Binary Program Rewriting with Diablo

Bjorn De Sutter
Ghent University

PLDI06 , June 06, 2006

Credits

- Bruno De Bus
- Dominique Chanet
- Ludo Van Put
- Matias Madou
- Bertrand Anckaert
- Koen De Bosschere

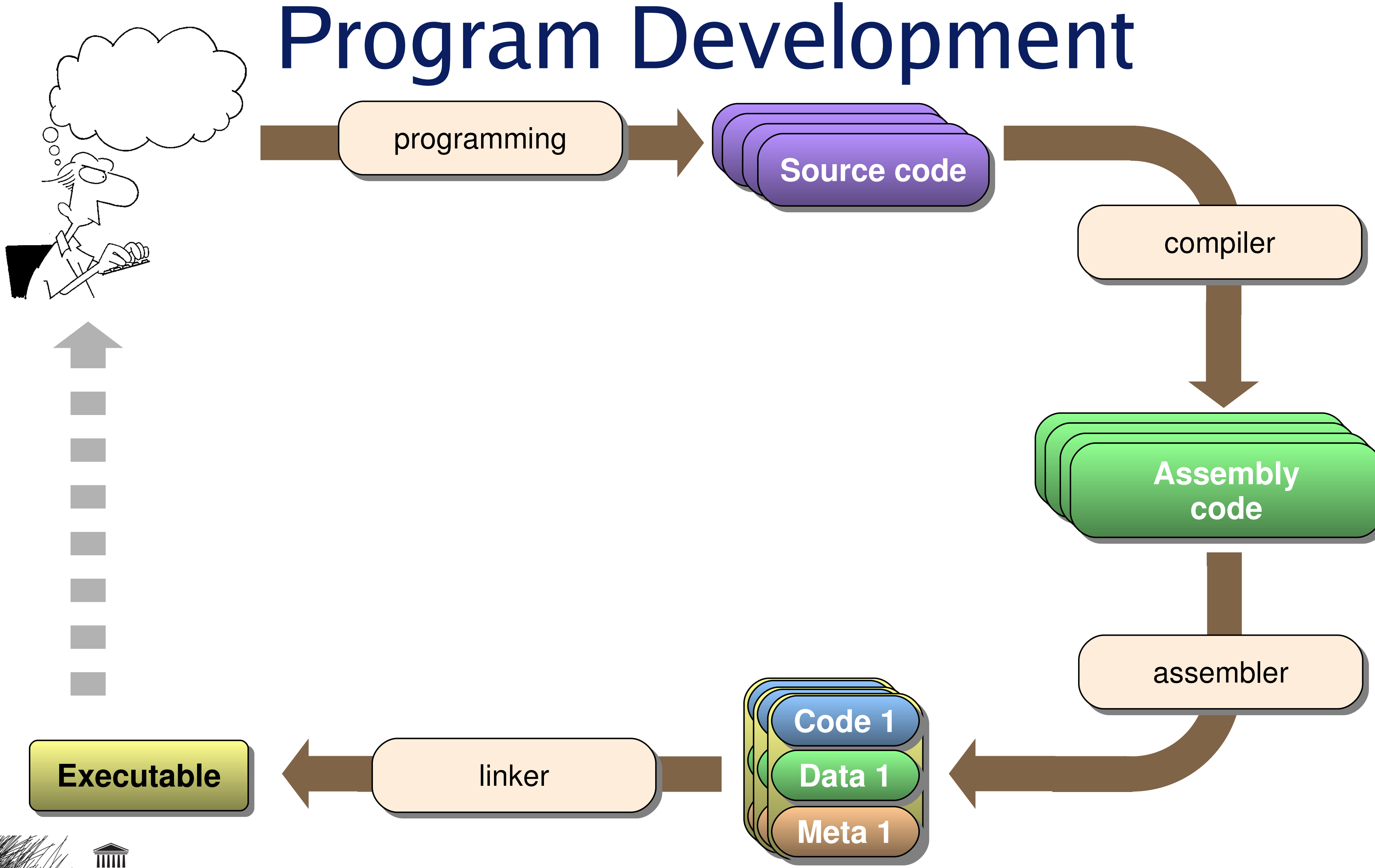
Overview

- Some background
- Diablo
  - Extensibility
  - Retargetability
  - Reliability

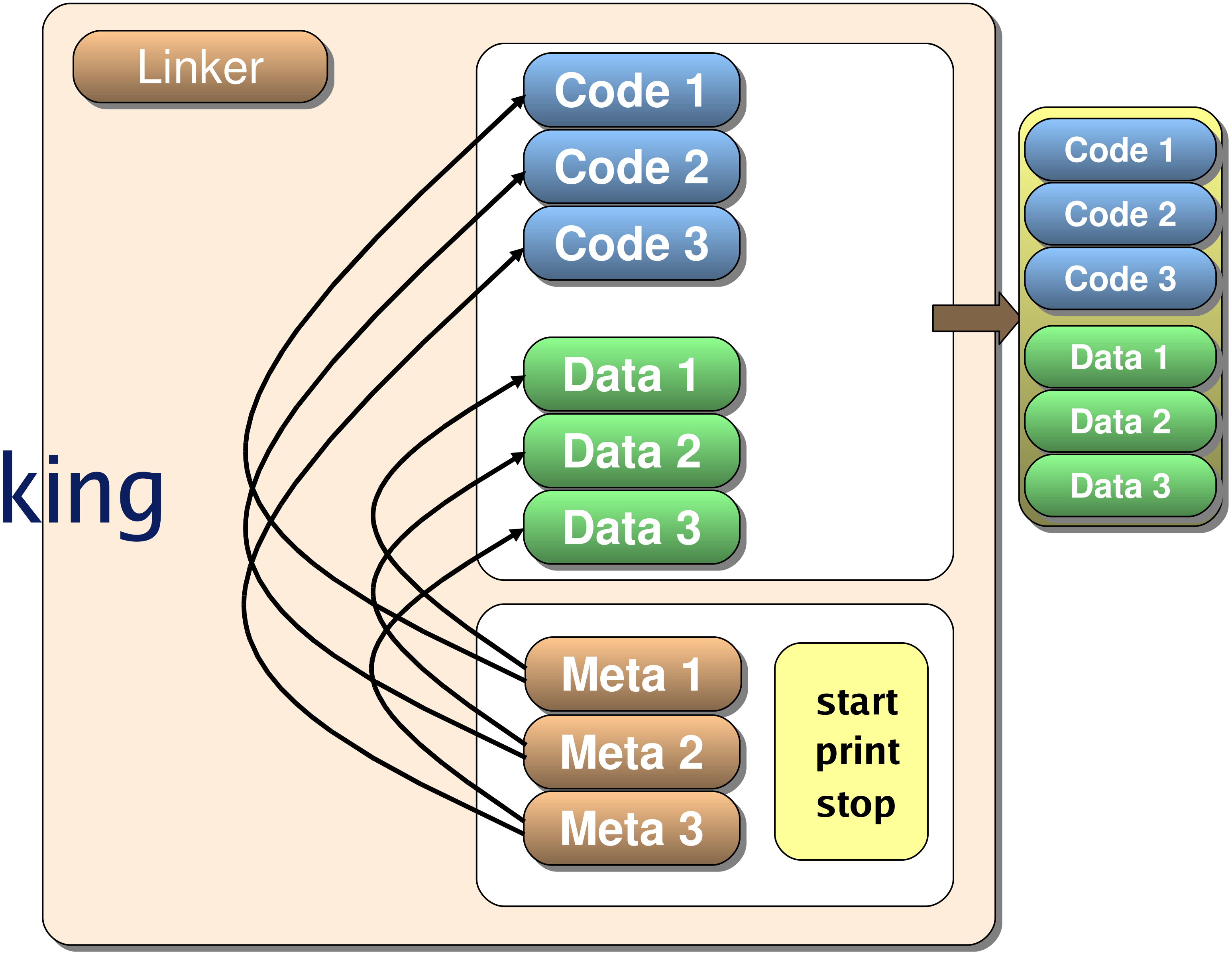
Overview

- INTRODUCTION (45 min)
- DATASTRUCTURES (1 hr)
- ANALYSES AND TRANSFORMATIONS (45 min)
- BACKENDS (30 min)

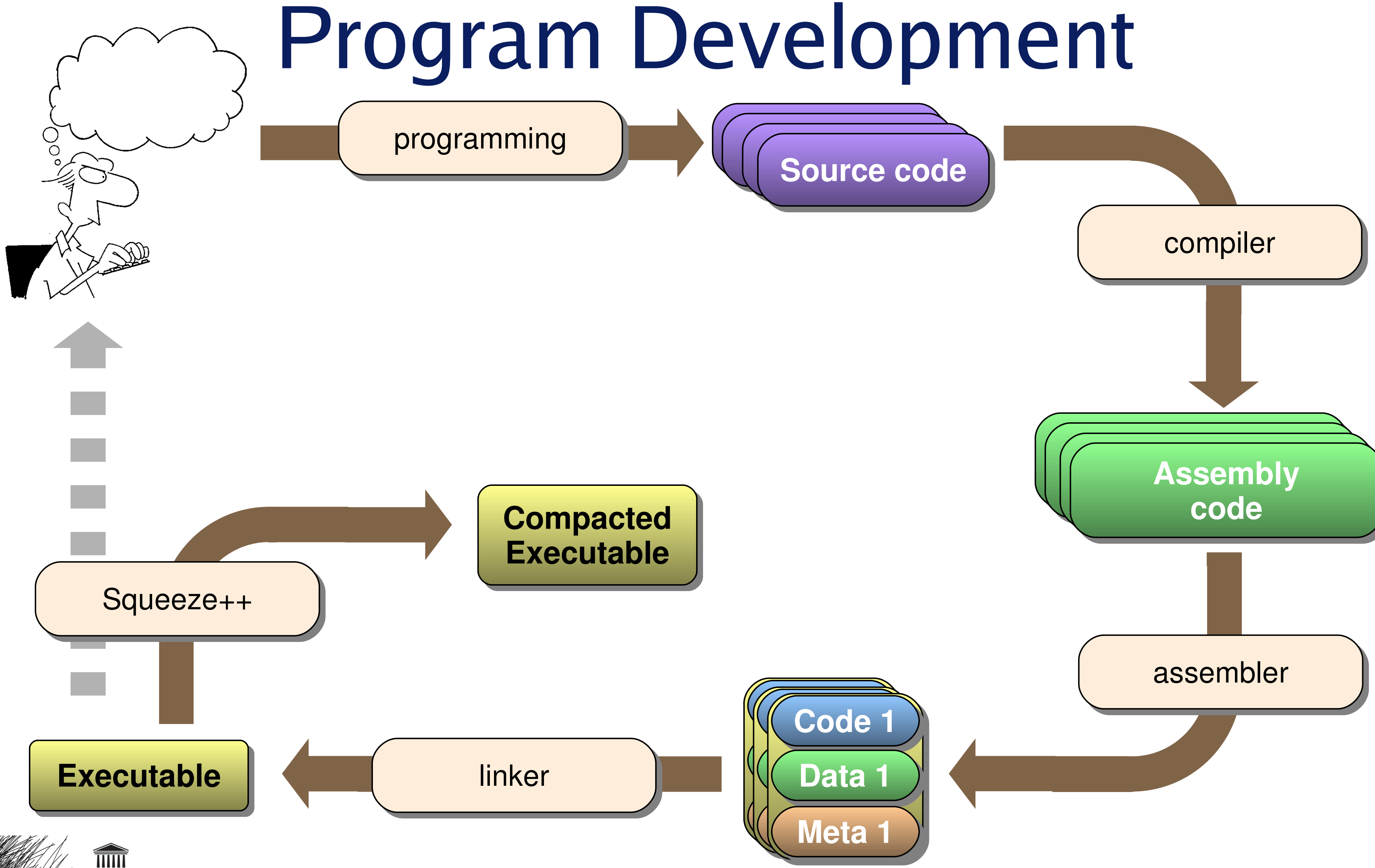
Program Development



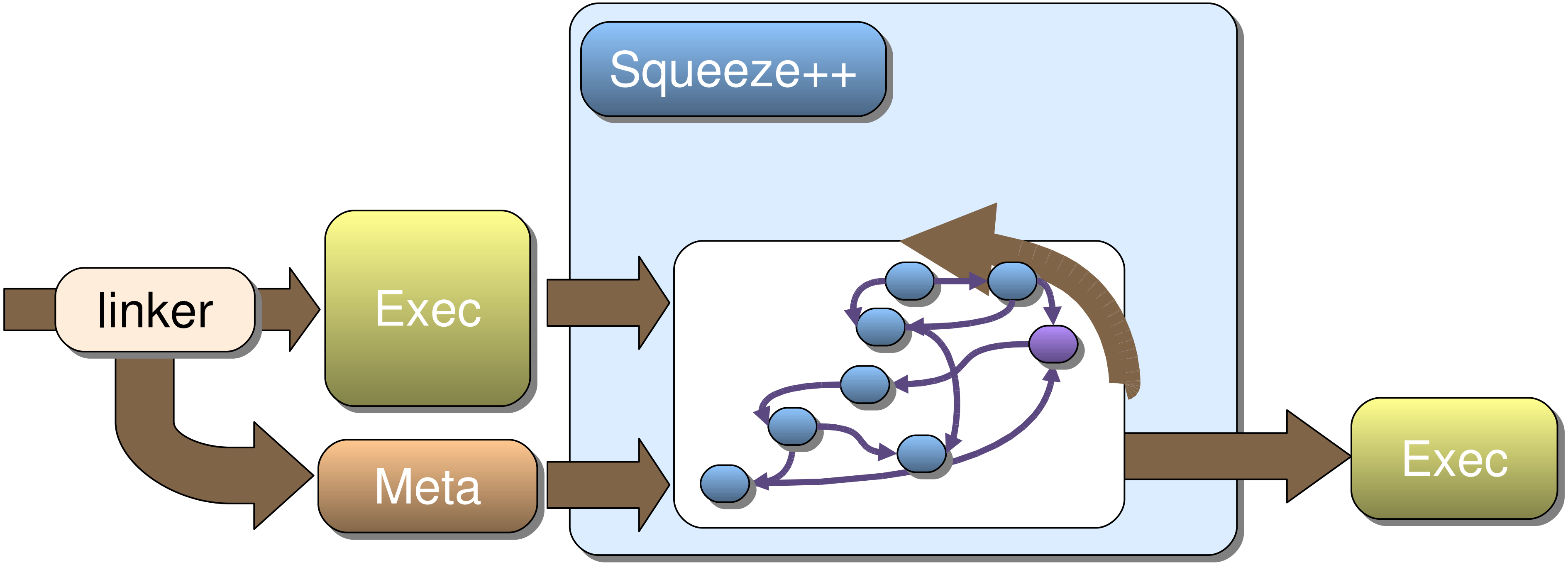
Linking


Linker
Code 1
Code 2
Code 3
Data 1
Data 2
Data 3
Meta 1
Meta 2
Meta 3
start
print
stop
Code 1
Code 2
Code 3
Data 1
Data 2
Data 3

