Symbolic Crosschecking of Floating-Point and SIMD Code

Peter Collingbourne, Cristian Cadar, Paul H J Kelly

Department of Computing, Imperial College London

13 April, 2011

SIMD

- Single Instruction Multiple Data
- A popular means of improving the performance of programs by exploiting data level parallelism
- SIMD vectorised code operates over one-dimensional arrays of data called vectors

```
__m128 c = _mm_mul_ps(a, b);
                /* c = { a[0]*b[0], a[1]*b[1],
                          a[2]*b[2], a[3]*b[3] } */
```

- SIMD code is typically translated manually based on a reference scalar implementation
- Manually translating scalar code into an equivalent SIMD version is a difficult and error-prone task

SIMD and Floating Point

- SIMD vectorised code frequently makes intensive use of floating point arithmetic
- Developers have to reason about subtle floating point semantics:
  - Associativity
  - Distributivity
  - Precision
  - Rounding

Spot the Difference

Scalar

```
out[0] = x[0] * y[0] * z[0];
```

SIMD

```
outv = _mm_mul_ps(xv, _mm_mul_ps(yv, zv));
```

Scalar

```
out[0] = std::min(x[0], y[0]);
```

SIMD

```
outv = _mm_min_ps(xv, yv);
```

min and max are not commutative or associative in FP!

Scalar

```
out[0] = std::min(x[0], y[0]);
```

SIMD

```
outv = _mm_min_ps(xv, yv);
```

- SSE `_mm_min_ps`:

$$\texttt{min}(X, Y) = \texttt{Select}(X <_{\text{ord}} Y, X, Y)$$

- $X <_{\text{ord}} Y$ evaluates to `false` if either of X or Y is `NaN`

$$\begin{aligned} \texttt{min}(X, \texttt{NaN}) &= \texttt{NaN} \\ \texttt{min}(\texttt{NaN}, Y) &= Y \end{aligned}$$

$$\begin{aligned} \texttt{min}(\texttt{min}(X, \texttt{NaN}), Y) &= \texttt{min}(\texttt{NaN}, Y) &= Y \\ \texttt{min}(X, \texttt{min}(\texttt{NaN}, Y)) &= \texttt{min}(X, Y) \end{aligned}$$

min and max are not commutative or associative in FP!

Scalar

```
out[0] = std::min(x[0], y[0]);
```

SIMD

```
outv = _mm_min_ps(xv, yv);
```

- SSE `_mm_min_ps`:

$$\texttt{min}(X, Y) = \texttt{Select}(X <_{\text{ord}} Y, X, Y)$$

- $X <_{\text{ord}} Y$ evaluates to `false` if either of X or Y is `NaN`

$$\begin{aligned} \texttt{min}(X, \texttt{NaN}) &= \texttt{NaN} \\ \texttt{min}(\texttt{NaN}, Y) &= Y \end{aligned}$$

$$\begin{aligned} \texttt{min}(\texttt{min}(1, \texttt{NaN}), 200) &= \texttt{min}(\texttt{NaN}, 200) &= 200 \\ \texttt{min}(1, \texttt{min}(\texttt{NaN}, 200)) &= \texttt{min}(1, 200) &= 1 \end{aligned}$$

min and max are not commutative or associative in FP!

Scalar

```
out[0] = std::min(x[0], y[0]);
```

SIMD

```
outv = _mm_min_ps(xv, yv);
```

- SSE `_mm_min_ps`:

$$\texttt{min}(X, Y) = \texttt{Select}(X <_{\text{ord}} Y, X, Y)$$

- $X <_{\text{ord}} Y$ evaluates to `false` if either of X or Y is `NaN`

- libstdc++ `std::min`

$$\texttt{stl\_min}(X, Y) = \texttt{min}(Y, X)$$

- $out[0] = \texttt{min}(x[0], y[0])$
- $outv[0] = \texttt{min}(yv[0], xv[0])$

Symbolic Execution for SIMD

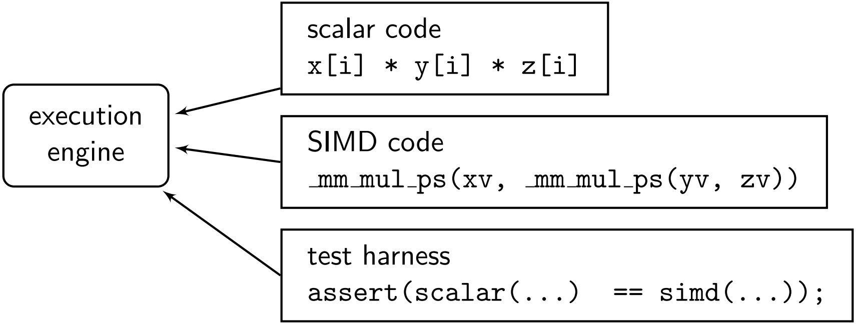
- A novel automatic technique based on symbolic execution for verifying that the SIMD version of a piece of code is equivalent to its (original) scalar version

- Symbolic execution can automatically explore multiple paths through the program
- Determines the feasibility of a particular path by reasoning about all possible values using a constraint solver

