Symbolic Crosschecking of Floating-Point and SIMD Code

Peter Collingbourne, Cristian Cadar, Paul H J Kelly

Department of Computing, Imperial College London

4 November, 2010

Outline

- SIMD
- Symbolic Execution
- KLEE-FP
- Evaluation

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] } */
```

Symbolic Execution for SIMD

- Manually translating scalar code into an equivalent SIMD version is a difficult and error-prone task
- We propose a novel automatic technique for verifying that the SIMD version of a piece of code is equivalent to its (original) scalar version

Symbolic Execution

- Symbolic execution tests multiple paths through the program
- Determines the feasibility of a particular path by reasoning about all possible values using a constraint solver
- Can verify code correctness by verifying the absence of certain error types (such as array bounds errors or division by zero) on a per-path basis

Symbolic Execution – Operation

- Each variable may hold either a concrete or a symbolic value
- Symbolic value – an expression consisting of mathematical or boolean operations and symbols
- For example, an integer variable `i` may hold a value such as $x + 3$

Symbolic Execution – Example

x	$\perp$

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
  ;
else
  abort();
```

Symbolic Execution – Example

x	X

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
  ;
else
  abort();
```

Symbolic Execution – Example

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
  ;
else
  abort();
```

x	x
$x > x - 1$	

x	x
$\neg(x > x - 1)$	

Symbolic Execution – Example

```c
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
assert(x > x - 1);
if (x > x - 1)
  ;
else
  abort();
```

x	x
$x > x - 1$	

x	x
$\neg(x > x - 1)$	

Symbolic Execution – Example

x	$\perp$

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
  assert(x > x - 1);
  if (x > x - 1)
    ;
  else
    abort();
}
```

Symbolic Execution – Example

x	X

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
  assert(x > x - 1);
  if (x > x - 1)
    ;
  else
    abort();
}
```

Symbolic Execution – Example

x	x
$x > 0$	

x	x
$\neg(x > 0)$	

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
  assert(x > x - 1);
  if (x > x - 1)
    ;
  else
    abort();
}
```

Symbolic Execution – Example

```
unsigned int x;
klee_make_symbolic(&x, sizeof(x), "x");
if (x > 0) {
  assert(x > x - 1);
  if (x > x - 1)
    ;
  else
    abort();
}
```

x	x
$x > 0$	
$x > x - 1$	

x	x
$x > 0$	
$\neg(x > x - 1)$	

x	x
$\neg(x > 0)$	

Issues

- SIMD vectorised code frequently makes intensive use of floating point arithmetic
- The current generation of symbolic execution tools lack or have poor support for floating point and SIMD

Technique

- For the purposes of this work, we need to test *equality*, not inequality
- The requirements for equality of two floating point values are harder to satisfy than for integers
- Usually, the two numbers need to be built up in the same way to be sure of equality
- This suits us just fine for verification purposes

KLEE

- Tool for symbolic testing of C and C++ code [Cadar, Dunbar, Engler, OSDI 2008]
- Based on the LLVM compiler framework
- `svn co http://llvm.org/svn/llvm-project/klee/trunk klee`
- Supports integer constraints only; symbolic FP not allowed

KLEE-FP: our modified version of KLEE

- Our improvements to KLEE:
  - Symbolic construction of floating-point operations
  - A set of expression matching and canonicalization rules for establishing FP equality
  - Support for SIMD instructions `insertelement`, `extractelement`, `shufflevector`
  - Semantics for a substantial portion of Intel SSE instruction set
- Related contributions to LLVM:
  - Atomic intrinsic lowering
  - An aggressive variant of phi-node folding
- Selected modifications contributed upstream to KLEE/LLVM
- `git clone git://git.pcc.me.uk/~peter/klee-fp.git`

Scalar/SIMD Implementation

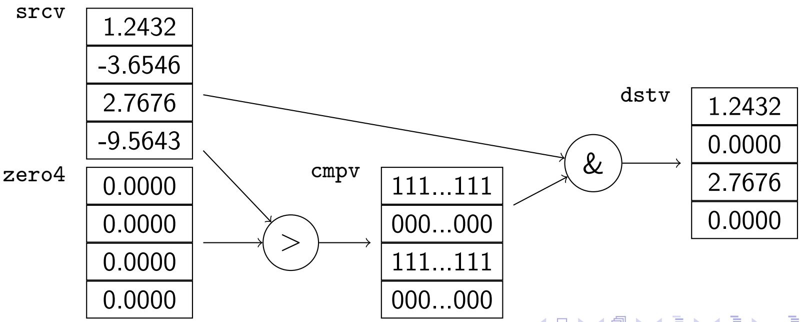
```
void zlimit(int simd, float *src, float *dst,
            size_t size) {
   if (simd) {
      __m128 zero4 = _mm_set1_ps(0.f);
      while (size >= 4) {
         __m128 srcv = _mm_loadu_ps(src);
         __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
         __m128 dstv = _mm_and_ps(cmpv, srcv);
    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