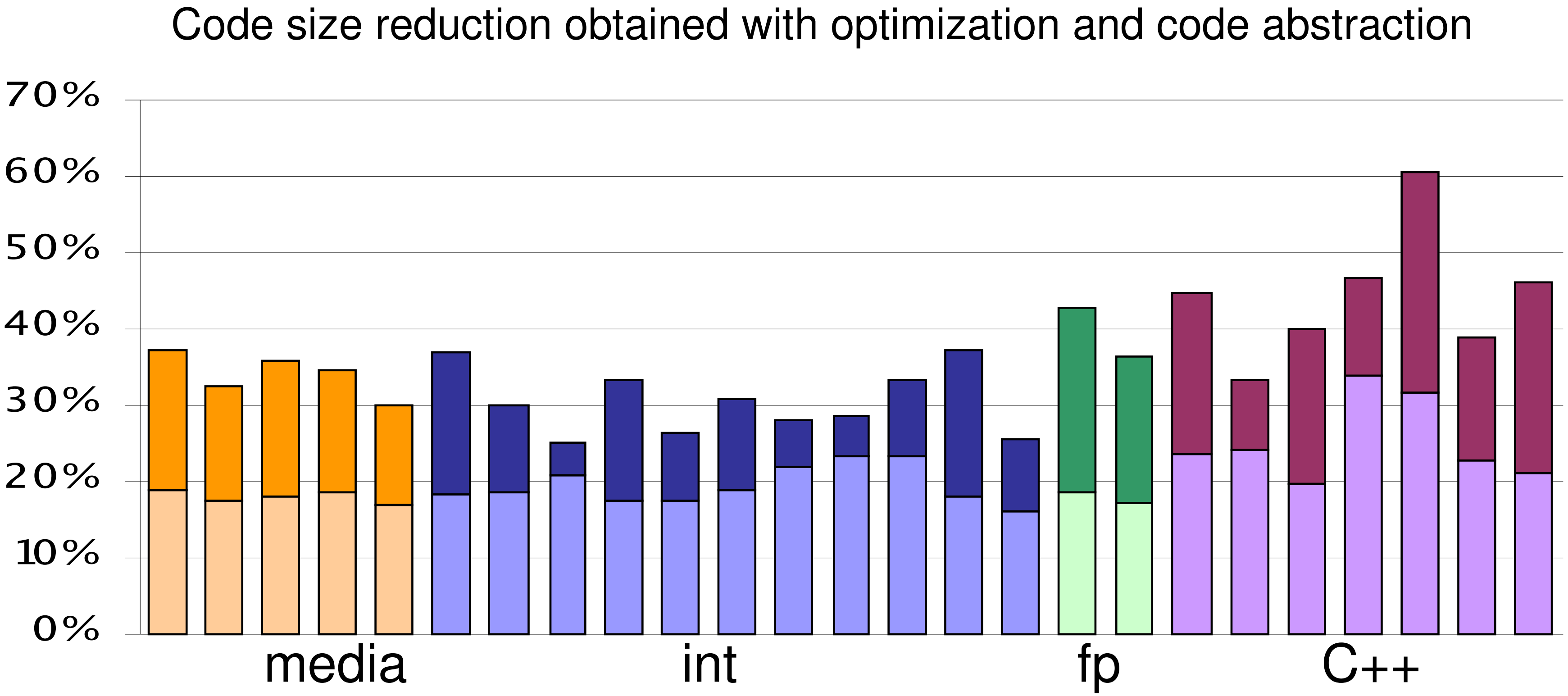
Program Development



Additional optimization opportunities?

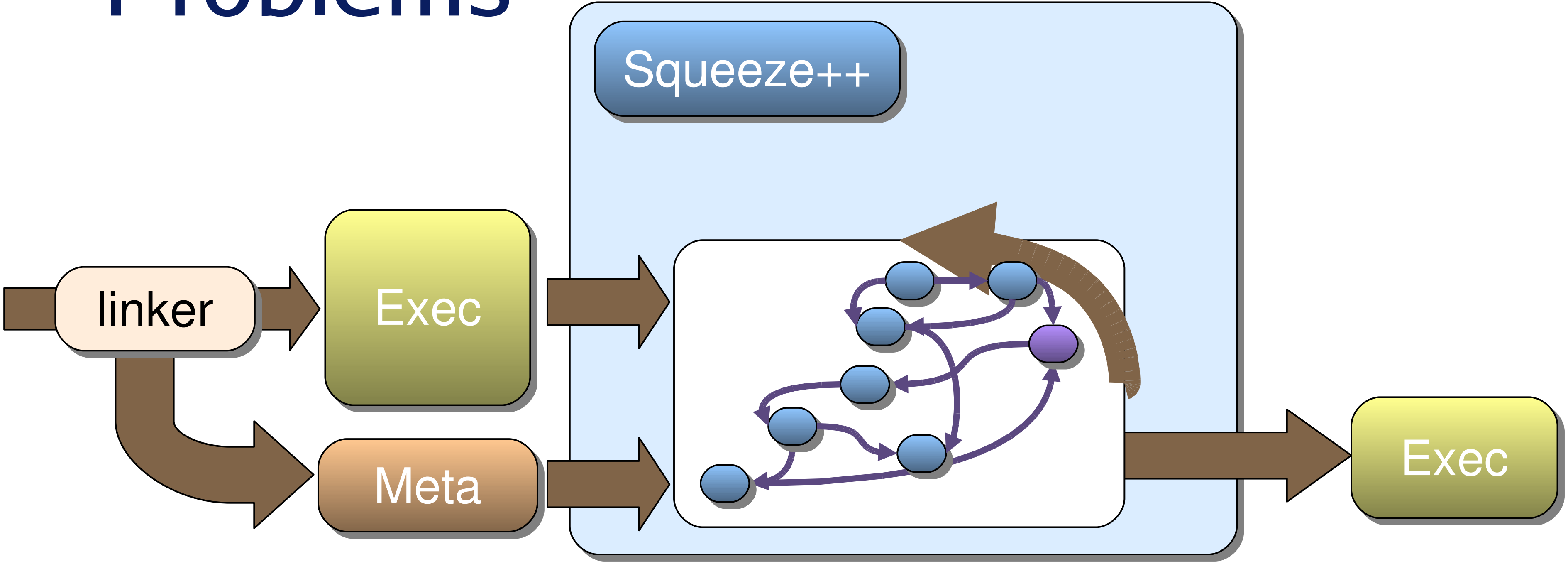


Results (Squeeze++)



[De Sutter, De Bus and De Bosschere, ACM TOPLAS, Sept 2005]

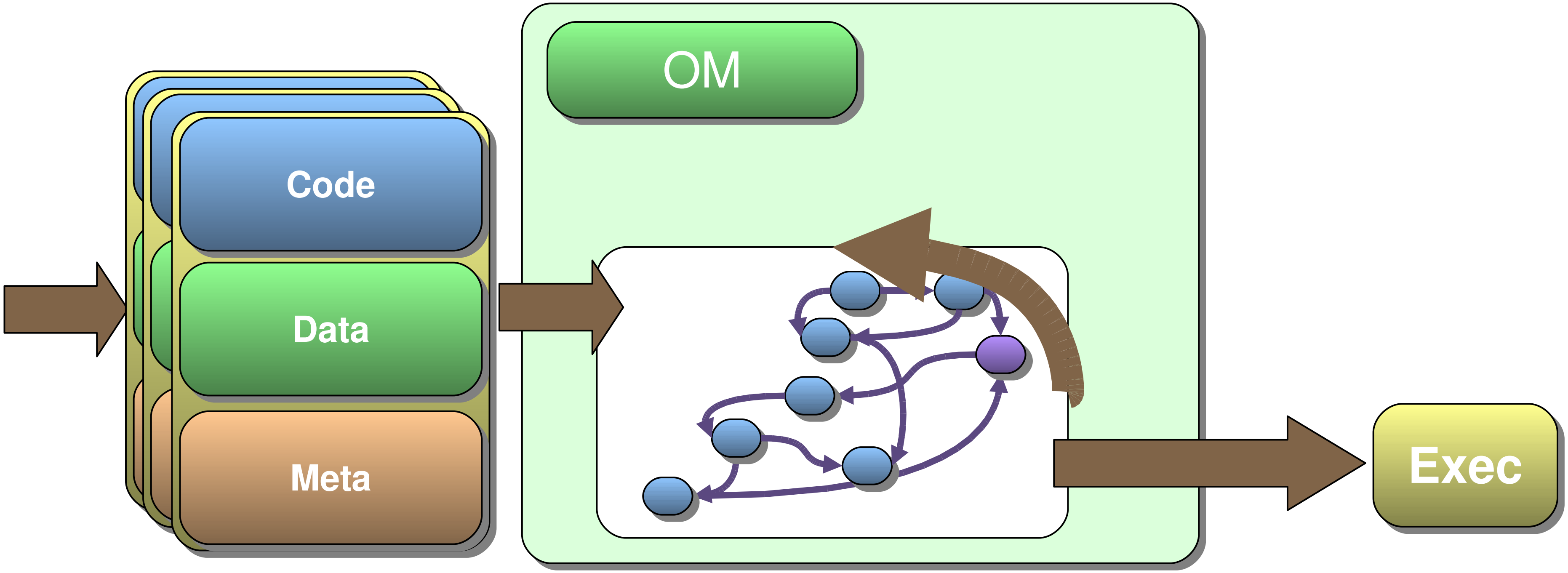
Problems



Only for the Alpha architecture.
Only for compaction/optimization
Small change implies days of debugging

