A Case for Shipping ALL Software Using Virtual Instruction Sets: The ALLVM Project

Vikram Adve and Will Dietz

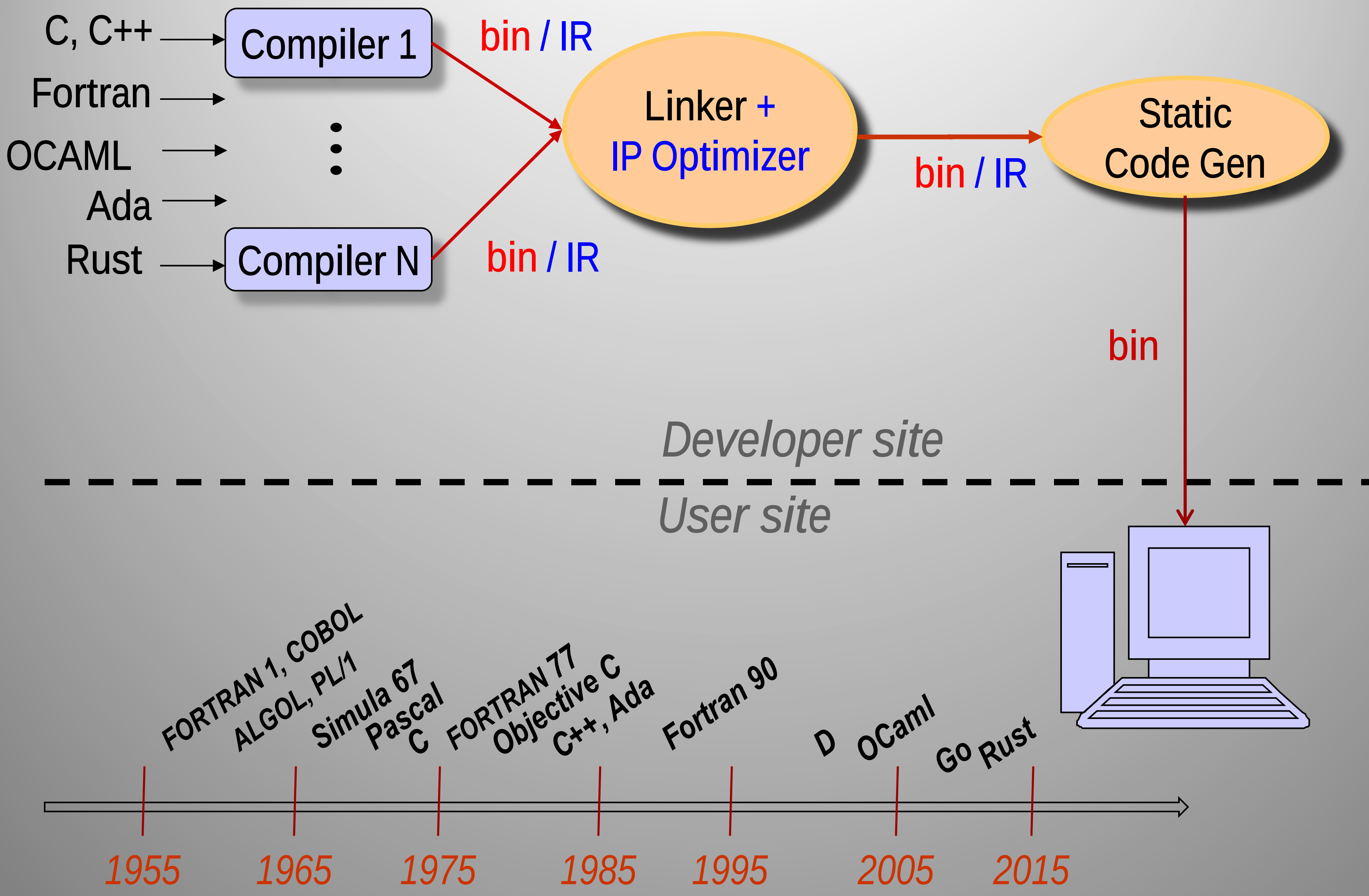
With: Sean Bartell, Tom Chen, Sandeep Dasgupta, Theodoros Kasampalis, Maria Kotsifakou and Hashim Sharif

University of Illinois at Urbana-Champaign

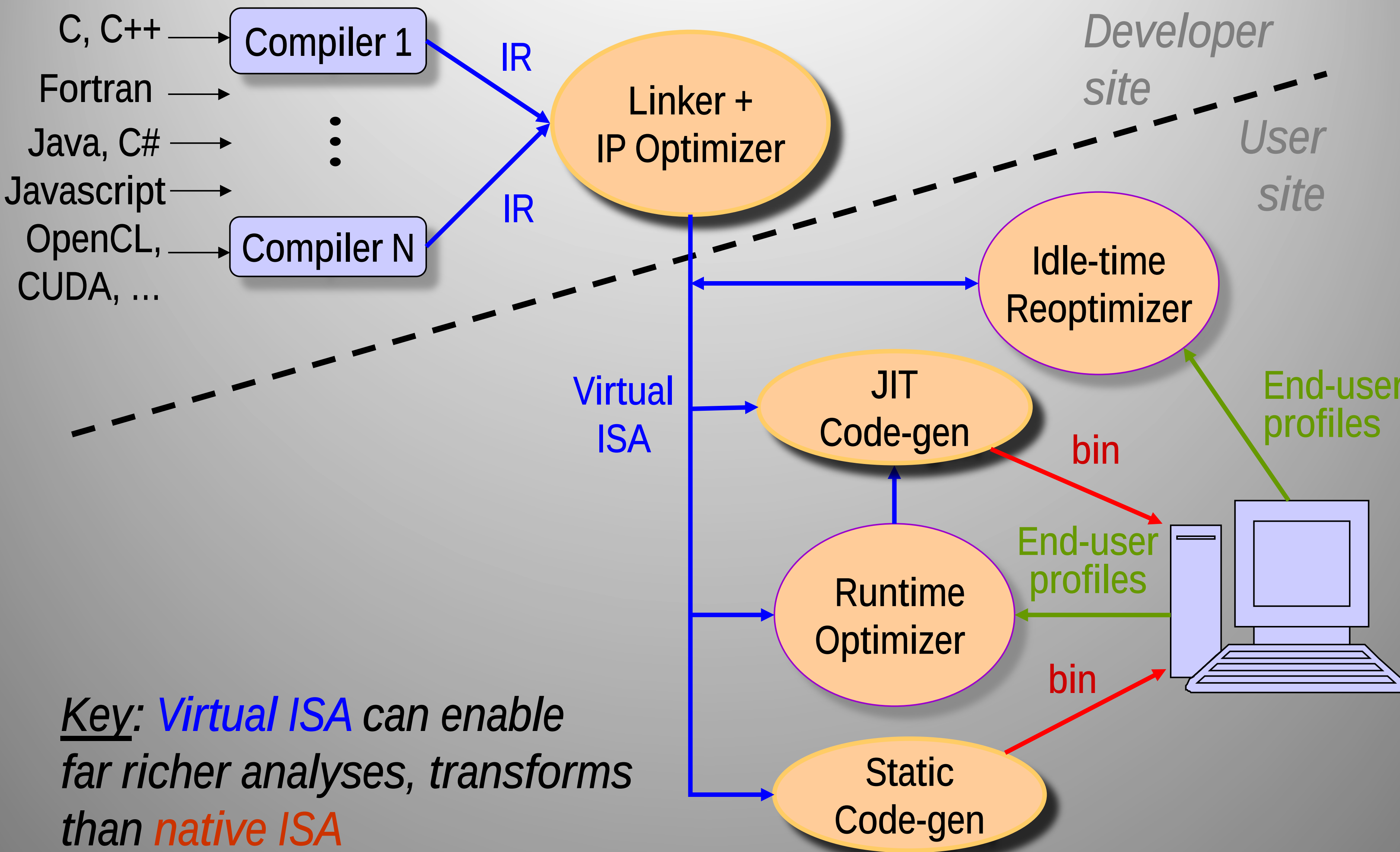
Alumni: Chris Lattner, John Criswell, Swarup Sahoo

