LLVM-Study-Notes Documentation

Release 0.1

Enna1

Oct 22, 2022

CONTENTS:

CHAPTER

ONE

IMPORTANT AND USEFUL LLVM APIS

1.1 RTTI in LLVM

不管是阅读 LLVM/Clang 的源代码，还是基于 LLVM/Clang 自己动手写一些代码时，最常用到的就是 LLVM 中的 RTTI 了，也就是 `isa<>` ，`cast<>` 和 `dyn_cast<>` ，当然还有 `cast_or_null<>` 和 `dyn_cast_or_null<>`，不过我自己好像不怎么常用这两个……

下面给一个使用 `dyn_cast<>` 的例子，在这个例子中，我们遍历函数中的所有指令，并对其中的 CallInst 指令进行一些处理：

```
for (inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i)
{
    Instruction *I = &(*i);
    if (auto *CI = dyn_cast<CallInst>(I))
    {
        /* do something*/
    }
}
```

熟悉 LLVM 的应该知道（不熟悉 LLVM 的，也能从 CallInst 类和 Instruction 类的名称推测出来）CallInst 类是继承自 Instruction 类的。上述代码就是使用 `dyn_cast<>` 将 Instruction 的对象 cast 成 CallInst 对象，如果一条 Instruction 是 CallInst ，那么 CI 就不是空指针 nullptr ，会执行 do something。

下面进入正文，`isa<>` , `cast<>` 和 `dyn_cast<>` 到底是怎么实现的。

在 LLVM-5.0.1 中，他们的实现代码位于 llvm-5.0.1.src/include/llvm/Support/Casting.h 中。

注：`isa<>` , `cast<>` 和 `dyn_cast<>` 的实现中关于模板的代码等我学了模板再填坑。

1.1.1 isa<> 的实现

可以看到 isa<> 的实现依赖于依赖于 classof 函数。

```
// The core of the implementation of isa<X> is here; To and From should be
// the names of classes.  This template can be specialized to customize the
// implementation of isa<> without rewriting it from scratch.
template <typename To, typename From, typename Enabler = void>
struct isa_impl
{
    static inline bool doit(const From &Val)
    {
        return To::classof(&Val);
    }
};
```

我们以 Value 类和 Argument 类为例来进行说明，Argument 类是由 Value 继承而来。

在 Argument 类的头文件 (llvm-5.0.1.src/include/llvm/IR/Argument.h) 中，我们可以找到 classof 函数，可以看到注释，classof 函数就是用于支持 LLVM 中的 RTTI 的。

```
/// Method for support type inquiry through isa, cast, and dyn_cast.
static bool classof(const Value *V)
{
    return V->getValueID() == ArgumentVal;
}
```

Value 类的实现中与 classof 函数相关的内容如下：

```
class Value
{
    const unsigned char SubclassID;  // Subclass identifier (for isa/dyn_cast)
    /// An enumeration for keeping track of the concrete subclass of Value that
    /// is actually instantiated. Values of this enumeration are kept in the
    /// Value classes SubclassID field. They are used for concrete type
    /// identification.
    enum ValueTy
    {
#define HANDLE_VALUE(Name) Name##Val,
#include "llvm/IR/Value.def"

    // Markers:
#define HANDLE_CONSTANT_MARKER(Marker, Constant) Marker = Constant##Val,
#include "llvm/IR/Value.def"
    };
```

(continues on next page)

(continued from previous page)

```
    /// This is used to implement the classof checks.  This should not be used
    /// for any other purpose, as the values may change as LLVM evolves.  Also,
    /// note that for instructions, the Instruction's opcode is added to
    /// InstructionVal. So this means three things:
    /// # there is no value with code InstructionVal (no opcode==0).
    /// # there are more possible values for the value type than in ValueTy
    /// enum. # the InstructionVal enumerator must be the highest valued
    /// enumerator in
    ///   the ValueTy enum.
    unsigned getValueID() const
    {
        return SubclassID;
    }
    ...
};
```

在 Value 类内部定义了一个枚举变量 ValueTy，通过 HANDLE_VALUE 宏和文件 Value.def 配合来定义各种 ValueTy 中的枚举常量，我们可以在其中看到枚举常量 ArgumentVal 的定义方式。

```
HANDLE_VALUE(Argument)
```

所以在 enum ValueTy 中的内容相当于，省略了 ArgumentVal 外的其他枚举常量：

```
enum ValueTy
{
    ...
    ArgumentVal,
    ...
};
```

然后我们看 Argument 类的构造函数的定义 (llvm-5.0.1.src/include/llvm/lib/Argument.cpp) 和 Value 类构造函数的定义 (llvm-5.0.1.src/include/llvm/lib/Value.cpp)

```
Argument::Argument(Type *Ty, const Twine &Name, Function *Par, unsigned ArgNo)
    : Value(Ty, Value::ArgumentVal), Parent(Par), ArgNo(ArgNo)
{
    ...
}

Value::Value(Type *ty, unsigned scid)
    : VTy(checkType(ty)),
      UseList(nullptr),
      SubclassID(scid),
      HasValueHandle(0),
```

(continues on next page)

(continued from previous page)

```
      SubclassOptionalData(0),
      SubclassData(0),
      NumUserOperands(0),
      IsUsedByMD(false),
      HasName(false)
{
   ...
}
```

我们重点关注的是，当构造一个 Argument 类的对象时，会手动调用基类 Value 的构造函数并且传给 Value 构造函数的第二个参数是 Value::ArgumentVal，而 Value 构造函数会把它成员变量 SubclassID 的值设置为其第二个参数的值。所以如果有一个 Argument 类的对象，然后我们拿到的是指向该 Argument 对象的 Value 类型的指针 V 时，我们以该指针作为参数调用 `isa<Argument>(V)` 时，会返回 `Argument::classof(V)` 的值，而前面我们看到，`Argument::classof(V)` 的值就是 `return V->getValueID() == ArgumentVal;`，因为在构造该 Argument 对象时，已经将其基类 Value 的 SubclassID 设置为 ArgumentVal，所以最后会返回 true，即指针 V 指向的对象是一个 Argument 类型的对象。

1.1.2 cast<> 的实现

```
// cast<X> - Return the argument parameter cast to the specified type.  This
// casting operator asserts that the type is correct, so it does not return null
// on failure.  It does not allow a null argument (use cast_or_null for that).
// It is typically used like this:
//
//  cast<Instruction>(myVal)->getParent()
//
template <class X, class Y>
inline typename std::enable_if<!is_simple_type<Y>::value,
                               typename cast_retty<X, const Y>::ret_type>::type
cast(const Y &Val)
{
    assert(isa<X>(Val) && "cast<Ty>() argument of incompatible type!");
    return cast_convert_val<
        X, const Y, typename simplify_type<const Y>::SimpleType>::doit(Val);
}

template <class X, class Y>
inline typename cast_retty<X, Y>::ret_type cast(Y &Val)
{
    assert(isa<X>(Val) && "cast<Ty>() argument of incompatible type!");
    return cast_convert_val<X, Y, typename simplify_type<Y>::SimpleType>::doit(
        Val);
```

(continues on next page)

(continued from previous page)

```
}
```

可以看到 cast<> 的实现依赖于 cast_convert_val::doit 函数，其定义如下。

```
template <class To, class FromTy>
struct cast_convert_val<To, FromTy, FromTy>
{
    // This _is_ a simple type, just cast it.
    static typename cast_retty<To, FromTy>::ret_type doit(const FromTy &Val)
    {
        typename cast_retty<To, FromTy>::ret_type Res2 =
            (typename cast_retty<To, FromTy>::ret_type) const_cast<FromTy &>(
                Val);
        return Res2;
    }
};
```

先使用 C++ const_cast 然后对 const_cast 的结果进行 C 风格的强制类型转换。

1.1.3 dyn_cast<> 的实现

```
// dyn_cast<X> - Return the argument parameter cast to the specified type.  This
// casting operator returns null if the argument is of the wrong type, so it can
// be used to test for a type as well as cast if successful.  This should be
// used in the context of an if statement like this:
//
//  if (const Instruction *I = dyn_cast<Instruction>(myVal)) { ... }
//

template <class X, class Y>
LLVM_NODISCARD inline
    typename std::enable_if<!is_simple_type<Y>::value,
                            typename cast_retty<X, const Y>::ret_type>::type
    dyn_cast(const Y &Val)
{
    return isa<X>(Val) ? cast<X>(Val) : nullptr;
}
```

可以看到 dyn_cast 的是通过三元运算符实现的，如果 isa<X>(val) 返回 true (val 是 X 类的一个对象)，则将 val cast 为 X 类后返回，否则返回空指针 nullptr 。

1.1.4 让 LLVM-style RTTI 支持自己的编写的类

假设要编写如下继承关系的类

```
| Shape
  | Square
    | SpecialSquare
  | Circle
```

了解 LLVM 中 RTTI 的实现后，我们想要让其支持自己编写的类，模仿 Value 类和 Argument 类的写法，声明一个枚举变量，在子类构造函数中显示调用父类构造函数并传递给父类构造函数一个表示子类类型的枚举常量，还需要定义 `classof` 函数。

具体的实现代码如下：

```
#include "llvm/Support/Casting.h"
#include <iostream>
#include <vector>
using namespace llvm;

class Shape
{
public:
    // 类似 class Value 中 enum ValueTy 的定义
    enum ShapeKind
    {
        /* Square Kind Begin */
        SK_SQUARE,
        SK_SEPCIALSQUARE,
        /* Square Kind end */
        SK_CIRCLE,
    };

private:
    const ShapeKind kind_;

public:
    Shape(ShapeKind kind) : kind_(kind) {}

    ShapeKind getKind() const
    {
        return kind_;
    }

    virtual double computeArea() = 0;
```

(continues on next page)

(continued from previous page)

```
};

class Square : public Shape
{
public:
    double side_length_;

public:
    Square(double side_length) : Shape(SK_SQUARE), side_length_(side_length) {}

    Square(ShapeKind kind, double side_length)
        : Shape(kind), side_length_(side_length)
    {
    }

    double computeArea() override
    {
        return side_length_ * side_length_;
    }

    static bool classof(const Shape *s)
    {
        return s->getKind() >= SK_SQUARE && s->getKind() <= SK_SEPCIALSQUARE;
    }
};

class SepcialSquare : public Square
{
public:
    double another_side_length_;

public:
    SepcialSquare(double side_length, double another_side_length)
        : Square(SK_SEPCIALSQUARE, side_length),
          another_side_length_(another_side_length)
    {
    }

    double computeArea() override
    {
        return side_length_ * another_side_length_;
    }
```

(continues on next page)

(continued from previous page)

```
    static bool classof(const Shape *s)
    {
        return s->getKind() == SK_SEPCIALSQUARE;
    }
};

class Circle : public Shape
{
public:
    double radius_;

public:
    Circle(double radius) : Shape(SK_CIRCLE), radius_(radius) {}

    double computeArea() override
    {
        return 3.14 * radius_ * radius_;
    }

    static bool classof(const Shape *s)
    {
        return s->getKind() == SK_CIRCLE;
    }
};

int main()
{
    Square s1(1);
    SepcialSquare s2(1, 2);
    Circle s3(3);
    std::vector<Shape *> v{ &s1, &s2, &s3 };
    for (auto i : v)
    {
        if (auto *S = dyn_cast<Square>(i))
        {
            std::cout << "This is a Square object\n";
            std::cout << "Area is : " << S->computeArea() << "\n";
        }
        if (auto *SS = dyn_cast<SepcialSquare>(i))
        {
            std::cout << "This is a SepcialSquare object\n";
            std::cout << "Area is : " << SS->computeArea() << "\n";
        }
```

(continues on next page)

(continued from previous page)

```
        if (auto *C = dyn_cast<Circle>(i))
        {
            std::cout << "This is a Circle object\n";
            std::cout << "Area is : " << C->computeArea() << "\n";
        }
        std::cout << "-----\n";
    }

    return 0;
}
```

参考链接：

1. http://llvm.org/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates
2. http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html
3. https://stackoverflow.com/questions/6038330/how-is-llvm-isa-implemented

1.2 StringRef & Twine

在 LLVM 的 API 中为了高效地传递字符串，定义了 StringRef 类和 Twine 类。

1.2.1 StringRef

StringRef 类的定义位于 llvm-5.0.1.src/include/llvm/ADT/StringRef.h

StringRef 类用于表示对常量字符串的一个引用，它支持我们常用的 std::string 所支持的那些操作，但是 StringRef 不需要动态内存分配 (heap allocation)

因为 StringRef 类的对象所占用的空间足够小，所以我们在使用 StringRef 时应该总是使用值传递的方式。

因为 StringRef 类中包含一个指向某个内存空间（常量字符串所在的内存空间）的一个指针，所以只有在保证指向的内存空间不会被释放的情况下，保存一个 StringRef 的对象才是安全的。否则可能会发生 UAF (Use After Free)。

StringRef 类的定义（省略了一些内容）如下：

```
class StringRef
{
public:
    static const size_t npos = ~size_t(0);

    using iterator = const char *;
    using const_iterator = const char *;
```

(continues on next page)

(continued from previous page)

```
    using size_type = size_t;

private:
    /// The start of the string, in an external buffer.
    const char *Data = nullptr;

    /// The length of the string.
    size_t Length = 0;

public:
    ......

    /// Construct a string ref from an std::string.
    LLVM_ATTRIBUTE_ALWAYS_INLINE
    /*implicit*/ StringRef(const std::string &Str)
        : Data(Str.data()), Length(Str.length()) {}

    /// str - Get the contents as an std::string.
    LLVM_NODISCARD
    std::string str() const
    {
        if (!Data)
            return std::string();
        return std::string(Data, Length);
    }

    ......
};
```

可以看到，StringRef 类有两个成员变量：Data 和 Length，Data 是一个指向 const char 的指针，Length 用于存储字符串的长度。

与 std::string 类似，StringRef 支持 `data`, `empty`, `size`, `startswith`, `endswith` 等常用的函数；当然 StringRef 还支持一些 std::string 中没有的成员函数，如 `equals`, `split`, `trim` 等。

StringRef 支持多种构造方式，可以通过 C style null-terminated string , std::string 或者通过指定 StringRef 的两个成员变量 Data 和 Length 来构造 StringRef 。

StringRef 还支持一个 `str` 函数，该函数返回一个 std::string (以 StringRef 的成员变量 Data 和 Length 作为参数调用 std::string 构造函数来得到) 。

在使用 StringRef 时，要注意一下几点限制：

1. 不能直接将一个 StringRef 类型的对象转换为一个 const char * ，因为由于 StringRef 成员变量 Length 的存在，StringRef 所指向的字符串是可以包含 “\0” 的，例如: `StringRef("\0baz", 4)`
2. StringRef 不能控制其指向的常量字符串的生命周期，所以通常不应该以 StringRef 的对象作为某一个类

的成员变量

3. 同样地，如果一个函数的返回值是一个通过计算得到的字符串，那么该函数的返回值类型不应该用 StringRef ，而应该使用 std::string

4. StringRef 不允许对其指向的常量字符串的字节内容进行修改 (mutate the pointed-to string bytes, insert or remove bytes from the range)，对于这样的操作，应使用 Twine 与 StringRef 进行配合。

1.2.2 Twine

Twine 类头文件位于 llvm-5.0.1.src/include/llvm/ADT/Twine.h，源文件位于 llvm-5.0.1.src/lib/Support/Twine.cpp 。

Twine 类用于高效地表示字符串的拼接操作。比如在 LLVM 的 API 中一种常见范式就是以已有的一条指令的名称再加上一个后缀的方式为一条新的指令命名：

```
New = CmpInst::Create(..., SO->getName() + ".cmp");
```

Twine 类是一个高效且轻量的 rope，可以由字符串 (C-strings, std::string, StringRef) 之间的 + 运算符的结果隐式构造得到 (上面的例子就是由 StringRef 和 C-strings 进行 + 运算后隐式得到 Twine)。Twine 类只有在字符串之间的拼接结果被实际需要时，才真正执行拼接操作，因此可以避免由于对字符串拼接产生的临时结果进行构造所带来的堆内存分配操作的开销。

例如，下面的代码片段：

```
void foo(const Twine &T);
...
StringRef X = ...
unsigned i = ...
foo(X + "." + Twine(i));
```

函数 `foo` 的参数是由多个字符串拼接而来，假设 StringRef 指向的常量字符串是“arg”，unsigned i 为 123，此拼接并不会构造出临时的中间字符串“arg”或者“arg.”，而是只产生“arg.123”来作为函数 `foo` 的参数。

需要注意的是，因为 Twine 的内部结点（Twine 是以二叉树实现的）是构造在栈上的，在该条语句（构造 Twine 的那条语句）结束之后，Twine 对象就会被销毁，通常 Twine 只应该被用作函数的参数，而且应该以 `const Twine &` 的方式被使用，如上面的示例代码。

下面的使用方式是错误的!!!：

```
void foo(const Twine &T);
...
StringRef X = ...
unsigned i = ...
const Twine &Tmp = X + "." + Twine(i);
foo(Tmp);
```

因为在 Tmp 作为函数 `foo` 的参数之前，已经结束生命周期被销毁。

关于 Twine 的源码实现。

首先是 Twine 的构造函数，Twine 有很多的构造函数，其中包含了支持隐式类型转换的构造函数：

```
/*implicit*/ Twine(const char *Str);
/*implicit*/ Twine(const std::string &Str);
/*implicit*/ Twine(const StringRef &Str);
...... //省略
```

Twine 是以二叉树实现的，在 Twine 的内部使用枚举变量 enum NodeKind 来表示结点的可能的类型，因为结点可能类型有很多，所以使用 union 作为结点的值的类型。

```
class Twine
{
    enum NodeKind : unsigned char
    {
        NullKind,
        EmptyKind,
        TwineKind,
        CStringKind,
        StdStringKind,
        StringRefKind,
        SmallStringKind,
        FormatvObjectKind,
        CharKind,
        DecUIKind,
        DecIKind,
        DecULKind,
        DecLKind,
        DecULLKind,
        DecLLKind,
        UHexKind
    };

    union Child {
        const Twine *twine;
        const char *cString;
        const std::string *stdString;
        const StringRef *stringRef;
        const SmallVectorImpl<char> *smallString;
        const formatv_object_base *formatvObject;
        char character;
        unsigned int decUI;
        int decI;
        const unsigned long *decUL;
```

(continues on next page)

(continued from previous page)

```
        const long *decL;
        const unsigned long long *decULL;
        const long long *decLL;
        const uint64_t *uHex;
    };

    Child LHS;
    Child RHS;
    NodeKind LHSKind;
    NodeKind RHSKind;
    ..... // 省略
};
```

我们重点关注一下，关于拼接的实现

```
inline Twine Twine::concat(const Twine &Suffix) const
{
    // Concatenation with null is null.
    if (isNull() || Suffix.isNull())
        return Twine(NullKind);

    // Concatenation with empty yields the other side.
    if (isEmpty())
        return Suffix;
    if (Suffix.isEmpty())
        return *this;

    // Otherwise we need to create a new node, taking care to fold in unary
    // twines.
    Child NewLHS, NewRHS;
    NewLHS.twine = this;
    NewRHS.twine = &Suffix;
    NodeKind NewLHSKind = TwineKind, NewRHSKind = TwineKind;
    if (isUnary())
    {
        NewLHS = LHS;
        NewLHSKind = getLHSKind();
    }
    if (Suffix.isUnary())
    {
        NewRHS = Suffix.LHS;
        NewRHSKind = Suffix.getLHSKind();
    }
```

(continues on next page)

(continued from previous page)

```
    return Twine(NewLHS, NewLHSKind, NewRHS, NewRHSKind);
}

inline Twine operator+(const Twine &LHS, const Twine &RHS)
{
    return LHS.concat(RHS);
}
```

实现拼接的是成员函数 `concat`，很简单，就是将左操作数和右操作数分别作为新的 Twine 对象的左结点和右结点来构造一个新的 Twine 对象，对左操作数、左操作数只含有一个结点的情况做了特别处理。函数 `concat` 只是构造了拼接后的字符串的 Twine 表示，并没有生成 std::string 。

如果要得到拼接后的字符串 std::string ，需要调用函数 `std::string Twine::str() const` ，该函数通过递归遍历左结点和右结点来产生实际的拼接结果 std::string。

参考链接：http://llvm.org/docs/ProgrammersManual.html#passing-strings-the-stringref-and-twine-classes

---

- genindex
- modindex
- search

CHAPTER

TWO

LLVM IR

2.1 ConstantExpr

2.1.1 Constant

在 LLVM IR 中有这样一种值: constants，这种值在 LLVM IR 中能独立于基本块和函数存在，包括数、全局变量、常量字符数组等。我们拿 LLVM IR 中的一条指令为例来简单说明一下:

```
llvm ir call void @llvm.memset.p0i8.i64(i8* nonnull align 8 %some, i8 0, i64
40, i1 false)
```

`llvm.memset.*` 是 LLVM 中的一类 intrinsics 指令，LLVM 为 C 标准库中的一些函数提供了 intrinsics 实现，有了这些 intrinsics 函数，就允许编译器前端 (如 clang) 将有关指针对齐的相关信息传给代码生成器，能够使代码生成这一过程变得更加高效。

回到 constants 上来，注意上述 callinst 的后三个参数 `i8 0, i64 40, i1 false`，倒数第一个参数 `i1 false` 是 boolean constants，实际上 `true` 和 `false` 都是为 i1 type 的 constants。倒数第二个参数和倒数第三个参数 `i8 0`, `i64 40` 也同样是 constant，具体来说是 integer constant。类似的还有 floating-point constants, null pointer constants (‘null’ in LLVM IR) 等。上面提到的都是 simple constants，同样也有 complex constants，比如 structure constants, array constants (包括字符数组), vector constants 等。下面就是一个 character array constants:

```
@.str.123 = private unnamed_addr constant [5 x i8] c"YES!\00", align 1
```

Constants 除了包括上面提到的 simple constants 和 complex constants，还有 Global Variable and Function Addresses，Undefined Values，Poison Values，Addresses of Basic Blocks，Constant Expressions。其实上面 `@.str.123` 就是一个 global variable。我们本文主要讨论 ConstantExpr (即 Constant Expressions)。

2.1.2 ConstantExpr

ConstantExpr is a constant value that is initialized with an expression using other constant values，也就是说如果一个表达式的所有操作数都是 constant 的话，那么这个表达式就是一个 constant experssion，当然它同样也是一个 constant。

一个具体的例子：

```
$ cat a.c
int a;
int main()
{
    return 5+(long)(&a);
}
```

将 a.c 通过 clang 编译得到 LLVM IR 和可执行程序：

```
$ clang -S -O2 -emit-llvm a.c -o a.o2.ll
$ clang -O2 a.c -o a.o2.out
```

`a.o2.ll` 的部分内容如下：

```
@a = common dso_local global i32 0, align 4

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
entry:
  ret i32 trunc (i64 add (i64 ptrtoint (i32* @a to i64), i64 5) to i32)
}
```

这里 `trunc (i64 add (i64 ptrtoint (i32* @a to i64), i64 5) to i32)` 整个表达式是一个 ConstantExpr。

使用 objdump 查看 main 函数对应的汇编代码：

```
mov    $0x601039,%eax
retq
```

可见在 LLVM IR 中的 ContantExpr 在汇编代码中实际上是一个 constant value。实际上 ContantExpr 经过很多个阶段的处理后 (some in the backend, some by the linker, some by the dynamic loader)，最终在程序被加载时成为一个 constant value。

2.1.3 BreakConstantExpr

每一种 ContantExpr 都对应一种 Instruction，在 LLVM 中经常可以看到这样的实现 `visitConstantExpr(ConstantExpr *CE)` 函数的写法：

```
void visitConstantExpr(ConstantExpr *CE)
{
    switch (CE->getOpcode())
    {
    case Instruction::Trunc:
    case Instruction::ZExt:
    case Instruction::SExt:
    case Instruction::FPTrunc:
    case Instruction::FPExt:
    case Instruction::FPToUI:
    case Instruction::FPToSI:
    case Instruction::UIToFP:
    case Instruction::SIToFP:
    case Instruction::PtrToInt:
    case Instruction::IntToPtr:
    case Instruction::BitCast:
    case Instruction::AddrSpaceCast:
    case Instruction::GetElementPtr:
    case Instruction::Select:
    case Instruction::ICmp:
    case Instruction::FCmp:
    case Instruction::ExtractElement:
    case Instruction::InsertElement:
    case Instruction::ShuffleVector:
    case Instruction::ExtractValue:
    case Instruction::InsertValue:
    case Instruction::Add:
    case Instruction::Sub:
    case Instruction::FSub:
    case Instruction::Mul:
    case Instruction::FMul:
    case Instruction::UDiv:
    case Instruction::SDiv:
    case Instruction::FDiv:
    case Instruction::URem:
    case Instruction::SRem:
    case Instruction::FRem:
    case Instruction::And:
    case Instruction::Or:
    case Instruction::Xor:
```

(continues on next page)

(continued from previous page)

```
    case Instruction::Shl:
    case Instruction::LShr:
    case Instruction::AShr:
    default:
        llvm_unreachable("Unknown constantexpr type encountered!");
    }
}
```

`visitConstantExpr` 的用处就是先判断某条 Instruction 的操作数是否为 ConstantExpr，如果是则调用该函数进行处理。

但是在我接触的一些基于 LLVM IR 做程序分析的开源的工具中 (如 SVF)，发现这些工具通常会对 LLVM IR 进行处理：如果一条 Instruction 的某个操作数是 ConstantExpr ，那么将该 ConstantExpr 转换为对应 Instruction 并插入到使用该 ConstantExpr 的 Instruction 之前，将该 ConstantExpr 的所有使用的地方替换为新插入的转换后的 Instruction。该功能的实现代码可以参考 https://github.com/Enna1/LLVM-Clang-Examples/tree/master/break-constantexpr 。

将前面提到的 LLVM IR 文件 `a.o2.ll` 经过 BreakConstantExpr 处理后，得到的 LLVM IR 如下，可以看到已经将 ConstantExpr 转换为了 Instruction 。

```
@a = common dso_local global i32 0, align 4

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
entry:
  %0 = ptrtoint i32* @a to i64
  %1 = add i64 %0, 5
  %2 = trunc i64 %1 to i32
  ret i32 %2
}
```

2.1.4 参考链接:

1. http://llvm.org/docs/LangRef.html#constant-expressions
2. http://lists.llvm.org/pipermail/llvm-dev/2017-March/110885.html

---

- genindex
- modindex
- search

CHAPTER

THREE

SSA

3.1 SSA-construction

本文中的内容基本上均来自 CMU 15-745 的课程讲义。

标准的 SSA 构建算法分成两步：

1. Place all Φ()
2. Rename all variables

在讲 SSA 构建算法之前，需要一些基础知识。

3.1.1 Basics

Dominator

- N dominates M (N dom M) <==> 在 CFG 上，从 entry node 到 M 的所有路径都经过 N
- 真支配 (strictly dominate, sdom)，如果 N dom M 并且 N != M，则 N sdom M
- 直接支配 (immediate dominate, idom)，如果 N dom M 并且不存在 N'，使 N dom N'，N' dom M，则 N idom M

Dominator Tree

- 父节点是子节点的直接支配节点

Dominance Frontier

对于图节点 N，The Dominance Frontier of node N 是一个集合，该集合包含 W 如果 W 满足以下条件：

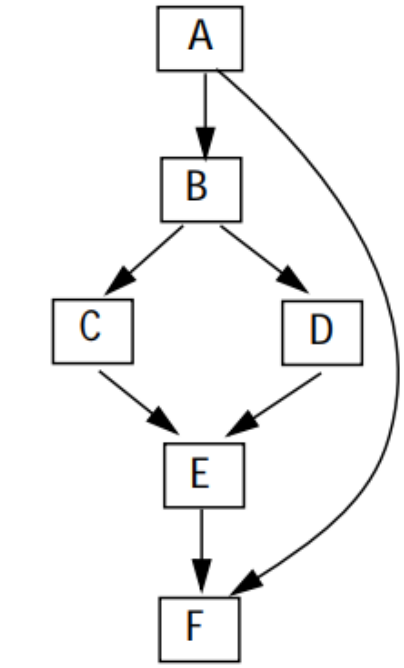
1. N 是 W 的某个前驱结点的支配节点
2. N 不是 W 的真支配节点

即 DF(N) = { W | N dom pred(W) AND !(N sdom W)}

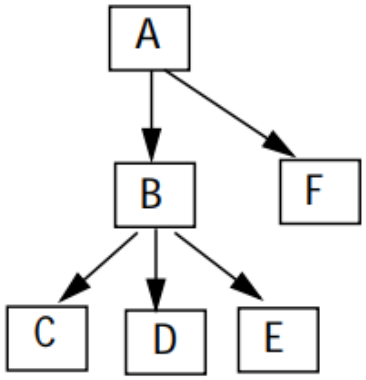
Computing the Dominance Frontier: Algorithm

```
for each node n in the post-order traversal of the D-tree
compute-DF(n)
        S = {}
        foreach node c in succ[n]
                if !(n sdom c)
                        S = S ∪ { c }
        foreach child a of n in D-tree
                compute-DF(a)
                foreach x in DF[a]
                        if !(n dom x)
                                S = S ∪ { x }
        DF[n] = S
```

下图是一个计算 Dominance Frontier 的例子：



Control Flow Graph



Dominator Tree

Block	A	B	C	D	E	F
Dominance Frontier	ϕ	{F}	{E}	{E}	{F}	ϕ

Iterated Dominance Frontier

首先定义节点集合的 Dominance Frontier：设节点集合 S = { X0, X1, X2 ⋯XN }，那么 DF(S) = DF(X0) U DF(X1) U DF(X2) ⋯U DF(XN)

节点集合的 Iterated Dominance Frontier 记作 DF+(S)，DF+(S) 就是不断地计算 S 及其 DF 集合的 DF 集合，直至不动点。

以上面计算 Dominance Frontier 的例子来计算 Iterated Dominance Frontier：

1. DF1({A, B, C}) = DF({A, B, C}) = { E, F}

2. DF2({A, B, C}) = DF({A, B, C} U DF1({A, B, C})) = DF({A, B, C, E, F}) = { E, F }
3. DF+({ A, B, C}) = { E, F }

3.1.2 Place all Φ()

Using Dominance Frontier to Place Φ()

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DominanceFrontier(defsite)
      - if we haven't put Φ() in node, then put one in
      - if this node didn't define the variable before, then add this node to the defsites (because Φ counts as def)

- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of Φ() neccesary

思考一下，这里说“This essentially computes the Iterated Dominance Frontier on the fly”为什么?

上述算法首先计算了 DF(defsites)，对于 DF(defsites) 中的元素，如果该元素不在 defsites 中就将其加入到 defsites 中，再计算 DF(new defsites)，其实就是 DF (defsites U DF(defsites))，如此直至没有新的节点加入到 defsites 中。可见这就是在计算 Iterated Dominance Frontier。

Using Dominance Frontier to Place Φ(): Algorithm

```
foreach node n {
   foreach variable v defined in n {
     orig[n] ∪= {v}          /* variables defined in basic block n */
     defsites[v] ∪= {n}      /* basic blocks that define variable v */
   }
}
foreach variable v {
   W = defsites[v]                          /* work list of basic blocks */
   while W not empty {
     n = remove node from W
     foreach y in DF[n]
       if y ∉ PHI[v] {
         insert "v ← Φ(v,v,…)" at top of y
         PHI[v] = PHI[v] ∪ {y}              /* BBs containing a Φ for v */
         if v ∉ orig[y]: W = W ∪ {y}        /* add BB to work list */
       }
   }
}
```

3.1.3 Rename all variables

Rename 算法如下，参数 B 表示基本块。

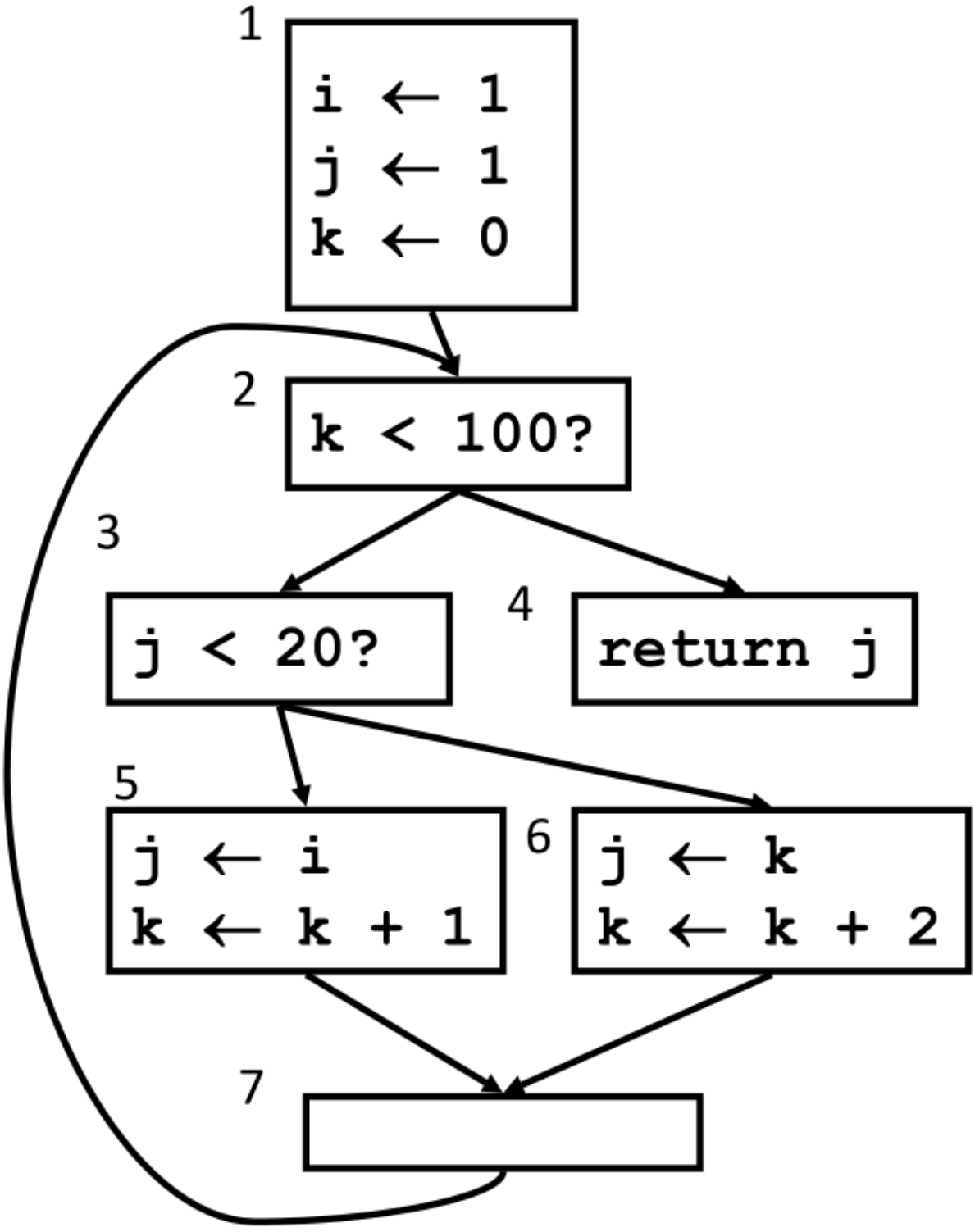
```
rename(B):
for each assignment in B:
    replace (non-Φ) use of v with top of stack(v)
    replace def of v with v_new, push v_new onto stack(v)
for each successor S of B in CFG:
    replace k'th arg. of Φ(v,…v) with top of stack(v),
          where B is k'th predecessor of S
call rename(C) on all children C of B in D-tree
pop all defs in B from stacks
```

当完成 Phase1 (place all Φ()) 后，执行 phase2 (rename all variables) 就是调用 rename(entry block)，entry block 就

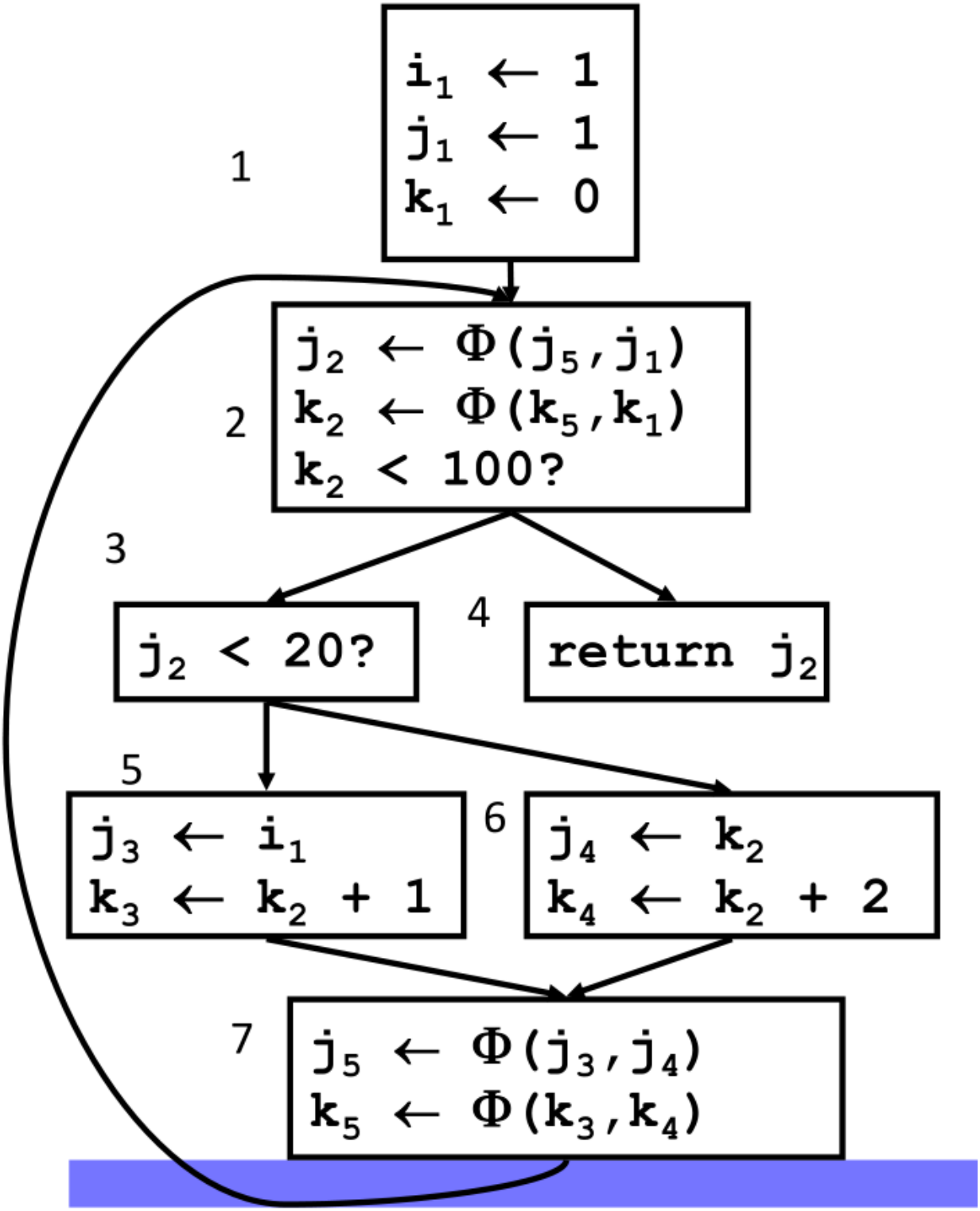
是 CFG 的入口基本块。

3.1.4 Example

Normal Form IR



SSA Form IR


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k_1 \leftarrow 0$
2
$j_2 \leftarrow \Phi(j_5, j_1)$
$k_2 \leftarrow \Phi(k_5, k_1)$
$k_2 < 100?$
3
$j_2 < 20?$
4
return j_2
5
$j_3 \leftarrow i_1$
$k_3 \leftarrow k_2 + 1$
6
$j_4 \leftarrow k_2$
$k_4 \leftarrow k_2 + 2$
7
$j_5 \leftarrow \Phi(j_3, j_4)$
$k_5 \leftarrow \Phi(k_3, k_4)$

3.1.5 LLVM

Basics 部分的内容在 LLVM 中均有实现，Iterated Dominance Frontier 的实现位于 `llvm-7.0.0.src/include/llvm/Analysis/IteratedDominanceFrontier.h` 和 `llvm-7.0.0.src/lib/Analysis/IteratedDominanceFrontier.cpp`。

```
template <class NodeTy, bool IsPostDom>
class IDFCalculator {
 public:
  IDFCalculator(DominatorTreeBase<BasicBlock, IsPostDom> &DT)
      : DT(DT), useLiveIn(false) {}

  void setDefiningBlocks(const SmallPtrSetImpl<BasicBlock *> &Blocks) {
    DefBlocks = &Blocks;
  }

  void setLiveInBlocks(const SmallPtrSetImpl<BasicBlock *> &Blocks) {
    LiveInBlocks = &Blocks;
    useLiveIn = true;
  }

  void resetLiveInBlocks() {
    LiveInBlocks = nullptr;
    useLiveIn = false;
  }

  void calculate(SmallVectorImpl<BasicBlock *> &IDFBlocks);

private:
 DominatorTreeBase<BasicBlock, IsPostDom> &DT;
 bool useLiveIn;
 const SmallPtrSetImpl<BasicBlock *> *LiveInBlocks;
 const SmallPtrSetImpl<BasicBlock *> *DefBlocks;
};
```

`IDFCalculator` 的构造函数很简单，初始化了成员变量 `DominatorTreeBase<BasicBlock, IsPostDom> &DT` 和 `bool useLiveIn`，成员变量 `bool useLiveIn` 就是标识是否使用 `LiveInBlocks`，`LiveInBlocks` 就是这样的基本块集合，value 在这些基本块的入口是活跃的。`DefBlocks` 就是那些对 value 进行定值的基本块。

函数 `setDefiningBlocks()` 和函数 `setLiveInBlocks()` 就是用来设置成员变量 `DefBlocks` 和 `LiveInBlocks` 的。

`IDFCalculator` 真正来计算 Iterated Dominance Frontier 的函数就是成员函数 `calculate()`。类 `IDFCalculator` 的使用方式就是首先定义该类的一个对象，然后调用成员函数 `setDefiningBlocks()` 和 `setLiveInBlocks()`，其中对 `setLiveInBlocks()` 的调用是可选的。最后调用成员函数 `calculate()`

计算出 Iterated Dominance Frontier。

成员函数 `calculate()` 的定义如下：

```
template <class NodeTy, bool IsPostDom>
void IDFCalculator<NodeTy, IsPostDom>::calculate(
    SmallVectorImpl<BasicBlock *> &PHIBlocks) {
  // Use a priority queue keyed on dominator tree level so that inserted nodes
  // are handled from the bottom of the dominator tree upwards. We also augment
  // the level with a DFS number to ensure that the blocks are ordered in a
  // deterministic way.
  typedef std::pair<DomTreeNode *, std::pair<unsigned, unsigned>>
      DomTreeNodePair;
  typedef std::priority_queue<DomTreeNodePair, SmallVector<DomTreeNodePair, 32>,
                              less_second> IDFPriorityQueue;
  IDFPriorityQueue PQ;

  DT.updateDFSNumbers();

  for (BasicBlock *BB : *DefBlocks) {
    if (DomTreeNode *Node = DT.getNode(BB))
      PQ.push({Node, std::make_pair(Node->getLevel(), Node->getDFSNumIn())});
  }

  SmallVector<DomTreeNode *, 32> Worklist;
  SmallPtrSet<DomTreeNode *, 32> VisitedPQ;
  SmallPtrSet<DomTreeNode *, 32> VisitedWorklist;

  while (!PQ.empty()) {
    DomTreeNodePair RootPair = PQ.top();
    PQ.pop();
    DomTreeNode *Root = RootPair.first;
    unsigned RootLevel = RootPair.second.first;

    // Walk all dominator tree children of Root, inspecting their CFG edges with
    // targets elsewhere on the dominator tree. Only targets whose level is at
    // most Root's level are added to the iterated dominance frontier of the
    // definition set.

    Worklist.clear();
    Worklist.push_back(Root);
    VisitedWorklist.insert(Root);

    while (!Worklist.empty()) {
      DomTreeNode *Node = Worklist.pop_back_val();
```

(continues on next page)

(continued from previous page)

```
      BasicBlock *BB = Node->getBlock();
      // Succ is the successor in the direction we are calculating IDF, so it is
      // successor for IDF, and predecessor for Reverse IDF.
      for (auto *Succ : children<NodeTy>(BB)) {
        DomTreeNode *SuccNode = DT.getNode(Succ);

        // Quickly skip all CFG edges that are also dominator tree edges instead
        // of catching them below.
        if (SuccNode->getIDom() == Node)
          continue;

        const unsigned SuccLevel = SuccNode->getLevel();
        if (SuccLevel > RootLevel)
          continue;

        if (!VisitedPQ.insert(SuccNode).second)
          continue;

        BasicBlock *SuccBB = SuccNode->getBlock();
        if (useLiveIn && !LiveInBlocks->count(SuccBB))
          continue;

        PHIBlocks.emplace_back(SuccBB);
        if (!DefBlocks->count(SuccBB))
          PQ.push(std::make_pair(
              SuccNode, std::make_pair(SuccLevel, SuccNode->getDFSNumIn())));
      }

      for (auto DomChild : *Node) {
        if (VisitedWorklist.insert(DomChild).second)
          Worklist.push_back(DomChild);
      }
    }
  }
}
```

虽然看起来函数 `calculate()` 有很多行，但是实际上还是比较清晰的。该函数中使用优先队列来存储那些对 value 进行定值的基本块，基本块在 Dominator Tree 上的 level 越低（越靠近叶子节点），基本块在 Dominator Tree 上的 DFS 访问次序越小，就越排在优先队列的前面。优先队列的排序方式对应 Computing the Dominance Frontier: Algorithm 中以 post-order traversal of the Dominator tree 的方式来依次处理 Dominator tree 上的每个节点。后面该函数的迭代方式就是将计算 Dominance Frontier 和 Place Φ() 的算法结合在了一起。

3.1.6 Reference

https://www.cs.cmu.edu/~15745/lectures/L11-SSA.pdf

3.2 Mem2Reg

mem2reg 是一个 LLVM transform pass，它用于将 LLVM IR 转换为 SSA 形式的 LLVM IR。具体是什么意思?

