Compilers Laboratory

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ICEx Instituto de Ciências Exatas

UFMG

Dawn Project
DawnCC vs. Large Real-World Applications

Breno Campos Ferreira Guimarães
Gleison Souza Diniz Mendonça

Motivation

Motivation

- DawnCC has worked well with:
  - Hand-made unit tests (during project's inception)
  - Small application-specific benchmarks (Polybench, MgBench, etc.)
  - Medium-sized, slightly more general benchmark applications (parts of Parboil, Rodinia, etc.)
- Selection bias?
- What about large, actively developed, real-world applications?
  - Can we infer memory bounds and disambiguate pointers in real-world code patterns?
  - Can we annotate complex actual computation loops in large-scale programs?
  - If so, can we run the annotated pieces of code and obtain performance gains?

Setup

Setting Up the Experiment

- We needed large C code bases with potential for data parallelism
- Sifted through largest C GitHub repositories
- Chose a few promising applications:
  - FFmpeg
    - audio/video encoding/decoding libraries
    - ~2400 C source files
    - ~900 header files
  - VLC Media Player
    - (mostly its muxer/demuxer and codec implementations, in libVLC)
    - ~1000 C source files
    - ~500 header files

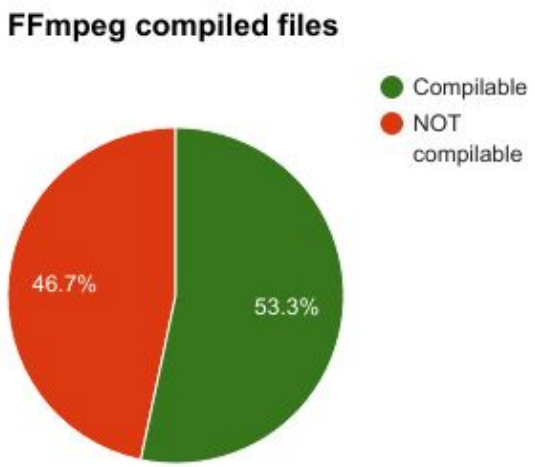
Setting Up the Experiment

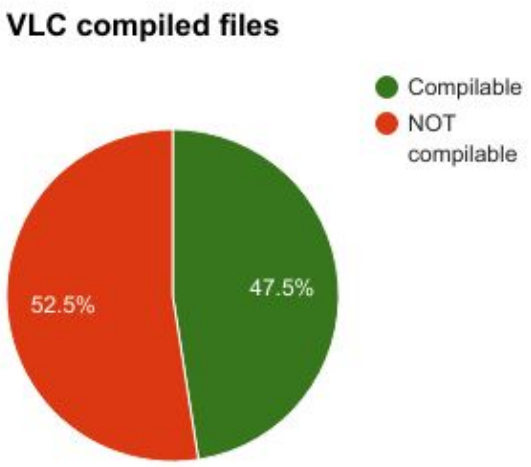
- A few caveats:
  - We compile, analyze and annotate C files individually
    - Code with inter-file dependencies is not compilable
    - Can also limit results for files we do analyze
      - For instance, variables/constants that are statically known, but defined in other C files become unanalyzable in a single file's scope
  - Some other files we cannot compile for different reasons
    - Most libraries have code that uses GCC-specific features, while we use Clang
  - We do not link files or generate binary code
    - Simply generate IR then analyze and annotate source files
    - Compilable code is not necessarily linkable

Results

Results

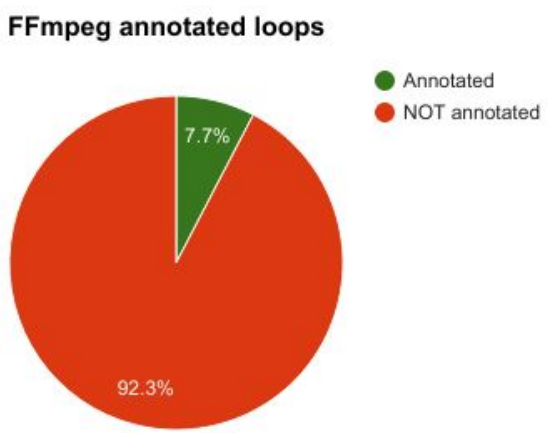
- Number of files compiled/total
  - FFmpeg: 1266/2374
    - ~53%
  - VLC: 451/949
    - ~47%