Supported by: ONR, NSF, SRC, DARPA

Compilation Model for Static Languages

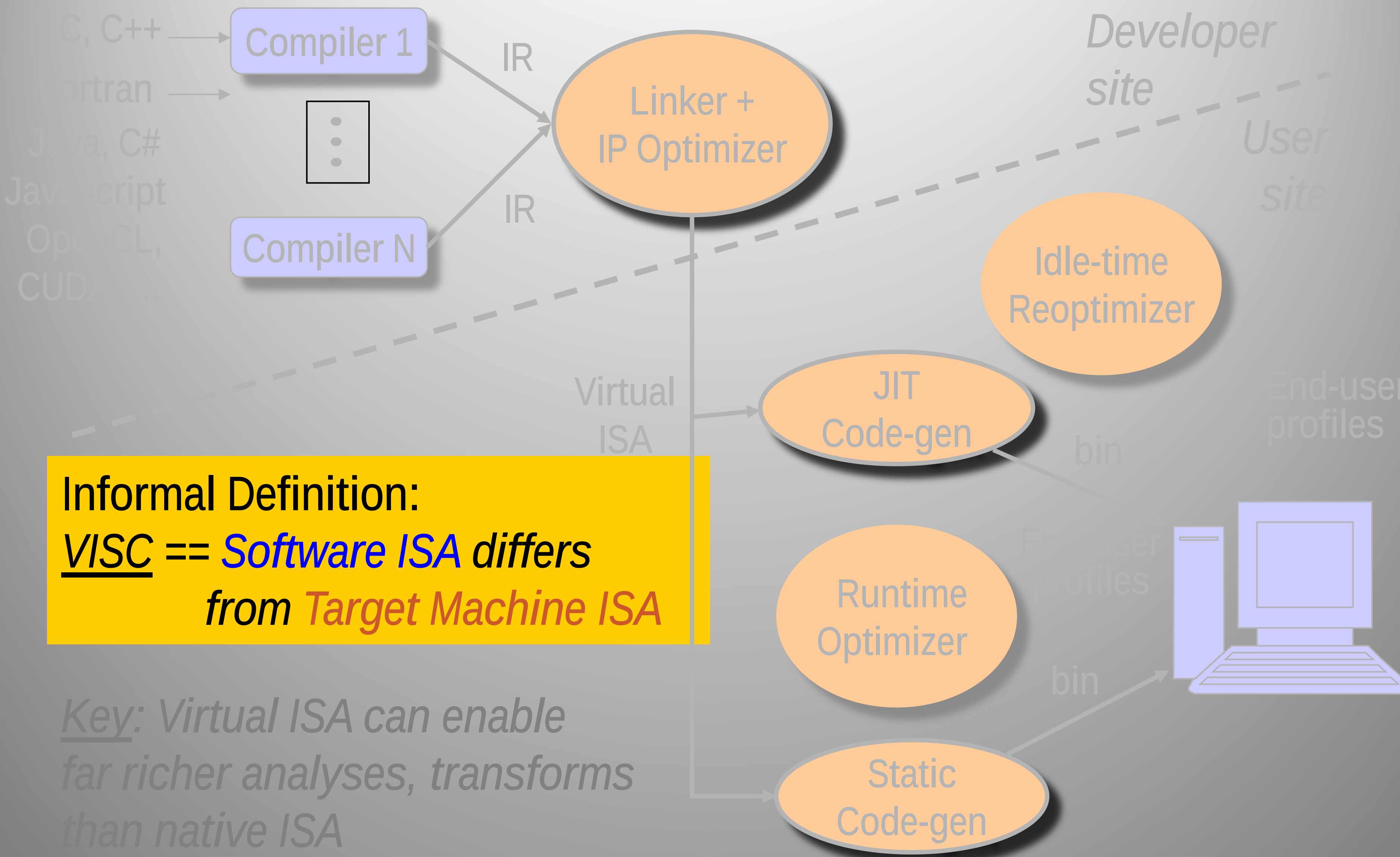


Virtual Instruction Set Computing



Key: Virtual ISA can enable far richer analyses, transforms than native ISA

Virtual Instruction Set Computing



Informal Definition:

VISC == *Software ISA differs from Target Machine ISA*

Key: Virtual ISA can enable far richer analyses, transforms than native ISA

Popular Native Code Systems (Not VISC)

"VISC" == Ship code as Virtual ISA (e.g,. JVM, PTX)

Native code is pervasive for two broad classes of software

High performance software is *largely* shipped as native code	
HPC applications	
Media, Gaming, Finance, CAD, …	
Web browsers	
Database systems	
Libraries galore	

Static Compilation is NOT Enough

Modern software architectures

- Install-time configurations, software environments
- User-installed extensions, dynamically loaded libraries, layering

Modern hardware architectures

- Diverse vector hardware, GPUs, accelerators in SoCs

Modern security challenges due to untrusted code

- Browser extensions, mobile app markets, BYOD

Need rich analyses and transformations on end-user systems

Proposal

All future software should "ship" using Virtual ISAs.

NOTE: Different systems can use different Virtual ISAs.

- The security benefits are strong
- There are no inherent performance penalties (and novel performance benefits are possible)
- It is technically feasible and commercially acceptable

Myth: Virtual ISA Threatens IP

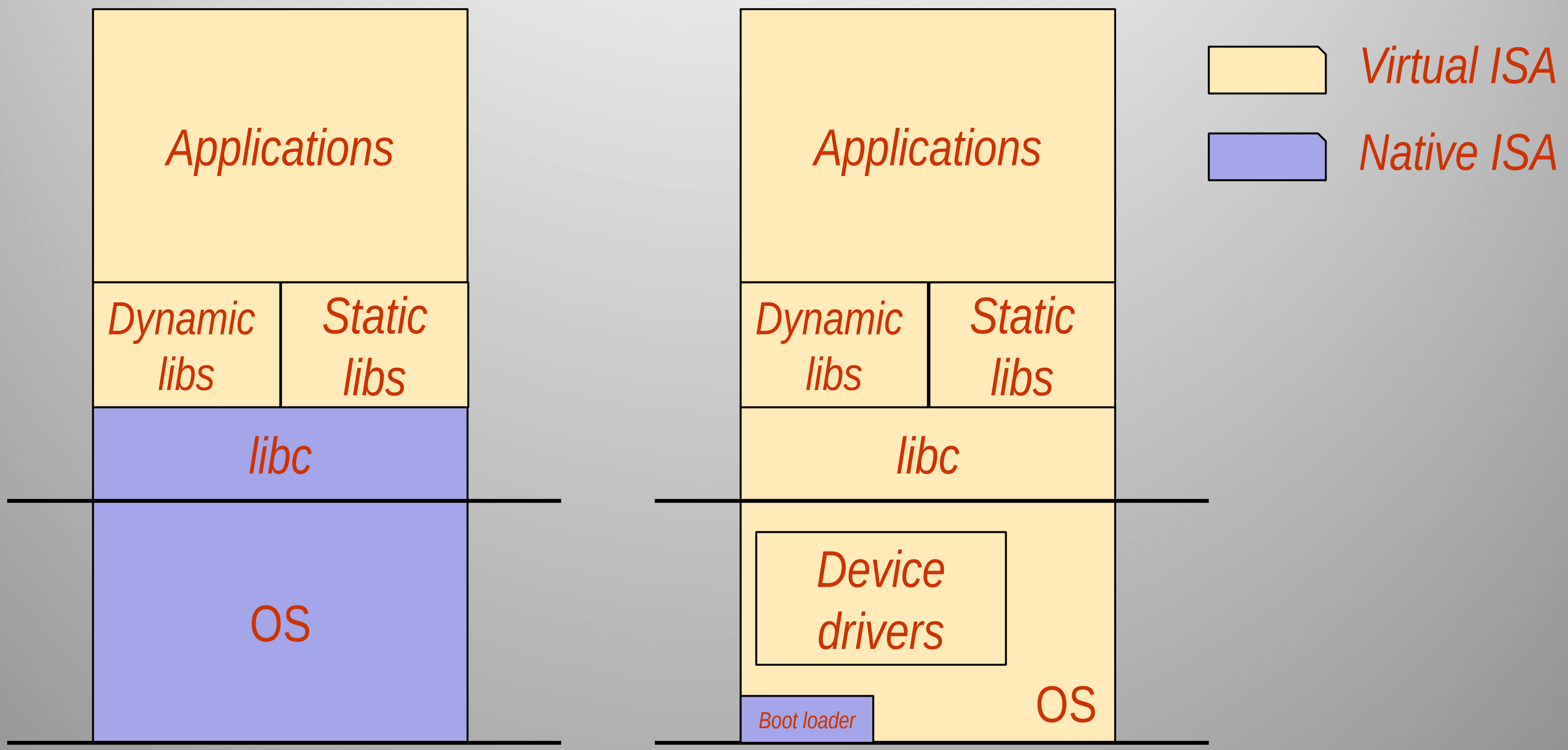
Fact: *Binary code can be* *reverse engineered* *effectively* using interactive tools + *manual* analysis

- Better Solution #1: Encryption + Code Signing
- Better Solution #2: Obfuscation tools (must not interfere with program analyses)

ALLVM: Ship *All* Software as Virtual ISAs

Will Dietz and V. Adve

Key: Virtual ISAs enable far richer analyses, transforms than native ISA



Userspace only

OS <u>and</u> Userspace

LLVM Virtual Instruction Set and IR

```
/* C Source Code */
int SumArray(int A[], int Num)
{
  int i, sum = 0;
  for (i = 0; i < Num; ++i)
    sum += A[i];
  return sum;
}
```

- Simple, 3-address IR
- Architecture-neutral
- Language-neutral
- Explicit CFG
- Always in SSA form
- Typed memory, regs

```
;; LLVM Code
int %SumArray(int* %A, int %Num)
{
bb1:
    %cond = icmp sgt i32 %Num, 0
    br i1 %cond, label %bb2, label %bb3
bb2:
    %sum0 = phi  i32 [%t10, %bb2], [0, %bb1]
    %iv   = phi  i64 [%inc, %bb2], [0, %bb1]
    %t2   = getelementptr inbounds i32* %A, i64 %t7
    %t3   = load  i32* %t2, align 4
    %t4   = add nsw  i32 %t3, %sum0
    %inc  = add nuw  i64 %iv, 1
    %t5   = trunc i64 %iv to i32
    %exitcond = icmp eq i32 %inc, %Num
    br i1 %exitcond, label %bb3, label %bb2
bb3:
    %sum1  = phi  i32 [0, %bb1], [%t4, %bb2]
    ret int %sum1
}
```

LLVM enables sophisticated program analyses and transformations

Why LLVM IR for ALLVM? (1 of 2)

1. Fully executable virtual ISA

- Language-neutral; hardware-neutral; and a *rich* IR
- Extensive production-quality infrastructure and tools
- Widely used: Apple, Google, Intel, QCOM, ARM, …
- Numerous front-ends: C, C++, (Fortran), .NET, Swift, Python, Ruby, Haskell, …

Available at: *llvm.org*
First release: October 2003

Why LLVM IR for ALLVM? (2 of 2)

2. Emerging adoption as a Virtual ISA

	Compile-time	Link-time	Install-time	Load/Run-time	Idle-time	
Apple, Sony, Intel, QCOM, …	✔	✔				*Static compilers*
(Apple) tvOS, watchOS, iOS	✔		✔			
Ma						*VISC ystems*
Op						
Re						
(Google) PNaCl	✔	?		✔		

SHIP

"For iOS™ apps, bitcode is the default, but optional. For watchOS™ and tvOS™ apps, bitcode is required."
-- iOS App Distribution Guide, Apple

