Formal Synthesis of Efficient Verified Emulators

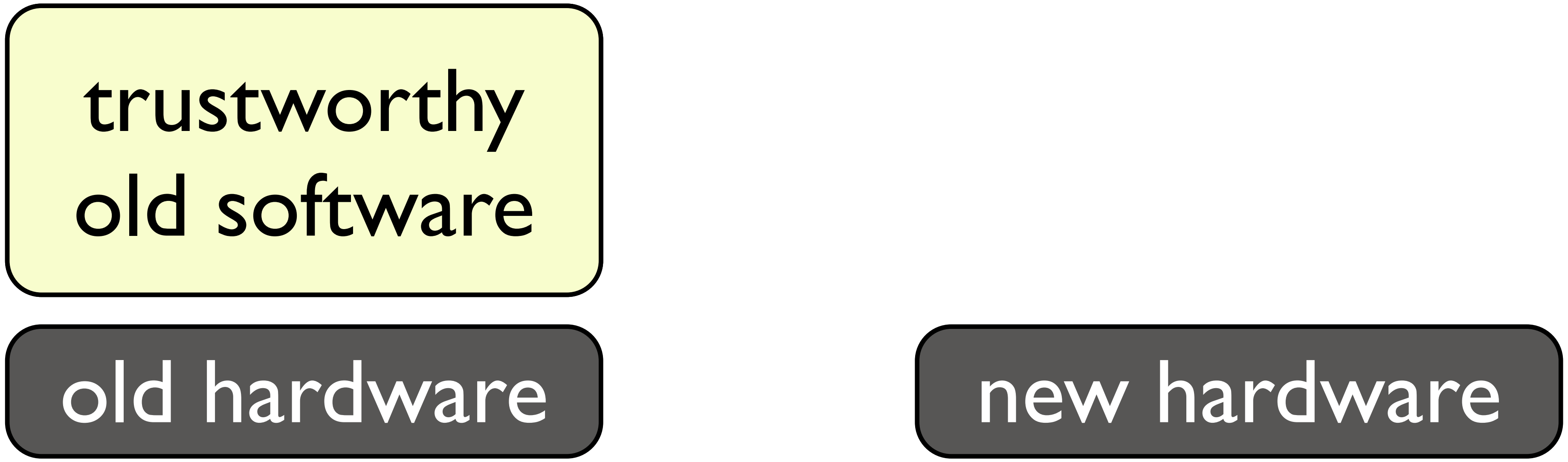
Magnus Myreen[1]