Not retargetable
Not extensible
Not reliable

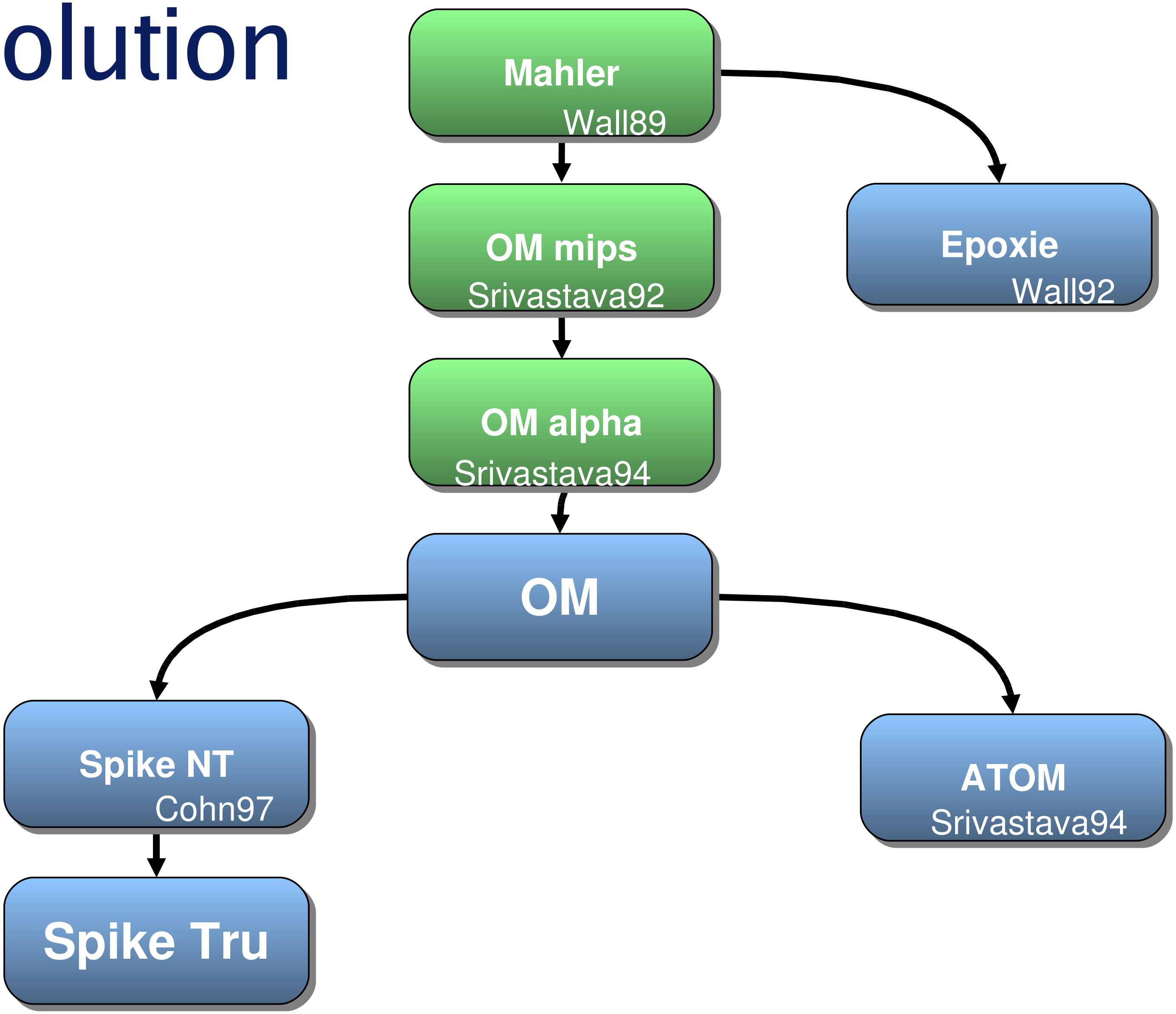
Other rewriters?



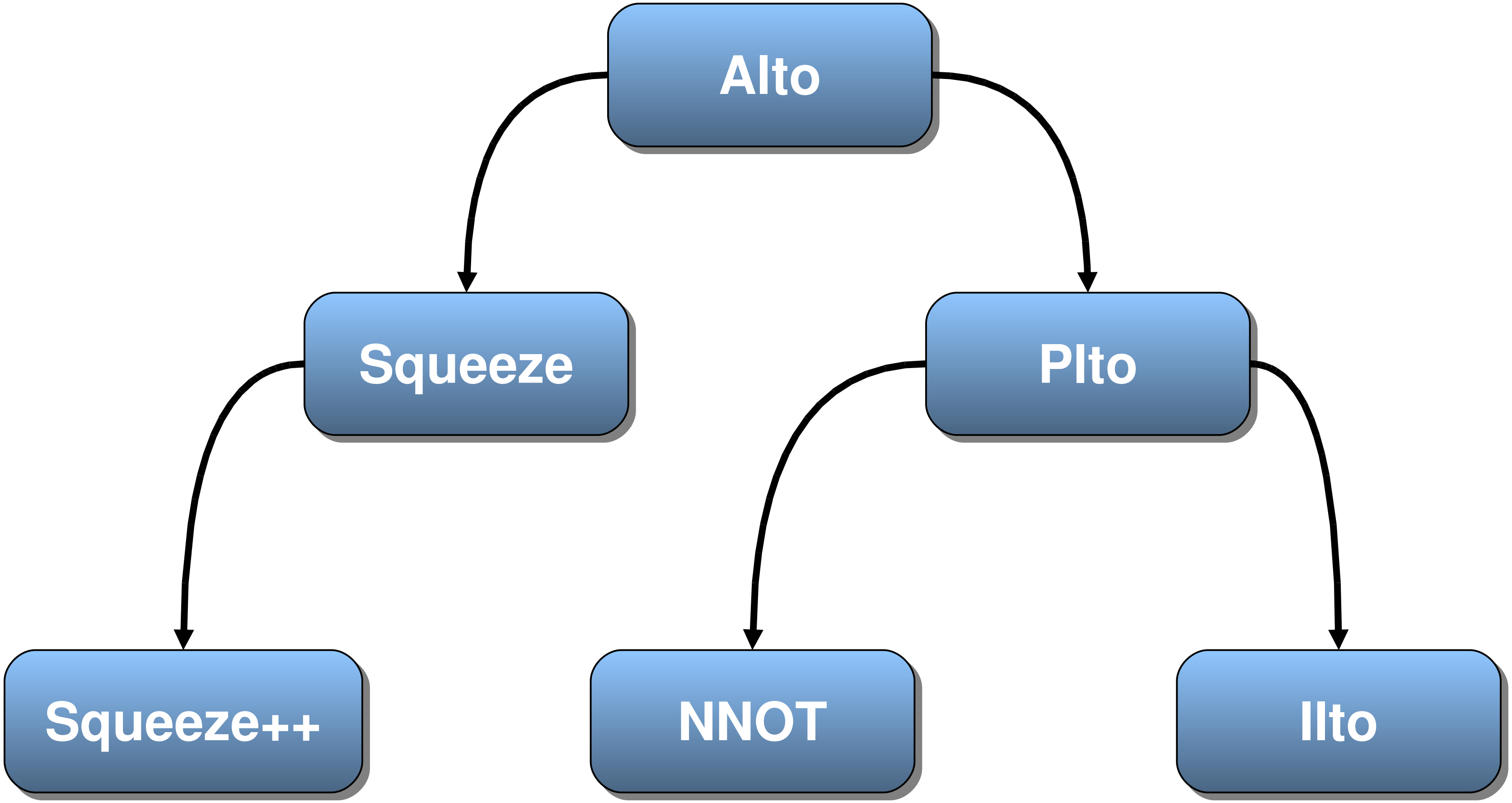
Same problems:

- not retargetable
- not extensible
- not reliable

OM's evolution



Alto's Evolution



static binary rewriting is useful…

Applications

- Optimization, compaction
- Instrumentation
- Obfuscation

- Program understanding, visualisation
- Debugging
- ...

but it is a bit problematic…

Problems

- Not retargetable
- Not extensible
- Not reliable

Overview

- Some background
- Diablo
  - Extensibility
  - Retargetability
  - Reliability

Pre-link or post-link?

Pre-link

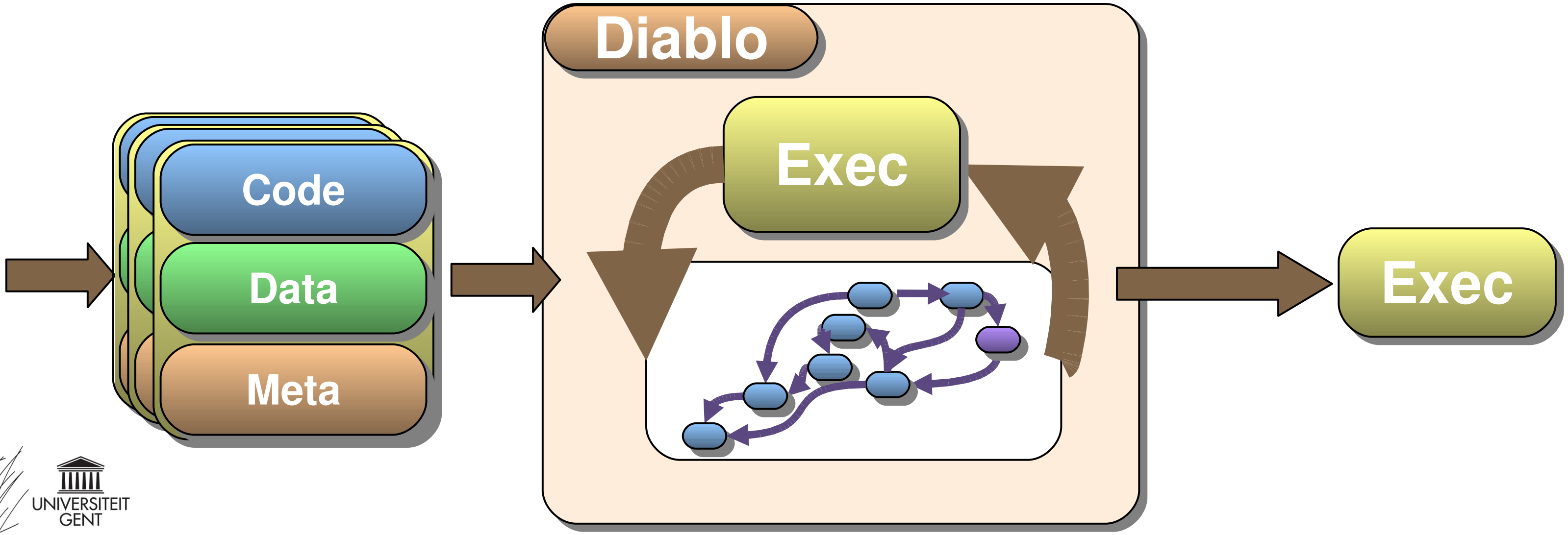
- + **More meta information**
  = more aggressive transformations
- - **No program overview**

Link - time

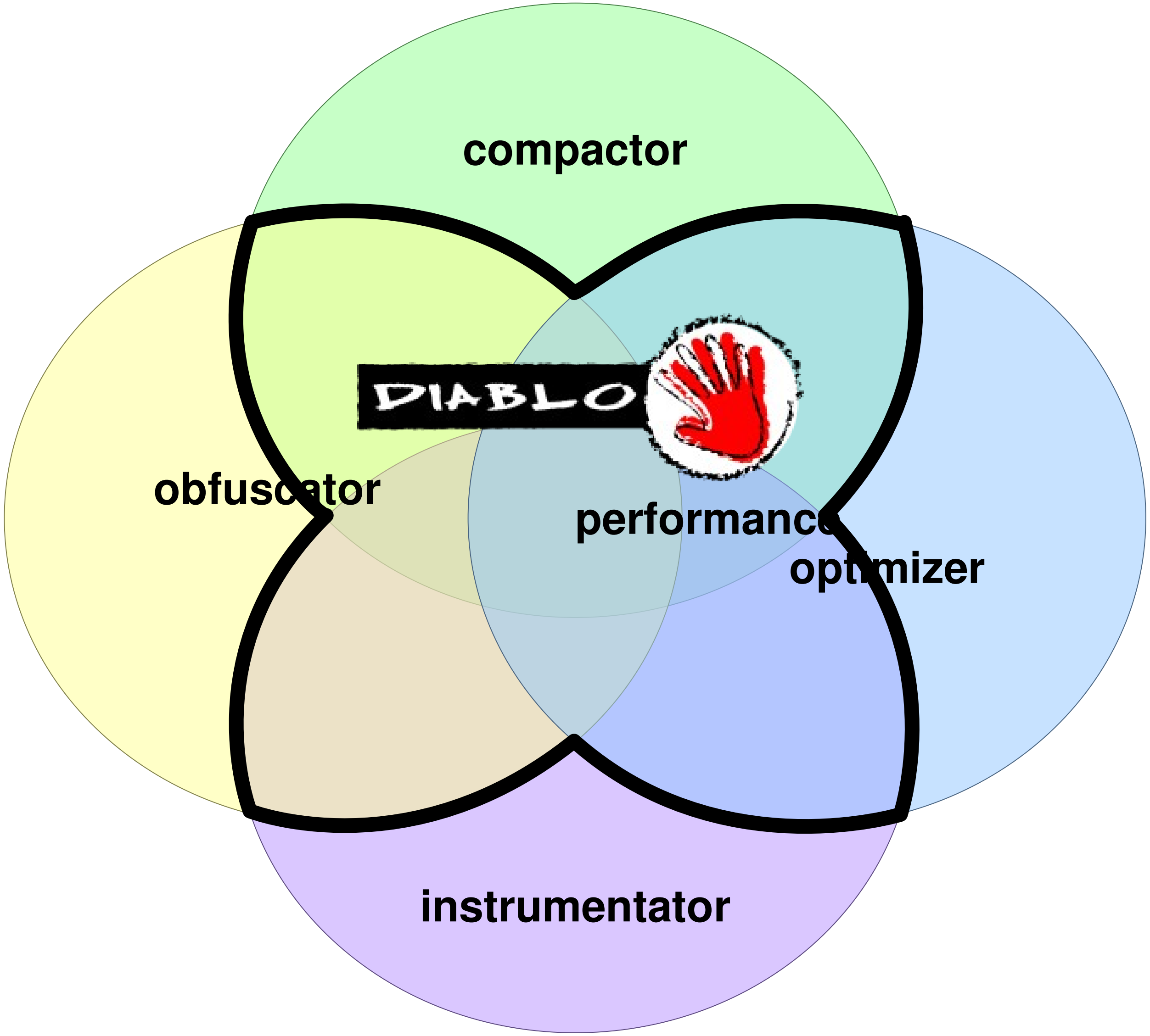
- + **More meta information**
- + **Whole-program overview**
- - **More implementation work**