But Many Unanswered Questions

Not enough research on benefits of a Virtual ISA

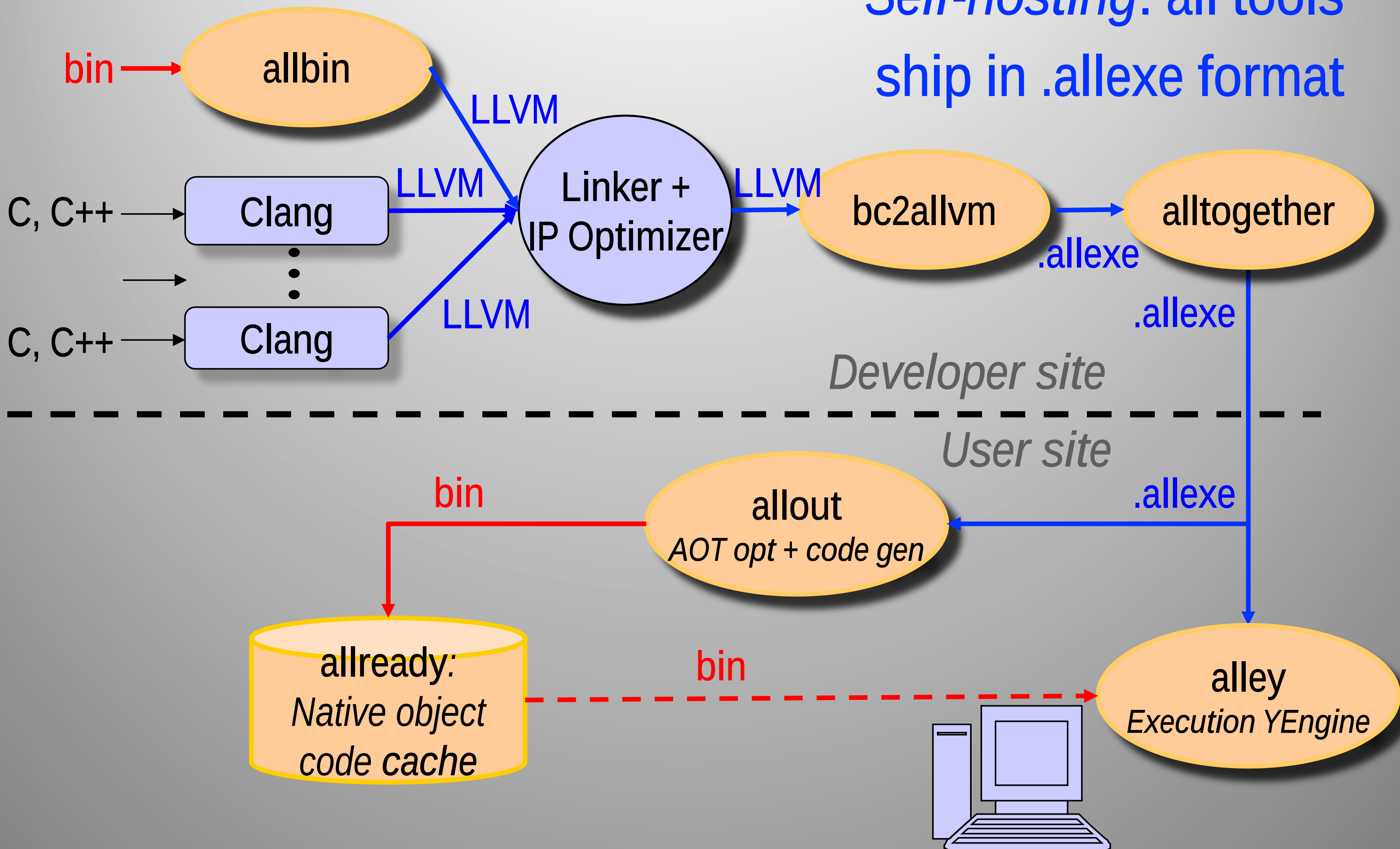
For software in static languages (C, C++, Fortran, OpenMP, ...)

Uses to date are limited, ad hoc, and haphazard

- What are the benefits for performance?
- What are the benefits for security?
- What are the benefits for software reliability?

ALLVM Toolchain

Self-hosting: all tools ship in .allexe format



ALLVM Status

Self-bootstrap: Clang (C++) : bash + cmake → make + clang →
bc2allvm → alltogether → allout / cache → alley

Substantial userspace software, tools work in ALLVM:

- xterm, libX11, vim, spidermonkey
- openssl, openssh
- (apache) httpd, nginx, redis, memcached, postgresql
- subversion, git
- binutils, coreutils, bash, zsh, tcsh
- lua, perl, python, ocaml (the C-based bits anyway)

Substantial capabilities for userspace:

- Runs on top of existing Linux OS, or in Docker
- Binary cache: Local and remote (trusted)
- Nix package manager: Atomic software upgrades

Adding more packages is “easy” if build system is somewhat sane

ALLVM Research Goals: What are the Benefits?

Security

- Secure Virtual Architecture (*John Criswell*; PhD '14)

  Runner-up, ACM Doctoral Dissertation Award

ALLVM Research Goals: What are the Benefits?

Security

- Secure Virtual Architecture (*John Criswell*; PhD '14)

Software Reliability

- Automated fault localization (*Swarup Sahoo*; PhD '12)
- Distributed system fault diagnosis (*Sean Bartell*)
- Verified codegen: Increasing trust in shipping virtual ISAs (*Theodoros Kasampalis*)

ALLVM Research Goals: What are the Benefits?

Security

- Secure Virtual Architecture (*John Criswell*; PhD '14)

Software Reliability

- Automated fault localization (*Swarup Sahoo*; PhD '12)
- Distributed system fault diagnosis (*Sean Bartell*)
- Verified codegen: Increasing trust in shipping virtual ISAs (*Theodoros Kasampalis*)

Performance

- Whole-*system* optimization; deduplication (*Will Dietz*)
- Software specialization and debloating (*Hashim Sharif*)
- *allready*: Binary-to-LLVM (*Sandeep Dasgupta*)
- Autotuning: install-time search (*Yishen Chen*)
- HPVM: Heterogeneous parallel systems (*Maria Kotsifakou*)

Outline: Applications of ALLVM

- Code deduplication with *software multiplexing*
- Debloating via program customization
- Binary translation to LLVM IR

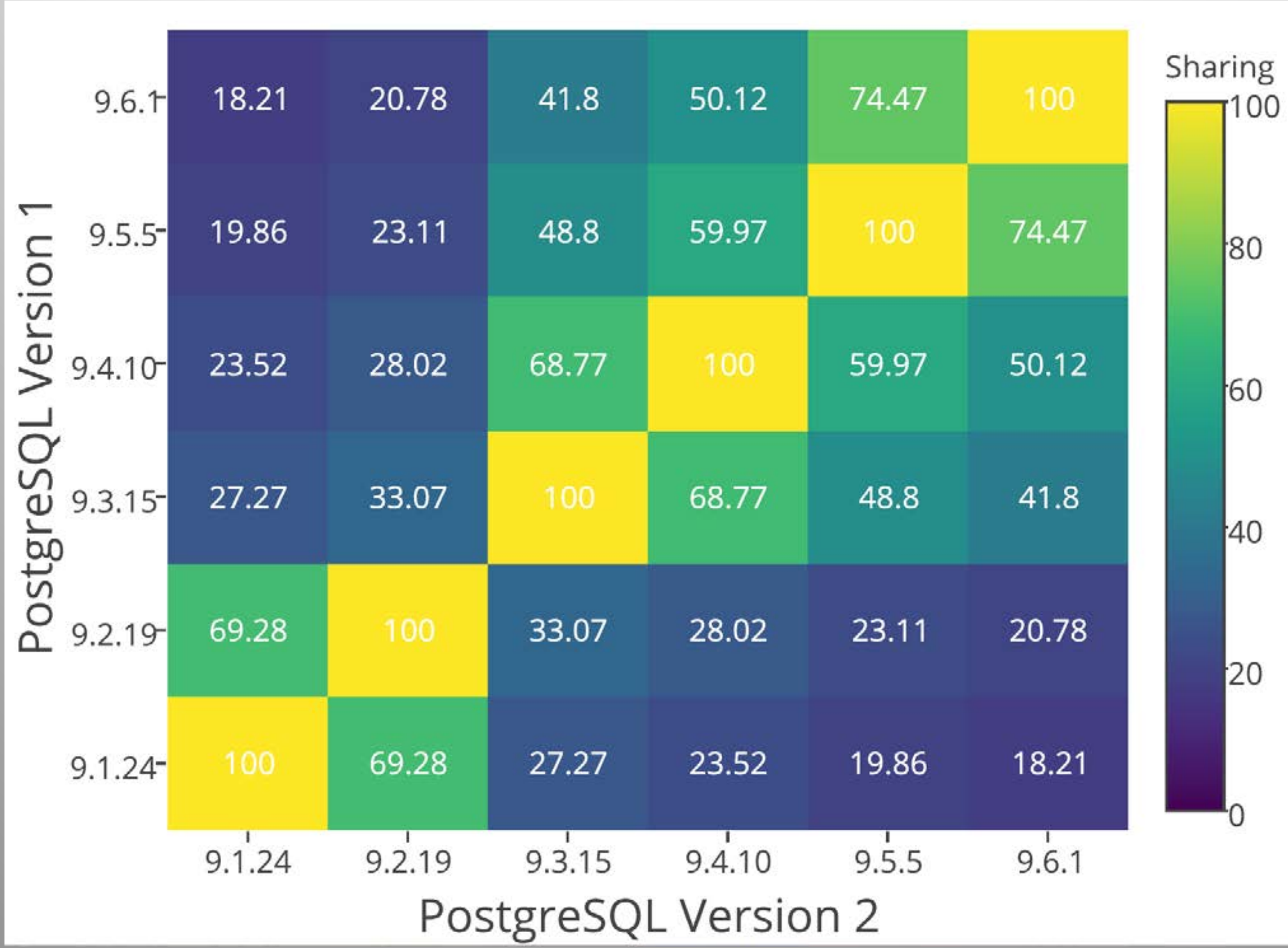
Sources of Code Duplication

- Duplicated libraries across applications
- Multiple versions of libraries or applications on a system
- Duplicated functions or code fragments within / across applications

Software multiplexing is a framework
to address all three issues
(current system addresses first two)