in collaboration with Anthony Fox and Mike Gordon

University of Cambridge

[1] funded by Defence Science & Technology Laboratory (DSTL), UK

Problem of new hardware

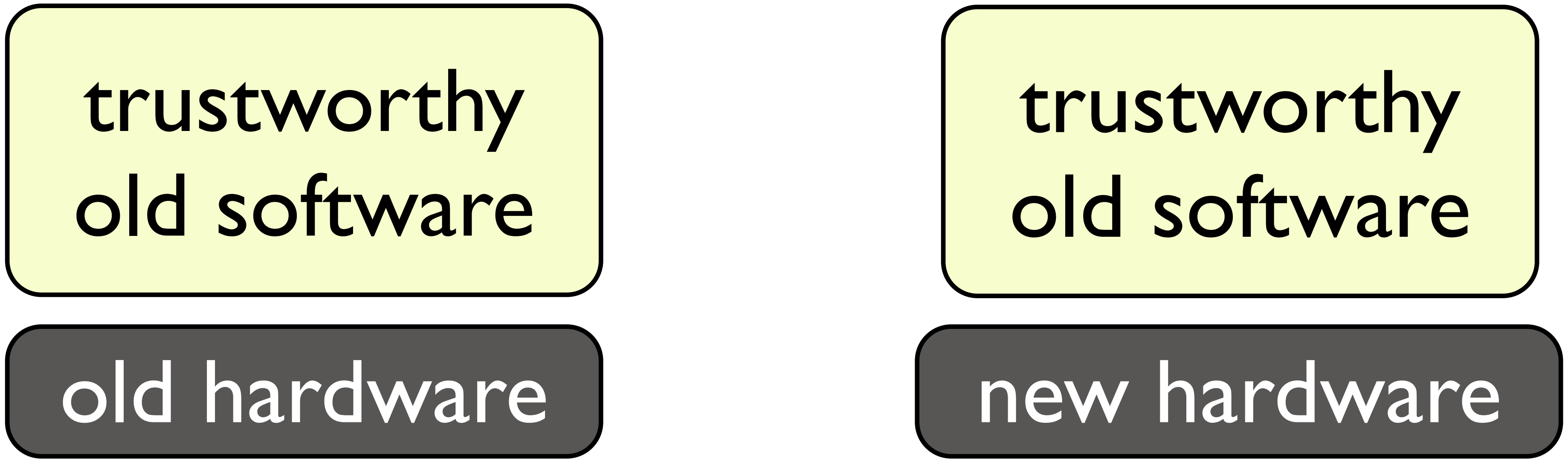


trustworthy old software

old hardware

new hardware

Problem of new hardware



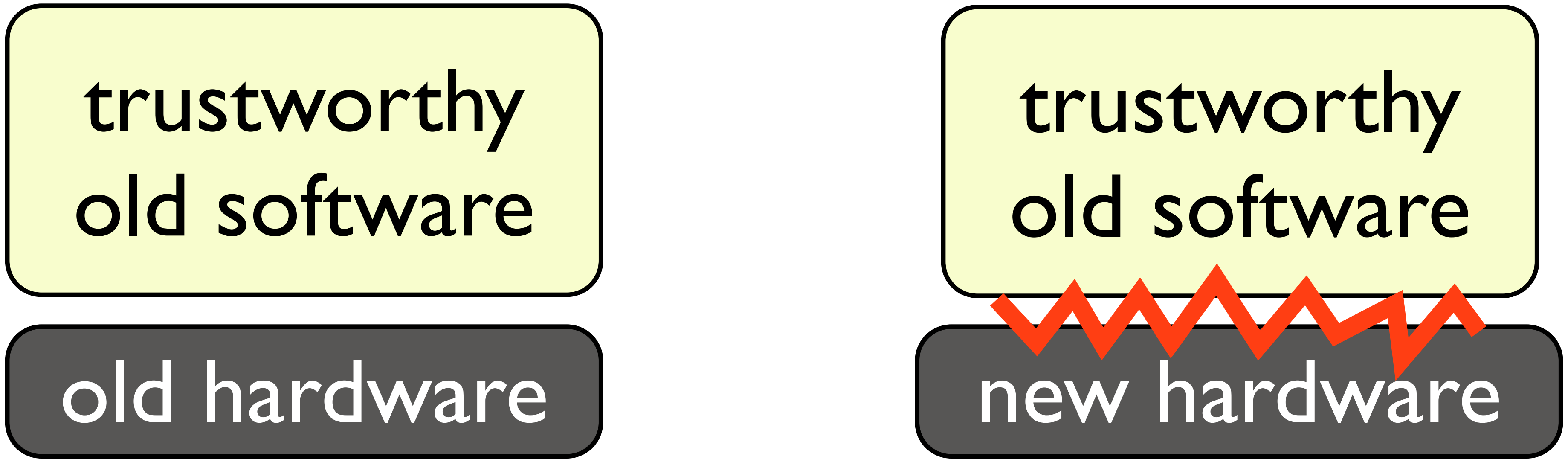
trustworthy
old software

old hardware

trustworthy
old software

new hardware

Problem of new hardware



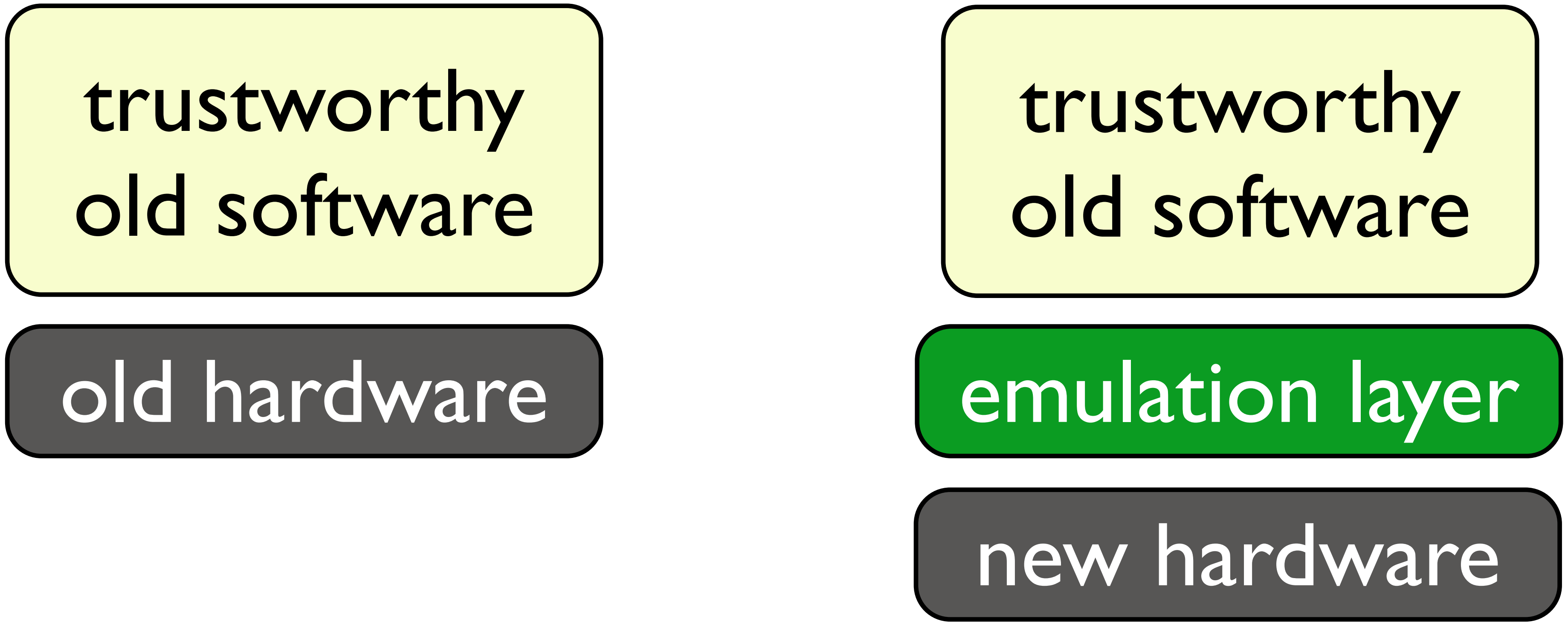
trustworthy old software

old hardware

trustworthy old software

new hardware

Problem of new hardware



trustworthy old software

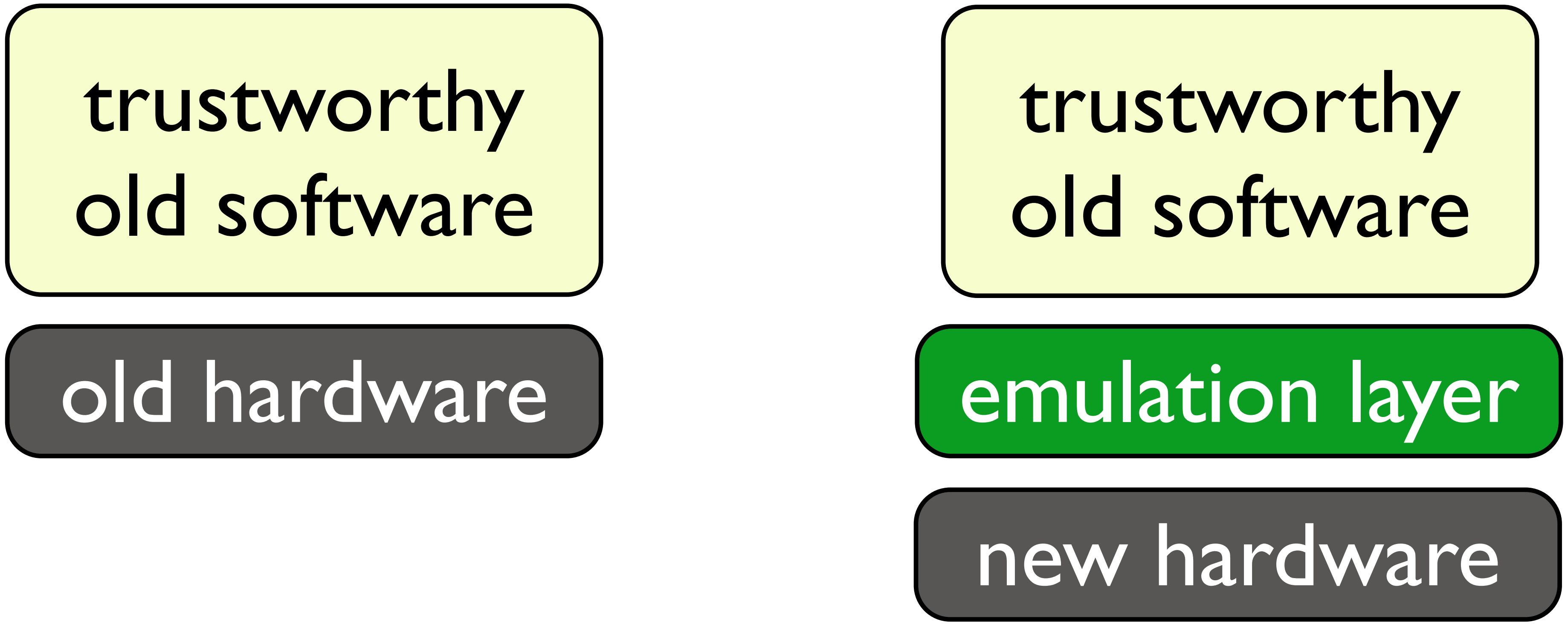
old hardware

trustworthy old software

emulation layer

new hardware

Problem of new hardware


trustworthy old software

old hardware

trustworthy old software

emulation layer

new hardware

- Re-verification/validation is expensive
- This talk: how to build trustworthy emulators

Emulators

Purpose: recreating an original computer environment

Goals: recreate hardware or hardware+OS

Emulators

Purpose: recreating an original computer environment

Goals: recreate hardware or hardware+OS

This talk: emulating ARM programs on 64-bit x86

- emphasis: correctness and efficiency
- focus: self-contained, user-mode programs