LLVM IR 借助 “memory 不是 SSA value” 的特点开了个后门。编译器前端在生成 LLVM IR 时，可以选择不生成真正的 SSA 形式，而是把局部变量生成为 alloca/load/store 形式：

- 用 alloca 指令来” 声明 “变量，得到一个指向该变量的指针
- 用 store 指令来把值存在变量里
- 用 load 指令来把值读出为 SSA value

LLVM 在 mem2reg 这个 pass 中，会识别出上述这种模式的 alloca，把它提升为 SSA value，在提升为 SSA value 时会对应地消除 store 与 load，修改为 SSA 的 def-use/use-def 关系，并且在适当的位置安插 Phi 和进行变量重命名。

实际上 Clang 生成的就是这样的 alloca/load/store 形式的 LLVM IR：

对于如下函数 foo：

```
int foo(int x, int cond)
{
    if (cond > 0)
        x = 1;
    else
        x = -1;
    return x;
}
```

`clang -Xclang -disable-O0-optnone -O0 -emit-llvm -S foo.c` 得到的 LLVM IR：

```
define dso_local i32 @foo(i32 %x, i32 %cond) #0 {
entry:
  %x.addr = alloca i32, align 4
  %cond.addr = alloca i32, align 4
  store i32 %x, i32* %x.addr, align 4
  store i32 %cond, i32* %cond.addr, align 4
  %0 = load i32, i32* %cond.addr, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
```

(continues on next page)

(continued from previous page)

```
  store i32 1, i32* %x.addr, align 4
  br label %if.end

if.else:                                          ; preds = %entry
  store i32 -1, i32* %x.addr, align 4
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %1 = load i32, i32* %x.addr, align 4
  ret i32 %1
}
```

可以看到，局部变量都是在函数的入口基本块通过 alloca 来“声明”的，并且后续对局部变量的赋值都是通过 store 指令，获取局部变量的值都是通过 load 指令，正是前面所说的 alloca/load/store 形式的 LLVM IR。

`opt -S -mem2reg -o foo.m2r.ll foo.ll` 对上述 LLVM IR 运行 mem2reg pass 后得到的新的 LLVM IR：

```
define dso_local i32 @foo(i32 %x, i32 %cond) #0 {
entry:
  %cmp = icmp sgt i32 %cond, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  br label %if.end

if.else:                                          ; preds = %entry
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %x.addr.0 = phi i32 [ 1, %if.then ], [ -1, %if.else ]
  ret i32 %x.addr.0
}
```

可以看到 alloca/load/store 形式的局部变量被提升为了 SSA value，并且在 `if.end` 基本块中还安插了 Phi。

mem2reg pass 的代码实现位于：`llvm-7.0.0.src/lib/Transforms/Utils/Mem2Reg.cpp` `llvm-7.0.0.src/lib/Transforms/Utils/PromoteMemoryToRegister.cpp`

3.2.1 算法描述

下面详细先看一下 mem2reg 的算法描述（转自 R 大在知乎上的回答，见本文的参考链接）：

1. LLVM assumes that all locals are introduced in the entry basic block of a function with an alloca instruction. LLVM also assumes that all allocas appear at the start of the entry block continuously. This assumption could be easily violated by the front-end, but it is a reasonable assumption to make.
2. For each alloca seen in step 1, LLVM checks if it is promotable based on the use of this local. It is deemed promotable iff:
    1. It is not used in a volatile instruction.
    2. It is loaded or stored directly, i.e, its address is not taken.
3. For each local selected for promotion in step 2, a list of blocks which use it, and a list of block which define it are maintained by making a linear sweep over each instruction of each block.
4. Some basic optimizations are performed:
    1. Unused allocas are removed.
    2. If there is only one defining block for an alloca, all loads which are dominated by the definition are replaced with the value.
    3. allocas which are read and written only in a block can avoid traversing CFG, and PHI-node insertion by simply inserting each load with the value from nearest store.
5. A dominator tree is constructed.
6. For each candidate for promotion, points to insert PHI nodes is computed as follows:
    1. A list of blocks which use it without defining it (live-in blocks or upward exposed blocks) are determined with the help of using and defining blocks created in Step 3.
    2. A priority queue keyed on dominator tree level is maintained so that inserted nodes corresponding to defining blocks are handled from the bottom of the dominator tree upwards. This is done by giving each block a level based on its position in the dominator tree.
    3. For each node —root, in the priority queue:
        1. Iterated dominance frontier of a definition is computed by walking all dominator tree children of root, inspecting their CFG edges with targets elsewhere on the dominator tree. Only targets whose level is at most root level are added to the iterated dominance frontier.
        2. PHI-nodes are inserted at the beginning in each block in the iterated dominance frontier computed in the previous step. There will be predecessor number of dummy argument to the PHI function at this point.
7. Once all PHI-nodes are prepared, a rename phase start with a worklist containing just entry block as follows:
    1. A hash table of IncomingVals which is a map from a alloca to its most recent name is created. Most recent name of each alloca is an undef value to start with.

2. While (worklist != NULL)
    1. Remove block B from worklist and mark B as visited.
    2. For each instruction in B:
        1. If instruction is a load instruction from location L (where L is a promotable candidate) to value V, delete load instruction, replace all uses of V with most recent value of L i.e, IncomingVals[L].
        2. If instruction is a store instruction to location L (where L is a promotable candidate) with value V, delete store instruction, set most recent name of L i.e, IncomingVals[L] = V.
        3. For each PHI-node corresponding to a alloca —L , in each successor of B, fill the corresponding PHI-node argument with most recent name for that location i.e, IncomingVals[L].
    3. Add each unvisited successor of B to worklist.

3.2.2 算法实现

mem2reg pass 对应的类是 `PromoteLegacyPass`

```
struct PromoteLegacyPass : public FunctionPass {
  // Pass identification, replacement for typeid
  static char ID;

  PromoteLegacyPass() : FunctionPass(ID) {
    initializePromoteLegacyPassPass(*PassRegistry::getPassRegistry());
  }

  // runOnFunction - To run this pass, first we calculate the alloca
  // instructions that are safe for promotion, then we promote each one.
  bool runOnFunction(Function &F) override {
    if (skipFunction(F))
      return false;

    DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
    AssumptionCache &AC =
        getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
    return promoteMemoryToRegister(F, DT, AC);
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<AssumptionCacheTracker>();
    AU.addRequired<DominatorTreeWrapperPass>();
    AU.setPreservesCFG();
  }
};
```

PromoteLegacyPass 是一个 FunctionPass，函数 runOnFunction 的内容很简单，就是对函数 promoteMemoryToRegister 的调用。

函数 promoteMemoryToRegister 的实现如下：

```
static bool promoteMemoryToRegister(Function &F, DominatorTree &DT,
                                    AssumptionCache &AC) {
  std::vector<AllocaInst *> Allocas;
  BasicBlock &BB = F.getEntryBlock(); // Get the entry node for the function
  bool Changed = false;

  while (true) {
    Allocas.clear();

    // Find allocas that are safe to promote, by looking at all instructions in
    // the entry node
    for (BasicBlock::iterator I = BB.begin(), E = --BB.end(); I != E; ++I)
      if (AllocaInst *AI = dyn_cast<AllocaInst>(I)) // Is it an alloca?
        if (isAllocaPromotable(AI))
          Allocas.push_back(AI);

    if (Allocas.empty())
      break;

    PromoteMemToReg(Allocas, DT, &AC);
    NumPromoted += Allocas.size();
    Changed = true;
  }
  return Changed;
}
```

该函数就是收集入口基本块中的所有 Promotable 的 AllocaInst，然后调用函数 PromoteMemToReg 。这里 LLVM 有一个假设：一个函数中所有的 AllocaInst 都是只出现在函数的入口基本块。所以编译器前端在生成 LLVM IR 时应该遵守该假设。

那么什么样的 AllocaInst 是 Promotable？简单来说如果该 AllocaInst 没有被用于 volatile instruction，并且它直接被用于 LoadInst 或 StoreInst（即没有被取过地址），那么就认为该 AllocaInst 是 Promotable。详细的可以看 isAllocaPromotable 函数的代码实现。

下面看函数 PromoteMemToReg 的实现：

```
void llvm::PromoteMemToReg(ArrayRef<AllocaInst *> Allocas, DominatorTree &DT,
                           AssumptionCache *AC) {
  // If there is nothing to do, bail out...
```

(continues on next page)

(continued from previous page)

```
  if (Allocas.empty())
    return;

  PromoteMem2Reg(Allocas, DT, AC).run();
}
```

如果没有 Promotable 的 AllocaInst，那么毫无疑问直接返回；否则构造一个结构体 `PromoteMem2Reg` 的对象，然后调用该对象的 `run` 函数，注意 `PromoteMem2Reg(Allocas, DT, AC).run()` 中的 `PromoteMem2Reg` 是一个结构体。

结构体 `PromoteMem2Reg` 的定义比较复杂，这里不给出完整的定义代码了，我们先看一下其构造函数：

```
PromoteMem2Reg(ArrayRef<AllocaInst *> Allocas, DominatorTree &DT,
               AssumptionCache *AC)
    : Allocas(Allocas.begin(), Allocas.end()), DT(DT),
      DIB(*DT.getRoot()->getParent()->getParent(), /*AllowUnresolved*/ false),
      AC(AC), SQ(DT.getRoot()->getParent()->getParent()->getDataLayout(),
                 nullptr, &DT, AC) {}
```

- 成员变量 `std::vector<AllocaInst *> Allocas`，用于保存正在被 promoted 的 AllocaInst，其初始化为所有的 Promotable 的 AllocaInst
- 成员变量 `DIBuilder DIB` Debug Information Builder 用于在 LLVM IR 中创建调试信息
- 成员变量 `AssumptionCache *AC`，对该成员变量的注释：A cache of @llvm.assume intrinsics used by SimplifyInstruction
- 成员变量 `const SimplifyQuery SQ`，SimplifyQuery 用于将 instructions 变为更简的形式，例如：*(“add i32 1, 1” -> “2”)*，*(“and i32 %x, 0” -> “0”)*，*(“and i32 %x, %x” -> “%x”)*
- etc，该结构体还有很多其他的成员变量

函数 `PromoteMem2Reg::run()` 是前面提到的 mem2reg 算法的真正代码实现，该函数共 200+ 行，这里不贴完整的代码了，而是一段一段的分析该函数。

```
void PromoteMem2Reg::run() {
  Function &F = *DT.getRoot()->getParent();
  ... // 略
  for (unsigned AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum) {
    AllocaInst *AI = Allocas[AllocaNum];

    assert(isAllocaPromotable(AI) && "Cannot promote non-promotable alloca!");
    assert(AI->getParent()->getParent() == &F &&
           "All allocas should be in the same function, which is same as DF!");
```

(continues on next page)

(continued from previous page)

```
    removeLifetimeIntrinsicUsers(AI);

    if (AI->use_empty()) {
      // If there are no uses of the alloca, just delete it now.
      AI->eraseFromParent();

      // Remove the alloca from the Allocas list, since it has been processed
      RemoveFromAllocasList(AllocaNum);
      ++NumDeadAlloca;
      continue;
    }
    ... // 略
  }
  ... // 略
}
```

F 是这些 AllocaInst 所在的函数，AllocaDbgDeclares 用于记录描述 AllocaInst 的 *dbg.declare intrinsic* ，在后续 AllocaInst 被 promoted 之后，就可以将 *dbg.declare intrinsic* 转换为 *dbg.value intrinsic* 。

for 循环依次处理每一个 AllocaInst ，在进行一些 assert 判断之后，调用函数 `removeLifetimeIntrinsicUsers` 在 LLVM IR 中删除该 AllocaInst 的 User 中除了 LoadInst 和 StoreInst 以外的 Instructions。如果某个 AllocaInst 没有 User，那么直接删除该 AllocaInst，并且将该 AllocaInst 从成员变量 `std::vector<AllocaInst *> Allocas` 中删除，因为该 AllocaInst 已经被处理完成。这段代码对应 mem2reg 算法的 4.1 部分。

紧接着是后续的这段代码对应 mem2reg 算法的 4.2 和 4.3 部分。

```
void PromoteMem2Reg::run() {
  AllocaDbgDeclares.resize(Allocas.size());
  AllocaInfo Info;
  ... // 略
  for (unsigned AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum) {
    AllocaInst *AI = Allocas[AllocaNum];

    ... // 略

    // Calculate the set of read and write-locations for each alloca.  This is
    // analogous to finding the 'uses' and 'definitions' of each variable.
    Info.AnalyzeAlloca(AI);

    // If there is only a single store to this value, replace any loads of
    // it that are directly dominated by the definition with the value stored.
    if (Info.DefiningBlocks.size() == 1) {
      if (rewriteSingleStoreAlloca(AI, Info, LBI, SQ.DL, DT, AC)) {
        // The alloca has been processed, move on.
```

(continues on next page)

(continued from previous page)

```
        RemoveFromAllocasList(AllocaNum);
        ++NumSingleStore;
        continue;
      }
    }

    // If the alloca is only read and written in one basic block, just perform a
    // linear sweep over the block to eliminate it.
    if (Info.OnlyUsedInOneBlock &&
        promoteSingleBlockAlloca(AI, Info, LBI, SQ.DL, DT, AC)) {
      // The alloca has been processed, move on.
      RemoveFromAllocasList(AllocaNum);
      continue;
    }
    ... // 略
  }
  ... // 略
}
```

调用 `AllocaInfo::AnalyzeAlloca(AllocaInst **AI*)`（详见 *Functionality:AllocaInfo*）分析该 AllocaInst 的相关信息。根据 `AllocaInfo::AnalyzeAlloca(AllocaInst **AI*)` 的分析结果，如果对该 AllocaInst 的定值只有一处，那么通过函数 `rewriteSingleStoreAlloca`（详见 *Functionality:rewriteSingleStoreAlloca*）将所有的被该定值点 (def) 所支配的使用点 (use) 都替换为相应的定值，即如果对该 AllocaInst 的 StoreInst 只有一条，那么将所有被该 StoreInst 支配的用于获取 AllocaInst 的值的 LoadInst 替换为被 store 的值；如果对该 AllocaInst 的 def 和 use 都在同一个基本块内，则调用函数 `promoteSingleBlockAlloca`（详见 *Functionality:promoteSingleBlockAlloca*）通过线性扫描来消除 AllocaInst / StoreInst/ LoadInst 。

在完成对上述一些特别的情况的处理之后，则通过 IDF (Iterated Dominance Frontier) 和标准的 SSA 构建算法来将 alloca/load/store 形式的 LLVM IR 提升为真正的 SSA 形式的 LLVM IR。

```
void PromoteMem2Reg::run() {
  AllocaDbgDeclares.resize(Allocas.size());
  AllocaInfo Info;
  ... // 略
  for (unsigned AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum) {
    AllocaInst *AI = Allocas[AllocaNum];

    ... // 略

    // If we haven't computed a numbering for the BB's in the function, do so
    // now.
```

(continues on next page)

(continued from previous page)

```
  if (BBNumbers.empty()) {
    unsigned ID = 0;
    for (auto &BB : F)
      BBNumbers[&BB] = ID++;
  }

  // Remember the dbg.declare intrinsic describing this alloca, if any.
  if (!Info.DbgDeclares.empty())
    AllocaDbgDeclares[AllocaNum] = Info.DbgDeclares;

  // Keep the reverse mapping of the 'Allocas' array for the rename pass.
  AllocaLookup[Allocas[AllocaNum]] = AllocaNum;

  // At this point, we're committed to promoting the alloca using IDF's, and
  // the standard SSA construction algorithm.  Determine which blocks need PHI
  // nodes and see if we can optimize out some work by avoiding insertion of
  // dead phi nodes.

  // Unique the set of defining blocks for efficient lookup.
  SmallPtrSet<BasicBlock *, 32> DefBlocks;
  DefBlocks.insert(Info.DefiningBlocks.begin(), Info.DefiningBlocks.end());

  // Determine which blocks the value is live in.  These are blocks which lead
  // to uses.
  SmallPtrSet<BasicBlock *, 32> LiveInBlocks;
  ComputeLiveInBlocks(AI, Info, DefBlocks, LiveInBlocks);

  // At this point, we're committed to promoting the alloca using IDF's, and
  // the standard SSA construction algorithm.  Determine which blocks need phi
  // nodes and see if we can optimize out some work by avoiding insertion of
  // dead phi nodes.
  IDF.setLiveInBlocks(LiveInBlocks);
  IDF.setDefiningBlocks(DefBlocks);
  SmallVector<BasicBlock *, 32> PHIBlocks;
  IDF.calculate(PHIBlocks);
  if (PHIBlocks.size() > 1)
    llvm::sort(PHIBlocks.begin(), PHIBlocks.end(),
               [this](BasicBlock *A, BasicBlock *B) {
                 return BBNumbers.lookup(A) < BBNumbers.lookup(B);
               });

  unsigned CurrentVersion = 0;
  for (BasicBlock *BB : PHIBlocks)
```

(continues on next page)

(continued from previous page)

```
      QueuePhiNode(BB, AllocaNum, CurrentVersion);

  }

  if (Allocas.empty())
    return; // All of the allocas must have been trivial!

  LBI.clear();

  // Set the incoming values for the basic block to be null values for all of
  // the alloca's.  We do this in case there is a load of a value that has not
  // been stored yet.  In this case, it will get this null value.
  RenamePassData::ValVector Values(Allocas.size());
  for (unsigned i = 0, e = Allocas.size(); i != e; ++i)
    Values[i] = UndefValue::get(Allocas[i]->getAllocatedType());

  // When handling debug info, treat all incoming values as if they have unknown
  // locations until proven otherwise.
  RenamePassData::LocationVector Locations(Allocas.size());

  // Walks all basic blocks in the function performing the SSA rename algorithm
  // and inserting the phi nodes we marked as necessary
  std::vector<RenamePassData> RenamePassWorkList;
  RenamePassWorkList.emplace_back(&F.front(), nullptr, std::move(Values),
                                  std::move(Locations));
  do {
    RenamePassData RPD = std::move(RenamePassWorkList.back());
    RenamePassWorkList.pop_back();
    // RenamePass may add new worklist entries.
    RenamePass(RPD.BB, RPD.Pred, RPD.Values, RPD.Locations, RenamePassWorkList);
  } while (!RenamePassWorkList.empty());

  // The renamer uses the Visited set to avoid infinite loops.  Clear it now.
  Visited.clear();

  ... // 略
}
```

上述代码对 mem2reg 算法的步骤 6 和步骤 7。

应用变量 `SmallPtrSet<BasicBlock *, 32> DefBlocks` 来存储所有的对某 AllocaInst 定值的基本块，用变量 `SmallPtrSet<BasicBlock *, 32> LiveInBlocks` 来存储函数 `ComputeLiveInBlocks`（详见 *Functionality:ComputeLiveInBlocks*）的返回结果。通过 IDF 计算出需要插入 Phi 的基本块集合 PHIBlocks，遍历 PHIBlocks 调用函数 `QueuePhiNode`，创建待更新的 PHINode 。

所有需要插入 PHINode 的位置都已经插入了待更新的 PHINode，然后 worklist 算法调用函数 `RenamePass`（详见 *Functionality:RenamePass*）对 PHINode 进行更新。RenamePassWorkList 被初始化为首先处理函数 F 的入口基本块，然后从入口基本块开始沿着 CFG 不断迭代处理。

最后就是一些收尾的工作。

```
// Remove the allocas themselves from the function.
for (Instruction *A : Allocas) {
  // If there are any uses of the alloca instructions left, they must be in
  // unreachable basic blocks that were not processed by walking the dominator
  // tree. Just delete the users now.
  if (!A->use_empty())
    A->replaceAllUsesWith(UndefValue::get(A->getType()));
  A->eraseFromParent();
}

// Remove alloca's dbg.declare instrinsics from the function.
for (auto &Declares : AllocaDbgDeclares)
  for (auto *DII : Declares)
    DII->eraseFromParent();
```

如果经过前面处理之后在 LLVM IR 中还有 AllocaInst，那么将所有使用该 AllocaInst 的地方替换为 UndefValue。然后将 AllocaInst 从 LLVM IR 中删除。在 LLVM IR 中删掉 AllocaInst 的 *dbg.declare instrinsics*。

```
// Loop over all of the PHI nodes and see if there are any that we can get
// rid of because they merge all of the same incoming values.  This can
// happen due to undef values coming into the PHI nodes.  This process is
// iterative, because eliminating one PHI node can cause others to be removed.
bool EliminatedAPHI = true;
while (EliminatedAPHI) {
  EliminatedAPHI = false;

  // Iterating over NewPhiNodes is deterministic, so it is safe to try to
  // simplify and RAUW them as we go.  If it was not, we could add uses to
  // the values we replace with in a non-deterministic order, thus creating
  // non-deterministic def->use chains.
  for (DenseMap<std::pair<unsigned, unsigned>, PHINode *>::iterator
            I = NewPhiNodes.begin(),
            E = NewPhiNodes.end();
       I != E;) {
    PHINode *PN = I->second;

    // If this PHI node merges one value and/or undefs, get the value.
    if (Value *V = SimplifyInstruction(PN, SQ)) {
```

(continues on next page)

(continued from previous page)

```
      PN->replaceAllUsesWith(V);
      PN->eraseFromParent();
      NewPhiNodes.erase(I++);
      EliminatedAPHI = true;
      continue;
    }
    ++I;
  }
}
```

接着对 PHINode 进行一些优化，如果 PHINode 的 IncomingValue 中有 UndefValue，那么通过函数 `SimplifyInstruction` 简化该 PHINode，并相应地将使用该 PHINode 的地方替换为简化后的 Value。

```
// At this point, the renamer has added entries to PHI nodes for all reachable
// code.  Unfortunately, there may be unreachable blocks which the renamer
// hasn't traversed.  If this is the case, the PHI nodes may not
// have incoming values for all predecessors.  Loop over all PHI nodes we have
// created, inserting undef values if they are missing any incoming values.
for (DenseMap<std::pair<unsigned, unsigned>, PHINode *>::iterator
         I = NewPhiNodes.begin(),
         E = NewPhiNodes.end();
     I != E; ++I) {
  // We want to do this once per basic block.  As such, only process a block
  // when we find the PHI that is the first entry in the block.
  PHINode *SomePHI = I->second;
  BasicBlock *BB = SomePHI->getParent();
  if (&BB->front() != SomePHI)
    continue;

  // Only do work here if there the PHI nodes are missing incoming values.  We
  // know that all PHI nodes that were inserted in a block will have the same
  // number of incoming values, so we can just check any of them.
  if (SomePHI->getNumIncomingValues() == getNumPreds(BB))
    continue;

  // Get the preds for BB.
  SmallVector<BasicBlock *, 16> Preds(pred_begin(BB), pred_end(BB));

  // Ok, now we know that all of the PHI nodes are missing entries for some
  // basic blocks.  Start by sorting the incoming predecessors for efficient
  // access.
  llvm::sort(Preds.begin(), Preds.end());

  // Now we loop through all BB's which have entries in SomePHI and remove
```

(continues on next page)

(continued from previous page)

```
  // them from the Preds list.
  for (unsigned i = 0, e = SomePHI->getNumIncomingValues(); i != e; ++i) {
    // Do a log(n) search of the Preds list for the entry we want.
    SmallVectorImpl<BasicBlock *>::iterator EntIt = std::lower_bound(
        Preds.begin(), Preds.end(), SomePHI->getIncomingBlock(i));
    assert(EntIt != Preds.end() && *EntIt == SomePHI->getIncomingBlock(i) &&
           "PHI node has entry for a block which is not a predecessor!");

    // Remove the entry
    Preds.erase(EntIt);
  }

  // At this point, the blocks left in the preds list must have dummy
  // entries inserted into every PHI nodes for the block.  Update all the phi
  // nodes in this block that we are inserting (there could be phis before
  // mem2reg runs).
  unsigned NumBadPreds = SomePHI->getNumIncomingValues();
  BasicBlock::iterator BBI = BB->begin();
  while ((SomePHI = dyn_cast<PHINode>(BBI++)) &&
         SomePHI->getNumIncomingValues() == NumBadPreds) {
    Value *UndefVal = UndefValue::get(SomePHI->getType());
    for (BasicBlock *Pred : Preds)
      SomePHI->addIncoming(UndefVal, Pred);
  }
}

NewPhiNodes.clear();
```

经过前面 RenamePass 的处理后基本上 PHINode 都已经更新好了，但是因为 RenamePass 是沿着 CFG 进行处理的，所有可能对于那些 unreachable blocks 即那些沿着 CFG 不可达的基本块，PHINode 的 incoming values 还不完整，对于这种情况，将这些 incoming values 设置为 UndefValue。

3.2.3 Functionality

为了让 LLVM 的 mem2reg 算法实现更加易读，这里将函数 `PromoteMem2Reg::run()` 中用到的一些数据结构和函数单独进行分析。

AllocaInfo

```
struct AllocaInfo {
  SmallVector<BasicBlock *, 32> DefiningBlocks;
  SmallVector<BasicBlock *, 32> UsingBlocks;

  StoreInst *OnlyStore;
  BasicBlock *OnlyBlock;
  bool OnlyUsedInOneBlock;

  Value *AllocaPointerVal;
  TinyPtrVector<DbgInfoIntrinsic *> DbgDeclares;

  void clear() {
    DefiningBlocks.clear();
    UsingBlocks.clear();
    OnlyStore = nullptr;
    OnlyBlock = nullptr;
    OnlyUsedInOneBlock = true;
    AllocaPointerVal = nullptr;
    DbgDeclares.clear();
  }

  /// Scan the uses of the specified alloca, filling in the AllocaInfo used
  /// by the rest of the pass to reason about the uses of this alloca.
  void AnalyzeAlloca(AllocaInst *AI) {
    clear();

    // As we scan the uses of the alloca instruction, keep track of stores,
    // and decide whether all of the loads and stores to the alloca are within
    // the same basic block.
    for (auto UI = AI->user_begin(), E = AI->user_end(); UI != E;) {
      Instruction *User = cast<Instruction>(*UI++);

      if (StoreInst *SI = dyn_cast<StoreInst>(User)) {
        // Remember the basic blocks which define new values for the alloca
        DefiningBlocks.push_back(SI->getParent());
        AllocaPointerVal = SI->getOperand(0);
        OnlyStore = SI;
      } else {
        LoadInst *LI = cast<LoadInst>(User);
        // Otherwise it must be a load instruction, keep track of variable
        // reads.
        UsingBlocks.push_back(LI->getParent());
```

(continues on next page)

(continued from previous page)

```
        AllocaPointerVal = LI;
      }

      if (OnlyUsedInOneBlock) {
        if (!OnlyBlock)
          OnlyBlock = User->getParent();
        else if (OnlyBlock != User->getParent())
          OnlyUsedInOneBlock = false;
      }
    }

    DbgDeclares = FindDbgAddrUses(AI);
  }
};
```

`AllocaInfo::AnalyzeAlloca(AllocaInst **AI*)` 函数，记录了给定的一条 AllocaInst 的相关信息，AllocaInfo 的成员变量 DefiningBlocks 记录了所有对 AllocaInst 进行定值 (def) 的基本块；成员变量 UsingBlocks 记录了所有对 AllocaInst 进行使用 (use) 的基本块；成员变量 OnlyUsedInOneBlock 记录了是否所有对该条 AllocaInst 的 def 和 use 都在同一个基本块中，如果是，则将该基本块记录在成员变量 OnlyBlock 中；如果对 AllocaInst 的定值 (def) 即 StoreInst 只有一条，那么该 StoreInst 则存储在成员变量 OnlyStore 中。

LargeBlockInfo

```
/// This assigns and keeps a per-bb relative ordering of load/store
/// instructions in the block that directly load or store an alloca.
///
/// This functionality is important because it avoids scanning large basic
/// blocks multiple times when promoting many allocas in the same block.
class LargeBlockInfo {
  /// For each instruction that we track, keep the index of the
  /// instruction.
  ///
  /// The index starts out as the number of the instruction from the start of
  /// the block.
  DenseMap<const Instruction *, unsigned> InstNumbers;

public:
  /// This code only looks at accesses to allocas.
  static bool isInterestingInstruction(const Instruction *I) {
    return (isa<LoadInst>(I) && isa<AllocaInst>(I->getOperand(0))) ||
           (isa<StoreInst>(I) && isa<AllocaInst>(I->getOperand(1)));
  }
```

(continues on next page)

(continued from previous page)

```
  /// Get or calculate the index of the specified instruction.
  unsigned getInstructionIndex(const Instruction *I) {
    assert(isInterestingInstruction(I) &&
           "Not a load/store to/from an alloca?");

    // If we already have this instruction number, return it.
    DenseMap<const Instruction *, unsigned>::iterator It = InstNumbers.find(I);
    if (It != InstNumbers.end())
      return It->second;

    // Scan the whole block to get the instruction.  This accumulates
    // information for every interesting instruction in the block, in order to
    // avoid gratuitus rescans.
    const BasicBlock *BB = I->getParent();
    unsigned InstNo = 0;
    for (const Instruction &BBI : *BB)
      if (isInterestingInstruction(&BBI))
        InstNumbers[&BBI] = InstNo++;
    It = InstNumbers.find(I);

    assert(It != InstNumbers.end() && "Didn't insert instruction?");
    return It->second;
  }

  void deleteValue(const Instruction *I) { InstNumbers.erase(I); }

  void clear() { InstNumbers.clear(); }
};
```

LargeBlockInfo 用于记录和获取同一基本块中出现的 LoadInst 和 StoreInst 先后顺序。

rewriteSingleStoreAlloca

```
/// Rewrite as many loads as possible given a single store.
///
/// When there is only a single store, we can use the domtree to trivially
/// replace all of the dominated loads with the stored value. Do so, and return
/// true if this has successfully promoted the alloca entirely. If this returns
/// false there were some loads which were not dominated by the single store
/// and thus must be phi-ed with undef. We fall back to the standard alloca
/// promotion algorithm in that case.
static bool rewriteSingleStoreAlloca(AllocaInst *AI, AllocaInfo &Info,
```

(continues on next page)

(continued from previous page)

```
                                    LargeBlockInfo &LBI, const DataLayout &DL,
                                    DominatorTree &DT, AssumptionCache *AC) {
StoreInst *OnlyStore = Info.OnlyStore;
bool StoringGlobalVal = !isa<Instruction>(OnlyStore->getOperand(0));
BasicBlock *StoreBB = OnlyStore->getParent();
int StoreIndex = -1;

// Clear out UsingBlocks.  We will reconstruct it here if needed.
Info.UsingBlocks.clear();

for (auto UI = AI->user_begin(), E = AI->user_end(); UI != E;) {
  Instruction *UserInst = cast<Instruction>(*UI++);
  if (!isa<LoadInst>(UserInst)) {
    assert(UserInst == OnlyStore && "Should only have load/stores");
    continue;
  }
  LoadInst *LI = cast<LoadInst>(UserInst);

  // Okay, if we have a load from the alloca, we want to replace it with the
  // only value stored to the alloca.  We can do this if the value is
  // dominated by the store.  If not, we use the rest of the mem2reg machinery
  // to insert the phi nodes as needed.
  if (!StoringGlobalVal) { // Non-instructions are always dominated.
    if (LI->getParent() == StoreBB) {
      // If we have a use that is in the same block as the store, compare the
      // indices of the two instructions to see which one came first.  If the
      // load came before the store, we can't handle it.
      if (StoreIndex == -1)
        StoreIndex = LBI.getInstructionIndex(OnlyStore);

      if (unsigned(StoreIndex) > LBI.getInstructionIndex(LI)) {
        // Can't handle this load, bail out.
        Info.UsingBlocks.push_back(StoreBB);
        continue;
      }
    } else if (LI->getParent() != StoreBB &&
               !DT.dominates(StoreBB, LI->getParent())) {
      // If the load and store are in different blocks, use BB dominance to
      // check their relationships.  If the store doesn't dom the use, bail
      // out.
      Info.UsingBlocks.push_back(LI->getParent());
      continue;
    }
```

(continues on next page)

(continued from previous page)

```
    }

    // Otherwise, we *can* safely rewrite this load.
    Value *ReplVal = OnlyStore->getOperand(0);
    // If the replacement value is the load, this must occur in unreachable
    // code.
    if (ReplVal == LI)
      ReplVal = UndefValue::get(LI->getType());

    // If the load was marked as nonnull we don't want to lose
    // that information when we erase this Load. So we preserve
    // it with an assume.
    if (AC && LI->getMetadata(LLVMContext::MD_nonnull) &&
        !isKnownNonZero(ReplVal, DL, 0, AC, LI, &DT))
      addAssumeNonNull(AC, LI);

    LI->replaceAllUsesWith(ReplVal);
    LI->eraseFromParent();
    LBI.deleteValue(LI);
  }

  // Finally, after the scan, check to see if the store is all that is left.
  if (!Info.UsingBlocks.empty())
    return false; // If not, we'll have to fall back for the remainder.

  // Record debuginfo for the store and remove the declaration's
  // debuginfo.
  for (DbgInfoIntrinsic *DII : Info.DbgDeclares) {
    DIBuilder DIB(*AI->getModule(), /*AllowUnresolved*/ false);
    ConvertDebugDeclareToDebugValue(DII, Info.OnlyStore, DIB);
    DII->eraseFromParent();
    LBI.deleteValue(DII);
  }
  // Remove the (now dead) store and alloca.
  Info.OnlyStore->eraseFromParent();
  LBI.deleteValue(Info.OnlyStore);

  AI->eraseFromParent();
  LBI.deleteValue(AI);
  return true;
}
```

`rewriteSingleStoreAlloca` 该函数的注释写的很清晰。在对于某条 AllocaInst 只有一处定值点 (def) 的情况下会调用该函数，该定值点即为 OnlyStore。因为只有一处定值点，所以可以将所有的被定值点**支配**的使

用点 (use) 即 LoadInst 都替换该被定值点定义的值。但是这里有几个特殊情况需要处理：如果 def 和 use 在同一基本块内，那么需要保证 def 在 use 之前，这里就是通过 `LargeBlockInfo::getInstructionIndex()` 来计算 def 和 use 的先后顺序的，如果 use 在 def 之前，那么则该 use 则不能被定值所替换；如果 use 没有被 def 支配，当然该 use 也不能被定值所替换掉。

promoteSingleBlockAlloca

```
/// Many allocas are only used within a single basic block.  If this is the
/// case, avoid traversing the CFG and inserting a lot of potentially useless
/// PHI nodes by just performing a single linear pass over the basic block
/// using the Alloca.
///
/// If we cannot promote this alloca (because it is read before it is written),
/// return false.  This is necessary in cases where, due to control flow, the
/// alloca is undefined only on some control flow paths.  e.g. code like
/// this is correct in LLVM IR:
///  // A is an alloca with no stores so far
///  for (...) {
///    int t = *A;
///    if (!first_iteration)
///      use(t);
///    *A = 42;
///  }
static bool promoteSingleBlockAlloca(AllocaInst *AI, const AllocaInfo &Info,
                                     LargeBlockInfo &LBI, const DataLayout &DL,
                                     DominatorTree &DT, AssumptionCache *AC) {
  // The trickiest case to handle is when we have large blocks. Because of this,
  // this code is optimized assuming that large blocks happen.  This does not
  // significantly pessimize the small block case.  This uses LargeBlockInfo to
  // make it efficient to get the index of various operations in the block.

  // Walk the use-def list of the alloca, getting the locations of all stores.
  using StoresByIndexTy = SmallVector<std::pair<unsigned, StoreInst *>, 64>;
  StoresByIndexTy StoresByIndex;

  for (User *U : AI->users())
    if (StoreInst *SI = dyn_cast<StoreInst>(U))
      StoresByIndex.push_back(std::make_pair(LBI.getInstructionIndex(SI), SI));

  // Sort the stores by their index, making it efficient to do a lookup with a
  // binary search.
  llvm::sort(StoresByIndex.begin(), StoresByIndex.end(), less_first());

  // Walk all of the loads from this alloca, replacing them with the nearest
```

(continues on next page)

(continued from previous page)

```
  // store above them, if any.
  for (auto UI = AI->user_begin(), E = AI->user_end(); UI != E;) {
    LoadInst *LI = dyn_cast<LoadInst>(*UI++);
    if (!LI)
      continue;

    unsigned LoadIdx = LBI.getInstructionIndex(LI);

    // Find the nearest store that has a lower index than this load.
    StoresByIndexTy::iterator I = std::lower_bound(
        StoresByIndex.begin(), StoresByIndex.end(),
        std::make_pair(LoadIdx, static_cast<StoreInst *>(nullptr)),
        less_first());
    if (I == StoresByIndex.begin()) {
      if (StoresByIndex.empty())
        // If there are no stores, the load takes the undef value.
        LI->replaceAllUsesWith(UndefValue::get(LI->getType()));
      else
        // There is no store before this load, bail out (load may be affected
        // by the following stores - see main comment).
        return false;
    } else {
      // Otherwise, there was a store before this load, the load takes its
      // value. Note, if the load was marked as nonnull we don't want to lose
      // that information when we erase it. So we preserve it with an assume.
      Value *ReplVal = std::prev(I)->second->getOperand(0);
      if (AC && LI->getMetadata(LLVMContext::MD_nonnull) &&
          !isKnownNonZero(ReplVal, DL, 0, AC, LI, &DT))
        addAssumeNonNull(AC, LI);

      // If the replacement value is the load, this must occur in unreachable
      // code.
      if (ReplVal == LI)
        ReplVal = UndefValue::get(LI->getType());

      LI->replaceAllUsesWith(ReplVal);
    }

    LI->eraseFromParent();
    LBI.deleteValue(LI);
  }

  // Remove the (now dead) stores and alloca.
```

