Program Synthesis for Forth

Forth Day 2012

Computer Science
UC Berkeley

Ras Bodik
Mangpo Phitchaya Phothilimthana
Tikhon Jelvis
Rohin Shah

Synthesis with "sketches"

Extend your language with two constructs

spec:

```
int foo (int x) {
    return x + x;
}
```

$\phi(x, y)\colon y = \mathbf{foo}(x)$

sketch:

```
int bar (int x) implements foo {
    return x << ??;
}
```

?? substituted with an int constant meeting ϕ

result:

```
int bar (int x) implements foo {
    return x << 1;
}
```

instead of **`implements`**, assertions over safety properties can be used

Example: Parallel Matrix Transpose

Example: 4x4-matrix transpose with SIMD

a functional (executable) specification:

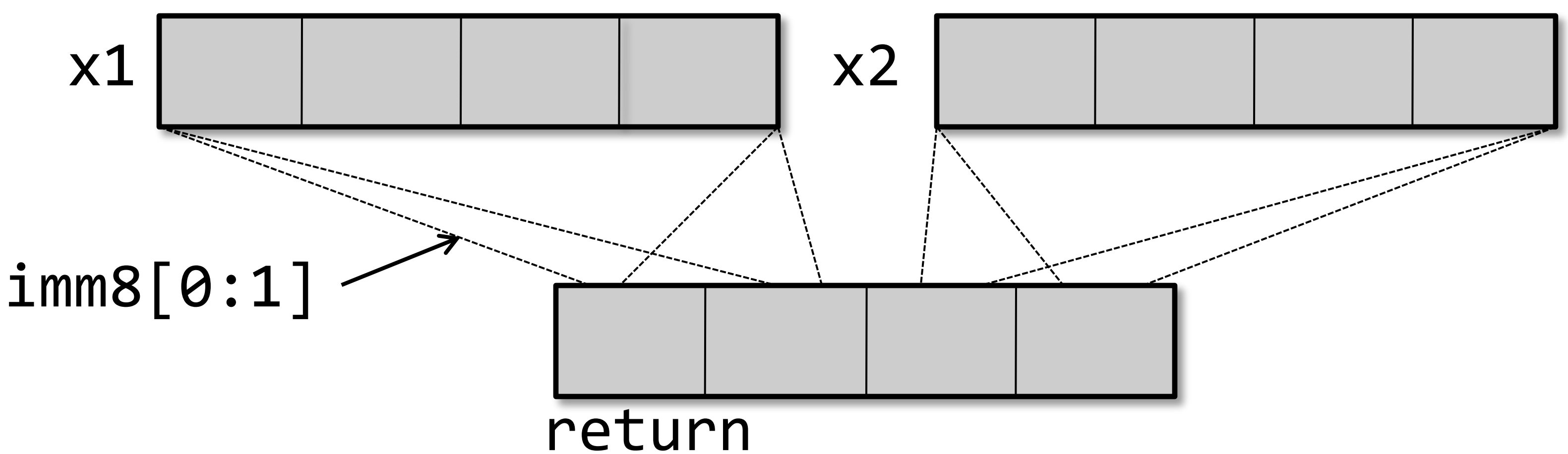
```
int[16] transpose(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

This example comes from a Sketch grad-student contest

Implementation idea: parallelize with SIMD

Intel SHUFP (shuffle parallel scalars) SIMD instruction:

```
return = shufps(x1, x2, imm8 :: bitvector8)
```



High-level insight of the algorithm designer

Matrix M transposed in two shuffle phases

Phase 1: shuffle M into an intermediate matrix S with some number of `shufps` instructions

Phase 2: shuffle S into an result matrix T with some number of `shufps` instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.

The SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;

  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);
  …
  S[??::4] = shufps(M[??::4], M[??::4], ??);

  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);
  …
  T[??::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

Phase 1 (the S assignments)

Phase 2 (the T assignments)

The SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

