A Framework for Understanding the Portability of C Types

Marius Nita

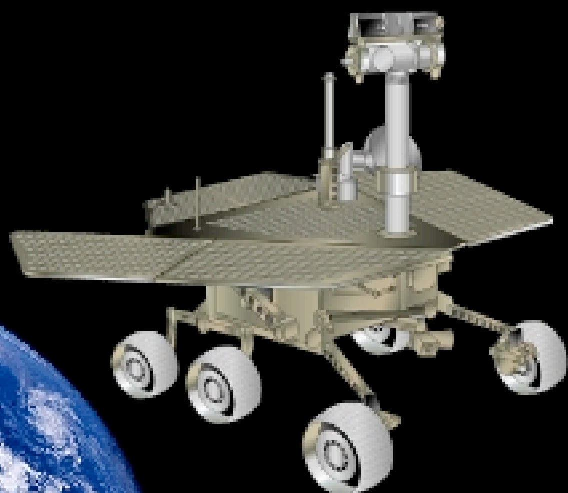
May 16, 2007

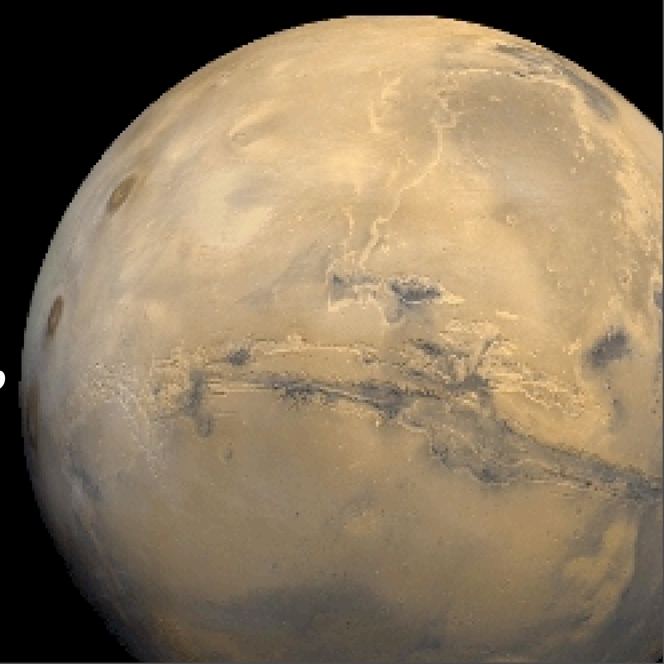
Advised by Dan Grossman

The Ultimate Portability Question

“Will it run over there?”

I tackled a slightly smaller problem...





"Will it run over there?"

Outline

- Motivation & Background
- Key Ingredients
- Implementation
- Case Study
- Conclusions & Future Work

The Problem

- C definition leaves type layout unspecified.
- Different platforms make different choices.
- The same type may work as intended on one platform but cause unexpected behavior on another.

For Example,

```
struct A { int  a; long b; };
struct B { long x; long y; };

struct A *pa = ...;
...
struct B *pb = (struct B*)pa;
...
memcpy(x, y, pb->x);
```

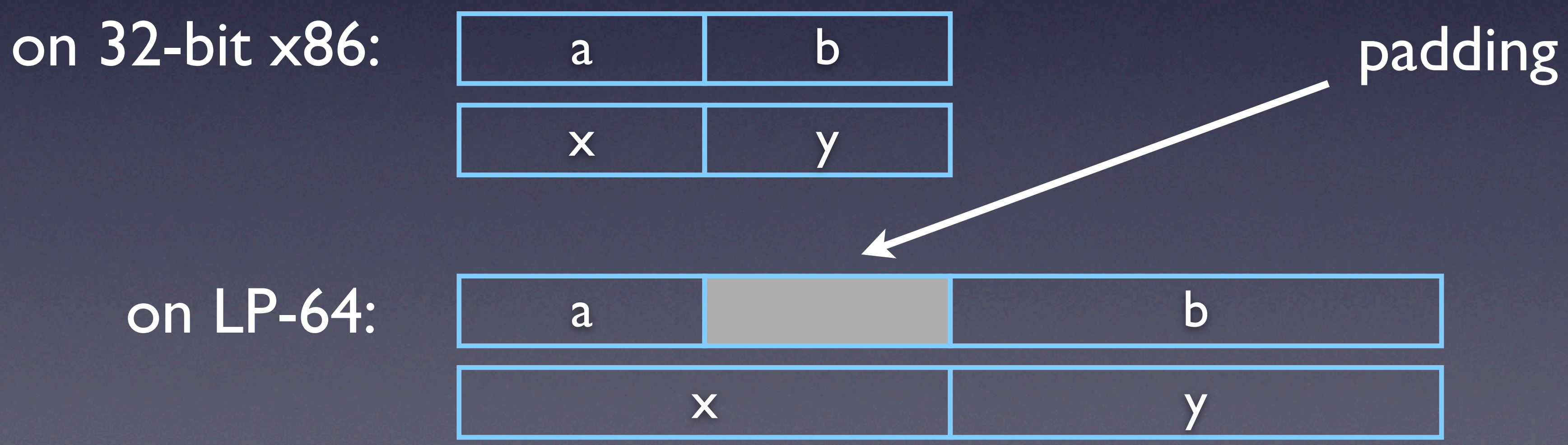
For Example,

```
struct A { int  a; long b; };
struct B { long x; long y; };

struct A *pa = ...;
...
struct B *pb = (struct B*)pa;
...
memcpy(x, y, pb->x);
```