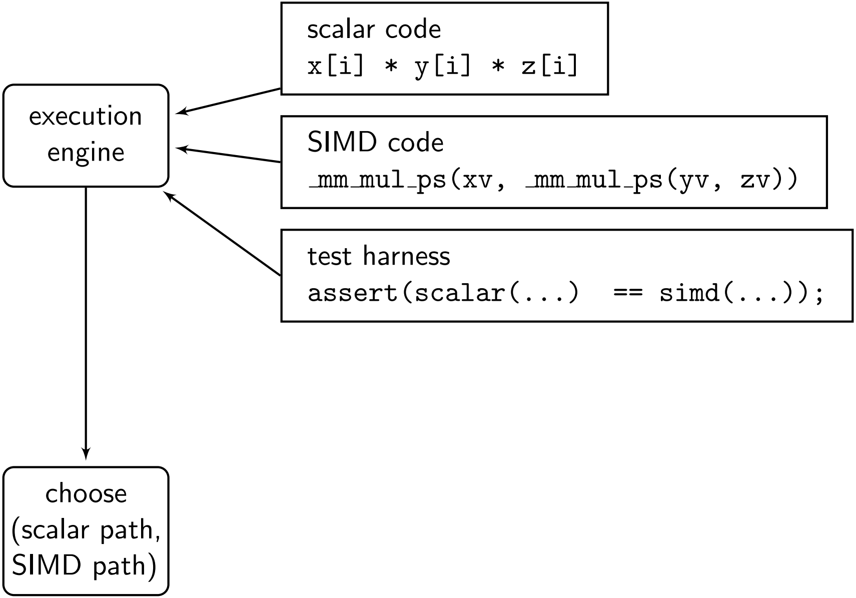
Challenges

- Huge number of paths involved in typical SIMD vectorisations
- The current generation of symbolic execution tools lack symbolic support for floating point and SIMD
  - Due to lack of available constraint solvers
  - (Recent development: floating point support in CBMC)