```
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
  S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
  S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
  S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
  T[4::4]   = shufps(S[11::4],  S[1::4],  10111100b);
  T[12::4]  = shufps(S[3
  T[8::4]   = shufps(S[4
  T[0::4]   = shufps(S[1
}
```

From the contestant email:
Over the summer, I spent about 1/2 a day manually figuring it out.
Synthesis time: <5 minutes.

Demo: transpose on Sketch

Try Sketch online at http://bit.ly/sketch-language

Inductive Synthesis,
Phrased as Constraint Solving

What to do with a program as a formula?

Assume a formula $S_P(x,y)$ which holds iff program P(x) outputs value y

program: `f(x) { return x + x }`

formula: $S_f(x, y)$: $y = x + x$

This formula is created as in program verification with concrete semantics [CMBC, Java Pathfinder, …]

With program as a formula, solver is versatile

Solver as an **interpreter**: given x, evaluate f(x)

$S(x, y) \wedge x = 3$ solve for y $\boldsymbol{y \mapsto 6}$

Solver as a program **inverter**: given f(x), find x

$S(x, y) \wedge y = 6$ solve for x $\boldsymbol{x \mapsto 3}$

This solver "bidirectionality" enables synthesis

Search of candidates as constraint solving

$S_P(x, h, y)$ holds iff sketch $P[h](x)$ outputs y.

