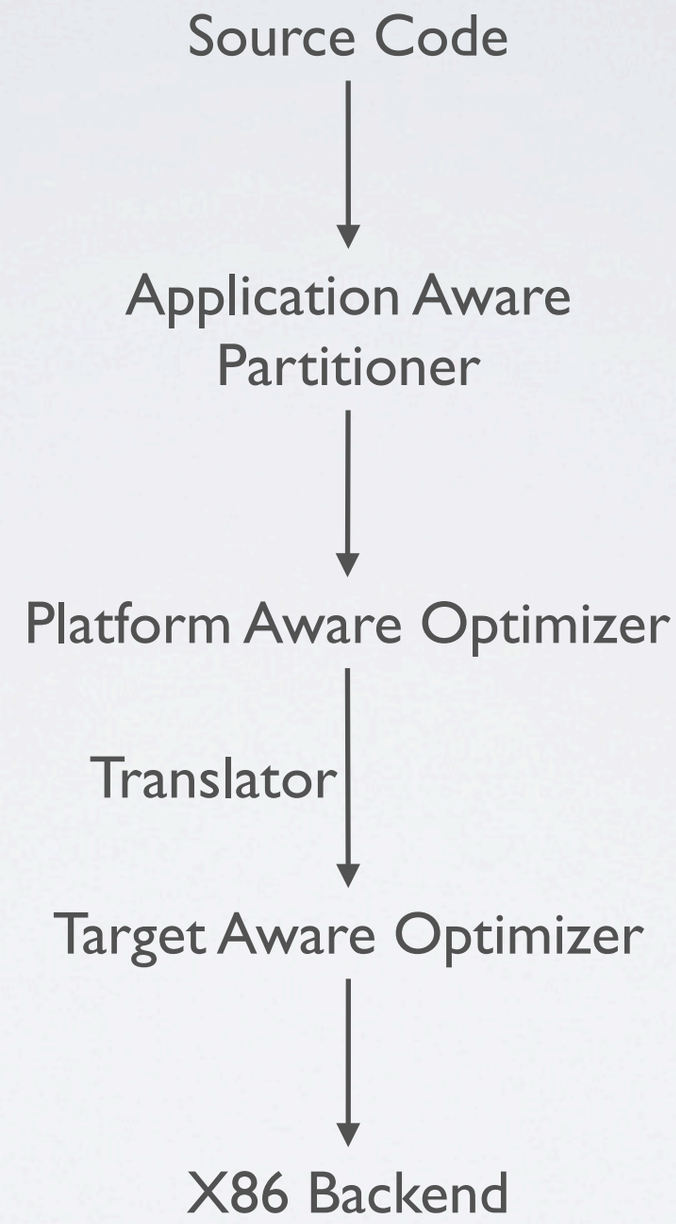
PACE COMPILER

HIGH LEVEL DESIGN



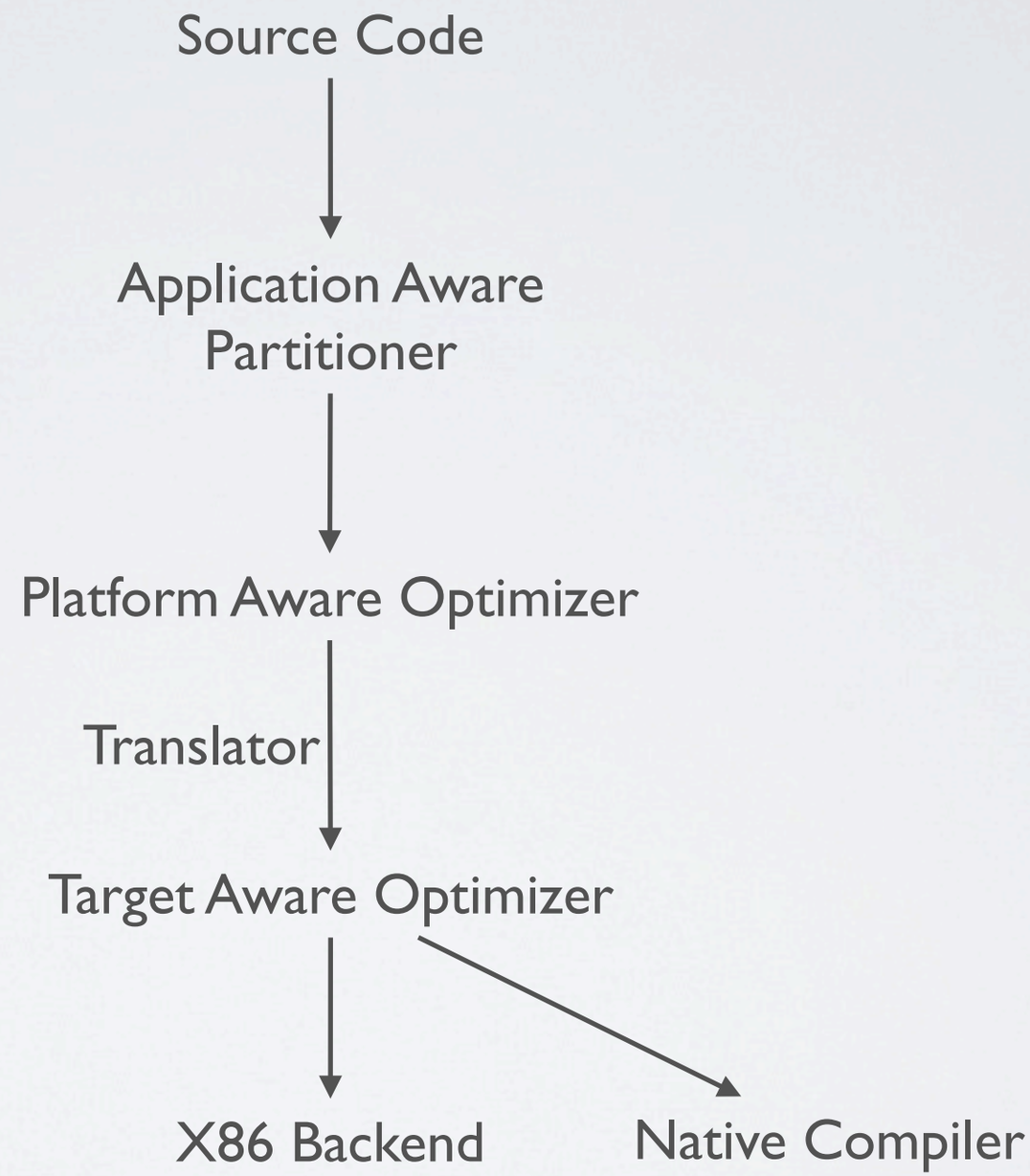
PACE COMPILER

HIGH LEVEL DESIGN



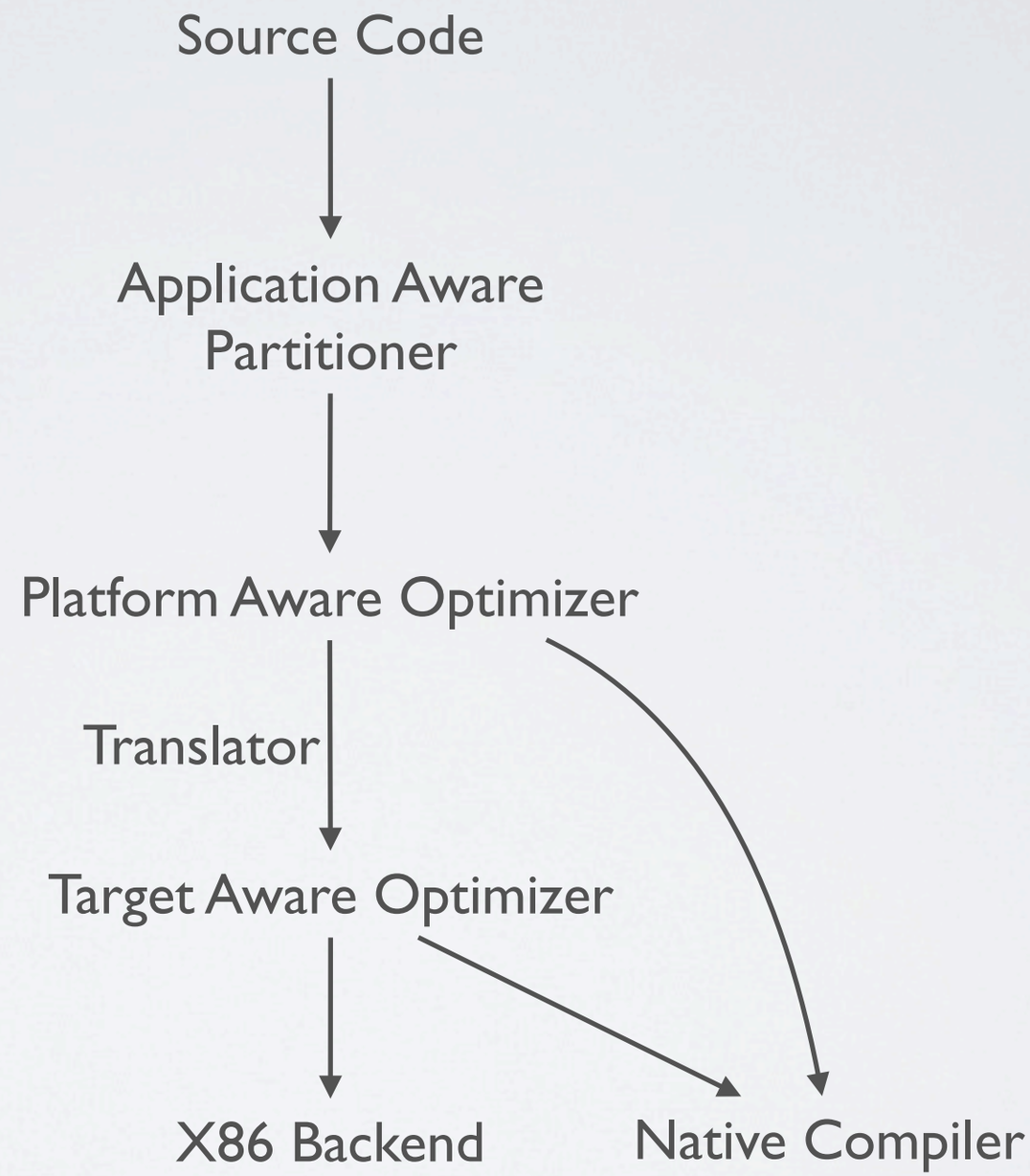
PACE COMPILER

HIGH LEVEL DESIGN



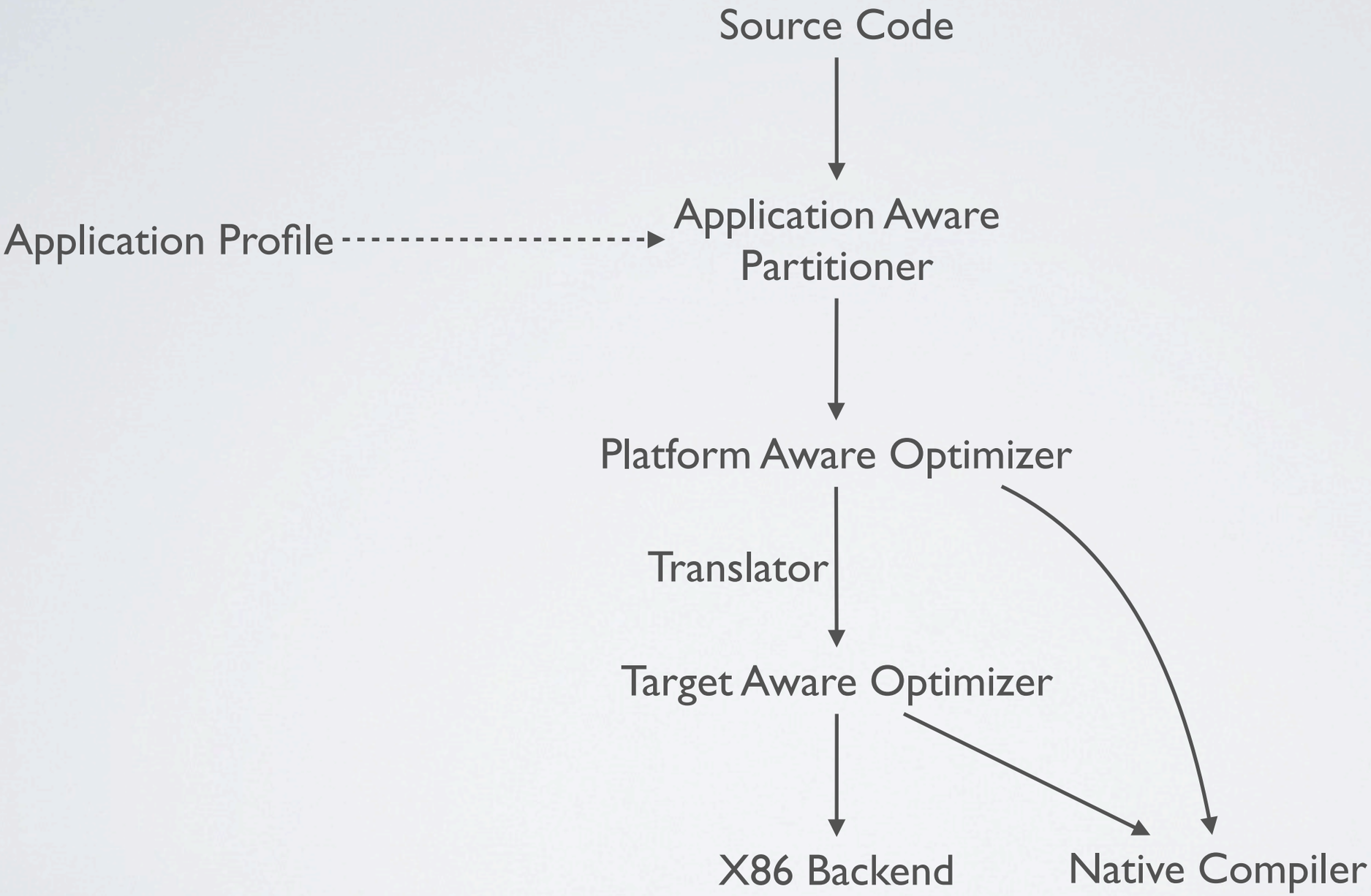
PACE COMPILER