Code Duplication Across Software Versions

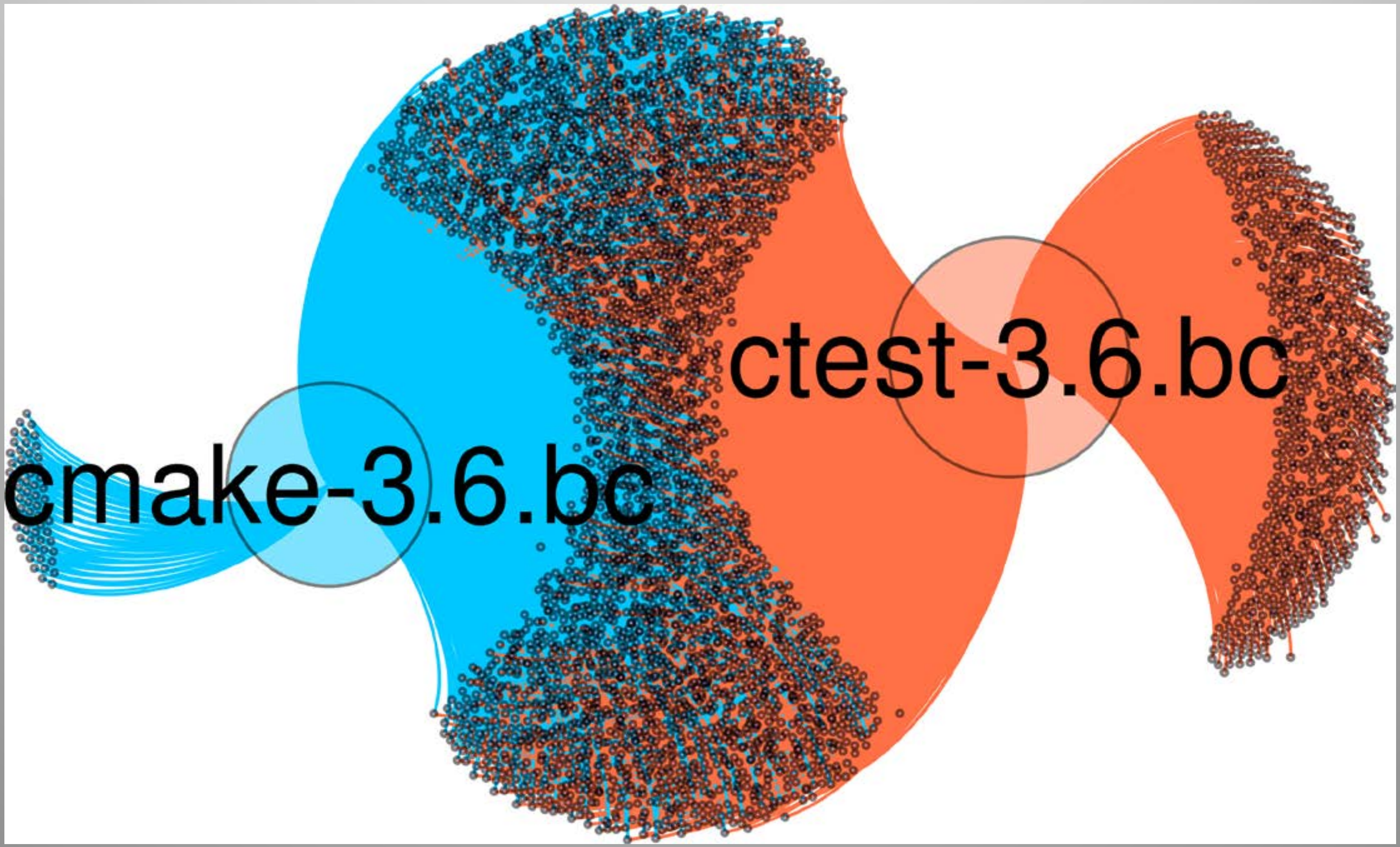
Multiple versions of a tool or library often co-exist on the same machine
=> extensive duplication



Code Duplication Across Programs in a Package

Multiple tools in a package often share extensive code

Nodes are functions (as hash value)
Edges mark equivalence; colored regions are dense with edges

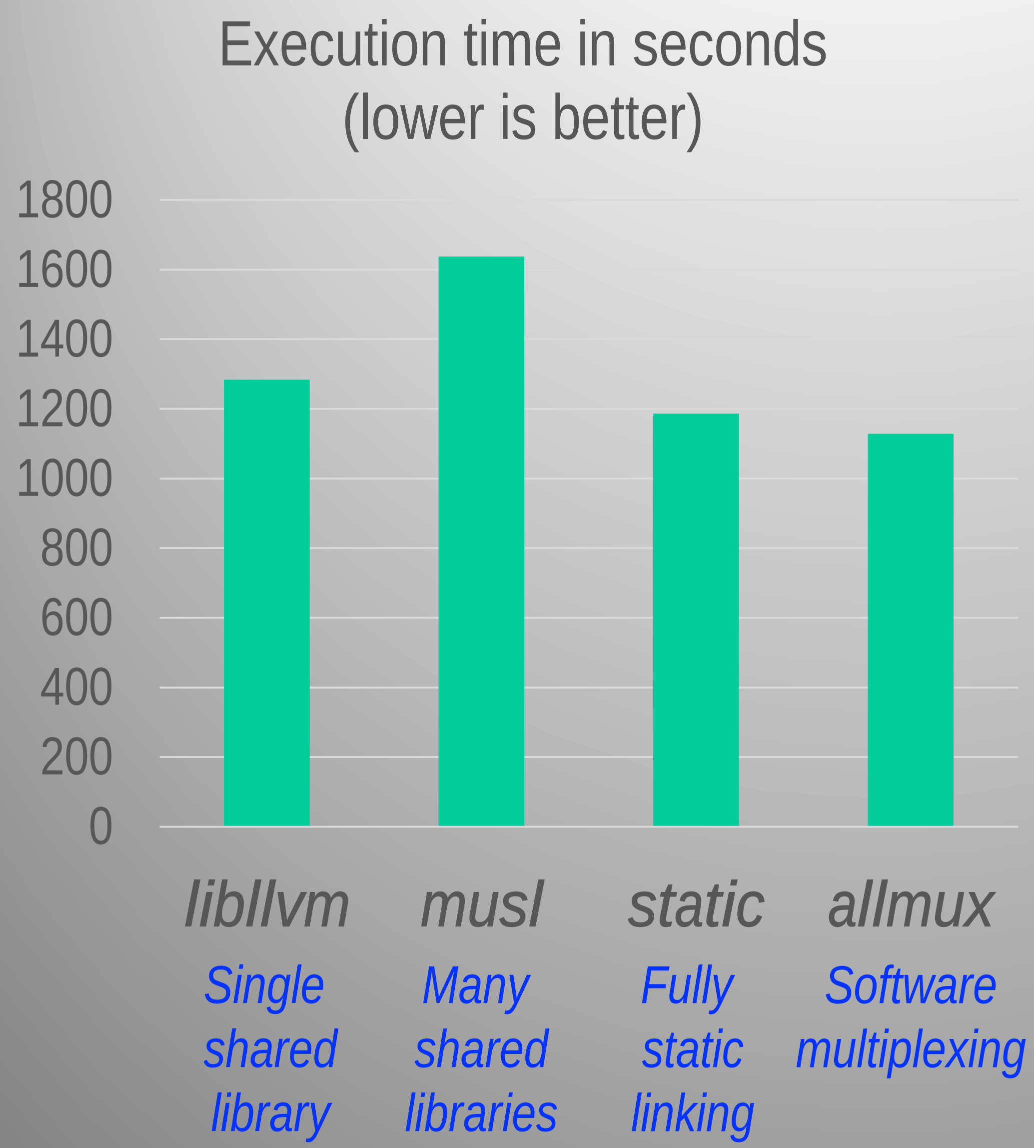


Example: Code Deduplication with *Allmux*

Size and performance of LLVM linux-x86-64 software release

Build config	Binaries size	Libraries size	Total size	Performance	Startup
Static	590M	2.1M	592M	Better	Fast
Shared (libllvm)	231M	38M	269M	Worse	Slow
Shared (separate libs)	11M	93M	104M	Worst	Slowest
Allmux	85M	0M	85M	Best	Fastest

ALLVM Quasi-static Linking



Total time for building Clang system with four LLVM versions:

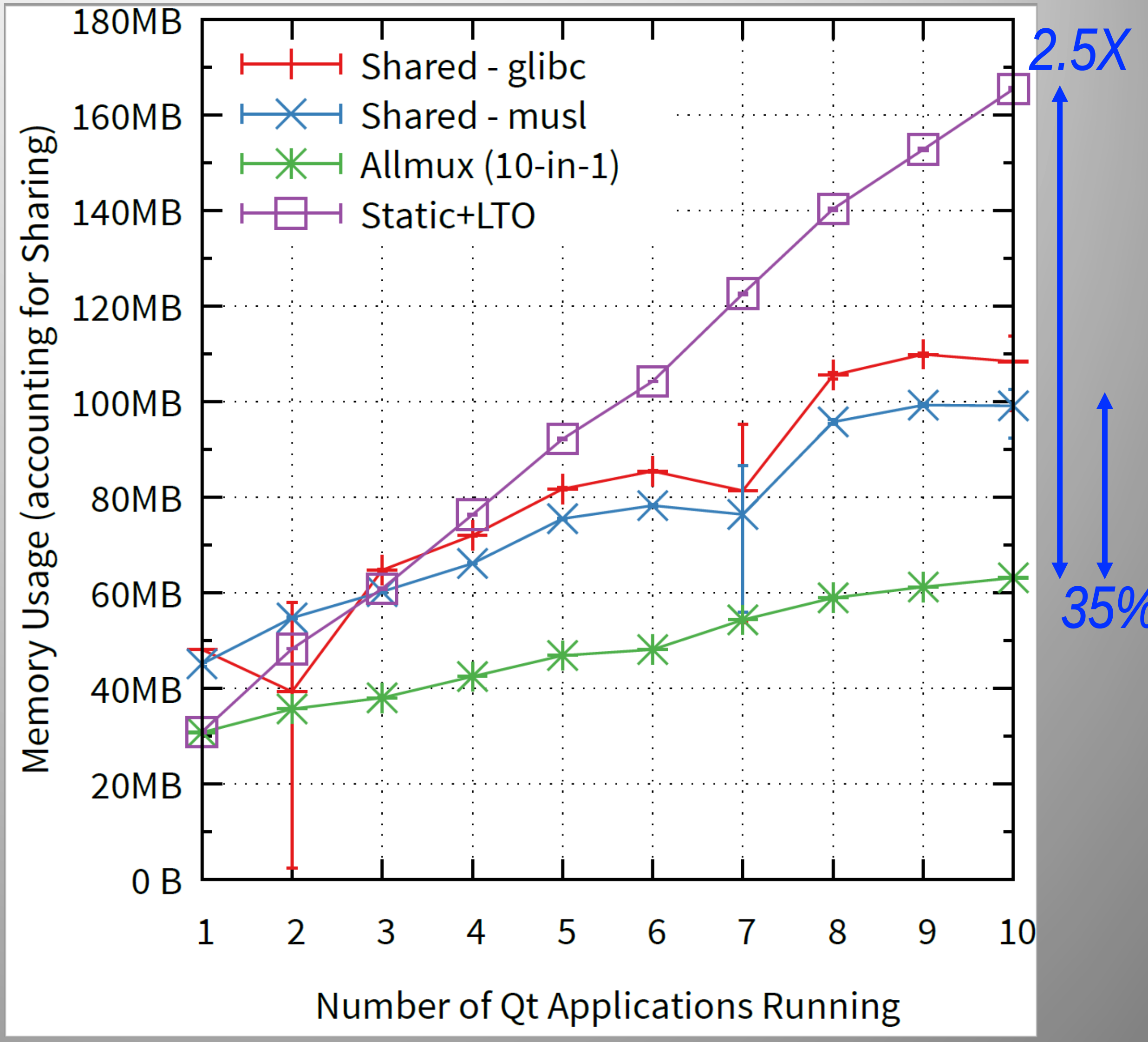
Allmux *version faster than* dynamically linked versions because lower startup cost