Emulator alternatives

Emulators implement

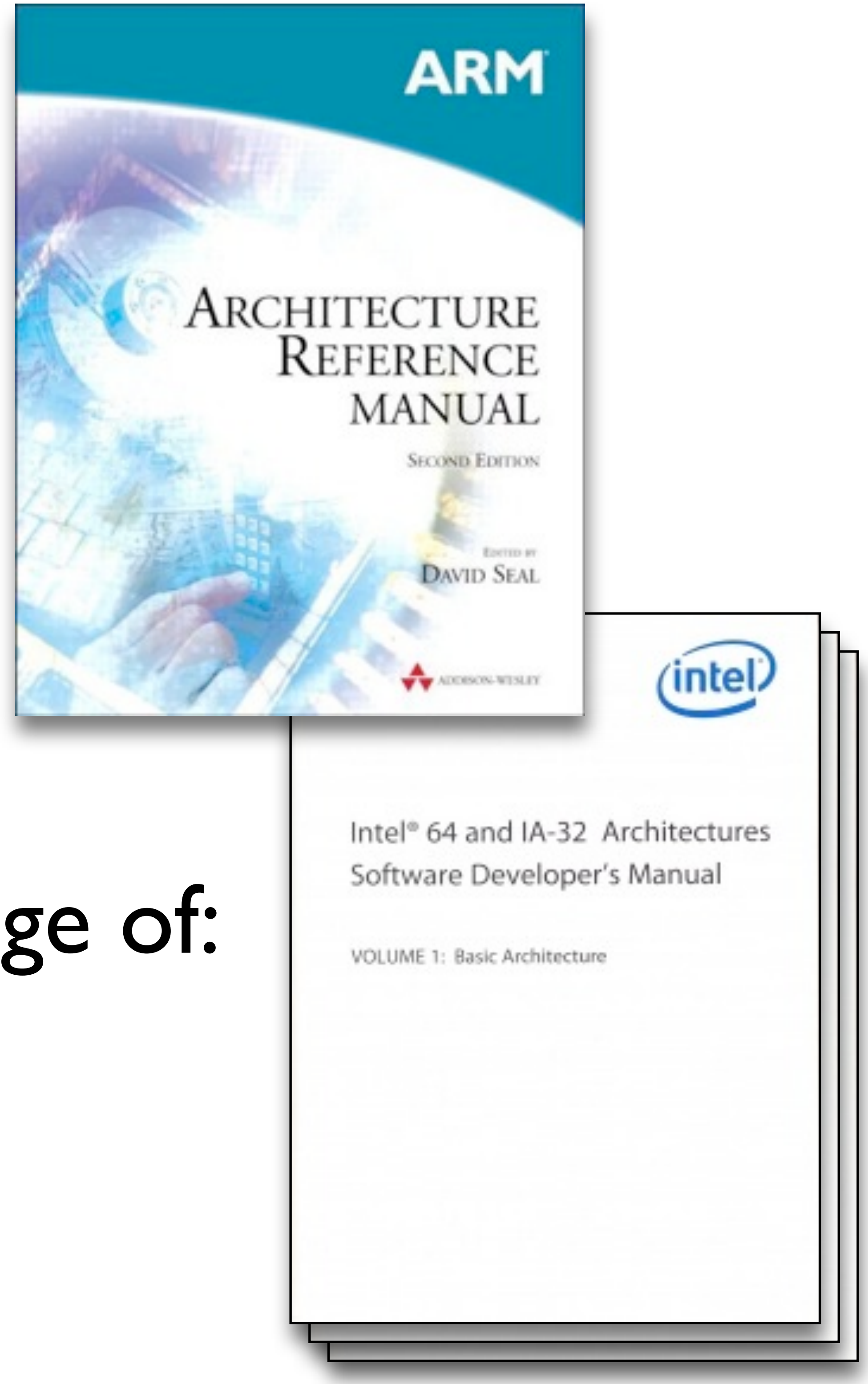
fetch-decode-exec-cycle of foreign architecture

Implementation alternatives:

- fetch-decode-exec-cycle interpretation
- just-in-time compilation
- one-off binary translation

Trustworthy?

Writing an emulator
involves implementing:



ARM

ARCHITECTURE
REFERENCE
MANUAL

SECOND EDITION

EDITED BY
DAVID SEAL

ADDISON-WESLEY

intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

VOLUME 1: Basic Architecture

in the language of:

... an error-prone task.

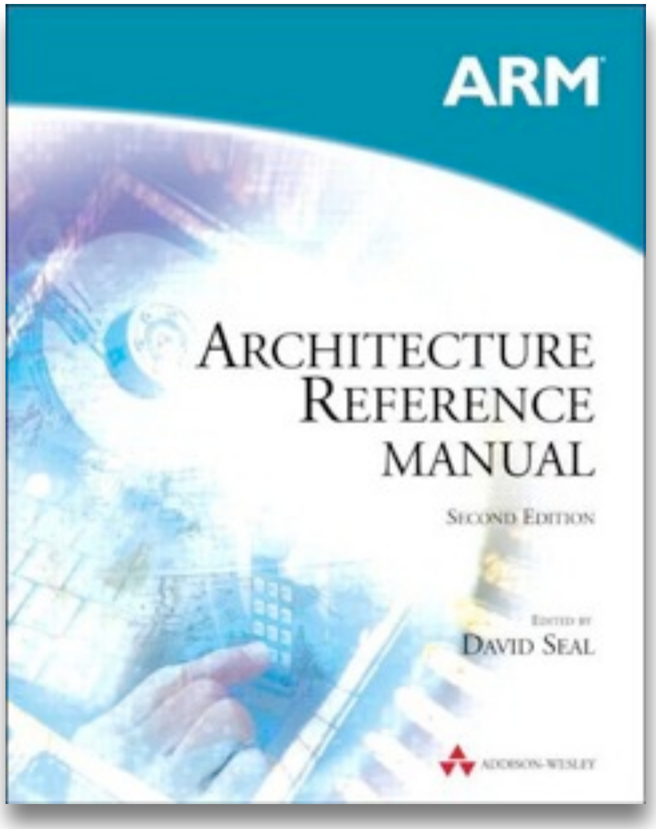
This Talk

1. Construction of trustworthy emulators:
    - direct interpretation
    - just-in-time compilation
    - one-off binary translation
2. Comparison & performance numbers

Direct Interpretation

Specification

- Instruction set architectures, foreign and native:


ARM

ARCHITECTURE
REFERENCE
MANUAL

SECOND EDITION

EDITED BY
DAVID SEAL

ADDISON-WESLEY


intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

VOLUME 1: Basic Architecture

- We use Fox [ITP'10] and Sarkar et al. [POPL'09]

Formal specification

- Formal models defined as interpreters, e.g.

```
arm_next(state) =
  let ast = decode(fetch(state),state) in
    exec(ast,state)
```

in the logic of a theorem prover.

Formal specification

- Formal models defined as interpreters, e.g.

```
arm_next(state) =
  let ast = decode(fetch(state),state) in
    exec(ast,state)
```

  in the logic of a theorem prover.

