Extraction and Traceability of Annotations for WCET Estimation

Hanbing LI, Inria Rennes, France

9 October 2015, INRIA Rennes

Funded by W-SEPT, L'Agence nationale de la recherche

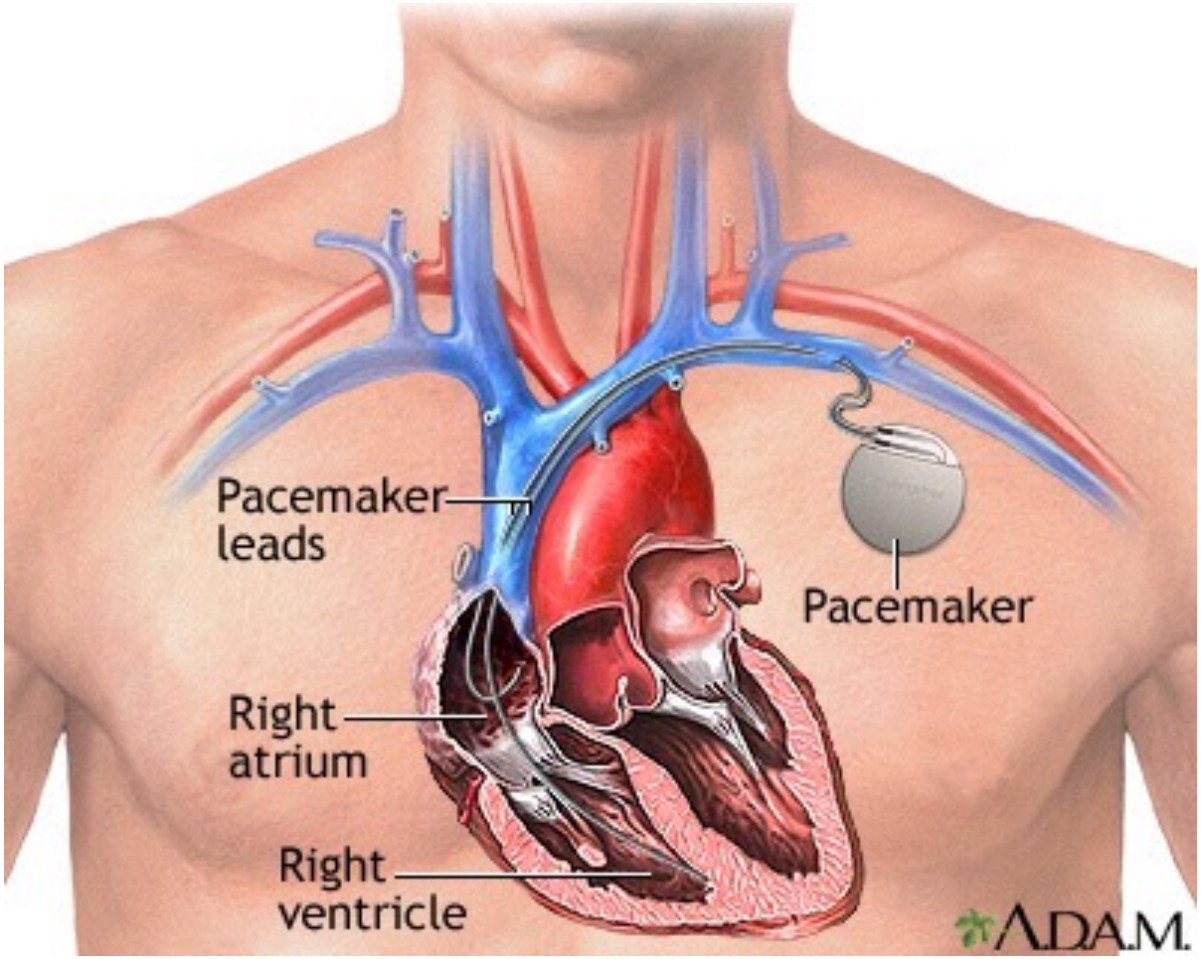
Motivation

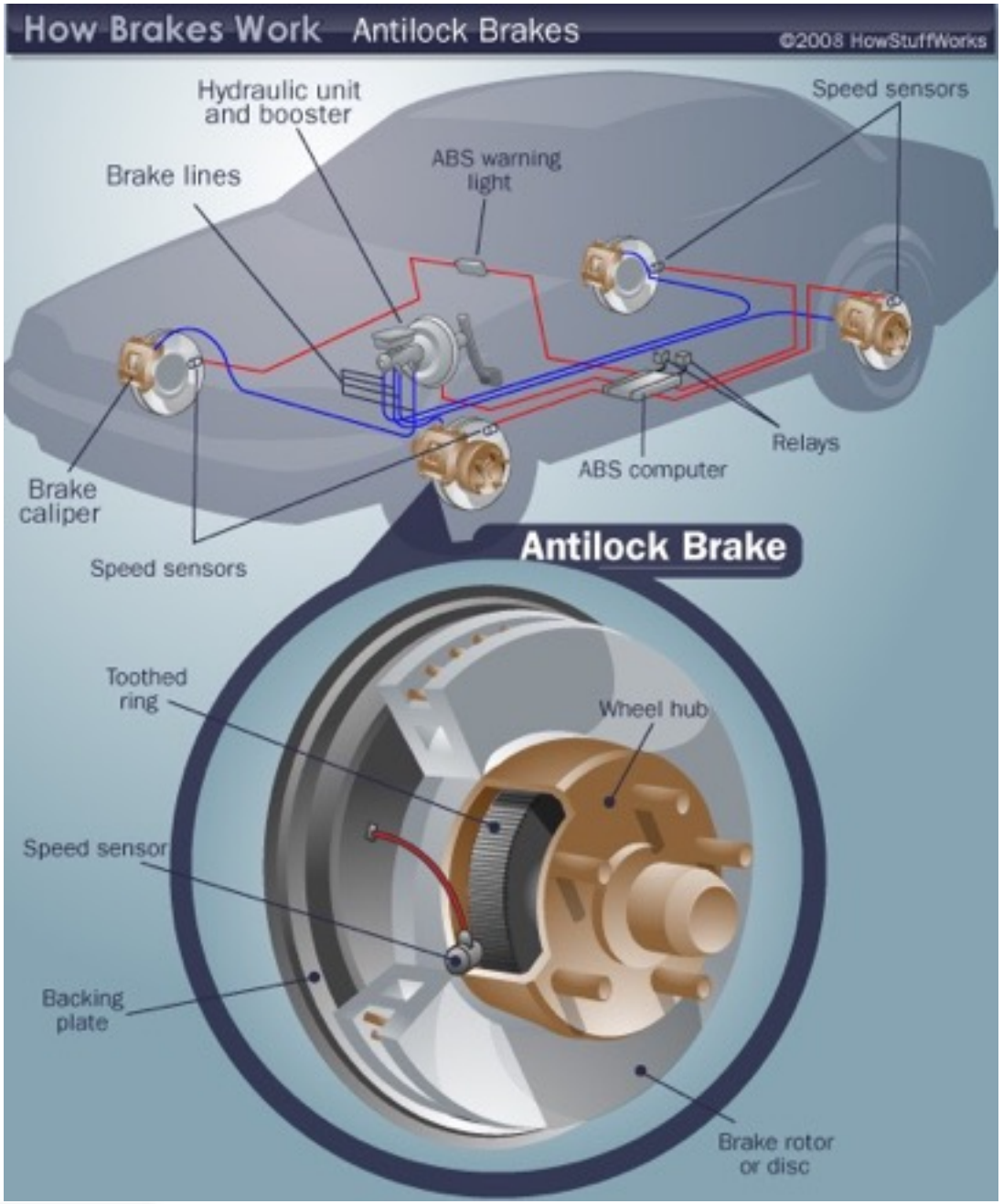
Real-time embedded system

- Embedded system
  - Systems with a function in a larger system
  - Hardware and software are unnoticed by the users
- Real-time system
  - Execution time is subject to timing constraints
  - Missing a deadline:
    - Hard real-time system: catastrophe
    - Soft real-time system: degrades the quality

Hard real-time system

- Components of pacemakers



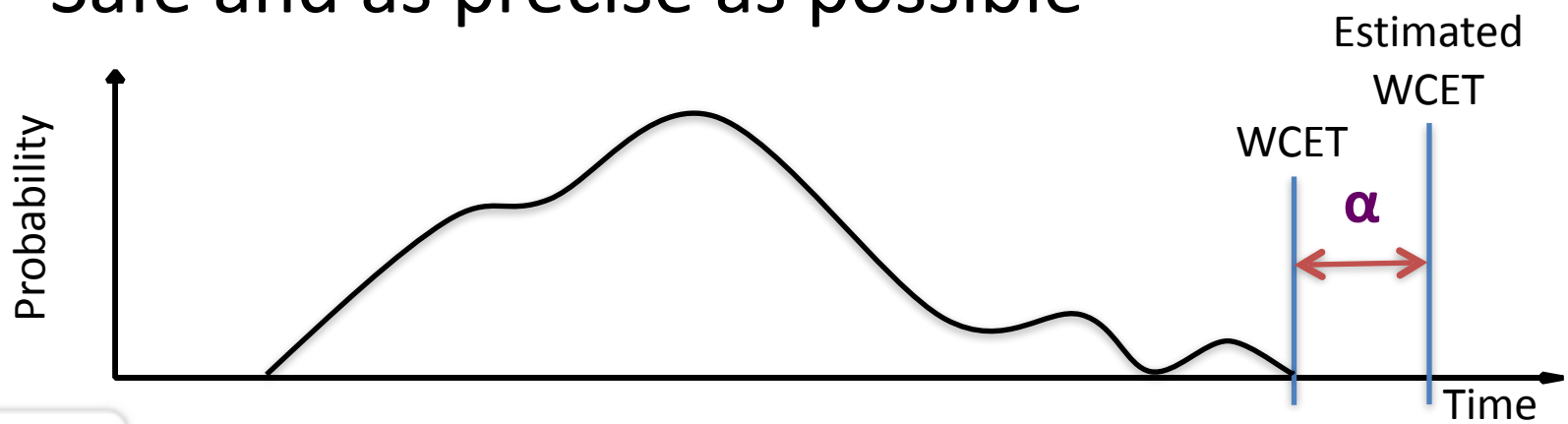


- Anti-lock braking system

- Aircraft control systems

WCET

- WCET: Worst-Case Execution Time
- To demonstrate that the system meets its timing constraints
  - Different inputs
  - For a given platform
- Safe and as precise as possible

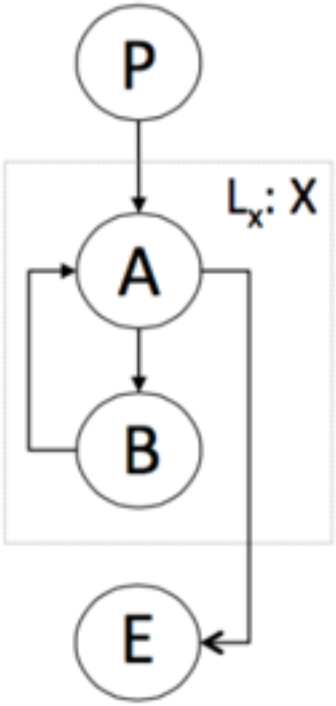


WCET estimation

- Static methods
  - Computed at machine code level
    - Need the timing of processor operations
  - Overestimate WCET
  - Emphasize safety
  - Precision?

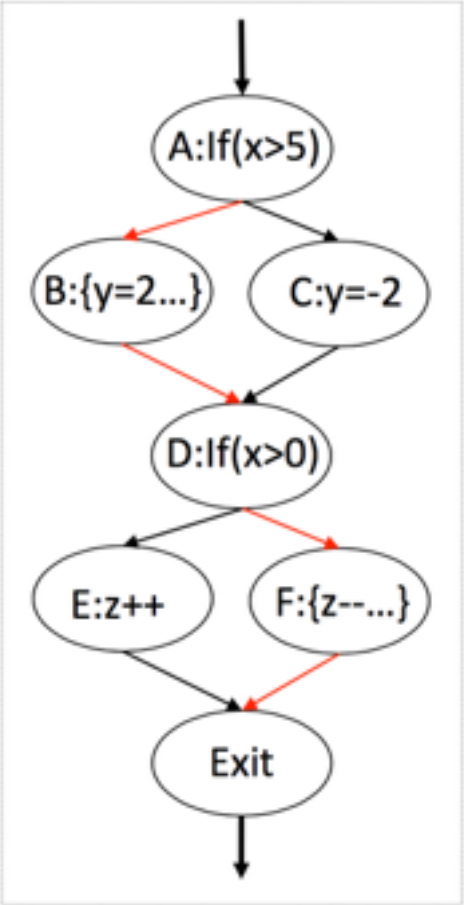
Flow information

- Flow information: Information on possible flows of control
  - Automatically
  - Manually at source code level
  - Example:
    - loop bound information



Flow information

- Flow information: Information on possible flows of control
  - Automatically
  - Manually at source code level
  - Example:
    - infeasible path

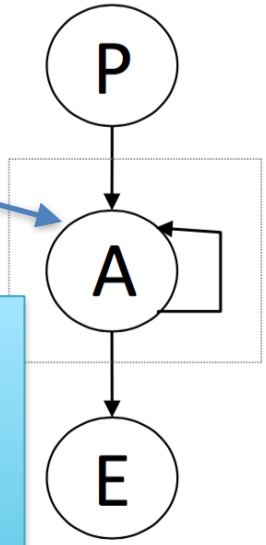


WCET calculation

- For compilation without optimizations:

loop_bound

```
for(i=0; i<X; i++) {
   body(i);
}
```



oop_bound*T_A +T_E

Basic Block: a straight-line piece of code within a program with only one entry point and only one exit point

CFG: a directed graph made of a set of nodes representing basic blocks, and a set of edges representing possible control flows between basic blocks