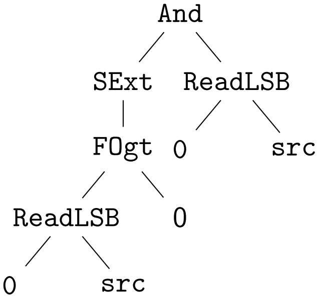
Scalar/SIMD Implementation

```
__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;
}
```

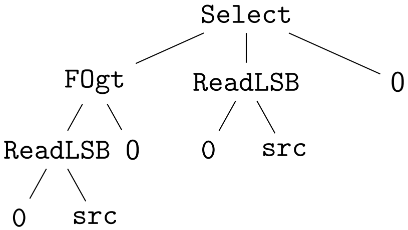


Scalar/SIMD Implementation

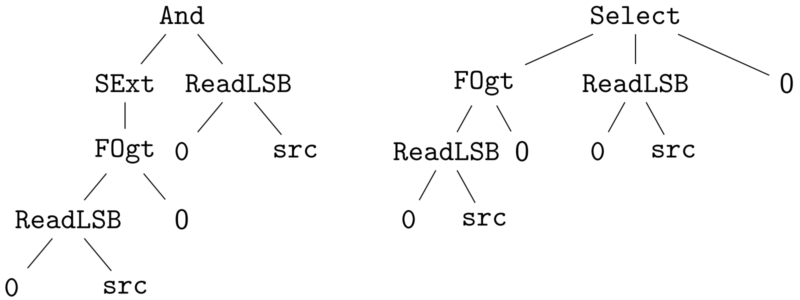
```
void zlimit(int simd, float *src, float *dst,
            size_t size) {
   if (simd) {
      __m128 zero4 = _mm_set1_ps(0.f);
      while (size >= 4) {
         __m128 srcv = _mm_loadu_ps(src);
         __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
         __m128 dstv = _mm_and_ps(cmpv, srcv);
         _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/SIMD Implementation



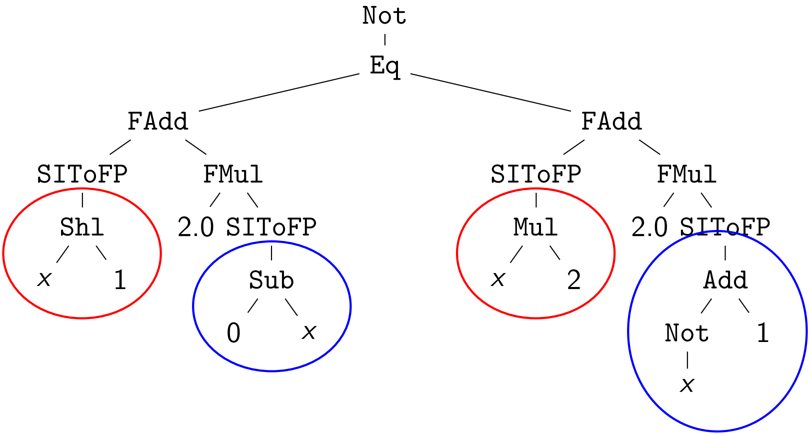
$$\texttt{And}(\texttt{SExt}(P^1), X) \to \texttt{Select}(P^1, X, 0)$$

Cross-checking SIMD Code