- ... so let's synthesise verified x86 from the definition arm_next.

Synthesis of interpreter

- Drawing on experience of proof-producing synthesis [CC'09, TPHOLs'09, ITP'11]
- ARM model difficult to directly synthesise to efficient x86 code: definition uses
  - heterogenous datatypes (AST)
  - higher-order functions

Synthesis of interpreter

- Drawing on experience of proof-producing synthesis [CC'09, TPHOLs'09, ITP'11]
- ARM model difficult to directly synthesise to efficient x86 code: definition uses
  - heterogenous datatypes (AST)
  - higher-order functions
- Solution: reformulate arm_next.

Reformulation

- Instead of: decode-then-execute, i.e.

  decode : word32 → AST

  execute : AST x state → state

Reformulation

- Instead of: decode-then-execute, i.e.

  decode : word32 → AST

  execute : AST x state → state

- Use: interpretation via bytecode

  translate : word32 → bytecode list

  interpret : (bytecode list) x state → state

i.e. arm_next(s) = interpret(translate(...),s)

Bytecode

Bytecode state:

- four new registers: A, B, C, D
- ARM processor state

Bytecode

Bytecode state:

- four new registers: A, B, C, D
- ARM processor state

Bytecode instructions:

- basic operations between A-D registers (add A,A,B or sub A,A,B or mov A,D etc.)
- operations for reading and updating ARM state (e.g. mov A,r0 or mov r0,A)
- one operation for skipping instructions

Synthesis

We write definition of:

translate : word32 → bytecode list

interpret : (bytecode list) x state → state

in a language which we can easily be compiled by proof-producing synthesis [CC'09] (explained later)

(Implementing a full translate function is work in progress...)

Example emulation

- Fib for even numbers in C and ARM

```
m = 0;          mov r0,#0
n = 1;          mov r1,#1
repeat {      L:
  m += n;       add r0,r1
  n += m;       add r1,r0
  k -= 2;       subs r2,#2
} (k == 0);     bne L
```

- Emulates fib(200,000,000) in 48 seconds
- x86-complied C runs in 0.1 seconds (500x faster)

Just-in-time compilation

Just-in-time compilation

Idea: partial evaluation

- try to perform fetch-and-decode only once
- QEMU design principle (animation next slide...)

JIT animation

Foreign code:

```
→ 40: mov r0,#0
  44: mov r1,#1
  48: add r0,r1
  52: add r1,r0
  56: subs r2,#2
  60: bne 48
```

Native code:

```
→   call COMPILER(40)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

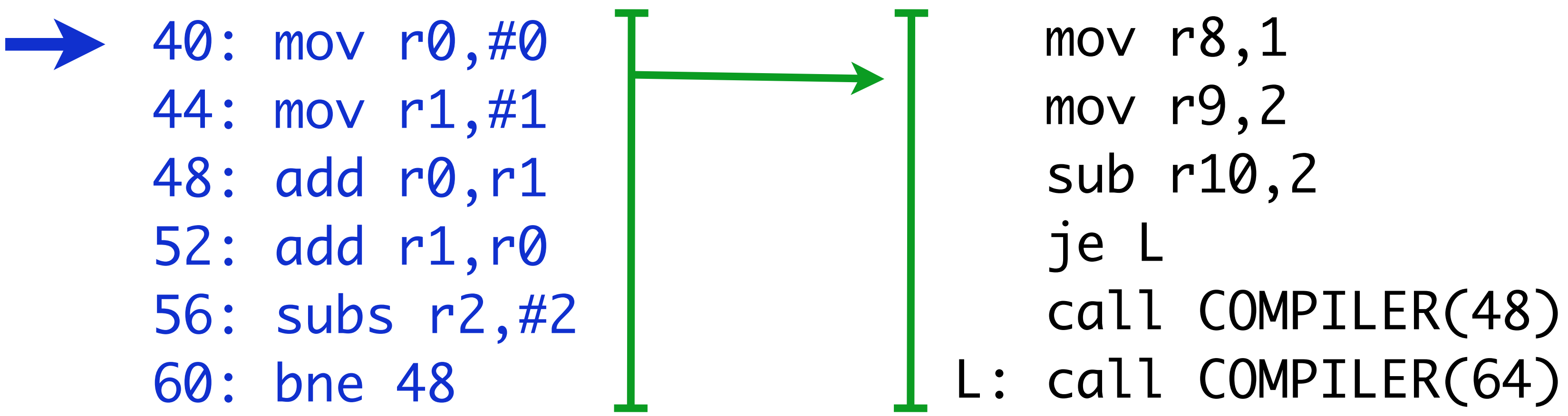
Native code: → **JIT compiler**

```
call COMPILER(40)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code: Native code: → **JIT compiler**



```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

```
    mov r8,1
    mov r9,2
    sub r10,2
    je L
    call COMPILER(48)
L:  call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

Native code: → JIT compiler

```
   mov r8,1
   mov r9,2
   sub r10,2
   je L
   call COMPILER(48)
L: call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

Native code:

JIT compiler

```
→ 40: mov r0,#0
  44: mov r1,#1
  48: add r0,r1
  52: add r1,r0
  56: subs r2,#2
  60: bne 48
```

```
→    mov r8,1
     mov r9,2
     sub r10,2
     je L
     call COMPILER(48)
  L: call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