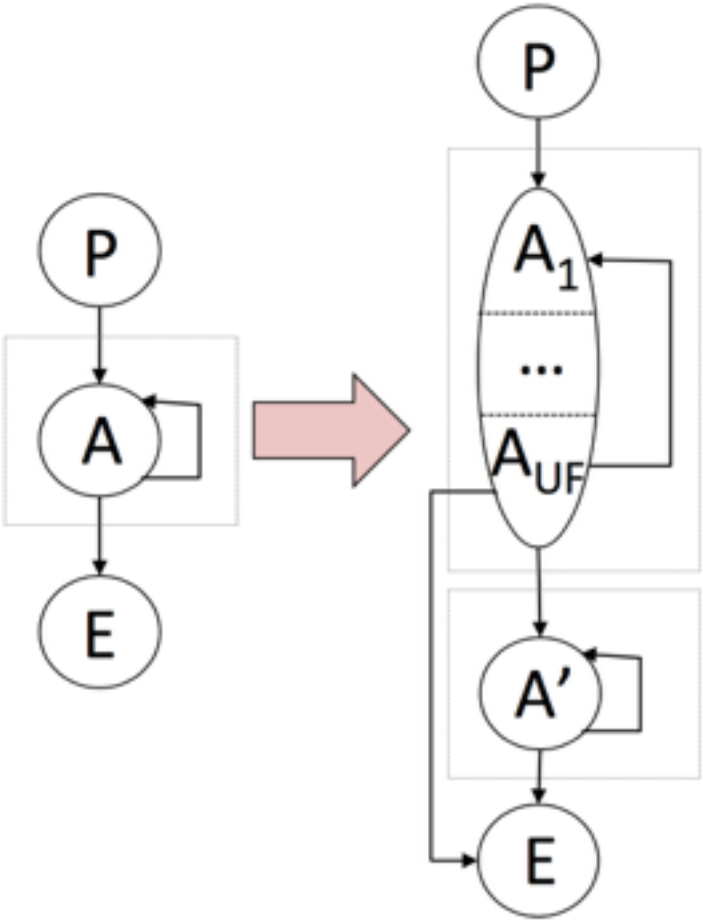
Compiler optimizations

- Modern compilers apply hundreds of optimizations
- Deliver more performance
- Modify the structure of code
- Affect WCET estimation

Example

Loop unrolling: replicate the body of the loop in one iteration according to the unrolling factor	
Advantages: reduce loop overhead and increase instruction parallelism	
Original code	Optimized code

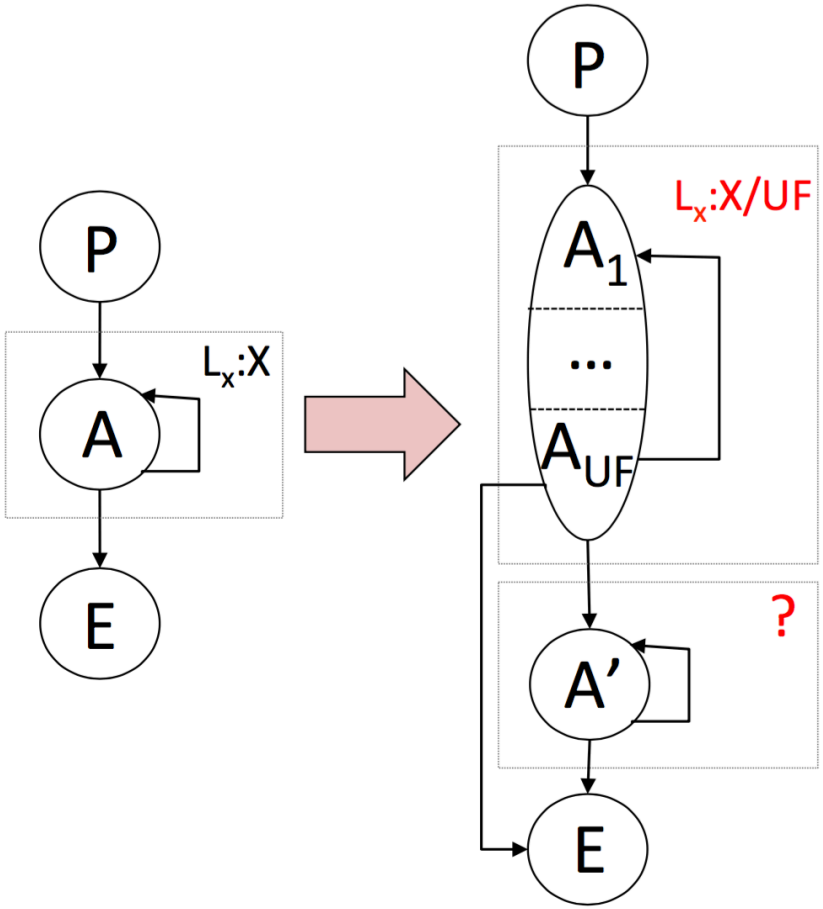
for(i=0; i<X; i++) {
  body(i);
}
	for(i=0; i<X; i+=UF) {
	  body(i);
	  body(i+1);
	  …
	  body(i+UF-1);
	}
	For(; i<X; i++) {
	  body(i);
	}



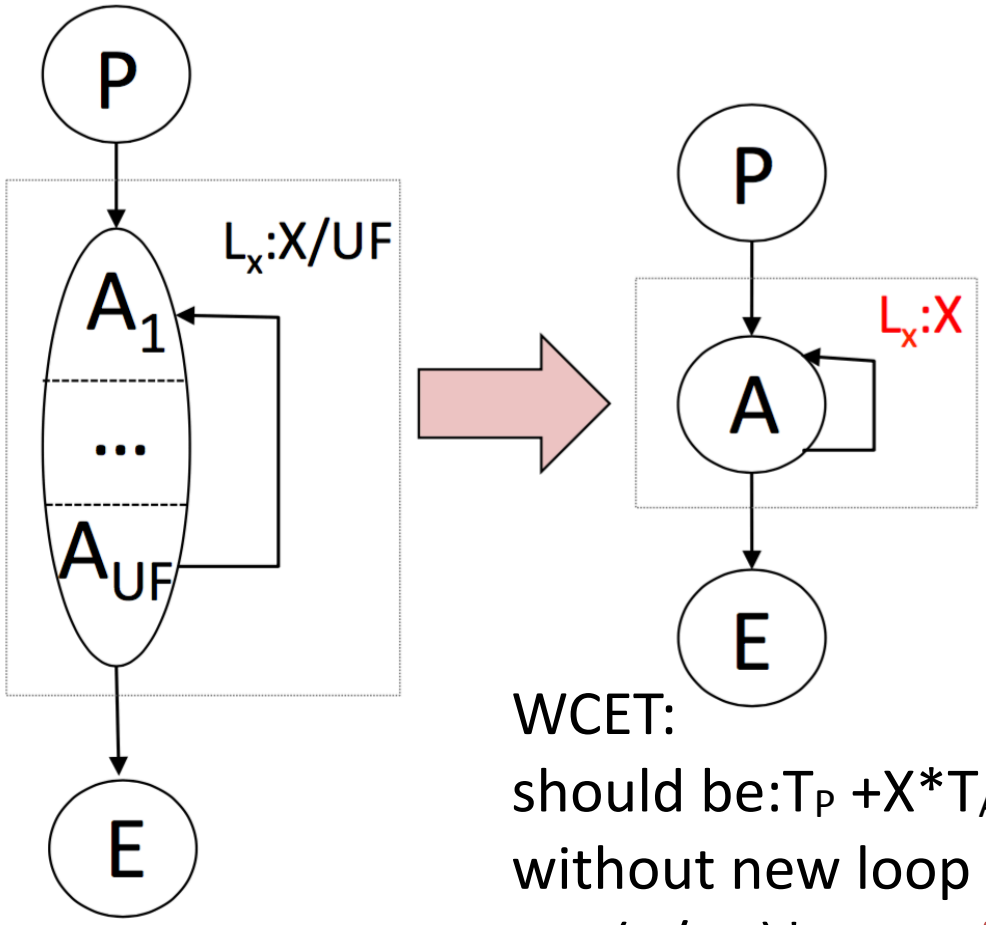
Example

- Change of structure of code
- Changes of flow information
- Make the WCET estimation not precise, not safe even impossible

for(i=0; i<X; i++) {
  body(i);
}
	for(i=0; i<X; i+=UF) {
	  body(i);
	  body(i+1);
	  …
	  body(i+UF-1);
	}
	For(; i<X; i++) {
	  body(i);
	}



Example



WCET:

should be: $T_P + X * T_A + T_E$

without new loop bound:

$T_P + (X/UF) * T_A + T_E$ (unsafe!)

loop rerolling

Related work

- Focus on different programming language
  - Java [Schoeberl et al., SP&E, 2010]
  - Lustre [Raymond et al., RTNS, 2013]
- Focus on one or several optimizations
  - Procedure cloning [Lokuciejewski et al., CODES, 2007]
  - Simple optimizations [Engblom et al., Euromicro, 1998]
- Focus on source-to-source transformation
  - TuBound [Knoop et al., LPAR, 2012]
  - Kirner et al. [PhD thesis, WORDS 2003, Real-Time Systems Journal 2010]

Contribution

- Transformation Framework
  - source-to-binary
  - C/C++
  - support most general optimizations
- Implementation within a modern optimizing compiler
- Experiments

Background

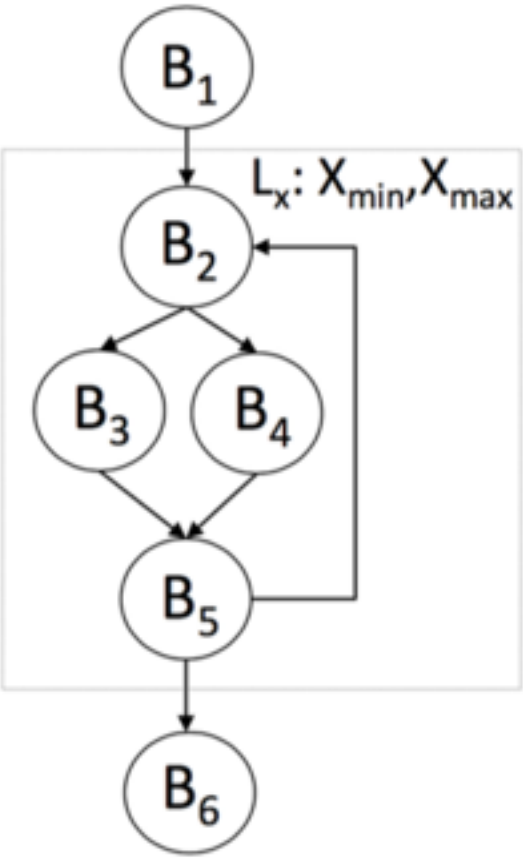
WCET estimation

- Calculation techniques:
  - Static methods
    1. Extract the CFG from program
    2. Analyze the CFG and find possible paths
    3. Obtain execution time of BB on hardware model
    4. Combine them and get the WCET result
  - Measurement-based methods

IPET

- Implicit Path Enumeration Technique
- Most common WCET calculation method
- Operates on CFG (control flow graph)
- Models the WCET calculation as an ILP (Integer Liner Programming) formulation

Timing analysis with IPET



Objective function

$$Maximize \sum_{i \in CFG} f_i \times T_i$$

Structural constraints

$$f_1 = 1$$
$$f_{1\_2} + f_{5\_2} = f_{2\_3} + f_{2\_4} = f_2$$
$$f_{2\_3} = f_{3\_5} = f_3$$
$$f_{2\_4} = f_{4\_5} = f_4$$
$$f_{3\_5} + f_{4\_5} = f_{5\_2} + f_{5\_6} = f_5$$
$$f_{5\_6} = f_6$$

Additional constraints

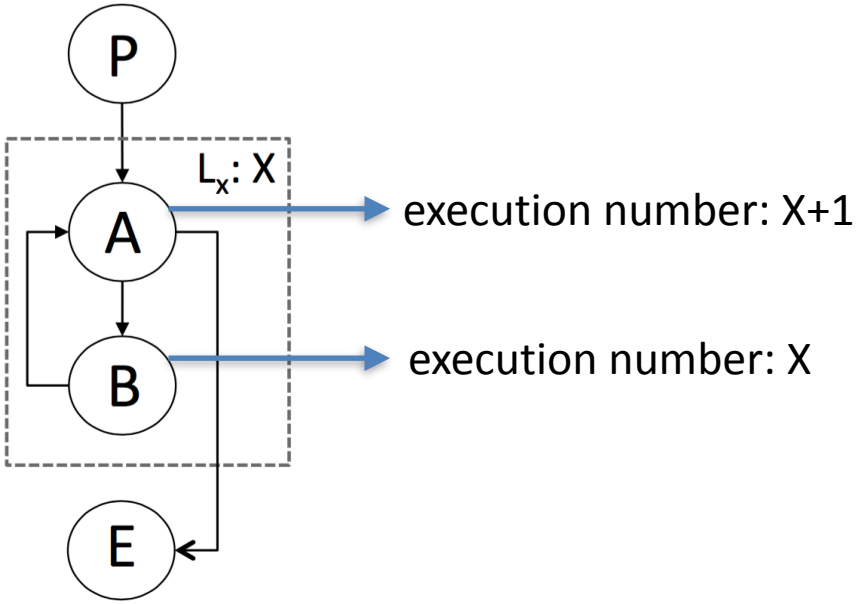
$$f_2 \leq X_{max}$$
$$f_2 \geq X_{min}$$
$$f_3 \leq 2 \times f_4$$

Flow information

- Constraints are known as flow information
- Use annotation to represent flow information
- Needed by most WCET analysis methods and tools
- Content
  - Loop bound: the essential part
  - Infeasible path

Loop bound

- Loop bounds
  - the maximum number of executions of the nodes in the loop body except the node(s) testing the loop exit

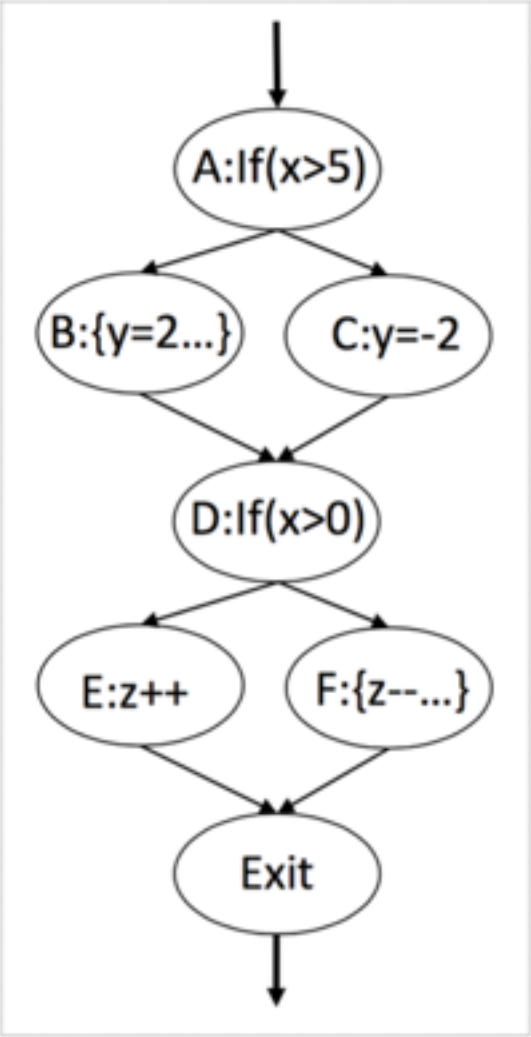


Infeasible path

- Infeasible path
  - executable according to the CFG
  - not feasible when considering the semantics of the program, the context and possible inputs
  - tighten WCET, not indispensable