(continues on next page)

(continued from previous page)

```
  while (!AI->use_empty()) {
    StoreInst *SI = cast<StoreInst>(AI->user_back());
    // Record debuginfo for the store before removing it.
    for (DbgInfoIntrinsic *DII : Info.DbgDeclares) {
      DIBuilder DIB(*AI->getModule(), /*AllowUnresolved*/ false);
      ConvertDebugDeclareToDebugValue(DII, SI, DIB);
    }
    SI->eraseFromParent();
    LBI.deleteValue(SI);
  }

  AI->eraseFromParent();
  LBI.deleteValue(AI);

  // The alloca's debuginfo can be removed as well.
  for (DbgInfoIntrinsic *DII : Info.DbgDeclares) {
    DII->eraseFromParent();
    LBI.deleteValue(DII);
  }

  ++NumLocalPromoted;
  return true;
}
```

函数 `promoteSingleBlockAlloca` 用于处理 AllocaInst 的 LoadInst 和 StoreInst 只出现在同一基本块中的情况。对于每个 LoadInst 指令寻找在其之前出现的、相邻最近的 StoreInst，将所有通过 LoadInst 获取的值都替换为对应的 Stored value。存在几种特殊情况：如果对于某个 AllocaInst 来说，相应的 StoreInst 集合为空，那么将所有的通过 LoadInst 获取的值都替换为 UndefValue；如果对于某条 LoadInst 来说，没有在其之前出现的、相邻最近的 StoreInst，那么函数 `promoteSingleBlockAlloca` 返回 false，后续通过标准 SSA 构建算法来处理。

ComputeLiveInBlocks

```
/// Determine which blocks the value is live in.
///
/// These are blocks which lead to uses.  Knowing this allows us to avoid
/// inserting PHI nodes into blocks which don't lead to uses (thus, the
/// inserted phi nodes would be dead).
void PromoteMem2Reg::ComputeLiveInBlocks(
    AllocaInst *AI, AllocaInfo &Info,
    const SmallPtrSetImpl<BasicBlock *> &DefBlocks,
    SmallPtrSetImpl<BasicBlock *> &LiveInBlocks) {
```

(continues on next page)

(continued from previous page)

```
  // To determine liveness, we must iterate through the predecessors of blocks
  // where the def is live.  Blocks are added to the worklist if we need to
  // check their predecessors.  Start with all the using blocks.
  SmallVector<BasicBlock *, 64> LiveInBlockWorklist(Info.UsingBlocks.begin(),
                                                    Info.UsingBlocks.end());

  // If any of the using blocks is also a definition block, check to see if the
  // definition occurs before or after the use.  If it happens before the use,
  // the value isn't really live-in.
  for (unsigned i = 0, e = LiveInBlockWorklist.size(); i != e; ++i) {
    BasicBlock *BB = LiveInBlockWorklist[i];
    if (!DefBlocks.count(BB))
      continue;

    // Okay, this is a block that both uses and defines the value.  If the first
    // reference to the alloca is a def (store), then we know it isn't live-in.
    for (BasicBlock::iterator I = BB->begin();; ++I) {
      if (StoreInst *SI = dyn_cast<StoreInst>(I)) {
        if (SI->getOperand(1) != AI)
          continue;

        // We found a store to the alloca before a load.  The alloca is not
        // actually live-in here.
        LiveInBlockWorklist[i] = LiveInBlockWorklist.back();
        LiveInBlockWorklist.pop_back();
        --i;
        --e;
        break;
      }

      if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
        if (LI->getOperand(0) != AI)
          continue;

        // Okay, we found a load before a store to the alloca.  It is actually
        // live into this block.
        break;
      }
    }
  }

  // Now that we have a set of blocks where the phi is live-in, recursively add
  // their predecessors until we find the full region the value is live.
```

(continues on next page)

(continued from previous page)

```
  while (!LiveInBlockWorklist.empty()) {
    BasicBlock *BB = LiveInBlockWorklist.pop_back_val();

    // The block really is live in here, insert it into the set.  If already in
    // the set, then it has already been processed.
    if (!LiveInBlocks.insert(BB).second)
      continue;

    // Since the value is live into BB, it is either defined in a predecessor or
    // live into it to.  Add the preds to the worklist unless they are a
    // defining block.
    for (BasicBlock *P : predecessors(BB)) {
      // The value is not live into a predecessor if it defines the value.
      if (DefBlocks.count(P))
        continue;

      // Otherwise it is, add to the worklist.
      LiveInBlockWorklist.push_back(P);
    }
  }
}
```

函数 `PromoteMem2Reg::ComputeLiveInBlocks` 注释很清晰。LiveInBlockWorklist 被初始化为所有对 AllocaInst 进行使用 (use) 的基本块，如果 LiveInBlockWorklist 存在对 AllocaInst 进行定义 (def) 的基本块，并且在该基本块中对该 AllocaInst 的第一条 StoreInst 出现在对该 AllocaInst 的第一条 LoadInst 之前，那么在 LiveInBlockWorklist 中去掉该基本块。然后以此时的 LiveInBlockWorklist 作为初始集合进行 worklist 算法迭代：对于 LiveInBlockWorklist 中的每个元素，如果其不在 LiveInBlocks 中，则将其添加至 LiveInBlocks，如果其前驱基本块不是对 AllocaInst 进行定义 (def) 的基本块，则将此前驱基本块也添加至 LiveInBlockWorklist，一直迭代至 LiveInBlockWorklist 为空。

QueuePhiNode

```
/// Queue a phi-node to be added to a basic-block for a specific Alloca.
///
/// Returns true if there wasn't already a phi-node for that variable
bool PromoteMem2Reg::QueuePhiNode(BasicBlock *BB, unsigned AllocaNo,
                                  unsigned &Version) {
  // Look up the basic-block in question.
  PHINode *&PN = NewPhiNodes[std::make_pair(BBNumbers[BB], AllocaNo)];

  // If the BB already has a phi node added for the i'th alloca then we're done!
  if (PN)
```

(continues on next page)

(continued from previous page)

```
    return false;

  // Create a PhiNode using the dereferenced type... and add the phi-node to the
  // BasicBlock.
  PN = PHINode::Create(Allocas[AllocaNo]->getAllocatedType(), getNumPreds(BB),
                       Allocas[AllocaNo]->getName() + "." + Twine(Version++),
                       &BB->front());
  ++NumPHIInsert;
  PhiToAllocaMap[PN] = AllocaNo;
  return true;
}
```

函数 `QueuePhiNode` 用于在基本块 BB 入口处为第 AllocaNo 条 AllocaInst 创建一个待更新的 PHINode。待更新指的是这里 PHINode 指令的操作数还不完全，需要后续对操作数进行更新（在函数 `PromoteMem2Reg::RenamePass` 中对此处插入的 PHINode 进行更新）。

RenamePass

```
/// Recursively traverse the CFG of the function, renaming loads and
/// stores to the allocas which we are promoting.
///
/// IncomingVals indicates what value each Alloca contains on exit from the
/// predecessor block Pred.
void PromoteMem2Reg::RenamePass(BasicBlock *BB, BasicBlock *Pred,
                                RenamePassData::ValVector &IncomingVals,
                                RenamePassData::LocationVector &IncomingLocs,
                                std::vector<RenamePassData> &Worklist) {
NextIteration:
  // If we are inserting any phi nodes into this BB, they will already be in the
  // block.
  if (PHINode *APN = dyn_cast<PHINode>(BB->begin())) {
    // If we have PHI nodes to update, compute the number of edges from Pred to
    // BB.
    if (PhiToAllocaMap.count(APN)) {
      // We want to be able to distinguish between PHI nodes being inserted by
      // this invocation of mem2reg from those phi nodes that already existed in
      // the IR before mem2reg was run.  We determine that APN is being inserted
      // because it is missing incoming edges.  All other PHI nodes being
      // inserted by this pass of mem2reg will have the same number of incoming
      // operands so far.  Remember this count.
      unsigned NewPHINumOperands = APN->getNumOperands();

      unsigned NumEdges = std::count(succ_begin(Pred), succ_end(Pred), BB);
```

(continues on next page)

(continued from previous page)

```
    assert(NumEdges && "Must be at least one edge from Pred to BB!");

    // Add entries for all the phis.
    BasicBlock::iterator PNI = BB->begin();
    do {
      unsigned AllocaNo = PhiToAllocaMap[APN];

      // Update the location of the phi node.
      updateForIncomingValueLocation(APN, IncomingLocs[AllocaNo],
                                     APN->getNumIncomingValues() > 0);

      // Add N incoming values to the PHI node.
      for (unsigned i = 0; i != NumEdges; ++i)
        APN->addIncoming(IncomingVals[AllocaNo], Pred);

      // The currently active variable for this block is now the PHI.
      IncomingVals[AllocaNo] = APN;
      for (DbgInfoIntrinsic *DII : AllocaDbgDeclares[AllocaNo])
        ConvertDebugDeclareToDebugValue(DII, APN, DIB);

      // Get the next phi node.
      ++PNI;
      APN = dyn_cast<PHINode>(PNI);
      if (!APN)
        break;

      // Verify that it is missing entries.  If not, it is not being inserted
      // by this mem2reg invocation so we want to ignore it.
    } while (APN->getNumOperands() == NewPHINumOperands);
  }
}

// Don't revisit blocks.
if (!Visited.insert(BB).second)
  return;

for (BasicBlock::iterator II = BB->begin(); !isa<TerminatorInst>(II);) {
  Instruction *I = &*II++; // get the instruction, increment iterator

  if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
    AllocaInst *Src = dyn_cast<AllocaInst>(LI->getPointerOperand());
    if (!Src)
      continue;
```

(continues on next page)

(continued from previous page)

```
    DenseMap<AllocaInst *, unsigned>::iterator AI = AllocaLookup.find(Src);
    if (AI == AllocaLookup.end())
      continue;

    Value *V = IncomingVals[AI->second];

    // If the load was marked as nonnull we don't want to lose
    // that information when we erase this Load. So we preserve
    // it with an assume.
    if (AC && LI->getMetadata(LLVMContext::MD_nonnull) &&
        !isKnownNonZero(V, SQ.DL, 0, AC, LI, &DT))
      addAssumeNonNull(AC, LI);

    // Anything using the load now uses the current value.
    LI->replaceAllUsesWith(V);
    BB->getInstList().erase(LI);
  } else if (StoreInst *SI = dyn_cast<StoreInst>(I)) {
    // Delete this instruction and mark the name as the current holder of the
    // value
    AllocaInst *Dest = dyn_cast<AllocaInst>(SI->getPointerOperand());
    if (!Dest)
      continue;

    DenseMap<AllocaInst *, unsigned>::iterator ai = AllocaLookup.find(Dest);
    if (ai == AllocaLookup.end())
      continue;

    // what value were we writing?
    unsigned AllocaNo = ai->second;
    IncomingVals[AllocaNo] = SI->getOperand(0);

    // Record debuginfo for the store before removing it.
    IncomingLocs[AllocaNo] = SI->getDebugLoc();
    for (DbgInfoIntrinsic *DII : AllocaDbgDeclares[ai->second])
      ConvertDebugDeclareToDebugValue(DII, SI, DIB);
    BB->getInstList().erase(SI);
  }
}

// 'Recurse' to our successors.
succ_iterator I = succ_begin(BB), E = succ_end(BB);
if (I == E)
```

(continues on next page)

(continued from previous page)

```
    return;

  // Keep track of the successors so we don't visit the same successor twice
  SmallPtrSet<BasicBlock *, 8> VisitedSuccs;

  // Handle the first successor without using the worklist.
  VisitedSuccs.insert(*I);
  Pred = BB;
  BB = *I;
  ++I;

  for (; I != E; ++I)
    if (VisitedSuccs.insert(*I).second)
      Worklist.emplace_back(*I, Pred, IncomingVals, IncomingLocs);

  goto NextIteration;
}
```

该函数的代码主要对应 mem2reg 算法中的步骤 7，不再赘述。

While (worklist != NULL)

1. Remove block B from worklist and mark B as visited.
2. For each instruction in B:
    1. If instruction is a load instruction from location L (where L is a promotable candidate) to value V, delete load instruction, replace all uses of V with most recent value of L i.e, IncomingVals[L].
    2. If instruction is a store instruction to location L (where L is a promotable candidate) with value V, delete store instruction, set most recent name of L i.e, IncomingVals[L] = V.
    3. For each PHI-node corresponding to a alloca —L , in each successor of B, fill the corresponding PHI-node argument with most recent name for that location i.e, IncomingVals[L].
3. Add each unvisited successor of B to worklist.

3.2.4 参考链接

https://www.zhihu.com/question/41999500/answer/93243408

- genindex
- modindex
- search

CHAPTER

FOUR

ANALYSIS

4.1 Alias Analysis

4.1.1 Alias Analysis Basic

Introduction

别名分析（又称指针分析）是程序分析中的一类技术，试图确定两个指针是否指向内存中的同一个对象。别名分析有很多中算法，也有很多种分类的方法：flow-sensitive(流敏感) vs. flow-insensitive(流不敏感), context-sensitive(上下文敏感) vs. context-insensitive(上下文不敏感), field-sensitive (域敏感) vs. field-insensitive(域不敏感), unification-based vs. subset-based, etc.

LLVM 中实现了很多别名分析算法：

- basicaa - Basic Alias Analysis (stateless AA impl)
- cfl-anders-aa - Inclusion-Based CFL Alias Analysis
- cfl-steens-aa - Unification-Based CFL Alias Analysis
- external-aa - External Alias Analysis
- globals-aa - Globals Alias Analysis
- scev-aa - ScalarEvolution-based Alias Analysis
- scoped-noalias - Scoped NoAlias Alias Analysis
- tbaa - Type-Based Alias Analysis
- ……

可以使用如下命令 `$ opt -cfl-anders-aa -aa-eval foo.bc -disable-output -stats` 来评估 LLVM 中已经实现的别名分析算法的分析效果。命令行参数 `cfl-anders-aa` 表示调用 `Inclusion-Based CFL Alias Analysis` 算法，命令行参数 `aa-eval` 调用的 `aa-eval` pass 用于输出一些统计信息，该 pass 的实现文件位于 `llvm-5.0.1.src/include/llvm/Analysis/AliasAnalysisEvaluator.h` 和 `llvm-5.0.1.src/lib/Analysis/AliasAnalysisEvaluator.cpp`,这些统计信息反映了别名分析的精确度（别名分析越精确，输出的统计信息中 may alias 所占的比例越小)。

上述命令的一个输出示例如下：

```
  ~ opt -cfl-anders-aa -aa-eval ./test.bc -disable-output -stats
===== Alias Analysis Evaluator Report =====
  210 Total Alias Queries Performed
  161 no alias responses (76.6%)
  31 may alias responses (14.7%)
  0 partial alias responses (0.0%)
  18 must alias responses (8.5%)
  Alias Analysis Evaluator Pointer Alias Summary: 76%/14%/0%/8%
  69 Total ModRef Queries Performed
  25 no mod/ref responses (36.2%)
  0 mod responses (0.0%)
  0 ref responses (0.0%)
  44 mod & ref responses (63.7%)
  Alias Analysis Evaluator Mod/Ref Summary: 36%/0%/0%/63%
```

Must, May, Partial and No Alias Responses

正如上述命令的输出示例所示，在 LLVM 中别名分析的结果有四种：

- NoAlias，两个指针之间没有直接依赖的关系时就是 NoAlias 。比如：两个指针指向非重叠的内存区域；两个指针只被用于读取内存（？）；有一段内存空间，存在一个指针用于访问该段内存，该段内存空间被 free（释放），然后被 realloc（重新分配），另外一个指针用于访问该段内存空间，这两个指针之间为 NoAlias。
- MayAlias，两个指针可能指向同一个对象，MayAlias 是最不精确（保守）的分析结果
- PartialAlias，两个内存对象以某种方式存在重叠的部分，注意：不管两个内存对象起始地址是否相同，只要有重叠的部分，它们之间就是 PartialAlias
- MustAlias，两个内存对象互为别名

AliasAttrs

类 AliasAttrs 用来描述一个指针是否具有的某些对别名分析有用的特殊属性，包括：

- AttrNone, represent whether the said pointer comes from an unknown source
- AttrUnknown, represent whether the said pointer comes from a source not known to alias analyses
- AttrCaller, represent whether the said pointer comes from a source not known to the current function but known to the caller. Values pointed to by the arguments of the current function have this attribute set
- AttrEscaped, represent whether the said pointer comes from a known source but escapes to the unknown world (e.g. casted to an integer, or passed as an argument to opaque function). Unlike non-escaped pointers, escaped ones may alias pointers coming from unknown sources

- AttrGlobal, represent whether the said pointer is a global value
- AttrArg, represent whether the said pointer is an argument, and if so, what index the argument has
- ExternallyVisibleAttrs, return a new AliasAttrs that only contains attributes meaningful to the caller. This function is primarily used for interprocedural analysis Currently, externally visible AliasAttrs include AttrUnknown, AttrGlobal, and AttrEscaped

Alias Analysis Precision Evaluator (aa-eval)

aa-eval 的实现是 AAEvalLegacyPass 类，其实现代码如下：

```
class AAEvalLegacyPass : public FunctionPass {
  std::unique_ptr<AAEvaluator> P;

public:
  static char ID; // Pass identification, replacement for typeid
  AAEvalLegacyPass() : FunctionPass(ID) {
    initializeAAEvalLegacyPassPass(*PassRegistry::getPassRegistry());
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<AAResultsWrapperPass>();
    AU.setPreservesAll();
  }

  bool doInitialization(Module &M) override {
    P.reset(new AAEvaluator());
    return false;
  }

  bool runOnFunction(Function &F) override {
    P->runInternal(F, getAnalysis<AAResultsWrapperPass>().getAAResults());
    return false;
  }
  bool doFinalization(Module &M) override {
    P.reset();
    return false;
  }
};
```

由 getAnalysisUsage 函数体的内容，可知 AAEvalLegacyPass 依赖于 AAResultsWrapperPass 的执行。而 AAEvalLegacyPass 在 doInitialization 中创建了一个 AAEvaluator 的对象，然后在 runOnFunction 函数中调用了 AAEvaluator 的 runInternal 函数，最后在 doFinalization 将指向之前创建的 AAEvaluator 对象的删除，可见 aa-eval 功能的实现是通过 AAEvaluator 这个类。

根据 AAResultsWrapperPass 这个 pass 的命名，猜测该 pass 用于收集 AliasAnalysis 的结果。找到 AAResultsWrapperPass 的实现：

```
/// A wrapper pass to provide the legacy pass manager access to a suitably
/// prepared AAResults object.
class AAResultsWrapperPass : public FunctionPass {
  std::unique_ptr<AAResults> AAR;

public:
  static char ID;

  AAResultsWrapperPass();

  AAResults &getAAResults() { return *AAR; }
  const AAResults &getAAResults() const { return *AAR; }

  bool runOnFunction(Function &F) override;

  void getAnalysisUsage(AnalysisUsage &AU) const override;
};
```

注释的内容印证了我们的猜测，AAResultsWrapperPass 就是提供一个接口供 pass manager 来访问别名分析的结果。

如果看一下 AAResultsWrapperPass::runOnFunction 的实现，会看到很多类似于如下片段的代码。

```
if (auto *WrapperPass = getAnalysisIfAvailable<CFLAndersAAWrapperPass>())
    AAR->addAAResult(WrapperPass->getResult());
```

将各种别名分析算法（如果该别名分析算法 Available，即被指定执行）的结果加入到 AAResults 中。

我们回到 AAEvalLegacyPass::runOnFunction 的实现：

```
bool runOnFunction(Function &F) override {
    P->runInternal(F, getAnalysis<AAResultsWrapperPass>().getAAResults());
    return false;
}
```

getAnalysis<AAResultsWrapperPass>().getAAResults() 的返回结果就是 AAResultsWrapperPass 的成员变量 AAR 所指向 AAResults。接下来，我们步入到 void AAEvaluator::runInternal(Function &F, AAResults &AA) 函数会看到，在该函数中是通过调用 AAResults::alias 来获取别名信息的。查看 AAResults::alias 的实现代码：

```
AliasResult AAResults::alias(const MemoryLocation &LocA,
                             const MemoryLocation &LocB) {
  for (const auto &AA : AAs) {
```

(continues on next page)

(continued from previous page)

```
    auto Result = AA->alias(LocA, LocB);
    if (Result != MayAlias)
      return Result;
  }
  return MayAlias;
}
```

前面提到，AAResults 中会保存多种别名分析算法（如果该别名分析算法 Available）的结果，对于两个 `MemoryLocation` 来讲其别名关系就是前面提到的四种：NoAlias, MayAlias, PartialAlias, MustAlias；其中 MayAlias 是最不精确的结果，在 `AAResults::alias` 的实现中，在遍历不同的别名分析算法的结果给出的两个 `MemoryLocation` 之间的别名关系时，如果别名关系不是 MayAlias ，就返回该别名分析结果。`AAResults::alias` 这样的实现方式，可以在指定别名分析算法时，组合多种别名分析算法，获取精更确的分析结果。

对于不同的别名分析算法在实现时，都要定义一个继承自 `AAResultBase` 的 Result 类，并重写 `AliasResult alias(const MemoryLocation &, const MemoryLocation &);` 函数。

4.1.2 CFL Anderson Alias Analysis

在 LLVM5.0.1 中实现了很多种 Alias Analysis，其中包括了 CFL-Anderson-AliasAnalysis，该 Alias Analysis 用到的算法基于“Demand-driven alias analysis for C”(http://www.cs.cornell.edu/~xinz/papers/alias-popl08.pdf) 这篇论文，并做了一些适配修改。

Demand-Driven Alias Analysis for C

该论文提出了一种 a demand-driven, flow-insensitive analysis algorithm for answering may-alias queries.

Program Representation

在该论文中，将程序形式化表示为一种 C-like program：

1. 程序中所有的值都是指针
2. 程序中包含指针赋值语句
3. 由于该别名分析 (Alias analysis) 是流不敏感的，所以这些赋值语句之间的控制流是 irrelevant ，这些赋值语句可以按照任意顺序执行任意多次

论文中使用一种称为 Program Expression Graph (PEG) 的数据结构来表示程序中出现的所有的表达式和赋值语句，图中的结点表示程序中的表达式，结点之间的边分为两类：

- Pointer dereference edges (D)，用于表示指针解引用语句
- Assignment edges (A)，用于表示赋值语句

对于每一条 A 边和 D 边，都有一条对应的反方向 (reverse) 的边与之对应，用在 A 上加上划线，在 D 上加上划线来表示。

$$\overline{A}\overline{D}$$

一个 PEG 的示例，如下图所示：

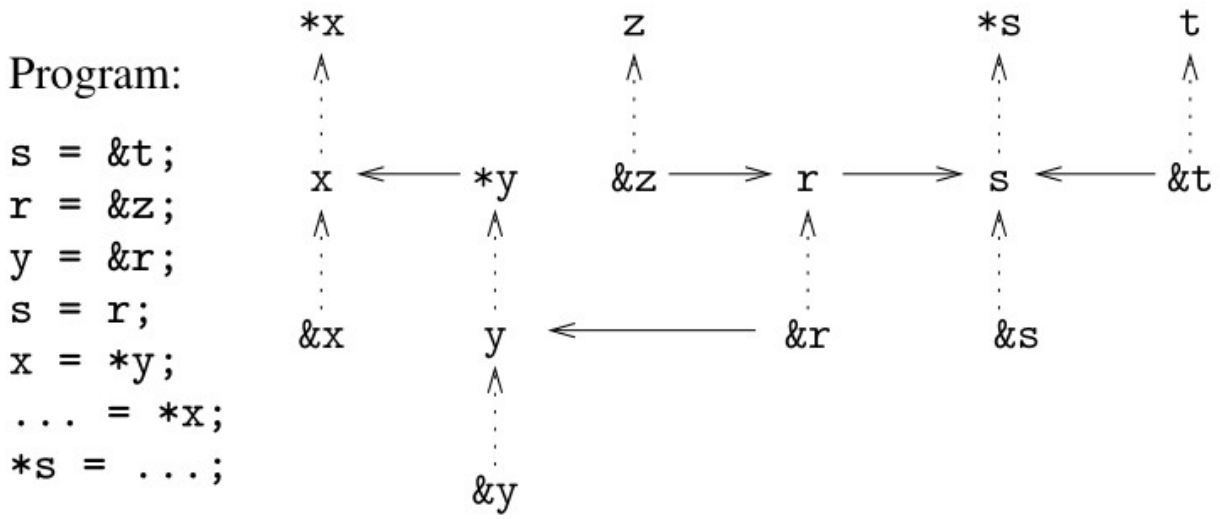


Fig. 1: Figure.1-PEG

该图的左边显示了程序中包含的指针赋值语句，该图的右边就是该程序的 PEG 。水平的实线表示 Assignment edges (A- edges)，垂直的虚线表示 Pointer dereference edges (D-edges)。

Alias Analysis via CFL-Reachabiliry

论文中定义了两种别名关系：

- Memory (or location) alias: 如果两个表达式的内存位置相同，那么它们就是 memory alias
- Value alias: 如果两个表达式在计算 (evaluate) 后得到相同的指针值，那么他们就是 value alias

论文中用二元关系 M 来描述 memory alias：$M \subseteq Expr \times Expr$，用二元关系 V 来描述 value alias：$V \subseteq Expr \times Expr$。每一个关系都可以看作是 PEG 中的一条边，这样就将 M 和 V 的计算形式化为一个在 PEG 上的 context-free language (CFL) reachability problem (见 “Program Analysis via Graph Reachability” http://research.cs.wisc.edu/wpis/papers/tr1386.pdf 和 “The Set Constraint/CFL Reachability Connection in Practice” https://theory.stanford.edu/~aiken/publications/papers/pldi04.pdf)。CFL-Reachability 的大致思想是，给定一个图，图上的边都带有标记，则图中的结点之间是否满足关系 R 就可以通过以下方式形式化为 CFL-Reachability 问题来进行判断：构造一个文法 G，结点 n 与 n‘之间满足关系 R 当且仅当图中由结点 n 至 n’之间的边上的标记所构成的序列属于文法 G 所定义的语言 L(G)。

$$
\begin{array}{lrl}
\text{Memory aliases:} & M & ::= \quad \overline{D}\ V\ D \\
\text{Value aliases:} & V & ::= \quad \overline{F}\ M?\ F \\
\text{Flows of values:} & F & ::= \quad (A\ M?)^* \\
 & \overline{F} & ::= \quad (M?\ \overline{A})^*
\end{array}
$$

Fig. 2: Figure.2-Grammar1

下图是该论文为解决别名分析问题，在 PEG 图上构造的上下文无关文法 G0：

文法中的“?”表示该符号是可选的，文法中的“*”是 Kleene star operator，符号 D 和 A 是终结符，其余的符号都是非终结符，D 即 PEG 图中的 Pointer dereference edges，A 即 PEG 图中的 Assignment edges。可以消除 Figure.2 所示的文法 G0 中的非终结符 F 得到新的文法 G，如下图 Figure.3 所示：

$$
\begin{array}{lrl}
M & ::= & \overline{D}\ V\ D \\
V & ::= & (M?\ \overline{A})^*\ M?\ (A\ M?)^*
\end{array}
$$

Fig. 3: Figure.3-Grammar2

下面以 Figure.1 所示的 PEG 为例，说明如何判断 `*x` 和 `*s` 之间是否为 memory alias。首先，表达式 `&r` 和 `y` 是 value alias，`V(&r, y)`，因为存在 `y = &r` 这样一条语句，即存在这样一条边 `A(&r, y)`，符合文法 G 中非终结符 V 的产生式。因此对 `&r` 和 `y` 的解引用得到的表达式之间就是 memory alias，即 `M(r, *y)`，表达式 `r` 和 `*y` 之间是 memory alias；然后，存在语句 `*y` 的值流向了 `x`，`r` 的值流向了 `s`，二元关系 (x, s) 符合文法 G 中非终结符 V 的产生式，所以 `x` 和 `s` 之间是 value alias，V(x, s)。所以 `*x` 和 `*s` 之间是 memory alias，`M( *x, *s)`。从 CFL-reachability 的角度来看，在 PEG 中存在一条由 `*x` 到 `*s` 的路径 `[ *x, x, *y, y, &r, r, s, *s ]`，对应于 PEG 中边的序列：

$$\overline{DADADAD}$$

该序列满足文法 G 中 M 的产生式，所以 `*x` 和 `*s` 之间是 memory alias，`M( *x, *s)`。

Hierarchical State Machine Representation

使用 Hierarchical State Machine 来表示 Figure.3 所示的上下文无关文法 G，下一节中的 alias analysis algorithm 就是基于该 Hierarchical State Machine Model 来构建的，Hierarchical State Machine 是一种自动机，它的结点是 state 或者是 box，state 就和常见的有限状态自动机中的 state 一样，而 box 表示对另外一个 state machine 的调用，并且 box 有一系列的输入和输出，与被调用的 state machine 的初始状态和结束状态相对应。

根据 Figure.3 所示的上下文无关文法 G 所构造的 hierarchical, recursive state machine 如下图所示：

图的上半部分是 memory alias M 所对应的 hierarchical state machine，下半部分是 value alias V 所对应的 hierarchical state machine。

Alias Analysis Algorithm

论文提出的 Alias Analysis Algorithm 如下所示（论文原文中的算法疑似有 typo，下图所示的算法是我修正后的算法，如有错误，欢迎指正）：

算法中 worklist 中的元素是三元组 `<s, n, c>`，表示 PEG 中的结点 s 到达了结点 n，并且此时的自动机状态为 c 。算法中出现的一些函数的意义如下：

- *addr(n)* 表示对结点 n 的取地址操作后得到的结点，对应于 PEG 中的 reverse dereference edge；
- *deref(n)* 表示对结点 n 的解引用操作后得到的结点，对应于 PEG 中的 dereference edge；
- *assignTo(n)* 表示对所有的被 n 赋值的那些结点，即在 PEG 中以结点 n 为起点的那些 assignment edges 的终点结点，对应于 PEG 中的 assignment edges；
- *assignFrom(n)* 表示所有的赋值给 n 的那些结点，即在 PEG 中以结点 n 为终点的那些 assignment edges 的起点结点，对应于 PEG 中的 reverse assignment edges；
- *reach(n)* 是二元组 `<s, c>` 的集合，表示 PEG 中的结点 s 到达了结点 n，并且此时的自动机状态为 c ；
- *aliasMem(n)* 是那些当前已知的结点 n 的 memory alias 的结点；

下面还是以 Figure.1 所示的 PEG 为例，说明如何使用该算法判断 `*x` 和 `*s` 之间是否为 memory alias。

- 首先，worklist 被初始化为 `{ <x, x, S1> }`；
- 进入第一轮循环，取出 worklist 中的元素 `<x, x, S1>`，因为 `assignFrom(x)` 为 `{ *y }`（执行“propagate reachability through value flows” 部分的代码）并且 `addr(x) != null && c == S1`（执行” propagate information downward” 部分的代码），所以第一轮循环后 worklist 为 `{ <x, *y, S1>, <&x, &x, S1> }`；
- 进入第二轮循环，取出 worklist 中的元素：`<x, *y, S1>`，因为 `assignTo(*y)` 为 `{ x }`（执行“propagate reachability through value flows” 部分的代码）并且 `addr(*y) != null && c == S1`（执行” propagate information downward” 部分的代码），所以第二轮循环后 worklist 为 `{ <x, x, S3>, <y, y, S1> , <&x, &x, S1>}`，`reach(*y)` 为 `{ <x, S1> }`；
- 进入第三轮循环，取出 worklist 中的元素：`<y, y, S1>`，因为 `assignFrom(y)` 为 `{ &r }`（执行“propagate reachability through value flows” 部分的代码）并且 `addr(y) != null && c == S1`（执行”

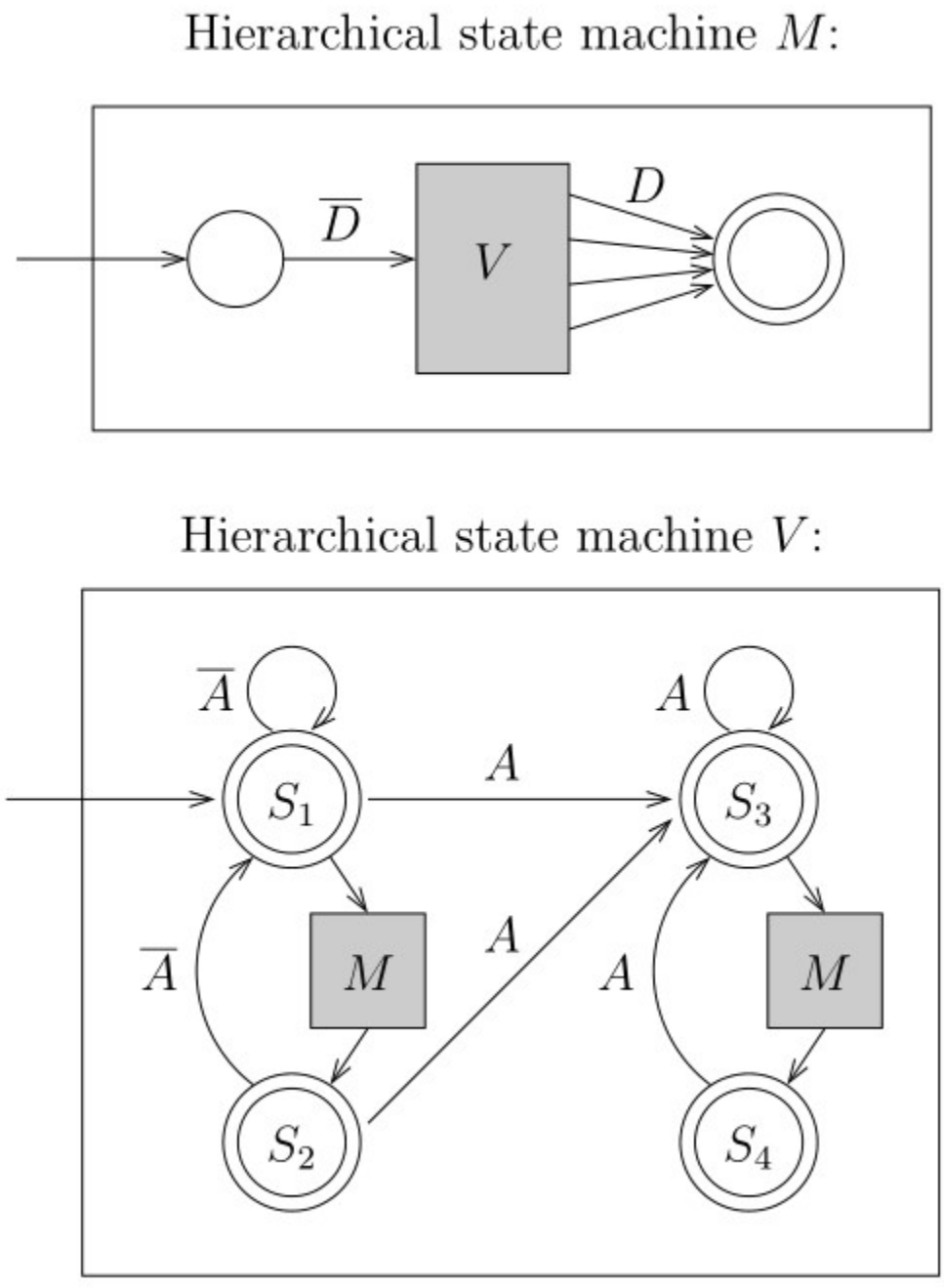


Fig. 4: Figure.4-Hierarchical-State-Machine

```
MAYALIAS(e1 : Expr, e2 : Expr)
 1  /* initialize worklist */
 2  w ← { ⟨addr(e1), addr(e1), S1⟩ }
 3
 4  while (w is not empty)
 5      remove ⟨s, n, c⟩ from w
 6      s' ← deref(s)
 7      n' ← deref(n)
 8
 9      /* check if the destination has been reached */
10      if (s' = e1 ∧ n' = e2)
11        then return true
12
13      /* propagate information upward */
14      if (n' ≠ null ∧ n' ∉ aliasMem(s'))
15        then aliasMem(s') = aliasMem(s') ∪ {n'}
16             for each ⟨s'', c''⟩ in reach(s') :
17                 switch (c'')
18                   case S1 : PROPAGATE(w, n', s'', S2)
19                   case S3 : PROPAGATE(w, n', s'', S4)
20
21      /* propagate reachability through value flows */
22      switch (c)
23        case S1 :
24             for each m in assignFrom(n) :
25                 PROPAGATE(w, m, s, S1)
26             for each m in aliasMem(n) :
27                 PROPAGATE(w, m, s, S2)
28             for each m in assignTo(n) :
29                 PROPAGATE(w, m, s, S3)
30
31        case S2 :
32             for each m in assignFrom(n) :
33                 PROPAGATE(w, m, s, S1)
34             for each m in assignTo(n) :
35                 PROPAGATE(w, m, s, S3)
36
37        case S3 :
38             for each m in assignTo(n) :
39                 PROPAGATE(w, m, s, S3)
40             for each m in aliasMem(n) :
41                 PROPAGATE(w, m, s, S4)
42
43        case S4 :
44             for each m in assignTo(n) :
45                 PROPAGATE(w, m, s, S3)
46
47      /* propagate information downward */
48      if (addr(n) ≠ null ∧ (c = S1 ∨ c = S3))
49        then PROPAGATE(w, addr(n), addr(n), S1)
50
51  return false


PROPAGATE(w, n, s, c)
1  if (⟨s, c⟩ ∉ reach(n))
2    then reach(n) ← reach(n) ∪ { ⟨s, c⟩ }
3         w ← w ∪ { ⟨s, n, c⟩ }
```

Fig. 5: Figure.5-Alias-Analysis-Algorithm

propagate information downward" 部分的代码)，所以第三轮循环后 worklist 为 { <y, &r, S1>, <&y, &y, S1>, <x, x, S3>, <&x, &x, S1> };

- 进入第四轮循环，取出 worklist 中的元素：<y, &r, S1>，s' 为 *y，n' 为 r，aliasMem(*y) 为空，reach(*y) 为 { <x, S1> }，因为 r != null && r 不属于 aliasMem(*y)（执行 "propagate information upward" 部分的代码)，所以第四轮循环后 worklist 为 { <x, r, S2>, <&y, &y, S1>, <x, x, S3>, <&x, &x, S1> };
- 进入第五轮循环，取出 worklist 中的元素：<x, r, S2>，因为 assignTo(r) 为 { s }，assignFrom(r) 为 { &z }（执行 "propagate reachability through value flows" 部分的代码)，所以第五轮循环后 worklist 为 { <x, s, S3>, <x, &z, S1>, <&y, &y, S1>, <x, x, S3>, <&x, &x, S1> };
- 进入第六轮循环，取出 worklist 中的元素：<x, s, S3>，s' 为 *x，n' 为 *s，因为 s' == e1 && n' == e2，所以 *x 和 *s 之间为 memory alias，算法结束。

Implementation of CFL Anderson Alias Analysis

CFL-Anderson-AliasAnalysis 的代码实现位于 llvm-5.0.1.src/include/llvm/Analysis/CFLAndersAliasAnalysis.h 和 llvm-5.0.1.src/lib/Analysis/CFLAndersAliasAnalysis.cpp。在前面的章节提到过，对于 LLVM 中不同的别名分析算法，在实现时都要定义一个继承自 AAResultBase 的 Result 类，并重写函数 AliasResult alias(const MemoryLocation &, const MemoryLocation &)，对于 CFL-Anderson-AliasAnalysis 就是 CFLAndersAAResult::alias 函数。CFLAndersAAResult::alias 函数体如下：

```
AliasResult CFLAndersAAResult::alias(const MemoryLocation &LocA,
                                     const MemoryLocation &LocB) {
  if (LocA.Ptr == LocB.Ptr)
    return LocA.Size == LocB.Size ? MustAlias : PartialAlias;

  // Comparisons between global variables and other constants should be
  // handled by BasicAA.
  // CFLAndersAA may report NoAlias when comparing a GlobalValue and
  // ConstantExpr, but every query needs to have at least one Value tied to a
  // Function, and neither GlobalValues nor ConstantExprs are.
  if (isa<Constant>(LocA.Ptr) && isa<Constant>(LocB.Ptr))
    return AAResultBase::alias(LocA, LocB);

  AliasResult QueryResult = query(LocA, LocB);
  if (QueryResult == MayAlias)
    return AAResultBase::alias(LocA, LocB);

  return QueryResult;
}
```

CFLAndersAAResult::alias 函数的参数为两个 MemoryLocation 类型的变量。MemoryLocation 类

有 3 个成员变量：`const Value *Ptr`, `uint64_t Size`, `AAMDNodes AATags`。`MemoryLocation` 用于表示一个指定的内存位置。`const Value *Ptr` 用于记录起始地址；`uint64_t Size` 用于记录该内存位置的大小，如果大小不确定用 UnknownSize (在 `MemoryLocation` 中定义) 来表示；`AAMDNodes AATags` 用于记录该内存位置的有关别名信息的 metadata 信息。