HIGH LEVEL DESIGN



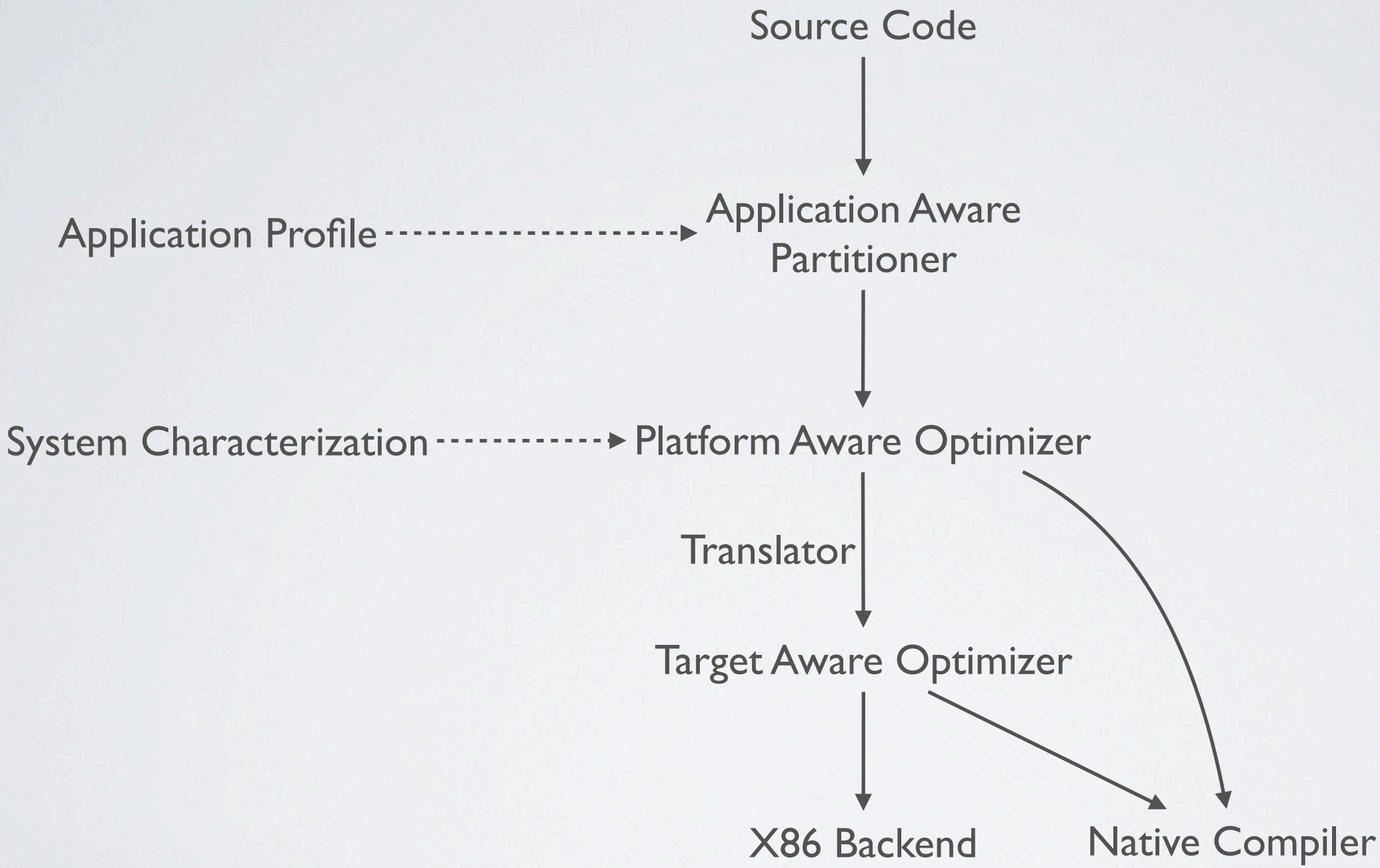
PACE COMPILER

HIGH LEVEL DESIGN



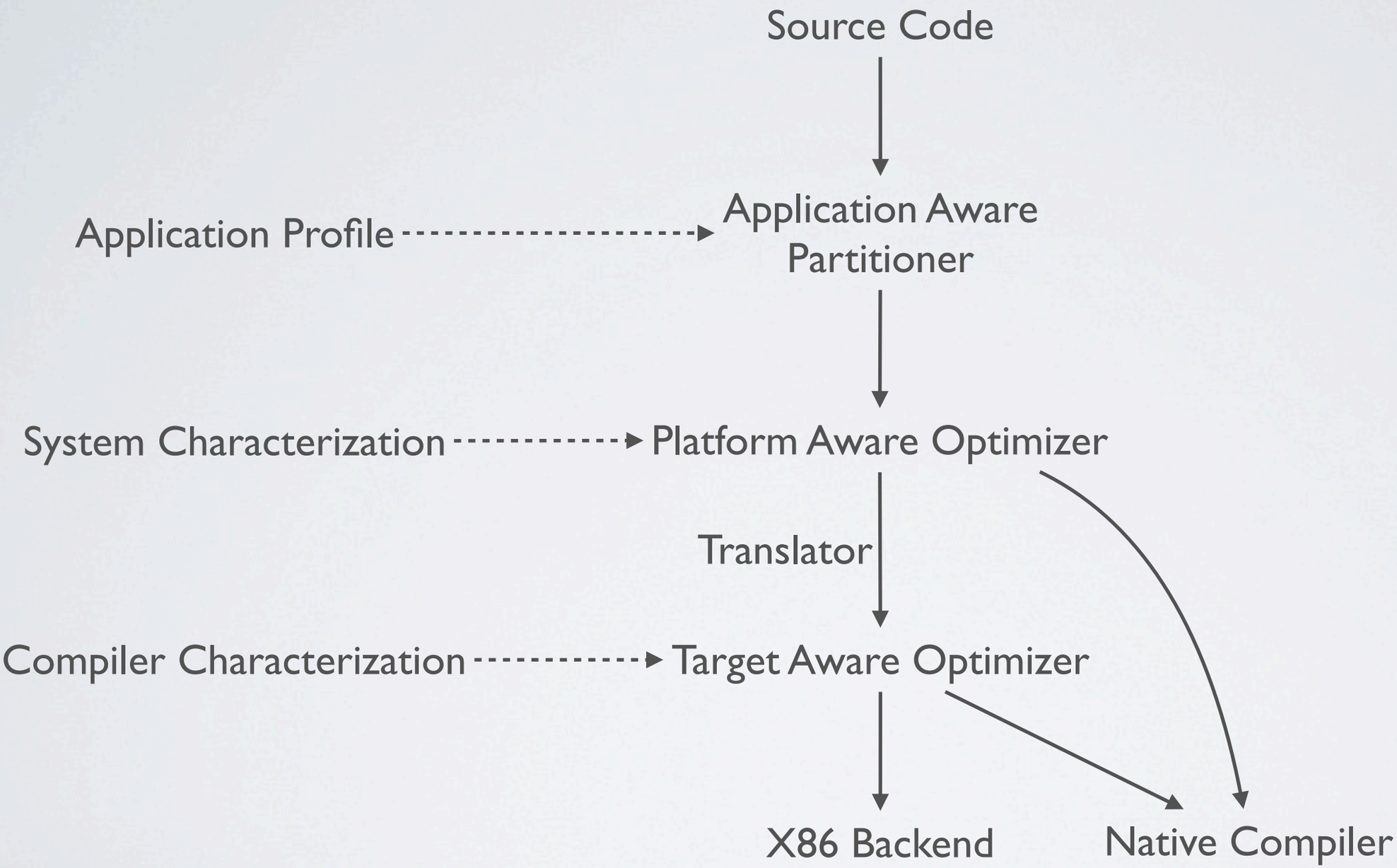
PACE COMPILER

HIGH LEVEL DESIGN



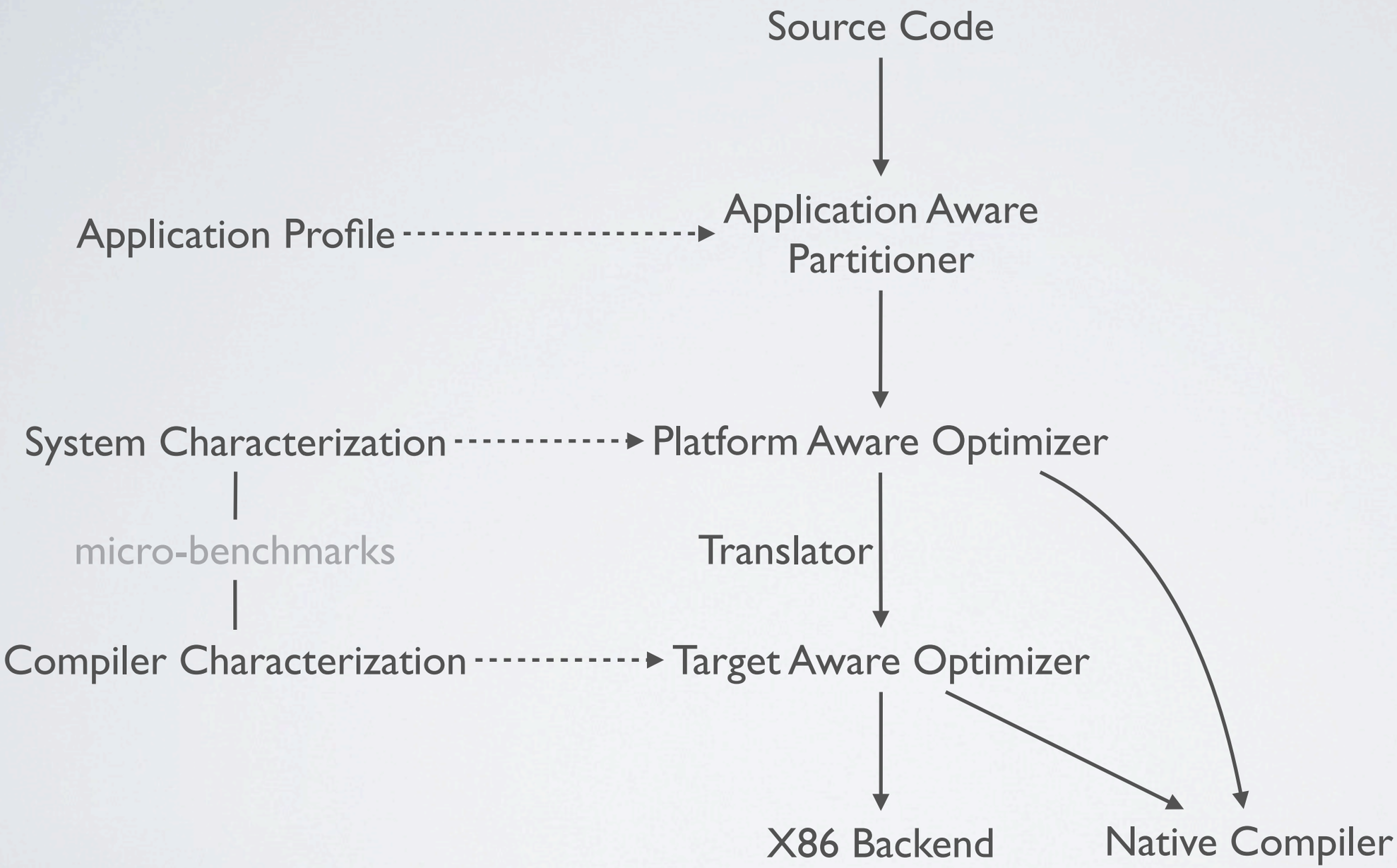
PACE COMPILER

HIGH LEVEL DESIGN



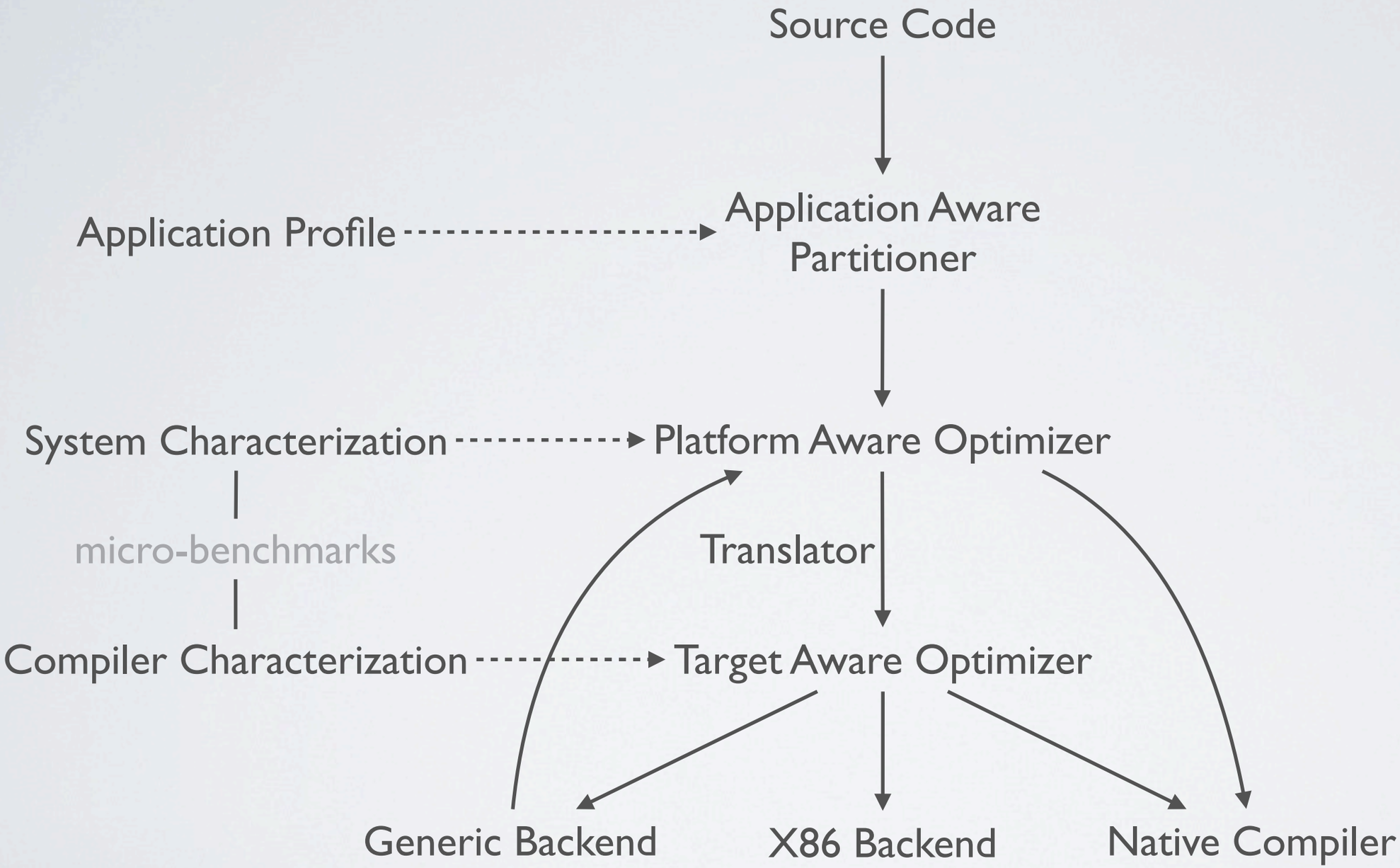