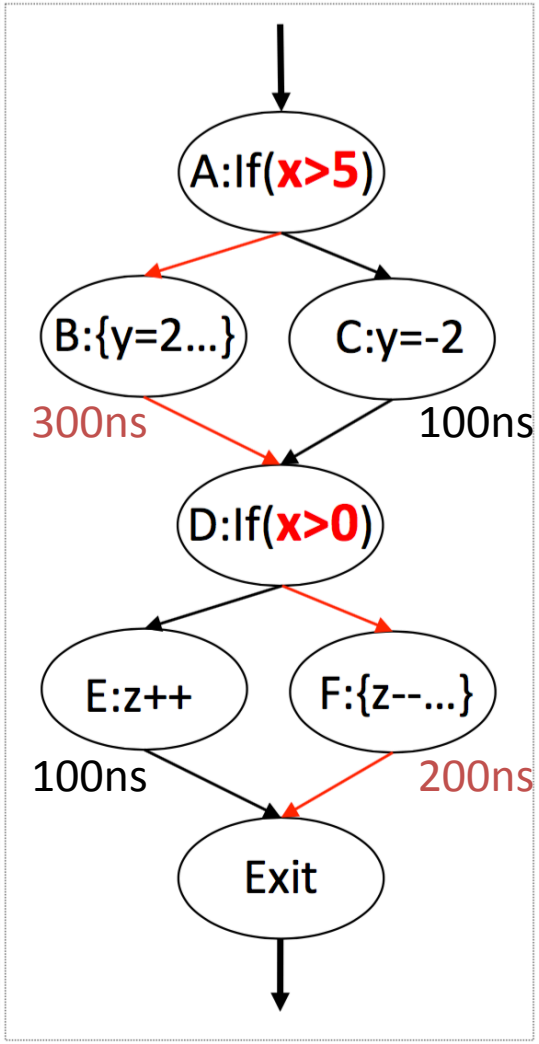
Example of infeasible path

```
if (x>5){
   y=2;
   a=3;
   z=a+y;
}
else
   y=-2;
if (x>0)
   z++;
else{
   z--;
   b=4;
}
```



Example of infeasible path

```
if (x>5){
   y=2;
   a=3;          300ns
   z=a+y;
}
else
   y=-2;         100ns
if (x>0)
   z++;          100ns
else{
   z--;          200ns
   b=4;
}
```



Normal WCET calculation: 300ns+200ns

Calculation with infeasible path: 300ns+100ns

Annotation

- Method of addition
  - Manual annotations
    - error-prone
  - Automatic methods/static analysis tools
    - example: Trickle [Barany et al., RTAS, 2014] and oRange [Michiel et al., ISoLA, 2010]
  - Hybrid methods
    - based on automatic extraction
    - use manual addition to fix unobtainable parts
    - example: aiS[1] and Bound-T[2]

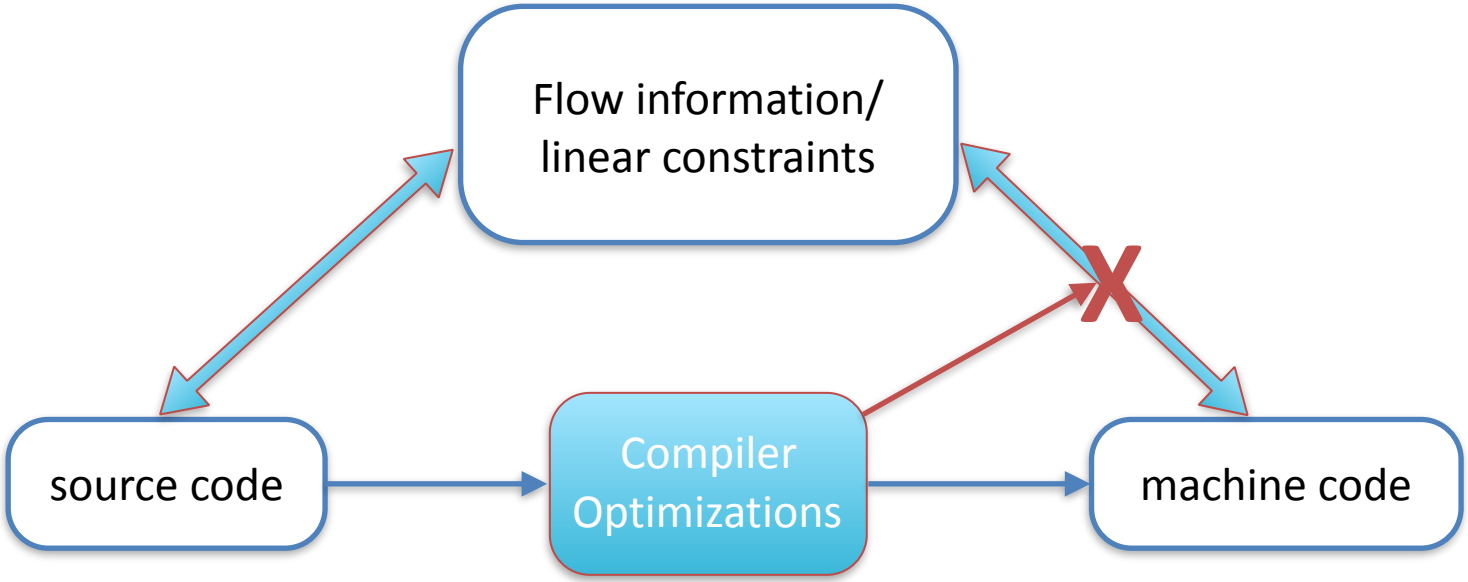
[1]www.absint.com/ait
[2]www.bound-t.com

Transformation Framework

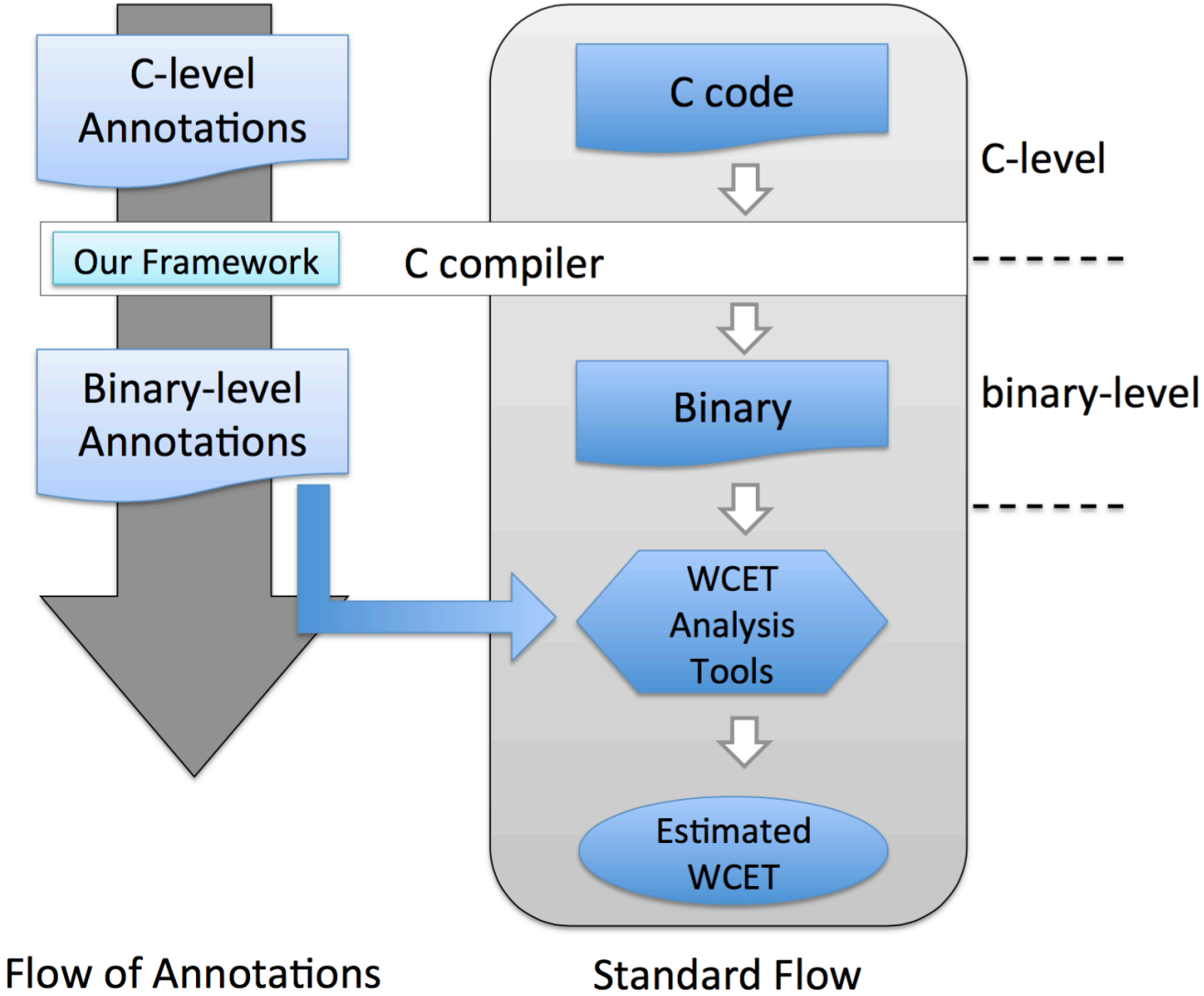
main contribution

Transformation Framework

- Linear constraints represent the flow information in the CFG of source code
- Optimizations break the mapping between the constraints and CFG/code



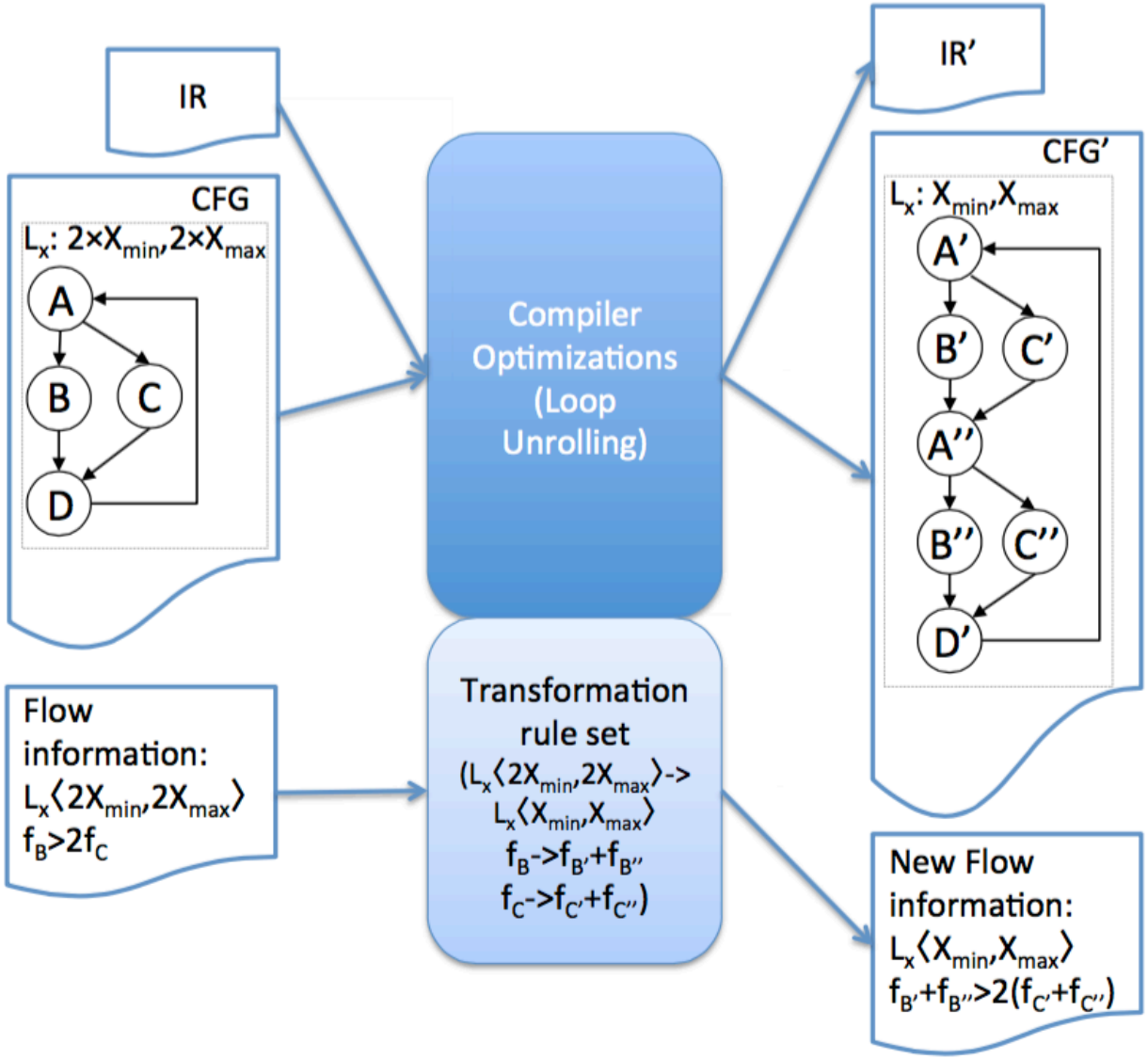
Transformation Framework



Transformation Framework

- Supports common compiler optimizations
- Sets of rules to rewrite constraints in IPET
- Convey flow information from source code level to binary code level

Example of input and output



Transformation rules

- Three basic classes of rewriting rules for transforming flow information:
  - Change rule: $\alpha \rightarrow \beta$
    - Optimizations change execution counts of basic blocks
    - Optimizations change loop bounds
  - Removal rule: $\alpha \rightarrow \varnothing$
    - Optimizations remove basic block or loop from CFG
  - Addition rule
    - Optimizations add new basic block or loop to CFG

Example of change rule

Array **a** is written by the first loop, and read by the second one.
Move some computation from one loop to another to minimize the parallel execution time.

```
for(i=0; i<X; i++)
{
  a[i]=a[i]+d;
}
for(i=0; i<X+Y; i++)
{
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
```

(a) Source code

```
for(i=0; i<X; i++)
{
  a[i]=a[i]+d;
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
for(i=X; i<X+Y; i++)
{
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
```

(b) Code after loop spreading

Example of change rule

Array **a** is written by the first loop, and read by the second one.
Move some computation from one loop to another to minimize the parallel execution time.

```
for(i=0; i<X; i++)
{
  a[i]=a[i]+d;
}
for(i=0; i<X+Y; i++)
{
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
```