该函数的第一部分代码：

```
if (LocA.Ptr == LocB.Ptr)
  return LocA.Size == LocB.Size ? MustAlias : PartialAlias;
```

判断两个 `MemoryLocation` 类型的参数的起始地址是否相同，如果起始地址相同，并且大小也相同，那么返回 MustAlias，大小不同，则返回 PartialAlias（因为此时两个 `MemoryLocation` 类型的对象必定有重叠部分）。

第二部分代码，对于 CFLAndersAA 不能处理的情况，调用了 `AAResultBase::alias` 进行处理（注释：`GlobalValue` 是 `Constant` 的子类）。

第三部分代码，CFLAndersAA 的核心代码的入口，使用 CFLAndersAA 算法来判断两个 `MemoryLocation` 类型的对象的别名关系。

`CFLAndersAAResult::query` 的实现如下：

```
AliasResult CFLAndersAAResult::query(const MemoryLocation &LocA,
                                     const MemoryLocation &LocB) {
  auto *ValA = LocA.Ptr;
  auto *ValB = LocB.Ptr;

  if (!ValA->getType()->isPointerTy() || !ValB->getType()->isPointerTy())
    return NoAlias;

  auto *Fn = parentFunctionOfValue(ValA);
  if (!Fn) {
    Fn = parentFunctionOfValue(ValB);
    if (!Fn) {
      // The only times this is known to happen are when globals + InlineAsm are
      // involved
      DEBUG(dbgs()
            << "CFLAndersAA: could not extract parent function information.\n");
      return MayAlias;
    }
  } else {
    assert(!parentFunctionOfValue(ValB) || parentFunctionOfValue(ValB) == Fn);
  }

  assert(Fn != nullptr);
  auto &FunInfo = ensureCached(*Fn);
```

(continues on next page)

(continued from previous page)

```
  // AliasMap lookup
  if (FunInfo->mayAlias(ValA, LocA.Size, ValB, LocB.Size))
    return MayAlias;
  return NoAlias;
}
```

注意到，`assert(Fn != nullptr);` 之前的语句都是一些前置条件的判断及处理，在 `CFLAndersAAResult::query` 函数中判断两个 `MemoryLocation` 类型的变量是否为别名关系时，最核心是语句 `FunInfo->mayAlias(ValA, LocA.Size, ValB, LocB.Size)`，即对函数 `CFLAndersAAResult::FunctionInfo::mayAlias` 的调用。

`FunctionInfo` 类的定义如下：

```
class CFLAndersAAResult::FunctionInfo {
  /// Map a value to other values that may alias it
  /// Since the alias relation is symmetric, to save some space we assume values
  /// are properly ordered: if a and b alias each other, and a < b, then b is in
  /// AliasMap[a] but not vice versa.
  DenseMap<const Value *, std::vector<OffsetValue>> AliasMap;

  /// Map a value to its corresponding AliasAttrs
  DenseMap<const Value *, AliasAttrs> AttrMap;

  /// Summary of externally visible effects.
  AliasSummary Summary;

  Optional<AliasAttrs> getAttrs(const Value *) const;

public:
  FunctionInfo(const Function &, const SmallVectorImpl<Value *> &,
               const ReachabilitySet &, const AliasAttrMap &);

  bool mayAlias(const Value *, uint64_t, const Value *, uint64_t) const;
  const AliasSummary &getAliasSummary() const { return Summary; }
};
```

`FunctionInfo` 类的成员变量的意义由注释写的很清楚，成员变量 `AliasMap` 用于表示 value 与其可能互为别名的其他 value 的映射，成员变量 `AttrMap` 用于表示 value 与其 AliasAttrs 属性。成员变量 `Summary` 用于表示该函数的参数/返回值之间的别名关系等信息的摘要，这样当处理对某个函数的调用点时，可以通过该摘要信息得到实参/返回值之间的别名关系等信息。

函数 `CFLAndersAAResult::FunctionInfo::mayAlias` 的定义如下：

```
bool CFLAndersAAResult::FunctionInfo::mayAlias(const Value *LHS,
                                               uint64_t LHSSize,
                                               const Value *RHS,
                                               uint64_t RHSSize) const {
  assert(LHS && RHS);

  // Check if we've seen LHS and RHS before. Sometimes LHS or RHS can be created
  // after the analysis gets executed, and we want to be conservative in those
  // cases.
  auto MaybeAttrsA = getAttrs(LHS);
  auto MaybeAttrsB = getAttrs(RHS);
  if (!MaybeAttrsA || !MaybeAttrsB)
    return true;

  // Check AliasAttrs before AliasMap lookup since it's cheaper
  auto AttrsA = *MaybeAttrsA;
  auto AttrsB = *MaybeAttrsB;
  if (hasUnknownOrCallerAttr(AttrsA))
    return AttrsB.any();
  if (hasUnknownOrCallerAttr(AttrsB))
    return AttrsA.any();
  if (isGlobalOrArgAttr(AttrsA))
    return isGlobalOrArgAttr(AttrsB);
  if (isGlobalOrArgAttr(AttrsB))
    return isGlobalOrArgAttr(AttrsA);

  // At this point both LHS and RHS should point to locally allocated objects

  auto Itr = AliasMap.find(LHS);
  if (Itr != AliasMap.end()) {

    // Find out all (X, Offset) where X == RHS
    auto Comparator = [](OffsetValue LHS, OffsetValue RHS) {
      return std::less<const Value *>()(LHS.Val, RHS.Val);
    };
#ifdef EXPENSIVE_CHECKS
    assert(std::is_sorted(Itr->second.begin(), Itr->second.end(), Comparator));
#endif
    auto RangePair = std::equal_range(Itr->second.begin(), Itr->second.end(),
                                      OffsetValue{RHS, 0}, Comparator);

    if (RangePair.first != RangePair.second) {
      // Be conservative about UnknownSize
      if (LHSSize == MemoryLocation::UnknownSize ||
```

(continues on next page)

(continued from previous page)

```
          RHSSize == MemoryLocation::UnknownSize)
        return true;

      for (const auto &OVal : make_range(RangePair)) {
        // Be conservative about UnknownOffset
        if (OVal.Offset == UnknownOffset)
          return true;

        // We know that LHS aliases (RHS + OVal.Offset) if the control flow
        // reaches here. The may-alias query essentially becomes integer
        // range-overlap queries over two ranges [OVal.Offset, OVal.Offset +
        // LHSSize) and [0, RHSSize).

        // Try to be conservative on super large offsets
        if (LLVM_UNLIKELY(LHSSize > INT64_MAX || RHSSize > INT64_MAX))
          return true;

        auto LHSStart = OVal.Offset;
        // FIXME: Do we need to guard against integer overflow?
        auto LHSEnd = OVal.Offset + static_cast<int64_t>(LHSSize);
        auto RHSStart = 0;
        auto RHSEnd = static_cast<int64_t>(RHSSize);
        if (LHSEnd > RHSStart && LHSStart < RHSEnd)
          return true;
      }
    }
  }

  return false;
}
```

函数 CFLAndersAAResult::FunctionInfo::mayAlias 的注释写的很详细，不再赘述。

在 CFLAndersAAResult::query 函数中判断两个 MemoryLocation 类型的变量是否为别名关系时，最核心是语句 FunInfo->mayAlias(ValA, LocA.Size, ValB, LocB.Size)，而 FunInfo 则是函数 CFLAndersAAResult::ensureCached 的返回值，下面说明在函数 CFLAndersAAResult::ensureCached 中是如何构造 FunInfo 的。

CFLAndersAAResult::ensureCached 的实现如下：

```
const Optional<CFLAndersAAResult::FunctionInfo> &
CFLAndersAAResult::ensureCached(const Function &Fn) {
  auto Iter = Cache.find(&Fn);
  if (Iter == Cache.end()) {
```

(continues on next page)

(continued from previous page)

```
    scan(Fn);
    Iter = Cache.find(&Fn);
    assert(Iter != Cache.end());
    assert(Iter->second.hasValue());
  }
  return Iter->second;
}
```

在 CFLAndersAAResult 中使用缓存，存储函数 Fn 的 CFLAndersAAResult::FunctionInfo 信息。在函数 scan 中通过调用 buildInfoFrom 函数来构造函数 Fn 的 CFLAndersAAResult::FunctionInfo 信息，并加入到缓存 Cache 中。

buildInfoFrom 函数的定义如下：

```
CFLAndersAAResult::FunctionInfo
CFLAndersAAResult::buildInfoFrom(const Function &Fn) {
  CFLGraphBuilder<CFLAndersAAResult> GraphBuilder(
      *this, TLI,
      // Cast away the constness here due to GraphBuilder's API requirement
      const_cast<Function &>(Fn));
  auto &Graph = GraphBuilder.getCFLGraph();

  ReachabilitySet ReachSet;
  AliasMemSet MemSet;

  std::vector<WorkListItem> WorkList, NextList;
  initializeWorkList(WorkList, ReachSet, Graph);
  // TODO: make sure we don't stop before the fix point is reached
  while (!WorkList.empty()) {
    for (const auto &Item : WorkList)
      processWorkListItem(Item, Graph, ReachSet, MemSet, NextList);

    NextList.swap(WorkList);
    NextList.clear();
  }

  // Now that we have all the reachability info, propagate AliasAttrs according
  // to it
  auto IValueAttrMap = buildAttrMap(Graph, ReachSet);

  return FunctionInfo(Fn, GraphBuilder.getReturnValues(), ReachSet,
                      std::move(IValueAttrMap));
}
```

buildInfoFrom 函数体的第一部分代码，为函数 Fn 建立 CFLGraph（与论文中的 Program Expression Graph

相对应，做了一些修改)。

```
CFLAndersAAResult::FunctionInfo
CFLAndersAAResult::buildInfoFrom(const Function &Fn) {
  CFLGraphBuilder<CFLAndersAAResult> GraphBuilder(
      *this, TLI,
      // Cast away the constness here due to GraphBuilder's API requirement
      const_cast<Function &>(Fn));
  auto &Graph = GraphBuilder.getCFLGraph();
  ..... // 省略
}
```

在 CFLGraph 中，结点用数据结构 Node (`typedef InstantiatedValue Node`) 来表示，该数据结构有两个成员变量：`Value *Val` 和 `unsigned DerefLevel` 。与论文中的 PEG 不同的是，CFLGraph 中的 edge 表示的只是 assignment edges，而 pointer dereference edges 则是**隐式**地保存在 CFLGraph 中，即：对于每一个结点 (Val, DerefLevel) 都有一条连向 (Val, DerefLevel+1) 的 dereference edge 和一条连向 (Val, DerefLevel-1) 的 reference edge 。CFLGraph 中的 edge（struct Edge 数据结构有两个成员变量：`Node Other` 和 `int64_t Offset`，Other 就是该结点连向的另外一个结点，offset 是用于描述指针指向复杂结构的某个域的情况，比如一个指针指向的是数组中的某个元素时）是作为一个 Node 的属性出现的，即对于每一个 Node，它有很多条连向其他结点的边；类似地，AliasAttrs 也作为每个 Node 的属性出现。

```
struct NodeInfo {
    EdgeList Edges, ReverseEdges; // 该结点的边集
    AliasAttrs Attr; // 该结点所具有的对别名分析有用的一些属性标记
};
```

在 `CFLGraphBuilder` 中构建 CFLGraph 时通过 vsitor pattern 实现，定义了一个继承自 `InstVisitor` 的 `GetEdgesVisitor` 类，重写 `visitXXX` (xxx 代表不同的的 Instruction，如 `visitGetElementPtrInst`, `visitLoadInst`, `visitStoreInst`, etc) 函数，对不同的 Instruction 执行不同的操作以实现向 CFLGraph 中添加结点和边（处理函数调用有关的 Instruction 时就用到了 FunctionInfo 的成员变量 AliasSummary Summary)。

下面以 LoadInst 为例说明，如何构建 CFLGraph，先举一个简单的 LoadInst 的例子，LoadInst 用于从内存中读取内容：

```
%ptr = alloca i32                               ; yields i32*:ptr
store i32 3, i32* %ptr                          ; yields void
%val = load i32, i32* %ptr                      ; yields i32:val = i32 3
```

上述例子，就是 ptr 指向 i32 大小的内存，该内存的值被写入为 3，然后通过 LoadInst 读取该内存的值记作 val。`visitLoadInst` 函数的定义如下：

```
void visitLoadInst(LoadInst &Inst) {
    auto *From = Inst.getPointerOperand();
    auto *To = &Inst;
    addLoadEdge(From, To);
```

(continues on next page)

(continued from previous page)

```
}
```

From 就是 LLVM IR 中的 ptr，To 就是 LLVM IR 中的 val，然后以它们为参数调用 `addLoadEdge`，`addLoadEdge` 函数体的内容很简单，就是对 `addDerefEdge(From, To, true)` 的调用。`addDerefEdge` 函数体如下：

```
void addDerefEdge(Value *From, Value *To, bool IsRead) {
    assert(From != nullptr && To != nullptr);
    if (!From->getType()->isPointerTy() || !To->getType()->isPointerTy())
        return;
    addNode(From);
    addNode(To);
    if (IsRead) {
        Graph.addNode(InstantiatedValue{From, 1});
        Graph.addEdge(InstantiatedValue{From, 1}, InstantiatedValue{To, 0});
    } else {
        Graph.addNode(InstantiatedValue{To, 1});
        Graph.addEdge(InstantiatedValue{From, 0}, InstantiatedValue{To, 1});
    }
}
```

注意到 `addDerefEdge` 函数的第三个参数为 bool IsRead，并且 `addLoadEdge` 函数在调用 `addDerefEdge` 函数时，将第三个参数设置为 true。

首先对 From 和 To 调用 `addNode` 函数，其函数体如下：

```
void addNode(Value *Val, AliasAttrs Attr = AliasAttrs()) {
    assert(Val != nullptr && Val->getType()->isPointerTy());
    if (auto GVal = dyn_cast<GlobalValue>(Val)) {
        if (Graph.addNode(InstantiatedValue{GVal, 0},
                          getGlobalOrArgAttrFromValue(*GVal)))
            Graph.addNode(InstantiatedValue{GVal, 1}, getAttrUnknown());
    } else if (auto CExpr = dyn_cast<ConstantExpr>(Val)) {
        if (hasUsefulEdges(CExpr)) {
            if (Graph.addNode(InstantiatedValue{CExpr, 0}))
                visitConstantExpr(CExpr);
        }
    } else
        Graph.addNode(InstantiatedValue{Val, 0}, Attr);
}
```

如果 Val 是 `GlobalValue`，并且 `Graph.addNode(InstantiatedValue{GVal, 0}, getGlobalOrArgAttrFromValue(*GVal))` 返回 true，则将 `InstantiatedValue{GVal, 1}` 加入到 CFLGraph 中，**猜测**加 `InstantiatedValue{GVal, 1}` 应该为了是维护 CFLGraph 中的隐式的由 (Val, DerefLevel) 连向

(Val, DerefLevel+1) 的 dereference edge 。如果 Val 是我们关心的 `ConstantExpr`（指那些不包括 Cmp 指令的 ConstantExpr，关于 ConstantExpr，https://llvm.org/docs/LangRef.html#constant-expressions），并且 `Graph.addNode(InstantiatedValue{CExpr, 0})` 返回 true，调用 `visitConstantExpr(CExpr)`，在 `visitConstantExpr(CExpr)` 中，根据 `ConstantExpr` 的不同，执行 `addNode`，`addAssignEdge` 等操作。如果 Val 既不是 `GlobalValue` 也不是 `ConstantExpr`，直接将 `InstantiatedValue{Val, 0}` 加入 CFLGraph 中。

因为在 `addLoadEdge` 函数中调用 `addDerefEdge` 函数时，将第三个参数 `IsRead` 设置为 true ，所以对 From 和 To 调用 `addNode` 函数后，进入 `IsRead` 为 true 的分支：

```
Graph.addNode(InstantiatedValue{From, 1});
Graph.addEdge(InstantiatedValue{From, 1}, InstantiatedValue{To, 0});
```

首先调用 `Graph.addNode` 将结点 {From, 1} 加入到 CFLGraph 中，然后调用 `Graph.addEdge` 将以结点 {From, 1} 为起点，以结点 {To, 0} 为终点的边加入到 CFLGraph 中。`%val = load i32, i32* %ptr` 可以看作是这样一条 C-like 语句：`val = *ptr`，将结点 {From, 1} 加入到 CFLGraph 中是为了隐式地增加了一条由 {From, 0} 连向 {From, 1} 的边，对应论文中 PEG 中的边 `ptr -> *ptr`；将以结点 {From, 1} 为起点，以结点 {To, 0} 为终点的边加入到 CFLGraph 中，对应论文中 PEG 中的边 `*ptr -> val` 。

注意，前述代码中 `addNode` 与 `Graph.addNode` 不同，`addNode` 是 `CFLGraphBuilder` 的成员函数，Graph 是 `CFLGraphBuilder` 的一个 `CFLGraph` 类型的成员变量，`Graph.addNode` 即 `CFLGraph::addNode`

`CFLGraph::addNode` 代码如下：

```
bool addNode(Node N, AliasAttrs Attr = AliasAttrs()) {
  assert(N.Val != nullptr);
  auto &ValInfo = ValueImpls[N.Val];
  auto Changed = ValInfo.addNodeToLevel(N.DerefLevel);
  ValInfo.getNodeInfoAtLevel(N.DerefLevel).Attr |= Attr;
  return Changed;
}
```

`CFLGraph::addEdge` 代码如下：

```
void addEdge(Node From, Node To, int64_t Offset = 0) {
    auto *FromInfo = getNode(From);
    assert(FromInfo != nullptr);
    auto *ToInfo = getNode(To);
    assert(ToInfo != nullptr);

    FromInfo->Edges.push_back(Edge{To, Offset});
    ToInfo->ReverseEdges.push_back(Edge{From, Offset});
}
```

`buildInfoFrom` 函数体的第二部分，通过 worklist 算法计算 ReachSet 和 MemSet 。

```
CFLAndersAAResult::FunctionInfo
CFLAndersAAResult::buildInfoFrom(const Function &Fn) {
  ...... // 省略
  ReachabilitySet ReachSet;
  AliasMemSet MemSet;

  std::vector<WorkListItem> WorkList, NextList;
  initializeWorkList(WorkList, ReachSet, Graph);
  // TODO: make sure we don't stop before the fix point is reached
  while (!WorkList.empty()) {
    for (const auto &Item : WorkList)
      processWorkListItem(Item, Graph, ReachSet, MemSet, NextList);

    NextList.swap(WorkList);
    NextList.clear();
  }
  ...... // 省略
}
```

`ReachabilitySet` 类用于实现论文中提出的 MAYALIAS 算法中的 *reach(n)* 。`AliasMemSet` 类用于实现 MAYALIAS 算法中的 *aliasMem(n)* 。

`WorkListItem` 的定义如下：

```
struct WorkListItem {
  InstantiatedValue From;
  InstantiatedValue To;
  MatchState State;
};
```

`WorkListItem` 与 MAYALIAS 算法中的 worklist element 相对应，是一个 `<From, To, State>` 三元组。

MatchState 表示自动机的状态：

```
enum class MatchState : uint8_t {
  // The following state represents S1 in the paper.
  FlowFromReadOnly = 0,
  // The following two states together represent S2 in the paper.
  FlowFromMemAliasNoReadWrite,
  FlowFromMemAliasReadOnly,
  // The following two states together represent S3 in the paper.
  FlowToWriteOnly,
  FlowToReadWrite,
  // The following two states together represent S4 in the paper.
  FlowToMemAliasWriteOnly,
```

(continues on next page)

(continued from previous page)

```
  FlowToMemAliasReadWrite,
};
```

- state `FlowFromReadOnly` 用于表示 Figure.4 所示的 hierarchical state machine 中的 state S1；
- state `FlowFromMemAliasNoReadWrite` 和 `FlowFromMemAliasReadOnly` 用于表示 hierarchical state machine 中的 state S2；
- state `FlowToWriteOnly` 和 `FlowToReadWrite` 用于表示 hierarchical state machine 中的 state S3；
- state `FlowToMemAliasWriteOnly` 和 `FlowToMemAliasReadWrite` 用于表示 hierarchical state machine 中的 state S4；
- 其中后缀 `ReadOnly` 表示存在一条不包含 reverse assignment edges 的别名路径，后缀 `WriteOnly` 表示存在一条只包含 assignment edges 的别名路径，后缀 `ReadWrite` 表示存在存在一条只包含 assignment 和 reverse assignment edges 的别名路径，后缀 `WriteOnly` 表示存在一条只包含 assignment edges 的别名路径，后缀 `NoReadWrite` 表示存在一条不包含 assignment 和 reverse assignment edges 的别名路径。

`initializeWorkList` 函数的定义如下：

```
static void initializeWorkList(std::vector<WorkListItem> &WorkList,
                               ReachabilitySet &ReachSet,
                               const CFLGraph &Graph) {
  for (const auto &Mapping : Graph.value_mappings()) {
    auto Val = Mapping.first;
    auto &ValueInfo = Mapping.second;
    assert(ValueInfo.getNumLevels() > 0);

    // Insert all immediate assignment neighbors to the worklist
    for (unsigned I = 0, E = ValueInfo.getNumLevels(); I < E; ++I) {
      auto Src = InstantiatedValue{Val, I};
      // If there's an assignment edge from X to Y, it means Y is reachable from
      // X at S3 and X is reachable from Y at S1
      for (auto &Edge : ValueInfo.getNodeInfoAtLevel(I).Edges) {
        propagate(Edge.Other, Src, MatchState::FlowFromReadOnly, ReachSet,
                  WorkList);
        propagate(Src, Edge.Other, MatchState::FlowToWriteOnly, ReachSet,
                  WorkList);
      }
    }
  }
}
```

在 `initializeWorkList` 函数中，处理 CFLGraph 中的 assignment edges，如果存在一条 assignment edge `X -> Y` (即存在语句 y = x)，意味着 X 能够到达 Y，并且此时的自动机状态是 S3；Y 能够到达 X，并且此时的自动机状态是 S1。这里通过调用 `propagate` 函数来上述三元组 `<Y, X, S1>`，`<X, Y, S3>` 加入到 worklist

中，并且更新 ReachSet 。

propagate 函数就是论文中 MAYALIAS 算法中的 PROPAGATE 函数的实现。

```
static void propagate(InstantiatedValue From, InstantiatedValue To,
                      MatchState State, ReachabilitySet &ReachSet,
                      std::vector<WorkListItem> &WorkList) {
  if (From == To)
    return;
  if (ReachSet.insert(From, To, State))
    WorkList.push_back(WorkListItem{From, To, State});
}
```

回到 buildInfoFrom 函数中，在对 worklist 进行初始化后，不断更新 worklist 直至到达不动点，关键函数是 processWorkListItem。

processWorkListItem 第一部分代码如下，对应于论文中 MAYALIAS 算法中的 “propagate information upward” 部分：

```
static void processWorkListItem(const WorkListItem &Item, const CFLGraph &Graph,
                                ReachabilitySet &ReachSet, AliasMemSet &MemSet,
                                std::vector<WorkListItem> &WorkList) {
  auto FromNode = Item.From;
  auto ToNode = Item.To;

  auto NodeInfo = Graph.getNode(ToNode);
  assert(NodeInfo != nullptr);

  // The newly added value alias pair may pontentially generate more memory
  // alias pairs. Check for them here.
  auto FromNodeBelow = getNodeBelow(Graph, FromNode);
  auto ToNodeBelow = getNodeBelow(Graph, ToNode);
  if (FromNodeBelow && ToNodeBelow &&
      MemSet.insert(*FromNodeBelow, *ToNodeBelow)) {
    propagate(*FromNodeBelow, *ToNodeBelow,
              MatchState::FlowFromMemAliasNoReadWrite, ReachSet, WorkList);
    for (const auto &Mapping : ReachSet.reachableValueAliases(*FromNodeBelow)) {
      auto Src = Mapping.first;
      auto MemAliasPropagate = [&](MatchState FromState, MatchState ToState) {
        if (Mapping.second.test(static_cast<size_t>(FromState)))
          propagate(Src, *ToNodeBelow, ToState, ReachSet, WorkList);
      };

      MemAliasPropagate(MatchState::FlowFromReadOnly,
                        MatchState::FlowFromMemAliasReadOnly);
      MemAliasPropagate(MatchState::FlowToWriteOnly,
```

(continues on next page)

(continued from previous page)

```
                        MatchState::FlowToMemAliasWriteOnly);
      MemAliasPropagate(MatchState::FlowToReadWrite,
                        MatchState::FlowToMemAliasReadWrite);
    }
  }
  ..... // 省略
}
```

函数 `getNodeBelow` 就是输入一个 CFLGraph 和一个 Node { Val, DerefLevel }，返回对 Node 的解引用，即 {Val, DerefeLevel+1}。对于当前的 WorkList item，通过调用 `getNodeBelow`，获得 FromNode 和 ToNode 的解引用对应的结点 FromNodeBelow 和 ToNodeBelow，如果 FromNodeBelow 和 ToNodeBelow 不在 aliasMemSet 集合中，将其加入到 aliasMemSet 中，然后根据 FromNodeBelow 在 ReachSet 中的元素的情况，调用 propagate 函数更新 WorkList 和 ReachSet 。注意到此处实现与 MayAlias 算法有一处不同：`propagate(*FromNodeBelow, *ToNodeBelow, MatchState::FlowFromMemAliasNoReadWrite, ReachSet, WorkList)`，该条语句将 `<FromNodeBelow, ToNodeBelow, MatchState::FlowFromMemAliasNoReadWrite>` 加入到了 worklist 中，与 hierarchical state machine V 中的 `S1 -> M -> S2` 对应，同时更新了 `ReachSet`。

`processWorkListItem` 的第二部分代码如下，对应于论文中 MayAlias 算法中的 “propagate reachability through value flows” 部分：

```
static void processWorkListItem(const WorkListItem &Item, const CFLGraph &Graph,
                                ReachabilitySet &ReachSet, AliasMemSet &MemSet,
                                std::vector<WorkListItem> &WorkList) {
  ...... // 省略

  // This is the core of the state machine walking algorithm. We expand ReachSet
  // based on which state we are at (which in turn dictates what edges we
  // should examine)
  // From a high-level point of view, the state machine here guarantees two
  // properties:
  // - If *X and *Y are memory aliases, then X and Y are value aliases
  // - If Y is an alias of X, then reverse assignment edges (if there is any)
  // should precede any assignment edges on the path from X to Y.
  auto NextAssignState = [&](MatchState State) {
    for (const auto &AssignEdge : NodeInfo->Edges)
      propagate(FromNode, AssignEdge.Other, State, ReachSet, WorkList);
  };
  auto NextRevAssignState = [&](MatchState State) {
    for (const auto &RevAssignEdge : NodeInfo->ReverseEdges)
      propagate(FromNode, RevAssignEdge.Other, State, ReachSet, WorkList);
  };
  auto NextMemState = [&](MatchState State) {
    if (auto AliasSet = MemSet.getMemoryAliases(ToNode)) {
```

(continues on next page)

(continued from previous page)

```
      for (const auto &MemAlias : *AliasSet)
        propagate(FromNode, MemAlias, State, ReachSet, WorkList);
    }
  };

  switch (Item.State) {
  case MatchState::FlowFromReadOnly: {
    NextRevAssignState(MatchState::FlowFromReadOnly);
    NextAssignState(MatchState::FlowToReadWrite);
    NextMemState(MatchState::FlowFromMemAliasReadOnly);
    break;
  }
  case MatchState::FlowFromMemAliasNoReadWrite: {
    NextRevAssignState(MatchState::FlowFromReadOnly);
    NextAssignState(MatchState::FlowToWriteOnly);
    break;
  }
  case MatchState::FlowFromMemAliasReadOnly: {
    NextRevAssignState(MatchState::FlowFromReadOnly);
    NextAssignState(MatchState::FlowToReadWrite);
    break;
  }
  case MatchState::FlowToWriteOnly: {
    NextAssignState(MatchState::FlowToWriteOnly);
    NextMemState(MatchState::FlowToMemAliasWriteOnly);
    break;
  }
  case MatchState::FlowToReadWrite: {
    NextAssignState(MatchState::FlowToReadWrite);
    NextMemState(MatchState::FlowToMemAliasReadWrite);
    break;
  }
  case MatchState::FlowToMemAliasWriteOnly: {
    NextAssignState(MatchState::FlowToWriteOnly);
    break;
  }
  case MatchState::FlowToMemAliasReadWrite: {
    NextAssignState(MatchState::FlowToReadWrite);
    break;
  }
  }
```

该部分代码的实现与论文 MayAlias 算法中“propagate reachability through value flows”部分一一对应。

`buildInfoFrom` 函数体的第三部分，计算 AliasAttrs 并返回 FunctionInfo 。

```
CFLAndersAAResult::FunctionInfo
CFLAndersAAResult::buildInfoFrom(const Function &Fn) {
  ...... // 省略

  // Now that we have all the reachability info, propagate AliasAttrs according
  // to it
  auto IValueAttrMap = buildAttrMap(Graph, ReachSet);

  return FunctionInfo(Fn, GraphBuilder.getReturnValues(), ReachSet,
                      std::move(IValueAttrMap));
}
```

buildAttrMap 函数的定义如下：

```
static AliasAttrMap buildAttrMap(const CFLGraph &Graph,
                                 const ReachabilitySet &ReachSet) {
  AliasAttrMap AttrMap;
  std::vector<InstantiatedValue> WorkList, NextList;

  // Initialize each node with its original AliasAttrs in CFLGraph
  for (const auto &Mapping : Graph.value_mappings()) {
    auto Val = Mapping.first;
    auto &ValueInfo = Mapping.second;
    for (unsigned I = 0, E = ValueInfo.getNumLevels(); I < E; ++I) {
      auto Node = InstantiatedValue{Val, I};
      AttrMap.add(Node, ValueInfo.getNodeInfoAtLevel(I).Attr);
      WorkList.push_back(Node);
    }
  }

  while (!WorkList.empty()) {
    for (const auto &Dst : WorkList) {
      auto DstAttr = AttrMap.getAttrs(Dst);
      if (DstAttr.none())
        continue;

      // Propagate attr on the same level
      for (const auto &Mapping : ReachSet.reachableValueAliases(Dst)) {
        auto Src = Mapping.first;
        if (AttrMap.add(Src, DstAttr))
          NextList.push_back(Src);
      }

      // Propagate attr to the levels below
```

(continues on next page)

(continued from previous page)

```
      auto DstBelow = getNodeBelow(Graph, Dst);
      while (DstBelow) {
        if (AttrMap.add(*DstBelow, DstAttr)) {
          NextList.push_back(*DstBelow);
          break;
        }
        DstBelow = getNodeBelow(Graph, *DstBelow);
      }
    }
    WorkList.swap(NextList);
    NextList.clear();
  }

  return AttrMap;
}
```

前面提到过，对于每一个 Node 都有一个 NodeInfo 来存储与该 Node 相关的信息：

```
struct NodeInfo {
    EdgeList Edges, ReverseEdges; // 该结点的边集
    AliasAttrs Attr; // 该结点所具有的对别名分析有用的一些属性标记
};
```

`AliasAttrMap` 类是一个用于用于存储 Node 和与其对应的 AliasAttrs 的 Map 结构。`buildAttrMap` 函数的代码逻辑很直观，就是根据 ReachSet 的内容，通过 worklist 算法向其他的相关结点传播 AliasAttrs 信息。

在 `buildInfoFrom` 函数的最后，调用 `FunctionInfo` 的构造函数，返回 `FunctionInfo` 的一个实例。`FunctionInfo` 的构造函数：

```
CFLAndersAAResult::FunctionInfo::FunctionInfo(
    const Function &Fn, const SmallVectorImpl<Value *> &RetVals,
    const ReachabilitySet &ReachSet, const AliasAttrMap &AMap) {
  populateAttrMap(AttrMap, AMap);
  populateExternalAttributes(Summary.RetParamAttributes, Fn, RetVals, AMap);
  populateAliasMap(AliasMap, ReachSet);
  populateExternalRelations(Summary.RetParamRelations, Fn, RetVals, ReachSet);
}
```

`populateAttrMap` 函数的定义如下：

```
static void populateAttrMap(DenseMap<const Value *, AliasAttrs> &AttrMap,
                            const AliasAttrMap &AMap) {
  for (const auto &Mapping : AMap.mappings()) {
    auto IVal = Mapping.first;
```

(continues on next page)

(continued from previous page)

```
    // Insert IVal into the map
    auto &Attr = AttrMap[IVal.Val];
    // AttrMap only cares about top-level values
    if (IVal.DerefLevel == 0)
      Attr |= Mapping.second;
  }
}
```

populateAttrMap 函数就是将 AMap 中 DerefLevel 为 0 的结点的 AliasAttrs 信息复制到 FunctionInfo 类的成员变量 AttrMap 中。

populateExternalAttributes 函数的定义如下：

```
static void populateExternalAttributes(
    SmallVectorImpl<ExternalAttribute> &ExtAttributes, const Function &Fn,
    const SmallVectorImpl<Value *> &RetVals, const AliasAttrMap &AMap) {
  for (const auto &Mapping : AMap.mappings()) {
    if (auto IVal = getInterfaceValue(Mapping.first, RetVals)) {
      auto Attr = getExternallyVisibleAttrs(Mapping.second);
      if (Attr.any())
        ExtAttributes.push_back(ExternalAttribute{*IVal, Attr});
    }
  }
}
```

populateExternalAttributes 函数这里调用了函数 getInterfaceValue，InterfaceValue 是一个含有两个成员变量的结构体：struct InterfaceValue { unsigned Index; unsigned DerefLevel; };，InterfaceValue 用于描述一个函数的参数和返回值，Index 为 0 表示返回值，Index 为非零值时表示第 Index 个函数参数。函数 getInterfaceValue 的原型为 static Optional<InterfaceValue> getInterfaceValue(InstantiatedValue IValue, const SmallVectorImpl<Value *> &RetVals)，如果参数 IValue 是参数或者返回值的话，返回对应的 InterfaceValue，否则返回空指针。这样看来，populateExternalAttributes 函数的功能就很好理解，将 AMAP 中是函数参数或返回值的结点的 ExternallyVisibleAttrs（关于 ExternallyVisibleAttrs 见 AliasAnalysis-Basic 一节）信息存储至 FunctionInfo 的成员变量 Summary.RetParamAttributes 中。

populateAliasMap 函数的定义如下：

```
static void
populateAliasMap(DenseMap<const Value *, std::vector<OffsetValue>> &AliasMap,
                 const ReachabilitySet &ReachSet) {
  for (const auto &OuterMapping : ReachSet.value_mappings()) {
    // AliasMap only cares about top-level values
    if (OuterMapping.first.DerefLevel > 0)
```

(continues on next page)

(continued from previous page)

```
      continue;

    auto Val = OuterMapping.first.Val;
    auto &AliasList = AliasMap[Val];
    for (const auto &InnerMapping : OuterMapping.second) {
      // Again, AliasMap only cares about top-level values
      if (InnerMapping.first.DerefLevel == 0)
        AliasList.push_back(OffsetValue{InnerMapping.first.Val, UnknownOffset});
    }

    // Sort AliasList for faster lookup
    std::sort(AliasList.begin(), AliasList.end());
  }
}
```

populateAliasMap 函数就是根据 ReachSet 的内容，将 AMap 中 DerefLevel 为 0 的结点及与其互为别名的并且 DerefLevel 为 0 的结点加入到 AliasMap 中。

populateExternalRelations 函数的定义如下：

```
static void populateExternalRelations(
    SmallVectorImpl<ExternalRelation> &ExtRelations, const Function &Fn,
    const SmallVectorImpl<Value *> &RetVals, const ReachabilitySet &ReachSet) {
  // If a function only returns one of its argument X, then X will be both an
  // argument and a return value at the same time. This is an edge case that
  // needs special handling here.
  for (const auto &Arg : Fn.args()) {
    if (is_contained(RetVals, &Arg)) {
      auto ArgVal = InterfaceValue{Arg.getArgNo() + 1, 0};
      auto RetVal = InterfaceValue{0, 0};
      ExtRelations.push_back(ExternalRelation{ArgVal, RetVal, 0});
    }
  }

  // Below is the core summary construction logic.
  // A naive solution of adding only the value aliases that are parameters or
  // return values in ReachSet to the summary won't work: It is possible that a
  // parameter P is written into an intermediate value I, and the function
  // subsequently returns *I. In that case, *I is does not value alias anything
  // in ReachSet, and the naive solution will miss a summary edge from (P, 1) to
  // (I, 1).
  // To account for the aforementioned case, we need to check each non-parameter
  // and non-return value for the possibility of acting as an intermediate.
  // 'ValueMap' here records, for each value, which InterfaceValues read from or
```

(continues on next page)

(continued from previous page)

```
  // write into it. If both the read list and the write list of a given value
  // are non-empty, we know that a particular value is an intermidate and we
  // need to add summary edges from the writes to the reads.
  DenseMap<Value *, ValueSummary> ValueMap;
  for (const auto &OuterMapping : ReachSet.value_mappings()) {
    if (auto Dst = getInterfaceValue(OuterMapping.first, RetVals)) {
      for (const auto &InnerMapping : OuterMapping.second) {
        // If Src is a param/return value, we get a same-level assignment.
        if (auto Src = getInterfaceValue(InnerMapping.first, RetVals)) {
          // This may happen if both Dst and Src are return values
          if (*Dst == *Src)
            continue;

          if (hasReadOnlyState(InnerMapping.second))
            ExtRelations.push_back(ExternalRelation{*Dst, *Src, UnknownOffset});
          // No need to check for WriteOnly state, since ReachSet is symmetric
        } else {
          // If Src is not a param/return, add it to ValueMap
          auto SrcIVal = InnerMapping.first;
          if (hasReadOnlyState(InnerMapping.second))
            ValueMap[SrcIVal.Val].FromRecords.push_back(
                ValueSummary::Record{*Dst, SrcIVal.DerefLevel});
          if (hasWriteOnlyState(InnerMapping.second))
            ValueMap[SrcIVal.Val].ToRecords.push_back(
                ValueSummary::Record{*Dst, SrcIVal.DerefLevel});
        }
      }
    }
  }

  for (const auto &Mapping : ValueMap) {
    for (const auto &FromRecord : Mapping.second.FromRecords) {
      for (const auto &ToRecord : Mapping.second.ToRecords) {
        auto ToLevel = ToRecord.DerefLevel;
        auto FromLevel = FromRecord.DerefLevel;
        // Same-level assignments should have already been processed by now
        if (ToLevel == FromLevel)
          continue;

        auto SrcIndex = FromRecord.IValue.Index;
        auto SrcLevel = FromRecord.IValue.DerefLevel;
        auto DstIndex = ToRecord.IValue.Index;
        auto DstLevel = ToRecord.IValue.DerefLevel;
```

(continues on next page)

(continued from previous page)

```
        if (ToLevel > FromLevel)
          SrcLevel += ToLevel - FromLevel;
        else
          DstLevel += FromLevel - ToLevel;

        ExtRelations.push_back(ExternalRelation{
            InterfaceValue{SrcIndex, SrcLevel},
            InterfaceValue{DstIndex, DstLevel}, UnknownOffset});
      }
    }
  }

  // Remove duplicates in ExtRelations
  std::sort(ExtRelations.begin(), ExtRelations.end());
  ExtRelations.erase(std::unique(ExtRelations.begin(), ExtRelations.end()),
                     ExtRelations.end());
}
```

`ExternalRelation` 是一个含有三个成员变量的结构体 `struct ExternalRelation { InterfaceValue From, To; int64_t Offset; };`，用于表示一个函数的参数和返回值之间的别名关系，使得在分析对该函数的调用点能够得到实参与函数返回值之间的别名关系。`populateExternalRelations` 处理了以下几种情况：

- 函数返回值就是某个参数的情况，`populateExternalRelations` 函数的第一部分处理了这种情况
- 函数的参数/返回值之间存在别名关系。并且存在这样的特殊情况：函数的参数 P 被赋值给一个变量 I，函数的返回值是对 I 的解引用 `*I`，实际上 `*P` 与 `*I` 应该是别名关系（即 {P, 1} 与 {I, 1} 互为别名），但是在 ReachSet 中 `*I` 并没有与其有别名关系的值。`populateExternalRelations` 函数的第二部分处理了这种情况

至此，FunctionInfo 构建完毕。

- genindex
- modindex
- search

- genindex
- modindex
- search

CHAPTER

FIVE

TRANSFORM

5.1 Aggressive Dead Code Elimination

5.1.1 Aggressive Dead Code Elimination

Aggressive Dead Code Elimination (以下简称 ADCE) 是一个 LLVM transform pass，用于消除冗余代码。该 ADCE 本质上是 liveness analysis 的应用，是一个 backward dataflow analysis。Aggressive 的意思是对于每一条指令，该 pass 假设该指令是 dead 除非该指令被证明是 live 的，在分析结束后所有的被认为是 dead 的指令都会被消除。

该 pass 的代码实现位于 `llvm-8.0.1.src/include/llvm/Transforms/Scalar/ADCE.h` 和 `llvm-8.0.1.src/lib/Transforms/Scalar/ADCE.cpp` 。

Implementation of ADCE

我们从 `ADCEPass` 的入口开始分析：

```
PreservedAnalyses ADCEPass::run(Function &F, FunctionAnalysisManager &FAM) {
  // ADCE does not need DominatorTree, but require DominatorTree here
  // to update analysis if it is already available.
  auto *DT = FAM.getCachedResult<DominatorTreeAnalysis>(F);
  auto &PDT = FAM.getResult<PostDominatorTreeAnalysis>(F);
  if (!AggressiveDeadCodeElimination(F, DT, PDT).performDeadCodeElimination())
    return PreservedAnalyses::all();

  PreservedAnalyses PA;
  PA.preserveSet<CFGAnalyses>();
  PA.preserve<GlobalsAA>();
  PA.preserve<DominatorTreeAnalysis>();
  PA.preserve<PostDominatorTreeAnalysis>();
  return PA;
}
```

可以看到 ADCE 实际是在类 `AggressiveDeadCodeElimination` 中实现的，而该类的构造函数有三个参数：待分析的函数，待分析函数的支配树 `DominatorTree` 和后支配树 `PostDominatorTree`，即，对于一个待分析的函数来说，执行 ADCE 分析需要 `DominatorTree` 和 `PostDominatorTree`（因为该分析是一个 backward 分析，实际上在分析时需要的是 `PostDominatorTree`，而不需要 `DominatorTree`，但是因为该 pass 删除了一些基本块，所以 `DominatorTree` 作为参数传进来是为了对 `DominatorTree` 进行更新）。

函数 `AggressiveDeadCodeElimination::performDeadCodeElimination()` 的定义很简单：

```
bool AggressiveDeadCodeElimination::performDeadCodeElimination() {
  initialize();
  markLiveInstructions();
  return removeDeadInstructions();
}
```

根据该函数的实现，可以看到该 ADCE pass 的流程很简单清晰：首先是初始化工作（选取哪些指令作为分析的起点），然后标记 live 的指令，最后消除那些 dead 指令。

后续将按顺序分析这三个函数的实现。在这之前，先看下在这三个函数中会用到一些的变量和函数的定义。

1. `AggressiveDeadCodeElimination` 的成员变量 `MapVector<BasicBlock *, BlockInfoType> BlockInfo`

   将基本块 `BasicBlock` 映射到存储该基本块相关信息的 `BlockInfoType`，`BlockInfoType` 的定义如下：

   ```
   struct BlockInfoType {
     /// True when this block contains a live instructions.
     bool Live = false;
     /// True when this block ends in an unconditional branch.
     bool UnconditionaUnlBranch = false;
     /// True when this block is known to have live PHI nodes.
     bool HasLivePhiNodes = false;
     /// Control dependence sources need to be live for this block.
     bool CFLive = false;
     /// Quick access to the LiveInfo for the terminator,
     /// holds the value &InstInfo[Terminator]
     InstInfoType *TerminatorLiveInfo = nullptr;
     /// Corresponding BasicBlock.
     BasicBlock *BB = nullptr;
     /// Cache of BB->getTerminator().
     Instruction *Terminator = nullptr;
     /// Post-order numbering of reverse control flow graph.
     unsigned PostOrder;
     bool terminatorIsLive() const { return TerminatorLiveInfo->Live; }
   };
   ```

注释很清晰：Live 用于说明该基本块中是否存在 live 的指令 (instructions)；UnconditionaUnlBranch 用于说明该基本块是否以一个无条件分支指令 (unconditional branch instruction) 结束；HasLivePhiNodes 用于说明该基本块是否存在 live 的 PHINode；CFLive 用于说明该基本块所控制依赖的基本块应该是 live 的；TerminatorLiveInfo 指向该基本块的 Terminator 指令的相关信息 InstInfoType（见成员变量 DenseMap<Instruction *, InstInfoType> InstInfo）；BB 就是该 BlockInfoType 所描述的基本块；Terminator 就是该基本块的 Terminator 指令，存储在 BlockInfoType 中起到一个 cache 的作用；PostOrder 是该基本块在 reverse control flow graph 中的 post-order 编号。

2. AggressiveDeadCodeElimination 的成员变量 DenseMap<Instruction *, InstInfoType> InstInfo

   将指令 Instruction 映射到存储该指令相关信息的 InstInfoType，InstInfoType 的定义如下：

```
struct InstInfoType {
  /// True if the associated instruction is live.
  bool Live = false;
  /// Quick access to information for block containing associated Instruction.
  struct BlockInfoType *Block = nullptr;
};
```

   成员变量 Live 用于说明 InstInfoType 对应的指令是否为 live；成员变量 Block 指向的就是该指令存在的基本块所对应的 BlockInfoType 。

3. AggressiveDeadCodeElimination 的成员变量 SmallPtrSet<BasicBlock *, 16> BlocksWithDeadTerminators

   该成员变量存储那些基本块的 terminator 指令不是 live 的基本块。

4. AggressiveDeadCodeElimination 的成员变量 SmallPtrSet<BasicBlock *, 16> NewLiveBlocks

   该成员变量的注释：The set of blocks which we have determined whose control dependence sources must be live and which have not had those dependences analyzed.