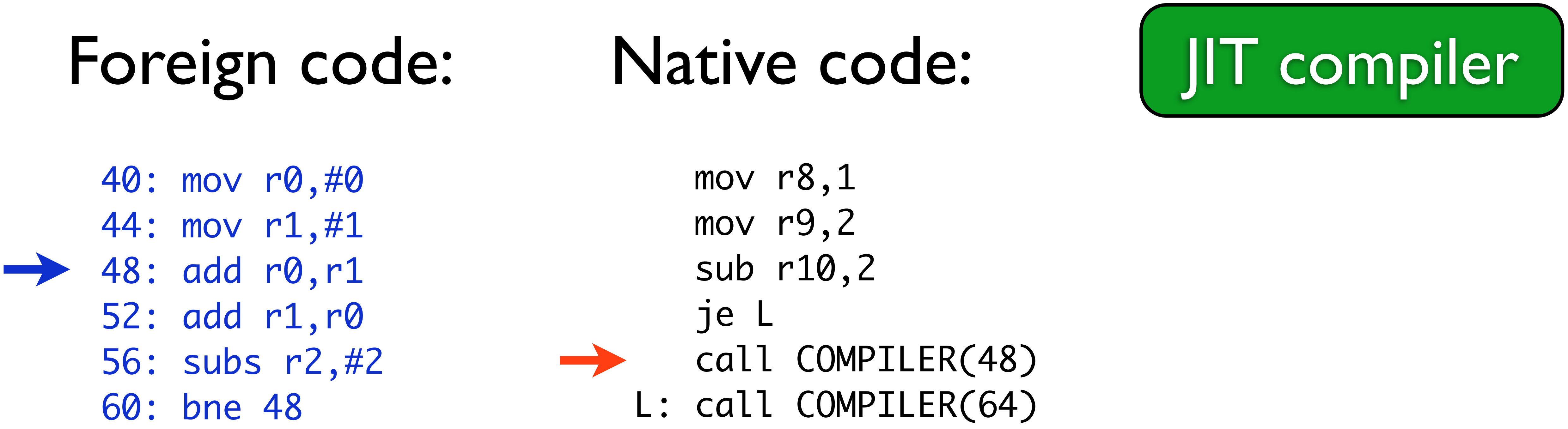
Native code:

```
   mov r8,1
   mov r9,2
   sub r10,2
   je L
   call COMPILER(48)
L: call COMPILER(64)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation



Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

Native code:

```
   mov r8,1
   mov r9,2
   sub r10,2
   je L
   call COMPILER(48)
L: call COMPILER(64)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

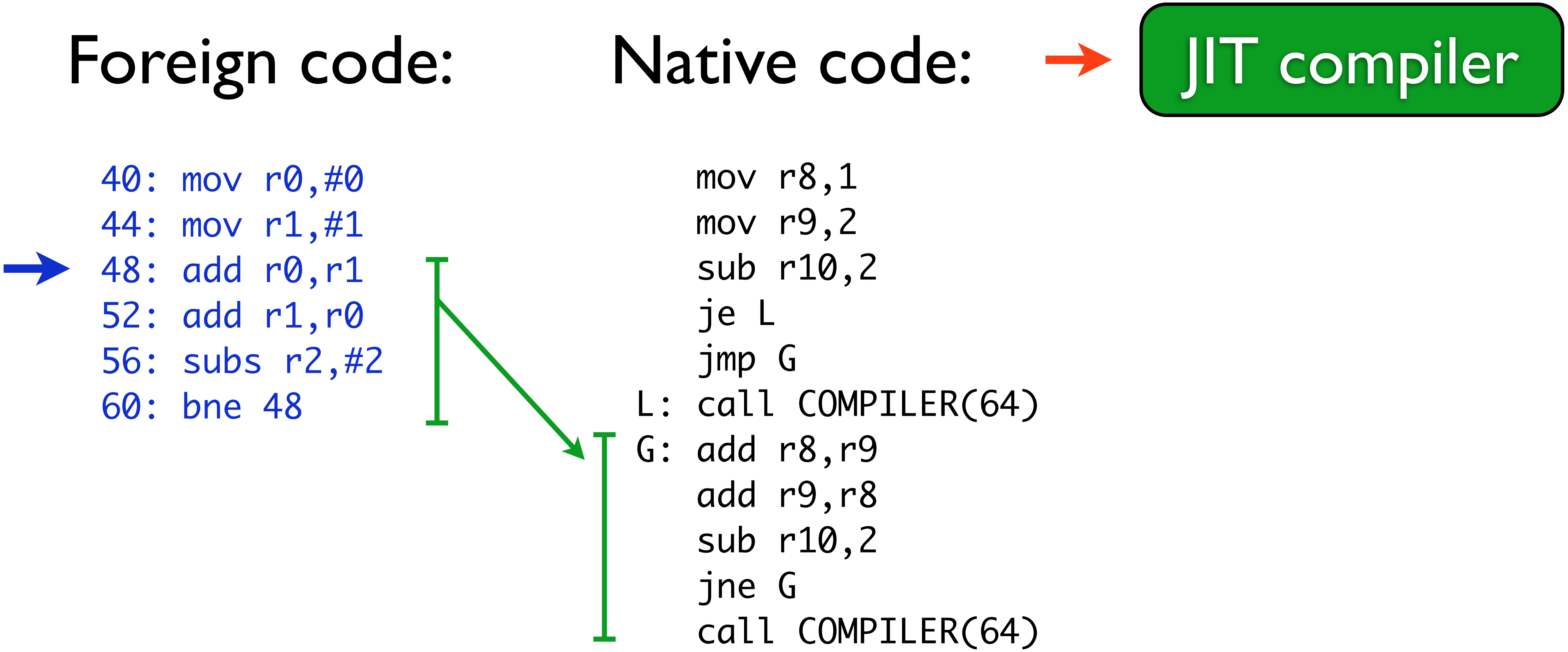
Foreign code: Native code: → JIT compiler

```
   40: mov r0,#0
   44: mov r1,#1
→  48: add r0,r1
   52: add r1,r0
   56: subs r2,#2
   60: bne 48
```

```
   mov r8,1
   mov r9,2
   sub r10,2
   je L
   call COMPILER(48)
L: call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation



Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

Native code:

JIT compiler