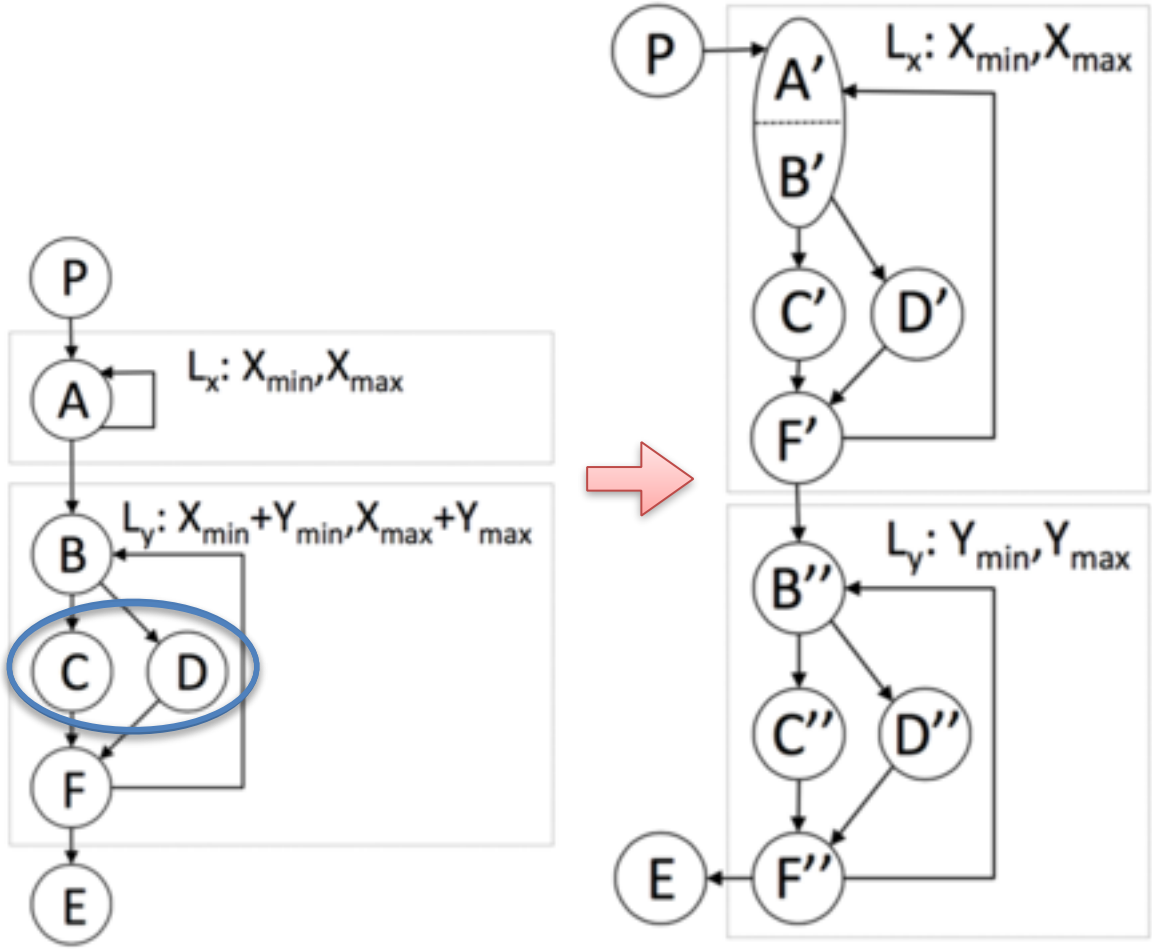
(a) Source code

```
for(i=0; i<X; i++)
{
  a[i]=a[i]+d;
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
for(i=X; i<X+Y; i++)
{
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
```

(b) Code after loop spreading

Loop bound changes from **X+Y** to **Y**

Example of change rule



For change of basic block:

Orig: $f_C \geq 2f_D$

Rule: $f_C \rightarrow f_{C'} + f_{C''}$

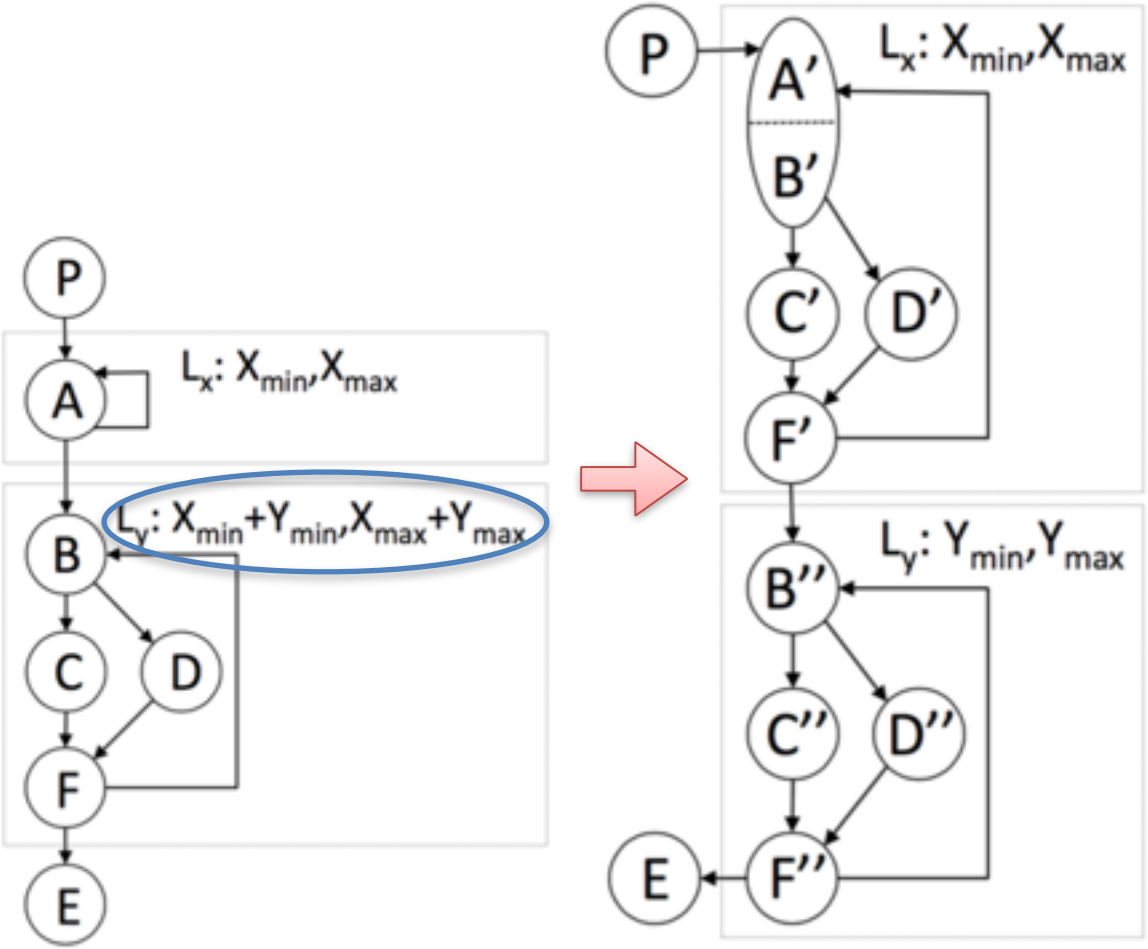
$f_D \rightarrow f_{D'} + f_{D''}$

New: $f_{C'} + f_{C''} \geq 2(f_{D'} + f_{D''})$

For change of loop bound:

Rule: $L_y \langle X_{min} + Y_{min}, X_{max} + Y_{max} \rangle \rightarrow L_y \langle Y_{min}, Y_{max} \rangle$

Example of change rule



For change of basic block:

Orig: $f_C \geq 2f_D$

Rule: $f_C \rightarrow f_{C'} + f_{C''}$

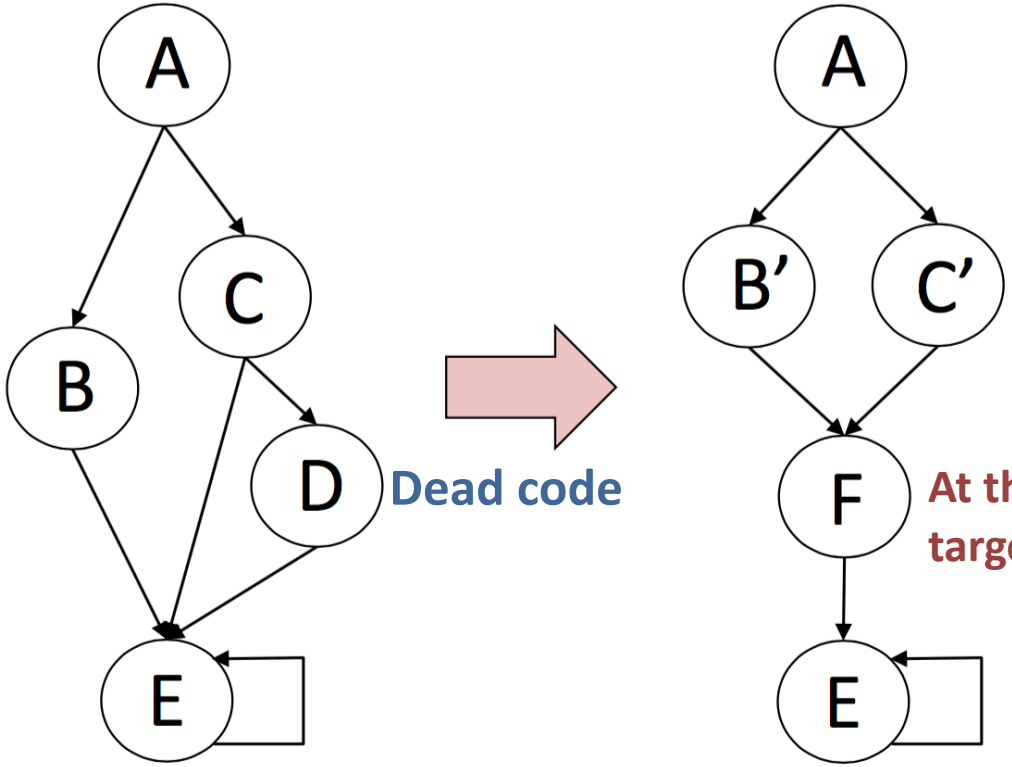
$f_D \rightarrow f_{D'} + f_{D''}$

New: $f_{C'} + f_{C''} \geq 2(f_{D'} + f_{D''})$

For change of loop bound:

Rule: $L_y \langle X_{min} + Y_{min}, X_{max} + Y_{max} \rangle \rightarrow L_y \langle Y_{min}, Y_{max} \rangle$

Example of removal & addition rule



Removal rule:

Orig: $f_B \geq 3f_D$

Rule: $f_D \rightarrow \varnothing$

New: $\varnothing$

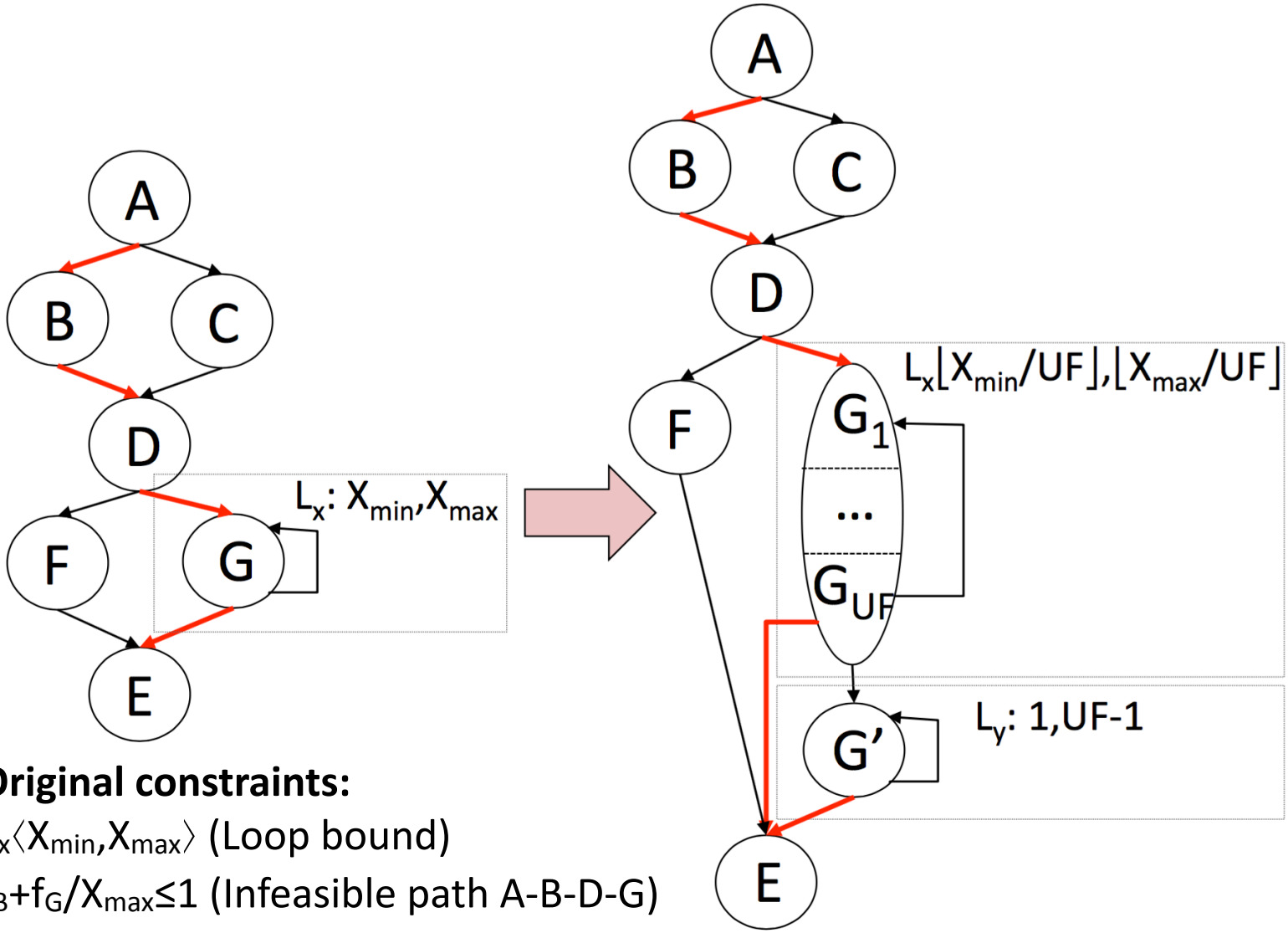
At the end of B and C, same instructions targeting same destination

Addition rule:

Rule: $f_F = f_{B'} + f_{C'}$

Dead-code elimination: reduce code size
Tail merging: eliminate duplicates in the code