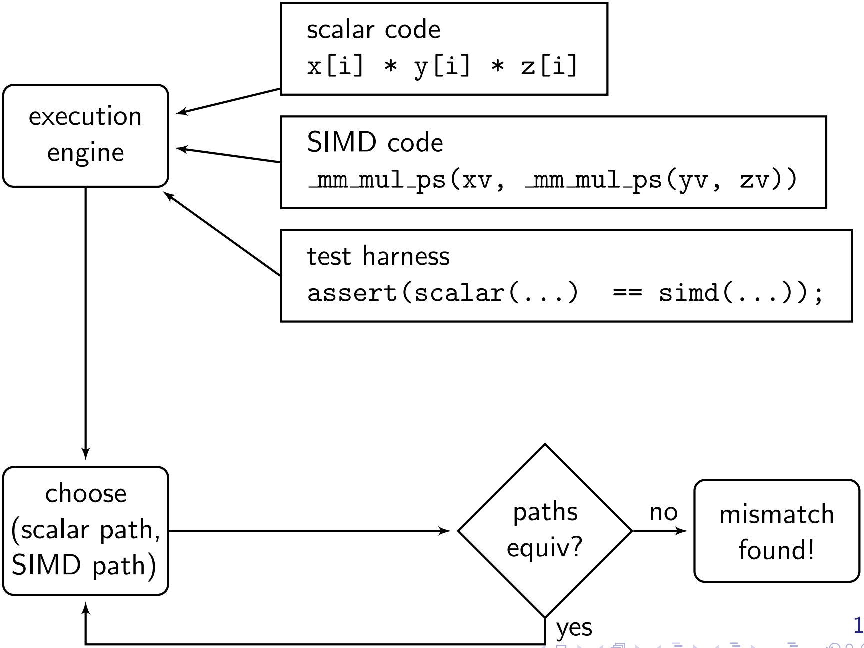
Architecture



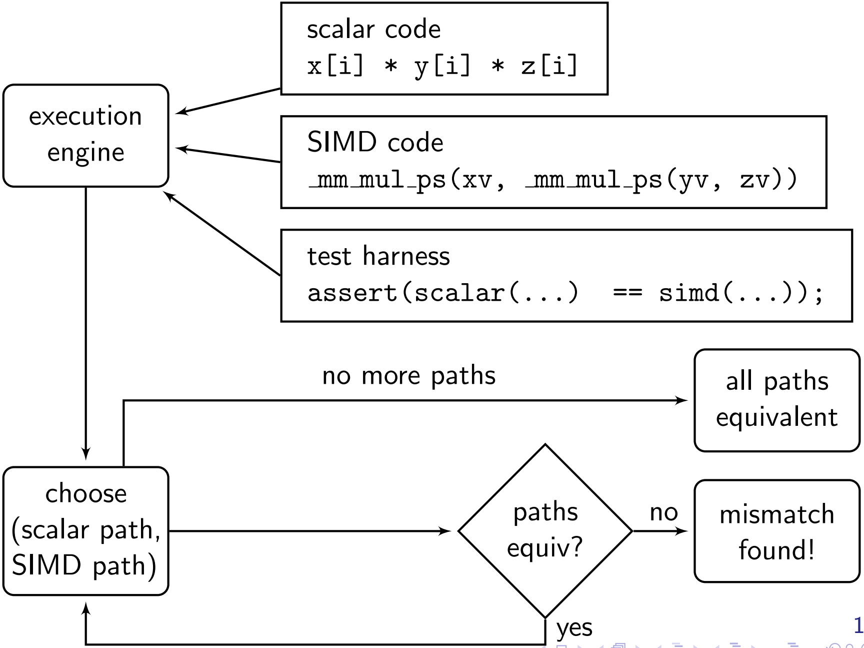
Architecture



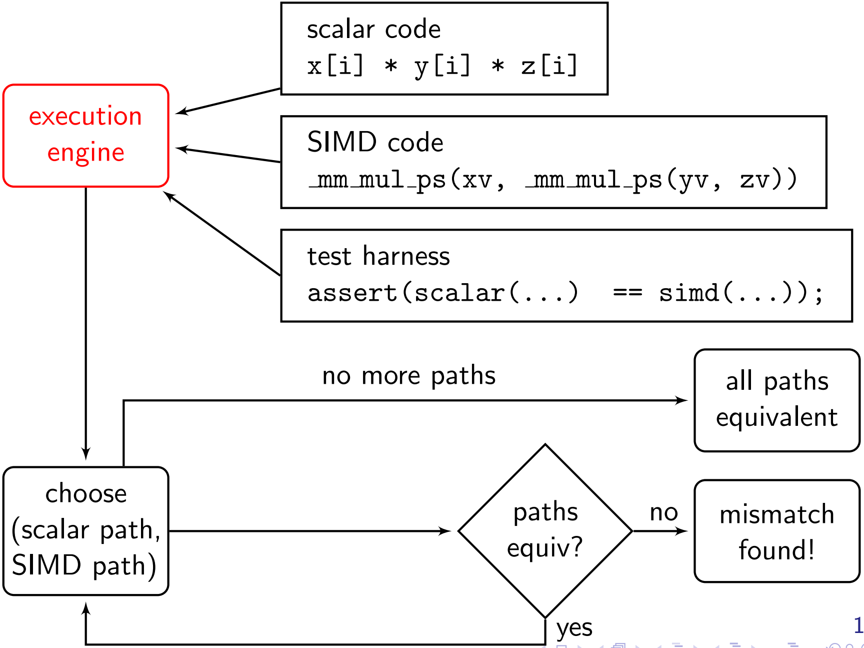
Architecture



Architecture



Architecture



Symbolic Execution – Operation

- Program runs on *symbolic input*, initially unconstrained
- Each variable may hold either a concrete or a symbolic value
- Symbolic value: an input dependent expression consisting of mathematical or boolean operations and symbols
  - For example, an integer variable `i` may hold a value such as $x + 3$
- When program reaches a branch depending on symbolic input
  - Determine feasibility of each side of the branch
  - If both feasible, *fork* execution and follow each path separately, adding corresponding constraints on each side

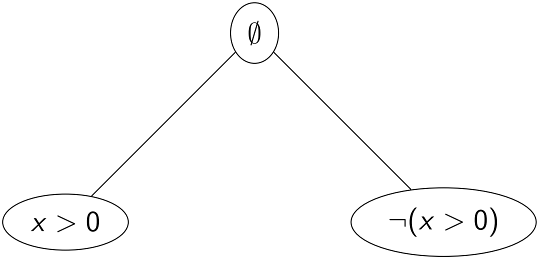
Symbolic Execution – Example

```
int x;
mksymbolic(x);

if (x > 0) {
  ...
} else {
  ...
}

if (x > 10) {
  ...
} else {
  ...
}
```

$\emptyset$

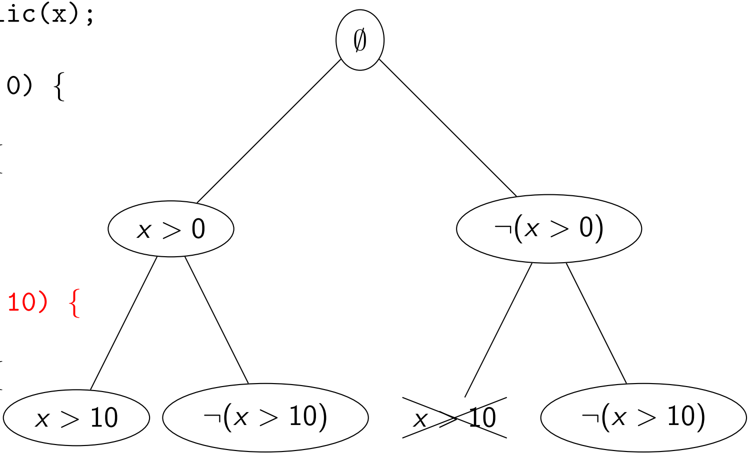
Symbolic Execution – Example

```
int x;
mksymbolic(x);

if (x > 0) {
   ...
} else {
   ...
}

if (x > 10) {
   ...
} else {
   ...
}
```



Symbolic Execution – Example

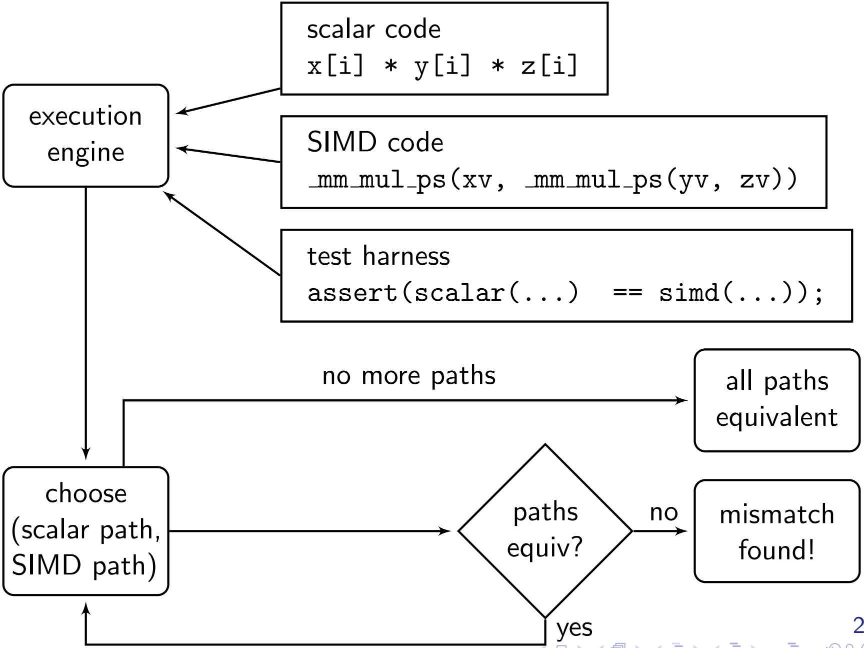
```
int x;
mksymbolic(x);

if (x > 0) {
   ...
} else {
   ...
}

if (x > 10) {
   ...
} else {
   ...
}
```