Memory Usage with Shared Libraries

E.g., 1-10 apps that use Qt toolkit

Allmux uses 2.5x less RAM vs. static linking; 35% less than dynamic

Allmux performance is much better than dynamic linking; comparable to static



What's the Secret? (1 of 2)

1) Software Multiplexing: *N* pgms + *K* libs → *1* pgm + *K libs*

Exposes duplicated code between programs, libs

```
int main(int argc, char* argv[], char* envp[]) {
    If (! strcmp(argv[0], "program-name1") main1(argc, …);
    If (! strcmp(argv[0], "program-name2") main2(argc, …);
    If (! strcmp(argv[0], "program-name3") main3(argc, …);
    If (! strcmp(argv[0], "program-name4") main4(argc, …);
    …
}
```

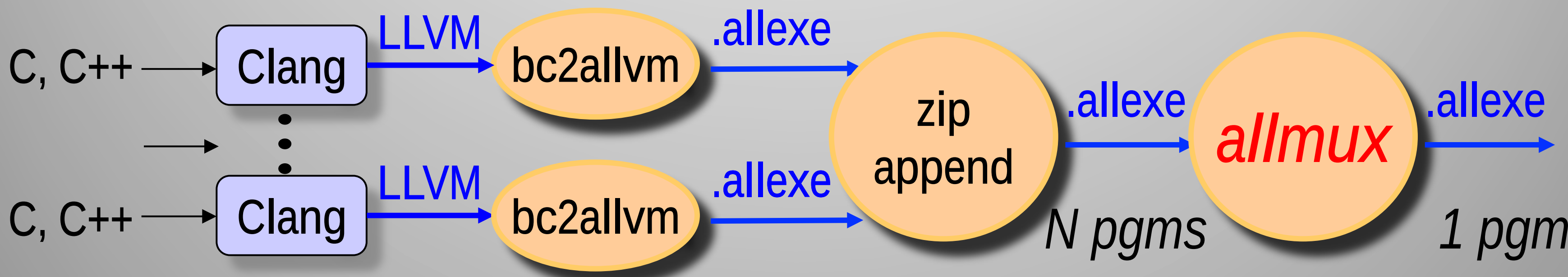
"Designed in" by a few packages, e.g., GCC

Affects the build system heavily → hard to add manually today

What's the Secret? (1 of 2)

1) Software Multiplexing: *N* pgms + *K* libs → *1* pgm + *K libs*

Exposes duplicated code between programs, libs

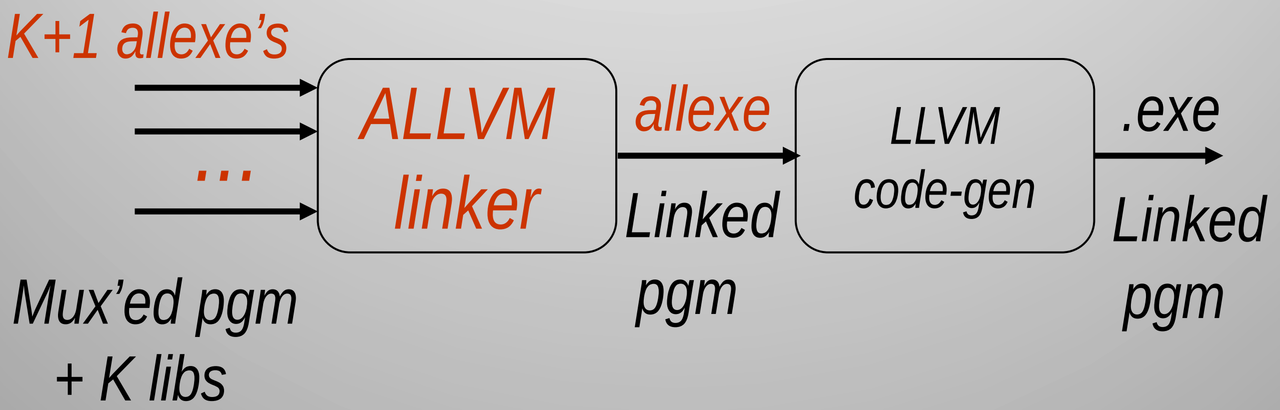


Key: IR-level compiler pass adds multiplexing

What's the Secret? (2 of 2)

2) Bitcode for *all* components including dynamic libraries enables linking before code generation

➔ static linking *without* rewriting build system!



Next Steps on Deduplication with ALLMUX

- Identify equivalent functions #1: structural equivalence
- Identify equivalent functions #2: semantic equivalence
- Identify equivalent fragments: perhaps by hashing

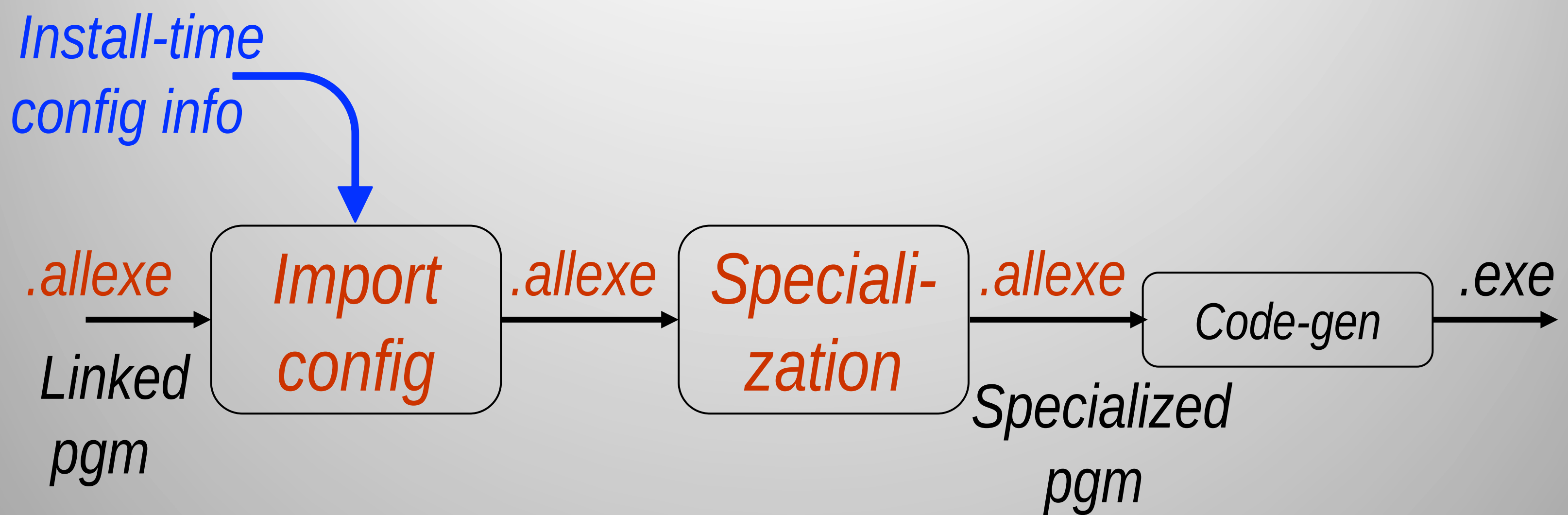
Towards a Bitcode Database

The one repository to rule them all!

Outline: Applications of ALLVM

- Code deduplication with *software multiplexing*
- Debloating via customization to a configuration
- Binary translation to LLVM IR

Configuration-based Slimming



- Customize for user-defined program configuration
  - Generate specialized binaries
  - Reducing code bloat as a result of specialization