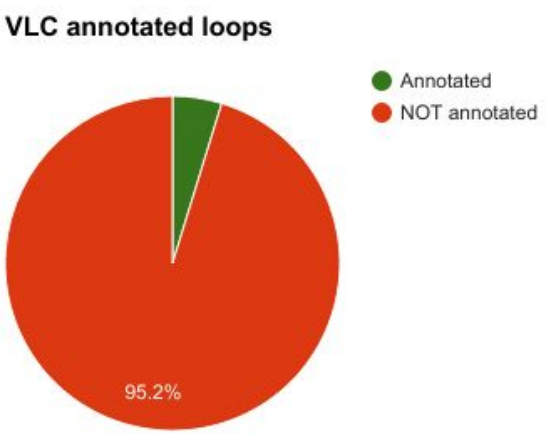




Results

- Number of loops annotated/total
  - FFmpeg: 843/11011
    - ~8%
  - VLC: 115/2420
    - ~5%





Results

- A few examples of the kinds of loops we can annotate...

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

Results

- A few examples of the kinds of loops we can annotate…

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

Four distinct pointers as parameters!

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

...which are all accessed within the loop.

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

Aliasing?

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

Aliasing?

Memory limits?

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + oversho
                dst[x] = brightsrc[x] + overshoo
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

Variables received as parameters...

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

...which are then used to iterate/index memory.

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undersho
            else if (bsrc[x] > brightsrc[x] +
                dst[x] = brightsrc[x] + overshoot
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

AND pointer arithmetics increments...

Results

- A few examples of the kinds of loops we can annotate...

```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksrc,
                         const uint8_t *brightsrc, uint8_t *dst,
                         ptrdiff_t blinesize, ptrdiff_t darklinesize,
                         ptrdiff_t brightlinesize, ptrdiff_t dlinesize, int w,
                         int h, int undershoot, int overshoot) {

    int x, y;

    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            if (bsrc[x] < darksrc[x] - undershoot)
                dst[x] = darksrc[x] - undershoot;
            else if (bsrc[x] > brightsrc[x] + overshoot)
                dst[x] = brightsrc[x] + overshoot;
            else
                dst[x] = bsrc[x];
            }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
    }
}
```

...which are then used to step through pointers being accessed.

Results

- What the annotations look like

Results

- Symbolic Inference

```
long long int AI[26];
AI1[0] = h + -1;
AI1[1] = blinesize * AI1[0];
AI1[2] = w + -1;
AI1[3] = AI1[1] + AI1[2];
AI1[4] = AI1[3] + 1;
AI1[5] = AI1[4] / 1;
AI1[6] = (AI1[5] > 0);
AI1[7] = (AI1[6] ? AI1[5] : 0);
AI1[8] = darklinesize * AI1[0];
AI1[9] = AI1[8] + AI1[2];
AI1[10] = AI1[9] + 1;
AI1[11] = AI1[10] / 1;
AI1[12] = (AI1[11] > 0);
AI1[13] = (AI1[12] ? AI1[11] : 0);
AI1[14] = brightlinesize * AI1[0];
AI1[15] = AI1[14] + AI1[2];
AI1[16] = AI1[15] + 1;
AI1[17] = AI1[16] / 1;
AI1[18] = (AI1[17] > 0);
AI1[19] = (AI1[18] ? AI1[17] : 0);
AI1[20] = dlinesize * AI1[0];
AI1[21] = AI1[20] + AI1[2];
AI1[22] = AI1[21] + 1;
AI1[23] = AI1[22] / 1;
AI1[24] = (AI1[23] > 0);
AI1[25] = (AI1[24] ? AI1[23] : 0);
```