PACE COMPILER

HIGH LEVEL DESIGN



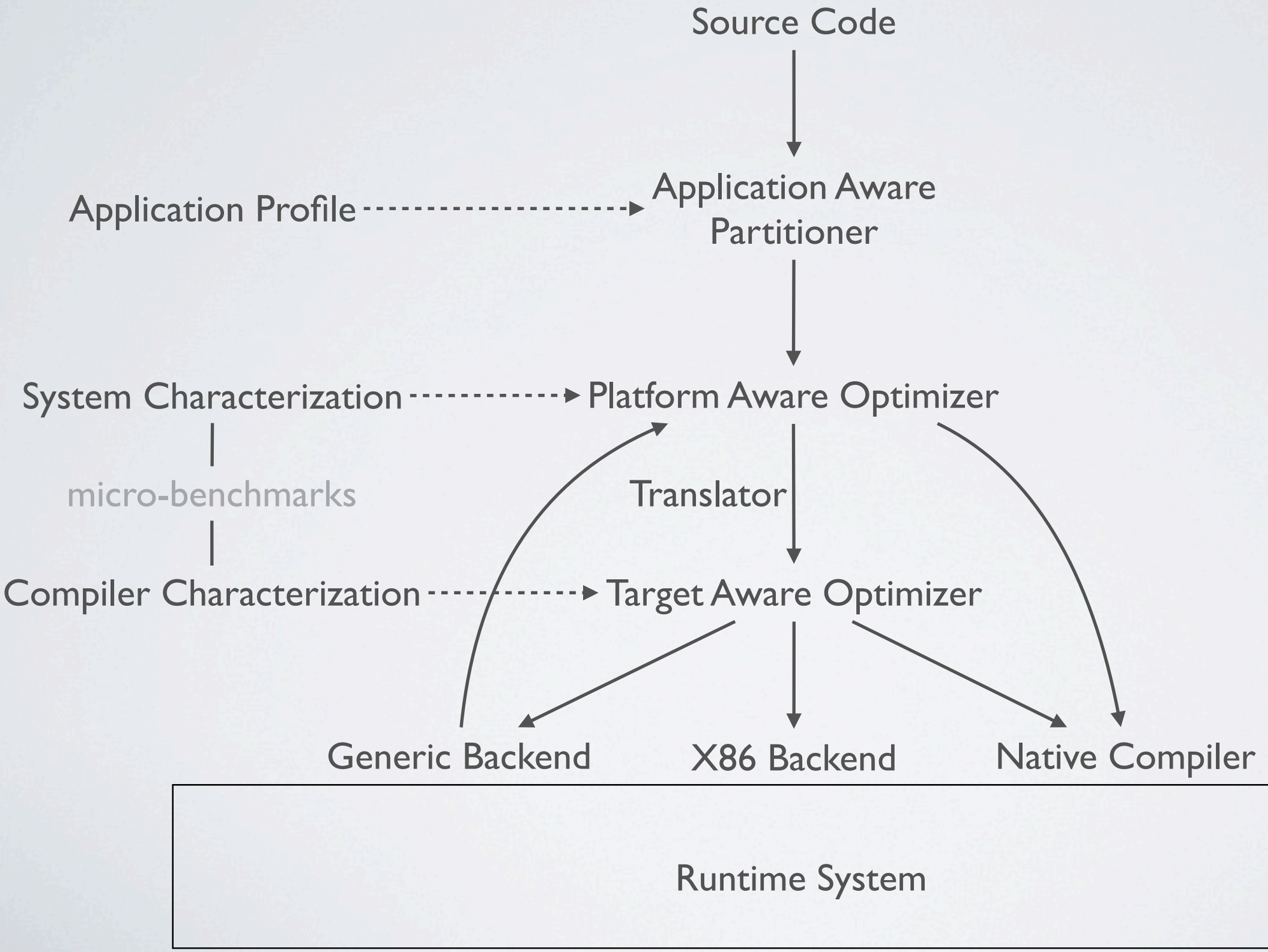
PACE COMPILER

HIGH LEVEL DESIGN



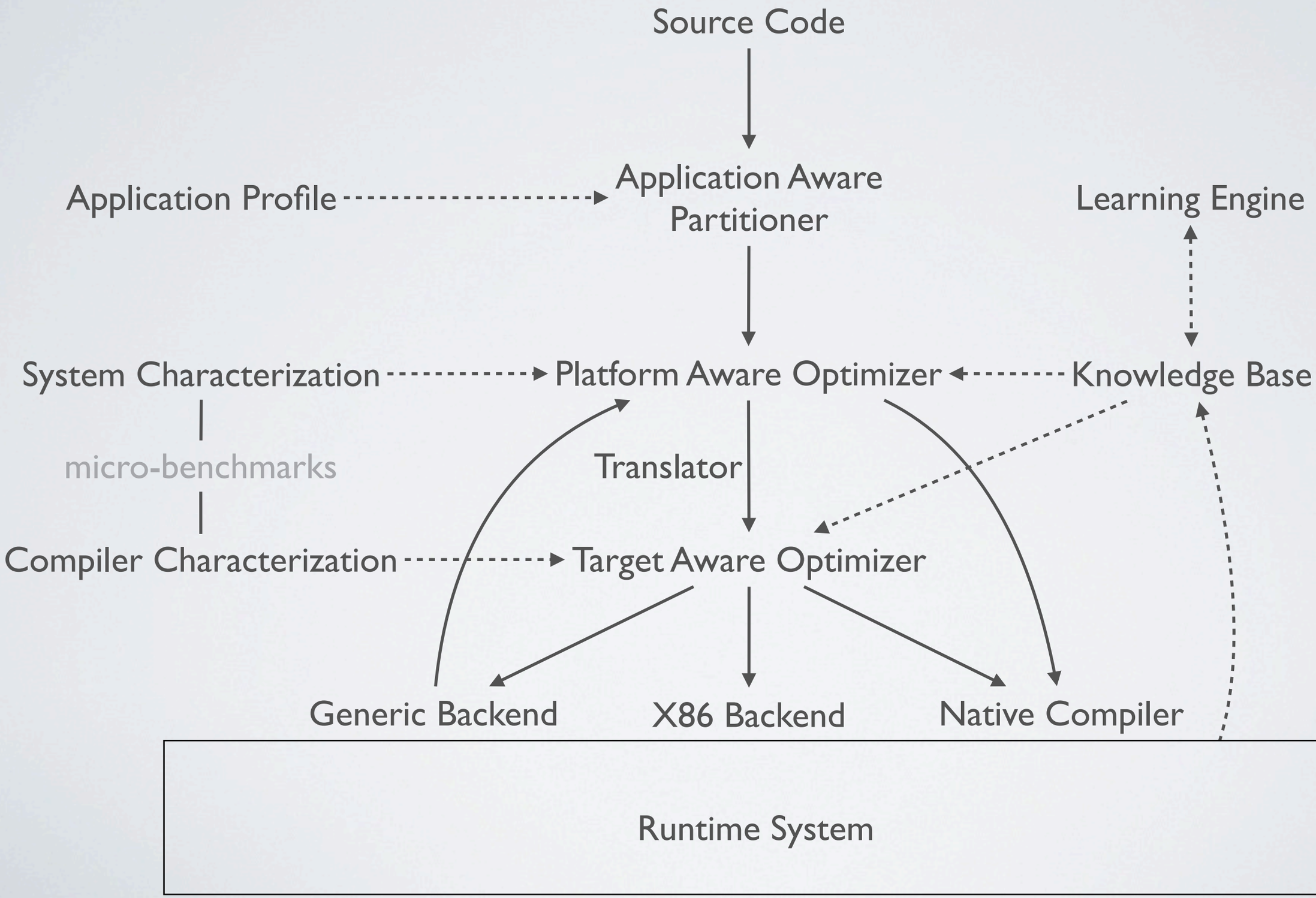
PACE COMPILER

HIGH LEVEL DESIGN



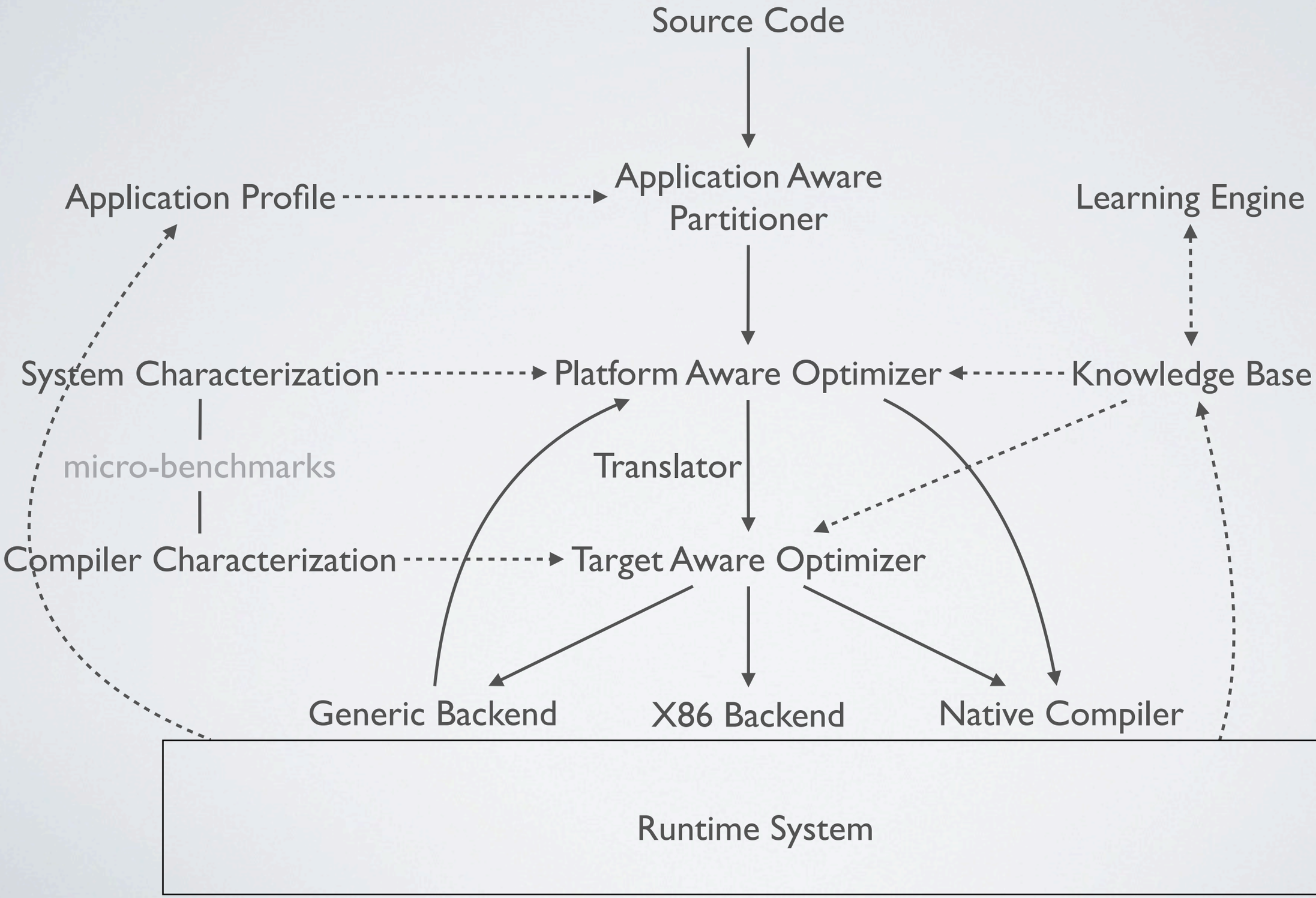
PACE COMPILER

HIGH LEVEL DESIGN