```
spec(x) { return x + x }
sketch(x) { return x << ?? }
```

$S_{sketch}(x, y, h)$: $y = x * 2^h$

The solver computes h, thus synthesizing a program correct for the given x (here, x=2)

$S_{sketch}(x, y, h) \wedge x = 2 \wedge y = 4$ solve for h $\boldsymbol{h} \mapsto \mathbf{1}$

Sometimes h must be constrained on several inputs

$S(x_1, y_1, h) \wedge x_1 = 0 \wedge y_1 = 0 \wedge$
$S(x_2, y_2, h) \wedge x_2 = 3 \wedge y_2 = 6$ solve for h $\boldsymbol{h} \mapsto \mathbf{1}$

Inductive synthesis

Our constraints encode **inductive synthesis:**

- We ask for a program P correct on a few inputs.
- We hope (or test, verify) that P is correct on rest of inputs.

Synthesis for Forth and ArrayForth

Applications of synthesis for ArrayForth

Synthesizing optimal code

Input: unoptimized code (the spec)

Search space of all programs

Synthesizing optimal library code

Input: sketch + spec

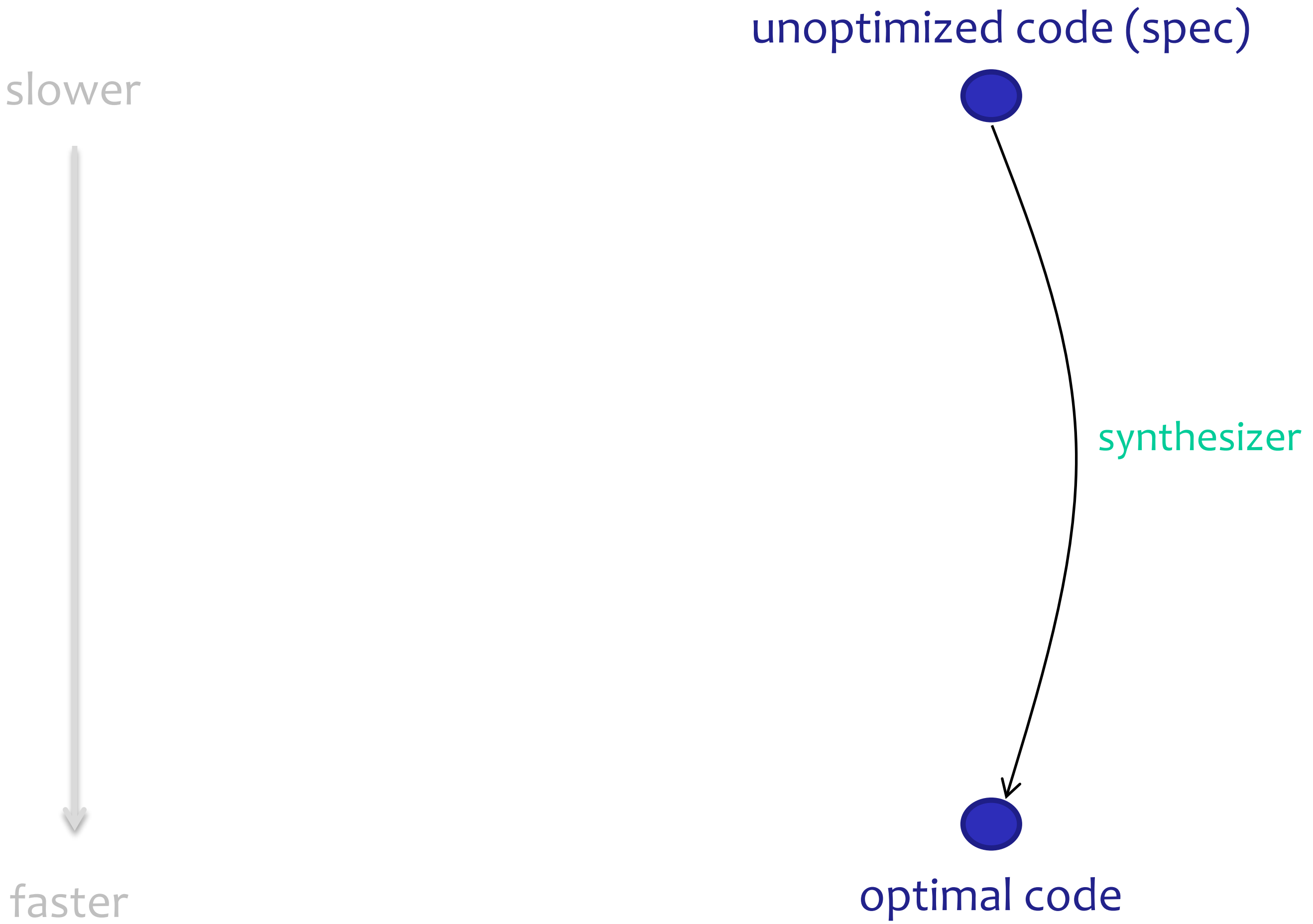
Search completions of the sketch

Synthesizing communication code for GreenArray

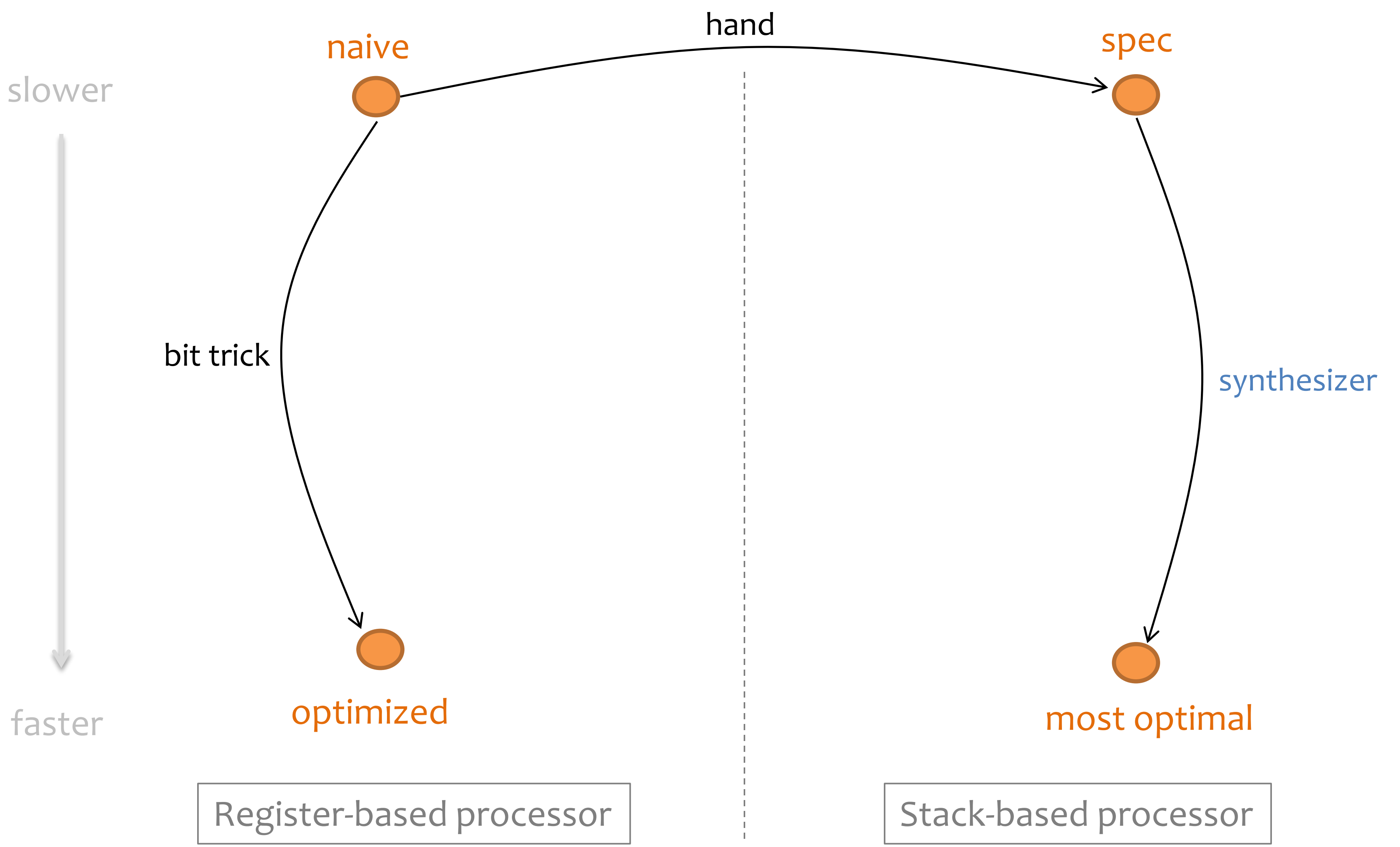
Input: program with virtual channels

Compile using synthesis

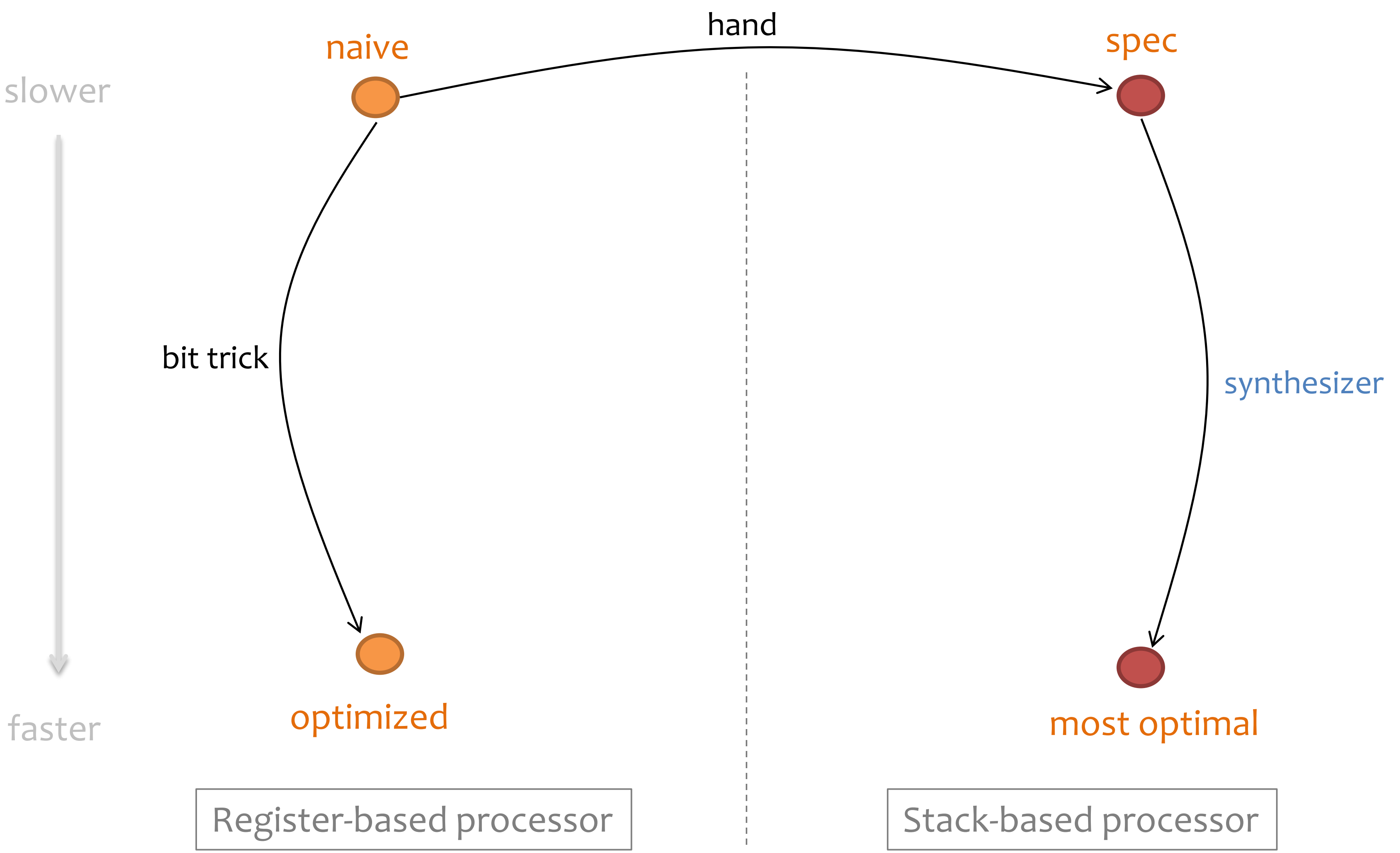