on 32-bit x86:

For Example,

```
struct A { int  a; long b; };
struct B { long x; long y; };

struct A *pa = ...;
...
struct B *pb = (struct B*)pa;
...
memcpy(x, y, pb->x);
```



How can you tell if your types are portable?

Programmer's Burden

- Understand the language definition.
- Understand each platform's choices.
- *Manually* isolate problems.
- Cross-compile, test on target platform.
- Very raw tool support:
  - Special compiler flags.
  - Lint-like technology

This Work: Envisioned Scenario

gcc/ALPHA

gcc/SPARC

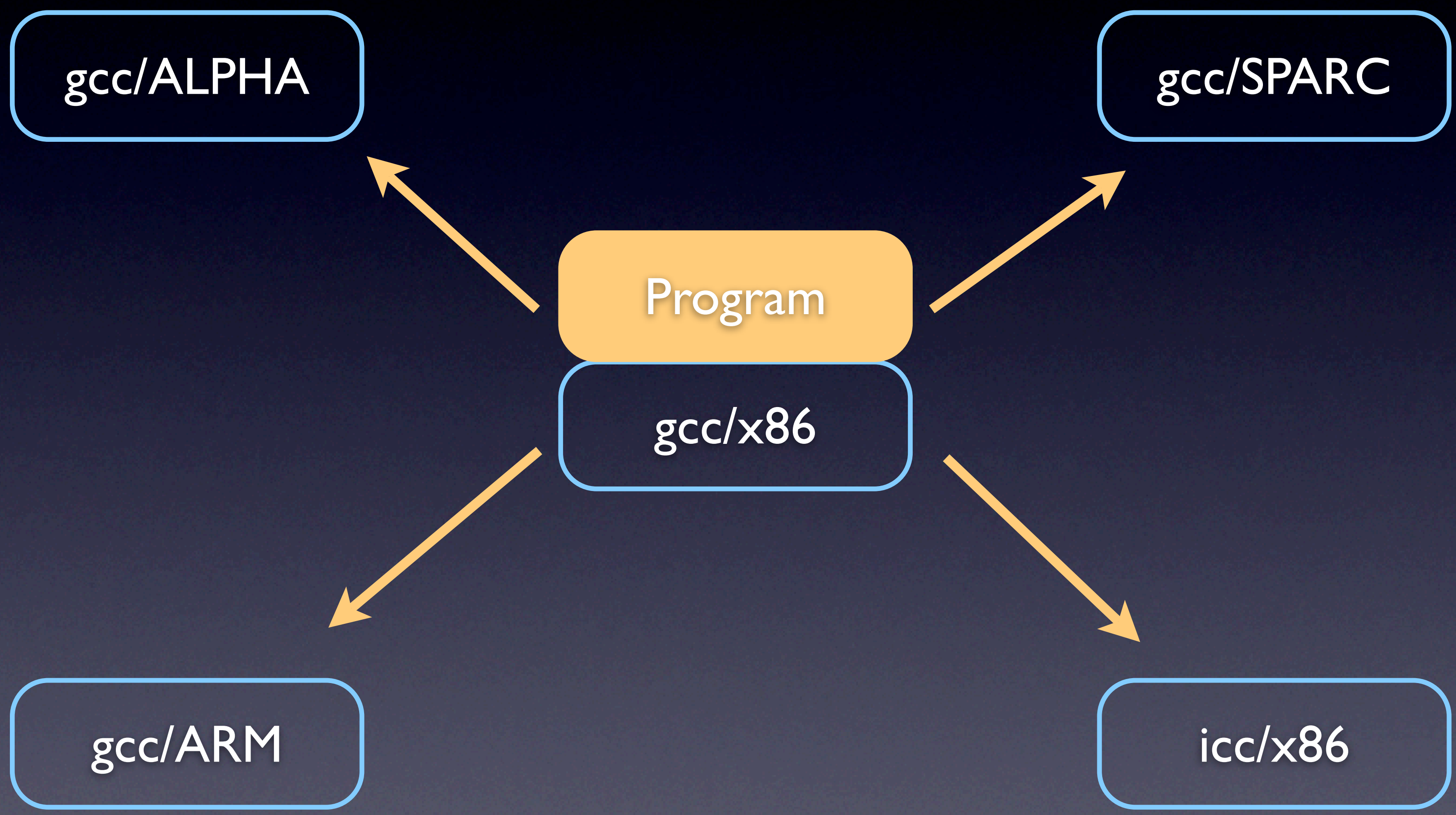
