DEALING WITH ALIASING USING CONTRACTS

BEATING FORTRAN'S PERFORMANCE

Gábor Horváth, PhD Student, Eötvös Loránd University
xazax.hun@gmail.com

ALIASING

```
int f(int &a, float &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, float*) {
    store i32 2, i32* %a
    store float 3, float* %b
    ret i32 2
}
```

ALIASING

```
int f(int &a, float &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, float*) {
    store i32 2, i32* %a
    store float 3, float* %b
    ret i32 2
}
```

```
int f(int &a, int &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, i32*) {
    store i32 2, i32* %a
    store i32 3, i32* %b
    %tmp = load i32, i32* %a
    ret i32 %tmp
}
```

ALIASING

```
int f(int &a, float &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, float*) {
    store i32 2, i32* %a
    store float 3, float* %b
    ret i32 2
}
```

```
int f(int &a, int &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, i32*) {
    store i32 2, i32* %a
    store i32 3, i32* %b
    %tmp = load i32, i32* %a
    ret i32 %tmp
}
```

Some parameters might alias!

Type based alias analysis

WHY DOES ALIASING MATTER?

LATENCY NUMBERS

```
L1 cache reference      0.5 ns
Branch mispredict       5   ns
L2 cache reference      7   ns   14x L1 cache
Mutex lock/unlock       25  ns
Main memory reference   100 ns   20x L2 cache, 200x L1 cache
```

WHY DOES ALIASING MATTER?

LATENCY NUMBERS

```
L1 cache reference      0.5 ns
Branch mispredict       5   ns
L2 cache reference      7   ns   14x L1 cache
Mutex lock/unlock       25  ns
Main memory reference   100 ns   20x L2 cache, 200x L1 cache
```

OPTIMIZATIONS

WHY DOES ALIASING MATTER?

LORE: LOop Repository for the Evaluation of compilers

Numbers from P1296R0

	Loops	**Sped up**	**Mean Speedup**	**Slowed**	**Mean slowdown**
GCC	1939	734 (38%)	2.39x	155 (8%)	0.766
ICC	1861	843 (45%)	2.59x	94 (5%)	0.61

In some cases `__restrict__` provides ~40X pref

FORTRAN

- Procedure arguments and variables may not alias
- Inception when CPU time was expensive
- To convince people not to write in assembly...
- ...you need to generate blazing fast code

FORTRAN

- Procedure arguments and variables may not alias
- Inception when CPU time was expensive
- To convince people not to write in assembly...
- ...you need to generate blazing fast code

C++

No standard way (other than types) to give aliasing related hints.

NOT VECTORIZED

```
void f(int *a, int *b, const int& num) {
    for(int i = 0; i < num; ++i) {
        a[i] = b[i] * b[i] + 1;
    }
}
```

NOT VECTORIZED

```
void f(int *a, int *b, const int& num) {
    for(int i = 0; i < num; ++i) {
        a[i] = b[i] * b[i] + 1;
    }
}
```

VECTORIZED

```
void f(int *a, int *b, int num) {
    for(int i = 0; i < num; ++i) {
        a[i] = b[i] * b[i] + 1;
    }
}
```

WHO WRITES CODE LIKE THAT?

WHO WRITES CODE LIKE THAT?

```
template<typename T, ...>
void foo(..., const T&) { ... }
```

WHO WRITES CODE LIKE THAT?

```
template<typename T, ...>
void foo(..., const T&) { ... }
```

Rings some bells?

JASON'S EXAMPLE

```
void extend(std::uint8_t *src,
            std::uint32_t *dst) {
    for(int i = 0; i < 16; ++i) {
        dst[i] = src[i];
    }
}
```

JASON'S EXAMPLE

```
void extend(std::uint8_t *src,
            std::uint32_t *dst) {
    for(int i = 0; i < 16; ++i) {
        dst[i] = src[i];
    }
}
```

Loop versioned, large unrolled code twice

JASON'S EXAMPLE

```
enum struct Data : std::uint8_t {};

void extend(Data *src,
            std::uint32_t *dst) {
    for(int i = 0; i < 16; ++i) {
        dst[i] = (std::uint8_t)src[i];
    }
}
```

JASON'S EXAMPLE

```
enum struct Data : std::uint8_t {};

void extend(Data *src,
            std::uint32_t *dst) {
    for(int i = 0; i < 16; ++i) {
        dst[i] = (std::uint8_t)src[i];
    }
}
```

Only the vectorized version

IS IT ALWAYS POSSIBLE TO UTILIZE THE TYPE BASED ALIASING RULES?