```
int main(void) {
    float src[64], dstv[64], dsts[64];
    uint32_t *dstvi = (uint32_t *)dstv;
    uint32_t *dstsi = (uint32_t *)dsts;
    unsigned i;
    klee_make_symbolic(src, sizeof(src), "src");
    zlimit(0, src, dsts, 64);
    zlimit(1, src, dstv, 64);
    for (i = 0; i < 64; ++i)
        assert(dstvi[i] == dstsi[i]);
}
```

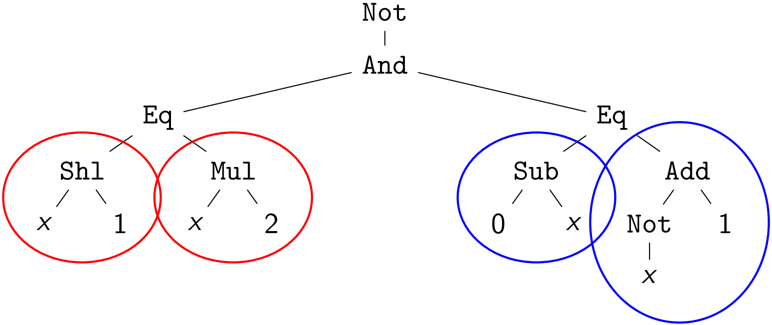
Checking Equality

$$\neg(I2F(x << 1) +_f 2.0 *_f I2F(0 - x) = I2F(x \times 2) +_f 2.0 *_f I2F(\sim x + 1))$$
$$\to \neg(x << 1 = x \times 2 \wedge 0 - x = \sim x + 1)$$



Checking Equality

$$\neg(I2F(x << 1) +_f 2.0 *_f I2F(0 - x)$$
$$= I2F(x \times 2) +_f 2.0 *_f I2F(\sim x + 1))$$
$$\rightarrow \neg(x << 1 = x \times 2 \land 0 - x = \sim x + 1)$$



Floating Point Operations

- New nodes:

`FAdd`	`FSub`	`FMul`	`FDiv`
`FRem`	`FPToSI`	`FPToUI`	`SIToFP`
`UIToFP`	`FPExt`	`FPTrunc`	`FCmp`

- Outcome sets:

$$\mathbf{O} = \{<, =, >, \texttt{UNO}\}$$

Shorthand	`CmpInst::Predicate`	`FCmp` operation	Meaning
$\texttt{FOeq}(X, Y)$	$\texttt{FCMP\_OEQ} = 0\ 0\ 0\ 1$	$\texttt{FCmp}(X, Y, \{=\})$	Ordered $=$
$\texttt{FOlt}(X, Y)$	$\texttt{FCMP\_OLT} = 0\ 1\ 0\ 0$	$\texttt{FCmp}(X, Y, \{<\})$	Ordered $<$
$\texttt{FOle}(X, Y)$	$\texttt{FCMP\_OLE} = 0\ 1\ 0\ 1$	$\texttt{FCmp}(X, Y, \{<, =\})$	Ordered $\leq$
$\texttt{FUno}(X, Y)$	$\texttt{FCMP\_UNO} = 1\ 0\ 0\ 0$	$\texttt{FCmp}(X, Y, \{\texttt{UNO}\})$	Unordered test

Expression Transformation Rules

- 18 rules, including:

$$\texttt{And}(\texttt{FCmp}(X, Y, O_1), \texttt{FCmp}(X, Y, O_2)) \rightarrow \texttt{FCmp}(X, Y, O_1 \cap O_2)$$

$$\texttt{Or}(\texttt{FCmp}(X, Y, O_1), \texttt{FCmp}(X, Y, O_2)) \rightarrow \texttt{FCmp}(X, Y, O_1 \cup O_2)$$

$$\texttt{Eq}(\texttt{FCmp}(X, Y, O), \texttt{false}) \rightarrow \texttt{FCmp}(X, Y, \mathbf{O} \setminus O)$$

$$\texttt{FOeq}(\texttt{SIToFP}(X), C) \rightarrow \texttt{Eq}(X, \texttt{FPToSI}(C))$$

$$\texttt{FOeq}(\texttt{UIToFP}(X), C) \rightarrow \texttt{Eq}(X, \texttt{FPToUI}(C))$$

$$\texttt{And}(\texttt{SExt}(P^1), X) \rightarrow \texttt{Select}(P^1, X, 0)$$

Category Analysis

$$\mathbf{C} = \{\texttt{NaN}, -\infty, -, 0, +, +\infty\}$$

$$\frac{+ \in \mathrm{cat}(x) \quad + \in \mathrm{cat}(y)}{\{+, +\infty\} \subseteq \mathrm{cat}(x + y)}$$

$$\frac{\mathrm{cat}(x) = \{0, -\} \quad \mathrm{cat}(y) = \{0, +\}}{\neg(x > y)}$$

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
```

Phi Node Folding

```
val = silhData[x] ? ts : val < delbound ? 0 : val;
```

```
bb156:
  ...
  %106 = load float* %scevgep345346, align 4
  %107 = load i8* %scevgep351, align 1
  %108 = icmp eq i8 %107, 0
  br i1 %108, label %bb158, label %bb163

bb158:
  %109 = fcmp uge float %106, %51
  %iftmp.388.0 = select i1 %109, float %106,
                                 float 0.000000e+00
  br label %bb163

bb163:
  %iftmp.387.0 = phi float [ %iftmp.388.0, %bb158 ],
                           [ %49, %bb156 ]
  ...
```

Phi Node Folding

```
val = silhData[x] ? ts : val < delbound ? 0 : val;
```

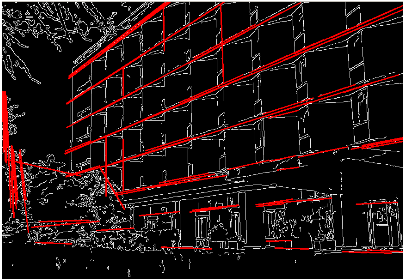
```
bb156:
  ...
  %106 = load float* %scevgep345346, align 4
  %107 = load i8* %scevgep351, align 1
  %108 = icmp eq i8 %107, 0
  %109 = fcmp uge float %106, %51
  %iftmp.388.0 = select i1 %109, float %106,
                                 float 0.000000e+00
  %iftmp.387.0 = select i1 %108, %iftmp.388.0, %49
  ...
```

