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...
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...

```c
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...

```c
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.
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...

```c
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksr, const uint8_t *brightsr, 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] < darksr[x] - undershoot)
                dst[x] = darksr[x] - undershoot;
            else if (bsrc[x] > brightsr[x] + overshoot)
                dst[x] = brightsr[x] + overshoot;
            else
                dst[x] = bsrc[x];

            dst += dlinesize;
            bsrc += blinesize;
            darksr += darklinesize;
            brightsr += brightlinesize;
        }
    }
```

**Aliasing?**

**Memory limits?**
Results

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

```c
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;
    }
}
```
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...

```c
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;
    }
}
```
static void maskedclamp8(const uint8_t *bsrc, const uint8_t *darksr, const uint8_t *brightsr, 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] < darksr[x] - undershoot)
                dst[x] = darksr[x] - undershoot;
            else if (bsrc[x] > brightsr[x] + overshoot)
                dst[x] = brightsr[x] + overshoot;
            else
                dst[x] = bsrc[x];
        }

        dst += dlinesize;
        bsrc += blinesize;
        darksr += darklinesize;
        brightsr += brightlinesize;
    }
}

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

- What the annotations look like
Results

- **Symbolic Inference**

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

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

- **Symbolic Inference**

  ```c
  long long int AI[26];
  AI[0] = h + -1;
  AI[1] = blinesize * AI[0];
  AI[2] = w + -1;
  AI[6] = (AI[5] > 0);
  AI[8] = darklinesize * AI[0];
  AI[12] = (AI[11] > 0);
  AI[14] = brightlinesize * AI[0];
  AI[17] = AI[16] / 1;
  AI[18] = (AI[17] > 0);
  AI[20] = dlinesize * AI[0];
  AI[22] = AI[21] + 1;
  AI[23] = AI[22] / 1;
  AI[24] = (AI[23] > 0);
  ```

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

- Pointer Disambiguation
Results

- Pointer Disambiguation

```c
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

```c
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*) (darksrc + 0) > (void*) (dst + AI1[25]))
  || ((void*) (dst + 0) > (void*) (darksrc + AI1[13])));
RST_AI1 |= !((void*) (darksrc + 0) > (void*) (dst + AI1[25]))
  || ((void*) (dst + 0) > (void*) (darksrc + AI1[13])));
```

Aliasing check validates all possible pointer aliasing pairs.
#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

```c
#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;
}
```

- Aliasing check
- Memory access bounds
Results

• A few other highlights...
Results

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

```c
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
    }
}
```
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

Macro expansion galore!
Results

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

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

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

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

```c
long long int AI1[334];
AI1[0] = 2 * src_stride;
AI1[1] = AI1[0] + 3;
AI1[3] = src_stride + 1;
(....)
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[333] = AI1[329] ? AI1[328] : AI1[331];
```
Results

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

```c
long long int AI1[334];
AI1[0] = 2 * src_stride;
AI1[1] = AI1[0] + 3;
AI1[3] = src_stride + 1;
(...)
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[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

```cpp
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)

```cpp
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

```c
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?