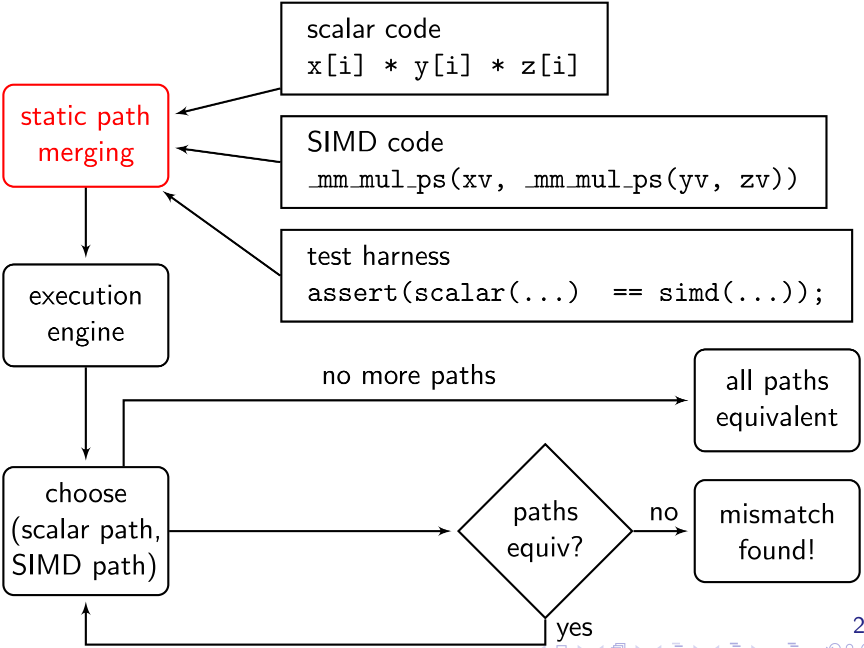
Challenges

- Huge number of paths involved in typical SIMD vectorisations
- The current generation of symbolic execution tools lack symbolic support for floating point and SIMD
  - Due to lack of available constraint solvers
  - (Recent development: floating point support in CBMC)

Architecture



Architecture



Static Path Merging

```
for (unsigned i = 0; i < N; ++i) {
    diff[i] = x[i]>y[i] ? x[i]-y[i] : y[i]-x[i];
}
```

- 2^N paths!

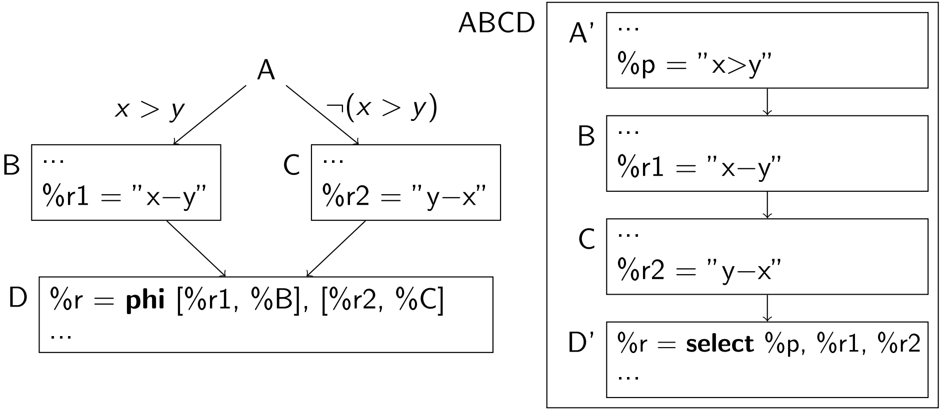
Static Path Merging

$$\text{diff}(x, y) = x > y \; ? \; x - y : y - x$$

$$\begin{array}{c} & A & \\ \swarrow^{x > y} & & \searrow^{\neg(x > y)} \\ B: \boxed{\begin{array}{l}\cdots \\ \%\text{r1} = \text{"x}-\text{y"}\end{array}} & & C: \boxed{\begin{array}{l}\cdots \\ \%\text{r2} = \text{"y}-\text{x"}\end{array}} \\ \searrow & & \swarrow \\ & D: \boxed{\begin{array}{l}\%\text{r} = \textbf{phi} \; [\%\text{r1}, \%\text{B}], [\%\text{r2}, \%\text{C}] \\ \cdots\end{array}} & \end{array}$$