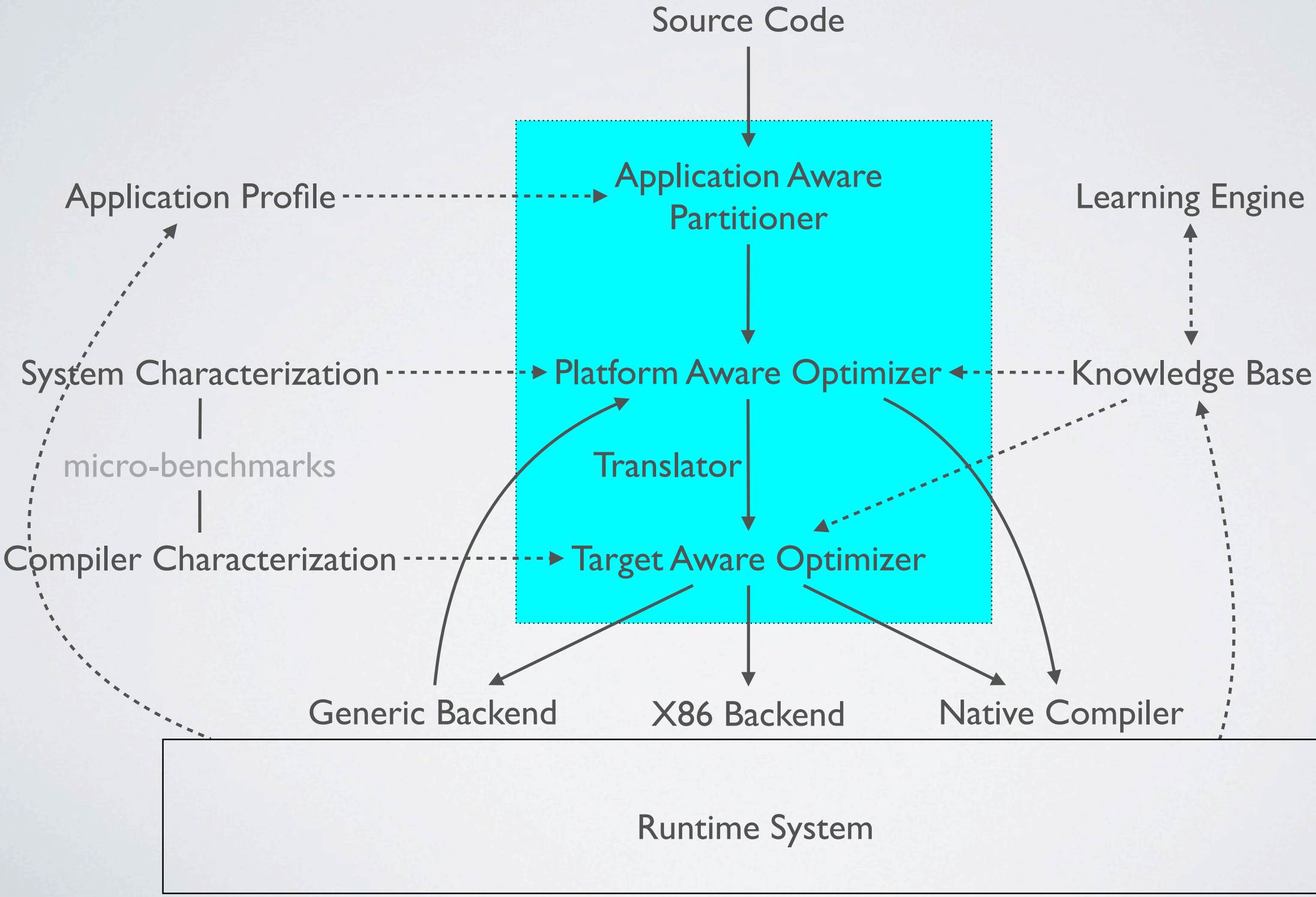
PACE COMPILER

HIGH LEVEL DESIGN



PACE COMPILER

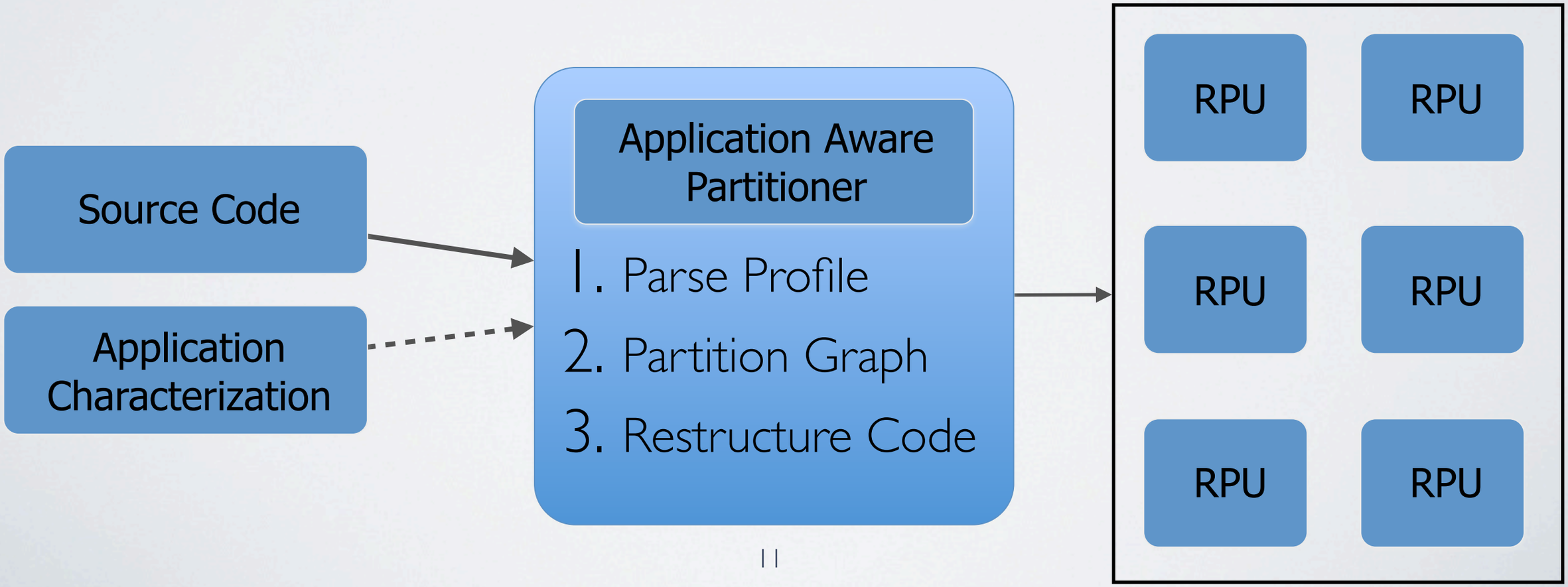
HIGH LEVEL DESIGN



AAP: INPUT & OUTPUT

- Input
  - program source code
  - application characterization — execution profiles
  - native compiler characterization

- Output
  - Refactored Program Units (RPUs)



PLATFORM AWARE OPTIMIZER

- Understands the target system attributes
  - relies on micro-benchmarks that extract the information

- Performs AST based transformations
  - data flow analysis, abstract interpretation, symbolic execution
  - value-based specialization
    based on program data values and addresses
  - polyhedral and non-polyhedral loop transformations
  - data transformations:
    privatization, scalar replacement, alignment, padding, array and struct redistribution