Post-link

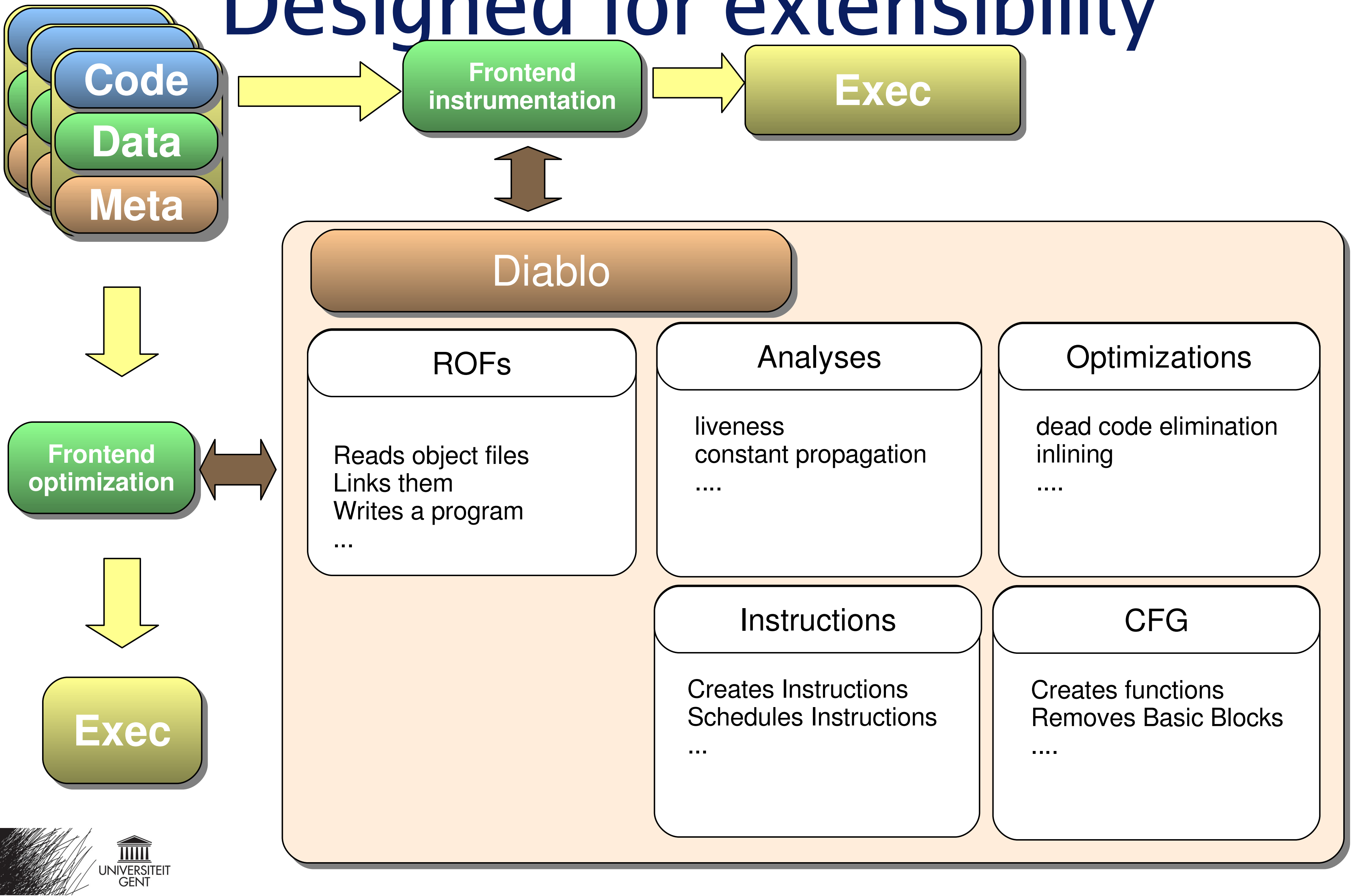
- - **Less meta information**
  = more conservative transformations
- + **Whole-program overview**



Extensibility – The problem



Designed for extensibility



ROFs

Reads object files
Links them
Writes a program
...

Analyses

liveness
constant propagation
....

Optimizations

dead code elimination
inlining
....

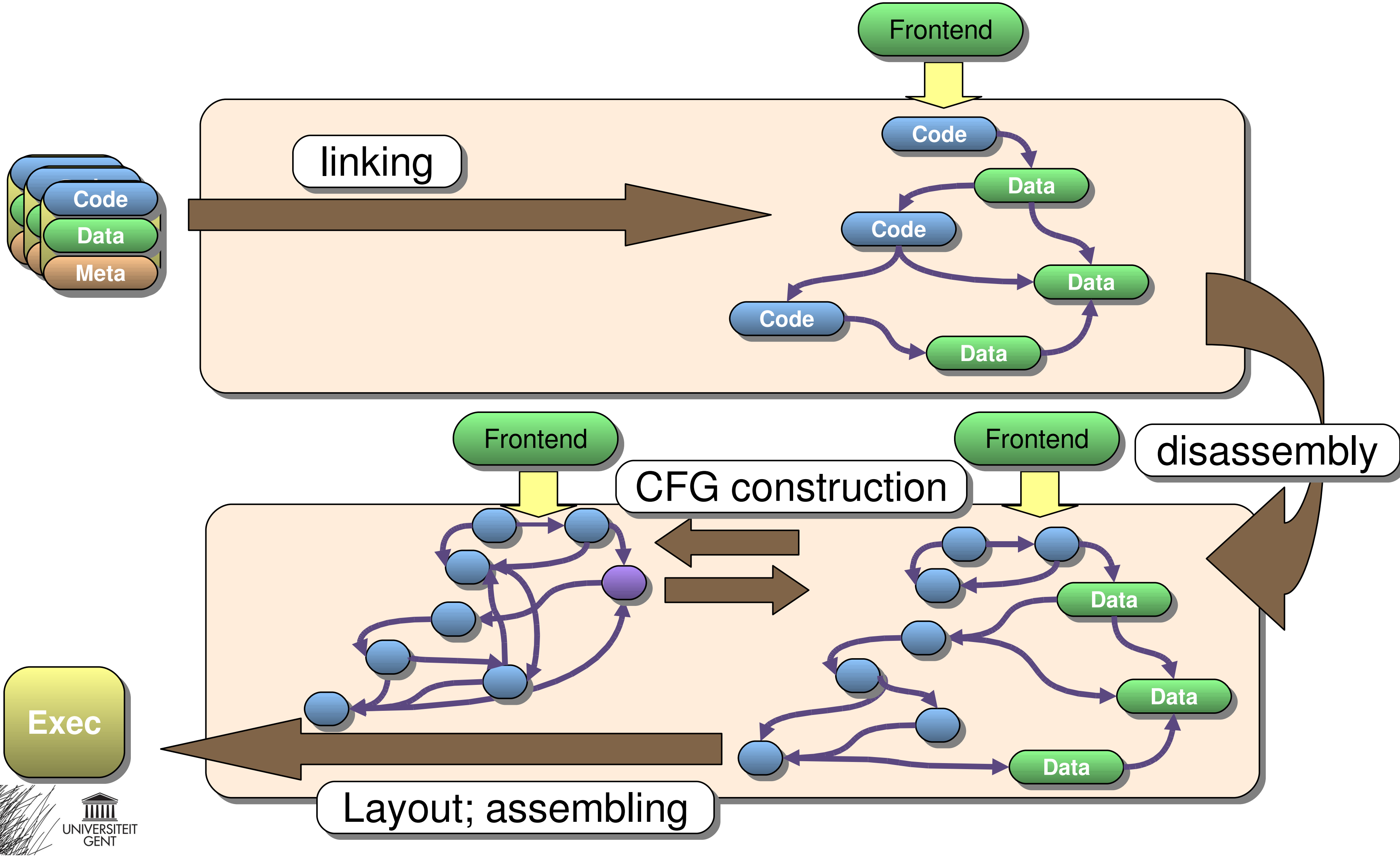
Instructions

Creates Instructions
Schedules Instructions
...

CFG

Creates functions
Removes Basic Blocks
....

Operation at different levels



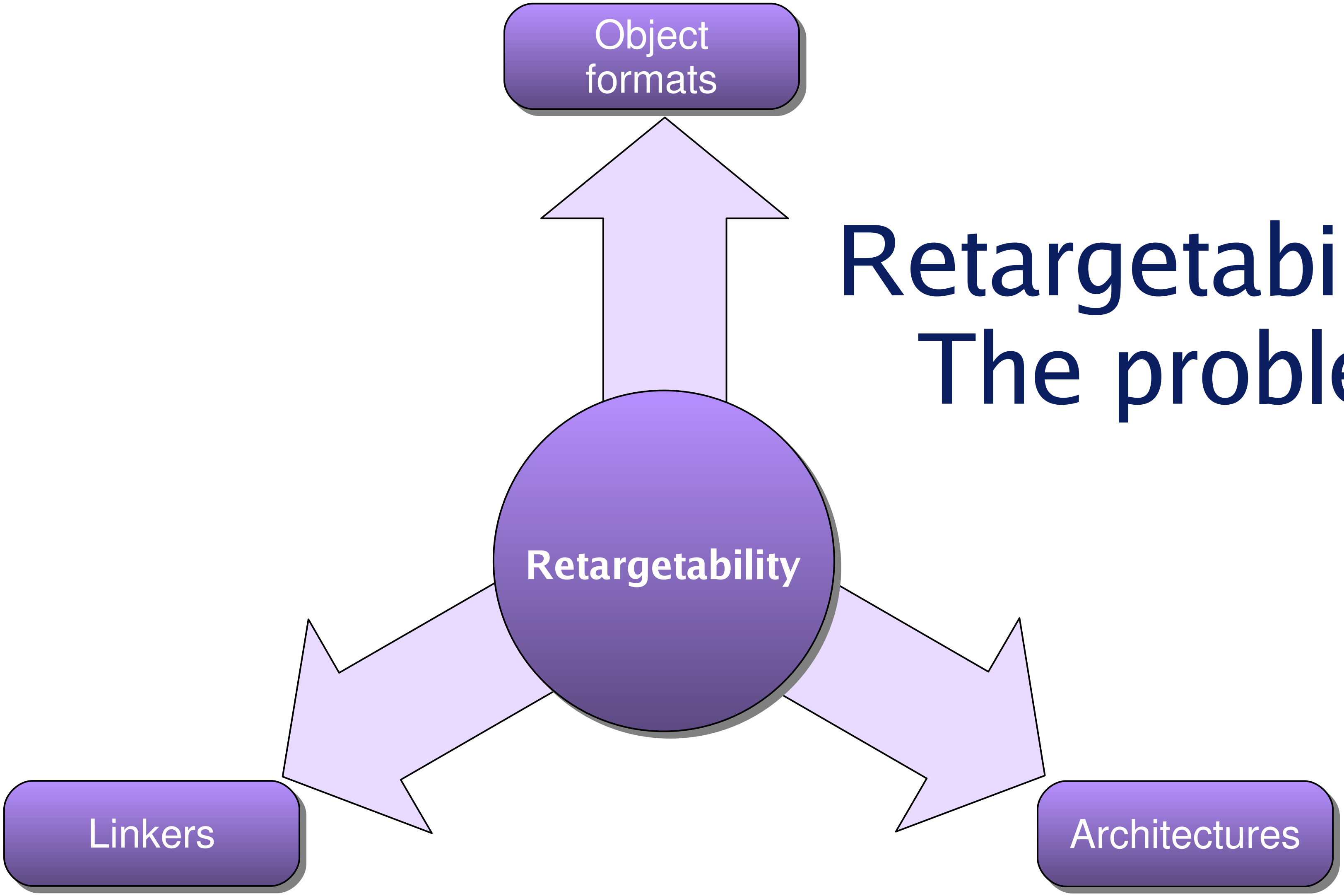
Extensibility

Diablo	Application optimization and compaction frontend	LCTES'04
FIT	Instrumentation frontend	PASTE'04
Stilo	Steganography frontend	ICISC'04
kDiablo	Linux kernel specialization frontend	LCTES'05
Lancet	Interactive binary program editor	PASTE'05
Loco	Interactive program obfuscator	PEPM'05