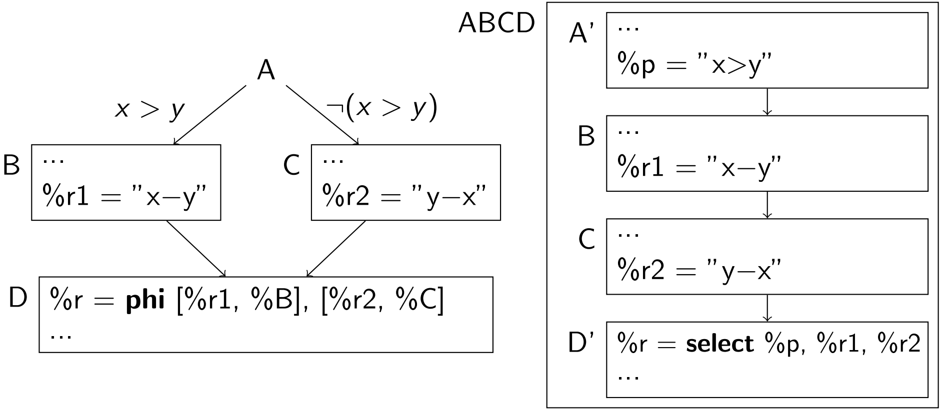
Static Path Merging

$$\mathrm{diff}(x, y) = x > y \ ? \ x - y : y - x$$



Static Path Merging

$$\mathrm{diff}(x, y) = x > y \ ?\ x - y : y - x$$



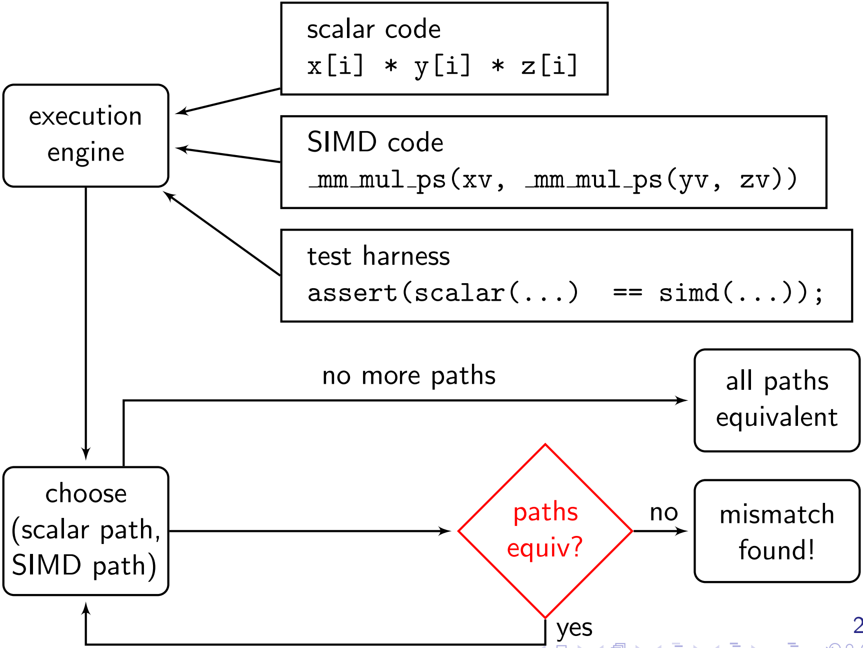
- morph benchmark, 16×16 matrix:

$$2^{256} \rightarrow 1$$

Challenges

- Huge number of paths involved in typical SIMD vectorisations
- The current generation of symbolic execution tools lack symbolic support for floating point and SIMD
  - Due to lack of available constraint solvers
  - (Recent development: floating point support in CBMC)

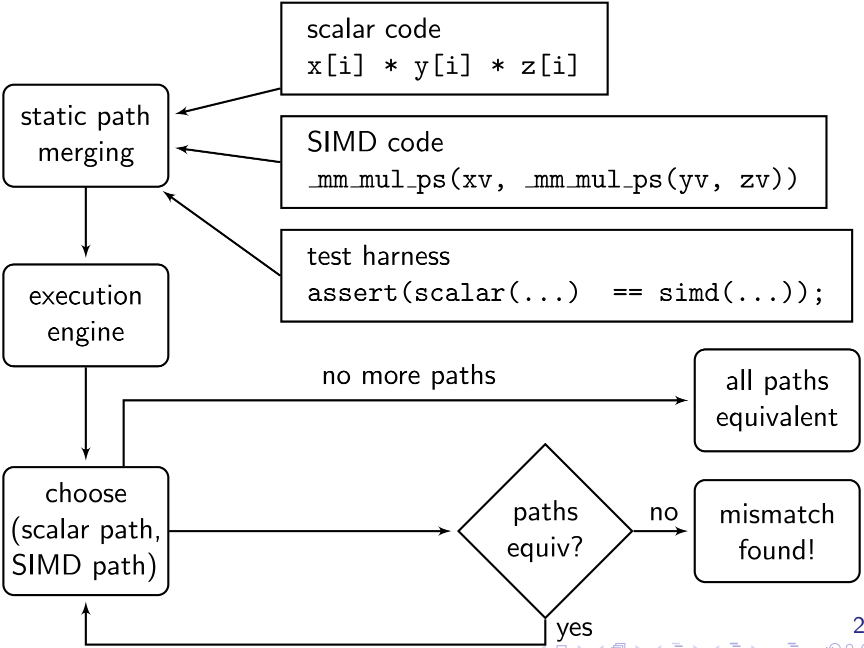
Architecture



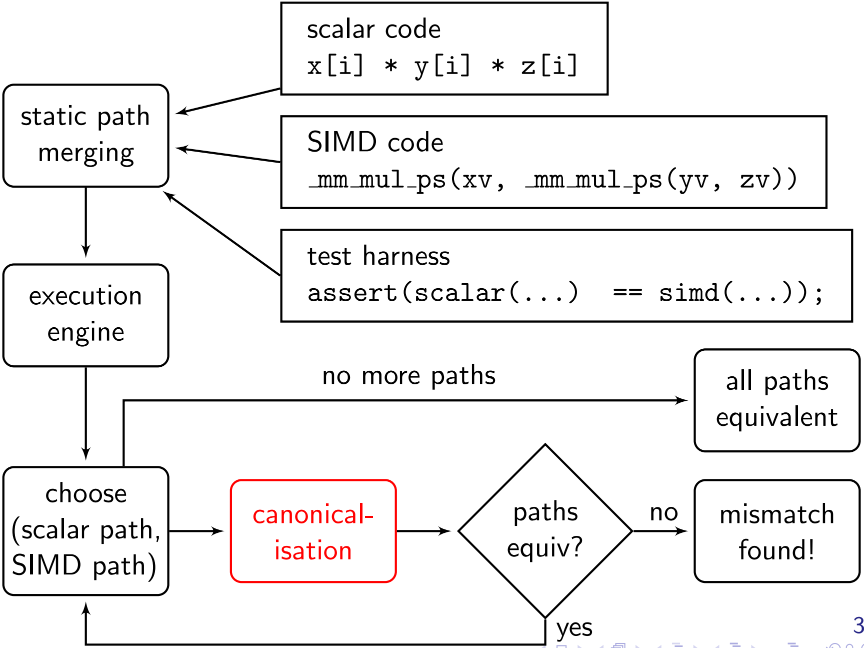
Technique

- The requirements for equality of two floating point expressions are harder to satisfy than for integers
- Usually, the two expressions need to be built up in the same way to be sure of equality
- We can check expression equivalence via simple expression matching!