Specialization transforms

1. Identify code that parses input configuration

2. Fully unroll only the loop(s) that parse inputs

3. Mark config variables that hold constant values

4. Aggressive interprocedural const. propagation for marked variables only

5. Aggressive constant evaluation, specialization

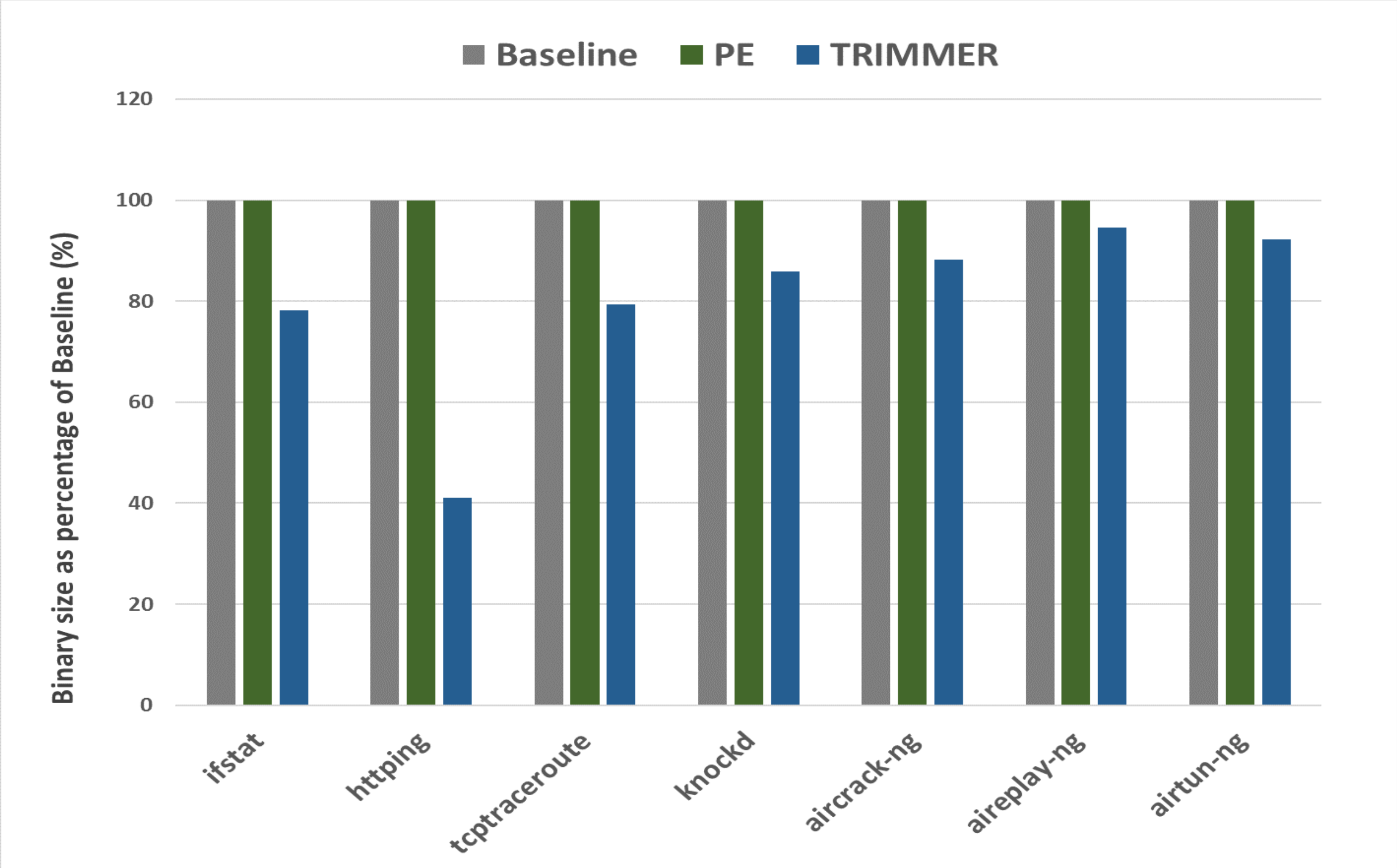
Experiments

- Goal: Compare against existing state-of-the-art Partial Evaluation tool (Occam)

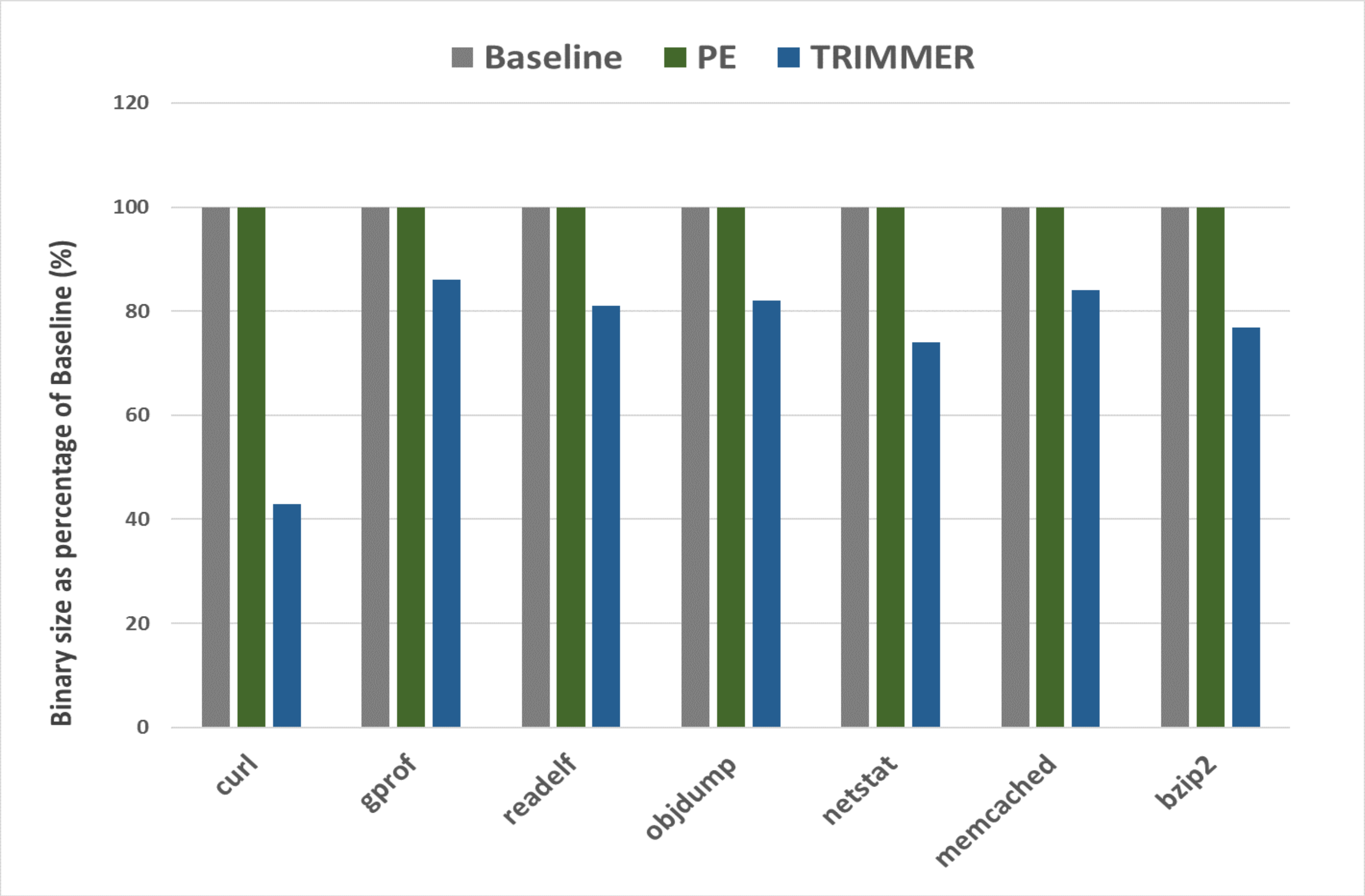
- Benchmarks:
  - 7 OpenWRT programs: *optimized for embedded systems*
  - 7 Commonly used Linux programs
  - Yices – SMT Solver

- 18.35% Geom. mean code reduction across 14 programs

OpenWRT programs



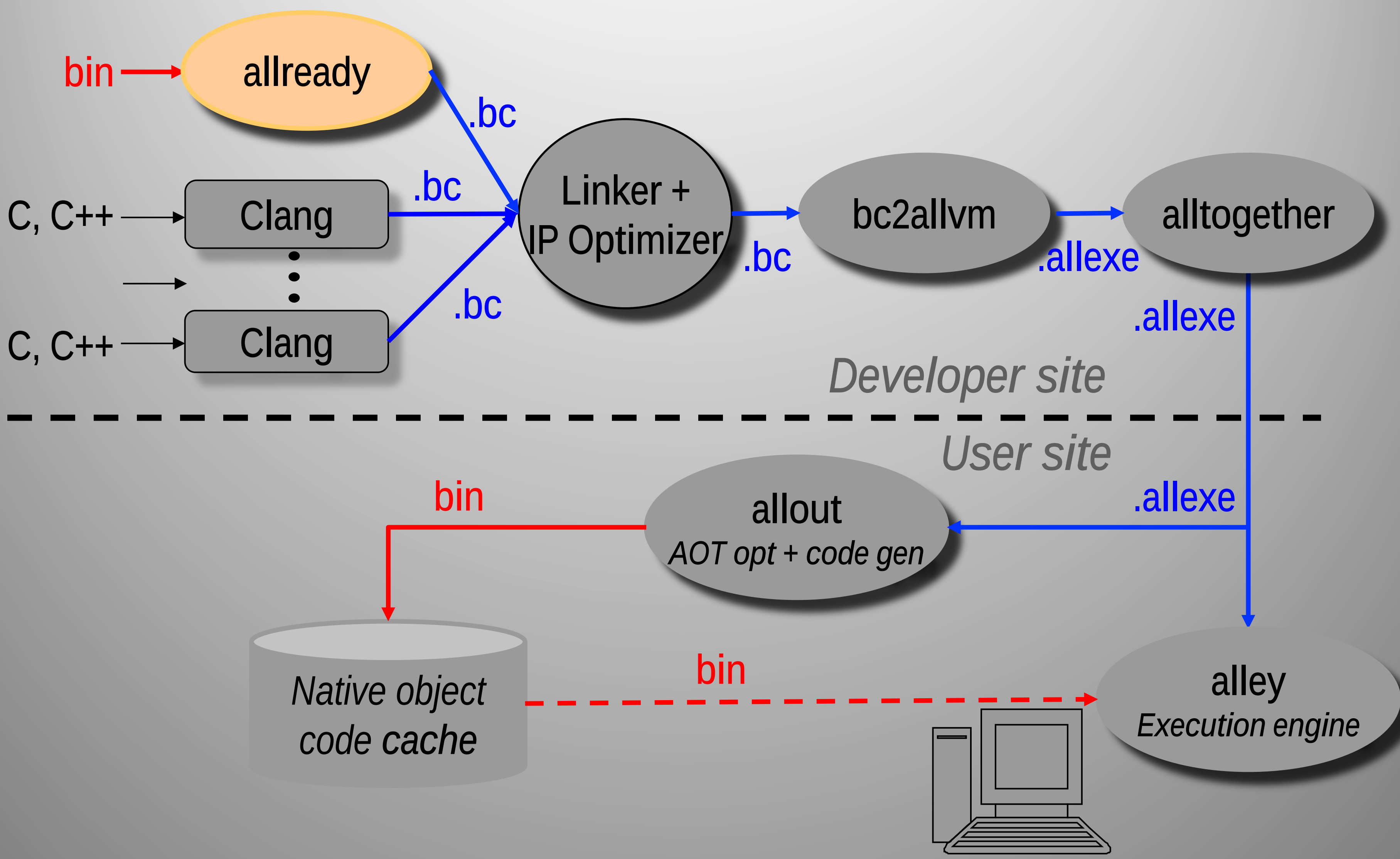
Linux Programs



Binary to LLVM Translation

Led by: Sandeep Dasgupta
with Ed Schwartz (CMU)