```
    mov r8,1
    mov r9,2
    sub r10,2
    je L
    jmp G
L:  call COMPILER(64)
G:  add r8,r9
    add r9,r8
    sub r10,2
    jne G
    call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code: Native code: → **JIT compiler**

```
→ 40: mov r0,#0
  44: mov r1,#1
  48: add r0,r1
  52: add r1,r0
  56: subs r2,#2
  60: bne 48
```

```
     mov r8,1
     mov r9,2
     sub r10,2
     je L
     jmp G
  L: call COMPILER(64)
  G: add r8,r9
     add r9,r8
     sub r10,2
     jne G
     call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
→ 40: mov r0,#0
  44: mov r1,#1
  48: add r0,r1
  52: add r1,r0
  56: subs r2,#2
  60: bne 48
```

Native code:

```
       mov r8,1
       mov r9,2
       sub r10,2
       je L
       jmp G
    L: call COMPILER(64)
  → G: add r8,r9
       add r9,r8
       sub r10,2
       jne G
       call COMPILER(64)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

Native code:

```
   mov r8,1
   mov r9,2
   sub r10,2
   je L
   jmp G
L: call COMPILER(64)
G: add r8,r9
   add r9,r8
   sub r10,2
   jne G
   call COMPILER(64)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
   40: mov r0,#0
   44: mov r1,#1
→  48: add r0,r1
   52: add r1,r0
   56: subs r2,#2
   60: bne 48
```

Native code:

```
       mov r8,1
       mov r9,2
       sub r10,2
       je L
       jmp G
    L: call COMPILER(64)
→   G: add r8,r9
       add r9,r8
       sub r10,2
       jne G
       call COMPILER(64)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
40: mov r0,#0
44: mov r1,#1
48: add r0,r1
52: add r1,r0
56: subs r2,#2
60: bne 48
```

Native code:

```
   mov r8,1
   mov r9,2
   sub r10,2
   je L
   jmp G
L: call COMPILER(64)
G: add r8,r9
   add r9,r8
   sub r10,2
   jne G
   call COMPILER(64)
```

JIT compiler

- blocks of foreign code is translated into native code
- eventually only native code is run

JIT animation

Foreign code:

```
 40: mov r0,#0
 44: mov r1,#1
 48: add r0,r1
 52: add r1,r0
 56: subs r2,#2
 60: bne 48
→
```

Native code: → **JIT compiler**

```
    mov r8,1
    mov r9,2
    sub r10,2
    je L
    jmp G
L:  call COMPILER(64)
G:  add r8,r9
    add r9,r8
    sub r10,2
    jne G
    call COMPILER(64)
```

- blocks of foreign code is translated into native code
- eventually only native code is run

Block translation

block in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

```
L:
  add r0,r1
  add r1,r0
  subs r2,#2
  bne L
```

Block translation

block in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

```
L:
  add r0,r1
  add r1,r0
  subs r2,#2
  bne L
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
inc pc,4
mov A,r1
mov B,r0
add A,B
mov r0,A
inc pc,4
...
skip 4
...
inc pc,4
```

Block translation

block in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

```
L:
   add r0,r1
   add r1,r0
   subs r2,#2
   bne L
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
inc pc,4
mov A,r1
mov B,r0
add A,B
mov r0,A
inc pc,4
...
skip 4
...
inc pc,4
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
mov A,r1
mov B,r0
add A,B
mov r0,A
...
skip 3
...
inc pc,16
```

Block translation

block in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

```
L:
   add r0,r1
   add r1,r0
   subs r2,#2
   bne L
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
inc pc,4
mov A,r1
mov B,r0
add A,B
mov r0,A
inc pc,4
...
skip 4
...
inc pc,4
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
mov A,r1
mov B,r0
add A,B
mov r0,A
...
skip 3
...
inc pc,16
```

block correct, but step-by-step instruction equivalence lost

Block translation

block in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

```
L:
   add r0,r1
   add r1,r0
   subs r2,#2
   bne L
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
inc pc,4
mov A,r1
mov B,r0
add A,B
mov r0,A
inc pc,4
...
skip 4
...
inc pc,4
```

```
mov A,r0
mov B,r1
add A,B
mov r0,A
mov A,r1
mov B,r0
add A,B
mov r0,A
...
skip 3
...
inc pc,16
```