Loop unrolling



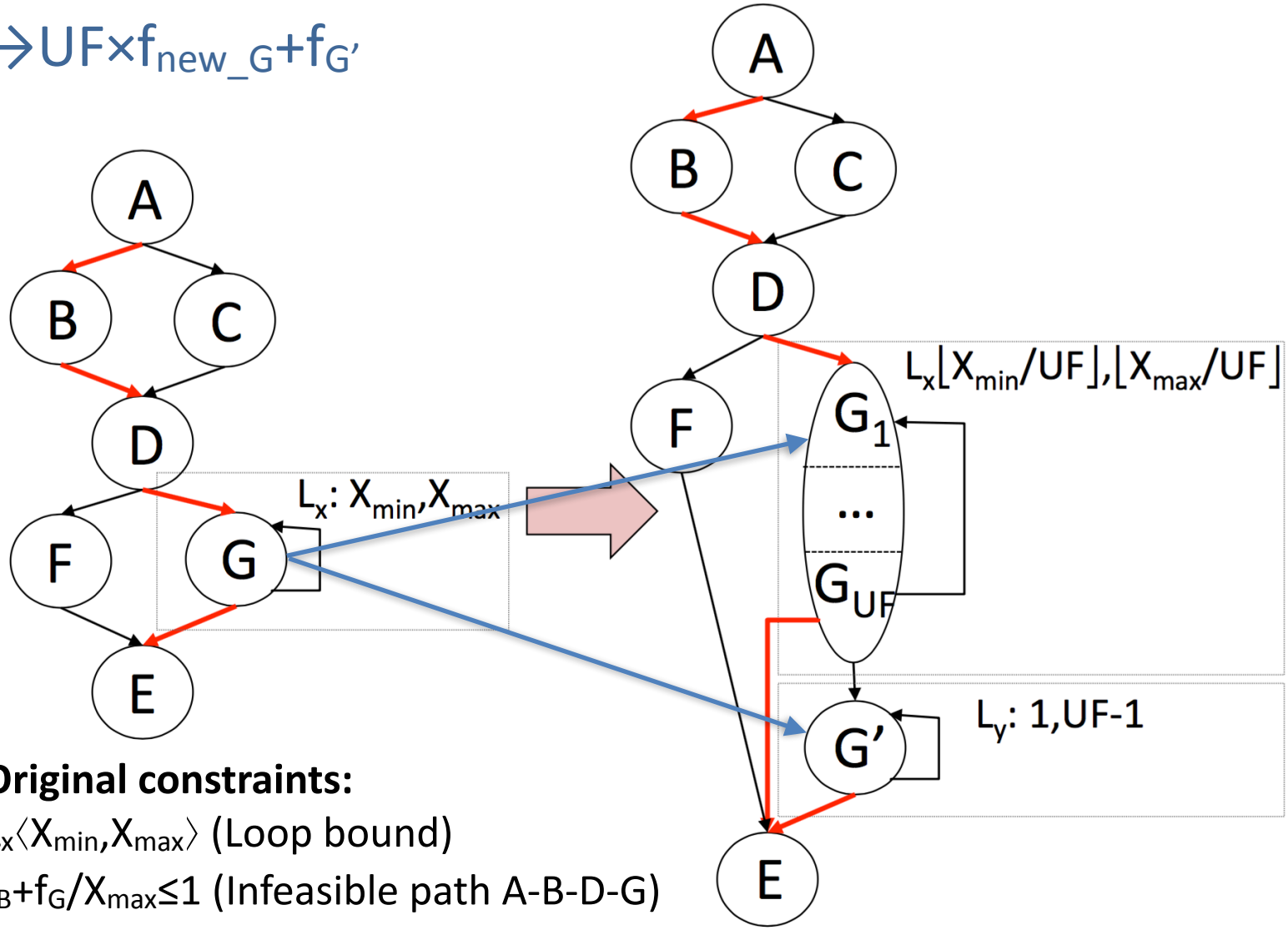
Original constraints:

$L_x\langle X_{min},X_{max}\rangle$ (Loop bound)

$f_B+f_G/X_{max}\leq 1$ (Infeasible path A-B-D-G)

Loop unrolling

$f_G \rightarrow UF \times f_{new\_G} + f_{G'}$



Original constraints:

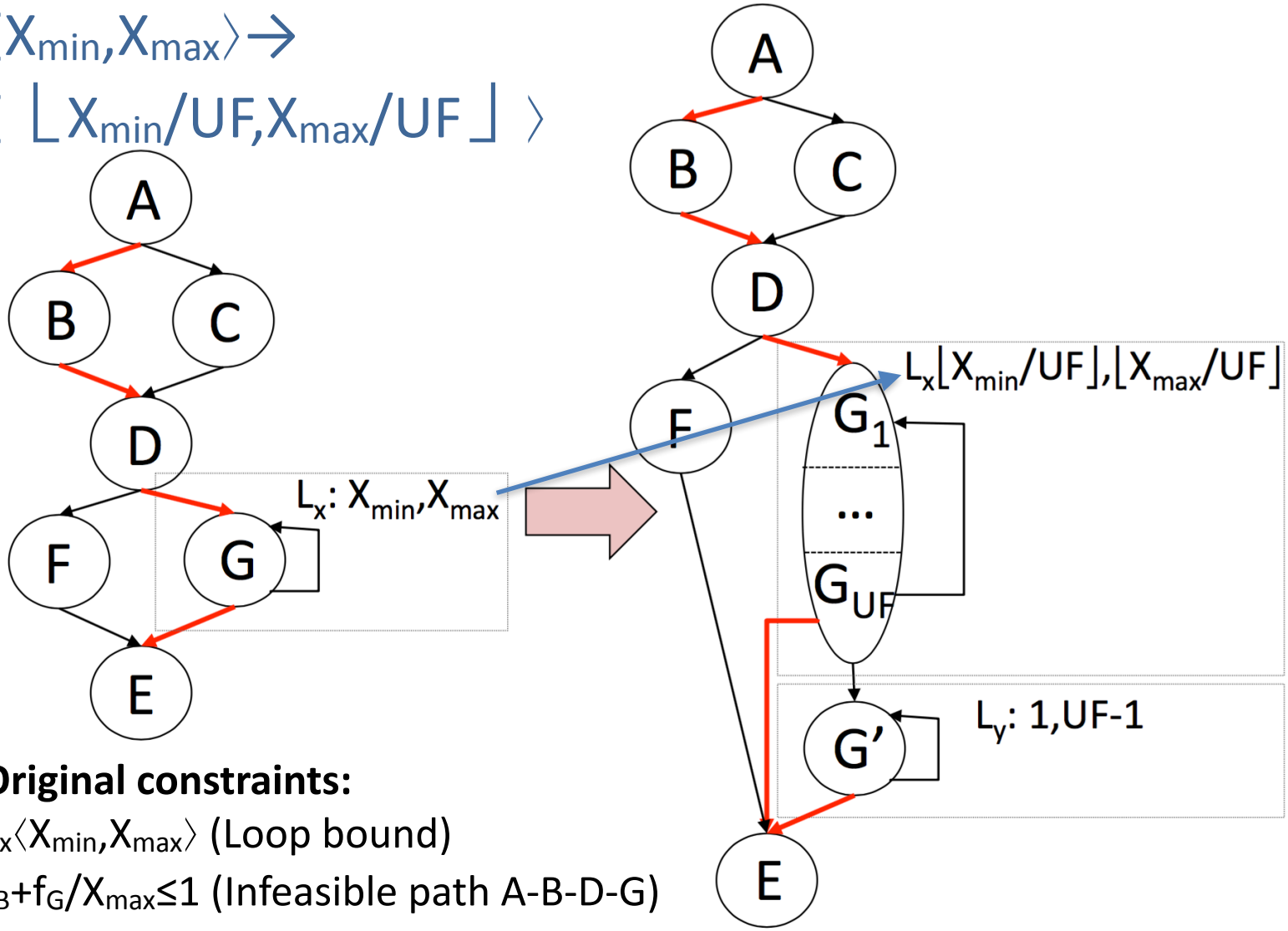
$L_x\langle X_{min},X_{max}\rangle$ (Loop bound)

$f_B+f_G/X_{max} \leq 1$ (Infeasible path A-B-D-G)

Loop unrolling

$L_x\langle X_{min},X_{max}\rangle \rightarrow$

$L_x\langle \lfloor X_{min}/UF, X_{max}/UF \rfloor \rangle$



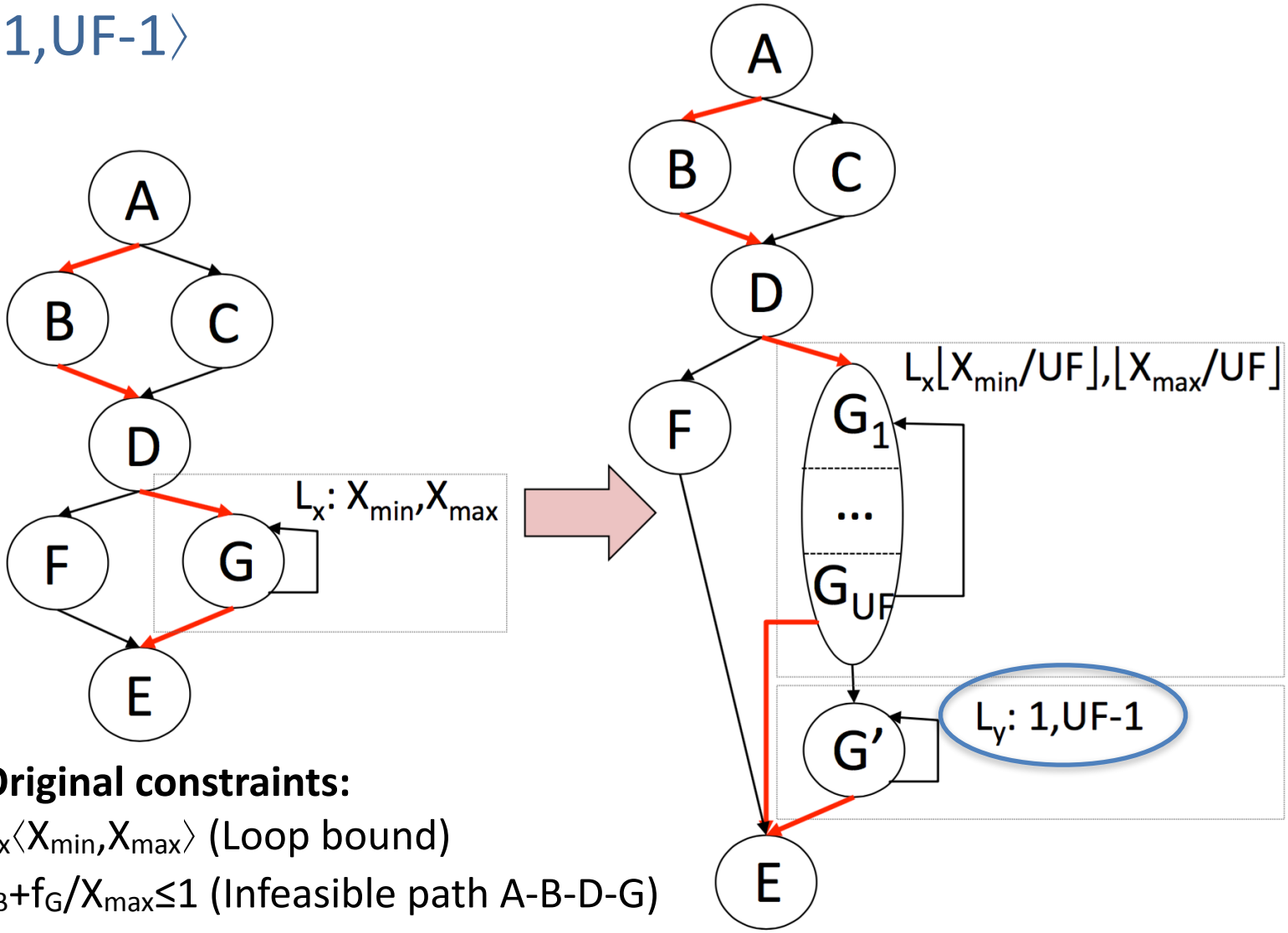
Original constraints:

$L_x\langle X_{min},X_{max}\rangle$ (Loop bound)

$f_B+f_G/X_{max}\leq 1$ (Infeasible path A-B-D-G)

Loop unrolling

$L_Y\langle 1,UF-1\rangle$



Original constraints:

$L_x\langle X_{min},X_{max}\rangle$ (Loop bound)

$f_B+f_G/X_{max}\leq 1$ (Infeasible path A-B-D-G)

Loop unrolling

Rules:

$L_X\langle X_{min},X_{max}\rangle \rightarrow L_X\langle \lfloor X_{min}/UF, X_{max}/UF \rfloor \rangle$

$L_Y\langle 1,UF-1\rangle$

$f_G \rightarrow UF \times f_{new\_G} + f_{G'}$

New constraints:

$L_X\langle \lfloor X_{min}/UF, X_{max}/UF \rfloor \rangle$ (Loop bound)

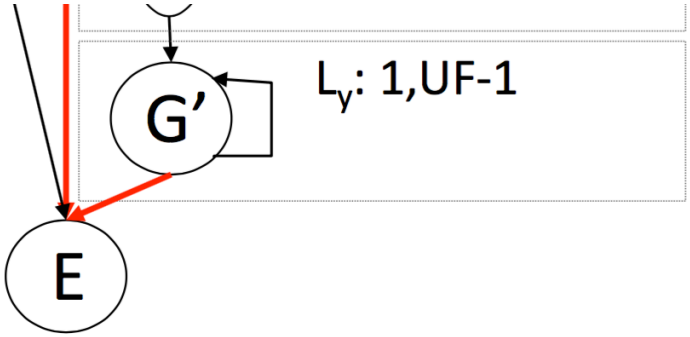
$L_Y\langle 1,UF-1\rangle$ (Loop bound)

$f_B + (UF \times f_{new\_G} + f_{G'})/X_{max} \leq 1$ (Infeasible path A-B-D-new_G-G')

Original constraints:

$L_x\langle X_{min},X_{max}\rangle$ (Loop bound)

$f_B + f_G/X_{max} \leq 1$ (Infeasible path A-B-D-G)