   就是说，如果某个基本块所控制依赖的那些基本块应该是 live 的，但是这控制依赖还没有分析，就将该基本块暂时存储在 NewLiveBlocks 中。

5. AggressiveDeadCodeElimination 的成员变量 SmallVector<Instruction *, 128> Worklist

   该成员变量用于存储已知是 live 的 Instruction，需要注意的是在函数 initialize()，markLiveInstructions() 执行完后，该变量为空，在函数 removeDeadInstructions() 中复用了该成员变量存储用于存储需要被消除的 dead instructions 。

6. 函数 static bool isUnconditionalBranch(Instruction *Term);

   该函数很简单，就是判断给定的 Instruction 是否是一个无条件分支指令，实现如下：

```
static bool isUnconditionalBranch(Instruction *Term) {
  auto *BR = dyn_cast<BranchInst>(Term);
  return BR && BR->isUnconditional();
}
```

首先看参数 `Instruction *Term` 是否为 `BranchInst` 类型，如果该参数确实是一个 `BranchInst`，并且是 `Unconditional` 的 `BranchInst`，则该函数返回 true 。

7. 类 `AggressiveDeadCodeElimination` 的成员函数 `isAlwaysLive(Instruction &I)`

   从函数名就能看出该函数对于分析来说很关键，因为该函数确定了什么样的指令是 always live 的。

```
bool AggressiveDeadCodeElimination::isAlwaysLive(Instruction &I) {
  // TODO -- use llvm::isInstructionTriviallyDead
  if (I.isEHPad() || I.mayHaveSideEffects()) {
    // Skip any value profile instrumentation calls if they are
    // instrumenting constants.
    if (isInstrumentsConstant(I))
      return false;
    return true;
  }
  if (!I.isTerminator())
    return false;
  if (RemoveControlFlowFlag && (isa<BranchInst>(I) || isa<SwitchInst>(I)))
    return false;
  return true;
}
```

   可以看到可能有副作用的指令 (如 `StoreInst`) 是 always live 的。变量 `RemoveControlFlowFlag` 为 true (默认为 true) 时，除了 `BranchInst` 和 `SwitchInst` 以外的 terminator 指令都 always live 的；如果 `RemoveControlFlowFlag` 为 false 的话，那么所有的 terminator 指令都 always live 的。

8. 类 `AggressiveDeadCodeElimination` 的成员函数 `void markLive(Instruction *I)`, `void markLive(BasicBlock *BB)` 和 `void markLive(BlockInfoType &BBInfo)`。

   这里将这三个重载的函数一同说明了。

   -
```
void AggressiveDeadCodeElimination::markLive(Instruction *I) {
  auto &Info = InstInfo[I];
  if (Info.Live)
    return;

  LLVM_DEBUG(dbgs() << "mark live: "; I->dump());
  Info.Live = true;
  Worklist.push_back(I);
```

(continues on next page)

(continued from previous page)

```
  // Collect the live debug info scopes attached to this instruction.
  if (const DILocation *DL = I->getDebugLoc())
    collectLiveScopes(*DL);

  // Mark the containing block live
  auto &BBInfo = *Info.Block;
  if (BBInfo.Terminator == I) {
    BlocksWithDeadTerminators.erase(BBInfo.BB);
    // For live terminators, mark destination blocks
    // live to preserve this control flow edges.
    if (!BBInfo.UnconditionalBranch)
      for (auto *BB : successors(I->getParent()))
        markLive(BB);
  }
  markLive(BBInfo);
}
```

首先就是将 `InstInfo[I].Live` 设置为 true，然后将该指令存储进成员变量 `Worklist` 中，没什么好说的。然后就是看该指令是否为其所在基本块的 terminator 指令，如果是的话，就更新 `BlocksWithDeadTerminators` (如果该指令在 ``BlocksWithDeadTerminators`` 中，就从中删除该指令)。如果该指令是其所在基本块的 terminator 指令，并且是一个无条件的 `BranchInst`，就调用 `void markLive(BasicBlock *BB)` 将其所在基本块的后继基本块都设置为 live，最后调用 `void markLive(BlockInfoType &BBInfo)` 将该指令所在的基本块设置为 live。

- `void markLive(BasicBlock *BB)` 的实现就是对 `void markLive(BlockInfoType &BBInfo)` 的一层封装。

```
void markLive(BasicBlock *BB) { markLive(BlockInfo[BB]); }
```

- `void markLive(BlockInfoType &BBInfo)` 的实现如下：

```
void AggressiveDeadCodeElimination::markLive(BlockInfoType &BBInfo) {
  if (BBInfo.Live)
    return;
  LLVM_DEBUG(dbgs() << "mark block live: " << BBInfo.BB->getName() << '\n');
  BBInfo.Live = true;
  if (!BBInfo.CFLive) {
    BBInfo.CFLive = true;
    NewLiveBlocks.insert(BBInfo.BB);
  }

  // Mark unconditional branches at the end of live
  // blocks as live since there is no work to do for them later
  if (BBInfo.UnconditionalBranch)
```

(continues on next page)

(continued from previous page)

```
    markLive(BBInfo.Terminator);
}
```

首先将 `BBInfo.Live` 设置为 true 。如果 `BBInfo.CFLive` 为 false，就将其设置为 true，并且将当前基本块存储进 `NewLiveBlocks` 中，即如果将当前基本块设置为 live 的，那么该基本块所控制依赖的基本块们也应该是 live 的。最后如果该基本块的 terminator 指令是一个无条件的 `BranchInst`，就对该基本块的 terminator 指令调用函数 `void markLive(Instruction *I)` 。

initialize()

函数 `initialize()` 用于确定选取哪些指令作为分析的起点。因函数的函数体很长，我们逐部分地分析：

第一部分代码如下：

```
void AggressiveDeadCodeElimination::initialize() {
  auto NumBlocks = F.size();

  // We will have an entry in the map for each block so we grow the
  // structure to twice that size to keep the load factor low in the hash table.
  BlockInfo.reserve(NumBlocks);
  size_t NumInsts = 0;

  // Iterate over blocks and initialize BlockInfoVec entries, count
  // instructions to size the InstInfo hash table.
  for (auto &BB : F) {
    NumInsts += BB.size();
    auto &Info = BlockInfo[&BB];
    Info.BB = &BB;
    Info.Terminator = BB.getTerminator();
    Info.UnconditionalBranch = isUnconditionalBranch(Info.Terminator);
  }

  // Initialize instruction map and set pointers to block info.
  InstInfo.reserve(NumInsts);
  for (auto &BBInfo : BlockInfo)
    for (Instruction &I : *BBInfo.second.BB)
      InstInfo[&I].Block = &BBInfo.second;

  // Since BlockInfoVec holds pointers into InstInfo and vice-versa, we may not
  // add any more elements to either after this point.
  for (auto &BBInfo : BlockInfo)
    BBInfo.second.TerminatorLiveInfo = &InstInfo[BBInfo.second.Terminator];
```

(continues on next page)

(continued from previous page)

```
  // Collect the set of "root" instructions that are known live.
  for (Instruction &I : instructions(F))
    if (isAlwaysLive(I))
      markLive(&I);
```

上面这一部分代码就是对 AggressiveDeadCodeElimination 的成员变量 BlockInfo 和 InstInfo 的初始化。

值得注意的地方主要是这三个函数调用：Info.UnconditionalBranch = isUnconditionalBranch(Info.Terminator);，isAlwaysLive(I) 和 markLive(&I) 。这三个函数的实现已经在前面说明过了，并且函数命名很清晰，很直观的知道函数的作用。

第二部分代码如下：

```
if (!RemoveControlFlowFlag)
  return;

if (!RemoveLoops) {
  // This stores state for the depth-first iterator. In addition
  // to recording which nodes have been visited we also record whether
  // a node is currently on the "stack" of active ancestors of the current
  // node.
  using StatusMap = DenseMap<BasicBlock *, bool>;

  class DFState : public StatusMap {
  public:
    std::pair<StatusMap::iterator, bool> insert(BasicBlock *BB) {
      return StatusMap::insert(std::make_pair(BB, true));
    }

    // Invoked after we have visited all children of a node.
    void completed(BasicBlock *BB) { (*this)[BB] = false; }

    // Return true if \p BB is currently on the active stack
    // of ancestors.
    bool onStack(BasicBlock *BB) {
      auto Iter = find(BB);
      return Iter != end() && Iter->second;
    }
  } State;

  State.reserve(F.size());
```

(continues on next page)

(continued from previous page)

```
  // Iterate over blocks in depth-first pre-order and
  // treat all edges to a block already seen as loop back edges
  // and mark the branch live it if there is a back edge.
  for (auto *BB: depth_first_ext(&F.getEntryBlock(), State)) {
    Instruction *Term = BB->getTerminator();
    if (isLive(Term))
      continue;

    for (auto *Succ : successors(BB))
      if (State.onStack(Succ)) {
        // back edge....
        markLive(Term);
        break;
      }
  }
}
```

如果变量 `RemoveControlFlowFlag` 为 false (默认为 true)，则直接 return；如果 `RemoveLoops` 为 false (默认为 false) 的话，那么从函数 F 的入口基本块开始 depth-first pre-order 顺序遍历函数的基本块，如果存在基本块的 terminator 指令不是 live，并且该基本块的后继基本块已经被遍历过了，我们认为这样的边是一条回边，对将基本块的 terminator 指令调用函数 `void markLive(Instruction *I)`。

第三部分代码如下：

```
// Mark blocks live if there is no path from the block to a
// return of the function.
// We do this by seeing which of the postdomtree root children exit the
// program, and for all others, mark the subtree live.
for (auto &PDTChild : children<DomTreeNode *>(PDT.getRootNode())) {
  auto *BB = PDTChild->getBlock();
  auto &Info = BlockInfo[BB];
  // Real function return
  if (isa<ReturnInst>(Info.Terminator)) {
    LLVM_DEBUG(dbgs() << "post-dom root child is a return: " << BB->getName()
                      << '\n';);
    continue;
  }

  // This child is something else, like an infinite loop.
  for (auto DFNode : depth_first(PDTChild))
    markLive(BlockInfo[DFNode->getBlock()].Terminator);
}
```

如果存在某些基本块，这些基本块没有路径到达函数的 return 指令，那么就将这些基本块设置为 live 。具体

实现时是这样的，首先看后支配树的根节点，如果根节点的子节点基本块的 terminator 指令是 `ReturnInst` 则跳过 (后支配树上 `ReturnInst` 所在的基本块的子孙节点一定能到达该 `ReturnInst` 所在的基本块)，对于根节点的子节点基本块，如果其 terminator 指令不是 `ReturnInst`，则以此基本块为起点 depth-first 遍历，对所有遍历到的基本块，对其 terminator 指令调用函数 `void markLive(Instruction *I)`。

第四部分代码：

```
  // Treat the entry block as always live
  auto *BB = &F.getEntryBlock();
  auto &EntryInfo = BlockInfo[BB];
  EntryInfo.Live = true;
  if (EntryInfo.UnconditionalBranch)
    markLive(EntryInfo.Terminator);

  // Build initial collection of blocks with dead terminators
  for (auto &BBInfo : BlockInfo)
    if (!BBInfo.second.terminatorIsLive())
      BlocksWithDeadTerminators.insert(BBInfo.second.BB);
}
```

函数 `initialize()` 的最后一部分代码，将函数的入口基本块设置为 live，如果入口基本块的 terminator 指令是无条件的 `BranchInst`，则对此 `BranchInst` 调用函数 `void markLive(Instruction *I)`，最后更新成员变量 `BlocksWithDeadTerminators` 。

---

markLiveInstructions()

`markLiveInstructions()` 就是标记所有 live 的 instructions 。

```
void AggressiveDeadCodeElimination::markLiveInstructions() {
  // Propagate liveness backwards to operands.
  do {
    // Worklist holds newly discovered live instructions
    // where we need to mark the inputs as live.
    while (!Worklist.empty()) {
      Instruction *LiveInst = Worklist.pop_back_val();
      LLVM_DEBUG(dbgs() << "work live: "; LiveInst->dump(););

      for (Use &OI : LiveInst->operands())
        if (Instruction *Inst = dyn_cast<Instruction>(OI))
          markLive(Inst);

      if (auto *PN = dyn_cast<PHINode>(LiveInst))
```

(continues on next page)

(continued from previous page)

```
        markPhiLive(PN);
    }

    // After data flow liveness has been identified, examine which branch
    // decisions are required to determine live instructions are executed.
    markLiveBranchesFromControlDependences();

  } while (!Worklist.empty());
}
```

在函数 `initialize()` 执行后，`Worklist` 中存储了被标记为 live 的 instructions (每次调用函数 `void markLive(Instruction *I)` 时，该函数将指令 `I` 存储进 `Worklist`)。在函数 `markLiveInstructions()` 中通过 worklist 算法不断标记新的 live instructions，沿着 use-def 方向，如果一个指令被标记为 live，那么对其操作数 def 的 instruction 也被标记为 live。如果 `Worlist` 中某条指令是 `PHINode`，则调用函数 `void markPhiLive(PHINode *PN)` 进行特殊处理。

```
void AggressiveDeadCodeElimination::markPhiLive(PHINode *PN) {
  auto &Info = BlockInfo[PN->getParent()];
  // Only need to check this once per block.
  if (Info.HasLivePhiNodes)
    return;
  Info.HasLivePhiNodes = true;

  // If a predecessor block is not live, mark it as control-flow live
  // which will trigger marking live branches upon which
  // that block is control dependent.
  for (auto *PredBB : predecessors(Info.BB)) {
    auto &Info = BlockInfo[PredBB];
    if (!Info.CFLive) {
      Info.CFLive = true;
      NewLiveBlocks.insert(PredBB);
    }
  }
}
```

首先将 `PHINode` 所在基本块对应的 `BlockInfoType` 的 `HasLivePhiNodes` 域设置为 true。如果该 `PHINode` 所在基本块的的前驱基本块对应的 `BlockInfoType` 的 `CFLive` 域为 false，则将该域设置为 true，并将此前驱基本块放入 `NewLiveBlocks` 中，即该前驱基本块所控制依赖的基本块也应该是 live 的。`markLiveBranchesFromControlDependences()` 函数就是用于将 live 基本块所控制依赖的基本块也标记为 live 的。`markLiveBranchesFromControlDependences()` 函数的实现如下：

```
void AggressiveDeadCodeElimination::markLiveBranchesFromControlDependences() {
  if (BlocksWithDeadTerminators.empty())
```

(continues on next page)

(continued from previous page)

```
    return;

  // The dominance frontier of a live block X in the reverse
  // control graph is the set of blocks upon which X is control
  // dependent. The following sequence computes the set of blocks
  // which currently have dead terminators that are control
  // dependence sources of a block which is in NewLiveBlocks.

  SmallVector<BasicBlock *, 32> IDFBlocks;
  ReverseIDFCalculator IDFs(PDT);
  IDFs.setDefiningBlocks(NewLiveBlocks);
  IDFs.setLiveInBlocks(BlocksWithDeadTerminators);
  IDFs.calculate(IDFBlocks);
  NewLiveBlocks.clear();

  // Dead terminators which control live blocks are now marked live.
  for (auto *BB : IDFBlocks) {
    LLVM_DEBUG(dbgs() << "live control in: " << BB->getName() << '\n');
    markLive(BB->getTerminator());
  }
}
```

首先计算 `NewLiveBlocks` 的 Post Iterated Dominance Frontier (即 `NewLiveBlocks` 中的基本块所控制依赖的基本块)，然后对得到的基本块的 terminator 指令调用函数 `void markLive(Instruction *I)`。

---

removeDeadInstructions()

`removeDeadInstructions()` 函数将所有没有被标记为 live 的 instructions 消除。

```
bool AggressiveDeadCodeElimination::removeDeadInstructions() {
  // Updates control and dataflow around dead blocks
  updateDeadRegions();

  // The inverse of the live set is the dead set.  These are those instructions
  // that have no side effects and do not influence the control flow or return
  // value of the function, and may therefore be deleted safely.
  // NOTE: We reuse the Worklist vector here for memory efficiency.
  for (Instruction &I : instructions(F)) {
    // Check if the instruction is alive.
    if (isLive(&I))
      continue;
```

(continues on next page)

(continued from previous page)

```
    if (auto *DII = dyn_cast<DbgInfoIntrinsic>(&I)) {
      // Check if the scope of this variable location is alive.
      if (AliveScopes.count(DII->getDebugLoc()->getScope()))
        continue;

      // Fallthrough and drop the intrinsic.
    }

    // Prepare to delete.
    Worklist.push_back(&I);
    I.dropAllReferences();
  }

  for (Instruction *&I : Worklist) {
    ++NumRemoved;
    I->eraseFromParent();
  }

  return !Worklist.empty();
}
```

这里重用了成员变量 `Worklist`，因为执行至该函数时，`Worklist` 已经是空的了，这里复用它来存储需要被消除的指令。该函数中值得注意的是对 `updateDeadRegions()` 函数的调用，因为我们删除了一些基本块，所以需要对 `DominatorTree` 进行更新，这就是在 `updateDeadRegions()` 中实现的，这里不再详细分析了。

Summary

大体上来说，该分析的流程如下：

1. 该算法的起点是所有 terminator 指令 (例如 `ReturnInst`), may side effecting 指令 (例如 `StoreInst`)，这些认为是 live 的
2. 然后，利用 SSA 形式的 use-def 信息，从上述起点出发迭代，把所有能通过 use-def 链到达的指令都标记为 live
3. 最后，没有被标记为 live 的指令就是 dead，遍历一次所有指令，把没有被标记为 live 的指令删除，DCE 就完成了

该分析可以看作是一个使用了只有 2 个元素的 lattice (bottom 是 live，top 是 dead) 的 backward 数据流分析。

- genindex
- modindex
- search

5.2 Called Value Propagation

5.2.1 SparsePropagation

Introduction

数据流分析是一种用于在计算在某个程序点的程序状态（数据流值）的技术。基于数据流分析的典型例子有常量传播、到达定值等。

根据 R 大在知乎的回答（见参考链接），因为 SSA 形式贯穿于 LLVM IR，所以在 LLVM 中都针对 SSA Value 的数据流分析都是用 sparse 方式去做的，而不像传统 IR 那样迭代遍历每条指令去传播信息直到到达不同点（需要注意的是，在 LLVM IR 中“memory”不是 SSA value，所以对“memory”分析的话，就无法用 sparse 的方式了；但是 LLVM 有一个 memory SSA 的项目，我对 memory SSA 没有了解，后面有时间写篇文章填坑）。

- dense 分析：要用个容器携带所有变量的信息去遍历所有指令，即便某条指令不关心的变量信息也会携带过去
- sparse 分析：变量的信息直接在 def 与 use 之间传播，中间不需要遍历其他不相关的指令

在 LLVM 中提供了一个用于实现 sparse analysis 的 infrastructure，位于 `llvm-7.0.0.src/include/llvm/Analysis/SparsePropagation.h` 。

在标准的数据流分析框架中，应该有如下的组成部分:

- D: 数据流分析方向，forward 还是 backward，即是前向的数据流分析还是后向的数据流分析
- V, ^ : 即数据流值和交汇运算。(V, ^) 需要满足半格的定义，即 (V, ^) 是一个半格
- F: V 到 V 的传递函数族。

基于 SparsePropagation 实例化一个分析时需要提供 LatticeKey, LatticeVal 和 LatticeFunction。其中 LatticeVal 对应数据流值，LatticeKey 用于将 LLVM Value 映射到 LatticeVal，而 LatticeFunction 对应传递函数。好像基于 SparsePropagation 实例化一个分析时，分析方向只能是前向的。

AbstractLatticeFunction

首先，需要继承 AbstractLatticeFunction 类来实现一个 LatticeFunction。

```
template <class LatticeKey, class LatticeVal> class AbstractLatticeFunction {
private:
  LatticeVal UndefVal, OverdefinedVal, UntrackedVal;

public:
  AbstractLatticeFunction(LatticeVal undefVal, LatticeVal overdefinedVal,
                          LatticeVal untrackedVal) {
    UndefVal = undefVal;
    OverdefinedVal = overdefinedVal;
```

(continues on next page)

(continued from previous page)

```
  UntrackedVal = untrackedVal;
}

virtual ~AbstractLatticeFunction() = default;

LatticeVal getUndefVal()       const { return UndefVal; }
LatticeVal getOverdefinedVal() const { return OverdefinedVal; }
LatticeVal getUntrackedVal()   const { return UntrackedVal; }

/// IsUntrackedValue - If the specified LatticeKey is obviously uninteresting
/// to the analysis (i.e., it would always return UntrackedVal), this
/// function can return true to avoid pointless work.
virtual bool IsUntrackedValue(LatticeKey Key) { return false; }

/// ComputeLatticeVal - Compute and return a LatticeVal corresponding to the
/// given LatticeKey.
virtual LatticeVal ComputeLatticeVal(LatticeKey Key) {
  return getOverdefinedVal();
}

/// IsSpecialCasedPHI - Given a PHI node, determine whether this PHI node is
/// one that the we want to handle through ComputeInstructionState.
virtual bool IsSpecialCasedPHI(PHINode *PN) { return false; }

/// MergeValues - Compute and return the merge of the two specified lattice
/// values.  Merging should only move one direction down the lattice to
/// guarantee convergence (toward overdefined).
virtual LatticeVal MergeValues(LatticeVal X, LatticeVal Y) {
  return getOverdefinedVal(); // always safe, never useful.
}

/// ComputeInstructionState - Compute the LatticeKeys that change as a result
/// of executing instruction \p I. Their associated LatticeVals are store in
/// \p ChangedValues.
virtual void
ComputeInstructionState(Instruction &I,
                        DenseMap<LatticeKey, LatticeVal> &ChangedValues,
                        SparseSolver<LatticeKey, LatticeVal> &SS) = 0;

/// PrintLatticeVal - Render the given LatticeVal to the specified stream.
virtual void PrintLatticeVal(LatticeVal LV, raw_ostream &OS);

/// PrintLatticeKey - Render the given LatticeKey to the specified stream.
```

(continues on next page)

(continued from previous page)

```
  virtual void PrintLatticeKey(LatticeKey Key, raw_ostream &OS);

  /// GetValueFromLatticeVal - If the given LatticeVal is representable as an
  /// LLVM value, return it; otherwise, return nullptr. If a type is given, the
  /// returned value must have the same type. This function is used by the
  /// generic solver in attempting to resolve branch and switch conditions.
  virtual Value *GetValueFromLatticeVal(LatticeVal LV, Type *Ty = nullptr) {
    return nullptr;
  }
};
```

核心函数是 `ComputeInstructionState` 和 `MergeValues`。`ComputeInstructionState` 对应数据流分析中的传递函数，当执行完一条 Instruction 后，应该怎么样更新数据流值。`MergeValues` 对应数据流分析中的交汇运算，即怎么样处理数据流值的“合并”。

SparseSolver

除了需要继承 AbstractLatticeFunction 类来实现一个 LatticeFunction。还要创建一个 SparseSolver 对象来进行求解。

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
class SparseSolver {

  /// LatticeFunc - This is the object that knows the lattice and how to
  /// compute transfer functions.
  AbstractLatticeFunction<LatticeKey, LatticeVal> *LatticeFunc;

  /// ValueState - Holds the LatticeVals associated with LatticeKeys.
  DenseMap<LatticeKey, LatticeVal> ValueState;

  /// BBExecutable - Holds the basic blocks that are executable.
  SmallPtrSet<BasicBlock *, 16> BBExecutable;

  /// ValueWorkList - Holds values that should be processed.
  SmallVector<Value *, 64> ValueWorkList;

  /// BBWorkList - Holds basic blocks that should be processed.
  SmallVector<BasicBlock *, 64> BBWorkList;

  using Edge = std::pair<BasicBlock *, BasicBlock *>;

  /// KnownFeasibleEdges - Entries in this set are edges which have already had
  /// PHI nodes retriggered.
```

(continues on next page)

(continued from previous page)

```
  std::set<Edge> KnownFeasibleEdges;

public:
  explicit SparseSolver(
      AbstractLatticeFunction<LatticeKey, LatticeVal> *Lattice)
      : LatticeFunc(Lattice) {}
  SparseSolver(const SparseSolver &) = delete;
  SparseSolver &operator=(const SparseSolver &) = delete;

  /// Solve - Solve for constants and executable blocks.
  void Solve();

  void Print(raw_ostream &OS) const;

  /// getExistingValueState - Return the LatticeVal object corresponding to the
  /// given value from the ValueState map. If the value is not in the map,
  /// UntrackedVal is returned, unlike the getValueState method.
  LatticeVal getExistingValueState(LatticeKey Key) const {
    auto I = ValueState.find(Key);
    return I != ValueState.end() ? I->second : LatticeFunc->getUntrackedVal();
  }

  /// getValueState - Return the LatticeVal object corresponding to the given
  /// value from the ValueState map. If the value is not in the map, its state
  /// is initialized.
  LatticeVal getValueState(LatticeKey Key);

  /// isEdgeFeasible - Return true if the control flow edge from the 'From'
  /// basic block to the 'To' basic block is currently feasible.  If
  /// AggressiveUndef is true, then this treats values with unknown lattice
  /// values as undefined.  This is generally only useful when solving the
  /// lattice, not when querying it.
  bool isEdgeFeasible(BasicBlock *From, BasicBlock *To,
                      bool AggressiveUndef = false);

  /// isBlockExecutable - Return true if there are any known feasible
  /// edges into the basic block.  This is generally only useful when
  /// querying the lattice.
  bool isBlockExecutable(BasicBlock *BB) const {
    return BBExecutable.count(BB);
  }

  /// MarkBlockExecutable - This method can be used by clients to mark all of
```

(continues on next page)

(continued from previous page)

```
  /// the blocks that are known to be intrinsically live in the processed unit.
  void MarkBlockExecutable(BasicBlock *BB);

private:
  /// UpdateState - When the state of some LatticeKey is potentially updated to
  /// the given LatticeVal, this function notices and adds the LLVM value
  /// corresponding the key to the work list, if needed.
  void UpdateState(LatticeKey Key, LatticeVal LV);

  /// markEdgeExecutable - Mark a basic block as executable, adding it to the BB
  /// work list if it is not already executable.
  void markEdgeExecutable(BasicBlock *Source, BasicBlock *Dest);

  /// getFeasibleSuccessors - Return a vector of booleans to indicate which
  /// successors are reachable from a given terminator instruction.
  void getFeasibleSuccessors(TerminatorInst &TI, SmallVectorImpl<bool> &Succs,
                             bool AggressiveUndef);

  void visitInst(Instruction &I);
  void visitPHINode(PHINode &I);
  void visitTerminatorInst(TerminatorInst &TI);
};
```

SparseSolver 通过 `Solve()` 函数求解数据流方程，`Solve()` 函数实现了 worklist 算法：

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
void SparseSolver<LatticeKey, LatticeVal, KeyInfo>::Solve() {
  // Process the work lists until they are empty!
  while (!BBWorkList.empty() || !ValueWorkList.empty()) {
    // Process the value work list.
    while (!ValueWorkList.empty()) {
      Value *V = ValueWorkList.back();
      ValueWorkList.pop_back();

      LLVM_DEBUG(dbgs() << "\nPopped off V-WL: " << *V << "\n");

      // "V" got into the work list because it made a transition. See if any
      // users are both live and in need of updating.
      for (User *U : V->users())
        if (Instruction *Inst = dyn_cast<Instruction>(U))
          if (BBExecutable.count(Inst->getParent())) // Inst is executable?
            visitInst(*Inst);
    }
```

(continues on next page)

(continued from previous page)

```
    // Process the basic block work list.
    while (!BBWorkList.empty()) {
      BasicBlock *BB = BBWorkList.back();
      BBWorkList.pop_back();

      LLVM_DEBUG(dbgs() << "\nPopped off BBWL: " << *BB);

      // Notify all instructions in this basic block that they are newly
      // executable.
      for (Instruction &I : *BB)
        visitInst(I);
    }
  }
}
```

在调用 `Solve()` 函数之前通过 `MarkBlockExecutable()` 设置 BBWorkList 和 BBExecutable，因此初始状态下 ValueWorkList 为空，BBWorkList 不为空。然后会执行到 `while (!BBWorkList.empty())` 这个循环中，对 BBWorkList 中的每一个 `BasicBlock` 中的每一条 `Instruction` 调用 `visitInst()` 函数。

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
void SparseSolver<LatticeKey, LatticeVal, KeyInfo>::visitInst(Instruction &I) {
  // PHIs are handled by the propagation logic, they are never passed into the
  // transfer functions.
  if (PHINode *PN = dyn_cast<PHINode>(&I))
    return visitPHINode(*PN);

  // Otherwise, ask the transfer function what the result is.  If this is
  // something that we care about, remember it.
  DenseMap<LatticeKey, LatticeVal> ChangedValues;
  LatticeFunc->ComputeInstructionState(I, ChangedValues, *this);
  for (auto &ChangedValue : ChangedValues)
    if (ChangedValue.second != LatticeFunc->getUntrackedVal())
      UpdateState(ChangedValue.first, ChangedValue.second);

  if (TerminatorInst *TI = dyn_cast<TerminatorInst>(&I))
    visitTerminatorInst(*TI);,
}
```

值得注意的是，在对 `TerminatorInst` 处理时会调用 `visitTerminatorInst()` 函数，该函数将 `TerminatorInst` 所在基本块的可达后继基本块加入到 BBWorkList 和 BBExecutable 中。

SparseSolver 通过 `UpdateState()` 函数对数据流值进行更新：

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
void SparseSolver<LatticeKey, LatticeVal, KeyInfo>::UpdateState(LatticeKey Key,
                                                                LatticeVal LV) {
  auto I = ValueState.find(Key);
  if (I != ValueState.end() && I->second == LV)
    return; // No change.

  // Update the state of the given LatticeKey and add its corresponding LLVM
  // value to the work list.
  ValueState[Key] = std::move(LV);
  if (Value *V = KeyInfo::getValueFromLatticeKey(Key))
    ValueWorkList.push_back(V);
}
```

如果数据流值被更新了，那么会将该数据流值对应的 LLVM Value 加入到 ValueWorkList 中，所以在 `Solve()` 函数的 `while (!BBWorkList.empty() || !ValueWorkList.empty())` 循环的下一轮迭代时，会进入到 `while (!ValueWorkList.empty())` 这个循环中对每一个 Value 的每一次使用调用 `visitInst()` 函数进行处理。

`Solve()` 函数就这样不断地进行迭代直至达到不动点位置。

Example

CalledValuePropagation 是一个 transform pass，基于 SparsePropagation 实现了对间接调用点 (indirect call sites) 的被调函数的可能取值进行分析。

Reference

https://www.zhihu.com/question/41959902/answer/93087273

5.2.2 CalledValuePropagation

Introduction

CalledValuePropagation 是一个 transform pass，对于一些间接调用点 (indirect call sites)，在调用点处添加了名为!callee 的 metadata 来表示被调函数 (callees) 的可能取值。

CalledValuePropagation 在 LLVM 6.x 版本中被引入，Differential Revision: https://reviews.llvm.org/D37355

本文中对该 pass 的代码分析基于 LLVM 7.0.0 版本，其实现代码位于 `llvm-7.0.0.src/include/llvm/Transforms/IPO/CalledValuePropagation.h` 和 `llvm-7.0.0.src/lib/Transforms/IPO/CalledValuePropagation.cpp` (IPO 是 Inter-Procedural Optimization 的简写)。CalledValuePropagation 基于 SparsePropagation 实现。(`llvm-7.0.0.src/include/llvm/Analysis/SparsePropagation.h`)

可以通过 opt 命令来调用该 pass 来获取间接调用点的被调函数的可能取值。例如

```
opt -S -called-value-propagation simple-arguments.ll -o simple-arguments.opt.ll
```

如果打开生成的 LLVM IR 文件 simple-arguments.opt.ll 会看到有如下的一条指令：

```
%tmp3 = call i1 %cmp(i64* %tmp1, i64* %tmp2), !callees !0
```

!callees !0 就是用于表示被调函数可能取值的 metadata，在文件 simple-arguments.opt.ll 的最后会看到如下的内容：

```
!0 = !{i1 (i64*, i64*)* @ugt, i1 (i64*, i64*)* @ule}
```

即 `%tmp3 = call i1 %cmp(i64* %tmp1, i64* %tmp2)` 中可能的被调函数是 `ugt` 和 `ule` 。

CalledValuePropagationPass

`CalledValuePropagationPass` 类的定义如下：

```
class CalledValuePropagationPass : public PassInfoMixin<CalledValuePropagationPass>
{
public:
    PreservedAnalyses run(Module &M, ModuleAnalysisManager &);
};

PreservedAnalyses CalledValuePropagationPass::run(Module &M, ModuleAnalysisManager &)
{
    runCVP(M);
    return PreservedAnalyses::all();
}
```

可以看到核心功能在函数 `runCVP()` 中实现：

```
static bool runCVP(Module &M)
{
    // Our custom lattice function and generic sparse propagation solver.
    CVPLatticeFunc Lattice;
    SparseSolver<CVPLatticeKey, CVPLatticeVal> Solver(&Lattice);

    // For each function in the module, if we can't track its arguments, let the
    // generic solver assume it is executable.
    for (Function &F : M)
        if (!F.isDeclaration() && !canTrackArgumentsInterprocedurally(&F))
            Solver.MarkBlockExecutable(&F.front());
```

(continues on next page)

(continued from previous page)

```
    // Solver our custom lattice. In doing so, we will also build a set of
    // indirect call sites.
    Solver.Solve();

    // Attach metadata to the indirect call sites that were collected indicating
    // the set of functions they can possibly target.
    bool Changed = false;
    MDBuilder MDB(M.getContext());
    for (Instruction *C : Lattice.getIndirectCalls())
    {
        CallSite CS(C);
        auto RegI = CVPLatticeKey(CS.getCalledValue(), IPOGrouping::Register);
        CVPLatticeVal LV = Solver.getExistingValueState(RegI);
        if (!LV.isFunctionSet() || LV.getFunctions().empty())
            continue;
        MDNode *Callees = MDB.createCallees(LV.getFunctions());
        C->setMetadata(LLVMContext::MD_callees, Callees);
        Changed = true;
    }

    return Changed;
}
```

这段代码的逻辑很清晰，首先创建了 LatticeFunction 和 SparseSolver，如果函数不是一个函数声明并且该函数的参数不能过程间地追踪，那么将该函数的入口基本块添加至集合 BBExecutable 和 BBWorkList 中。接着调用 `Solver.Solve();` 进行求解，在求解过程中对间接调用点的被调函数的可能取值进行了收集，最后将可能的被调函数以 call sites 的 metadata 的形式写入 LLVM IR。

所以 `CVPLatticeKey`, `CVPLatticeVal`, `CVPLatticeFunc` 都是怎么定义的?