Results

- Symbolic Inference

```
long long int AI[26];
AI1[0] = h + -1;
AI1[1] = blinesize * AI1[0];
AI1[2] = w + -1;
AI1[3] = AI1[1] + AI1[2];
AI1[4] = AI1[3] + 1;
AI1[5] = AI1[4] / 1;
AI1[6] = (AI1[5] > 0);
AI1[7] = (AI1[6] ? AI1[5] : 0);
AI1[8] = darklinesize * AI1[0];
AI1[9] = AI1[8] + AI1[2];
AI1[10] = AI1[9] + 1;
AI1[11] = AI1[10] / 1;
AI1[12] = (AI1[11] > 0);
AI1[13] = (AI1[12] ? AI1[11] : 0);
AI1[14] = brightlinesize * AI1[0];
AI1[15] = AI1[14] + AI1[2];
AI1[16] = AI1[15] + 1;
AI1[17] = AI1[16] / 1;
AI1[18] = (AI1[17] > 0);
AI1[19] = (AI1[18] ? AI1[17] : 0);
AI1[20] = dlinesize * AI1[0];
AI1[21] = AI1[20] + AI1[2];
AI1[22] = AI1[21] + 1;
AI1[23] = AI1[22] / 1;
AI1[24] = (AI1[23] > 0);
AI1[25] = (AI1[24] ? AI1[23] : 0);
```

All 6 variables used to control memory accesses are queried symbolically to infer proper memory bounds

Results

- Symbolic Inference

```
long long int AI[26];
AI1[0] = h + -1;
AI1[1] = blinesize * AI1[0];
AI1[2] = w + -1;
AI1[3] = AI1[1] + AI1[2];
AI1[4] = AI1[3] + 1;
AI1[5] = AI1[4] / 1;
AI1[6] = (AI1[5] > 0);
AI1[7] = (AI1[6] ? AI1[5] : 0);
AI1[8] = darklinesize * AI1[0];
AI1[9] = AI1[8] + AI1[2];
AI1[10] = AI1[9] + 1;
AI1[11] = AI1[10] / 1;
AI1[12] = (AI1[11] > 0);
AI1[13] = (AI1[12] ? AI1[11] : 0);
AI1[14] = brightlinesize * AI1[0];
AI1[15] = AI1[14] + AI1[2];
AI1[16] = AI1[15] + 1;
AI1[17] = AI1[16] / 1;
AI1[18] = (AI1[17] > 0);
AI1[19] = (AI1[18] ? AI1[17] : 0);
AI1[20] = dlinesize * AI1[0];
AI1[21] = AI1[20] + AI1[2];
AI1[22] = AI1[21] + 1;
AI1[23] = AI1[22] / 1;
AI1[24] = (AI1[23] > 0);
AI1[25] = (AI1[24] ? AI1[23] : 0);
```

Each of the 4 pointers have their upper bounds symbolically verified.

Results

- Pointer Disambiguation

Results

- Pointer Disambiguation

```
char RST_AI1 = 0;
RST_AI1 |= !(((void*) (brightsrc + 0) > (void*) (bsrc + AI1[7]))
|| ((void*) (bsrc + 0) > (void*) (brightsrc + AI1[19])));
RST_AI1 |= !(((void*) (brightsrc + 0) > (void*) (darksrc + AI1[13]))
|| ((void*) (darksrc + 0) > (void*) (brightsrc + AI1[19])));
RST_AI1 |= !(((void*) (brightsrc + 0) > (void*) (dst + AI1[25]))
|| ((void*) (dst + 0) > (void*) (brightsrc + AI1[19])));
RST_AI1 |= !(((void*) (bsrc + 0) > (void*) (darksrc + AI1[13]))
|| ((void*) (darksrc + 0) > (void*) (bsrc + AI1[7])));
RST_AI1 |= !(((void*) (bsrc + 0) > (void*) (dst + AI1[25]))
|| ((void*) (dst + 0) > (void*) (bsrc + AI1[7])));
RST_AI1 |= !(((void*) (darksrc + 0) > (void*) (dst + AI1[25]))
|| ((void*) (dst + 0) > (void*) (darksrc + AI1[13])));
```