Loop unrolling

$L_X\langle \lfloor X_{min}/UF, X_{max}/UF \rfloor \rangle$

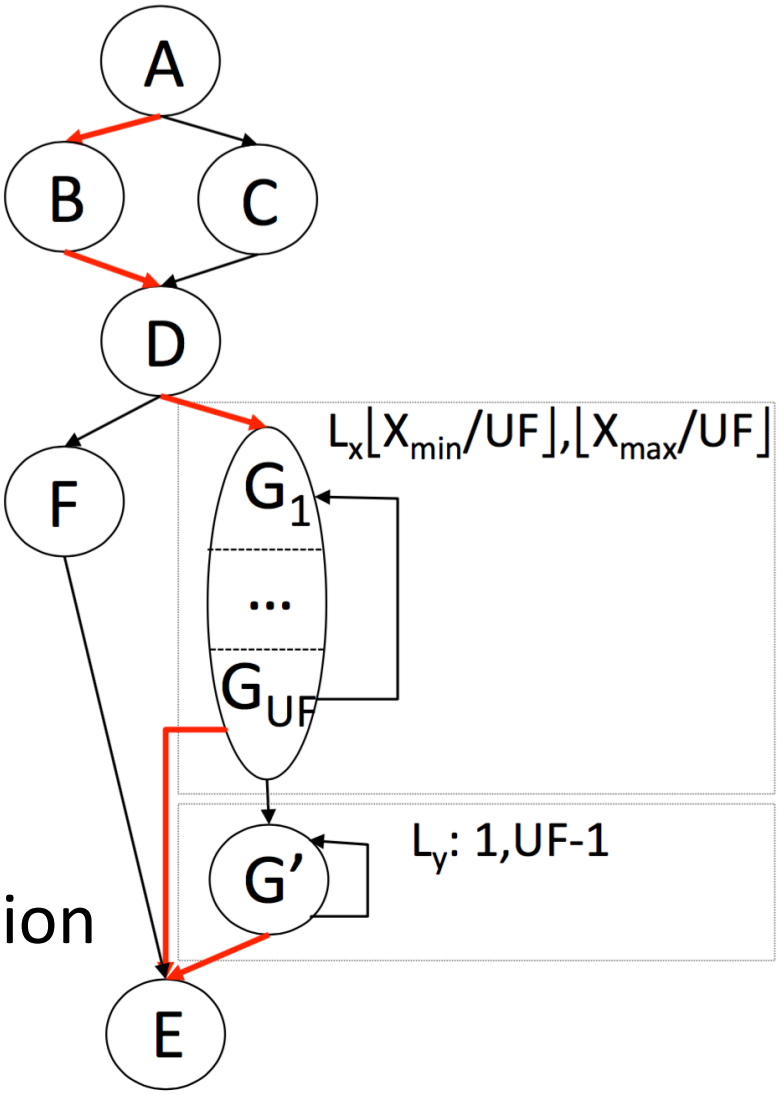
(Loop bound)

$L_Y\langle 1, UF-1 \rangle$

(Loop bound)

$f_B+(UF\times f_{new\_G}+f_{G'})/X_{max}\leq 1$

(Infeasible path A-B-D-new_G-G')



Safe and no loss of flow information

Supported compiler optimizations

- Support general optimizations (O3) in compilers
  - Redundancy elimination, control-flow and low-level optimizations
    - adce, correlated propagation, deadargelim, dse, early-cse, functionattrs, globalopt, ipsccp, jump-threading, mem2reg, sroa …
  - Loop optimizations
    - loop-simplify, lcssa, licm, loop-unswitch, indvars, loop-idiom, loop-deletion, loop rotation, loop-unroll, , loop interchange, loop fission, loop fusion

Implementation

WCET analysis tools

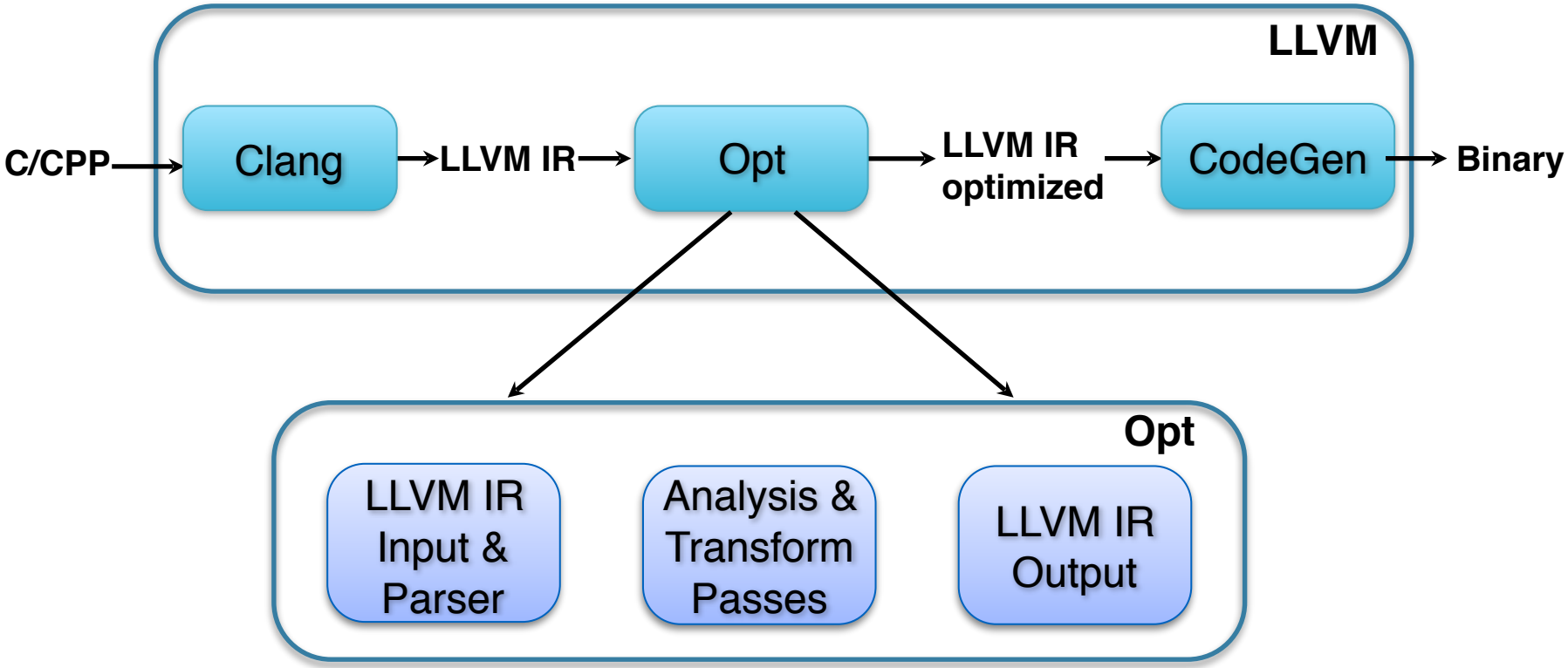
- Heptane[1] (IRISA)
  - IPET
  - MIPS and ARM
  - lp_solve or CPLEX
- OTAWA[2] (IRIT: one of the partners of W-SEPT)
  - IPET
  - A large range of target architecture (PowerPC, ARM,…)
  - lp_solve

[1]team.inria.fr/alf/software/heptane
[2]www.otawa.fr

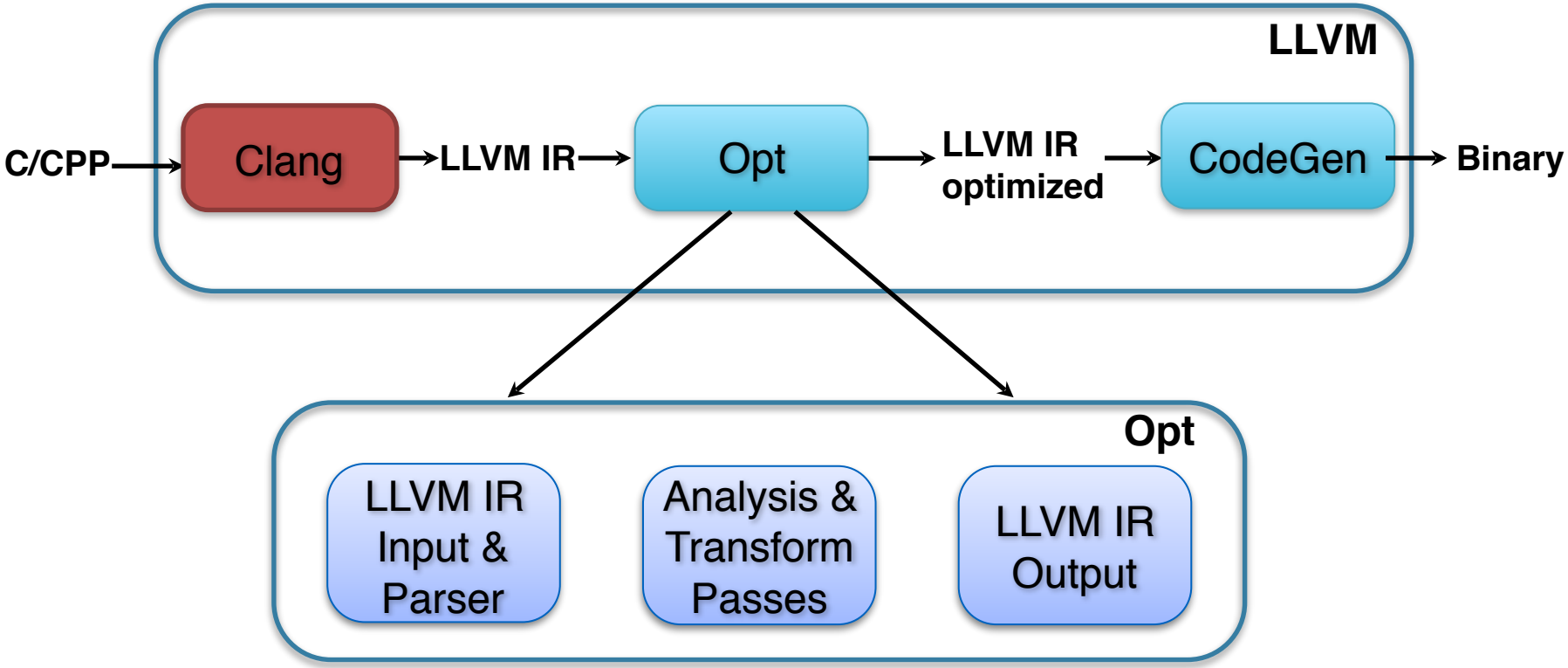
Flow information extraction

- oRange (IRIT) [Michiel et al., ISoLA, 2010]
  - static analysis tool
  - work on C source code
  - determine flow information including loop bounds
- FFX (Flow Facts in XML)
  - Portable WCET annotation language
  - store flow information

LLVM

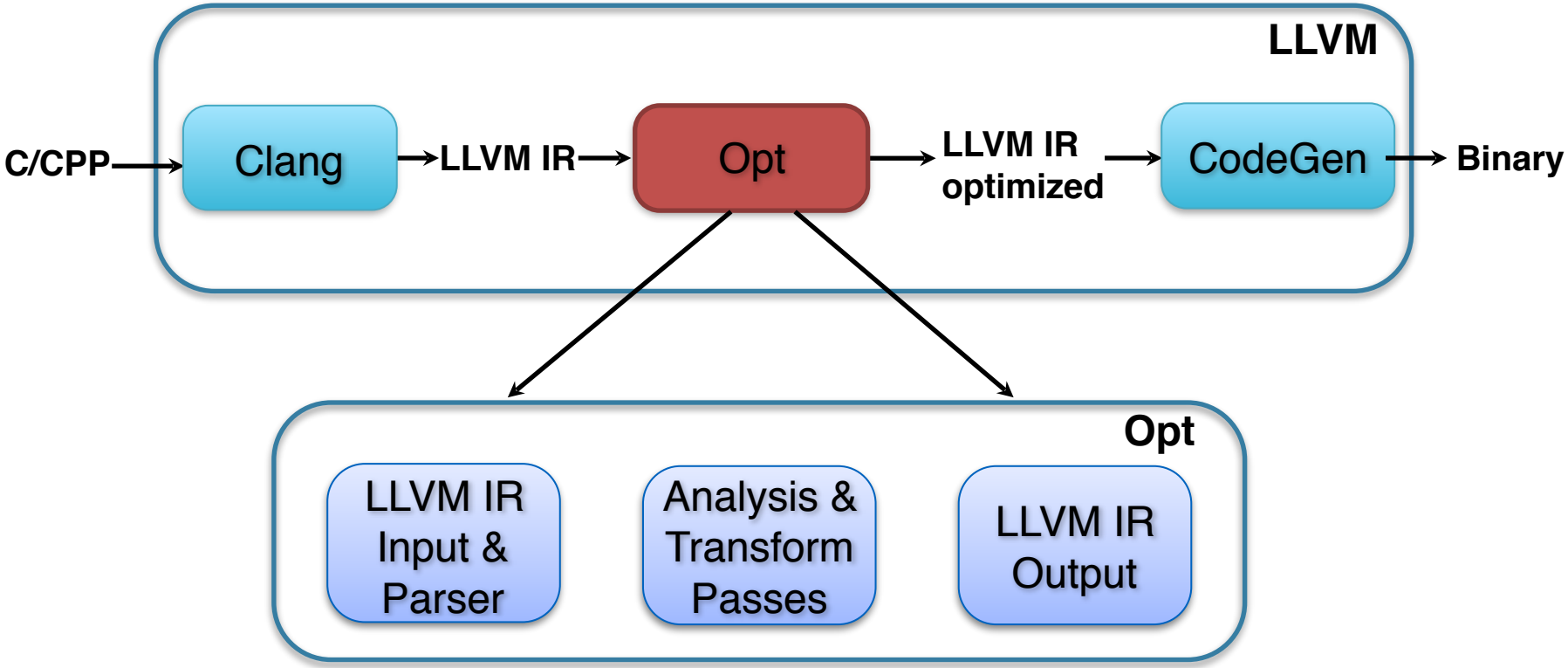

LLVM
C/CPP
Clang
LLVM IR
Opt
LLVM IR
optimized
CodeGen
Binary
Opt
LLVM IR
Input &
Parser
Analysis &
Transform
Passes
LLVM IR
Output