CVPLatticeKey

`CVPLatticeKey` 是 LatticeFunction 的 key type。

```
enum class IPOGrouping { Register, Return, Memory };
using CVPLatticeKey = PointerIntPair<Value *, 2, IPOGrouping>;
```

为了能够进行过程间分析，将 LLVM Values 分成了三类：Register, Return 和 Memory。Register 用于表示 SSA registers，Return 用于表示函数的返回值，Memory 用于表示 in-memory values (`StoreInst` 和 `LoadInst` 的 PointerOperand 是 `GlobalVariable` 时，会将该 PointerOperand 设置为 Memory 类型的 `CVPLatticeKey`)。

`CVPLatticeKey` 是由 LLVM `Value*` 和 `IPOGrouping` 组成的 `PointerIntPair`。

CVPLatticeVal

`CVPLatticeKey` 是 LatticeFunction 的 value type。

```
class CVPLatticeVal
{
public:
    enum CVPLatticeStateTy
    {
        Undefined,
        FunctionSet,
        Overdefined,
        Untracked
    };
    // 省略了 CVPLatticeVal 中的函数
private:
    CVPLatticeStateTy LatticeState;
    std::vector<Function *> Functions;
};
```

`CVPLatticeVal` 有两个成员变量：LatticeState，Functions。成员变量 Functions 用来存储 call sites 的被调函数的可能取值，成员变量 LatticeState 有四种可能取值：Undefined，FunctionSet，Overdefined，Untracked，当 LatticeState 是 FunctionSet 以外的其他三种状态时，Functions 为空。（Undefined 对应半格中的顶元素，根据半格的定义，对于任意数据流值 x，顶元素 ^ x = x，即顶元素与 x 的交汇运算的结果都是 x；Overdefined 对应半格中的底元素，根据半格的定义，对于任意数据流值 x，底元素 ^ x = 底元素）

CVPLatticeFunc

`CVPLatticeFunc` 继承自 `AbstractLatticeFunction`：`class CVPLatticeFunc : public AbstractLatticeFunction<CVPLatticeKey, CVPLatticeVal>`

首先看一下 `CVPLatticeFunc` 是怎么重写的 `MergeValues()` 函数（对应数据流分析中的交汇运算）。

```
CVPLatticeVal MergeValues(CVPLatticeVal X, CVPLatticeVal Y) override {
  if (X == getOverdefinedVal() || Y == getOverdefinedVal())
    return getOverdefinedVal();
  if (X == getUndefVal() && Y == getUndefVal())
    return getUndefVal();
  std::vector<Function *> Union;
  std::set_union(X.getFunctions().begin(), X.getFunctions().end(),
                 Y.getFunctions().begin(), Y.getFunctions().end(),
                 std::back_inserter(Union), CVPLatticeVal::Compare{});
  if (Union.size() > MaxFunctionsPerValue)
    return getOverdefinedVal();
```

(continues on next page)

(continued from previous page)

```
  return CVPLatticeVal(std::move(Union));
}
```

首先对需要进行交汇运算的两个操作数进行判断，如果是其中一个是底元素，那么交汇运算的结果就是底元素，直接返回 `getOverdefinedVal()`；如果两个操作数都是顶元素，那么交汇运算的结果就是顶元素，直接返回 `getUndefVal()`；其他情况就是对两个操作数的数据流值进行并集的操作。（注意到这里对并集的运算结果的大小进行判断，如果超过 MaxFunctionsPerValue (默认为 4)，就返回底元素 `getOverdefinedVal()`，代码的注释中的解释是：We likely can't do anything useful for call sites with a large number of possible targets, anyway.）

然后看一下 `CVPLatticeFunc` 是怎么重写的 `ComputeInstructionState()` 函数（对应数据流分析中的传递函数）。

```
void ComputeInstructionState(
    Instruction &I, DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
    SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) override {
  switch (I.getOpcode()) {
  case Instruction::Call:
    return visitCallSite(cast<CallInst>(&I), ChangedValues, SS);
  case Instruction::Invoke:
    return visitCallSite(cast<InvokeInst>(&I), ChangedValues, SS);
  case Instruction::Load:
    return visitLoad(*cast<LoadInst>(&I), ChangedValues, SS);
  case Instruction::Ret:
    return visitReturn(*cast<ReturnInst>(&I), ChangedValues, SS);
  case Instruction::Select:
    return visitSelect(*cast<SelectInst>(&I), ChangedValues, SS);
  case Instruction::Store:
    return visitStore(*cast<StoreInst>(&I), ChangedValues, SS);
  default:
    return visitInst(I, ChangedValues, SS);
  }
}
```

对于不同的 `Instruction` 实现不同的传递函数的逻辑。

visitSelect

下面先对 SelectInst 的传递函数 visitSelect() 进行分析，SelectInst 被用于实现基于条件的值的选择，SelectInst 不需要 LLVM IR 级别的分支指令的参与。语法如下：

```
<result> = select selty <cond>, <ty> <val1>, <ty> <val2>             ; yields ty

selty is either i1 or {<N x i1>}
```

一个 SelectInst 的例子如下：

```
%X = select i1 true, i8 17, i8 42          ; yields i8:17
```

在这条 SelectInst 中，<cond> 为 true，<val1> 为 17，它的 <ty> 为 i8，<val2> 为 42，它的 <ty> 为 i8。visitSelect() 函数的定义如下：

```
void visitSelect(SelectInst &I,
                 DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                 SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto RegI = CVPLatticeKey(&I, IPOGrouping::Register);
  auto RegT = CVPLatticeKey(I.getTrueValue(), IPOGrouping::Register);
  auto RegF = CVPLatticeKey(I.getFalseValue(), IPOGrouping::Register);
  ChangedValues[RegI] =
      MergeValues(SS.getValueState(RegT), SS.getValueState(RegF));
}
```

首先为这条 SelectInst 创建了一个类型为 IPOGrouping::Register 的 CVPLatticeKey RegI，然后为 SelectInst 的 TrueValue 和 FalseValue 分别创建了类型为 IPOGrouping::Register 的 CVPLatticeKey RegT 和 RegF。RegI 对应的数据流值 CVPLatticeVal 是由 RegT 的数据流值和 RegF 的数据流值进行交汇运算 MergeValues() 后得到的。RegT 的数据流值和 RegF 的数据流值是通过 SparseSolver 的成员函数 getValueState() 得到的，getValueState() 的定义如下：

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
LatticeVal
SparseSolver<LatticeKey, LatticeVal, KeyInfo>::getValueState(LatticeKey Key) {
  auto I = ValueState.find(Key);
  if (I != ValueState.end())
    return I->second; // Common case, in the map

  if (LatticeFunc->IsUntrackedValue(Key))
    return LatticeFunc->getUntrackedVal();
  LatticeVal LV = LatticeFunc->ComputeLatticeVal(Key);

  // If this value is untracked, don't add it to the map.
  if (LV == LatticeFunc->getUntrackedVal())
```

(continues on next page)

(continued from previous page)

```
    return LV;
  return ValueState[Key] = std::move(LV);
}
```

如果之前计算过某个 LatticeKey 对应的数据流值 LatticeVal，那么就会被存在 DenseMap<LatticeKey, LatticeVal> ValueState 中，如果是第一次查询这个 LatticeKey 对应的数据流值 LatticeVal，那么会调用函数 LatticeFunc->ComputeLatticeVal()，对于 CalledValuePropagationPass 来讲，LatticeFunc 就是 CVPLatticeFunc。CVPLatticeFunc 的 ComputeLatticeVal() 函数的定义如下：

```
CVPLatticeVal ComputeLatticeVal(CVPLatticeKey Key) override {
  switch (Key.getInt()) {
  case IPOGrouping::Register:
    if (isa<Instruction>(Key.getPointer())) {
      return getUndefVal();
    } else if (auto *A = dyn_cast<Argument>(Key.getPointer())) {
      if (canTrackArgumentsInterprocedurally(A->getParent()))
        return getUndefVal();
    } else if (auto *C = dyn_cast<Constant>(Key.getPointer())) {
      return computeConstant(C);
    }
    return getOverdefinedVal();
  case IPOGrouping::Memory:
  case IPOGrouping::Return:
    if (auto *GV = dyn_cast<GlobalVariable>(Key.getPointer())) {
      if (canTrackGlobalVariableInterprocedurally(GV))
        return computeConstant(GV->getInitializer());
    } else if (auto *F = cast<Function>(Key.getPointer()))
      if (canTrackReturnsInterprocedurally(F))
        return getUndefVal();
  }
  return getOverdefinedVal();
}
```

我们还是继续 visitSelect 函数的逻辑来跟进代码，在 visitSelect 函数中，通过 SparseSolver<LatticeKey, LatticeVal, KeyInfo>::getValueState(LatticeKey Key) 获取 RegT 的数据流值和 RegF 的数据流值时，如果是第一次查询它们的数据流值，就会调用 CVPLatticeFunc 的 ComputeLatticeVal() 函数，因为 RegT 和 RegF 都是 IPOGrouping::Register 所以会进入 case IPOGrouping::Register: 这个分支，可以看到当 SelectInst 的 TrueValue(RegT) 或者 FalseValue(RegF) 是 Constant 时，会调用 computeConstant() 函数。

```
CVPLatticeVal computeConstant(Constant *C) {
  if (isa<ConstantPointerNull>(C))
```

(continues on next page)

(continued from previous page)

```
    return CVPLatticeVal(CVPLatticeVal::FunctionSet);
  if (auto *F = dyn_cast<Function>(C->stripPointerCasts()))
    return CVPLatticeVal({F});
  return getOverdefinedVal();
}
```

在 `computeConstant()` 函数中，如果 Constant *C 是 Function，最终返回这个函数。

举个例子，应该会更直观，我们有这样一条 SelectInst：

```
%func = select i1 %cond, i1 (i64, i64)* @ugt, i1 (i64, i64)* @ule
```

%cond 是某条件，TrueValue 是这个 CalledValuePropagationPass 的分析对象 LLVM IR 中的函数 ugt，FalseValue 是函数 ugt。然后我们想要计算在执行完这条指令后的数据流值，因为这是一个 SelectInst，所以应用传递函数时执行调用的是 `visitSelect` 函数，然后我们是第一次查询该 SelectInst 的 TrueValue 和 FalseValue 对应的 `CVPLatticeVal`，所以会调用函数 `ComputeLatticeVal()`，最终得到的就是 `CVPLatticeVal{@ugt}` 和 `CVPLatticeVal{@ule}`，所以更新后的%func 对应的 `CVPLatticeVal` 就是 `MergeValues(CVPLatticeVal{@ugt}, CVPLatticeVal{@ule})` 即 `CVPLatticeVal{@ugt, @ule}`。

visitLoad

LoadInst 对应的传递函数是 `visitLoad()`：

```
void visitLoad(LoadInst &I,
               DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
               SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto RegI = CVPLatticeKey(&I, IPOGrouping::Register);
  if (auto *GV = dyn_cast<GlobalVariable>(I.getPointerOperand())) {
    auto MemGV = CVPLatticeKey(GV, IPOGrouping::Memory);
    ChangedValues[RegI] =
        MergeValues(SS.getValueState(RegI), SS.getValueState(MemGV));
  } else {
    ChangedValues[RegI] = getOverdefinedVal();
  }
}
```

可以发现，只有当 LoadInst 的 PointerOperand 是 GlobalVariable 时才进行分析，可见该 CalledValuePropagationPass 还是比较保守或者说是比较简单的。该传递函数很简单，RegI 对应的新数据流值 `CVPLatticeVal` 是由 RegI 的原数据流值和 MemGV 的数据流值进行交汇运算 `MergeValues()` 得到。

visitStore

StoreInst 对应的传递函数是 `visitStore()`：

```
void visitStore(StoreInst &I,
                DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto *GV = dyn_cast<GlobalVariable>(I.getPointerOperand());
  if (!GV)
    return;
  auto RegI = CVPLatticeKey(I.getValueOperand(), IPOGrouping::Register);
  auto MemGV = CVPLatticeKey(GV, IPOGrouping::Memory);
  ChangedValues[MemGV] =
      MergeValues(SS.getValueState(RegI), SS.getValueState(MemGV));
}
```

visitReturn

ReturnInst 对应的传递函数是 `visitReturn()`：

```
void visitReturn(ReturnInst &I,
                 DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                 SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  Function *F = I.getParent()->getParent();
  if (F->getReturnType()->isVoidTy())
    return;
  auto RegI = CVPLatticeKey(I.getReturnValue(), IPOGrouping::Register);
  auto RetF = CVPLatticeKey(F, IPOGrouping::Return);
  ChangedValues[RetF] =
      MergeValues(SS.getValueState(RegI), SS.getValueState(RetF));
}
```

该传递函数稍微有一点特殊，因为 CalledValuePropagationPass 是过程间分析的。所以对于 ReturnInst，会对该函数的 IPOGrouping::Register 类型的数据流值 `CVPLatticeVal` 进行更新，这样的话，当有 callsite 调用该函数时，就能计算出该 callsite 的返回值的数据流值。

visitCallSite

CallInst 和 InvokeInst 对应的传递函数都是 `visitCallSite()`：

```
void visitCallSite(CallSite CS,
                   DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                   SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  Function *F = CS.getCalledFunction();
  Instruction *I = CS.getInstruction();
  auto RegI = CVPLatticeKey(I, IPOGrouping::Register);

  // If this is an indirect call, save it so we can quickly revisit it when
  // attaching metadata.
  if (!F)
    IndirectCalls.insert(I);

  // If we can't track the function's return values, there's nothing to do.
  if (!F || !canTrackReturnsInterprocedurally(F)) {
    // Void return, No need to create and update CVPLattice state as no one
    // can use it.
    if (I->getType()->isVoidTy())
      return;
    ChangedValues[RegI] = getOverdefinedVal();
    return;
  }

  // Inform the solver that the called function is executable, and perform
  // the merges for the arguments and return value.
  SS.MarkBlockExecutable(&F->front());
  auto RetF = CVPLatticeKey(F, IPOGrouping::Return);
  for (Argument &A : F->args()) {
    auto RegFormal = CVPLatticeKey(&A, IPOGrouping::Register);
    auto RegActual =
        CVPLatticeKey(CS.getArgument(A.getArgNo()), IPOGrouping::Register);
    ChangedValues[RegFormal] =
        MergeValues(SS.getValueState(RegFormal), SS.getValueState(RegActual));
  }

  // Void return, No need to create and update CVPLattice state as no one can
  // use it.
  if (I->getType()->isVoidTy())
    return;

  ChangedValues[RegI] =
      MergeValues(SS.getValueState(RegI), SS.getValueState(RetF));
```

(continues on next page)

(continued from previous page)

```
}
```

对函数调用的参数和返回值进行处理，背后的逻辑很简单：对于参数来讲，被调函数的形参的可能取值就是对该函数的所有调用点的实参的并集，因此 `visitCallSite` 就是把当前 call site 的实参的数据流值并入被调函数的形参的数据流值中；而该 call site 的返回值就是所有可能被调函数的返回值的并集，所以 `visitCallSite` 就是把当前 call site 的被调函数的返回值的数据流值（在 VistieReturn 中被设置）并入当前 call site 的返回值的数据流值中。

值得注意的是：`SS.MarkBlockExecutable(&F->front());`，将被调函数的入口基本块添加至 SparseSolver 的 BBWorkList 和 BBExecutable 集合中。因为这里更新了被调函数的形参的数据流值，所以需要再次对被调函数中的数据流值进行迭代更新。

visitReturn 和 visitCallSite 的实现使得该 CalledValuePropagationPass 是过程间的分析。

visitInst

该函数是对除了上述指令外的其他指令的传递函数，就是简单的设置为数据流值设置为 `getOverdefinedVal()` 。

```
void visitInst(Instruction &I,
               DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
               SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto RegI = CVPLatticeKey(&I, IPOGrouping::Register);
  ChangedValues[RegI] = getOverdefinedVal();
}
```

How to call CalledValuePropagationPass in your code

除了可以通过 opt 命令来调用该 pass 来获取间接调用点的被调函数的可能取值，还可以通过 PassManager 在你的代码中调用 CalledValuePropagationPass 。

可以参考：https://github.com/Enna1/LLVM-Clang-Examples/tree/master/use-calledvaluepropagation-in-your-tool

---

- genindex
- modindex
- search

5.3 Correlated Value Propagation

5.3.1 CorrelatedValuePropagation

Introduction

CorrelatedValuePropagation 是一个 transform pass，在 LLVM 9.0.1 中该 pass 的实现代码位于 `llvm-9.0.1.src\include\llvm\Transforms\Scalar\CorrelatedValuePropagation.h` 和 `llvm-9.0.1.src\lib\Transforms\Scalar\CorrelatedValuePropagation.cpp` 中。

该 pass 通过 propagate CFG-derived info 来对代码进行优化。

Implementation

该 pass 的定义如下：

```
class CorrelatedValuePropagation : public FunctionPass
{
public:
    static char ID;

    CorrelatedValuePropagation() : FunctionPass(ID)
    {
        initializeCorrelatedValuePropagationPass(
            *PassRegistry::getPassRegistry());
    }

    bool runOnFunction(Function &F) override;

    void getAnalysisUsage(AnalysisUsage &AU) const override
    {
        AU.addRequired<DominatorTreeWrapperPass>();
        AU.addRequired<LazyValueInfoWrapperPass>();
        AU.addPreserved<GlobalsAAWrapperPass>();
        AU.addPreserved<DominatorTreeWrapperPass>();
    }
};
```

可见该 pass 是一个 FunctionPass，在该 pass 中用到了 `DominatorTree` 和 `LazyValueInfo` 的 analysis pass 的分析结果，并且在该 pass 执行后 `GlobalAA` 和 `DominatorTree` 这两个 analysis pass 的分析结果仍然是有效的。

下面我们看一下 `bool CorrelatedValuePropagation::runOnFunction()` 的实现：

```
bool CorrelatedValuePropagation::runOnFunction(Function &F)
{
    if (skipFunction(F))
        return false;

    LazyValueInfo *LVI = &getAnalysis<LazyValueInfoWrapperPass>().getLVI();
    DominatorTree *DT = &getAnalysis<DominatorTreeWrapperPass>().getDomTree();

    return runImpl(F, LVI, DT, getBestSimplifyQuery(*this, F));
}
```

可以看到，CorrelatedValuePropagation 这个 pass 的核心功能是在 runImpl() 函数中实现的：

```
static bool runImpl(Function &F, LazyValueInfo *LVI, DominatorTree *DT,
                    const SimplifyQuery &SQ)
{
    bool FnChanged = false;
    // Visiting in a pre-order depth-first traversal causes us to simplify early
    // blocks before querying later blocks (which require us to analyze early
    // blocks).  Eagerly simplifying shallow blocks means there is strictly less
    // work to do for deep blocks.  This also means we don't visit unreachable
    // blocks.
    for (BasicBlock *BB : depth_first(&F.getEntryBlock()))
    {
        bool BBChanged = false;
        for (BasicBlock::iterator BI = BB->begin(), BE = BB->end(); BI != BE;)
        {
            Instruction *II = &*BI++;
            switch (II->getOpcode())
            {
            case Instruction::Select:
                BBChanged |= processSelect(cast<SelectInst>(II), LVI);
                break;
            case Instruction::PHI:
                BBChanged |= processPHI(cast<PHINode>(II), LVI, DT, SQ);
                break;
            case Instruction::ICmp:
            case Instruction::FCmp:
                BBChanged |= processCmp(cast<CmpInst>(II), LVI);
                break;
            case Instruction::Load:
            case Instruction::Store:
                BBChanged |= processMemAccess(II, LVI);
                break;
```

(continues on next page)

(continued from previous page)

```
            case Instruction::Call:
            case Instruction::Invoke:
                BBChanged |= processCallSite(CallSite(II), LVI);
                break;
            case Instruction::SRem:
                BBChanged |= processSRem(cast<BinaryOperator>(II), LVI);
                break;
            case Instruction::SDiv:
                BBChanged |= processSDiv(cast<BinaryOperator>(II), LVI);
                break;
            case Instruction::UDiv:
            case Instruction::URem:
                BBChanged |= processUDivOrURem(cast<BinaryOperator>(II), LVI);
                break;
            case Instruction::AShr:
                BBChanged |= processAShr(cast<BinaryOperator>(II), LVI);
                break;
            case Instruction::Add:
            case Instruction::Sub:
                BBChanged |= processBinOp(cast<BinaryOperator>(II), LVI);
                break;
            }
        }

        Instruction *Term = BB->getTerminator();
        switch (Term->getOpcode())
        {
        case Instruction::Switch:
            BBChanged |= processSwitch(cast<SwitchInst>(Term), LVI, DT);
            break;
        case Instruction::Ret:
        {
            auto *RI = cast<ReturnInst>(Term);
            // Try to determine the return value if we can.  This is mainly here
            // to simplify the writing of unit tests, but also helps to enable
            // IPO by constant folding the return values of callees.
            auto *RetVal = RI->getReturnValue();
            if (!RetVal)
                break;  // handle "ret void"
            if (isa<Constant>(RetVal))
                break;  // nothing to do
            if (auto *C = getConstantAt(RetVal, RI, LVI))
            {
```

(continues on next page)

(continued from previous page)

```
                ++NumReturns;
                RI->replaceUsesOfWith(RetVal, C);
                BBChanged = true;
            }
        }
        }

        FnChanged |= BBChanged;
    }

    return FnChanged;
}
```

该函数的作用就是，以深度优先遍历的方式遍历函数中的基本块，然后对于基本块中的不同类型的指令调用不同的处理函数。这里使用深度优先遍历是可以避免浪费时间来优化不可达的基本块上。

说到这里，我们通过 `for (BasicBlock &BB : F)` 来遍历基本块时，基本块是按照什么顺序被遍历，在 http://llvm.org/docs/ProgrammersManual.html#the-function-class 文档中有说明：

> The list of BasicBlocks is the most commonly used part of `Function` objects. **The list imposes an implicit ordering of the blocks in the function, which indicate how the code will be laid out by the backend**. Additionally, the first BasicBlock is the implicit entry node for the `Function`. It is not legal in LLVM to explicitly branch to this initial block. There are no implicit exit nodes, and in fact there may be multiple exit nodes from a single `Function`. If the BasicBlock list is empty, this indicates that the `Function` is actually a function declaration: the actual body of the function hasn't been linked in yet.

接下来我们分别分析对不同指令进行处理的函数：`processSelect()`, `processPHI()`, `processCmp()`, `processMemAccess()`, `processCallSite()`, `processSRem()`, `processSDiv()`, `processUDivOrURem()`, `processAShr()`, `processBinOp()`

processSelect()

关于 `select` Instruction，http://llvm.org/docs/LangRef.html#select-instruction

`select` 指令的 Syntax 如下：

```
<result> = select [fast-math flags] selty <cond>, <ty> <val1>, <ty> <val2>    ;␣
↪yields ty

selty is either i1 or {<N x i1>}
```

对于 `selty` 是 `i1` 情况，`select` 指令根据 `<cond>` 是否为 1 来选择 `<val1>` 或 `<val2>`：

```
%X = select i1 true, i8 17, i8 42          ; yields i8:17
```

在 runImpl() 函数中，如果遍历到了 select Instruction，则调用 processSelect() 进行处理：

```
static bool processSelect(SelectInst *S, LazyValueInfo *LVI)
{
    if (S->getType()->isVectorTy())
        return false;
    if (isa<Constant>(S->getOperand(0)))
        return false;

    Constant *C = LVI->getConstant(S->getCondition(), S->getParent(), S);
    if (!C)
        return false;

    ConstantInt *CI = dyn_cast<ConstantInt>(C);
    if (!CI)
        return false;

    Value *ReplaceWith = S->getTrueValue();
    Value *Other = S->getFalseValue();
    if (!CI->isOne())
        std::swap(ReplaceWith, Other);
    if (ReplaceWith == S)
        ReplaceWith = UndefValue::get(S->getType());

    S->replaceAllUsesWith(ReplaceWith);
    S->eraseFromParent();

    ++NumSelects;

    return true;
}
```

该函数不处理 select Instruction 的 selty 为 {<N x i1>} 的情况，只处理其为 i1 的情况。如果我们能通过 LazyValueInfo 的分析结果知道 <cond> 的取值只可能是一个 ConstaintInt，那么如果 <cond> 是 1，那么就将所有 <result> 的使用点直接替换为 <val1>，否则就将 <result> 的使用点直接替换为 <val2> 。

但是这个判断语句有点奇怪：if (ReplaceWith == S) ReplaceWith = UndefValue::get(S->getType()); 查看了 /lib/Transforms/Scalar/CorrelatedValuePropagation.cpp 这个文件的 change history 后发现，这行代码是在 https://reviews.llvm.org/rG35609d97ae89b8e13f40f4e6b9b056954f8baa83 中引入的：

下面的 LLVM IR 测试用例中的有一条这样的 select instruction，即 <reslt> 和 val1> 或 <val2> 是同一个 Value，如果没有这行特殊的判断代码，会造成 crash 。而这条语句 if (ReplaceWith == S) ReplaceWith = UndefValue::get(S->getType()) 将这样的 select instruction 的 <result> 的使用点替换为 undef 。

```
define void @test2() nounwind ssp {
entry:
  br label %func_29.exit

sdf.exit.i:
  %l_44.1.mux.i = select i1 %tobool5.not.i, i8 %l_44.1.mux.i, i8 1
  br label %srf.exit.i

srf.exit.i:
  %tobool5.not.i = icmp ne i8 undef, 0
  br i1 %tobool5.not.i, label %sdf.exit.i, label %func_29.exit

func_29.exit:
  ret void
}
```

P.S. 但是这个造成 crash 的 `select` instruction 是不可达的代码，而我们是通过深度优先遍历基本块（https://reviews.llvm.org/rGd10480657527ffb44ea213460fb3676a6b1300aa 引入）的，不可能访问该代码，所以可以去掉该部分的判断，因此我提了一个 patch (https://reviews.llvm.org/D76753) 来删掉这条冗余的语句 `if (ReplaceWith == S) ReplaceWith = UndefValue::get(S->getType())` 。

processPHI()

关于 `phi` Instruction，http://llvm.org/docs/LangRef.html#phi-instruction

```
static bool processPHI(PHINode *P, LazyValueInfo *LVI, DominatorTree *DT,
                       const SimplifyQuery &SQ)
{
    bool Changed = false;

    BasicBlock *BB = P->getParent();
    for (unsigned i = 0, e = P->getNumIncomingValues(); i < e; ++i)
    {
        Value *Incoming = P->getIncomingValue(i);
        if (isa<Constant>(Incoming))
            continue;

        Value *V =
            LVI->getConstantOnEdge(Incoming, P->getIncomingBlock(i), BB, P);

        // Look if the incoming value is a select with a scalar condition for
        // which LVI can tells us the value. In that case replace the incoming
        // value with the appropriate value of the select. This often allows us
```

(continues on next page)

(continued from previous page)

```
        // to remove the select later.
        if (!V)
        {
            SelectInst *SI = dyn_cast<SelectInst>(Incoming);
            if (!SI)
                continue;

            Value *Condition = SI->getCondition();
            if (!Condition->getType()->isVectorTy())
            {
                if (Constant *C = LVI->getConstantOnEdge(
                        Condition, P->getIncomingBlock(i), BB, P))
                {
                    if (C->isOneValue())
                    {
                        V = SI->getTrueValue();
                    }
                    else if (C->isZeroValue())
                    {
                        V = SI->getFalseValue();
                    }
                    // Once LVI learns to handle vector types, we could also add
                    // support for vector type constants that are not all zeroes
                    // or all ones.
                }
            }

            // Look if the select has a constant but LVI tells us that the
            // incoming value can never be that constant. In that case replace
            // the incoming value with the other value of the select. This often
            // allows us to remove the select later.
            if (!V)
            {
                Constant *C = dyn_cast<Constant>(SI->getFalseValue());
                if (!C)
                    continue;

                if (LVI->getPredicateOnEdge(ICmpInst::ICMP_EQ, SI, C,
                                            P->getIncomingBlock(i), BB,
                                            P) != LazyValueInfo::False)
                    continue;
                V = SI->getTrueValue();
            }
```

(continues on next page)

(continued from previous page)

```
            LLVM_DEBUG(dbgs() << "CVP: Threading PHI over " << *SI << '\n');
        }

        P->setIncomingValue(i, V);
        Changed = true;
    }

    if (Value *V = SimplifyInstruction(P, SQ))
    {
        P->replaceAllUsesWith(V);
        P->eraseFromParent();
        Changed = true;
    }

    if (!Changed)
        Changed = simplifyCommonValuePhi(P, LVI, DT);

    if (Changed)
        ++NumPhis;

    return Changed;
}
```

`processPHI()` 函数的内容看起来很多，但是结合注释来看实际上还是比较清晰的。

首先遍历 `phi` instruction 的所有 incoming values，对于每个 incoming value，如果我们能够通过 LazyValueInfo 的分析结果确定 incoming value 从 incoming basic block 到 `phi` instruction 所在的 basic block 这条边上的取值只能是一个常量（通过 LazyValueInfo 的成员函数 `Constant *LazyValueInfo::getConstantOnEdge(Value *V, BasicBlock *FromBB, BasicBlock *ToBB, Instruction *CxtI)`），那么就将 `phi` instruction 中的该 incoming value 设置为这个常量；否则，考察该 incoming value 是否为一个 `select` instruction，通过 LazyValueInfo 对这个 `select` instruction 的分析结果将 `phi` instruction 的 incoming value 替换为这个 `select` instruction 中对应的值。

处理完上述 `phi` instruction 的所有 incoming values 后，然后对 `phi` instruction 调用 `SimplifyInstruction()` 函数（该函数位于 `lib\Analysis\InstructionSimplify.cpp`，该函数实现了将给定的指令折叠 (fold) 为更简单的形式，例如 “and i32 %x, 0” -> “0”）查看是否能将 `phi` instruction 折叠为更简单的形式。

如果经过了前面两个步骤，`phi` instruction 还是没能被优化，那么就调用 `simplifyCommonValuePhi()` 函数，这里就不贴 `simplifyCommonValuePhi()` 函数的代码实现了，该函数检查 `phi` instruction 的 incoming values 是不是由 1 个是 variable incoming value 和多个 constant incoming values 构成的，并且这多个 constant incoming values 能映射回这个 variable incoming value，如果是，就将所有的 `phi` instruction 的使用点替换为这个 variable incoming value，举个例子：

```
/// bb0:
///   %isnull = icmp eq i8* %x, null
///   br i1 %isnull, label %bb2, label %bb1
/// bb1:
///   br label %bb2
/// bb2:
///   %r = phi i8* [ %x, %bb1 ], [ null, %bb0 ]
/// -->
///   %r = %x
```

在这个 `phi` instruction 有两个 incoming values，一个是 variable incoming value `%x`，另一个是 constant incoming values `null`，并且通过 LazyValueInfo 的分析可知 `%x` 在 `%bb0` (`null` 对应的 incoming basic block) 到 `%bb2` 的这条边上的值就是 `null`，所以可以直接将 `phi` 的 instruction 替换为 `%x`

processCmp()

关于 `icmp` instruction，http://llvm.org/docs/LangRef.html#icmp-instruction

关于 `fcmp` instruction，http://llvm.org/docs/LangRef.html#fcmp-instruction

对于 `icmp` instruction 和 `fcmp` intructsion 会调用该函数 `processCmp()` 尝试对指令优化：

```
/// See if LazyValueInfo's ability to exploit edge conditions or range
/// information is sufficient to prove this comparison. Even for local
/// conditions, this can sometimes prove conditions instcombine can't by
/// exploiting range information.
static bool processCmp(CmpInst *Cmp, LazyValueInfo *LVI)
{
    Value *Op0 = Cmp->getOperand(0);
    auto *C = dyn_cast<Constant>(Cmp->getOperand(1));
    if (!C)
        return false;

    // As a policy choice, we choose not to waste compile time on anything where
    // the comparison is testing local values.  While LVI can sometimes reason
    // about such cases, it's not its primary purpose.  We do make sure to do
    // the block local query for uses from terminator instructions, but that's
    // handled in the code for each terminator.
    auto *I = dyn_cast<Instruction>(Op0);
    if (I && I->getParent() == Cmp->getParent())
        return false;

    LazyValueInfo::Tristate Result =
        LVI->getPredicateAt(Cmp->getPredicate(), Op0, C, Cmp);
```

(continues on next page)

(continued from previous page)

```
    if (Result == LazyValueInfo::Unknown)
        return false;

    ++NumCmps;
    Constant *TorF =
        ConstantInt::get(Type::getInt1Ty(Cmp->getContext()), Result);
    Cmp->replaceAllUsesWith(TorF);
    Cmp->eraseFromParent();
    return true;
}
```

首先判断 `CmpInst` 的第二个操作数是不是 Constant，如果不是 Constant 则直接退出函数。然后查看第一个操作数，如果第一个操作数是一个与 `CmpInst` 处于同一个基本块的 `Instruction` 的话，则退出函数。最后借助 LazyValueInfo 查看是否能够直接分析出 `CmpInst` 的计算结果是 true 还是 false，如果能，则直接将 `CmpInst` 的使用点都替换为 true 或 false。

下面举一个具体的例子，进行说明：

```
define void @test1(i64 %tmp35) {
bb:
  %tmp36 = icmp sgt i64 %tmp35, 0
  br i1 %tmp36, label %bb_true, label %bb_false

bb_true:
  %tmp47 = icmp slt i64 %tmp35, 0
  tail call void @check1(i1 %tmp47) #0
  unreachable

bb_false:
  %tmp48 = icmp sle i64 %tmp35, 0
  tail call void @check2(i1 %tmp48) #4
  unreachable
}
attributes #0 = { noreturn }
```

对于 bb_true 基本块中的 `%tmp47 = icmp slt i64 %tmp35, 0` 这条 `CmpInst`，第二个操作数是 Constant 0，第一个操作数不是和这条 `CmpInst` 在同一个基本块中的 `Instruction`。然后我们通过 LazyValueInfo 可以知道，这条 `CmpInst` 的结果就是 false，因为想要执行至该 `CmpInst`，那么一定是从 bb 基本块跳转至 bb_true 基本块的，那么 `%tmp36 = icmp sgt i64 %tmp35, 0` 就一定是 true，所以 `%tmp47 = icmp slt i64 %tmp35, 0` 就一定是 false，然后就可以将 `tail call void @check1(i1 %tmp47)` 中的 `%tmp47` 替换为 false，即 `tail call void @check1(i1 false)`。

对于 bb_false 基本块中的 `%tmp48 = icmp sle i64 %tmp35, 0` 同理。

经过 `processCmp()` 处理后，上述例子.ll 文件被优化为了如下所示：

```
define void @test1(i64 %tmp35) {
bb:
  %tmp36 = icmp sgt i64 %tmp35, 0
  br i1 %tmp36, label %bb_true, label %bb_false

bb_true:                                          ; preds = %bb
  tail call void @check1(i1 false) #0
  unreachable

bb_false:                                         ; preds = %bb
  tail call void @check2(i1 true) #0
  unreachable
}
attributes #0 = { noreturn }
```

processMemAccess()

关于 `load` instruction，http://llvm.org/docs/LangRef.html#load-instruction

关于 `store` instruction，http://llvm.org/docs/LangRef.html#store-instruction

对于 `load` instruction 和 `store` intructsion 会调用该函数 `processMemAccess()` 尝试对指令优化：

```
static bool processMemAccess(Instruction *I, LazyValueInfo *LVI)
{
    Value *Pointer = nullptr;
    if (LoadInst *L = dyn_cast<LoadInst>(I))
        Pointer = L->getPointerOperand();
    else
        Pointer = cast<StoreInst>(I)->getPointerOperand();

    if (isa<Constant>(Pointer))
        return false;

    Constant *C = LVI->getConstant(Pointer, I->getParent(), I);
    if (!C)
        return false;

    ++NumMemAccess;
    I->replaceUsesOfWith(Pointer, C);
    return true;
}
```

首先获取 `load` instruction 或 `store` instruction 的 pointer operand，如果 pointer operand 是 Constant 则退出函数。然后如果可以通过 LazyValueInfo 分析得出 pointer operand 的取值只能是一个 Constant，那么就将 `load`

instruction 或 `store` instruction 中的原 pointer operand 替换为该 Constant 。

举例说明：

```
define i8 @test3(i8* %a) {
entry:
  %cond = icmp eq i8* %a, @gv
  br i1 %cond, label %bb2, label %bb

bb:
  ret i8 0

bb2:
  %should_be_const = load i8, i8* %a
  ret i8 %should_be_const
}
```

对于 `%should_be_const = load i8, i8* %a` 这条 `load` instruction，其 pointer operand 为 `%a`，不是一个 Constant，但是通过 LazyValueInfo 分析可以知道 `%a` 的值只能是 `@gv` 这个 Constant，所以就可以直接将 `%should_be_const = load i8, i8* %a` 中的 `%a` 替换为 `@gv`：

```
define i8 @test3(i8* %a) {
entry:
  %cond = icmp eq i8* %a, @gv
  br i1 %cond, label %bb2, label %bb

bb:                                               ; preds = %entry
  ret i8 0

bb2:                                              ; preds = %entry
  %should_be_const = load i8, i8* @gv
  ret i8 %should_be_const
}
```

processCallSite()

关于 `call` instruction，http://llvm.org/docs/LangRef.html#call-instruction

关于 `invoke` intructsion，http://llvm.org/docs/LangRef.html#invoke-instruction

对于 `call` instruction 和 `invoke` intructsion 会调用该函数 `processCallSite()` 尝试对指令优化：

```
/// Infer nonnull attributes for the arguments at the specified callsite.
static bool processCallSite(CallSite CS, LazyValueInfo *LVI)
{
```

(continues on next page)

(continued from previous page)

```
    SmallVector<unsigned, 4> ArgNos;
    unsigned ArgNo = 0;

    if (auto *WO = dyn_cast<WithOverflowInst>(CS.getInstruction()))
    {
        if (WO->getLHS()->getType()->isIntegerTy() && willNotOverflow(WO, LVI))
        {
            processOverflowIntrinsic(WO);
            return true;
        }
    }

    if (auto *SI = dyn_cast<SaturatingInst>(CS.getInstruction()))
    {
        if (SI->getType()->isIntegerTy() && willNotOverflow(SI, LVI))
        {
            processSaturatingInst(SI);
            return true;
        }
    }

    // Deopt bundle operands are intended to capture state with minimal
    // perturbance of the code otherwise.  If we can find a constant value for
    // any such operand and remove a use of the original value, that's
    // desireable since it may allow further optimization of that value (e.g.
    // via single use rules in instcombine).  Since deopt uses tend to,
    // idiomatically, appear along rare conditional paths, it's reasonable
    // likely we may have a conditional fact with which LVI can fold.
    if (auto DeoptBundle = CS.getOperandBundle(LLVMContext::OB_deopt))
    {
        bool Progress = false;
        for (const Use &ConstU : DeoptBundle->Inputs)
        {
            Use &U = const_cast<Use &>(ConstU);
            Value *V = U.get();
            if (V->getType()->isVectorTy())
                continue;
            if (isa<Constant>(V))
                continue;

            Constant *C =
                LVI->getConstant(V, CS.getParent(), CS.getInstruction());
            if (!C)
```

(continues on next page)

(continued from previous page)

```
                continue;
            U.set(C);
            Progress = true;
        }
        if (Progress)
            return true;
    }

    for (Value *V : CS.args())
    {
        PointerType *Type = dyn_cast<PointerType>(V->getType());
        // Try to mark pointer typed parameters as non-null.  We skip the
        // relatively expensive analysis for constants which are obviously
        // either null or non-null to start with.
        if (Type && !CS.paramHasAttr(ArgNo, Attribute::NonNull) &&
            !isa<Constant>(V) &&
            LVI->getPredicateAt(ICmpInst::ICMP_EQ, V,
                                ConstantPointerNull::get(Type),
                                CS.getInstruction()) == LazyValueInfo::False)
            ArgNos.push_back(ArgNo);
        ArgNo++;
    }

    assert(ArgNo == CS.arg_size() && "sanity check");

    if (ArgNos.empty())
        return false;

    AttributeList AS = CS.getAttributes();
    LLVMContext &Ctx = CS.getInstruction()->getContext();
    AS = AS.addParamAttribute(Ctx, ArgNos,
                              Attribute::get(Ctx, Attribute::NonNull));
    CS.setAttributes(AS);

    return true;
}
```

该函数看起来比较复杂，实际上可以将该函数拆分为几个部分分别来看：

- 判断该 instruction 是否是一个 `WithOverflowInst` (如 `llvm.sadd.with.overflow.i32`)，如果是，并且操作数都是整数类型且确定不会发生溢出（`willNotOverflow` 函数返回 true），则调用 `processOverflowIntrinsic()` 后，退出函数；否则，继续执行。

```
static void processOverflowIntrinsic(WithOverflowInst *WO)
{
    IRBuilder<> B(WO);
    Value *NewOp = B.CreateBinOp(WO->getBinaryOp(), WO->getLHS(), WO->getRHS(),
                                 WO->getName());
    // Constant-folding could have happened.
    if (auto *Inst = dyn_cast<Instruction>(NewOp))
    {
        if (WO->isSigned())
            Inst->setHasNoSignedWrap();
        else
            Inst->setHasNoUnsignedWrap();
    }

    Value *NewI = B.CreateInsertValue(UndefValue::get(WO->getType()), NewOp, 0);
    NewI =
        B.CreateInsertValue(NewI, ConstantInt::getFalse(WO->getContext()), 1);
    WO->replaceAllUsesWith(NewI);
    WO->eraseFromParent();
    ++NumOverflows;
}
```

`processOverflowIntrinsic()` 函数就是将 `WithOverflowInst` 替换为相应的 `BinaryOperator`，并添加 `nsw` 或 `nuw` keyword

- 判断该 instruction 是否是一个 `SaturatingInst` (如 `llvm.sadd.sat.i32`)，如果是，并且操作数都是整数类型且确定不会发生溢出 (`willNotOverflow` 函数返回 true)，则调用 `processSaturatingInst()` 后，退出函数；否则，继续执行。

```
static void processSaturatingInst(SaturatingInst *SI)
{
    BinaryOperator *BinOp = BinaryOperator::Create(
        SI->getBinaryOp(), SI->getLHS(), SI->getRHS(), SI->getName(), SI);
    BinOp->setDebugLoc(SI->getDebugLoc());
    if (SI->isSigned())
        BinOp->setHasNoSignedWrap();
    else
        BinOp->setHasNoUnsignedWrap();

    SI->replaceAllUsesWith(BinOp);
    SI->eraseFromParent();
    ++NumSaturating;
}
```

`processSaturatingInst()` 函数就是将 `SaturatingInst` 替换为相应的 `BinaryOperator`，并

添加 `nsw` 或 `nuw` keyword

- 如果不满足上述两种情况，那么通过 LazyValueInfo 分析调用点的“deopt”的每一个操作数的取值是否只能为一个常数，如果能，则替换为这个常数，然后退出函数；否则，继续执行。
- 如果上述三种情况都不满足，那么通过 LazyValueInfo 分析函数调用点的每一个实参，对于每一个实参，如果没有设置 `nonnull` attribute，实参不是 Constaint，并且能够分析出实参的取值一定不能是 null，那么为这样的实参设置 `nonnull` attribute 。

下面举一个“deopt”的例子：

```
define void @test1(i1 %c, i1 %c2) {
  %sel = select i1 %c, i64 -1, i64 1
  %sel2 = select i1 %c2, i64 %sel, i64 0
  %cmp = icmp sgt i64 %sel2, 0
  br i1 %cmp, label %taken, label %untaken
taken:
  call void @use() ["deopt" (i64 %sel2)]
  ret void
untaken:
  ret void
}
```

通过 LazyValueInfo 分析出 `%sel2` 的取值只能是 1，上述 LLVM IR 经过 `processCallSite()` 处理后变为如下所示：

```
define void @test1(i1 %c, i1 %c2) {
  %sel = select i1 %c, i64 -1, i64 1
  %sel2 = select i1 %c2, i64 %sel, i64 0
  %cmp = icmp sgt i64 %sel2, 0
  br i1 %cmp, label %taken, label %untaken

taken:                                            ; preds = %0
  call void @use() [ "deopt"(i64 1) ]
  ret void

untaken:                                          ; preds = %0
  ret void
}
```

processUDivOrURem()

关于 udiv instruction，http://llvm.org/docs/LangRef.html#sdiv-instruction

关于 urem intructsion，http://llvm.org/docs/LangRef.html#urem-instruction

对于 udiv instruction 和 urem intructsion 会调用该函数 processUDivOrURem() 尝试对指令优化：