gcc/x86

gcc/ARM

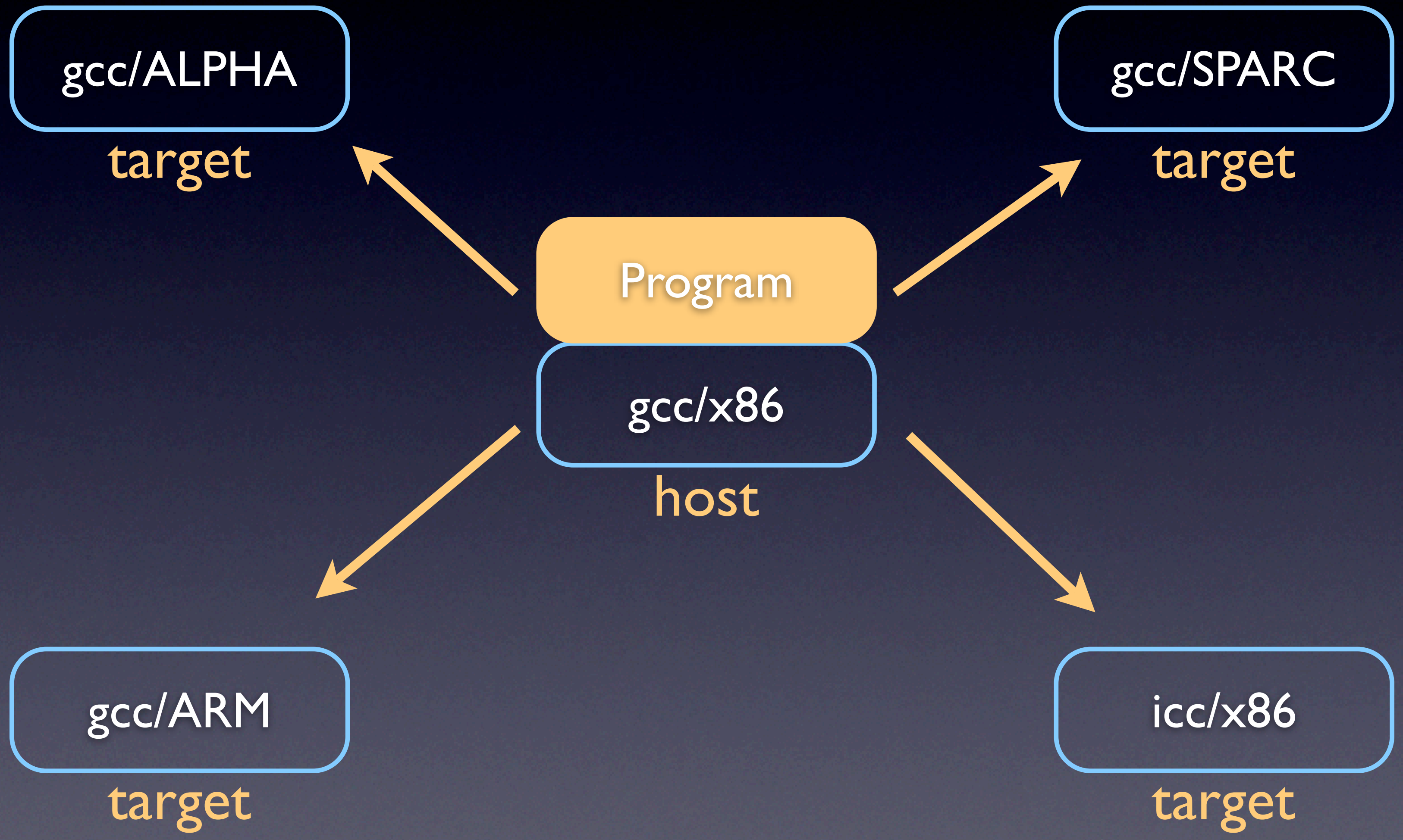
icc/x86

This Work: Envisioned Scenario

This Work: Envisioned Scenario



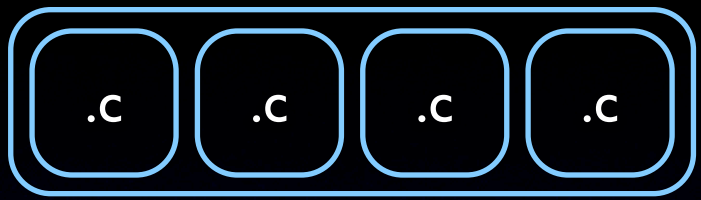
This Work: Envisioned Scenario



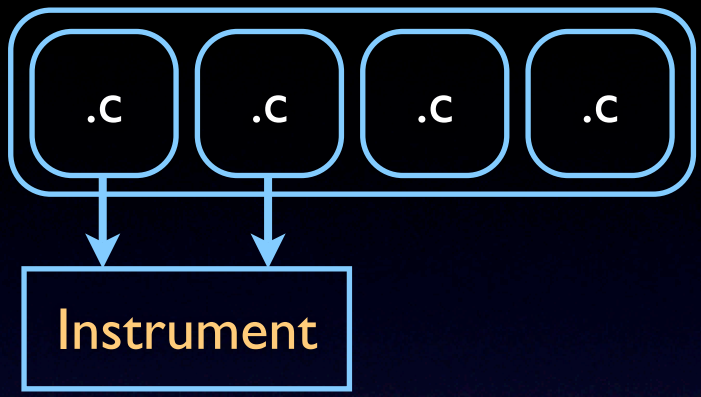
Tool Overview

- A set of ***layout assumptions*** are extracted from a C program.
- Each assumption is ***checked*** against the host and target(s).
- If the assumption is false, a warning is reported.
- A ***platform*** is a description of how a particular compiler lays out types on a particular machine.