LLVM



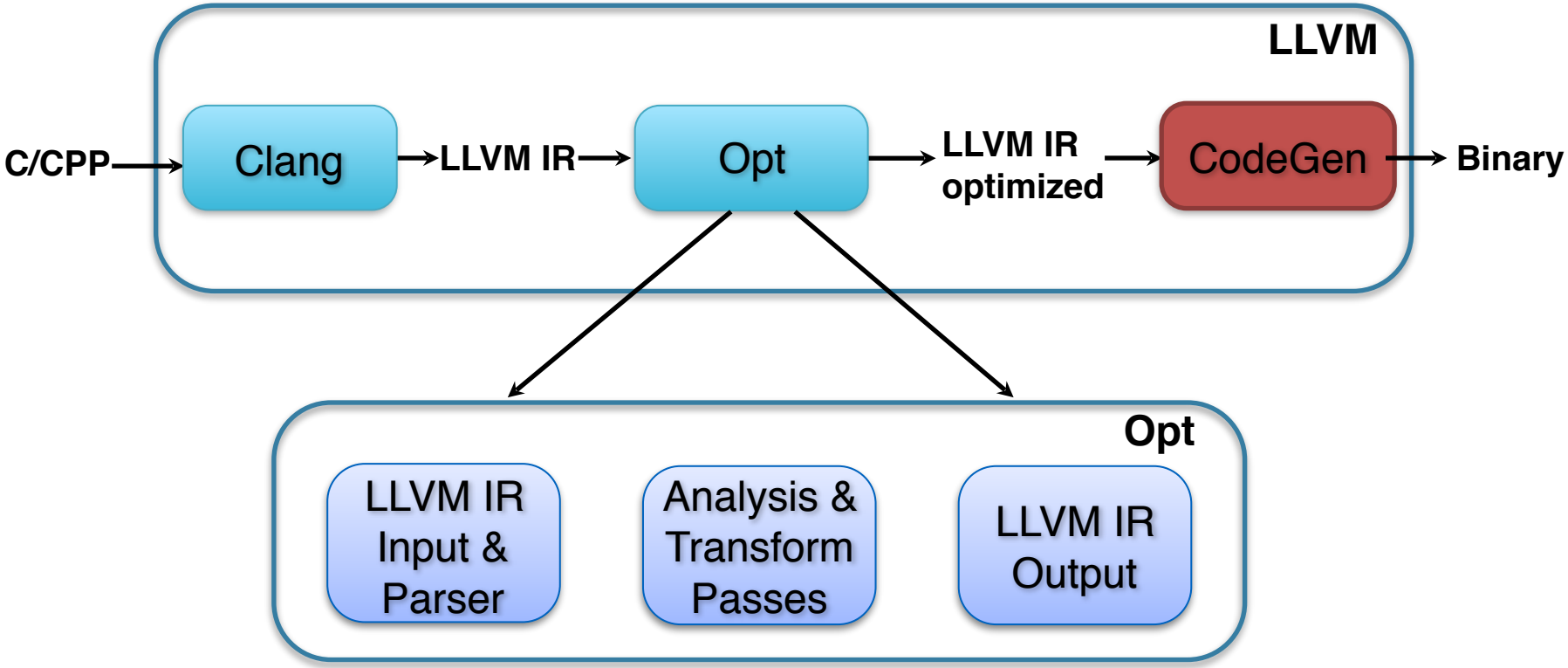
- Clang
  - compiler front end for the C, C++ programming languages
  - parse, validate and diagnose errors in the C/C++ code
  - translates the code into LLVM Intermediate Representation (IR)

LLVM



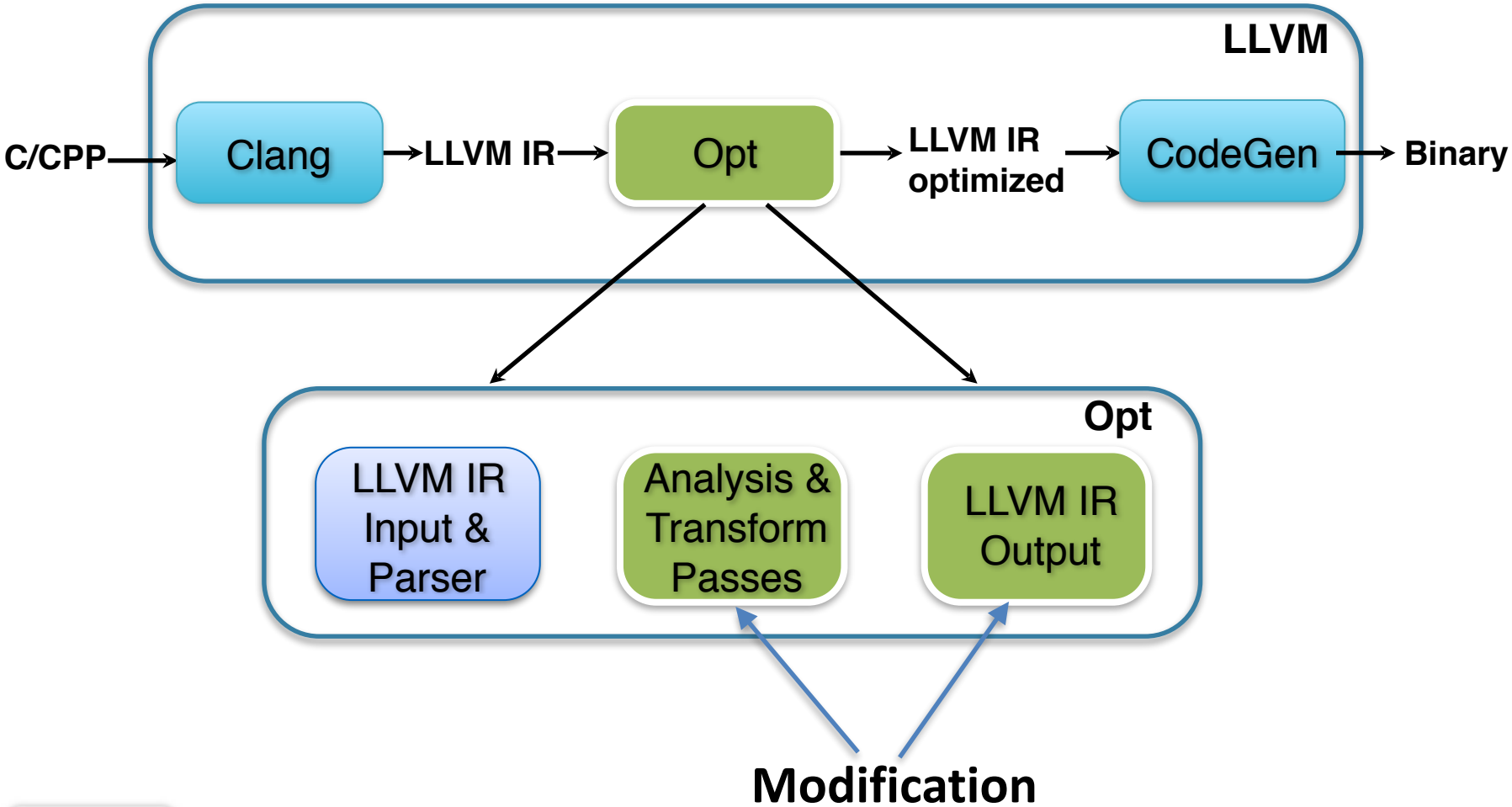
- Opt
  - LLVM optimizer and analyzer
  - take LLVM IR as input and parse LLVM IR.
  - the objective: improve the code quality

LLVM



- CodeGen
  - the compiler backend
  - translate LLVM IR to the machine code for a specified target

LLVM

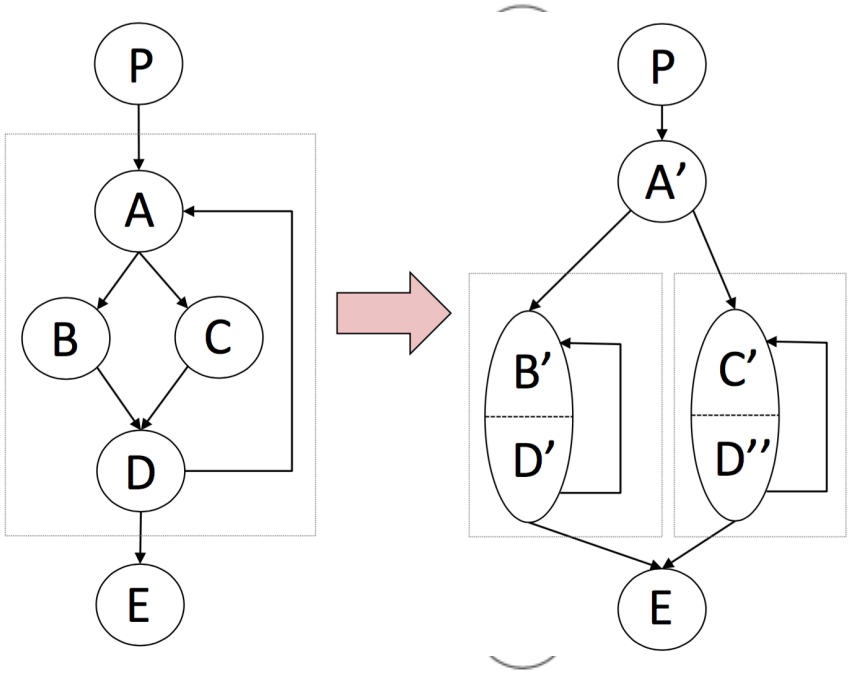

LLVM
C/CPP
Clang
LLVM IR
Opt
LLVM IR
optimized
CodeGen
Binary
Opt
LLVM IR
Input &
Parser
Analysis &
Transform
Passes
LLVM IR
Output
Modification

New storage type for flow information

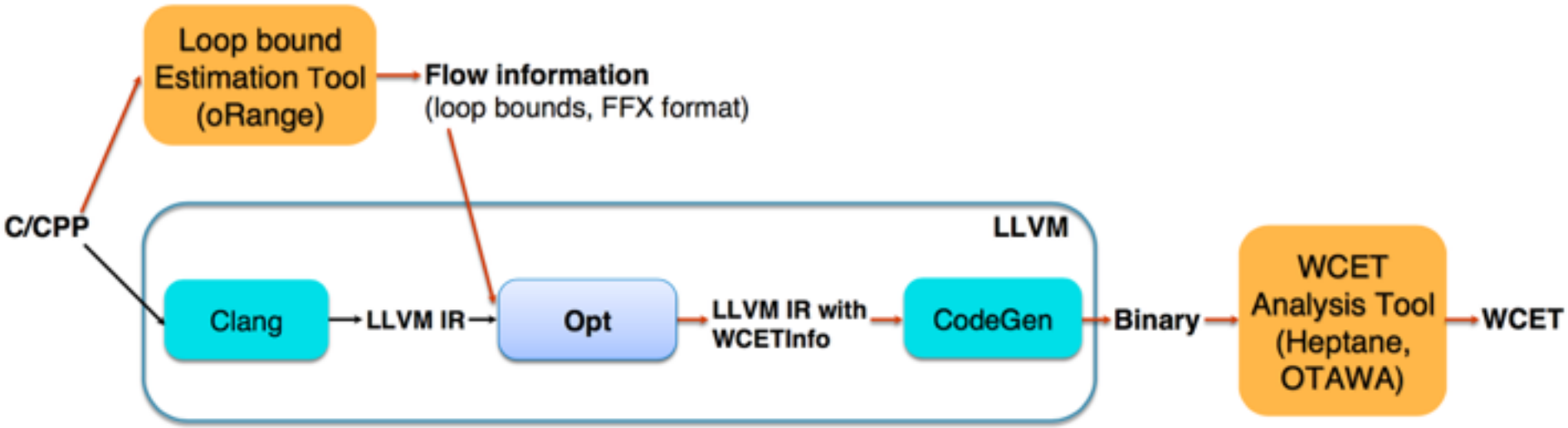
- Name: WCETInfo
- New type of information in LLVM
- Map loops (Loop object in LLVM) to corresponding loop bounds
  - For now, only loop bounds are supported

Transfer of flow information

- Preserve WCETInfo
  - loop bounds remain unchanged
  - instcombine: combine redundant instructions
- Update WCETInfo
  - loop bounds change
  - loop-unroll: unroll loops
- Add WCETInfo
  - a new loop is added
  - loop-unswitch: unswitch loops
- Delete WCETInfo



Process


Loop bound Estimation Tool (oRange)
Flow information
(loop bounds, FFX format)
C/CPP
LLVM
Clang
LLVM IR
Opt
LLVM IR with WCETInfo
CodeGen
Binary
WCET Analysis Tool (Heptane, OTAWA)
WCET

Experiments

- Experiments for traceability without vectorization
- Experiments for traceability with vectorization

Reason of the separation

- For without vectorization part: static methods
- For vectorization part: measurements
  - No support of SIMD instruction sets in supported WCET estimation tools

Experiments for traceability without vectorization

Objective

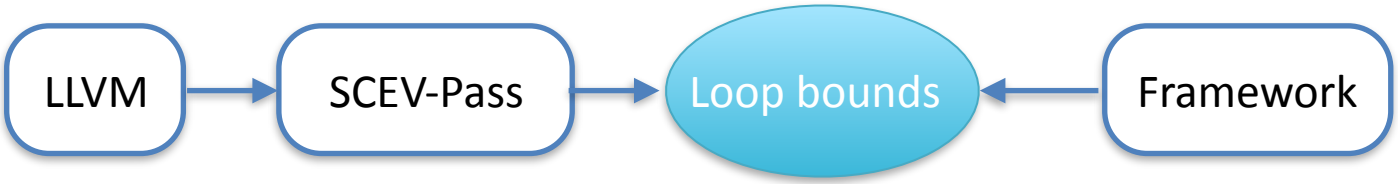
- Verify the implementation of our transformation framework in LLVM compiler infrastructure
- Find out the impact of compiler optimizations on estimated WCET
- Distinguish the individual impact among different optimizations

Benchmarks and target hardware

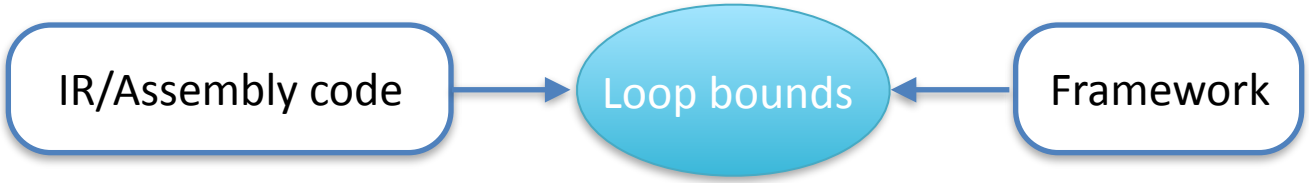
- Standardized set of WCET benchmarks from Mälardalen University
- ILP solver: CPLEX V12.5
- Hardware:
  - 32-bit MIPS processor
  - L1, L2 cache and Memory
- LLVM version 3.4
- Option -O3 for optimized codes
- WCET estimation tool: Heptane

Verify the correctness

- Verify all loop bounds in optimized binary
  - Automatic way: compare with those available using the *scalar-evolution* analysis pass