Architecture



Architecture



Scalar/SIMD Implementation

```
void zlimit(int simd, float *src, float *dst,
            size_t size) {
   if (simd) {
      __m128 zero4 = _mm_set1_ps(0.f);
      while (size >= 4) {
         __m128 srcv = _mm_loadu_ps(src);
         __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
         __m128 dstv = _mm_and_ps(cmpv, srcv);
         _mm_storeu_ps(dst, dstv);
         src += 4; dst += 4; size -= 4;
      }
   }
   while (size) {
      *dst = *src > 0.f ? *src : 0.f;
      src++; dst++; size--;
   }
}
```

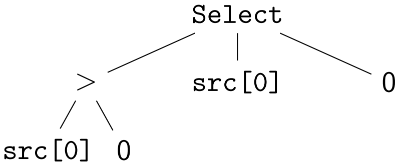
Scalar/SIMD Implementation

```
void zlimit(int simd, float *src, float *dst,
            size_t size) {
   if (simd) {
      __m128 zero4 = _mm_set1_ps(0.f);
      while (size >= 4) {
         __m128 srcv = _mm_loadu_ps(src);
         __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
         __m128 dstv = _mm_and_ps(cmpv, srcv);
         _mm_storeu_ps(dst, dstv);
         src += 4; dst += 4; size -= 4;
      }
   }
   while (size) {
      *dst = *src > 0.f ? *src : 0.f;
      src++; dst++; size--;
   }
}
```

Scalar/SIMD Implementation

```
while (size) {
   *dst = *src > 0.f ? *src : 0.f;
   src++; dst++; size--;
}
```

Scalar `dst[0]`

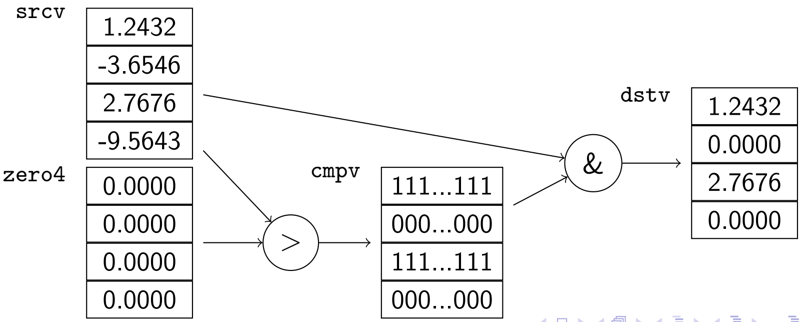


Scalar/SIMD Implementation

```
void zlimit(int simd, float *src, float *dst,
            size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
```

Scalar/SIMD Implementation

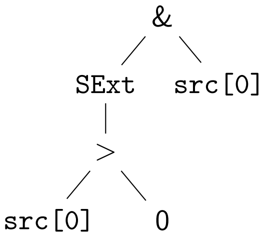
```
__m128 zero4 = _mm_set1_ps(0.f);
while (size >= 4) {
    __m128 srcv = _mm_loadu_ps(src);
    __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
    __m128 dstv = _mm_and_ps(cmpv, srcv);
    _mm_storeu_ps(dst, dstv);
    src += 4; dst += 4; size -= 4;
}
```



Scalar/SIMD Implementation

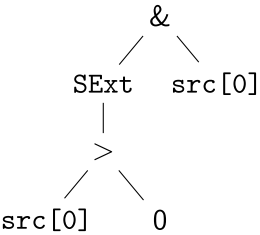
```
__m128 zero4 = _mm_set1_ps(0.f);
while (size >= 4) {
    __m128 srcv = _mm_loadu_ps(src);
    __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
    __m128 dstv = _mm_and_ps(cmpv, srcv);
    _mm_storeu_ps(dst, dstv);
    src += 4; dst += 4; size -= 4;
}
```

SIMD dst[0]

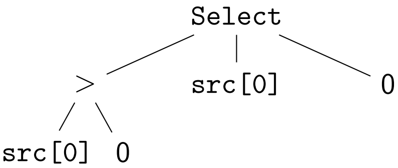


Scalar/SIMD Implementation

SIMD `dst[0]`



Scalar `dst[0]`



$$\mathtt{SExt}(P)\ \&\ X \rightarrow \mathtt{Select}(P, X, 0)$$

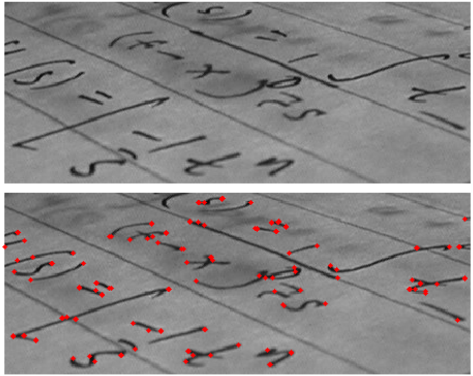
- One of our 18 canonicalisation rules

KLEE-FP

- Based on KLEE, a tool for symbolic testing of C and C++ code [Cadar, Dunbar, Engler, OSDI 2008]
- KLEE is based on the LLVM compiler [Lattner, Adve, CGO 2004]
- Supports integer constraints only; symbolic FP not allowed
- KLEE-FP: our modified version of KLEE, extended with support for:
  - Symbolic floating point
  - SIMD vector instructions
  - A substantial portion of Intel SSE instruction set
  - Static path merging
  - Extended expression canonicalisation and crosschecking
- `http://www.pcc.me.uk/~peter/klee-fp/`
  (or google `klee-fp`)