NOT VECTORIZED

```
void g(int *result, int **matrix, int height, int width) {
    for(int i = 0; i < height; ++i)
        for(int j = 0; j < width; ++j)
            result[i] += matrix[i][j];
}
```

NOT VECTORIZED

```
void g(int *result, int **matrix, int height, int width) {
    for(int i = 0; i < height; ++i)
        for(int j = 0; j < width; ++j)
            result[i] += matrix[i][j];
}
```

VECTORIZED

```
void g(int * restrict result,
       int * restrict * matrix,
       int height, int width) {
    for(int i = 0; i < height; ++i)
        for(int j = 0; j < width; ++j)
            result[i] += matrix[i][j];
}
```

`restrict`

During each execution of a block in which a restricted pointer P is declared, if some object that is accessible through P (directly or indirectly) is modified, by any means, then all accesses to that object (both reads and writes) in that block must occur through P (directly or indirectly), otherwise the behavior is undefined.

LET'S JUST ADD RESTRICT TO C++?

How to annotate the code below?

```
void g(vector<int> &result, vector<vector<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

LET'S JUST ADD RESTRICT TO C++?

How to annotate the code below?

```
void g(vector<int> &result, vector<vector<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

What would

```
vector<int restrict>
```

or

```
vector<int> restrict
```

mean?

ADDING `restrict` TO C++

- Many failed attempts, lots of unanswered questions
- Should `restrict` change the overload sets?
- Should `restrict` participate in name mangling?
- `restrict` was never designed to work with the class abstraction
- How should `restrict` carried through templates?
- Members, lambda captures, unions, ...
- C2X, n2260, clarifying `restrict`

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

Restrict is a precondition!

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

Restrict is a precondition!

Only if we had a way to describe preconditions in C++...

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

Restrict is a precondition!

Only if we had a way to describe preconditions in C++...

Voted into C++20 in June (Rapperswil meeting)

CONTRACTS TO THE RESCUE?

EXPLORING THE DESIGN SPACE

SIMPLE PRECONDITIONS

```
int f(int &a, int &b) [[expects axiom: &a != &b]] {
    a = 2;
    b = 3;
    return a;
}
```

- `f(x, x);` is undefined
- The precondition is documented
- We have two mitigations:
  - Runtime checks (with axiom removed)
  - Static analysis

SIMPLE PRECONDITIONS (LAMBDAS)

```
auto f = [](int &a, int &b) [[expects axiom: &a != &b]] {
    a = 2;
    b = 3;
    return a;
}
```

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

```
int *merge(int *a, int *b, int num)
  [[expects: __disjoint(a, b, num)]];
```

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

```
int *merge(int *a, int *b, int num)
  [[expects: __disjoint(a, b, num)]];
```

`__disjoint(a, b, c, ..., num)`?

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

```
int *merge(int *a, int *b, int num)
  [[expects: __disjoint(a, b, num)]];
```

__disjoint(a, b, c, ..., num)?

```
int *merge(int *a, int *b, int num)
  [[expects: __distinct(a) && __distinct(b)]];
```

POSSIBLE IMPLEMENTATION FOR `__disjoint`?

```
// From: P1296R0
template<typename T, typename U>
bool __disjoint(const T *pt, const U *pu, size_t n) {
    intptr_t bt = (intptr_t)pt,
             et = (intptr_t)(pt + n);
    intptr_t bu = (intptr_t)pu,
             eu = (intptr_t)(pu + n);

    return (et <= bu) || (eu <= bt);
}
```

POSSIBLE IMPLEMENTATION FOR `__disjoint`?

```
// From: P1296R0
template<typename T, typename U>
bool __disjoint(const T *pt, const U *pu, size_t n) {
    intptr_t bt = (intptr_t)pt,
             et = (intptr_t)(pt + n);
    intptr_t bu = (intptr_t)pu,
             eu = (intptr_t)(pu + n);

    return (et <= bu) || (eu <= bt);
}
```

Are we sure this is well defined? Compilers might want to have intrinsics instead.