```
/// Try to shrink a udiv/urem's width down to the smallest power of two that's
/// sufficient to contain its operands.
static bool processUDivOrURem(BinaryOperator *Instr, LazyValueInfo *LVI)
{
    assert(Instr->getOpcode() == Instruction::UDiv ||
           Instr->getOpcode() == Instruction::URem);
    if (Instr->getType()->isVectorTy())
        return false;

    // Find the smallest power of two bitwidth that's sufficient to hold Instr's
    // operands.
    auto OrigWidth = Instr->getType()->getIntegerBitWidth();
    ConstantRange OperandRange(OrigWidth, /*isFullSet=*/false);
    for (Value *Operand : Instr->operands())
    {
        OperandRange = OperandRange.unionWith(
            LVI->getConstantRange(Operand, Instr->getParent()));
    }
    // Don't shrink below 8 bits wide.
    unsigned NewWidth = std::max<unsigned>(
        PowerOf2Ceil(OperandRange.getUnsignedMax().getActiveBits()), 8);
    // NewWidth might be greater than OrigWidth if OrigWidth is not a power of
    // two.
    if (NewWidth >= OrigWidth)
        return false;

    ++NumUDivs;
    IRBuilder<> B{Instr};
    auto *TruncTy = Type::getIntNTy(Instr->getContext(), NewWidth);
    auto *LHS = B.CreateTruncOrBitCast(Instr->getOperand(0), TruncTy,
                                       Instr->getName() + ".lhs.trunc");
    auto *RHS = B.CreateTruncOrBitCast(Instr->getOperand(1), TruncTy,
                                       Instr->getName() + ".rhs.trunc");
    auto *BO = B.CreateBinOp(Instr->getOpcode(), LHS, RHS, Instr->getName());
    auto *Zext = B.CreateZExt(BO, Instr->getType(), Instr->getName() + ".zext");
    if (auto *BinOp = dyn_cast<BinaryOperator>(BO))
        if (BinOp->getOpcode() == Instruction::UDiv)
```

(continues on next page)

(continued from previous page)

```
            BinOp->setIsExact(Instr->isExact());

    Instr->replaceAllUsesWith(Zext);
    Instr->eraseFromParent();
    return true;
}
```

该函数的代码实现很直观，对于 `udiv` instruction 或 `urem` intructsion，寻找能容纳其操作数的最小位宽度，然后将原操作数替换为新的位宽度的操作数。

举例说明：

```
define void @test1(i32 %n) {
entry:
  %cmp = icmp ule i32 %n, 65535
  br i1 %cmp, label %bb, label %exit

bb:
  %div = udiv i32 %n, 100
  br label %exit

exit:
  ret void
}
```

对于 `%div = udiv i32 %n, 100` 这条 `udiv` instruction，其第一个操作数 `%n` 是小于 65535 的，而第二个操作数是 100，因为完全可以用 16 位的整数来表示这两个操作数，所以上述 LLVM IR 经过优化后，如下所示：

```
define void @test1(i32 %n) {
entry:
  %cmp = icmp ule i32 %n, 65535
  br i1 %cmp, label %bb, label %exit

bb:                                               ; preds = %entry
  %div.lhs.trunc = trunc i32 %n to i16
  %div1 = udiv i16 %div.lhs.trunc, 100
  %div.zext = zext i16 %div1 to i32
  br label %exit

exit:                                             ; preds = %bb, %entry
  ret void
}
```

processSRem()

关于 srem instruction，http://llvm.org/docs/LangRef.html#srem-instruction

对于 srem instruction 会调用该函数 processSRem() 尝试对指令优化：

```
static bool processSRem(BinaryOperator *SDI, LazyValueInfo *LVI)
{
    if (SDI->getType()->isVectorTy() || !hasPositiveOperands(SDI, LVI))
        return false;

    ++NumSRems;
    auto *BO = BinaryOperator::CreateURem(
        SDI->getOperand(0), SDI->getOperand(1), SDI->getName(), SDI);
    BO->setDebugLoc(SDI->getDebugLoc());
    SDI->replaceAllUsesWith(BO);
    SDI->eraseFromParent();

    // Try to process our new urem.
    processUDivOrURem(BO, LVI);

    return true;
}
```

如果 srem instruction 的操作数是 vector of integer values 或者 hasPositiveOperands() 函数返回 false，那么直接退出函数。

hasPositiveOperands() 函数通过 LazyValueInfo 来分析 srem 的操作数，如果能确定所有操作数的取值只能是正数，那么返回 true，否则返回 false：

```
static bool hasPositiveOperands(BinaryOperator *SDI, LazyValueInfo *LVI)
{
    Constant *Zero = ConstantInt::get(SDI->getType(), 0);
    for (Value *O : SDI->operands())
    {
        auto Result = LVI->getPredicateAt(ICmpInst::ICMP_SGE, O, Zero, SDI);
        if (Result != LazyValueInfo::True)
            return false;
    }
    return true;
}
```

也就是说，如果我们能够确定 srem instruction 的所有操作数都只能是正整数，那么我们就用一条有着相同操作数的 urem instruction 来替换掉这条 srem instruction 。然后再对这条新的 urem instruction 调用 processUDivOrURem() 函数进行处理。

举例说明：

```
define void @h(i32* nocapture %p, i32 %x){
entry:
  %cmp = icmp sgt i32 %x, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:
  %rem2 = srem i32 %x, 10
  store i32 %rem2, i32* %p, align 4
  br label %if.end

if.end:
  ret void
}
```

对于 `%rem2 = srem i32 %x, 10` 这条 `srem` 指令来说，第一个操作数 `i32 %x` 肯定是大于零的（即正整数），第二个操作数 10 也是正整数，所以该条指令就可以用对应的 `urem` 来替换，所以上述 LLVM IR 经过 `processSRem()` 函数处理后，如下所示：

```
define void @h(i32* nocapture %p, i32 %x){
entry:
  %cmp = icmp sgt i32 %x, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %rem21 = urem i32 %x, 10
  store i32 %rem21, i32* %p, align 4
  br label %if.end

if.end:                                           ; preds = %if.then, %entry
  ret void
}
```

processSDiv()

关于 `sdiv` instruction，http://llvm.org/docs/LangRef.html#sdiv-instruction

对于 `sdiv` instruction 会调用该函数 `processSDiv()` 尝试对指令优化，该 `processSDiv()` 函数的内容与 `processSRem()` 函数基本一致，略。

```
/// See if LazyValueInfo's ability to exploit edge conditions or range
/// information is sufficient to prove the both operands of this SDiv are
/// positive.  If this is the case, replace the SDiv with a UDiv. Even for local
/// conditions, this can sometimes prove conditions instcombine can't by
```

(continues on next page)

(continued from previous page)

```
/// exploiting range information.
static bool processSDiv(BinaryOperator *SDI, LazyValueInfo *LVI)
{
    if (SDI->getType()->isVectorTy() || !hasPositiveOperands(SDI, LVI))
        return false;

    ++NumSDivs;
    auto *BO = BinaryOperator::CreateUDiv(
        SDI->getOperand(0), SDI->getOperand(1), SDI->getName(), SDI);
    BO->setDebugLoc(SDI->getDebugLoc());
    BO->setIsExact(SDI->isExact());
    SDI->replaceAllUsesWith(BO);
    SDI->eraseFromParent();

    // Try to simplify our new udiv.
    processUDivOrURem(BO, LVI);

    return true;
}
```

processAShr()

关于 `ashr` instruction，http://llvm.org/docs/LangRef.html#ashr-instruction

对于 `ashr` instruction 会调用该函数 `processAShr()` 尝试对指令优化：

```
static bool processAShr(BinaryOperator *SDI, LazyValueInfo *LVI)
{
    if (SDI->getType()->isVectorTy())
        return false;

    Constant *Zero = ConstantInt::get(SDI->getType(), 0);
    if (LVI->getPredicateAt(ICmpInst::ICMP_SGE, SDI->getOperand(0), Zero,
                            SDI) != LazyValueInfo::True)
        return false;

    ++NumAShrs;
    auto *BO = BinaryOperator::CreateLShr(
        SDI->getOperand(0), SDI->getOperand(1), SDI->getName(), SDI);
    BO->setDebugLoc(SDI->getDebugLoc());
    BO->setIsExact(SDI->isExact());
    SDI->replaceAllUsesWith(BO);
    SDI->eraseFromParent();
```

(continues on next page)

(continued from previous page)

```
    return true;
}
```

对于 `ashr` instruction 的第一个操作数，如果通过 LazyValueInfo 可以知道它的取值一定是大于等于 0 的，那么就将该 `ashr` instruction 用有着相同操作数的 `lshr` 来替换。

举例说明：

```
define void @test1(i32 %n) {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.body, %entry
  %a = phi i32 [ %n, %entry ], [ %shr, %for.body ]
  %cmp = icmp sgt i32 %a, 1
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %shr = ashr i32 %a, 5
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret void
}
```

经过 `processAShr()` 优化后，上述 LLVM IR 变为如下所示：

```
define void @test1(i32 %n) {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.body, %entry
  %a = phi i32 [ %n, %entry ], [ %shr1, %for.body ]
  %cmp = icmp sgt i32 %a, 1
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %shr1 = lshr i32 %a, 5
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret void
}
```

processBinOp()

关于 `add` instruction，http://llvm.org/docs/LangRef.html#add-instruction

关于 `sub` intructsion，http://llvm.org/docs/LangRef.html#sub-instruction

对于 `add` instruction 和 `sub` intructsion 会调用该函数 `processBinOp()` 尝试对指令优化：

```
static bool processBinOp(BinaryOperator *BinOp, LazyValueInfo *LVI)
{
    using OBO = OverflowingBinaryOperator;

    if (DontAddNoWrapFlags)
        return false;

    if (BinOp->getType()->isVectorTy())
        return false;

    bool NSW = BinOp->hasNoSignedWrap();
    bool NUW = BinOp->hasNoUnsignedWrap();
    if (NSW && NUW)
        return false;

    BasicBlock *BB = BinOp->getParent();

    Value *LHS = BinOp->getOperand(0);
    Value *RHS = BinOp->getOperand(1);

    ConstantRange LRange = LVI->getConstantRange(LHS, BB, BinOp);
    ConstantRange RRange = LVI->getConstantRange(RHS, BB, BinOp);

    bool Changed = false;
    if (!NUW)
    {
        ConstantRange NUWRange = ConstantRange::makeGuaranteedNoWrapRegion(
            BinOp->getOpcode(), RRange, OBO::NoUnsignedWrap);
        bool NewNUW = NUWRange.contains(LRange);
        BinOp->setHasNoUnsignedWrap(NewNUW);
        Changed |= NewNUW;
    }
    if (!NSW)
    {
        ConstantRange NSWRange = ConstantRange::makeGuaranteedNoWrapRegion(
            BinOp->getOpcode(), RRange, OBO::NoSignedWrap);
        bool NewNSW = NSWRange.contains(LRange);
```

(continues on next page)

(continued from previous page)

```
        BinOp->setHasNoSignedWrap(NewNSW);
        Changed |= NewNSW;
    }

    return Changed;
}
```

对于 add instruction 或 sub instruction，如果既有 nsw keyword 又有 nuw keyword，那么就直接退出函数。然后通过 LazyValueInfo 获取左操作数和右操作数的取值范围，LRange 和 RRange 。如果 instruction 没有 nuw keyword，那么就根据右操作数的取值范围 RRange，通过 ConstantRange::makeGuaranteedNoWrapRegion() 函数获取一定不会使无符号运算的结果发生回绕的左操作数的最大取值范围，如果 LRange 在这个范围内的话，就为该 instruction 设置 nuw keyword 。如果 instruction 没有 nSw keyword，那么就根据右操作数的取值范围 RRange，通过 ConstantRange::makeGuaranteedNoWrapRegion() 函数获取一定不会使有符号运算的结果发生回绕的左操作数的最大取值范围，如果 LRange 在这个范围内的话，就为该 instruction 设置 nsw keyword 。

举例说明：

```
define void @test0(i32 %a) {
entry:
  %cmp = icmp slt i32 %a, 100
  br i1 %cmp, label %bb, label %exit

bb:
  %add = add i32 %a, 1
  br label %exit

exit:
  ret void
}
```

对于 %add = add i32 %a, 1 这条 add instruction，通过 getConstantRange() 函数，我们得到 LRange = [-2147483648, 100)，RRange = [1, 2)，通过 ConstantRange::makeGuaranteedNoWrapRegion(BinOp->getOpcode(), RRange, OBO::NoUnsignedWrap) 函数，我们得到 NUWRange = [0, -1)，所以 NUWRange.contains(LRange) 为 false，不为该 add instruction 添加 nuw keyword，通过 ConstantRange::makeGuaranteedNoWrapRegion(BinOp->getOpcode(), RRange, OBO::NoSignedWrap) 函数，我们得到 NSWRange= [-2147483648, 2147483647)，所以 NSWRange.containts(LRange) 为 true，为该 add instruction 添加 nsw keyword 。

上述 LLVM IR 经过 processBinOp() 处理后，变为如下所示：

```
define void @test0(i32 %a) {
entry:
```

(continues on next page)

(continued from previous page)

```
  %cmp = icmp slt i32 %a, 100
  br i1 %cmp, label %bb, label %exit

bb:                                               ; preds = %entry
  %add = add nsw i32 %a, 1
  br label %exit

exit:                                             ; preds = %bb, %entry
  ret void
}
```

Summary

`CorrelatedValuePropagation` 这个 pass 的代码实现是很清晰明了的，而且“演示”了 LazyValueInfo 的用法，非常适合新手进行学习。

- genindex
- modindex
- search

5.4 SLP Vectorizer

5.4.1 SLP Vectorizer

本文是论文《Exploiting Superword Level Parallelism with Multimedia Instruction Sets》的阅读笔记。论文提出一种称作 SLP 的向量化技术，作者是 Samuel Larsen 和 Saman Amarasinghe，发表在 PLDI’2000。

About SLP (Superword Level Parallelism)

SLP 即 Superword Level Parallelism，是自动向量化技术的一种（另一种是 Loop vectorizer）。SLP vectorization 的目标是将多条 independent isomorphic 指令组合成一条向量化指令。

例如，图 1 中的四条语句对应位置的操作数都可以 pack 到一个向量寄存器 (vector register) 中（b, e, s, x 被 pack 到一个向量寄存器中，c, f, t, y 被 pack 到一个向量寄存器中，z[i+0], z[i+1], z[i+2], z[i+3] 被 pack 到一个向量寄存器中），然后就可以通过 SIMD 指令并行执行这四条语句。

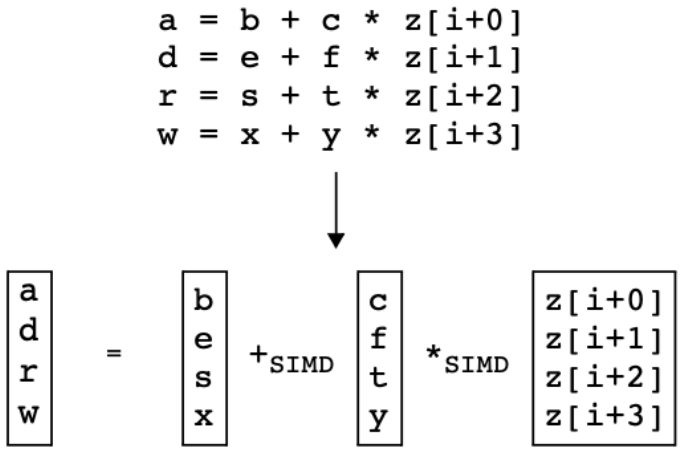


Figure 1: Isomorphic statements that can be packed and executed in parallel.

因为通过 SIMD 指令并行执行得到的结果也是在向量寄存器中的，所以根据 a, d, r, w 的（被）使用方式，可能还需要将 a, d, r, w 从向量寄存器中 load 出来。该操作称为 unpack。

如果并行执行的时间开销 + packing 操作数时间开销 + unpacking 操作数时间开销 **小于**原本执行的时间开销，就说明 SLP vectorization 有性能收益。

总结一下 SLP：

- Generally applicable: SLP is not restricted on parallelism of loops
- Find independent isomorphic instructions within basic block
- Goal
    1. Gain more speed up via parallelism
    2. Minimize the cost of packing and unpacking
- Prefer operating on adjacent memory, whose cost of packing is minimum

Compared To Previous Approach

在作者撰写论文时，向量编译器 (vector compilers) 通常以循环为目标来寻找 vector parallelism 的机会，因为循环天然提供了对多个数据执行相同指令的机会。向量编译器通过 loop transformations 将一段代码转换为可以被向量化的形式 (vectorizable form)。

例如，如下循环：

```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

应用 scalar expansion 和 loop fission 后就被转换为了可以被向量化的形式：

```
for (i=0; i<16; i++) {
  T[i] = ref[i] - curr[i];
}
for (i=0; i<16; i++) {
  diff += abs(T[i]);
}
```

注意：应用了 scalar expansion 和 loop fission 后的代码，只有第一个循环是可以通过 SIMD 指令一次执行多次减法操作的，第二个循环则不能。

SLP 同样能够对上循环进行向量化，并且是以一个完全不同的角度：

如下循环：

```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

经过 loop unroll 和 rename 后得到：

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  diff += abs(localdiff0);

  localdiff1 = ref[i+1] - curr[i+1];
  diff += abs(localdiff1);

  localdiff2 = ref[i+2] - curr[i+2];
  diff += abs(localdiff2);

  localdiff3 = ref[i+3] - curr[i+3];
  diff += abs(localdiff3);
}
```

这样 SLP 就能够将计算 localdiff{0, 1, 2, 3} 的这四条 independent isomorphic 指令组合成一条向量化指令 (SIMD-):

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff += abs(localdiff0);
  diff += abs(localdiff1);
  diff += abs(localdiff2);
  diff += abs(localdiff3);
}
```

但是对于如下的代码片段，向量编译器 (vector compilers) 要想向量化该循环，需要将 do while 循环转换为 for 循环，恢复归纳变量 (induction variable)，将展开后的循环恢复为未展开的形式 (loop rerolling)。而 SLP 向量化该循环则非常容易，直接将计算 dst[{0, 1, 2, 3}] 的这四条 independent isomorphic 语句组合成一条使用向量化指令的语句即可。

```
do {
  dst[0] = (src1[0] + src2[0]) >> 1;
  dst[1] = (src1[1] + src2[1]) >> 1;
  dst[2] = (src1[2] + src2[2]) >> 1;
  dst[3] = (src1[3] + src2[3]) >> 1;

  dst  += 4;
  src1 += 4;
  src2 += 4;
}
```

SLP Extraction Algorithm

作者提出了一种简单的算法，将具有 SLP 机会的基本块转换为使用 SIMD 指令的基本块。该算法寻找 independent（无数据依赖）、isomorphic（相同操作）的指令组合成一条向量化指令。

作者观察到 (observation)：

> Packed statements that containt adjacent memory references among corresponding operands are particularly well suited for SLP execution

即如果被 pack 的指令的操作数引用的是相邻的内存，那么则特别适合 SLP 执行。

所以 SLP Extraction Algorithm 的核心算法就是从识别 adjacent memory references 开始的。

在识别 adjacent memory references 开始之前实际上还有一些准备工作要做：

1. **Loop unrolling.** transform vector parallelism into basic blocks with superword level parallelism，见 *Compared To Previous Approach*

2. **Alignment analysis.** memory load, store, simd
3. **Pre-Optimization.** constant propagation, dead code elimination, common subexpression elimination, loop invariant code motion and redundant load/store elimination. 避免向量化不必要的代码（死代码、冗余代码）

SLP Extraction Algorithm 的核心算法如下：

$$
\begin{aligned}
&\text{SLP\_extract: BasicBlock } B \rightarrow \text{BasicBlock} \\
&\quad \text{PackSet } P \leftarrow \emptyset \\
&\quad P \leftarrow \text{find\_adj\_refs}(B, P) \\
&\quad P \leftarrow \text{extend\_packlist}(B, P) \\
&\quad P \leftarrow \text{combine\_packs}(P) \\
&\quad \textbf{return } \text{schedule}(B, [\,], P)
\end{aligned}
$$

主要分为以下 4 步：

1. Identifying Adjacent Memory References
2. Extending the PackSet
3. Combination
4. Scheduling

下面进行详细解释。

Identifying Adjacent Memory References

Identifying Adjacent Memory References 即 find_adj_refs，伪代码如下：

```
find_adj_refs: BasicBlock B × PackSet P → PackSet
    foreach Stmt s ∈ B do
      foreach Stmt s' ∈ B where s ≠ s' do
        if has_mem_ref(s) ∧ has_mem_ref(s') then
          if adjacent(s, s') then
            Int align ← get_alignment(s)
            if stmts_can_pack(B, P, s, s', align) then
              P ← P ∪ {⟨s, s'⟩}
    return P
```

find_adj_refs 的输入是 BasicBlock，输出为集合 PackSet。

对于 BasicBlock 中的任意语句对 <s, s'>，如果语句 s 和 s' 访问了相邻的内存（如，s 访问了 array[1]，s' 访问了 array[2]），并且语句 s 和 s' 能 pack 到一起（函数 stmts_can_pack 返回 true ），那么就将语句对 <s, s'> 加入集合 PackSet 中。

函数 stmts_can_pack 的伪代码如下：

```
stmts_can_pack: BasicBlock B × PackSet P ×
                Stmt s × Stmt s' × Int align → Boolean
    if isomorphic(s, s') then
      if independent(s, s') then
        if ∀⟨t, t'⟩ ∈ P.t ≠ s then
          if ∀⟨t, t'⟩ ∈ P.t' ≠ s' then
            Int align_s ← get_alignment(s)
            Int align_s' ← get_alignment(s')
            if align_s ≡ ⊤ ∨ align_s ≡ align then
              if align_s' ≡ ⊤ ∨ align_s' ≡ align+data_size(s') then
                return true
    return false
```

即，如果两条语句 s 和 s' 满足如下条件，那么语句 s 和 s' 就能 pack 到一起：

- s 和 s' 是相同操作 (isomorphic)
- s 和 s' 无数据依赖 (independent)
- s 之前没有作为左操作数出现在 PackSet 中，s' 之前没有作为右操作数出现在 PackSet 中

- s 和 s’ 的满足对齐要求 (consistent)

find_adj_refs 执行结束后，我们就得到了集合 PackSet，PackSet 中元素是 <s, s’ > 这样的语句对。

Extending the PackSet

在 find_adj_refs 我们构建了 PackSet 集合，在这一步中我们沿着被 pack 的语句的 defs 和 uses 来扩充 PackSet 集合。extent_packlist 的输入集合 PackSet，输出为集合 PackSet。

extent_packlist 的伪代码如下：

extend_packlist: BasicBlock B × PackSet P → PackSet
  repeat
    PackSet $P_{prev} \leftarrow P$
    foreach Pack $p \in P$ **do**
      $P \leftarrow$ follow_use_defs(B, P, p)
      $P \leftarrow$ follow_def_uses(B, P, p)
  until $P \equiv P_{prev}$
  return P

对 Packet 中每一个元素 Pack，执行函数 follow_use_defs 和 follow_def_uses 扩充 PackSet 集合，不断扩充直至 PackSet 不能再加入新的 Pack。

先看 follow_use_defs：

follow_use_defs: BasicBlock B × PackSet P × Pack p → PackSet
    where $p = \langle s, s' \rangle$, $s = [\,\mathtt{x_0} := \mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_m})\,]$, $s' = [\,\mathtt{x'_0} := \mathtt{f}(\mathtt{x'_1}, \ldots, \mathtt{x'_m})\,]$
  Int $align \leftarrow$ get_alignment(s)
  for $j \leftarrow 1$ **to** m **do**
    if $\exists t \in B.t = [\,\mathtt{x_j} := \ldots\,] \wedge \exists t' \in B.t' = [\,\mathtt{x'_j} := \ldots\,]$ **then**
      if stmts_can_pack$(B, P, t, t', align)$
        if est_savings $(\langle t, t' \rangle, P) \geq 0$ **then**
          $P \leftarrow P \cup \{\langle t, t' \rangle\}$
          set_alignment$(s, s', align)$
  return P

对于一个 Pack，即语句对 <s, s’ >：考察 s 和 s’ 的每一对源操作数 xj 和 xj’，如果 s 和 s’ 所在 BasicBlock 中存在对 xj 和 xj’ 定值 (def) 的语句 t 和 t’，语句 t 和 t’ 还能 pack 到一起（函数 stmts_can_pack 返回 true ），并且根据 cost model，将 <t, t’ > 加入 PackSet 中有收益，那么就将 <t, t’ > 加入集合 PackSet 中。

再看 follow_def_uses：

follow_def_uses: BasicBlock B × PackSet P × Pack p → PackSet
where $p = \langle s, s' \rangle$, $s = [\, \mathtt{x_0} := \mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_m}) \,]$, $s' = [\, \mathtt{x'_0} := \mathtt{f}(\mathtt{x'_1}, \ldots, \mathtt{x'_m}) \,]$
Int $align \leftarrow$ get_alignment(s)
Int $savings \leftarrow -1$
foreach Stmt $t \in B$ **where** $t = [\, \ldots := \mathtt{g}(\ldots, \mathtt{x_0}, \ldots) \,]$ **do**
  foreach Stmt $t' \in B$ **where** $t \neq t' = [\, \ldots := \mathtt{h}(\ldots, \mathtt{x'_0}, \ldots) \,]$ **do**
    if stmts_can_pack($B, P, t, t', align$) **then**
      if est_savings($\langle t, t' \rangle, P$) $> savings$ **then**
        $savings \leftarrow$ est_savings($\langle t, t' \rangle, P$)
        Stmt $u \leftarrow t$
        Stmt $u' \leftarrow t'$
if $savings \geq 0$ **then**
  $P \leftarrow P \cup \{\langle u, u' \rangle\}$
  set_alignment(u, u')
return P

对于一个 Pack，即语句对 <s, s’>：考察 s 和 s’的目的操作数 x0 和 x0’，如果 s 和 s’所在 BasicBlock 中存在使用 (use) x0 和 x0’的语句 t 和 t’，语句 t 和 t’还能 pack 到一起（函数 stmts_can_pack 返回 true），根据 cost model，找到将 <t, t’> 加入 PackSet 后获得收益最大的使用 (use) x0 和 x0’的语句 u 和 u’（存在多个使用 x0 和 x0’的语句 t 和 t’），将 <u, u’> 加入集合 PackSet 中。

extent_packlist 执行结束后，我们就扩充了集合 PackSet，PackSet 中元素是 <s, s’> 这样的语句对。

Combination

在 find_adj_refs 我们构建了 PackSet 集合，在 extent_packlist 中我们扩充了 PackSet 集合。此时 PackSet 中元素是 <s, s’> 这样的语句对。

这一步我们对 PackSet 中的语句对进行合并，combine_packs 的输入集合 PackSet，输出为集合 PackSet。伪代码如下：

combine_packs: PackSet P → PackSet
repeat
  PackSet $P_{prev} \leftarrow P$
  foreach Pack $p = \langle s_1, \ldots, s_n \rangle \in P$ **do**
    foreach Pack $p' = \langle s'_1, \ldots, s'_m \rangle \in P$ **do**
      if $s_n \equiv s'_1$ **then**
        $P \leftarrow P - \{p, p'\} \cup \{\langle s_1, \ldots, s_n, s'_2, \ldots, s'_m \rangle\}$
until $P \equiv P_{prev}$
return P

对于 PackSet 中的任意两个 Pack，p = <s1, ⋯, sn>，p’ = <s1’, ⋯, sm’>，如果 p 的最后一个语句和 p’的第

一个语句是同一个语句，那么就将 p 和 p’ 合并。

这一步很容易理解。combine_packs 执行结束后，PackSet 中元素是 <s, ···, sn> 这样的语句 n 元组，n >= 2。

Scheduling

最后一步对基本块中的指令进行调度，生成包含 SIMD 指令的基本块。对于 PackSet 中的一个 Pack（Pack 是语句 n 元组），Pack 可能依赖于之前定义，因此我们需要按照数据依赖图的拓扑顺序生成指令。如果存在循环依赖，我们 revert 导致循环的 Pack 不在对该 Pack 使用 SIMD 指令。

$$
\begin{array}{l}
\text{schedule: } \text{BasicBlock } B \times \text{BasicBlock } B' \times \text{PackSet } P \\
\qquad \rightarrow \text{BasicBlock} \\
\quad \textbf{for } i \leftarrow 0 \textbf{ to } |B| \textbf{ do} \\
\quad\quad \textbf{if } \exists p = \langle \ldots, s_i, \ldots \rangle \in P \textbf{ then} \\
\quad\quad\quad \textbf{if } \forall s \in p.\ \text{deps\_scheduled}(s, B') \textbf{ then} \\
\quad\quad\quad\quad \textbf{foreach } \text{Stmt } s \in p \textbf{ do} \\
\quad\quad\quad\quad\quad B \leftarrow B - s \\
\quad\quad\quad\quad\quad B' \leftarrow B' \cdot s \\
\quad\quad\quad\quad \textbf{return } \text{schedule}(B, B', P) \\
\quad\quad \textbf{else if } \text{deps\_scheduled}(s_i, B') \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{schedule}(B - s_i, B' \cdot s_i, P) \\
\quad \textbf{if } |B| \neq 0 \textbf{ then} \\
\quad\quad P \leftarrow P - \{p\} \textbf{ where } p = \text{first}(B, P) \\
\quad\quad \textbf{return } \text{schedule}(B, B', P) \\
\quad \textbf{return } B'
\end{array}
$$

Scheduling 这一步输入是原本的 BasicBlock 和 PackSet，输出是包含 SIMD 指令的 BasicBlock。

Example

这里我们用论文中的例子来理解一下整个算法的流程：

1. 初始状态，BasicBlock 中包含的指令序列如 (a) 所示。
2. 执行 find_adj_refs，我们发现语句 (1) 和语句 (4) 访问的分别是 a[i+0] 和 a[i+1]，并且满足 stmts_can_pack，所以将 <(1), (4)> 加入到 PackSet 中。语句 (4) 和语句 (7) 访问的分别是 a[i+1] 和 a[i+2]，语句 (4) 和语句 (7) 是 independent 和 isomorphic 的，并且语句 (4) 没有作为 Pack 的左操作数出现在 Pack 中（<(1), (4)> 中语句 (4) 是作为 Pack 的右操作数），语句 (7) 也没有作为 Pack 的右操作数出现在 Pack 中，且语句 (4)

和语句 (7) 满足对齐要求，所以再将 <(4), (7)> 加入到 PackSet 中。find_adj_refs 执行结束，此时 PackSet 内容为 {<(1), (4)>, <(4), (7)>}

3. 执行 extent_packlist：
    1. follow_use_defs，在 BasicBlock 中没有对 a[i+0], a[i+1], a[i+2] 进行 def 的语句，所以第一次 follow_use_defs 没有改变 PackSet。
    2. follow_def_uses，这一次将 (3) 和 (6)、(6) 和 (9) 加入到 PackSet 中，分别是根据 (1) 和 (4)、(4) 和 (7) follow_def_uses 得到的。
    3. 再一次执行 follow_use_defs，这一次将对 (3) 和 (6) 中定值 c 和 f 的语句 (2) 和 (5) 加入到 PackSet 中，将对 (6) 和 (9) 中定值 f 和 j 的语句 (5) 和 (8) 加入到 PackSet 中。
    4. 再一次执行 follow_use_defs，发现没有新的 Pack 能加入到 PackSet 中了，extent_packlist 执行结束。
4. 执行 combine_packs：
    1. <(1), (4)> 和 <(4), (7)> 合并为 <(1), (4), (7)>
    2. <(3), (6)> 和 <(6), (9)> 合并为 <(3), (6), (9)>
    3. <(2), (5)> 和 <(5), (8)> 合并为 <(2), (5), (8)>
5. 执行 scheduling，注意 (3) 依赖 (1) 和 (2)，(6) 依赖 (4) 和 (5)，(9) 依赖 (7) 和 (8)。

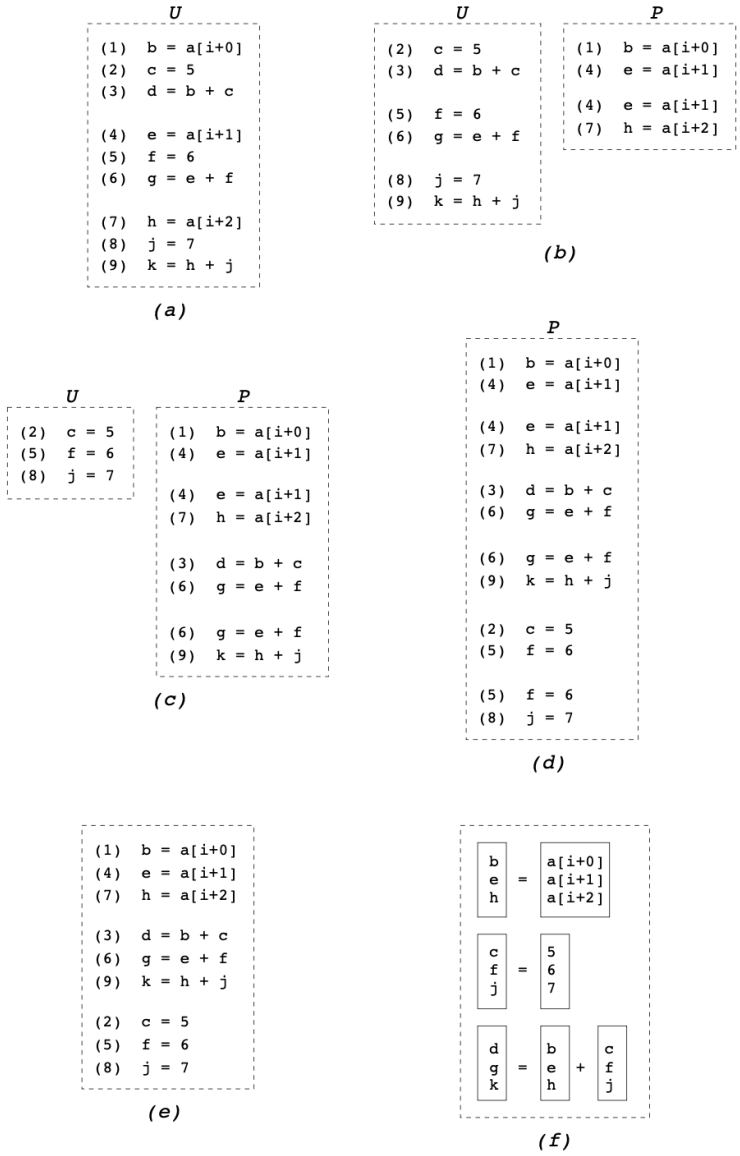


Figure 4: Various phases of SLP analysis. U and P represent the current set of unpacked and packed statements, respectively. (a) Initial sequence of instructions. (b) Statements with adjacent memory references are paired and added to the *PackSet*. (c) The *PackSet* is extended by following def-use chains of existing entries. (d) The *PackSet* is further extended by following use-def chains. (e) *Combination* merges groups containing the same expression. (f) Each group is scheduled and SIMD operations are emitted in their place.

Implementation

LLVM 实现 了 SLP vectorization 算法，是基于 “Loop-Aware SLP in GCC” by Ira Rosen, Dorit Nuzman, Ayal Zaks. 这篇论文。

接下来会先阅读 Loop-Aware SLP in GCC 这篇论文写一下阅读笔记，然后再学习 LLVM 的实现，写一下源码阅读笔记。

References

1. http://groups.csail.mit.edu/cag/slp/
2. https://www.cs.cornell.edu/courses/cs6120/2020fa/blog/slp/

- genindex
- modindex
- search

- genindex
- modindex
- search

CHAPTER

SIX

LINK TIME OPTIMIZATION

6.1 LTO Remove Dead Symbol

Link-time optimization (LTO)，顾名思义，编译器在链接时对程序执行的一种程序优化。对于像 C 这样语言，编译是逐个编译单元去编译的，然后通过链接器将这些编译单元链接在一起，LTO 就是在将这些编译单元链接在一起时执行的 intermodular optimization。

6.1.1 Remove Dead Symbol

在 LTO 阶段可以完成很多编译时无法做到的优化，如：在链接产物中删掉不会用到的死函数。

本文是对 LTO remove dead symbol 源码实现的阅读笔记（源码阅读基于 llvm 13.0.0 版本）。

Example

首先，我们举个例子，尝试一下 LTO remove dead symbol。

给定以下源文件：

```
--- tu1.c ---
int unused(int a);
int probably_inlined(int a);
int main(int argc, const char *argv[]) {
  return probably_inlined(argc);
}

--- tu2.c ---
int unused(int a) {
  return a + 1;
}
int probably_inlined(int a) {
  return a + 2;
}
```

编译 tu1.c 和 tu2.c 得到 tu1.o 和 tu2.o（通过选项 `-flto` 来开启 LTO）

```
% clang -flto -c tu1.c -o tu1.o
% clang -flto -c tu2.c -o tu2.o
```

链接 a.o 和 main.o 得到可执行文件 main（通过选项 `-fuse-ld=lld` 指定使用 lld linker）

```
% clang -flto -fuse-ld=lld tu1.o tu2.o -o main
```

可以通过 `readelf -sW ./main | awk '$4 == "FUNC"'` 查看生成的可执行文件的符号表中都有哪些函数：

```
% readelf -sW ./main | awk '$4 == "FUNC"'
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.
↪2.5 (2)
     3: 00000000002015e0     0 FUNC    LOCAL  DEFAULT   12 deregister_tm_clones
     4: 0000000000201610     0 FUNC    LOCAL  DEFAULT   12 register_tm_clones
     5: 0000000000201650     0 FUNC    LOCAL  DEFAULT   12 __do_global_dtors_aux
     8: 0000000000201680     0 FUNC    LOCAL  DEFAULT   12 frame_dummy
    15: 0000000000201740    15 FUNC    LOCAL  DEFAULT   12 probably_inlined
    16: 00000000002015d0     2 FUNC    LOCAL  HIDDEN    12 _dl_relocate_static_pie
    21: 0000000000201700     2 FUNC    GLOBAL DEFAULT   12 __libc_csu_fini
    22: 00000000002015a0    43 FUNC    GLOBAL DEFAULT   12 _start
    23: 0000000000201690   101 FUNC    GLOBAL DEFAULT   12 __libc_csu_init
    24: 0000000000201710    36 FUNC    GLOBAL DEFAULT   12 main
    27: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main
    30: 0000000000201750     0 FUNC    GLOBAL DEFAULT   13 _init
    31: 0000000000201768     0 FUNC    GLOBAL DEFAULT   14 _fini
```

可以看到，有 `main()` 函数，`probably_inlined()` 函数，没有了 `unused()` 函数。因为虽然 `unused()` 函数在 tu2.c 中定义了，但是实际上并没有它并没有被调用，所以该函数是个死函数，所以在 LTO 时会被删除。

我们可以再看一下，不开启 LTO 时编译 tu1.c 和 tu2.c 得到可执行文件 main.nonlto：

```
% clang -fuse-ld=lld tu1.c tu2.c -o main.nonlto
% readelf -sW ./main.nonlto | awk '$4 == "FUNC"'
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.
↪2.5 (2)
     3: 00000000002015f0     0 FUNC    LOCAL  DEFAULT   12 deregister_tm_clones
     4: 0000000000201620     0 FUNC    LOCAL  DEFAULT   12 register_tm_clones
     5: 0000000000201660     0 FUNC    LOCAL  DEFAULT   12 __do_global_dtors_aux
     8: 0000000000201690     0 FUNC    LOCAL  DEFAULT   12 frame_dummy
    16: 00000000002015e0     2 FUNC    LOCAL  HIDDEN    12 _dl_relocate_static_pie
    21: 0000000000201740     2 FUNC    GLOBAL DEFAULT   12 __libc_csu_fini
    22: 00000000002015b0    43 FUNC    GLOBAL DEFAULT   12 _start
```

(continues on next page)

(continued from previous page)

```
    23: 00000000002016d0   101 FUNC    GLOBAL DEFAULT   12 __libc_csu_init
    24: 00000000002016a0     5 FUNC    GLOBAL DEFAULT   12 main
    27: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main
    30: 0000000000201744     0 FUNC    GLOBAL DEFAULT   13 _init
    31: 000000000020175c     0 FUNC    GLOBAL DEFAULT   14 _fini
    34: 00000000002016c0     4 FUNC    GLOBAL DEFAULT   12 probably_inlined
    35: 00000000002016b0     4 FUNC    GLOBAL DEFAULT   12 unused
```

可以看到 `unused()` 函数被保留在了最终的可执行文件中。

通过这个例子，我们看到了 LTO 可以在链接时 remove dead symbol。

实际上，如果我们还可以通过 optimization remarks 得到在 LTO 优化时都删除了哪些函数：

```
% clang -flto -fuse-ld=lld -Wl,--opt-remarks-passes,lto -Wl,--opt-remarks-filename,
↪main.lto.yaml tu1.c tu2.c -o main
```

这里我们只保留了与 lto 相关的 optimization remarks，默认生成的 optimization remarks 是 YAML 格式文件，该文件 main.lto.yaml 的内容如下：

```
--- !Passed
Pass:            lto
Name:            deadfunction
Function:        unused
Args:
  - Function:        unused
  - String:          ' not added to the combined module '
...
```

从 main.lto.yam 文件的内容也可以看出来 `unused()` 函数在 lto 优化阶段被删除掉了。

Inside the source code

下面我们了解一下 LTO remove dead symbol 的代码实现。

这里给出使用 lld 作为 linker，链接过程执行到 remove dead symbol 所经过的函数：