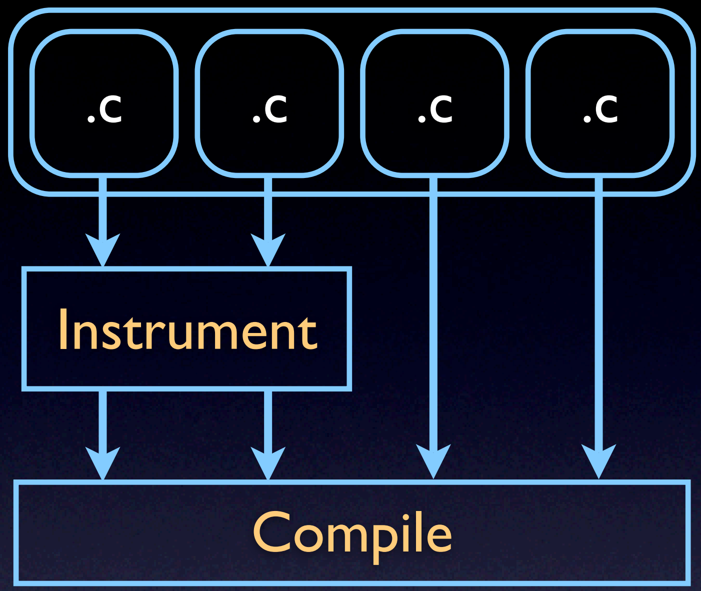
Implementation Architecture



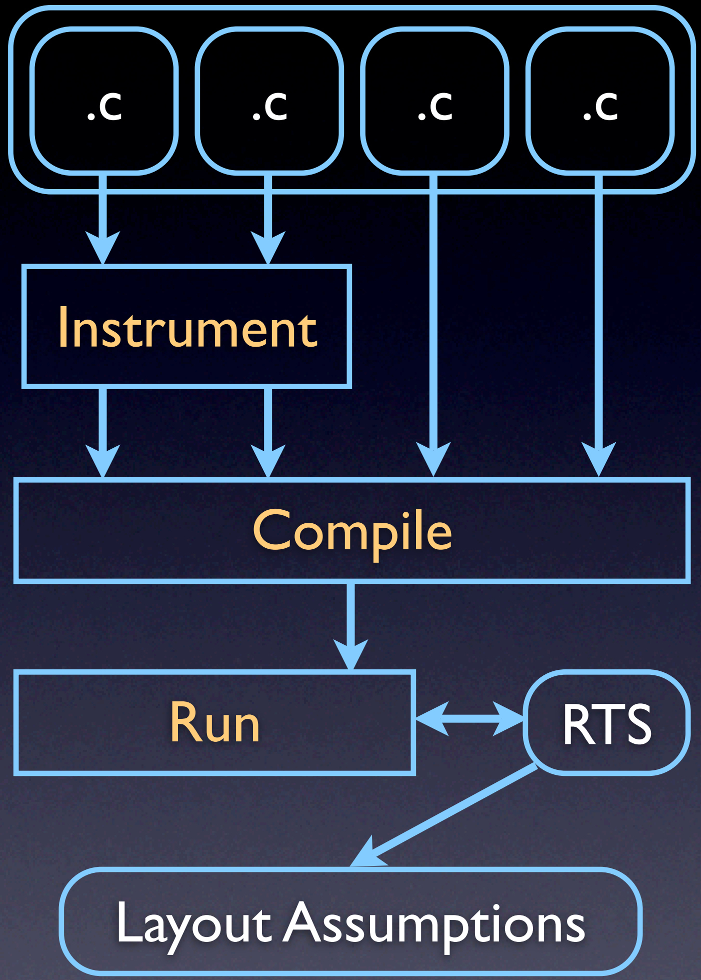
Implementation Architecture



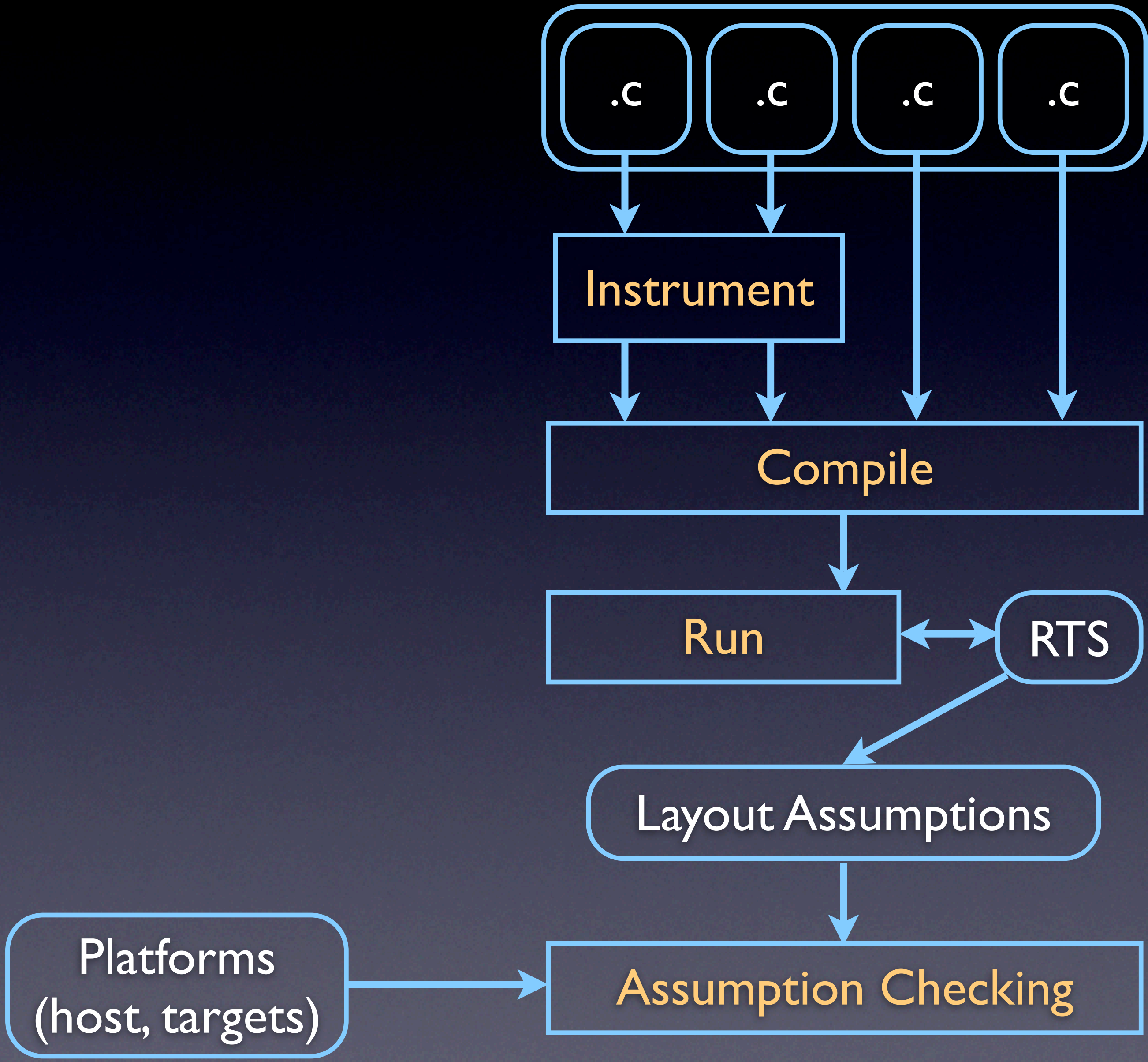
Implementation Architecture



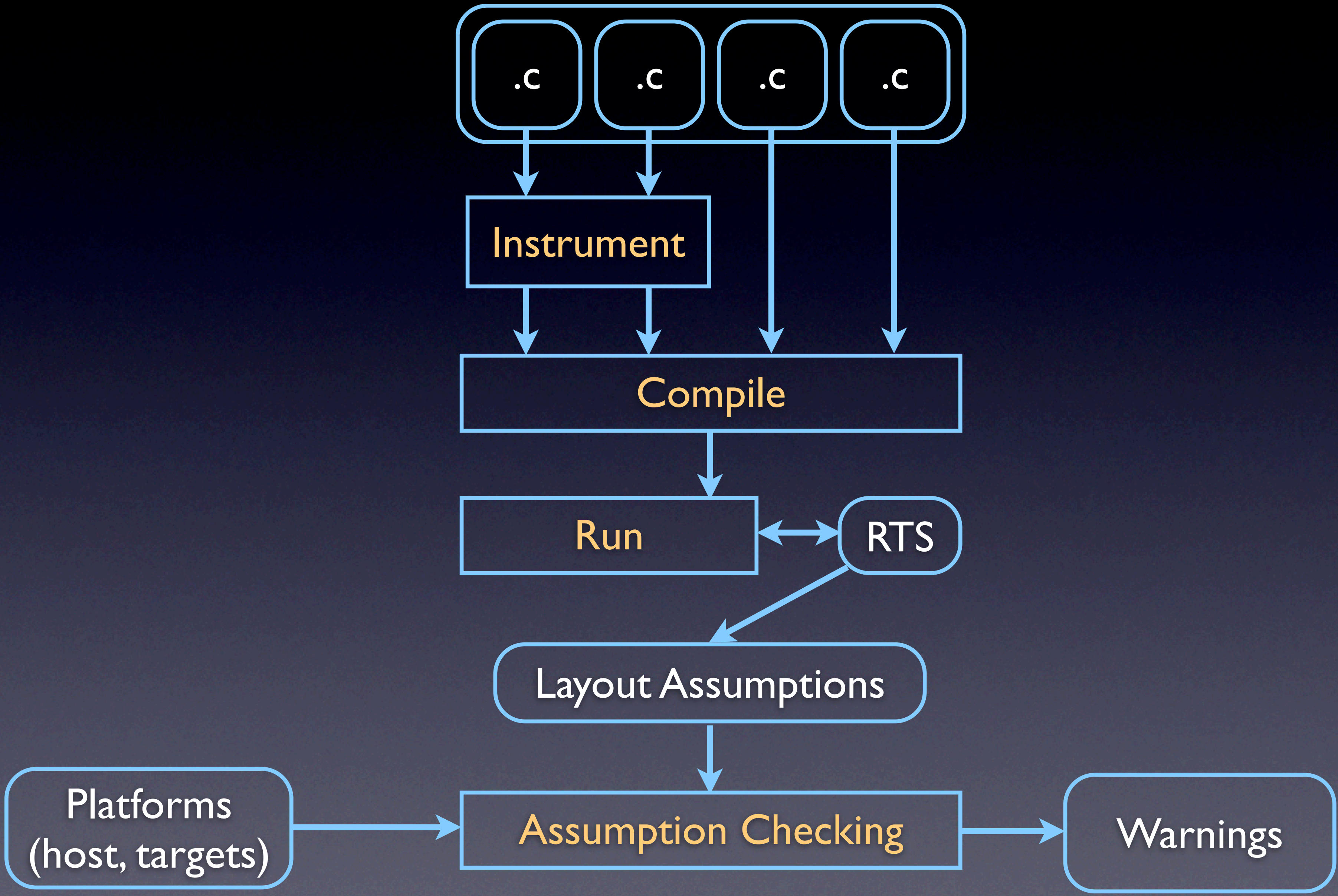
Implementation Architecture



Implementation Architecture



Implementation Architecture



Outline

- Motivation & Background
- **Key Ingredients**
    - Memory Layout Language
    - Representing Platforms
    - Layout Subtyping
    - Representing Layout Assumptions
- Implementation
- Case Study
- Conclusions & Future Work

Key Ingredients

- We want to reason about how a platform lays out types in memory.
- We need an explicit notion of layouts.
- We need a way to represent platforms.
- We need a way to represent, extract, and check the layout assumptions a C program makes.