USER DEFINED TYPES

```
int f(S a, S b)
    [[expects: __disjoint(a.member, b.member)]];
```

USER DEFINED TYPES

```
int f(S a, S b)
    [[expects: __disjoint(a.member, b.member)]];
```

```
int f(S a, S b)
    [[expects: __disjoint(a.method(), b.method())]];
```

USER DEFINED TYPES

```
int f(S a, S b)
    [[expects: __disjoint(a.member, b.member)]];
```

```
int f(S a, S b)
    [[expects: __disjoint(a.method(), b.method())]];
```

What if we need arguments? Use dummy symbols? Existentially or universally quantified?

```
int f(S a, S b)
    [[expects: __disjoint(a.method(???), b.method(???))]];
```

VIEWS TO THE RESCUE?

NON-ALIASING VIEW EXAMPLE

```
template <typename ... >
class unique_span {
    unique_span(...) [[expects: ???]];
    reference operator[](index_type idx) const
      [[ensures x: __distinct(x, this, idx)]];
};
```

```
f(unique_span(vec), unique_span(vec2));
```

BACK TO THE MATRIX EXAMPLE

```
void g(unique_span<int> result,
       vector<unique_span<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

BACK TO THE MATRIX EXAMPLE

```
void g(unique_span<int> result,
       vector<unique_span<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

Note that in real code we may want a multidimensional view or one dimensional matrix representation to avoid copying at the call site.

A NEW TYPE? ISN'T THAT HEAVY WEIGHT?

ARE THESE FUNCTIONS THE SAME?

```
double my_sqrt(double x) {
    return sqrt(x);
}
```

```
double my_sqrt(double x) {
    if (x < 0) return 0;
    return sqrt(x);
}
```

```
double my_sqrt(double x) {
    if (x < 0) throw ...;
    return sqrt(x);
}
```

ARE THESE FUNCTIONS THE SAME?

```
double my_sqrt(double x);
```

```
double my_sqrt(double x) [[expects: x >= 0]];
```

```
double my_sqrt(double x) [[expects: x >= 0]]
                         [[ensures ret: ret >= 0]];
```

ARE THESE TYPES THE SAME?

```
unique_span<int>
```

```
span<int>
```

Exercise: how different are these types?

Exercise: how different are these types?

Hint: How many methods need to be annotated?

Exercise: how different are these types?

Hint: How many methods need to be annotated?

Hint2: How many other things need to be annotated? Iterators?

Exercise: how different are these types?

Hint: How many methods need to be annotated?

Hint2: How many other things need to be annotated?

Iterators?

Is it feasible to do all that inline?

It might be a lot of work to create such types, but...

- These can be vocabulary types
- We should use such classes sparingly, as they impose burden on the caller
- Those methods/functions are now screaming that they are special and error prone
- We can do overloads!

WE ALREADY HAVE TO REASON ABOUT ALIASING

- `std::copy*`
- `memcpy` vs `memmove`
- We would get mitigations for existing UB!

RELATED WORK

- **p0856r0**: Restrict as a library feature
- **n3635**, **n4150**: Annotating alias sets
- **P1296R0**: Very similar design, cooperating with the authors
- The `malloc` attribute of GCC, `noalias` attribute of Clang
- All major compilers has `restrict` like features as extensions
- IBM XL's `#pragma disjoint`

P1296R0

- `std::disjoint` only
- Discussed at San Diego meeting
- In ealry stages, no way to get into C++20

ALIAS SETS

```
void * [[alias_set()]] malloc(size_t);
int * [[alias_set(Foo)]] p1 = ...;
int * [[alias_set(Bar), alias_set(Baz)]] p2 = ...;
int * p3 = ...;
```

THANKS FOR YOUR ATTENTION!