Results

- Pointer Disambiguation

```
char RST_AI1 = 0;
RST_AI1 |= !(((void*) (brightsrc + 0) > (void*) (bsrc + AI1[7]))
|| ((void*) (bsrc + 0) > (void*) (brightsrc + AI1[19])));
RST_AI1 |= !(((void*) (brightsrc + 0) > (void*) (darksrc + AI1[13]))
|| ((void*) (darksrc + 0) > (void*) (brightsrc + AI1[19])));
RST_AI1 |= !(((void*) (brightsrc + 0) > (void*) (dst + AI1[25]))
|| ((void*) (dst + 0) > (void*) (brightsrc + AI1[19])));
RST_AI1 |= !(((void*) (bsrc + 0) > (void*) (darksrc + AI1[13]))
|| ((void*) (darksrc + 0) > (void*) (bsrc + AI1[7])));
RST_AI1 |= !(((void*) (bsrc + 0) > (void*) (dst + AI1[25]))
|| ((void*) (dst + 0) > (void*) (bsrc + AI1[7])));
RST_AI1 |= !(((void*) (darksrc + 0) > (void*) (dst + AI1[25]))
|| ((void*) (dst + 0) > (void*) (darksrc + AI1[13])));
```

Aliasing check validates all possible pointer aliasing pairs.

Results

- Inserted annotations

```
#pragma acc data copy(brightsrc[0:AI1[19]],bsrc[0:AI1[7]],darksrc[0:AI1[13]],dst[0:AI1[25]])
#pragma acc kernels if(!RST_AI1)
for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
             if (bsrc[x] < darksrc[x] - undershoot)
                  dst[x] = darksrc[x] - undershoot;
             else if (bsrc[x] > brightsrc[x] + overshoot)
                  dst[x] = brightsrc[x] + overshoot;
             else
                  dst[x] = bsrc[x];
             }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
   }
```

Results

- Inserted annotations

```
#pragma acc data copy(brightsrc[0:AI1[19]],bsrc[0:AI1[7]],darksrc[0:AI1[13]],dst[0:AI1[25]])
#pragma acc kernels if(!RST_AI1)
for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
             if (bsrc[                                    oot)
                  da                                          ;
             else if                                     ershoot)
                  dst[x]                                 ershoot;
             else
                  dst[x] = bsrc[x];
             }

        dst += dlinesize;
        bsrc += blinesize;
        darksrc += darklinesize;
        brightsrc += brightlinesize;
   }
```

Aliasing check

Memory access bounds

Results

- A few other highlights...

Results

- A few other highlights…
- Fun with macros and Symbolic Inference

```
static void BAYER_RENAME(rgb24_interpolate)( const uint8_t *src, int src_stride,
                                             uint8_t *dst, int dst_stride,
                                             int width) {
  int i;
  BAYER_TO_RGB24_COPY
  src += 2 * BAYER_SIZEOF;
  dst += 6;

  for (i = 2; i < width - 2; i += 2) {
    BAYER_TO_RGB24_INTERPOLATE
    src += 2 * BAYER_SIZEOF;
    dst += 6;
  }

  if (width > 2) {
    BAYER_TO_RGB24_COPY
  }
}
```

Results

- A few other highlights…
- Fun with macros and Symbolic Inference

```
static void BAYER_RENAME(rgb24_interpolate)( const uint8_t *src, int src_stride,
                                             uint8_t *dst, int dst_stride,
                                             int width) {
  int i;
  BAYER_TO_RGB24_COPY
  src += 2 * BAYER_SIZEOF;
  dst += 6;

  for (i = 2; i < width - 2; i += 2) {
    BAYER_TO_RGB24_INTERPOLATE
    src += 2 * BAYER_SIZEOF;
    dst += 6;
  }

  if (width > 2) {
    BAYER_TO_RGB24_COPY
  }
}
```

Macro expansion galore!

Results

- A few other highlights…
- Fun with macros and Symbolic Inference

```
long long int AI1[];
AI1[0] = 2 * src_stride;
AI1[1] = AI1[0] + 3;
AI1[2] = AI1[1] * 1;
AI1[3] = src_stride + 1;
AI1[4] = AI1[3] + 2;
(...)
```

Results

- A few other highlights…
- Fun with macros and Symbolic Inference

```
long long int AI1[];
AI1[0] = 2 * src_stride;
AI1[1] = AI1[0] + 3;
AI1[2] = AI1[1] * 1;
AI1[3] = src_stride + 1;
AI1[4] = AI1[3] + 2;
(...)
AI1[157] = (AI1[156] ? AI1[110] : AI1[155]);
AI1[158] = AI1[109] > AI1[157];
AI1[159] = (AI1[158] ? AI1[109] : AI1[157]);
AI1[160] = AI1[103] > AI1[159];
AI1[161] = (AI1[160] ? AI1[103] : AI1[159]);
(...)
```