Binary-to-LLVM



allready: Binary-to-LLVM

Preference: Only a few components will be binary

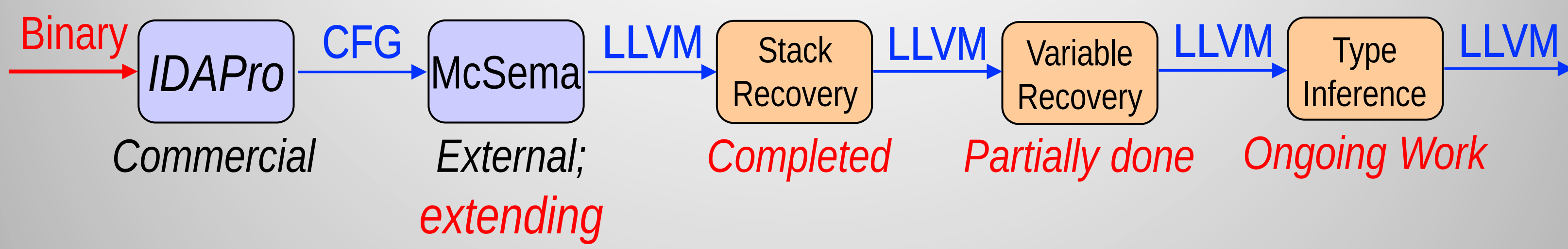
Motivation

- Some software components will only be in binary format
- Existing tools inadequate: McSema, BAP, SecondWrite, Qemu

Goal

- Extract “rich” LLVM IR from binary code
- Enable full set of ALLVM optimizations on partial-binary programs
- Needs variable info, type info, per-procedure stack frames

Current Status



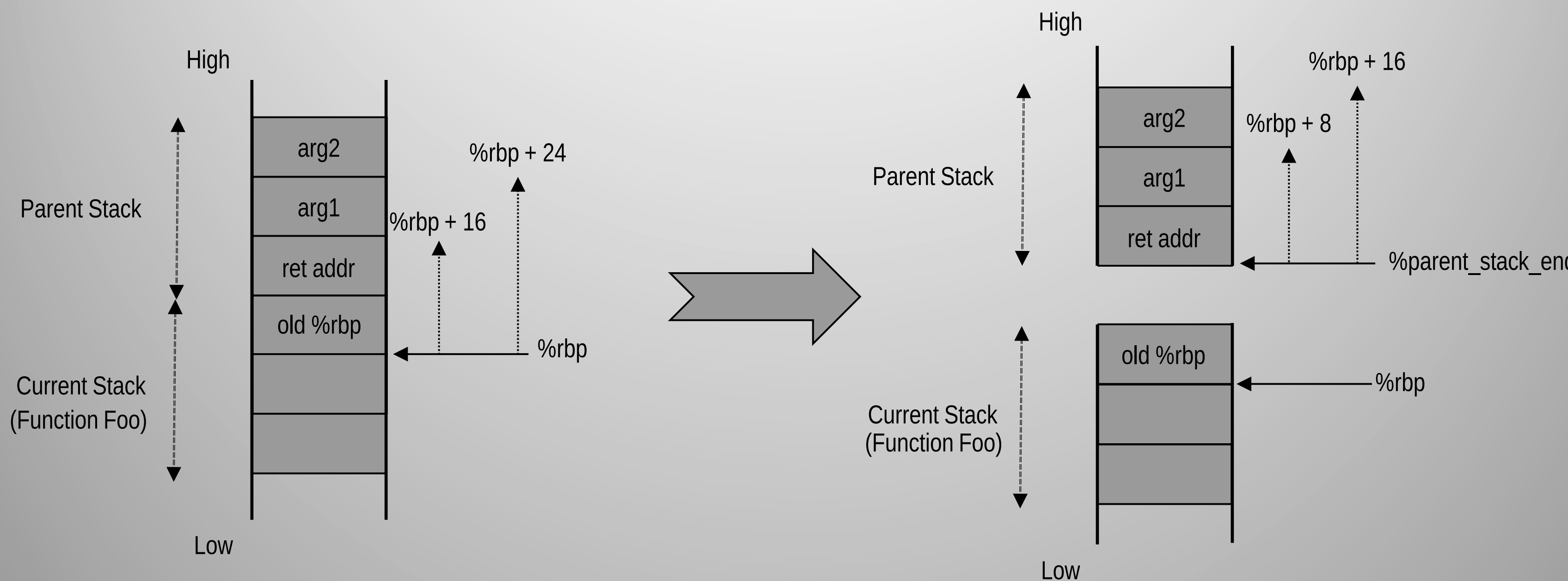
IR extracted by McSema is executable but very “low-level”

- Models runtime process stack as unified flat array
- Machine registers mapped in flat memory, not SSA virtual registers
- No information about variables, types, call graph, exceptions, etc.

Added stack deconstruction

- Recovers individual stack frames per function
- Distinguishes current vs. parent frame pointers
- Tested using McSema test suite; custom test cases

Stack Deconstruction




```
foo:
    push %rbp
    mov %rsp, %rbp
    mov 16(%rbp), %rax
    mov 24(%rbp), %r10
```

```
foo:
    push %rbp
    mov %rsp, %rbp
    mov 8(%parent_stack_end), %rax
    mov 16(%parent_stack_end), %r10
```

Ongoing Work

- Identify variables and promote them as symbols
- Represent every symbol in the IR with a meaningful type instead of the generic types provided by McSema

```
unsigned int foo(char* buf) {
        unsigned alligned_len = 0;
        unsigned int c = strlen(buf);
        if(c%8 == 0 ) {
                return c;
        }
        alligned_len = 8* (c/8) + 8;
        return allign_len;
}
```

Variable Names	C Type
1) buf	char*
2) c	unsigned int
3) alligned_len	unsigned int

1) and 2) inferred using *strlen* prototype
3) inferred using arithmetic operation

Takeaway Message

Proposal

All future software should ship as virtual ISAs

- The security benefits are strong
- There are no inherent performance penalties (and novel performance benefits are possible)
- It is technically feasible and commercially acceptable

http: // allvm.org

Summary and Implications

	Application / product areas
LLVM	Compilers; Mobile software; Security
HPVM	Mobile and embedded SoCs; Accelerators
DLVM	DNN toolkits and systems
ALLVM	Late-stage software customization; debloating; autotuning

Translation Validation for Increasing Trust in Compilation of Shipped Code

Led by: Theodoros Kasampalis
with Daejun Park and Prof. Grigore Rosu

Cross-Language Program Equivalence with Application to LLVM

Theodoros Kasampalis, PhD student

Joint work with Daejun Park, Vikram Adve, Grigore Rosu

Motivation

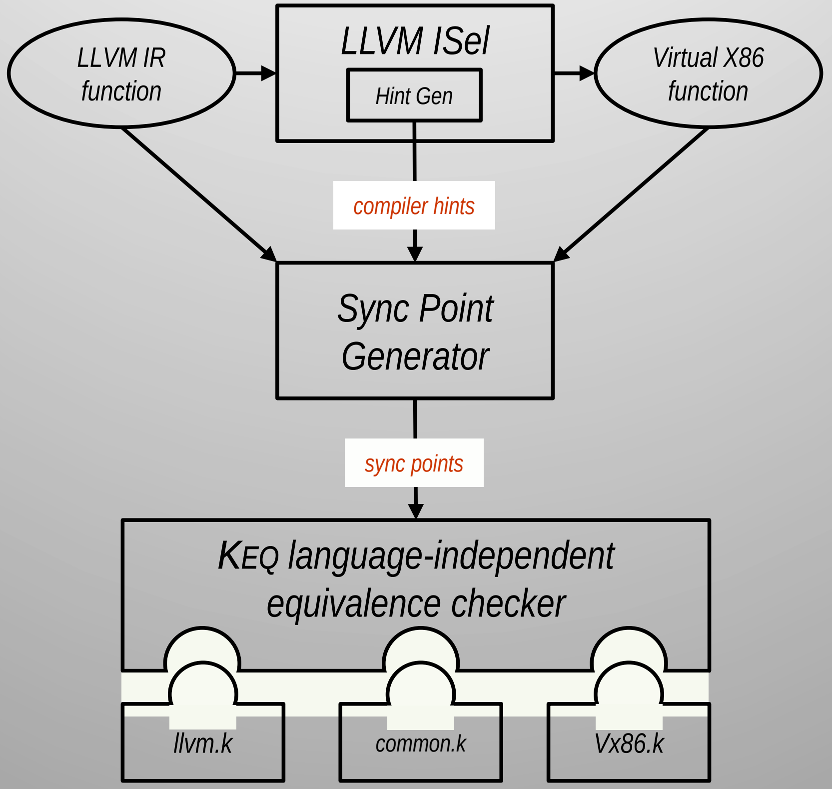
- Low trust in code generation process (*bugs, undefined behaviors*, etc.)