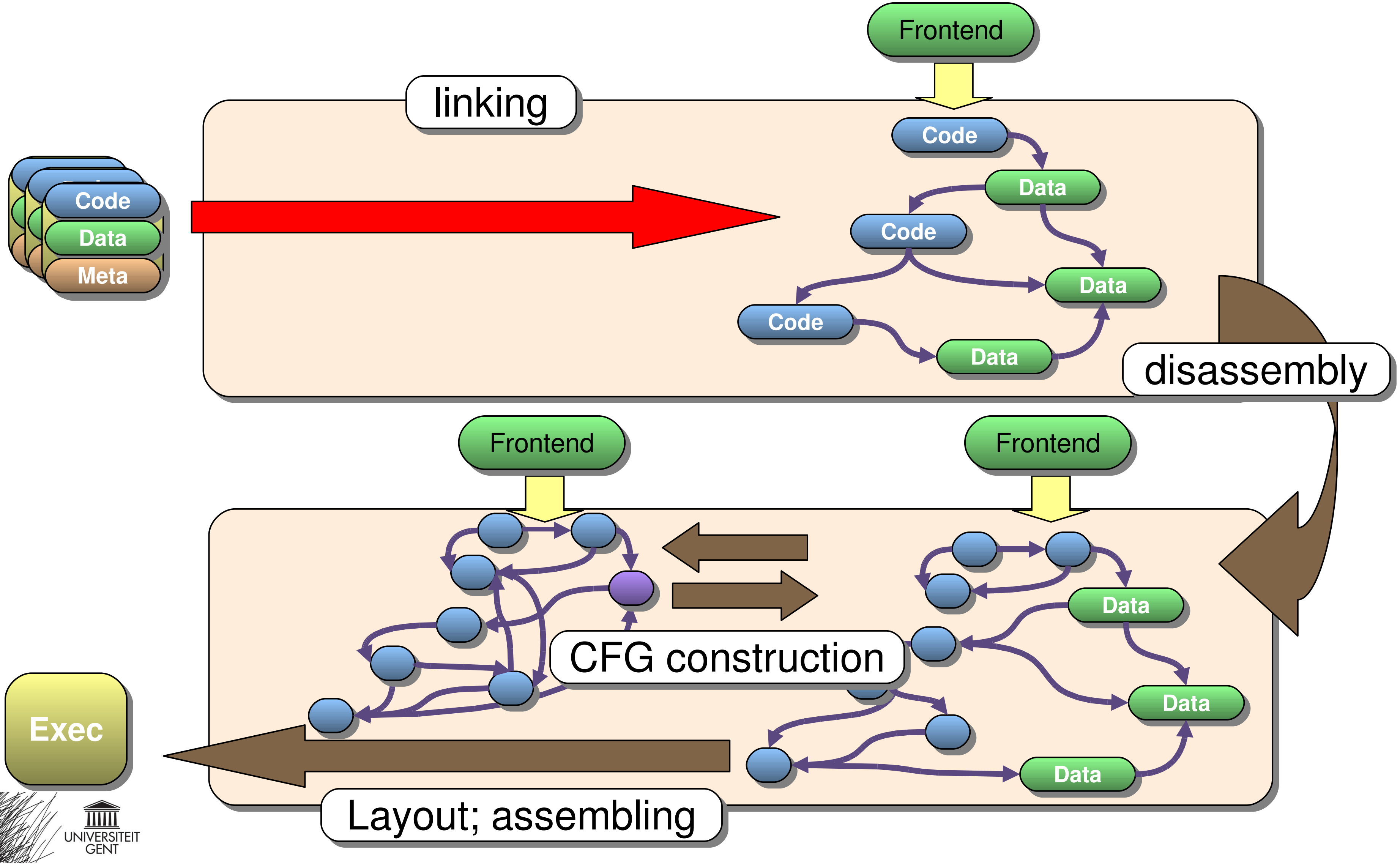
Overview

- Some background
- Diablo
  - Extensibility
  - Retargetability
  - Reliability

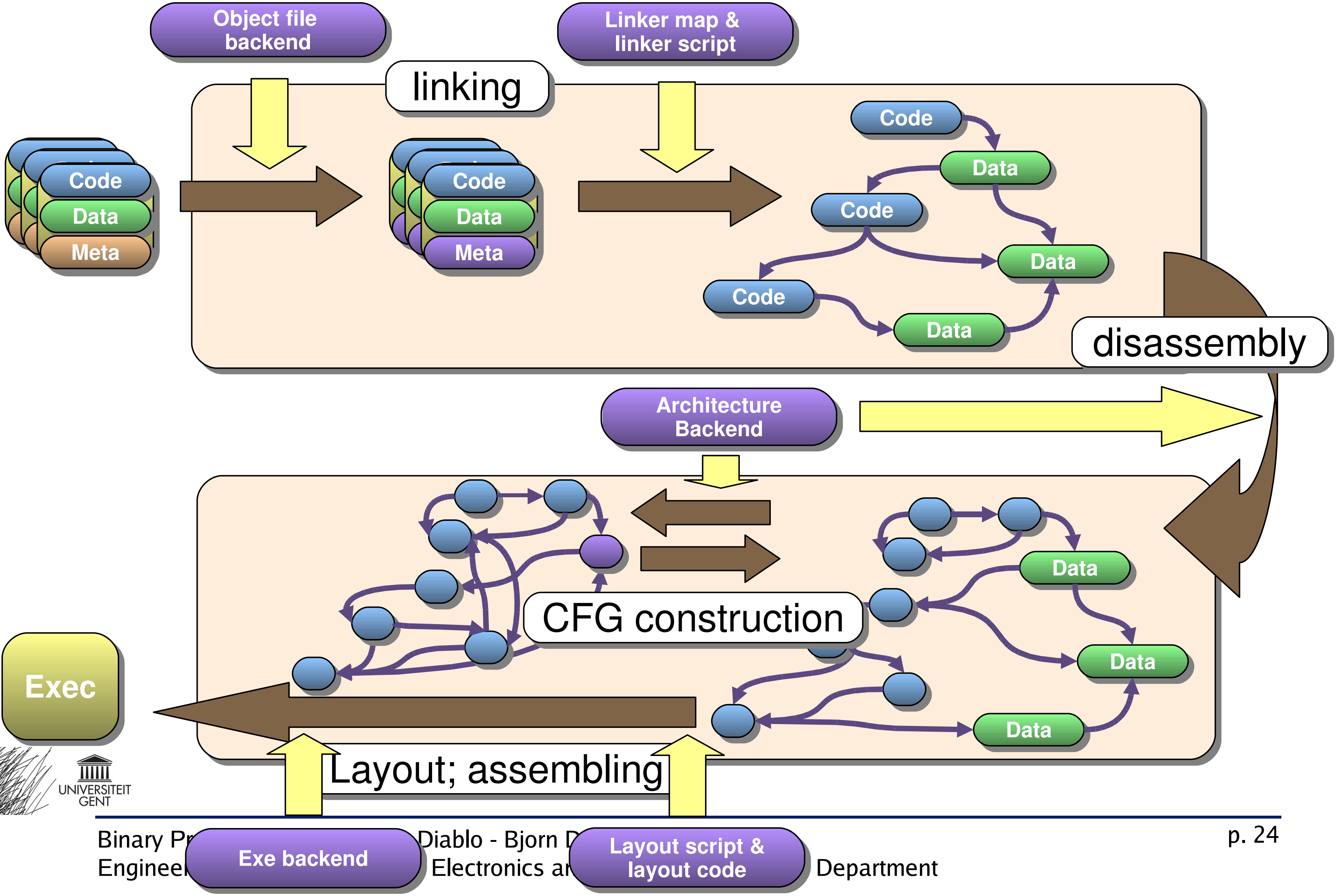
Retargetability - The problem



Operation at different levels



Retargeting multiple ROF & Linkers



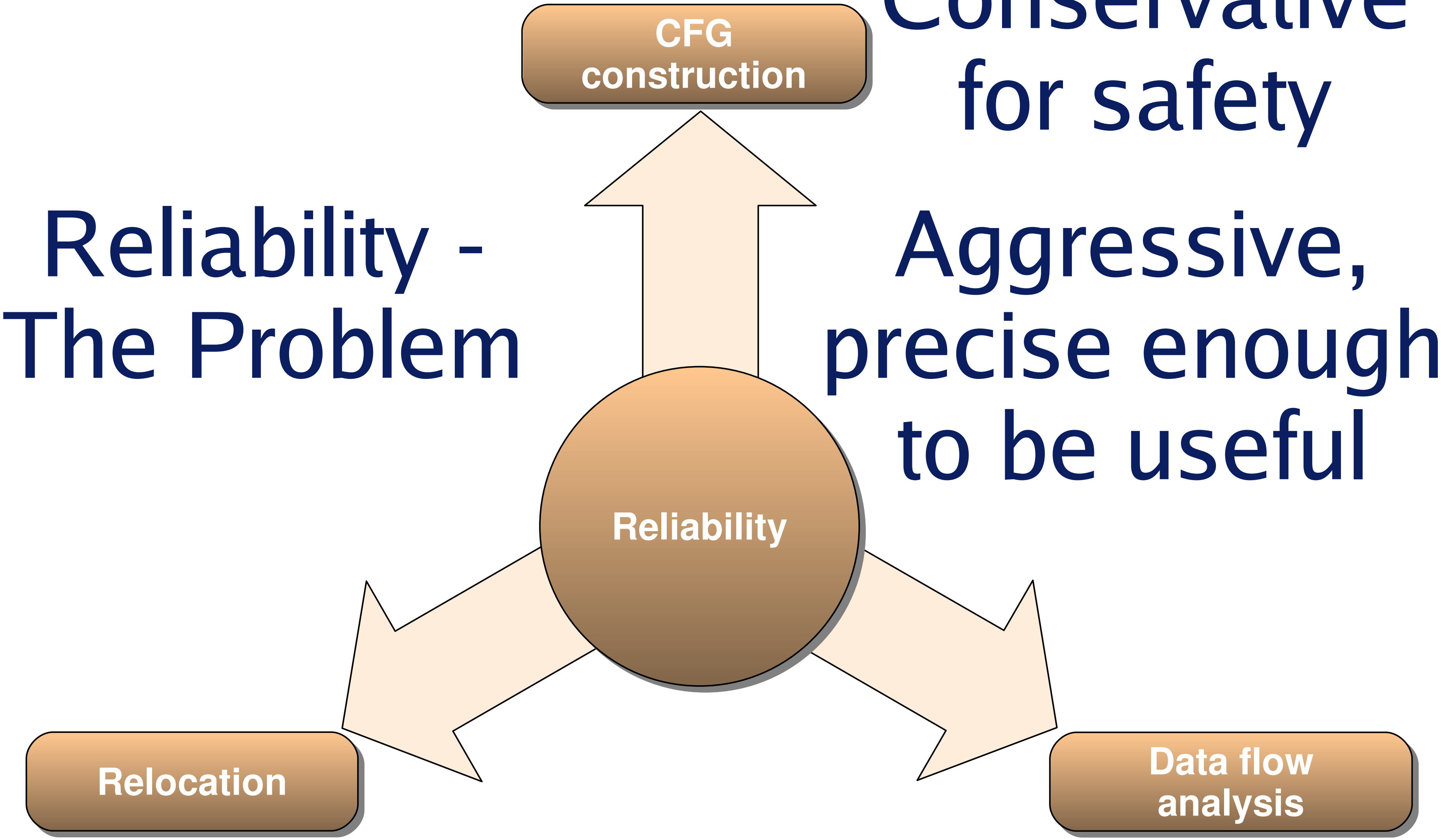
Major implemented backends

ARM	Diablo, FIT, kDiablo, Lancet	LCTES'04
x86	Diablo, FIT, Stilo, kDiablo, Lancet, Loco	LCTES'05
IA64	Diablo	Europar'04
MIPS32	Diablo	ESA'04
Alpha	Diablo, FIT	

Overview

- Some background
- Diablo
  - Extensibility
  - Retargetability
  - Reliability

Reliability - The Problem



Conservative for safety

Aggressive, precise enough to be useful

CFG Construction

Disassembling

Potential problems:

- Differentiate data from code
- Detect self-modifying code
- Detect unrewritable code

Solutions:

- Section information
- Symbols annotate data in code (ARM ABI)
- Self-modifying code in data:
  no problem at this point
- True self-modifying code:
  look at system calls and protection

Conservatively modelling control flow

Potential problems:

- Indirect control flow transfers
- Code that is treated as data
- Unrealizable paths (procedures)

Solutions:

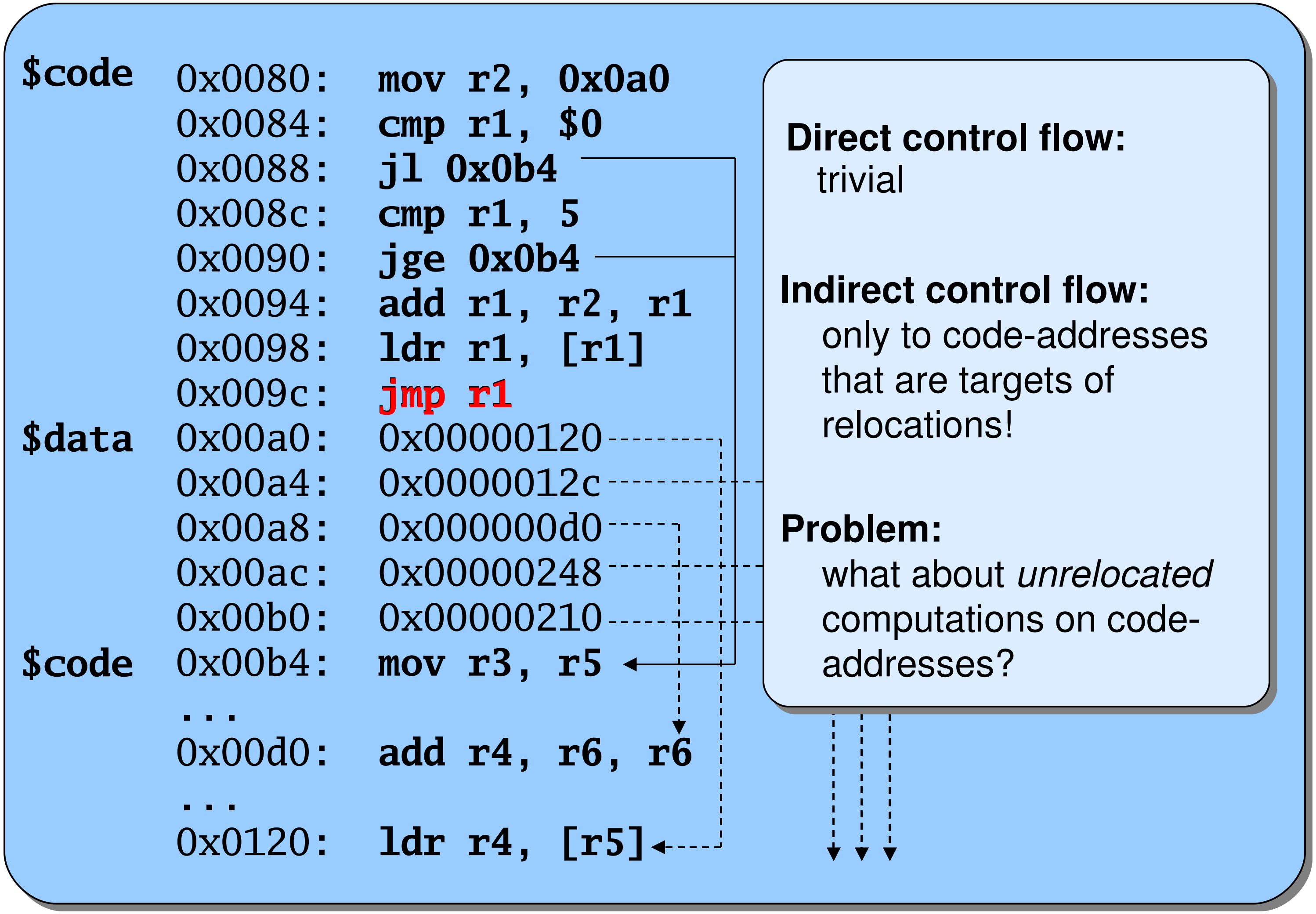
- Use relocation information:
  identifies computable addresses
- Use pattern matching:
  identifies known address computations
- Use knowledge on compiler-generated code

Detecting Data

```
$code  0x0080:  mov r2, 0x0a0
       0x0084:  cmp r1, $0
       0x0088:  jl 0x0b4
       0x008c:  cmp r1, 5
       0x0090:  jge 0x0b4
       0x0094:  add r1, r2, r1
       0x0098:  ldr r1, [r1]
       0x009c:  jmp r1
$data  0x00a0:  0x00000120
       0x00a4:  0x0000012c
       0x00a8:  0x000000d0
       0x00ac:  0x00000248
       0x00b0:  0x00000210
$code  0x00b4:  mov r3, r5
```

Solution:
add mapping symbols

Detecting Control Flow Targets



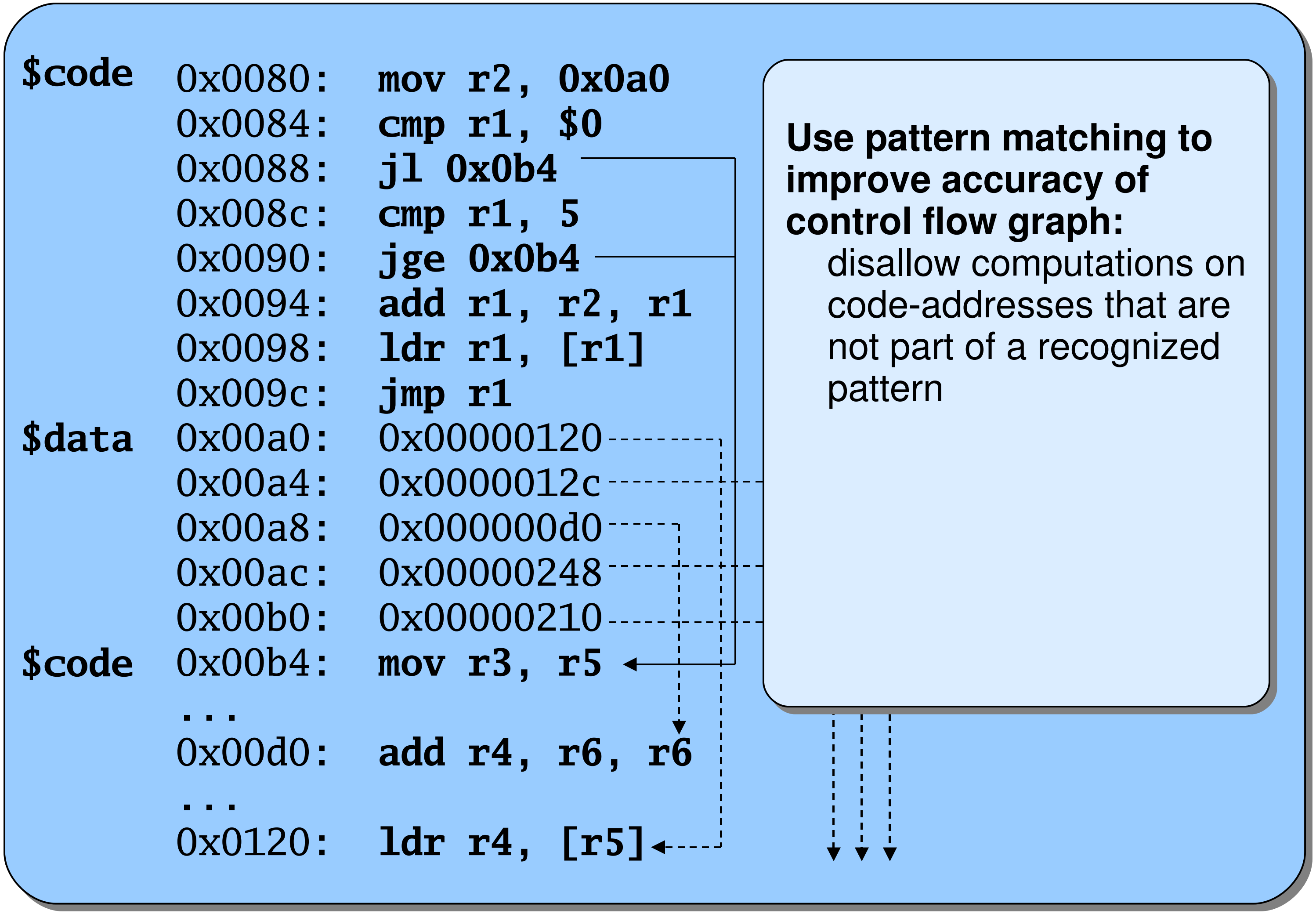
```
$code  0x0080:  mov r2, 0x0a0
       0x0084:  cmp r1, $0
       0x0088:  jl 0x0b4
       0x008c:  cmp r1, 5
       0x0090:  jge 0x0b4
       0x0094:  add r1, r2, r1
       0x0098:  ldr r1, [r1]
       0x009c:  jmp r1
$data  0x00a0:  0x00000120
       0x00a4:  0x0000012c
       0x00a8:  0x000000d0
       0x00ac:  0x00000248
       0x00b0:  0x00000210
$code  0x00b4:  mov r3, r5
       ...
       0x00d0:  add r4, r6, r6
       ...
       0x0120:  ldr r4, [r5]
```

Direct control flow:
trivial

Indirect control flow:
only to code-addresses that are targets of relocations!

Problem:
what about *unrelocated* computations on code-addresses?

Control Flow: Pattern Matching

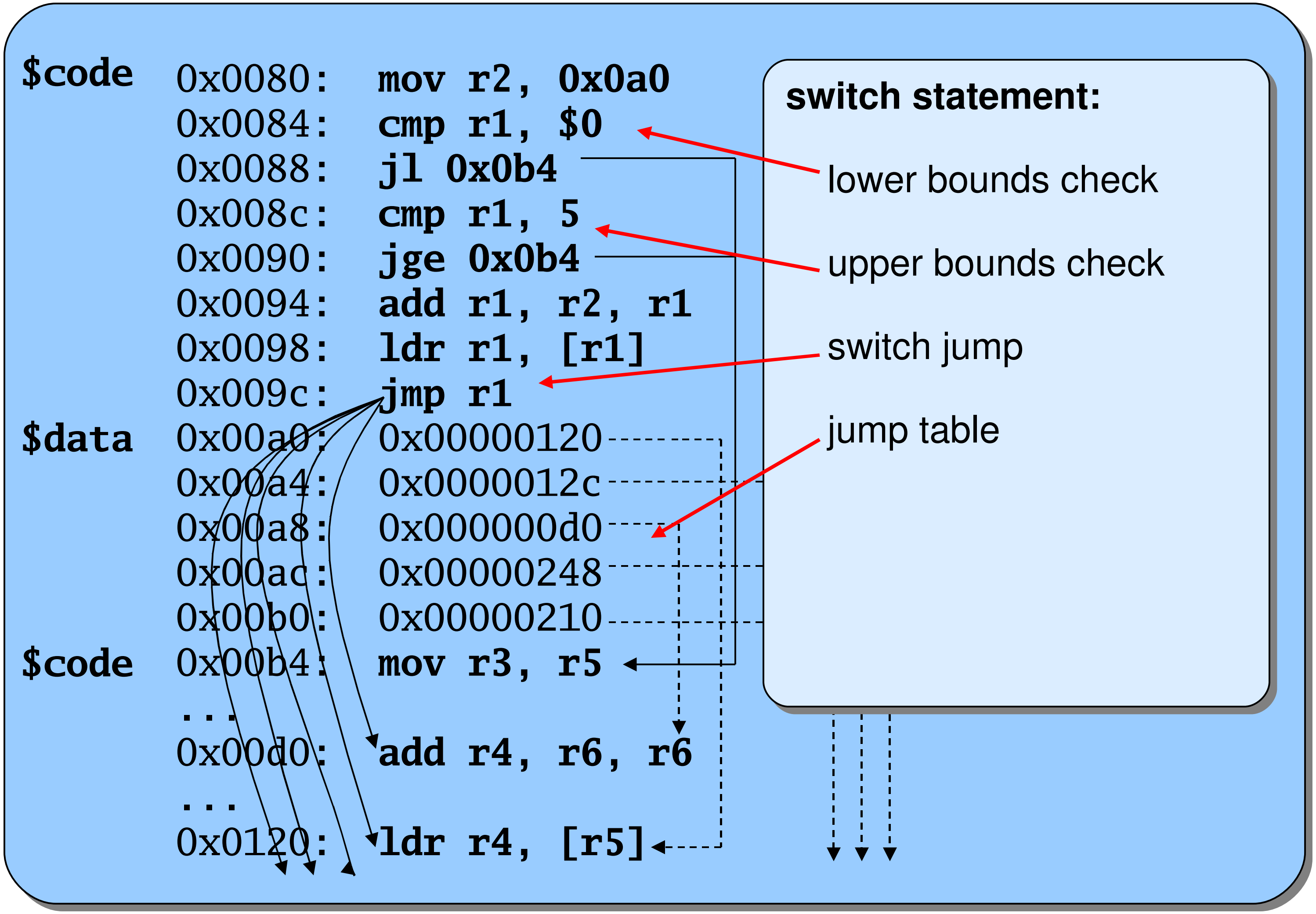


```
$code   0x0080:  mov r2, 0x0a0
        0x0084:  cmp r1, $0
        0x0088:  jl 0x0b4
        0x008c:  cmp r1, 5
        0x0090:  jge 0x0b4
        0x0094:  add r1, r2, r1
        0x0098:  ldr r1, [r1]
        0x009c:  jmp r1
$data   0x00a0:  0x00000120
        0x00a4:  0x0000012c
        0x00a8:  0x000000d0
        0x00ac:  0x00000248
        0x00b0:  0x00000210
$code   0x00b4:  mov r3, r5
        ...
        0x00d0:  add r4, r6, r6
        ...
        0x0120:  ldr r4, [r5]
```

Use pattern matching to improve accuracy of control flow graph:

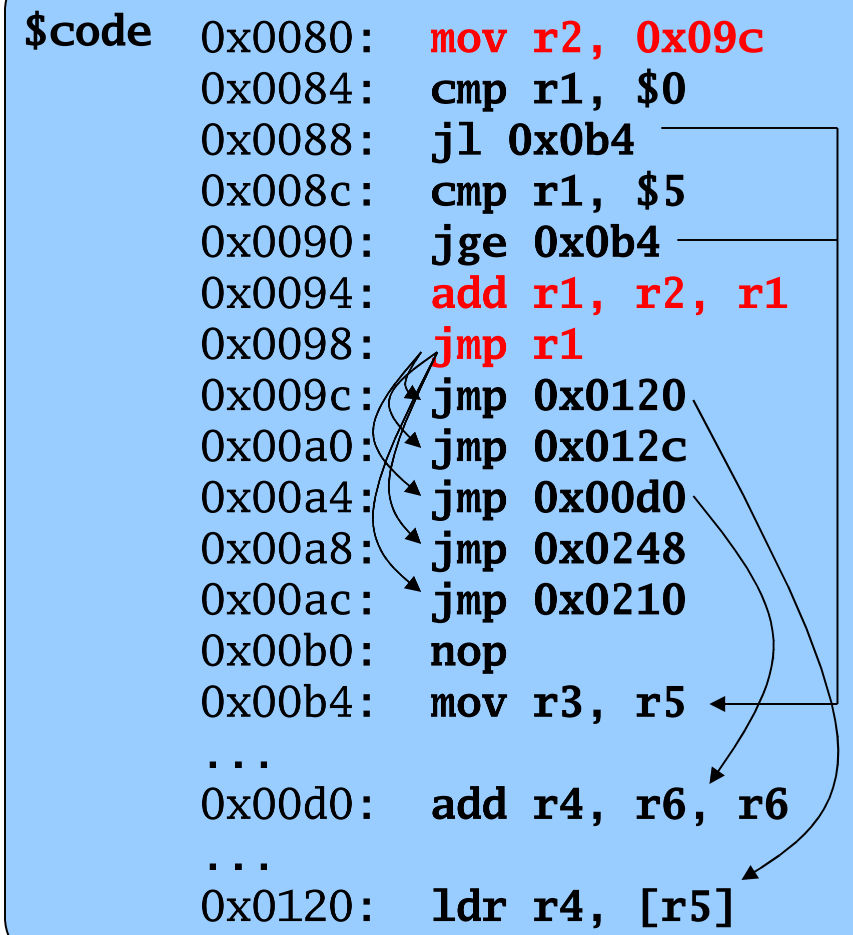
disallow computations on code-addresses that are not part of a recognized pattern

Pattern Matching: example



```
$code   0x0080:   mov r2, 0x0a0
        0x0084:   cmp r1, $0
        0x0088:   jl 0x0b4
        0x008c:   cmp r1, 5
        0x0090:   jge 0x0b4
        0x0094:   add r1, r2, r1
        0x0098:   ldr r1, [r1]
        0x009c:   jmp r1
$data   0x00a0:   0x00000120
        0x00a4:   0x0000012c
        0x00a8:   0x000000d0
        0x00ac:   0x00000248
        0x00b0:   0x00000210
$code   0x00b4:   mov r3, r5
        ...
        0x00d0:   add r4, r6, r6
        ...
        0x0120:   ldr r4, [r5]
```

Pattern Matching: example 2



```
$code   0x0080:  mov r2, 0x09c
        0x0084:  cmp r1, $0
        0x0088:  jl 0x0b4
        0x008c:  cmp r1, $5
        0x0090:  jge 0x0b4
        0x0094:  add r1, r2, r1
        0x0098:  jmp r1
        0x009c:  jmp 0x0120
        0x00a0:  jmp 0x012c
        0x00a4:  jmp 0x00d0
        0x00a8:  jmp 0x0248
        0x00ac:  jmp 0x0210
        0x00b0:  nop
        0x00b4:  mov r3, r5
        ...
        0x00d0:  add r4, r6, r6
        ...
        0x0120:  ldr r4, [r5]
```

switch statement 2:
address table is replaced by a series of direct jumps to the switch cases.

unrecognized pattern!

Solution:
add pattern to Diablo

Procedure Calls and Returns

ARM indirect procedure call:

```
mov r14, pc
mov pc, r2
```

ARM procedure return:

```
mov pc, r14
```

or

```
ldr pc, [r13], #4
```

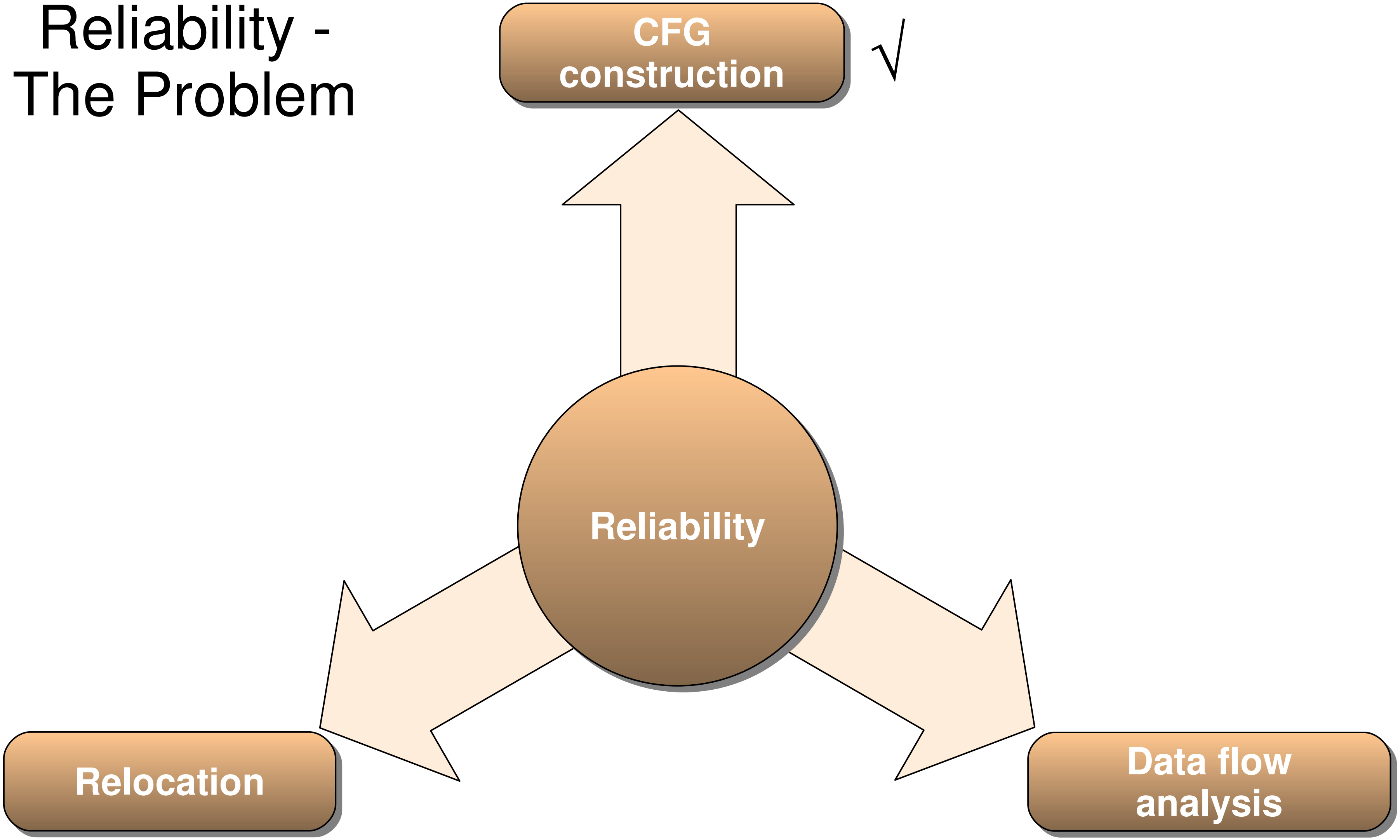
or

```
ldmia r13, {r4-r7,r15}!
```

Function calls and returns are often just “special” indirect jumps:
not recognizing them makes the flow graph much too conservative

Solution:
use pattern matching to recognize them:
- rely on ABI
- rely on compiler conventions

Reliability - The Problem



Data flow analyses

Stack analysis

Problem

- Difficult to analyse
- Necessary to improve precision
- Especially for C++-like languages (calls through function pointers)

Solution

- rely on calling-conventions
- use symbol information
- use mapping symbols
- use source code information
- use stack unwind information

Calling convention adherence

A.c

```
extern int B(int x);

int A(int x)
{
        return B(x);
}
```

B.c

```
int B(int x)
{
        return x * 2;
}

int C(int x)
{
        return B(x)*2;
}
```

B is unknown

call to B respects calling conventions

C is known
but *A is unknown*
B respects calling convention

Calling convention adherence

B.c

```
static int B(int x)
{
        return x * 2;
}

int C(int x)
{
        return B(x)*2;
}
```

C is known
no unknown callers of B
B does not need to respect calling convention

Calling convention adherence

A.c

```
extern int B(int x);

int A(int x)
{
        int y;
        asm("
            movl %ecx, x
            call B
            movl y, %ecx
        ");
        return y;
}
```

B.s

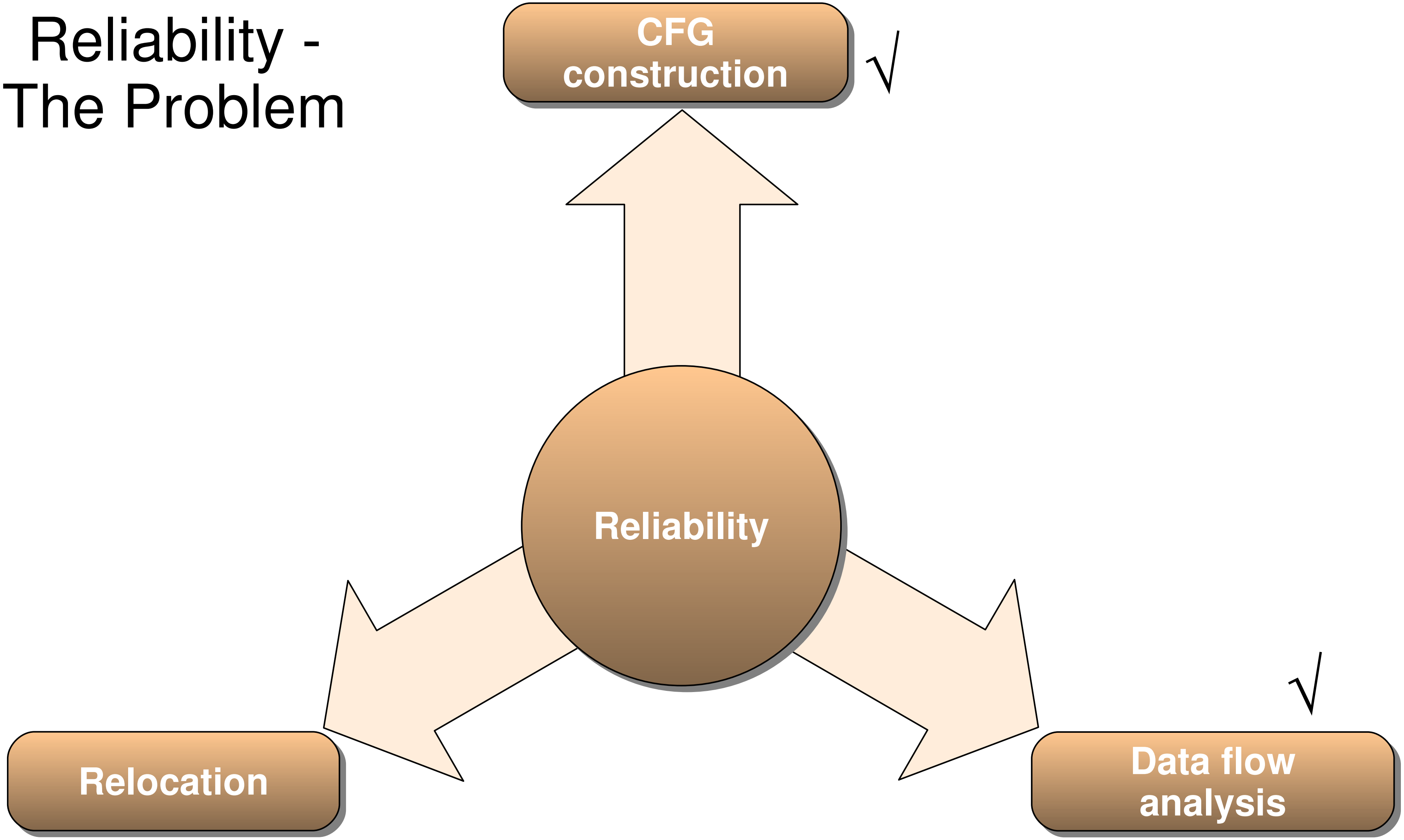
```
B:

shl %ecx, #1
ret
```

even though B is global, the programmer has control over all call sites
B does not need to adhere to calling conventions

solution:
identify assembler code through *mapping symbols* (for inline assembler) and object file header info

Reliability - The Problem



Relocation

Producing binary program again

Problem

- How to write a correct program?
- How to layout data?
- How to update pointers?
- How to update addresses?