Evaluation

- The code base that we selected was OpenCV 2.1.0, a popular C++ open source computer vision library



Corner detection

Evaluation

- Out of the twenty OpenCV source code files containing SIMD code, we selected ten files upon which to build benchmarks
- Crosschecked 58 SIMD/SSE implementations against scalar versions

  41: verified up to a certain image size (*bounded* equivalence)

  10: found inconsistencies

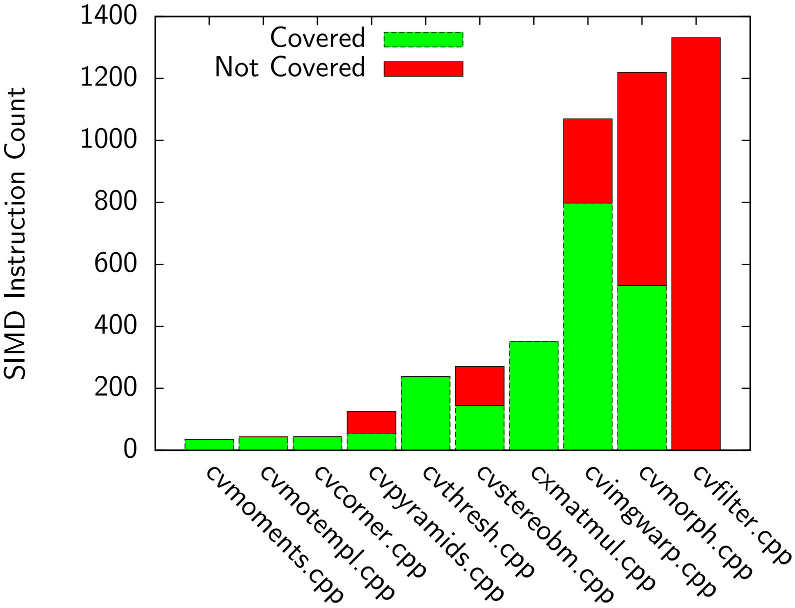
  3: false positives

  4: could not run

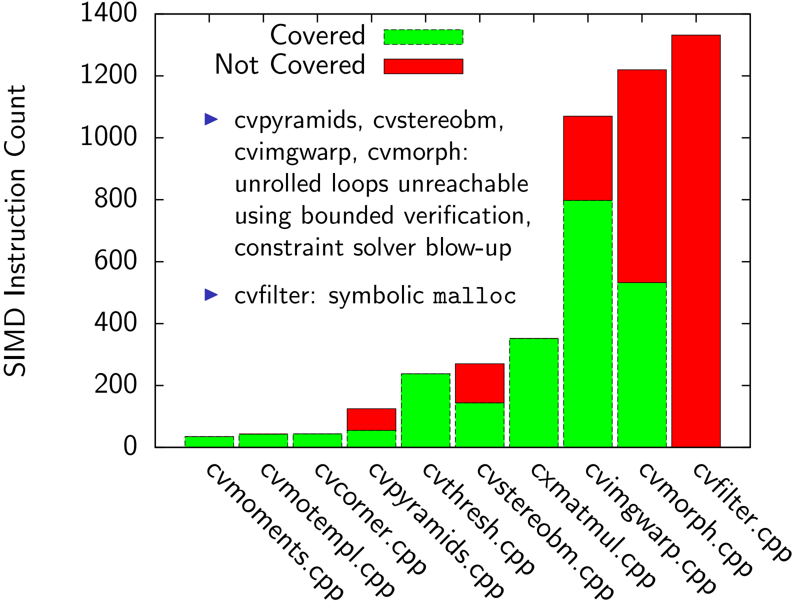
Evaluation – Methodology

- Bounded verification
  - Started with smallest possible image size (4×1 in most cases)
  - Tried all possible sizes up to 16×16 (or $8 \times 8 \rightarrow 8 \times 8$ for benchmarks with different sized input and output images)
  - ~ 200 or ~ 1600 combinations per benchmark
- Verified 34 benchmarks up to these limits
- 7 on a smaller set of image sizes due to:
  - Constant sized input/output images
  - Path explosion (time/memory constraints)
  - Constraint solver blow-up

Evaluation – Coverage



Evaluation – Limitations



- cvpyramids, cvstereobm, cvimgwarp, cvmorph: unrolled loops unreachable using bounded verification, constraint solver blow-up
- cvfilter: symbolic `malloc`

OpenCV – Mismatches found

#	Benchmark/Algorithm	Description
1	`eigenval` (`f32`)	Precision
2	`harris` (`f32`)	Precision, associativity
3	`morph` (dilate, R, `f32`)	Order of min/max operations
4	`morph` (dilate, NR, `f32`)	
5	`morph` (erode, R, `f32`)	
6	`thresh` (`TRUNC`, `f32`)	
7	`pyramid` (`f32`)	Associativity, distributivity
8	`resize` (linear, `u8`)	Precision
9	`transsf.43` (`s16 f32`)	Rounding issue
10	`transcf.43` (`u8 f32`)	Integer/FP differences