1) Synthesizing optimal code



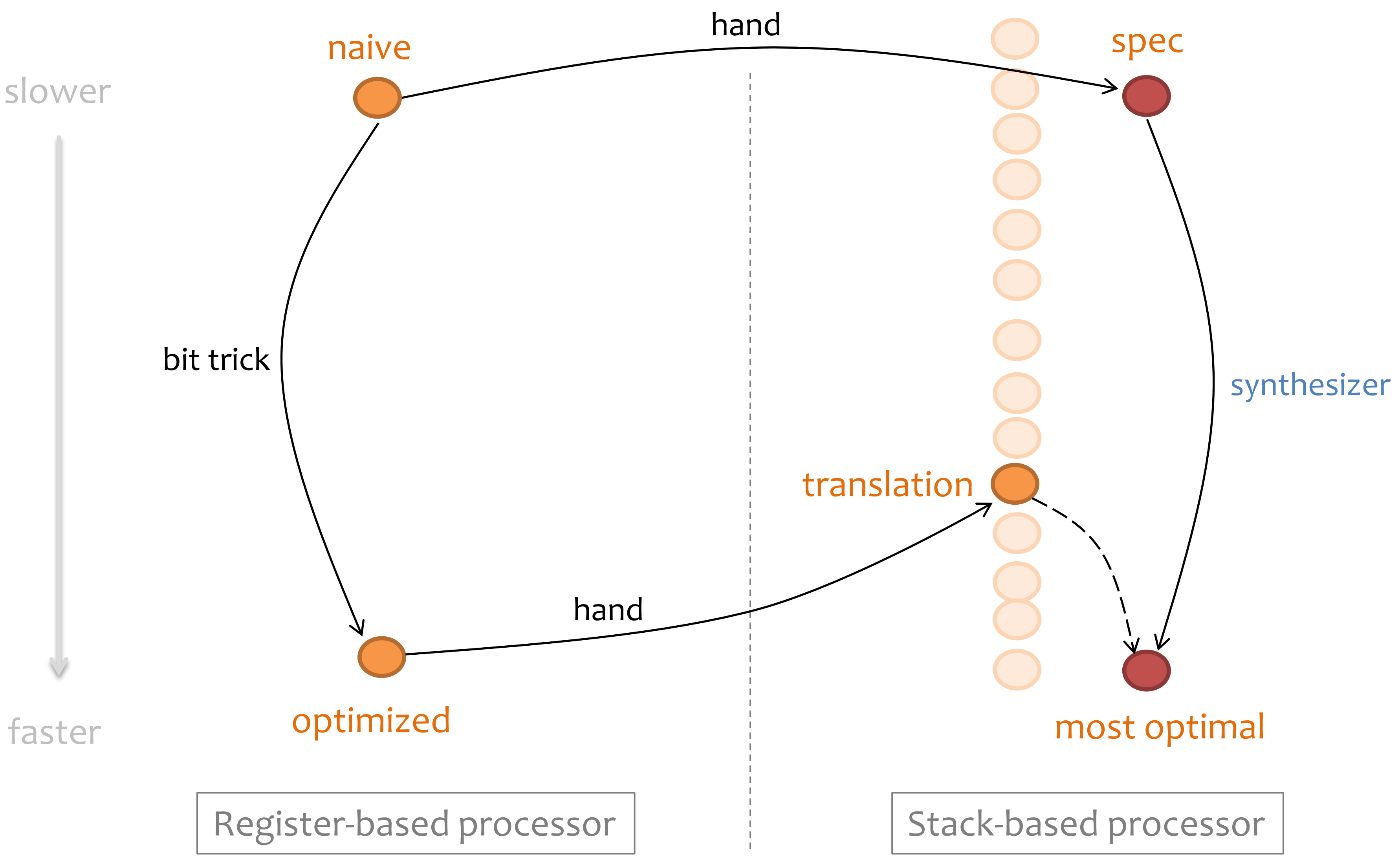
Our Experiment



Our Experiment



Comparison



slower
faster
naive
hand
spec
bit trick
synthesizer
translation
hand
optimized
most optimal
Register-based processor
Stack-based processor

Preliminary Synthesis Times

Synthesizing a program with

8 unknown instructions

takes 5 second to 5 minutes

Synthesizing a program up to

~25 unknown instructions

within 50 minutes

Preliminary Results

Program	Description	Approx. Speedup	Code length reduction
x – (x & y)	Exclude common bits	5.2x	4x