Results

- A few other highlights…
- Fun with macros and Symbolic Inference

```
long long int AI1[334];
AI1[0] = 2 * src_stride;
AI1[1] = AI1[0] + 3;
AI1[2] = AI1[1] * 1;
AI1[3] = src_stride + 1;
AI1[4] = AI1[3] + 2;
(...)
AI1[157] = (AI1[156] ? AI1[110] : AI1[155]);
AI1[158] = AI1[109] > AI1[157];
AI1[159] = (AI1[158] ? AI1[109] : AI1[157]);
AI1[160] = AI1[103] > AI1[159];
AI1[161] = (AI1[160] ? AI1[103] : AI1[159]);
(...)
AI1[330] = AI1[263] + AI1[328];
AI1[331] = -1 * AI1[328];
AI1[332] = AI1[329] ? AI1[263] : AI1[330];
AI1[333] = AI1[329] ? AI1[328] : AI1[331];
```

Results

- A few other highlights…
- Fun with macros and Symbolic Inference

```
long long int AI1[334];
AI1[0] = 2 * src_stride;
AI1[1] = AI1[0] + 3;
AI1[2] = AI1[1] * 1;
AI1[3] = src_stride + 1;
AI1[4] = AI1[3] + 2;
(...)
AI1[157] = (AI1[156] ? AI1[110] : AI1[155]);
AI1[158] = AI1[109] > AI1[157];
AI1[159] = (AI1[158] ? AI1[109] : AI1[157]);
AI1[160] = AI1[103] > AI1[159];
AI1[161] = (AI1[160] ? AI1[103] : AI1[159]);
(...)
AI1[330] = AI1[263] + AI1[328];
AI1[331] = -1 * AI1[328];
AI1[332] = AI1[329] ? AI1[263] : AI1[330];
AI1[333] = AI1[329] ? AI1[328] : AI1[331];
```

Yes, THREE HUNDRED AND THIRTY FOUR assignments/comparisons to infer upper bounds!

Results

- A few other highlights…
- Several cases with 5+ pointers
  - Generating 10+ disambiguation checks
- Lots of copyin/copyout optimizations
  - Only copying use-only memory to device, but not bringing it back, and vice-versa
- Most annotations made possible by symbolic inference
  - Memory bounds are rarely constants known at compile time
    - True for most real-world applications, where computations are usually implemented as kernel functions.

Results

- What about the lowlights?

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:
    - Function calls within loops

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:
    - Function calls within loops

```
for (i=0; i<x; i++) {
  n[i] = rand();
}
```

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:
    - Function calls within loops
      - Cannot verify memory safety without an interprocedural analysis (analyzes beyond function bounds)

```
for (i=0; i<x; i++) {
  n[i] = rand();
}
```

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:
    - Function calls within loops
      - Cannot verify memory safety without an interprocedural analysis (analyzes beyond function bounds)
    - Structure types + pointers

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:
    - Function calls within loops
      - Cannot verify memory safety without an interprocedural analysis (analyzes beyond function bounds)
    - Structure types + pointers

```
struct mytype {
  int *n;
}
```

Results

- What about the lowlights?
- Not many cases of copy coalescing optimizations
  - Usually loops that access the same data are split amongst different functions, invalidating the intraprocedural coalescing algorithm
- A few code patterns we cannot annotate present in a large amount of loops
  - Biggest culprits, by far, are:
    - Function calls within loops
      - Cannot verify memory safety without an interprocedural analysis (beyond function bounds)
    - Structure types + pointers
      - Cannot assess precisely the size of the data type at compile time

What else?

DawnCC + Psyche-c

- Ideally, we would want to compile annotated versions and check for performance gains
- But the large size of the applications make compiling and testing isolated code regions unfeasible

DawnCC + Psyche-c

- Ideally, we would want to compile annotated versions and check for performance gains
- But the large size of the applications make compiling and testing isolated code regions unfeasible
- Idea:
  - Isolate functions that contain promising annotated loops
  - Use psyche-c to reconstruct missing pieces
  - Generate valid (but synthetic) input values
  - Run compiled kernels with and without annotations enabled
  - Compare performance

- Work in progress! No results yet :(

That's it!

Questions?