Observation

- Most “strange” requirements come from linker manipulations

Solution

- make relocations expressive
- make relocations first class objects
- let transformations update relocations
- use linker scripts

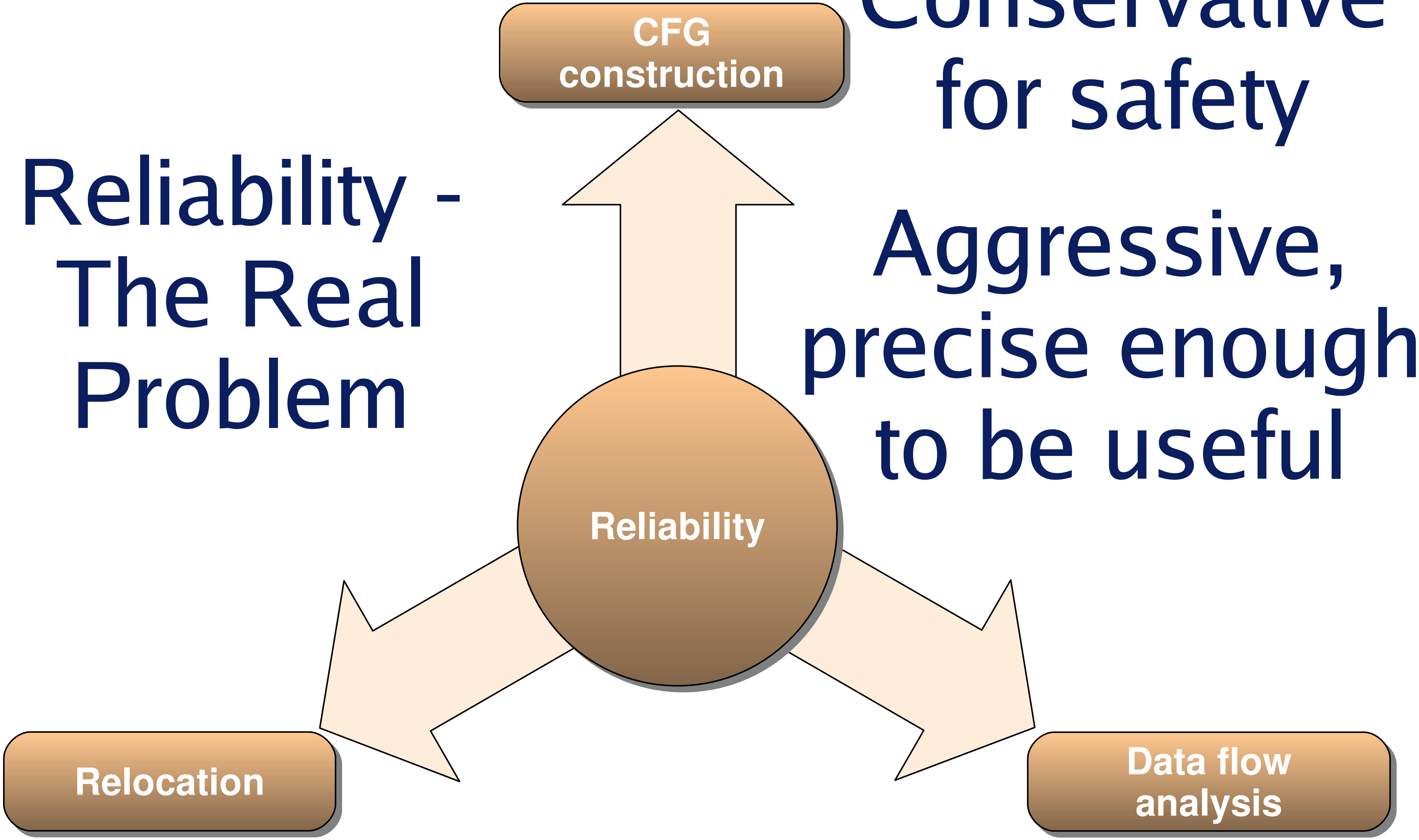
Overview

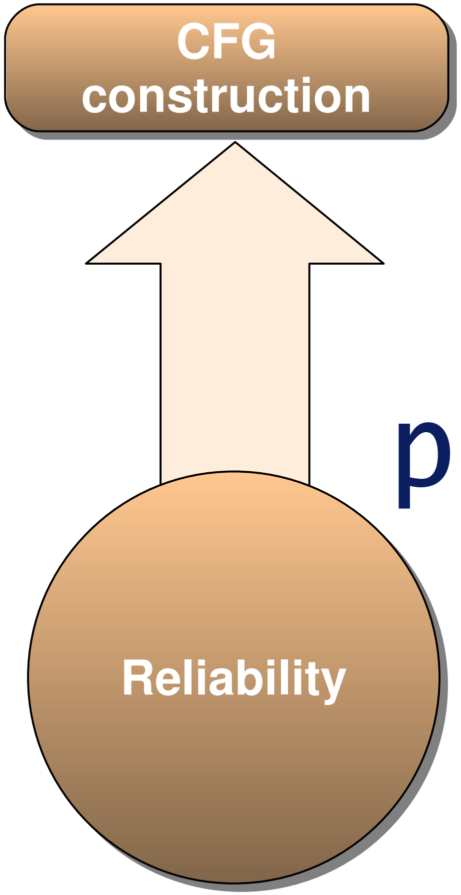
- Some background
- Diablo
  - Extensibility
  - Retargetability
  - Reliability (no, really now)

Reliability - The Real Problem

Conservative for safety

Aggressive, precise enough to be useful





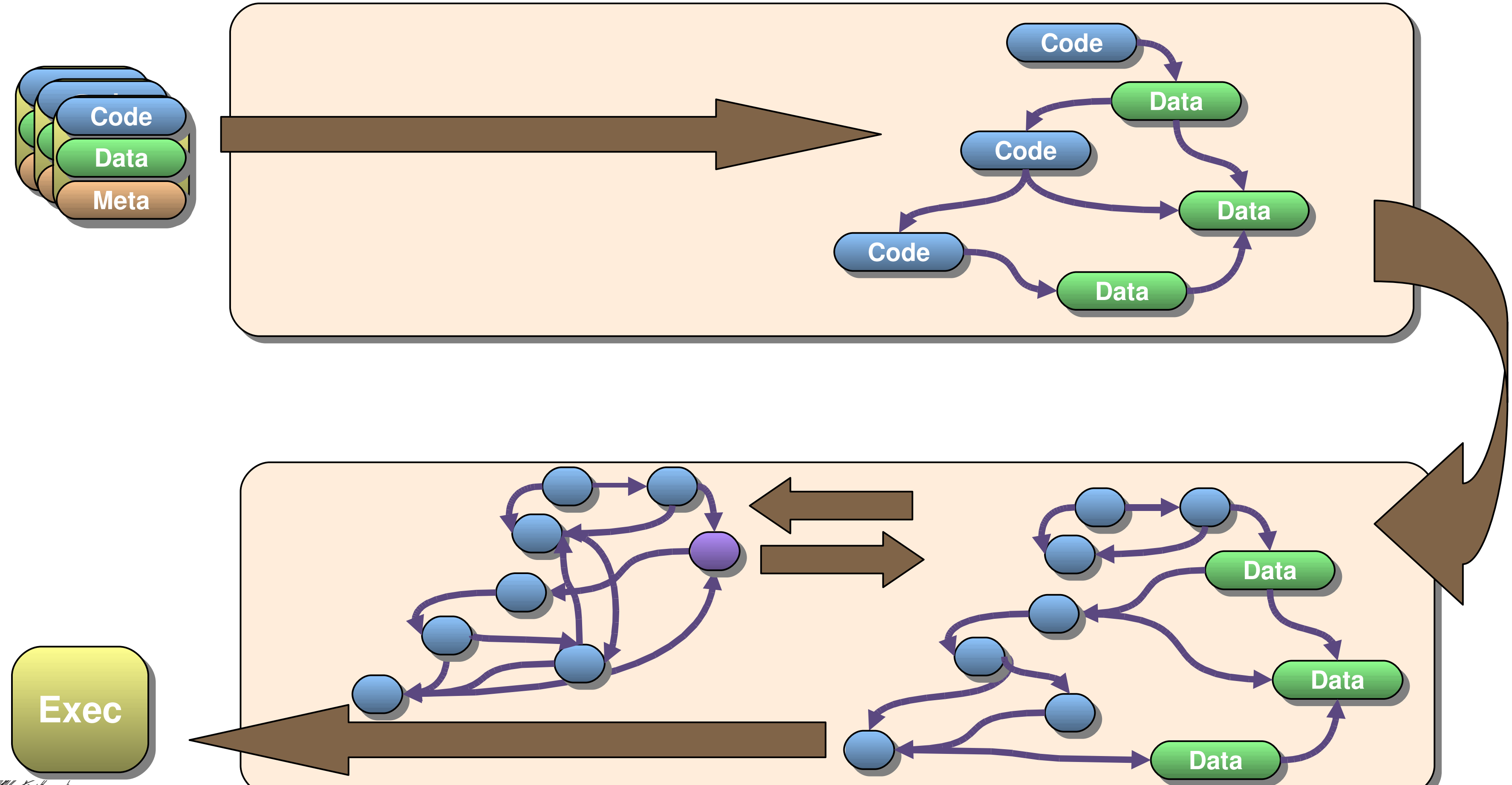
Conservative for safety

Reliability - The Real Problem

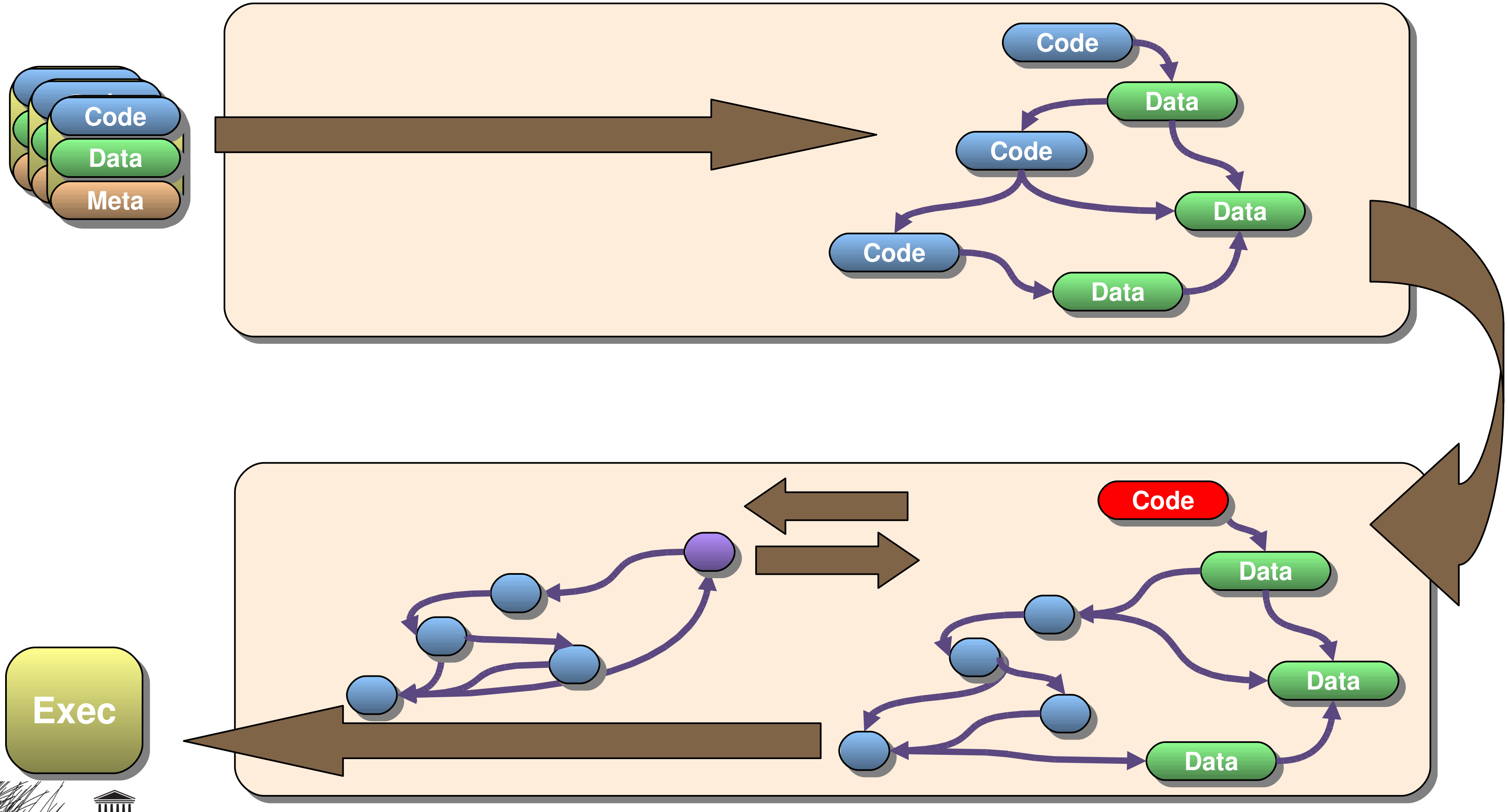
Aggressive, precise enough to be useful

Solution: limit most conservative assumptions to parts of program

Limit imprecision to some parts



Limit imprecision to some parts



What program parts?

- Sections from object files
  - only refer to each other via symbols
  - special code addresses identified by relocations
  - extend relocations where necessary
    - no relaxation
    - annotate PIC code with relocations if necessary
    - mark data
    - ...

When/why does this work?

- Under separate compilation
  - Partial-separate compilation
  - Compiler-generated code only, not manually-written assembler
- Compiler needs to maintain conventions
- Assembly writers do not know compiler-generated code
  - Because multiple compiler versions are available
- Whenever imprecision could become viral, the linker (rewriter) is informed!