~(x – y)	Negate difference	2.3x	2x
x | y	Inclusive or	1.8x	1.8x
(x + 7) & -8	Round up to multiple of 8	1.7x	1.8x
(x & m) | (y & ~m)	Replace **x** with **y** where bits of **m** are 1's	2x	2x
(y & m) | (x & ~m)	Replace **y** with **x** where bits of **m** are 1's	2.6x	2.6x
x' = (x & m) | (y & ~m) y' = (y & m) | (x & ~m)	Swap **x** and **y** where bits of **m** are 1's	2x	2x

Code Length

Program	Original Length	Output Length
x – (x & y)	8	2
~(x – y)	8	4
x | y	27	15
(x + 7) & -8	9	5
(x & m) | (y & ~m)	22	11
(y & m) | (x & ~m)	21	8
x’ = (x & m) | (y & ~m) y’ = (y & m) | (x & ~m)	43	21

2) Synthesizing optimal library code

Input:

Sketch: program with holes to be filled

Spec: program in any programing language

Output:

Complete program with filled holes

Example: Integer Division by Constant

Naïve Implementation:

Subtract divisor until reminder < divisor.

# of iterations = output value **Inefficient!**

Better Implementation:

$$\text{quotient} = (M * n) >> s$$

n - input

M - "magic" number

s - shifting value

M and s depend on the number of bits and constant divisor.

Example: Integer Division by 3

Sketch in ArrayForth:

```
: div3 ?? a! 0 17 for +* unext
push dup or pop
?? for +* unext a ;
```

Spec in C:

```
int div3(int n) {
        return n/3;
}
```

Preliminary Results

Program	Solution	Synthesis Time (s)	Verification Time (s)	# of Pairs
x/3	(43691 * x) >> 17	2.3	7.6	4
x/5	(209716 * x) >> 20	3	8.6	6
x/6	(43691 * x) >> 18	3.3	6.6	6
x/7	(149797 * x) >> 20	2	5.5	3
deBruijn: Log_2x (x is power of 2)	deBruijn = 46, Table = {7, 0, 1, 3, 6, 2, 5, 4}	3.8	N/A	8

Note: these programs work for 18-bit number except Log2x is for 8-bit number.

3) Communication Code for GreenArray

Synthesize communication code between nodes

Interleave communication code with computational code such that

- There is no deadlock.
- The runtime of the synthesized program is minimized.

Future Roadmap

Language?

Language Design

- Good for partitioning
- Easy to compile to arrayForth

Partitioning

- Minimize number of communication
- Each block fits in each node

Placement & Communication

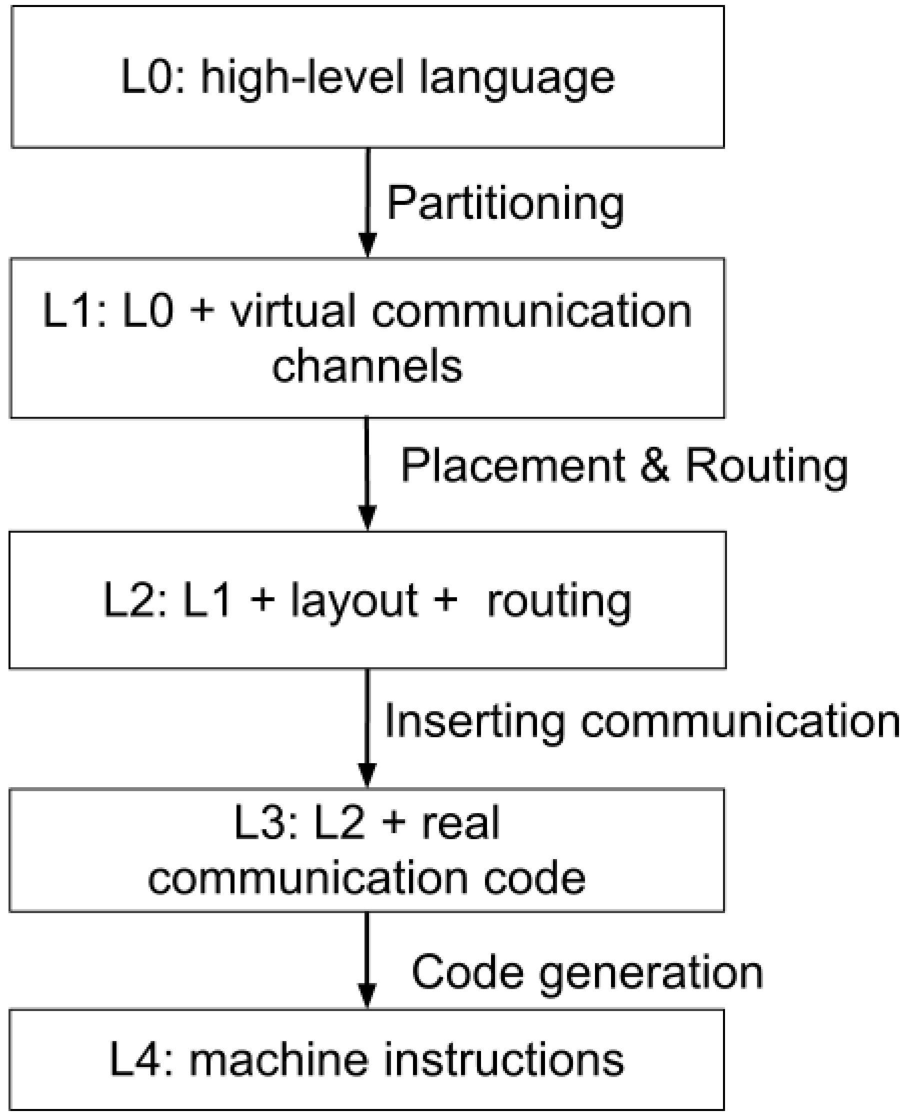
- Minimize communication cost
- Reason about I/O pins

Comp1
Comp2
Comp3
Send X
Comp4
Recv Y
Comp5

Scheduling & Optimization

- Order that does not break dependency
- No Deadlock
- Find the fastest schedule

Project Pipeline



Preliminary Results #1 (backup)

Program	Approx Runtime (ns)		Program Length	
	Original	Optimized	Original	Optimized
x – (x & y)	15.5	3	8	2
~(x – y)	14	6	8	4
x | y	9	5	27	15
(x + 7) & -8	24	14	9	5
(x & m) | (y & ~m)	33	16.5	22	11
(y & m) | (x & ~m)	31.5	12	21	8
x' = (x & m) | (y & ~m) y' = (y & m) | (x & ~m)	64.5	31.5	43	21

Preliminary Results #1 (backup)

Program	Original Program	Synthesized Program
x – (x & y)	over and - 1 . + . +	- and
~(x – y)	- 1 . + . + -	over - . +
x | y	over over or a! and a or	over – and . +
(x + 7) & -8	7 . + 8 – 1 . + and	7 . + 262136 and
(y & m) | (x & ~m)	a! over over a - and push a and pop over over or push and pop or push	a! over over or a and over or push
(x & m) | (y & ~m)	a and push a - and pop over over or push and pop or pop	over or a and or dup pop
x’ = (x & m) | (y & ~m) y’ = (y & m) | (x & ~m)	a! over over a - and push a and pop over over or push and pop or push a and push a - and pop over over or push and pop or pop	a! over over or a and over or push over or a and or dup pop

Log Base 2 of Power of 2 (backup)

Compute lg x, where x is a power of 2.

```
const uint64_t deBruijn = 0x022fdd63cc95386d;
const unsigned int convert[64] =
{   0,  1,  2, 53,  3,  7, 54, 27,
    4, 38, 41,  8, 34, 55, 48, 28,
   62,  5, 39, 46, 44, 42, 22,  9,
   24, 35, 59, 56, 49, 18, 29, 11,
   63, 52,  6, 26, 37, 40, 33, 47,
   61, 45, 43, 21, 23, 58, 17, 10,
   51, 25, 36, 32, 60, 20, 57, 16,
   50, 31, 19, 15, 30, 14, 13, 12};

r = convert[(x*deBruijn) >> 58];
```

Sketch:
dup dup or a!
?? !+ ?? !+ ?? !+ ?? !+ ?? !+ ?? !+ ?? !+ ?? !+
?? a! 0 17 for +* unext
a 2/ 2/ 2/ 2/ 2/ 7 and a! @