Memory Layout Language

Layout ::=	**byte**	*(Byte of data)*
	| **pad**	*(Byte of padding)*
	| $\mathbf{ptr}_n$(*Layout*)	*(Pointer to n-byte aligned data)*
	| *Layout Layout*	*(Sequence)*

Memory Layout Language: Example

Pointer to: `struct Pixel { char r,g,b; }`

x86: $\mathbf{ptr}_1$(**byte byte byte**)

ARM: $\mathbf{ptr}_4$(**byte byte byte pad**)

Representing Platforms

- Capture the subset of a platform that deals with type layout.
  - Sizes and alignments of types.
  - Offsets of struct fields.
  - Algorithm for recursively laying out structs.

A Platform is a Type Compiler

Platform`.ptrsize` : *Int*
Platform`.alignof` : *Type* → *Int*
Platform`.offsetof` : *FieldName* → *Int*

Platform*`.xtype`** **:** ***Type* → *Layout

To represent a real-world platform, faithfully implement these four functions.

Layout Subtyping

```
struct A *pa = ...;
...
struct B *pb = (struct B*)pa;
```

- When is the pointer-cast `(struct B*)pa` safe?
  - When the layout of `struct A*` can be treated as if it were the layout of `struct B*`.
- Captured by a notion of **layout subtyping**.

Layout Subtyping: Key Rule

If *n* is a multiple of *m*, then

$$\mathbf{ptr}_n(\mathit{Layout}_1\ \mathit{Layout}_2) \le \mathbf{ptr}_m(\mathit{Layout}_1)$$

A pointer **can be treated as** a pointer to a shorter prefix.

Representing Layout Assumptions

- A **layout assumption** is a statement demanding that platforms behave a certain way.
  - E.g. "alignment of int must be 8"
- Unspecified **type uses** generate assumptions.
- Ultimately represented as formulas in a first-order logic.

Extracting Assumptions

If the type of e is *S**, then for expression

(*D**)e

the corresponding assumption is

“If

host_layout(S) ≤ host_layout(D*)*,

then

target_layout(S) ≤ target_layout(D*)*.”

Checking Assumptions

- Assumptions are checked in the context of a host platform (*Host*) and a target platform (*Target*).
- Resolve symbols and see if the assumption is true/false.
- Resolving symbols, examples:

host_layout(*T*)	*Host*.`xtype`(*T*)
target_layout(*T*)	*Target*.`xtype`(*T*)

Outline

- Motivation & Background
- Key Ingredients
- **Implementation**
- Case Study
- Conclusions & Future Work

Implementation

- Implemented on top of George Necula's CIL.
- Platforms are directly implemented in Caml.
  - Records of functions.
  - Simple "type compilers."
- The instrumenter is a C compiler.
  - make CC=/the/instrumenter
- Runtime system is a small C library.

Runtime System

- Responsible for recording run-time layout assumptions.
- Disk usage/IO overhead potentially issues.
    - E.g. assumption being registered in a tight loop.
- Runtime buffers assumptions and discards duplicates.
- Program slowdown is negligible.

Outline

- Motivation & Background
- Key Ingredients
- Implementation
- **Case Study**
- Conclusions & Future Work

Spread

- A high-performance messaging service for distributed apps.
- Library + 5 executables.
- Instrumented all of it and ran it on basic inputs.
- Tool caught one 64-bit portability bug.
  - 32-bit x86 host platform.
  - 64-bit "LP-64" target platform.
  - 47 unique layout assumptions generated.
  - One false positive, one bug.

Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
...
struct scat_element *elem;
...
iov = (struct iovec*)elem;
...
//treat iov->len as length of iov->buf
```

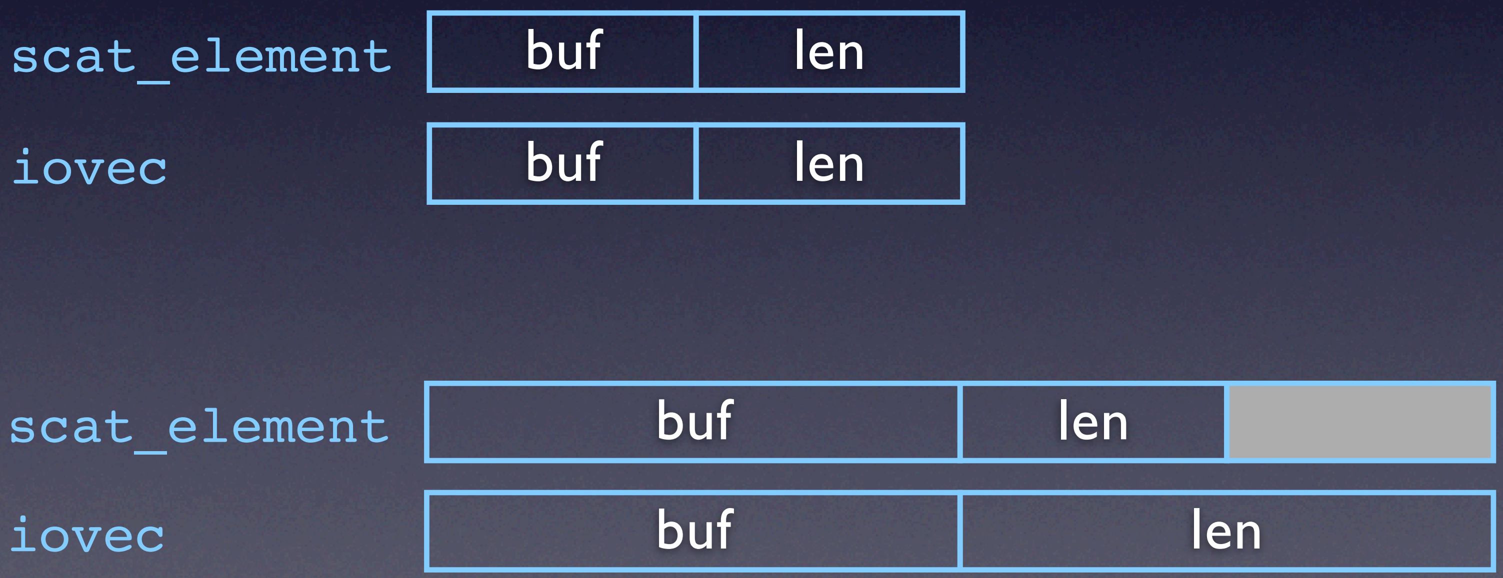
Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
...
struct scat_element *elem;
...
iov = (struct iovec*)elem;
...
//treat iov->len as length of iov->buf
```

Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
```

```
iov = (struct iovec*)elem;
```



Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
```

```
iov = (struct iovec*)elem;
```

Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
```

```
iov = (struct iovec*)elem;
```

↓

$$host\_layout(\texttt{scat\_element*}) \leq host\_layout(\texttt{iovec*})$$
$$\Rightarrow target\_layout(\texttt{scat\_element*}) \leq target\_layout(\texttt{iovec*}))$$

Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
```

```
iov = (struct iovec*)elem;
```

↓

host_layout(`scat_element*`) ≤ *host_layout*(`iovec*`)
⇒ *target_layout*(`scat_element*`) ≤ *target_layout*(`iovec*`))

↓

$\mathbf{ptr_4}(\mathbf{ptr_1}(\mathbf{byte})\ \mathbf{byte}^4) \leq \mathbf{ptr_4}(\mathbf{ptr_1}(\mathbf{byte})\ \mathbf{byte}^4)$
$\Rightarrow \mathbf{ptr_8}(\mathbf{ptr_1}(\mathbf{byte})\ \mathbf{byte}^4\ \mathbf{pad}^4) \leq \mathbf{ptr_8}(\mathbf{ptr_1}(\mathbf{byte})\ \mathbf{byte}^8)$

Broken Layout Assumption

```
struct scat_element { char *buf; int    len; };
struct iovec        { char *buf; size_t len; };
```

```
iov = (struct iovec*)elem;
```

↓

host_layout(`scat_element*`) ≤ *host_layout*(`iovec*`)
⇒ *target_layout*(`scat_element*`) ≤ *target_layout*(`iovec*`))

↓

$\mathbf{ptr_4(ptr_1(byte)\ byte^4) \le ptr_4(ptr_1(byte)\ byte^4)}$ True
$\Rightarrow \mathbf{ptr_8(ptr_1(byte)\ byte^4\ pad^4) \le ptr_8(ptr_1(byte)\ byte^8)}$ False

Outline

- Motivation & Background
- Key Ingredients
- Implementation
- Case Study
- **Conclusions & Future Work**

Conclusions

- Developed an approach and tool to discover type layout portability issues in C code.
- Can be used to find real bugs
    - by checking against platform descriptions modeling real concrete platforms.
- Can be used to understand portability boundaries
    - by checking against many “weird” platform descriptions that may not exist but are legal within the C standard.

Future Work

- Static analysis.
    - Dynamic analysis has many advantages.
    - But hard to apply to arbitrary C code (e.g. kernels).
- Strong Evaluation.

Questions?

http://www.cs.washington.edu/homes/marius/quals.pdf