TARGET AWARE OPTIMIZER

- low-level optimization tailor the code to specific target
  - measured capacities (e.g., Maxlive, # issue slots)
  - optimizations native compiler lacks (e.g. software pipelining)

- provides feedback to PAO on effectiveness of transformations
  - scheduler sees too little ILP → ask PAO to generate more
  - register pressure high in a loop → ask PAO to tone it down
  - decides to pipeline loop → ask PAO to remove control flow

- generates assembly code for various strains of IA32
  - IA32 is sufficiently popular to merit the attention
  - chance to experiment with static and runtime optimizations
  - proof of concept for characteristic-driven optimization

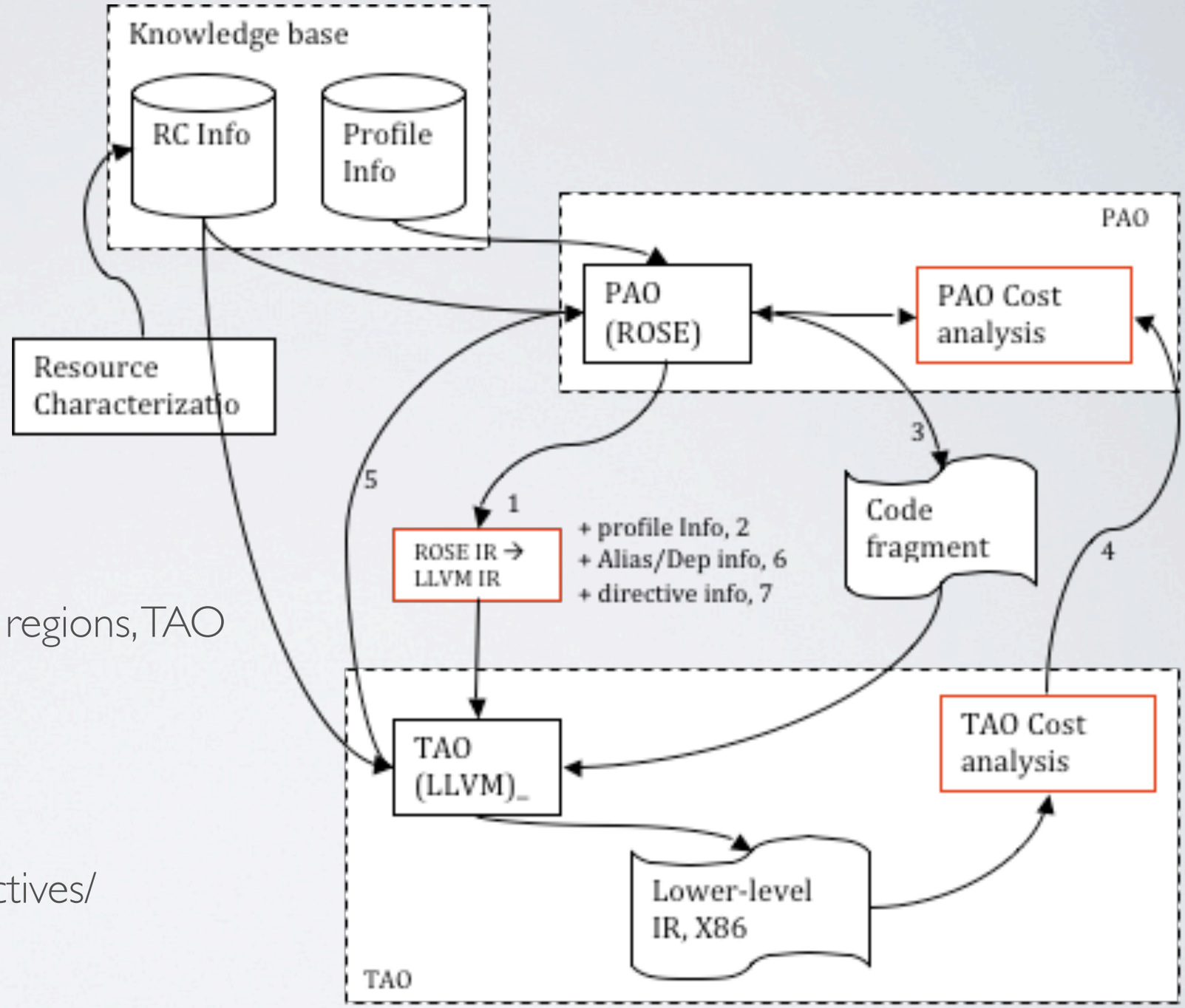
PAO TO TAO

QUERY INTERFACE

- PAO queries TAO for expected performance
  - Encapsulates code in a synthetic function with a query object
- TAO applies standard scalar compiler optimizations
  - Driven by processor characteristics
  - Uses a pseudo-backend based on the Resource Characteristics
  - Records static metrics for low-level code in the query
- PAO interprets query results and chooses transformation parameters based on TAO feedback
  - E.g., choosing unroll factors for each loop in a loop nest

PAO-TAO INTERFACES

1. IR Translation PAO→TAO
2. Profile Info PAO → TAO
3. Code fragments PAO→TAO
4. Cost Analysis TAO→PAO
5. Rate-limiting resource for code regions, TAO → PAO
6. Alias/Dep Info
7. Per-region control options/directives/ annotations
8. IR Translation PAO→TAO



PACE

GENERIC BACKEND

- Generalizes some target attributes
  - Register availability & aliasing
  - Instruction latency
  - Functional unit availability
- Supports the query answering mechanism
- Used for synthetic code generation
  (not actual compilation)