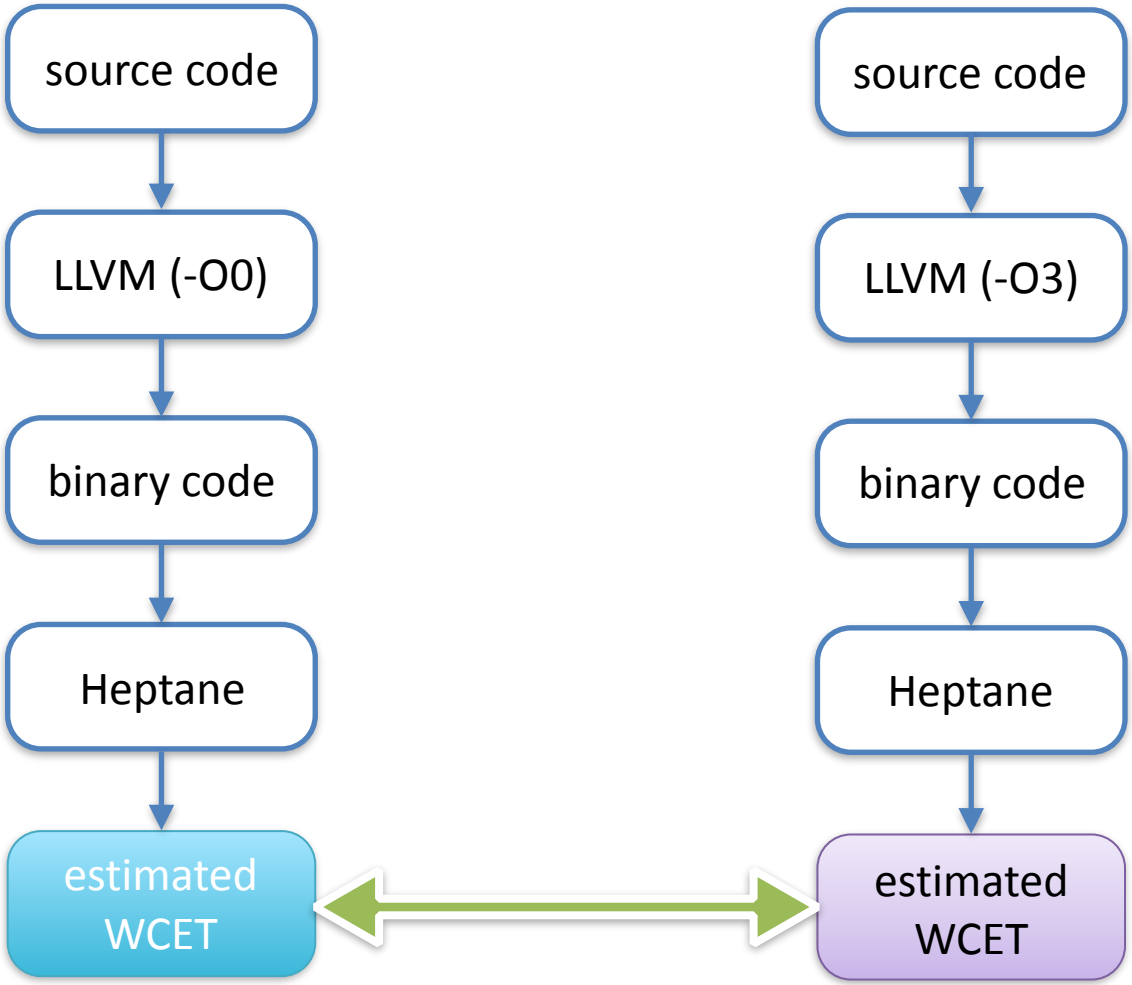


  - Manual way: compare with the loop bounds from the LLVM IR and assembly code



  - Proved by both methods

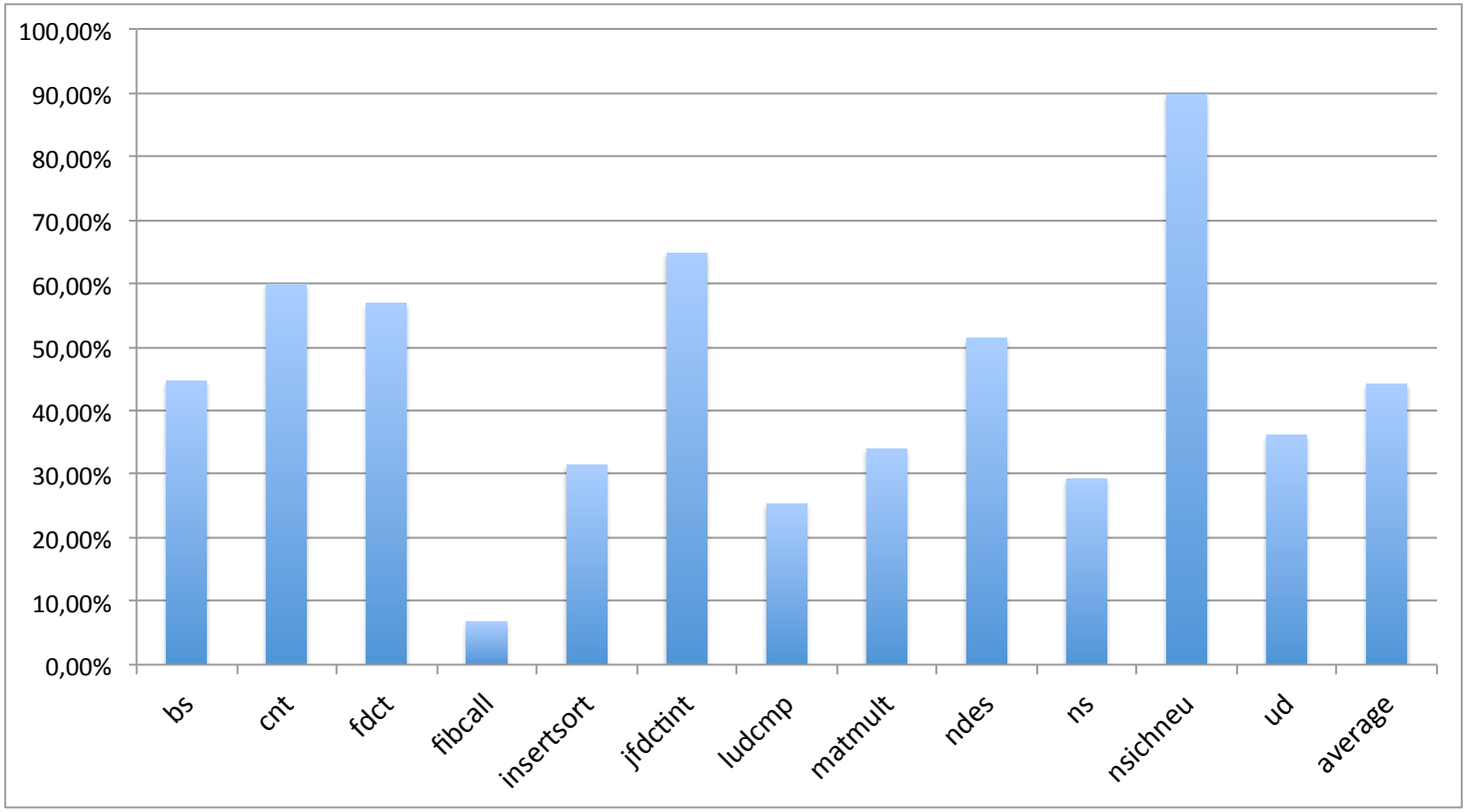
Experimental result



-O0 option: no optimization

-O3 option: enable most optimizations

Experimental result



- Y-axis: WCET (-O3), normalized with respect to the WCET (-O0)
- -O3 reduces estimated WCET: 55% in average

Impact of optimizations

- Individual impact of optimizations (1-off):
  - Inlining:
    - replace a function call with the body of the called function
    - disabling it: estimated WCET increase on many benchmarks (e.g. bs, fibcall, ndes, …)
- Combined impact of optimizations (2-off):
  - Sroa and Mem2reg: overlapping effects
    - Mem2reg: replace costly memory accesses by much faster register uses
    - Sroa: identify promotable elements of an aggregate alloca, and promote them to registers

Experiments for traceability with vectorization

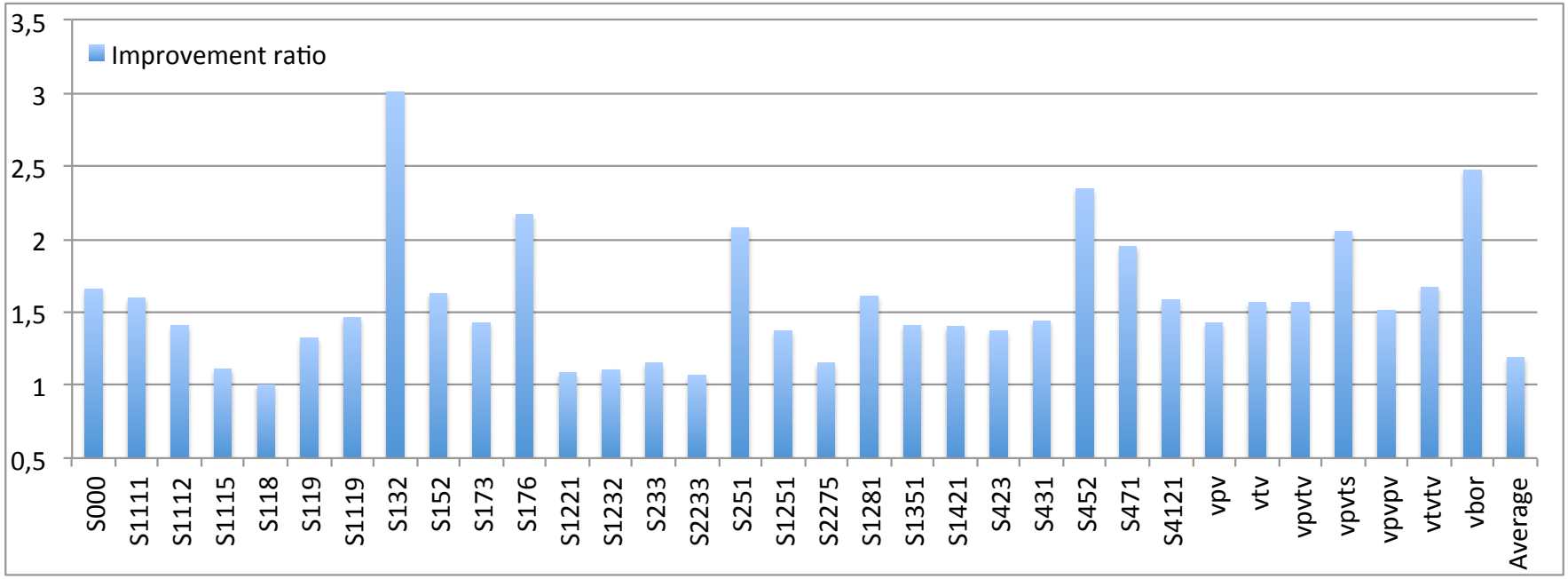
Measurements

- Measurements on actual hardware to collect real execution times
  - Restricted to single-path programs
  - Remove benchmarks with indirect addressing (e.g. a[b[i]]=…)
  - Completely unloaded system
  - Run each benchmarks five times and observed execution times are stable
  - Use the highest observed value

Benchmarks and Environment

- Benchmark suites
  - Test Suite for Vectorizing Compilers (TSVC): 112 single-path loops
  - Gcc-loops: 15 single-path loops
- Environment
  - ARM: Panda Board, ARMv7 processor, NEON 128-bit vectors, Ubuntu 12.04.5, LLVM3.3
  - Intel: Intel Core i7, SSE4.2 128-bit vectors, MAC OS X, LLVM3.3, Turbo Boost disabled

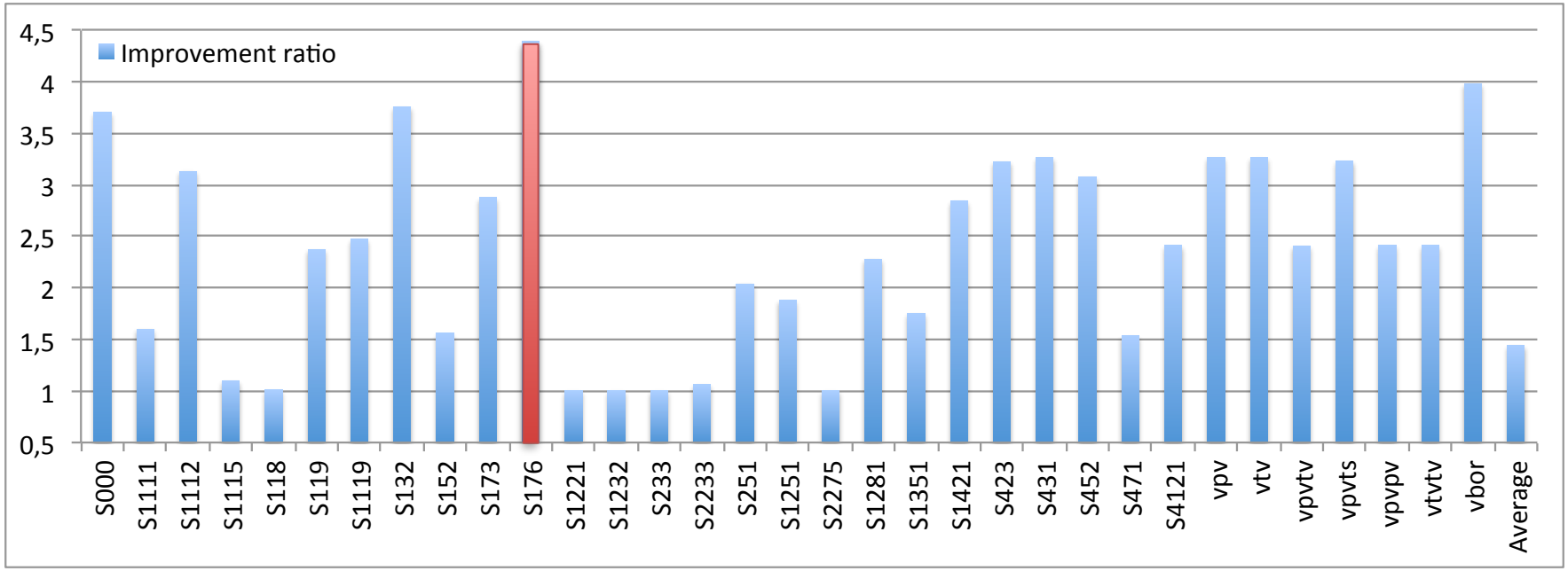
TSVC & ARM



Impact of vectorization on WCET (TSVC & ARM)

- Y-axis: improvement ratio compared with vectorization disabled
- Average improvement ratio: 1.19
- Theoretical improvement ratio: 4. Around 2 in most cases because of cache misses and available bandwidth

TSVC & Intel



Impact of vectorization on WCET (TSVC & Intel)

- Y-axis: improvement ratio compared with vectorization disabled
- Average improvement ratio: 1.44
- For benchmark s176: perfectly vectorized and with loop unrolling

Summary of contributions

- Transformation framework
- Implementation within LLVM compiler infrastructure
- Trace loop bounds within compiler optimizations
- Derive smaller and safe WCET result with compiler optimizations

Future work

- A more general and powerful framework (automatic generation of transformation rule set for each optimization)
- Provide support for SIMD instruction set in WCET estimation tools
- Support infeasible path and contextual information