- Existing solutions are not practical for verified code generation with *production-quality compilers (e.g. LLVM)*

  - Verified Compilers (e.g. CompCert) – built from scratch

  - Translation Validation (e.g. LLVM-MD) – used primarily for same language transformations

TV prototype for LLVM ISel (IR to x86-64)



KEQ: K Equivalence Checker

- Input: a relation over symbolic states – called synchronization points
- Output: a bisimulation proof of program equivalence
  - Leverages our cut-bisimulation theory
- Built on top of K Framework
  - Leverages the K symbolic execution engine
- Language-independent
  - Parametrized with the input and output language semantics
  - Definitions defined in K

LLVM Instruction Selection Phase

- Translates LLVM IR into various target ISAs
  - primary language translation step beyond the front-end
  - 140,000 lines of C++ and TableGen code
- IR to Selection DAG for each basic block
  - Amenable to optimal pattern matching selection
- Output: Machine IR
  - Target ISA representation extended with some high-level features
  - Virtual x86: Machine IR for x86-64

K Semantic Definitions

	LLVM IR Semantics	Virtual x86 Semantics
Types	• varied-width integer types • composite array and struct types • the corresponding pointer types	• unsigned integers • various flag bits • 64-bit addresses
Features	• (un)signed integer arithmetic • Casts between ptrs/ints • getelementptr • (un)conditional branches • call/ret • alloca/load/store	• unsigned integer arithmetic • (un)conditional jumps • eflags register • various mov instructions • call/ret
Memory abstraction	map from symbolic addresses to memory objects represented as byte arrays	

Synchronization Point Generator

- Where?
  - Beginning/end of each function
  - Before/after each callsite
  - Before each loop header
- These points are a cut for each function
- Constraints over symbolic variables
  - Describe what parts of the two states should be “the same”

Synchronization Point Generator

Sync Point Type	Constraint	How to generate
Entry	corresponding args	from calling conv
Exit	same return value	from calling conv
Before call	corresponding args, same callee	from calling conv
Loop header	corresponding live regs	hints + liveness analysis
After call	same return value, corresponding live regs	from calling conv (return value), hints + liveness analysis

- *Required Static Analysis*
  - *Loop detection (natural loops)*
  - *Liveness analysis*
- *Hints*
  - *Virtual register correspondence*

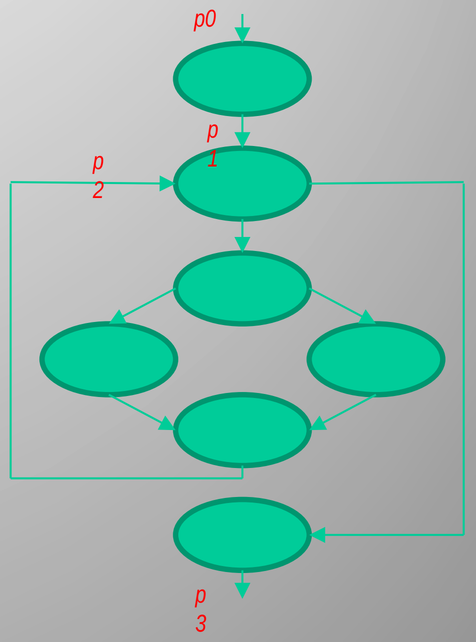
Example: The Collatz conjecture test

```
define i32 @collatz(i32 %n) {
entry:        ; p0
  br label %while.cond

while.cond:  ; p1, p2
  %c.0 = phi i32 [1,%entry], [%add1,%if.end]
  %n.0 = phi i32 [%n,%entry], [%n.1,%if.end]
  %cmp = icmp ne i32 %n.0, 1
  br i1 %cmp, label %while.body, label %while.end

while.body:
  %add1 = add i32 %c.0, 1
  %rem = urem i32 %n.0, 2
  %cmp1 = icmp ne i32 %rem, 0
  br i1 %cmp1, label %if.then, label %if.else

if.then:
  %mul1 = mul i32 %n.0, 3
  %add2 = add i32 %mul1, 1
  br label %if.end

if.else:
  %div = udiv i32 %n.0, 2
  br label %if.end

if.end:
  %n.1 = phi i32 [%add2,%if.then], [%div,%if.else]
  br label %while.cond

while.end:  ; p3
  ret i32 %c.0
}
```

(a) *LLVM IR*

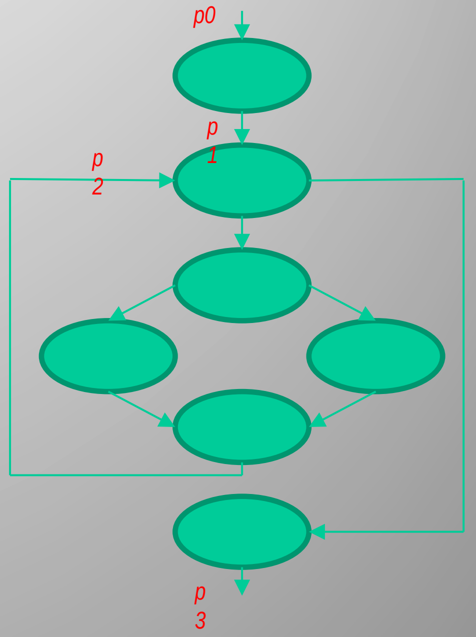
```
collatz:
.LBB0:      ; p0
    %vr6_32  = COPY edi
    %vr7_32  = mov  1
    jmp  .LBB1
.LBB1:      ; p1, p2
    %vr0_32  = PHI  %vr7_32, .LBB0, %vr2_32, .LBB5
    %vr1_32  = PHI  %vr6_32, .LBB0, %vr5_32, .LBB5
    %vr8_32  = sub  %vr1_32, 1
    je   .LBB6
    jmp  .LBB2
.LBB2:
    %vr2_32  = inc  %vr0_32
    %vr9_8   = COPY %vr1_32:sub_8bit
    test %vr9_8, 1
    je   .LBB4
    jmp  .LBB3
.LBB3:
    %vr10_64 = SUBREG_TO_REG %vr1_32
    %vr3_32  = lea  [%vr10_64 + 2*%vr10_64 + 1]
    jmp  .LBB5
.LBB4:
    %vr4_32  = shr  %vr1_32
    jmp  .LBB5
.LBB5:
    %vr5_32  = PHI  %vr4_32, .LBB4, %vr3_32, .LBB3
    jmp .LBB1
.LBB6:
    eax      = COPY %vr0_32
    ; p3
    ret
```

(b) *Virtual x86*



Example: The Collatz conjecture test

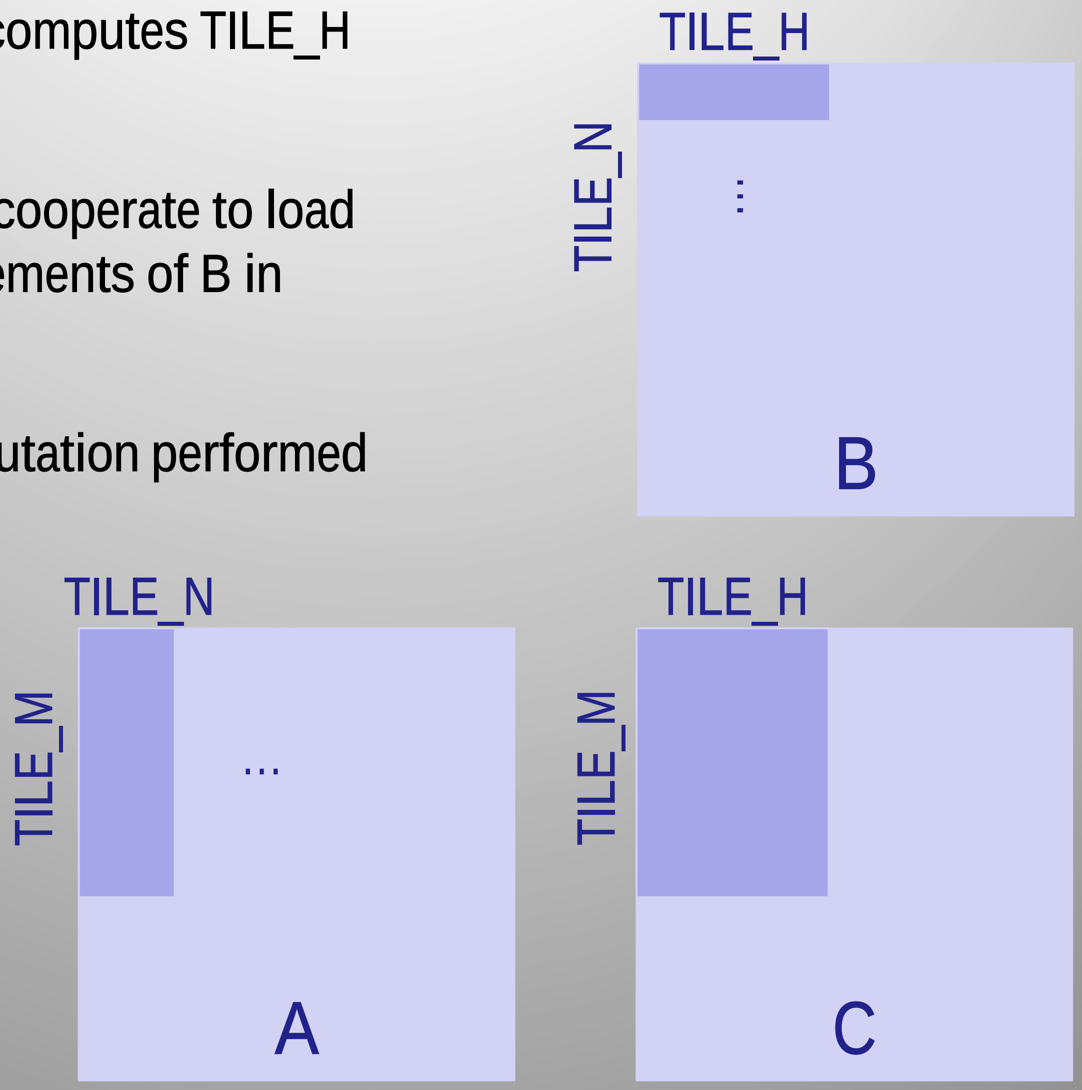
Sync Point	*Prev BB* (LLVM)	*Prev BB* (Vx86)	*Equality Constraints*	
p0 (entry)	-	-	`%n = edi`	
p1	`%entry`	`.LBB0`	`%n = %vr6_32`	`1 = %vr7_32`
p2	`%if.end`	`.LBB5`	`%add1 = %vr2_32`	`%n.1 = %vr5_32`
p3 (exit)	-	-	`%c.0 = eax`	



Questions?

Example: Sgemm

- A single work item computes TILE_H elements of C
- TILE_M work items cooperate to load TILE_H x TILE_N elements of B in local memory
- Figure shows computation performed by one work group



SGEMM – Dataflow Graph Structure