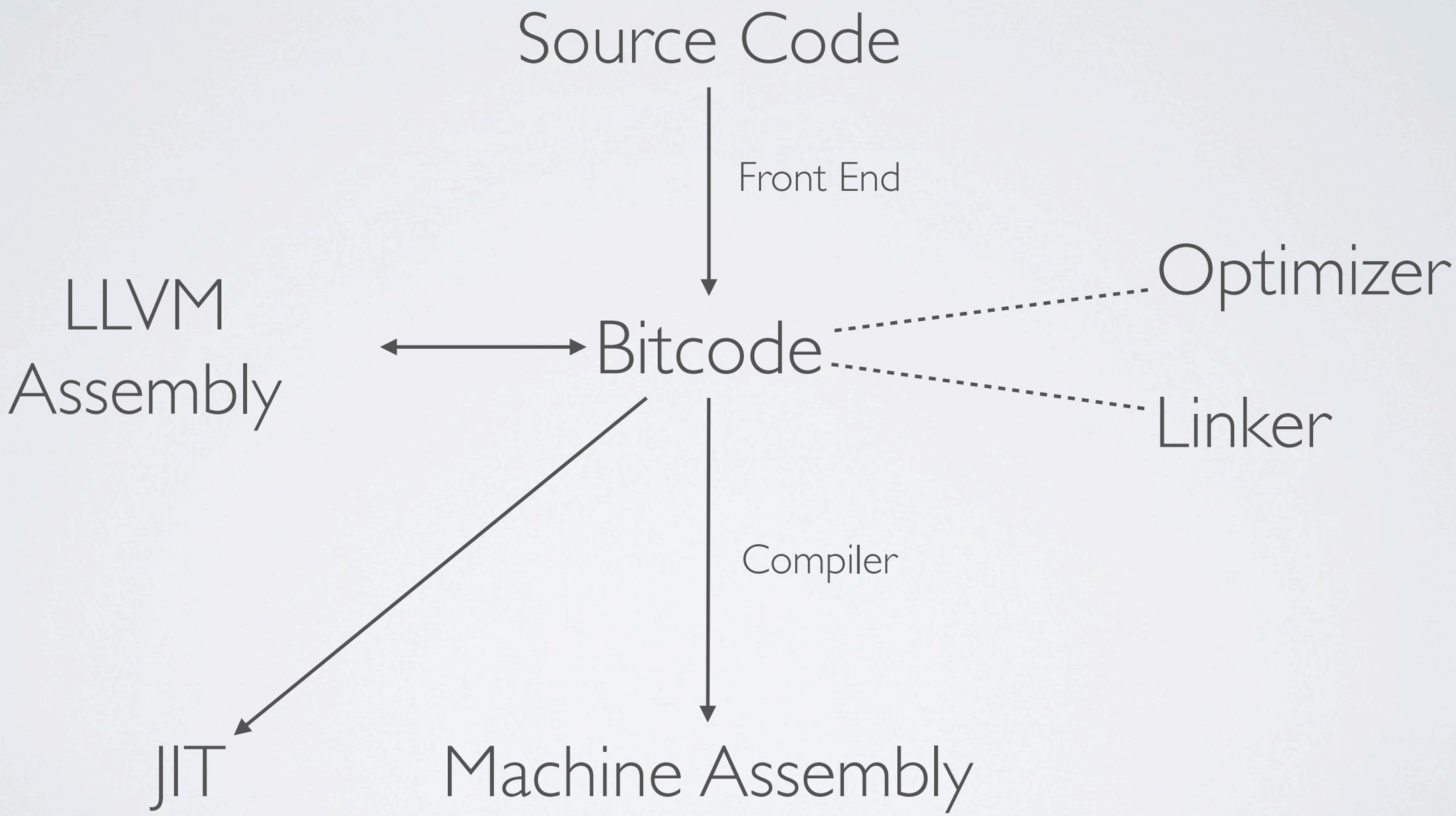
LLVM

LLVM

ORIGINS & OVERVIEW

- LLVM is:
  - not a virtual machine
  - a generic, low-level intermediate representation
  - a complete compiler tool-chain
- Developed at UIUC

LLVM STRUCTURE



LLVM

FRONT END

- Generates LLVM IR
- There are three available:
  - LLVM-gcc Modified version of GCC 4.2.1
  - Clang C Language Front End (and Obj-C, C++)
  - Dragon Egg plugin to GCC versions 4.5 and later

LLVM

CLANG

- Supports development of multiple client types
  - static analyzers, rewriters, code generation
- Compatible with GCC
  - Detailed and intuitive diagnostic messages
  - Better IDE support

LLVM

INTERMEDIATE REPRESENTATION

- Used in three forms
  - (in memory, on disk, human readable)
- SSA Based, low-level representation
- Typed and extensible
- Expressive
  - Multiple linkage types (visibility)
  - Multiple calling conventions

LLVM

PHI EXAMPLE

Syntax:

<result> = phi <ty> [<val0>, <label0>], ...

```
; Infinite loop that counts from 0 on up...
Loop:
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

HELLO WORLD

```
; Declare the string constant as a global constant.
@.LC0 = internalconstant[13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8*)

; Definition of main function
define i32 @main() {   ; i32()*
  ; Convert [13 x i8]* to i8  *...
  %cast210 = getelementptr [13 x i8]* @.LC0, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)                ; i32
  ret i32 0
}
; Named metadata
!1 = metadata !{i32 41}
!foo = !{!1, null}
```

LLVM
MIDDLE END

OPT

- Supports dynamically loaded passes
- All passes are automatically scheduled
- Provides analysis to later passes
  (alias, dependence, pointer bounds)
- Provides transformation passes
  (dead code, loop {invariant motion, unrolling, switching})
- Results in a new bitcode file

WRITING AN OPT PASS

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
struct Hello : public FunctionPass {
   static char ID;
   Hello() : FunctionPass(&ID) {}
      virtual bool runOnFunction(Function &F) {
      errs() << "Hello: " << F.getName() << "\n";
      return false;
   }
};
char Hello::ID = 0;
INITIALIZE_PASS(Hello, "Hello", "Hello World Pass", false, false);
```

BACKEND

LLVM TARGET INDEPENDENT

CODE GENERATOR

- Abstract target description
- Classes to represent the code being generated
- Representation of assembly level constructs

  (labels, sections, and instructions)
- Target-independent algorithms
- Target description interfaces for particular targets
- The target-independent JIT components

CODE GENERATION

STAGES

- Instruction Selection
- Scheduling and Formation.
- SSA-based Machine Code Optimizations
- Register Allocation
- Prolog/Epilog Code Insertion
- Late Machine Code Optimizations
- Code Emission

CODE GENERATION

STAGES

- Instruction Selection
- Scheduling and Formation.
- SSA-based Machine Code Optimizations
- Register Allocation
- Prolog/Epilog Code Insertion
- Late Machine Code Optimizations
- Code Emission

TARGET DESCRIPTION

- TargetMachine – provides access to:
- TargetData – endian-ness, alignments, pointer size
- TargetLowering – LLVM IR to Selection DAG information
- TargetRegisterInfo – number, class, aliases
- TargetInstructionInfo – mnemonic, operands, implicit registers
- TargetFrameInfo – stack direction, alignment, offset
- TargetSubtarget – supported instructions, their latencies, etc
- TargetJITInfo – [optional]

SELECTION DAG

- Graph representation of code
- Each node represents one instruction
- Each edges represents a dependency
- Used for instruction selection and scheduling

SELECTION DAG

EXAMPLE

```
%T1 = fadd float %W, %X
%T2 = fmul float %T1, %Y
%T3 = fadd float %T2, %Z
```

SELECTION DAG

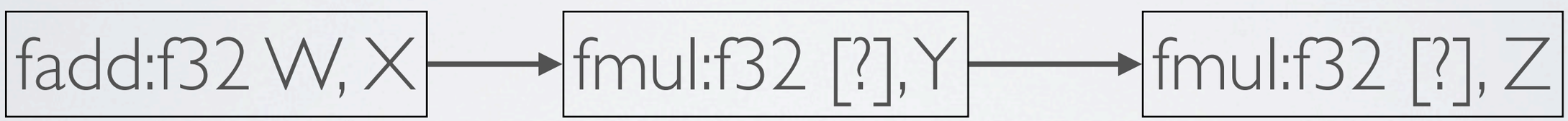
EXAMPLE

```
%T1 = fadd float %W, %X
%T2 = fmul float %T1, %Y
%T3 = fadd float %T2, %Z
```

SELECTION DAG

EXAMPLE

```
%T1 = fadd float %W, %X
%T2 = fmul float %T1, %Y
%T3 = fadd float %T2, %Z
```



(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)

SELECTION DAG

EXAMPLE

(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)

(FADDS (FMULS (FADDS W, X), Y), Z)

SELECTION DAG

EXAMPLE

(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)

(FADDS (FMULS (FADDS W, X), Y), Z)

(FMADDS (FADDS W, X), Y, Z)

EXAMPLE

- def FMADDS : AForm_1<59, 29,
  (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
    "fmadds $FRT, $FRA, $FRC, $FRB",
    [(set F4RC:$FRT,(fadd
      (fmul F4RC:$FRA, F4RC:$FRC), F4RC:$FRB))]>;

- def FADDS : AForm_2<59, 21,
    (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRB),
    "fadds $FRT, $FRA, $FRB",
    [(set F4RC:$FRT, (fadd F4RC:$FRA, F4RC:$FRB))]>;

INSTRUCTION SCHEDULING

- Several schedulers provided:
  - Standard list based scheduling (top-down)
  - Fast list based scheduling (bottom-up)
  - Register reducing list scheduler (top-down or bottom-up)
- Plus hybrids to balance ILP/register pressure/latency
- Each target specifies its own default

REGISTER ALLOCATION

- LLVM IR uses "infinite" virtual registers
- Each target specifies:
  - available registers
  - super/sub registers
  - allocation order (per class)
- LLVM provides three allocators
  - fast
  - linear scan
  - PBQP
    Partitioned Boolean Quadratic Programming

LLC PLUGINS

- Each LLC pass is a distinct library
- Passes can be dynamically loaded
  (e.g. register allocator, instruction scheduler, GC Support)
- Target information must be defined statically

LLVM

SUPPORTED TARGETS

- X86
- PowerPC
- ARM
- Sparc
- Alpha
- MSP430
- Mips
- SystemZ
- XCore
- Blackfin
- CellSPU
- MBlaze
- PTX

http://llvm.org/docs/CodeGenerator.html#targetfeatures

OTHER TOOLS

- llvm: wrapper program (much like gcc)
- llvm-link: 'links' together several bitcode files
- llvm-ar: creates and 'archive' of bitcode files (like ar)
- llvm-nm: lists symbol names (like nm)
- llvm-as: converts LLVM assembly to LLVM bitcode
- llvm-prof: profiles an LLVM execution