- 13×13 matrix:

$$2^{169} \to 1$$

Evaluation

- We evaluated our technique on a set of benchmarks that compare scalar and SIMD variants of code developed independently by third parties
- The code base that we selected was OpenCV 2.1.0, a popular C++ open source computer vision library



Evaluation

- Out of the twenty OpenCV source code files containing SSE code, we selected ten files upon which to build benchmarks
- Crosschecked 49 SIMD/SSE implementations against scalar versions
  - Proved the *bounded* equivalence (i.e. *verified*) 39
  - Found inconsistencies in the other 10

Evaluation – Coverage

Source File (src/)	# SIMD	Cov.
cv/cvcorner.cpp	44	100%
cv/cvfilter.cpp	1332	N/A
cv/cvimgwarp.cpp	1070	74.6%
cv/cvmoments.cpp	35	100%
cv/cvmorph.cpp	1220	43.6%
cv/cvmotempl.cpp	43	100%
cv/cvpyramids.cpp	125	44.0%
cv/cvstereobm.cpp	270	53.3%
cv/cvthresh.cpp	238	100%
cxcore/cxmatmul.cpp	352	100%

OpenCV – Verified up to a certain size

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

#	Bench	Algo	K	Fmt	Max Size
24	resize	linear		s16	$4 \times 4 \rightarrow 8 \times 8$
25				f32	$4 \times 4 \rightarrow 8 \times 8$
26		cubic		s16	$4 \times 4 \rightarrow 8 \times 8$
27				f32	$4 \times 4 \rightarrow 8 \times 8$
28	silhouette			u8 f32	4×4
29	thresh	BINARY		u8	13×13
30				f32	13×13
31		BINARY_INV		u8	13×13
32				f32	13×13
33		TRUNC		u8	13×13
34		TOZERO		u8	13×13
35				f32	13×13
36		TOZERO_INV		u8	13×13
37				f32	13×13
38	transff.43			f32	
39	transff.44			f32	

OpenCV – Mismatches found

#	Bench	Algo	K	Fmt	Size	Description
1	`eigenval`			`f32`	4×4	Precision
2	`harris`			`f32`	4×4	Precision, associativity
3	`morph`	dilate	R	`f32`	4×1	Order of min/max operations
4			NR	`f32`	4×1	
5		erode	R	`f32`	4×1	
6	`thresh`	`TRUNC`		`f32`	4×4	
7	`pyramid`			`f32`	$16 \times 2 \to 8 \times 1$	Associativity, distributivity
8	`resize`	linear		`u8`	$4 \times 4 \to 8 \times 8$	Precision
9	`transsf.43`			`s16 f32`		Rounding issue
10	`transcf.43`			`u8 f32`		Integer/FP differences

morph (f32) and thresh (TRUNC, f32)

- std::{min,max}, {MIN,MAX}PS:

$$\begin{aligned}\mathtt{min}(X, Y) &= \mathtt{Select}(\mathtt{FOlt}(X, Y), X, Y)\\ \mathtt{max}(X, Y) &= \mathtt{Select}(\mathtt{FOlt}(Y, X), X, Y)\end{aligned}$$

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

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

morph (f32) and thresh (TRUNC, f32)

- std::{min,max}, {MIN,MAX}PS:

$$\begin{aligned}\mathtt{min}(X, Y) &= \mathtt{Select}(\mathtt{FOlt}(X, Y), X, Y)\\ \mathtt{max}(X, Y) &= \mathtt{Select}(\mathtt{FOlt}(Y, X), X, Y)\end{aligned}$$

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

$$\begin{aligned}\mathtt{min}(\mathtt{min}(1, \mathrm{NaN}), 2) &\quad = \mathtt{min}(\mathrm{NaN}, 2) &\quad = 2\\ \mathtt{min}(1, \mathtt{min}(\mathrm{NaN}, 2)) &\quad = \mathtt{min}(1, 2) &\quad = 1\end{aligned}$$

Conclusion and Future Work

- KLEE-FP extends KLEE with floating point/SIMD capabilities
- Has been used to find bugs in code extracted from the wild
- Future work may involve:
  - Inequalities
    - Interval arithmetic
    - Affine arithmetic
  - Floating point counterexamples
  - GPUs: CUDA/OpenCL?

Q&A

```
git clone git://git.pcc.me.uk/~peter/klee-fp.git
```