- Reported to OpenCV developers
- 2 bugs (eigenval, harris) already confirmed

Conclusion and Future Work

- Automatic technique for checking correctness of SIMD vectorisations with support for floating point operations
- Applied to popular computer vision library, OpenCV
  - Proved the *bounded* equivalence of 41 implementations
  - Found inconsistencies in 10
    - Precision, associativity, distributivity, rounding, ...
- Future work may involve:
  - Inequalities
    - Interval arithmetic
    - Affine arithmetic
  - Floating point counterexamples
  - OpenCL
- `http://www.pcc.me.uk/~peter/klee-fp/` (or google `klee-fp`)

OpenCL

- Race detection
- OpenCL runtime library
- Uses Clang as OpenCL compiler
- Used to cross-check the following benchmarks:
  - AMD SDK – `TemplateC`
  - Parboil – `mri-q`, `mri-fhd`, `cp`
  - Bullet Physics Library – `softbody`
- Found memory bugs, implementation differences

SSE Intrinsic Lowering

- Total of 37 intrinsics supported
- Implemented via a lowering pass that translates the intrinsics into standard LLVM instructions

Input code:

```
%res = call <8 x i16> @llvm.x86.sse2.pslli.w(
          <8 x i16> %arg, i32 1)
```

Output code:

```
%1 = extractelement <8 x i16> %arg, i32 0
%2 = shl i16 %1, 1
%3 = insertelement <8 x i16> undef, i16 %2, i32 0
%4 = extractelement <8 x i16> %arg, i32 1
%5 = shl i16 %4, 1
%6 = insertelement <8 x i16> %3, i16 %5, i32 1
...
%22 = extractelement <8 x i16> %arg, i32 7
%23 = shl i16 %22, 1
%res = insertelement <8 x i16> %21, i16 %23, i32 7
```

OpenCV – Verified up to a certain size

#	Bench	Algo	K	Fmt	Max Size
1	morph	dilate	R	u8	5×5
2				s16	16×16
3				u16	16×16
4			NR	u8	8×3
5				s16	16×16
6				u16	16×16
7				f32	15×15
8		erode	R	u8	4×4
9				s16	16×16
10				u16	16×16
11			NR	s16	16×16
12				u16	16×16
13	pyramid			u8	$8 \times 2 \rightarrow 4 \times 1$
14	remap	nearest neighbor		u8	16×16
15				s16	16×16
16				u16	16×16
17				f32	16×16
18		linear		u8	16×16
19				s16	16×16
20				u16	16×16
21				f32	16×16
22		cubic		u8	16×16
23				s16	16×16
24				u16	16×16
25				f32	16×16

#	Bench	Algo	K	Fmt	Max Size
26	resize	linear		s16	$8 \times 8 \rightarrow 8 \times 8$
27				f32	$8 \times 8 \rightarrow 8 \times 8$
28		cubic		s16	$8 \times 8 \rightarrow 8 \times 8$
29				f32	$8 \times 8 \rightarrow 8 \times 8$
30	silhouette			u8 f32	16×16
31	thresh	BINARY		u8	16×16
32				f32	16×16
33		BINARY_INV		u8	16×16
34				f32	16×16
35		TRUNC		u8	16×16
36		TOZERO		u8	16×16
37				f32	16×16
38		TOZERO_INV		u8	16×16
39				f32	16×16
40	transff.43			f32	
41	transff.44			f32	

eigenval and harris

- Precision
- Associativity

- Scalar:

$$\mathtt{k} * (\mathtt{a} + \mathtt{c}) * (\mathtt{a} + \mathtt{c})$$

- SIMD:

$$\mathtt{\_mm\_mul\_ps}(\mathtt{\_mm\_mul\_ps}(\mathtt{t}, \mathtt{t}), \mathtt{k4})$$

$$\mathtt{k4} = (\mathtt{k}, \mathtt{k}, \mathtt{k}, \mathtt{k})$$

$$\mathtt{t} = (\mathtt{a}_0 + \mathtt{c}_0, \mathtt{a}_1 + \mathtt{c}_1, \mathtt{a}_2 + \mathtt{c}_2, \mathtt{a}_3 + \mathtt{c}_3)$$

- To be fixed in OpenCV

eigenval and harris

- Precision
- Associativity
- Scalar:

$$((\texttt{float})\texttt{k}) * (\texttt{a} + \texttt{c}) * (\texttt{a} + \texttt{c})$$

- SIMD:

$$\texttt{\_mm\_mul\_ps}(\texttt{\_mm\_mul\_ps}(\texttt{t}, \texttt{t}), \texttt{k4})$$

$$\texttt{k4} = (\texttt{k}, \texttt{k}, \texttt{k}, \texttt{k})$$

$$\texttt{t} = (\texttt{a}_0 + \texttt{c}_0, \texttt{a}_1 + \texttt{c}_1, \texttt{a}_2 + \texttt{c}_2, \texttt{a}_3 + \texttt{c}_3)$$

- To be fixed in OpenCV

eigenval and harris

- Precision
- Associativity
- Scalar:

$$((\texttt{float})\texttt{k}) * ((\texttt{a} + \texttt{c}) * (\texttt{a} + \texttt{c}))$$

- SIMD:

$$\texttt{\_mm\_mul\_ps}(\texttt{\_mm\_mul\_ps}(\texttt{t}, \texttt{t}), \texttt{k4})$$

$$\texttt{k4} = (\texttt{k}, \texttt{k}, \texttt{k}, \texttt{k})$$

$$\texttt{t} = (\texttt{a}_0 + \texttt{c}_0, \texttt{a}_1 + \texttt{c}_1, \texttt{a}_2 + \texttt{c}_2, \texttt{a}_3 + \texttt{c}_3)$$

- To be fixed in OpenCV