```
mov eax,r8
mov ebx,r9
add eax,ebx
mov r8,eax
...
```

block correct, but step-by-step instruction equivalence lost

New translations

New translations to synthesise:

list_translate : word32 list → bytecode list

optimize : bytecode list → bytecode list

compile : (bytecode list) x env → x86 instructions

where env is information of where previously compiled code is located.

Produce JIT compiler following Myreen [POPL'10]

Problem

Invariant:

- executing the generate x86 code has the effect of emulating some steps of the ARM code.

Problem

Invariant:

- executing the generate x86 code has the effect of emulating some steps of the ARM code.
- precise invariant relates ARM code (in memory) with generated x86 code.
- ... what about self-modification?

memcpy

Memory of emulated code:

```
40: ldr r8,[r9],#4
44: str r8,[r10],#4
48: subs r11
52: bne 40
```

Incorrect generated code:

```
L: mov r8,[r9]
   add r9,4
   mov [r10],r8
   add r10,4
   dec r11
   jne L
```

memcpy

Memory of emulated code:

```
40: ldr r8,[r9],#4
44: str r8,[r10],#4
48: subs r11
52: bne 40
```

Incorrect generated code:

```
L: mov r8,[r9]
   add r9,4
   mov [r10],r8
   add r10,4
   dec r11
   jne L
```

store instruction modifies memory of emulated code

memcpy

Memory of emulated code:

```
40: ldr r8,[r9],#4
44: str r8,[r10],#4
48: subs r11
52: bne 40
```

emulated code may change as a result

Incorrect generated code:

```
L: mov r8,[r9]
   add r9,4
   mov [r10],r8
   add r10,4
   dec r11
   jne L
```

store instruction modifies memory of emulated code

memcpy

Memory of emulated code:

```
40: ldr r8,[r9],#4
44: str r8,[r10],#4
48: subs r11
52: bne 40
```

emulated code may change as a result

Incorrect generated code:

```
L: mov r8,[r9]
   add r9,4
   mov [r10],r8
   add r10,4
   dec r11
   jne L
```

store instruction modifies memory of emulated code

generated code can become out-of-date

memcpy

Memory of emulated code:

```
40: ldr r8,[r9],#4
44: str r8,[r10],#4
48: subs r11
52: bne 40
```

emulated code may change as a result

Incorrect generated code:

```
L: mov r8,[r9]
   add r9,4
   mov [r10],r8
   add r10,4
   dec r11
   jne L
```

need different inv or runtime checks

store instruction modifies memory of emulated code

generated code can become out-of-date

Timings and trade-offs

Invariant options:

- assume no self-modification (fast code)
- insert checks, erase out-of-date code (slower)

Fib example:

fib(200,000,000) using JIT runs in 0.7 seconds

(directly x86-complied C runs only 7x faster)

Binary translation

One-off translation

Why not whole-program translation instead of per-block translation?

Can be done ahead of time (once only)

One-off translation

Why not whole-program translation instead of per-block translation?

Can be done ahead of time (once only)

Difficulties:

- what to do about self-modification?
- what is code, what is data?
- where do pointer jumps go?

Obvious route

program in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

Requires a more expressive bytecode, and more complicated verified compiler...

Obvious route

program in foreign code → equivalent bytecode → optimised bytecode → implementation in 64-bit x86

Requires a more expressive bytecode, and more complicated verified compiler...

Better approach: translation validation can produce better code and is easier to implement.

Producing good code

- Ideal translation:

ARM

```
   mov r0,#0
   mov r1,#1
L: add r0,r1
   add r1,r0
   subs r2,#2
   bne L
```

x86

```
   mov eax,0
   mov ebx,1
L: add eax,ebx
   add ebx,eax
   sub ecx,2
   jne L
```

- Translation validation can prove these equiv.

Translation validation

Part 1: decompilation

```
   mov r0,#0
   mov r1,#1
L: add r0,r1
   add r1,r0
   subs r2,#2
   bne L
```

Function:

```
f(r2) = g(0,1,r2)

g(r0,r1,r2) =
  let r0 = r0 + r1 in
  let r1 = r1 + r0 in
  let r2 = r2 - 2 in
    if r2 = 0 then (r0,r1,r2)
    else g(r0,r1,r2)
```

Theorem relating f with code

```
{ R0 r0 * R1 r1 * R2 r2 * PC p }
 p: arm_code
{ let (r0,r1,r2) = f(r2) in
    R0 r0 * R1 r1 * R2 r2 * PC (p+24) }
```

Translation validation

Part 2: proof-producing synthesis

To synthesise (x86) code for f:

1. generate code for f (without proof)
2. decompile generate code into f’
3. automatically prove f = f’

Result: certificate thm

Theorem: behaviour of ARM is f:

```
{ R0 r0 * R1 r1 * R2 r2 * PC p }
 p: arm_code
{ let (r0,r1,r2) = f(r2) in
    R0 r0 * R1 r1 * R2 r2 * PC (p+24) }
```

Theorem: behaviour of x86 is f:

```
{ EAX a * EBX b * ECX c * EIP p }
 p: x86_code
{ let (a,b,c) = f(c) in
    EAX a * EBX b * ECX c * EIP (p+20) }
```

Fib example

Translation validation:

fib(200,000,000) runs in 0.1 seconds

(matches speed of directly x86-complied C)

Caveat: translation validation not always applicable

Concluding remarks

Comparison

Different approaches:

- direct interpretation: simple invariant
  - fib(200,000,000) in 48 seconds
- JIT compilation: complicated invariant
  - fib(200,000,000) in 0.7 seconds
- one-off binary translation: simple if applicable
  - fib(200,000,000) in 0.1 seconds

Summary

Aim: construct different verified emulators for ARMv4 running on 64-bit x86.

This project is still work in progress.