```
=> void LinkerDriver::linkerMain(ArrayRef<const char *> argsArr) at lld\ELF\Driver.
↪cpp:475
===> void LinkerDriver::link(opt::InputArgList &args) at lld\ELF\Driver.cpp:2165
=====> void LinkerDriver::compileBitcodeFiles() at lld\ELF\Driver.cpp:1979
=======> std::vector<InputFile *> BitcodeCompiler::compile() at lld\ELF\LTO.cpp:299
=========> Error LTO::run(AddStreamFn AddStream, NativeObjectCache Cache) at llvm\lib\
↪LTO\LTO.cpp:995
===========> void llvm::computeDeadSymbolsWithConstProp(...) at llvm\lib\Transforms\
```

(continues on next page)

(continued from previous page)

```
↪IPO\FunctionImport.cpp:956
===========> Error LTO::runRegularLTO(AddStreamFn AddStream) at llvm\lib\LTO\LTO.
↪cpp:1043
=============> Error LTO::linkRegularLTO(RegularLTOState::AddedModule Mod, bool␣
↪LivenessFromIndex) at llvm\lib\LTO\LTO.cpp:853
```

根据函数名也可以看出，计算 dead symbol 的核心函数就是 `void llvm::computeDeadSymbolsWithConstProp(...)`，实现如下：

```
llvm-project\llvm\lib\Transforms\IPO\FunctionImport.cpp:955
955: // Compute dead symbols and propagate constants in combined index.
956: void llvm::computeDeadSymbolsWithConstProp(
957:     ModuleSummaryIndex &Index,
958:     const DenseSet<GlobalValue::GUID> &GUIDPreservedSymbols,
959:     function_ref<PrevailingType(GlobalValue::GUID)> isPrevailing,
960:     bool ImportEnabled) {
961:   computeDeadSymbols(Index, GUIDPreservedSymbols, isPrevailing);
962:   if (ImportEnabled)
963:     Index.propagateAttributes(GUIDPreservedSymbols);
964: }
```

函数 `computeDeadSymbols()` 的实现如下：

核心算法就是不动点的计算：将 GUIDPreservedSymbols 对应的 retained symbol 标记为 live，作为 worklist 的初始值。然后不断遍历 worklist 中的每一个 symbol，将该 symbol 引用的其他 symbol 标记为 live 的，加入到 worklist 中。一直迭代，直至没有新的被标记为 live 的 symbol。

在函数 `computeDeadSymbols()` 实现该 worklist 算法时，是用类似栈的方式处理的：将新标记为 live 的 symbol 入栈，然后不断处理栈顶的 symbol，该栈顶 symbol 出栈，将该 symbol 引用的其他之前没有添加过 worklist 中的 symbol 标记为 live 的，加入到栈顶。一直迭代，直至栈为空。

```
llvm-project\llvm\lib\Transforms\IPO\FunctionImport.cpp:842
842: void llvm::computeDeadSymbols(
843:     ModuleSummaryIndex &Index,
844:     const DenseSet<GlobalValue::GUID> &GUIDPreservedSymbols,
845:     function_ref<PrevailingType(GlobalValue::GUID)> isPrevailing) {
846:   assert(!Index.withGlobalValueDeadStripping());
847:   if (!ComputeDead)
848:     return;
849:   if (GUIDPreservedSymbols.empty())
850:     // Don't do anything when nothing is live, this is friendly with tests.
851:     return;
852:   unsigned LiveSymbols = 0;
853:   SmallVector<ValueInfo, 128> Worklist;
```

(continues on next page)

(continued from previous page)

第 854 - 873 行初始化 worklist

```
854:   Worklist.reserve(GUIDPreservedSymbols.size() * 2);
855:   for (auto GUID : GUIDPreservedSymbols) {
856:     ValueInfo VI = Index.getValueInfo(GUID);
857:     if (!VI)
858:       continue;
859:     for (auto &S : VI.getSummaryList())
860:       S->setLive(true);
861:   }
862:
863:   // Add values flagged in the index as live roots to the worklist.
864:   for (const auto &Entry : Index) {
865:     auto VI = Index.getValueInfo(Entry);
866:     for (auto &S : Entry.second.SummaryList)
867:       if (S->isLive()) {
868:         LLVM_DEBUG(dbgs() << "Live root: " << VI << "\n");
869:         Worklist.push_back(VI);
870:         ++LiveSymbols;
871:         break;
872:       }
873:   }
874:
```

visit 判断当前处理的 symbol 是否在已经被标记为 live，即之前已经加过 worklist 中被处理过了。

如果没有，则将其标记为 live，然后添加到 worklist 中。

```
875:   // Make value live and add it to the worklist if it was not live before.
876:   auto visit = [&](ValueInfo VI, bool IsAliasee) {
877:     // FIXME: If we knew which edges were created for indirect call profiles,
878:     // we could skip them here. Any that are live should be reached via
879:     // other edges, e.g. reference edges. Otherwise, using a profile collected
880:     // on a slightly different binary might provoke preserving, importing
881:     // and ultimately promoting calls to functions not linked into this
882:     // binary, which increases the binary size unnecessarily. Note that
883:     // if this code changes, the importer needs to change so that edges
884:     // to functions marked dead are skipped.
885:     VI = updateValueInfoForIndirectCalls(Index, VI);
886:     if (!VI)
887:       return;
888:
889:     if (llvm::any_of(VI.getSummaryList(),
```

(continues on next page)

(continued from previous page)

```
890:                       [](const std::unique_ptr<llvm::GlobalValueSummary> &S) {
891:                         return S->isLive();
892:                       }))
893:       return;
894:
895:     // We only keep live symbols that are known to be non-prevailing if any are
896:     // available_externally, linkonceodr, weakodr. Those symbols are discarded
897:     // later in the EliminateAvailableExternally pass and setting them to
898:     // not-live could break downstreams users of liveness information (PR36483)
899:     // or limit optimization opportunities.
900:     if (isPrevailing(VI.getGUID()) == PrevailingType::No) {
901:       bool KeepAliveLinkage = false;
902:       bool Interposable = false;
903:       for (auto &S : VI.getSummaryList()) {
904:         if (S->linkage() == GlobalValue::AvailableExternallyLinkage ||
905:             S->linkage() == GlobalValue::WeakODRLinkage ||
906:             S->linkage() == GlobalValue::LinkOnceODRLinkage)
907:           KeepAliveLinkage = true;
908:         else if (GlobalValue::isInterposableLinkage(S->linkage()))
909:           Interposable = true;
910:       }
911:
912:       if (!IsAliasee) {
913:         if (!KeepAliveLinkage)
914:           return;
915:
916:         if (Interposable)
917:           report_fatal_error(
918:               "Interposable and available_externally/linkonce_odr/weak_odr "
919:               "symbol");
920:       }
921:     }
922:
923:     for (auto &S : VI.getSummaryList())
924:       S->setLive(true);
925:     ++LiveSymbols;
926:     Worklist.push_back(VI);
927:   };
928:
```

迭代直至 worklist 为空，即没有新的 symboal 被标记为 live，添加至 worklist 中

```
929:   while (!Worklist.empty()) {
930:     auto VI = Worklist.pop_back_val();
```

(continues on next page)

(continued from previous page)

```
931:     for (auto &Summary : VI.getSummaryList()) {
932:       if (auto *AS = dyn_cast<AliasSummary>(Summary.get())) {
933:         // If this is an alias, visit the aliasee VI to ensure that all copies
934:         // are marked live and it is added to the worklist for further
935:         // processing of its references.
936:         visit(AS->getAliaseeVI(), true);
937:         continue;
938:       }
939:       for (auto Ref : Summary->refs())
940:         visit(Ref, false);
941:       if (auto *FS = dyn_cast<FunctionSummary>(Summary.get()))
942:         for (auto Call : FS->calls())
943:           visit(Call.first, false);
944:     }
945:   }
946:   Index.setWithGlobalValueDeadStripping();
947:
948:   unsigned DeadSymbols = Index.size() - LiveSymbols;
949:   LLVM_DEBUG(dbgs() << LiveSymbols << " symbols Live, and " << DeadSymbols
950:                     << " symbols Dead \n");
951:   NumDeadSymbols += DeadSymbols;
952:   NumLiveSymbols += LiveSymbols;
953: }
```

这里再次用在 Example 节中的例子来分析该函数 `computeDeadSymbols()`：

1. 第 854 - 873 行初始化 Worklist，对于 Example 节中的例子来说，Worklist 中此时只有一个元素，就是 `main()` 函数对应的 ValueInfo

```
(gdb)
864       for (const auto &Entry : Index) {
(gdb)
927       };
(gdb) p Worklist.size()
$28 = 1
(gdb) p Worklist.begin()->name().str()
$29 = "main"
```

2. 第 929 - 945 行第一轮迭代：因为 `main()` 函数调用了 `probably_inlined()` 函数，所以会执行第 943 行：`visit(Call.first, false);` 此时 Call.first 就是 `probably_inlined()` 函数对应的 ValueInfo

```
929       while (!Worklist.empty()) {
(gdb)
```

(continues on next page)

(continued from previous page)

```
930         auto VI = Worklist.pop_back_val();
(gdb)
931         for (auto &Summary : VI.getSummaryList()) {
(gdb)
932           if (auto *AS = dyn_cast<AliasSummary>(Summary.get())) {
(gdb)
939           for (auto Ref : Summary->refs())
(gdb)
941           if (auto *FS = dyn_cast<FunctionSummary>(Summary.get()))
(gdb)
942             for (auto Call : FS->calls())
(gdb)
943               visit(Call.first, false);
(gdb) p Call.first.name().str()
$31 = "probably_inlined"
```

3. 第 876 - 927 行处理 `probably_inlined()` 函数对应的 ValueInfo，因为 `probably_inlined()` 函数对应的 ValueInfo 不是 live 的，没有添加进 Worklist 中过，所以在将其设置为 live，然后添加至 Worklist 中

```
876       auto visit = [&](ValueInfo VI, bool IsAliasee) {
(gdb) n
885         VI = updateValueInfoForIndirectCalls(Index, VI);
(gdb)
886         if (!VI)
(gdb)
889         if (llvm::any_of(VI.getSummaryList(),
(gdb)
900         if (isPrevailing(VI.getGUID()) == PrevailingType::No) {
(gdb)
923         for (auto &S : VI.getSummaryList())
(gdb)
924           S->setLive(true);
(gdb)
923         for (auto &S : VI.getSummaryList())
(gdb)
925         ++LiveSymbols;
(gdb)
926         Worklist.push_back(VI);
(gdb)
927       };
```

4. 第 929 - 945 行第二轮迭代，此时 Worklist 中还是只有一个元素，是 `probably_inlined()` 函数对应的 ValueInfo，而 `probably_inlined()` 函数没有引用其他的 symbol，所以在没有添加任何 symbol 至

Worklist 中。第 929 - 945 行第三轮迭代，Worklist 为空，到达不动点，迭代结束。

```
929       while (!Worklist.empty()) {
(gdb) n
930         auto VI = Worklist.pop_back_val();
(gdb)
931         for (auto &Summary : VI.getSummaryList()) {
(gdb)
932           if (auto *AS = dyn_cast<AliasSummary>(Summary.get())) {
(gdb)
939           for (auto Ref : Summary->refs())
(gdb)
941           if (auto *FS = dyn_cast<FunctionSummary>(Summary.get()))
(gdb)
942             for (auto Call : FS->calls())
(gdb)
931         for (auto &Summary : VI.getSummaryList()) {
(gdb)
929       while (!Worklist.empty()) {
(gdb)
946       Index.setWithGlobalValueDeadStripping();
```

5. 函数 `computeDeadSymbols()` 结束，tu1 和 tu2 中一共有 3 个 symbol，其中 `main()` 和 `probably_inlined()` 是 live 的，而 `unused()` 是 dead，所以最后链接时，会删除 `unused()` 函数。

```
946       Index.setWithGlobalValueDeadStripping();
(gdb)
948       unsigned DeadSymbols = Index.size() - LiveSymbols;
(gdb)
949       LLVM_DEBUG(dbgs() << LiveSymbols << " symbols Live, and " << DeadSymbols
(gdb)
951       NumDeadSymbols += DeadSymbols;
(gdb)
952       NumLiveSymbols += LiveSymbols;
(gdb)
853       SmallVector<ValueInfo, 128> Worklist;
(gdb)
953     }
(gdb) p DeadSymbols
$32 = 1
(gdb) p LiveSymbols
$33 = 2
```

6.1.2 References

1. https://en.wikipedia.org/wiki/Interprocedural_optimization
2. http://llvm.org/docs/LinkTimeOptimization.html
3. http://llvm.org/docs/GoldPlugin.html

- genindex
- modindex
- search

CHAPTER

SEVEN

SANITIZER

7.1 How To Write a Sanitizer

7.1.1 How To Write A Dumb Sanitizer

本文所实现的 DumbSanitizer 的完整代码见 DumbSanitizer.patch · GitHub，基于 llvm 14.0.4。

Introduction —What is a sanitizer?

Sanitizers 是由 Google 开源的动态代码分析工具，包括：

- AddressSanitizer (ASan)
- LeakSanitizer (LSan)
- ThreadSanitizer (TSan)
- UndefinedBehaviorSanitizer (UBSsan)
- MemorySanitizer (MSan)

所有的 Sanitizer 都由编译时插桩和运行时库两部分组成，Sanitizer 自 Clang 3.1 和 GCC 4.8 开始被集成在 Clang 和 GCC 中。

例如 ASan 是用于检测 Use-after-free, heap-buffer-overflow, stack-buffer-overflow 等内存错误的。对于如下代码：

```
// clang -O0 -g -fsanitize=address test.cpp && ./a.out
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

使用命令 `clang -O0 -g -fsanitize=address test.cpp` 就可以得到开启 ASan 编译后的产物，然后运行编译产物 a.out 就会得到如下类似输入，说明在运行 a.out 时发现了一个 UAF：

```
=================================================================
==6254== ERROR: AddressSanitizer: heap-use-after-free on address 0x603e0001fc64 at pc␣
↪0x417f6a bp 0x7fff626b3250 sp 0x7fff626b3248
READ of size 4 at 0x603e0001fc64 thread T0
    #0 0x417f69 in main test.cpp:5
    #1 0x7fae62b5076c (/lib/x86_64-linux-gnu/libc.so.6+0x2176c)
    #2 0x417e54 (a.out+0x417e54)
0x603e0001fc64 is located 4 bytes inside of 400-byte region [0x603e0001fc60,
↪0x603e0001fdf0)
freed by thread T0 here:
    #0 0x40d4d2 in operator delete[](void*) llvm/projects/compiler-rt/lib/asan/asan_
↪new_delete.cc:61
    #1 0x417f2e in main test.cpp:4
previously allocated by thread T0 here:
    #0 0x40d312 in operator new[](unsigned long) llvm/projects/compiler-rt/lib/asan/
↪asan_new_delete.cc:46
    #1 0x417f1e in main test.cpp:3
Shadow bytes around the buggy address:
  0x1c07c0003f30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f50: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f60: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x1c07c0003f80: fa fa fa fa fa fa fa fa fa fa fa fa[fd]fd fd fd
  0x1c07c0003f90: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c07c0003fa0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c07c0003fb0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fa fa
  0x1c07c0003fc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003fd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap righ redzone:     fb
  Freed Heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
```

(continues on next page)

(continued from previous page)

```
  ASan internal:          fe
==6254== ABORTING
```

Quick Start —Writing dumb sanitizer

接下来这一节我们就来讲解下怎么实现一个简单的 Sanitizer（本文称之为 DumbSanitizer 或 DbSan）。我们的 DumbSanitizer 实现下述功能：对于程序中的每一个变量，我们都统计该变量在程序运行中被访问了多少次，并且在程序退出时打印出访问次数最多的变量。

Compile llvm project with compiler-rt

如何编译 llvm 可以参考 Building LLVM with CMake，需要注意的是为了使用 Sanitizer 我们需要将 `compiler-rt` 加入到 LLVM_ENABLE_PROJECTS 这个 CMake varibale 里。

```
$ git clone -b llvmorg-14.0.4 https://github.com/llvm/llvm-project.git --depth 1
$ cd llvm-project
$ cmake -DCMAKE_INSTALL_PREFIX=${HOME}/llvm-bin -DCMAKE_BUILD_TYPE=Release -DLLVM_
↪ENABLE_PROJECTS="clang;compiler-rt" -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_ENABLE_
↪DUMP=ON ../llvm-project/llvm
$ make -j12
$ make install
```

Implementing the instrumentation pass

我们在本文最开始提到：所有的 Sanitizer 都由编译时插桩和运行时库两部分组成，并且几乎所有的 Sanitizer 的插桩部分都是通过 LLVM pass 的方式实现的。我们的 DumbSanitizer 也不例外。（关于 LLVM pass 的编写，见 Writing an LLVM Pass）

本节就来说明 DumbSanitizer 的插桩部分是如何实现的。

这里只对一些关键点进行说明，完整实现见 DumbSanitizer.patch · GitHub 中：

- llvm-project/llvm/include/llvm/Transforms/Instrumentation/DumbSanitizer.h
- llvm-project/llvm/lib/Transforms/Instrumentation/DumbSanitizer.cpp。

首先说一下，我们实现“对于程序中的每一个变量，统计该变量在程序运行中被访问了多少次，并且在程序退出时打印出访问次数最多的变量”该功能的思路：

- 编译时插桩：对于每一次 memory access (load, store)，我们都会在此次 acccess 之前插入一个函数调用 (__dbsan_read, __dbsan_write)，该函数调用是在运行时库中实现的。
- 运行时库：维护一个全局 map，该 map 记录了每一个 address 被访问的次数。函数 __dbsan_read, __db_write 的实现就是去更新该 map 中 key 为本次访问变量的 address 所对应的 value 的值。

即，程序使用 DumbSanitizer 编译后，每一次对变量 x 的读/写之前都会先调用 __dbsan_read/__dbsan_write，把变量 x 的地址传过去，__dbsan_read/__dbsan_write 会将 access_count_map[&x]++。在程序退出时根据 access_count_map 的内容就能给出访问次数最多的变量/地址了。

那么如何实现在每一次 memory access (load, store) 之前都插入一个函数调用 (__dbsan_read, __dbsan_write) 呢? 核心代码其实非常简单：

```
SmallVector<Instruction *, 16> LoadsAndStores;
for (auto &BB : F) {
  for (auto &Inst : BB) {
    if (isa<LoadInst>(Inst) || isa<StoreInst>(Inst))
      LoadsAndStores.push_back(&Inst);
  }
}

for (auto *Inst : LoadsAndStores) {
  IRBuilder<> IRB(Inst);
  bool IsWrite;
  Value *Addr = nullptr;
  if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
    IsWrite = false;
    Addr = LI->getPointerOperand();
  } else if (StoreInst *SI = dyn_cast<StoreInst>(I)) {
    IsWrite = true;
    Addr = SI->getPointerOperand();
  }
  if (IsWrite) {
    IRB.CreateCall(DbsanWriteFunc, IRB.CreatePointerCast(Addr, IRB.getInt8PtrTy()));
  } else {
    IRB.CreateCall(DbsanReadFunc, IRB.CreatePointerCast(Addr, IRB.getInt8PtrTy()));
  }
}
```

简单解释一下。其实就是遍历 Function F 中的所有指令，收集其中的 LoadInst 和 StoreInst。然后对于每一个保存起来的 LoadInst 或 StoreInst，通过 IRBuilder 在其之前都插入一条 CallInst，被调函数就是 __dbsan_read 或 __dbsan_write。函数 __dbsan_read 或 __dbsan_write 只有一个参数，该参数就是 LoadInst 或 StoreInst 的 PointerOperand，即读写的 address。

Implementing the runtime library

介绍完编译时插桩的关键点后，再来介绍下运行时库的核心实现。

DumbSanitizer 运行时库部分的核心实现见 DumbSanitizer.patch · GitHub 中的：

- llvm-project/compiler-rt/lib/dbsan/dbsan_interface.h
- llvm-project/compiler-rt/lib/dbsan/dbsan_interface.cpp
- llvm-project/compiler-rt/lib/dbsan/dbsan_rtl.h
- llvm-project/compiler-rt/lib/dbsan/dbsan_rtl.cpp

dbsan_interface.h 和 dbsan_interface.cpp 中是对暴露给外部的函数 __dbsan_read 和 __dbsan_write 的实现：

```
void __dbsan_read(void *addr) { MemoryAccess((uptr)addr, kAccessRead); }

void __dbsan_write(void *addr) { MemoryAccess((uptr)addr, kAccessWrite); }
```

可以看到 __dbsan_read 和 __dbsan_write 的实现就是对函数 MemoryAccess 的包装，MemoryAccess 的实现位于 dbsan_rtl.h 和 dbsan_rtl.cpp。

```
void MemoryAccess(uptr addr, AccessType typ) {
  ctx->access_count_map[addr]++;
  uptr access_count = ctx->access_count_map[addr];
  if (access_count > ctx->most_frequently_accessed_count) {
    ctx->most_frequently_accessed_count = access_count;
    ctx->most_frequently_accessed_addr = addr;
  }
}
```

MemoryAccess 的实现也很简单，就是更新 access_count_map 中 key 为 addr 的 value 值，然后更新访问次数最多的 address。

这里 ctx 是运行时库中维护的一个 Context 类型的全局变量：

```
struct Context {
  bool initialized;
  uptr most_frequently_accessed_addr;
  uptr most_frequently_accessed_count;
  __sanitizer::DenseMap<uptr, uptr> access_count_map;
};
```

- most_frequently_accessed_addr 用于记录访问次数最多的地址
- most_frequently_accessed_count 用于记录最多的访问次数是多少
- access_count_map 则是记录了每一个地址被访问了多少次

最后讲一下我们是如何做到程序退出时打印访问次数最多的变量的。其实也很简单，就是通过 atexit 来注册程序退出时执行的函数，在该函数中直接打印我们在 Context 中保存的 most_frequently_accessed_addr 和 most_frequently_accessed_count 即可。

```
static void dbsan_atexit() {
  __sanitizer::Printf(
      "#Most frequently accessed address: %p, access count: %zd\n",
      (void *)ctx->most_frequently_accessed_addr,
      ctx->most_frequently_accessed_count);
}
```

Integrating the sanitizer

实现完 DumbSanitizer 的编译时插桩和运行时库这两个核心部分，剩下的就是将我们的 DumbSanitizer 集成在 clang/llvm 的编译流程中，使得能够通过编译选项 -fsanitize=dumb 来启用 DumbSanitizer。

这部分修改的文件多且杂，没有什么需要特别说明的地方。这里只给出所需要修改的文件，详见 DumbSanitizer.patch · GitHub

- llvm-project/clang/include/clang/Basic/Sanitizers.def，添加 DumbSanitizer 的定义
- llvm-project/clang/include/clang/Driver/SanitizerArgs.h，添加是否启用的 DumbSanitizer 的判断
- 修改 llvm-project/clang/lib/CodeGen/BackendUtil.cpp，将 DumbSanitizer 的插桩 pass 添加至 pass manager
- 修改 llvm-project/clang/lib/Driver/ToolChains/CommonArgs.cpp，如果启用了 DumbSanitizer，则链接 DumbSanitizer 的运行时库
- 修改 llvm-project/clang/lib/Driver/ToolChains/Linux.cpp，定义 DumbSanitizer 所支持的架构，简单起见我们 DumbSanitizer 只支持 X86_64

Running the dumb sanitizer

本节我们用一个例子来跑下 DumbSanitizer ，看看效果。

```
// clang++ -fsanitize=dumb test.cpp -o test
// DBSAN_OPTIONS='print_frequent_access=1' ./test

#include <stdio.h>
int main(int argc, char **argv) {
  int r = 0;
  int i = 1;
  printf("address of `r` is %p\n", &r);
  printf("address of `i` is %p\n", &i);
  for (; i < 2; ++i) {
```

(continues on next page)

(continued from previous page)

```
    r++;
  }
  return i + r;
}
```

这里我们在优化等级为 O0 的情况下，开启 DumbSanitizer（注：DumbSanitizer 是在所有的优化 pass 执行后，才执行插桩 pass，即 DumbSanitizer 插桩的是已经优化后的代码，可以尝试改变优化等级查看上述例子程序的输出）。

在执行编译后的二进制时，我们设置了环境变量 DBSAN_OPTIONS，通过 DBSAN_OPTIONS 中的参数 print_frequent_access 为 1 还是 0 来控制在程序退出时是否打印访问次数最多的变量地址是什么。

上述例子的运行结果如下所示：

```
address of `r` is 0x7fff5817396c
address of `i` is 0x7fff58173968
#Most frequently accessed address: 0x7fff58173968, access count: 6
```

可以看出被访问次数最多的变量是 i，被访问了的 6 次。

感兴趣可以通过 LLVM IR 来分析这是为什么，这里就不再赘述了。

```
define dso_local noundef i32 @main(i32 noundef %0, i8** noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca i32, align 4 ; address of r
  %7 = alloca i32, align 4 ; address of i
  %8 = bitcast i32* %3 to i8*
  call void @__dbsan_write4(i8* %8)
  store i32 0, i32* %3, align 4
  %9 = bitcast i32* %4 to i8*
  call void @__dbsan_write4(i8* %9)
  store i32 %0, i32* %4, align 4
  %10 = bitcast i8*** %5 to i8*
  call void @__dbsan_write8(i8* %10)
  store i8** %1, i8*** %5, align 8
  %11 = bitcast i32* %6 to i8*
  call void @__dbsan_write4(i8* %11) ; r = 0
  store i32 0, i32* %6, align 4
  %12 = bitcast i32* %7 to i8*
  call void @__dbsan_write4(i8* %12) ; i = 1
  store i32 1, i32* %7, align 4
  %13 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([22 x i8],␣
↪[22 x i8]* @.str, i64 0, i64 0), i32* noundef %6)
```

(continues on next page)

(continued from previous page)

```
  %14 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([22 x i8],␣
↪[22 x i8]* @.str.1, i64 0, i64 0), i32* noundef %7)
  br label %15

15:                                               ; preds = %24, %2
  %16 = bitcast i32* %7 to i8*
  call void @__dbsan_read4(i8* %16); i < 2
  %17 = load i32, i32* %7, align 4
  %18 = icmp slt i32 %17, 2
  br i1 %18, label %19, label %29

19:                                               ; preds = %15
  %20 = bitcast i32* %6 to i8*
  call void @__dbsan_read4(i8* %20) ; r' = r (part1 of r++)
  %21 = load i32, i32* %6, align 4
  %22 = add nsw i32 %21, 1
  %23 = bitcast i32* %6 to i8*
  call void @__dbsan_write4(i8* %23) ; r = r' + 1 (part2 of r++)
  store i32 %22, i32* %6, align 4
  br label %24

24:                                               ; preds = %19
  %25 = bitcast i32* %7 to i8*
  call void @__dbsan_read4(i8* %25) ; i' = i (part1 of ++i)
  %26 = load i32, i32* %7, align 4
  %27 = add nsw i32 %26, 1
  %28 = bitcast i32* %7 to i8*
  call void @__dbsan_write4(i8* %28) ; i = i' + 1 (part2 of ++i)
  store i32 %27, i32* %7, align 4
  br label %15, !llvm.loop !4

29:                                               ; preds = %15
  %30 = bitcast i32* %7 to i8*
  call void @__dbsan_read4(i8* %30) ; i' = i (part1 of i + r)
  %31 = load i32, i32* %7, align 4
  %32 = bitcast i32* %6 to i8*
  call void @__dbsan_read4(i8* %32) ; r' = r (part2 of i + r)
  %33 = load i32, i32* %6, align 4
  %34 = add nsw i32 %31, %33 ; i' + r' (part 3 of i + r)
  ret i32 %34
}
```

References

1. GitHub - google/sanitizers: AddressSanitizer, ThreadSanitizer, MemorySanitizer
2. GitHub - trailofbits/llvm-sanitizer-tutorial: An LLVM sanitizer tutorial

---

- genindex
- modindex
- search

7.2 How Sanitizer Runtime Initialized

7.2.1 How Sanitizer Runtime Initialized

本文分析了 sanitizer runtime 是如何做到在程序启动之前进行初始化的。

What is a sanitizer?

Sanitizers 是由 Google 开源的动态代码分析工具，包括：

- AddressSanitizer (ASan)
- LeakSanitizer (LSan)
- ThreadSanitizer (TSan)
- UndefinedBehaviorSanitizer (UBSsan)
- MemorySanitizer (MSan)

所有的 Sanitizer 都由编译时插桩和运行时库两部分组成，Sanitizer 自 Clang 3.1 和 GCC 4.8 开始被集成在 Clang 和 GCC 中。

Sanitizer runtime

我们先以 ASan 为例，简单看下 ASan runtime library 做了哪些事情。

ASan runtime 做的最主要的事情就是替换了 malloc/free, new/delete 的实现。这样应用程序的内存分配都由 ASan 实现的 allocator 来做，就能检测像 heap-use-after-free, double-free 这样的堆错误了。

程序在启用 ASan 后，virtual address space 会被分成 main application memory 和 shadow memory 两部分：

```
// Typical shadow mapping on Linux/x86_64 with SHADOW_OFFSET == 0x00007fff8000:
// || `[0x10007fff8000, 0x7fffffffffff]` || HighMem    ||
// || `[0x02008fff7000, 0x10007fff7fff]` || HighShadow ||
// || `[0x00008fff7000, 0x02008fff6fff]` || ShadowGap  ||
// || `[0x00007fff8000, 0x00008fff6fff]` || LowShadow  ||
// || `[0x000000000000, 0x00007fff7fff]` || LowMem     ||
```

那么 ASan shadow memory 是什么时候初始化的？答：在程序启动之前 sanitizer runtime 初始化时做的。

另外，在使用 ASan 时，我们可以通过环境变量 ASAN_OPTIONS 来设置一些运行时参数，如：

- log_path，指定 sanitizer 的报告输出的位置
- detect_stack_use_after_return，是否检测 stack-use-after-return 类型的 bug
- alloc_dealloc_mismatch，是否检测 alloc-dealloc-mismatch 类型的 bug
- ...

那么这些通过 ASAN_OPTIONS 设置的运行时参数又是在什么时候被解析生效的呢？答：也是在程序启动之前在 sanitizer runtime 初始化时做的。

ASan runtime 初始化的入口函数是 `__asan_init`（感兴趣的话，可以仔细阅读下代码，本文不做此做详细的分析了。我这里截取了与 runtime flags 和 shadow memory 初始化相关的函数）。

```
// compiler-rt/lib/asan/asan_rtl.cpp

void __asan_init() {
  AsanActivate();
  AsanInitInternal();
}

static void AsanInitInternal() {
  ...
  // Initialize flags. This must be done early, because most of the
  // initialization steps look at flags().
  InitializeFlags();
  ...
  // Set up the shadow memory.
  InitializeShadowMemory();
  ...
}
```

那 sanitizer runtime 是怎么做到在程序启动之前执行初始化的相关代码的呢？一言以蔽之：`.init_array`。下面我们详细看下是怎么实现的。

Instrumentation

首先看下 ASan 插装代码中与 sanitizer runtime 初始化相关的实现。

ASan 插装部分的代码实现位于 llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp。有如下代码片段（简化并省略部分代码）：

```
// llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp

const char kAsanInitName[] = "__asan_init";
const char kAsanModuleCtorName[] = "asan.module_ctor";

bool ModuleAddressSanitizer::instrumentModule(Module &M) {
  // ...
  // Create a module constructor.
  std::string AsanVersion = std::to_string(GetAsanVersion(M));
  std::string VersionCheckName =
      ClInsertVersionCheck ? (kAsanVersionCheckNamePrefix + AsanVersion) : "";
  std::tie(AsanCtorFunction, std::ignore) =
      createSanitizerCtorAndInitFunctions(M, kAsanModuleCtorName,
                                          kAsanInitName, /*InitArgTypes=*/{},
                                          /*InitArgs=*/{}, VersionCheckName);
  // ...
  const uint64_t Priority = GetCtorAndDtorPriority(TargetTriple);
  appendToGlobalCtors(M, AsanCtorFunction, Priority);

  return true;
}
```

`ModuleAddressSanitizer::instrumentModule(Module &M)` 的实现很简单：

1. 先调用 `createSanitizerCtorAndInitFunctions` 创建了一个名为 `kAsanModuleCtorName` 的函数，该函数的函数体很简单，就是对 `kAsanInitName` 函数和 `VersionCheckName` 函数的调用，其中因为 `kAsanInitName` 函数没有任何的参数，所以 `InitArgTypes` 和 `InitArgs` 都是空。
2. 然后通过调用 `appendToGlobalCtors` 将通过 `createSanitizerCtorAndInitFunctions` 创建的函数，添加至 GlobalCtors 中。

可能上述描述的还是不够清晰，下面结合一个例子来进一步说明。

考虑如下代码：

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

通过 Compiler Explorer 能很方便看到开启 ASan 后（-fsanitize=address）生成的 LLVM IR 是什么样（这里只截取了部分我们关注的 LLVM IR）：

```
@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32, void ()*,␣
↪i8* } { i32 1, void ()* @asan.module_ctor, i8* null }]

declare void @__asan_init()

declare void @__asan_version_mismatch_check_v8()

define internal void @asan.module_ctor() {
  call void @__asan_init()
  call void @__asan_version_mismatch_check_v8()
  ret void
}
```

开启 ASan 后，能明显的看到多了一个函数 `asan.module_ctor`，多了一个全局变量 `@llvm.global_ctors`。它们分别由 ASan 插装函数 `ModuleAddressSanitizer::instrumentModule(Module &M)` 调用 `createSanitizerCtorAndInitFunctions` 和 `appendToGlobalCtors` 创建的。

`asan.module_ctor` 函数体由两个函数调用组成：

- `call void @__asan_init()`
- `call void @__asan_version_mismatch_check_v8()`

函数 `__asan_init` 是在 runtime libraray 中实现的，其代码实现我们前面已经给出了；函数 `__asan_version_mismatch_check_v8` 也是在 runtime libaray 中实现的，顾名思义就是用于检测 asan instrumentation 和 runtime library 的版本是否匹配。

然后 `@llvm.global_ctors` 中包含了函数 `asan.module_ctor` 的指针。

那么 `@llvm.global_ctors` 是什么、是怎么实现的?

根据 https://llvm.org/docs/LangRef.html#the-llvm-global-ctors-global-variable：

> The `@llvm.global_ctors` array contains a list of constructor functions, priorities, and an associated global or function. The functions referenced by this array will be called in ascending order of priority (i.e. lowest first) when the module is loaded. The order of functions with the same priority is not defined.
>
> If the third field is non-null, and points to a global variable or function, the initializer function will only run if the associated data from the current module is not discarded. On ELF the referenced global variable or function must be in a comdat.

即 `@llvm.global_ctors` 是一个数组，包含了一些 constructor functions。这些 constructor functions 会按照 priority 升序在 module 被加载时被调用。

但是 llvm 文档中并没有说 `@llvm.global_ctors` 是如何做到“constructor functions 在 module 被加载时被调用“的。

.init_array

实际上 LLVM IR 中的 `@llvm.global_ctors` 在生成汇编代码时，对应的是 `.init_array`。

我们还是通过前面用到的示例代码来说明：

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

通过 `clang++ -fsanitize=address test.cpp -S` 可以得到开启 ASan 后生成的汇编代码（我们这里省略了 main 函数的汇编代码）：

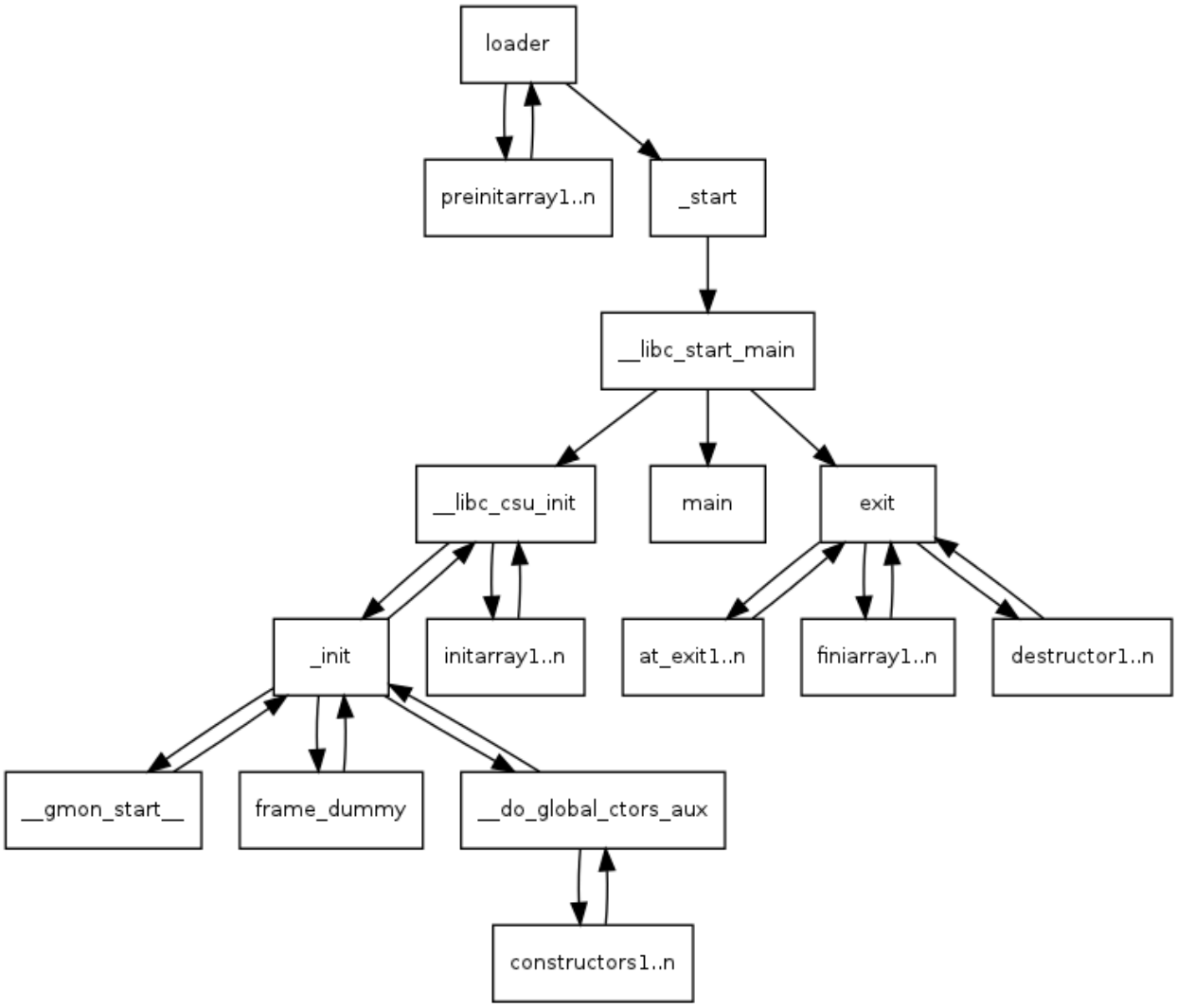
```
    .section    .text.asan.module_ctor,"axR",@progbits
    .p2align    4, 0x90     # -- Begin function asan.module_ctor
    .type    asan.module_ctor,@function
asan.module_ctor:     # @asan.module_ctor
    pushq    %rbp
    movq    %rsp, %rbp
    callq    __asan_init@PLT
    callq    __asan_version_mismatch_check_v8@PLT
    popq    %rbp
    retq

    .section    .init_array.1,"aw",@init_array
    .p2align    3
    .quad    asan.module_ctor
```

首先是 `asan.module_ctor` 的汇编实现，没什么特别：就是两条 call 指令，分别对函数 `__asan_init` 和 `__asan_version_mismatch_check_v8` 的调用。

需要注意的是 `.init_array.1 section`，存储了函数 `asan.module_ctor` 的指针。`.init_array.1` 中的 **".1"** 和 `asan.module_ctor` 的 priority 有关，因为 `{ i32 **1**, void ()* @asan.module_ctor, i8* null }` 即 `asan.module_ctor` 的 priority 是 1，所以这里就是 `.init_array.1`

关于 `.init_array`，maskray 的这篇文章写的很详细 https://maskray.me/blog/2021-11-07-init-ctors-init-array 。

在 ELF 文件被 loader 加载后，会先执行.init_array section 中的函数，再执行 main 函数。这样就达到了在程序启动之前执行 sanitizer runtime 初始化函数的效果。



Summary

总结一下，sanitizer runtime 是如何做到在程序启动之前进行初始化的，以 ASan 为例：

- 首先 sanitizer runtime library 中存在一个初始化函数 `__asan_init`，来做 ASan runtime 的初始化工作，如：初始化 shadow memory、初始化一些运行时参数。
- 然后在开启 ASan，编译时插装这个阶段，会创建一个名为 `asan.module_ctor` 的函数，该函数会调用 `__asan_init`，然后将 `asan.module_ctor` 的函数指针加入到 `@llvm.global_ctors` 中。
- 在生成汇编代码时，会将 `@llvm.global_ctors` 中的函数指针放在 `.init_array` section 中。
- 最后在 loader 加载 ELF 文件时，会先执行 `.init_array` 中函数指针指向的函数，然后再执行 `main()` 函数，这样就做到在程序启动之前初始化 ASan runtime 了。

P.S.

C/C++ 面试有一个常见问题就是问如何实现在 `main()` 函数执行之前，执行一条语句：https://www.zhihu.com/question/26031933

- 一种解决方案是通过 `__attribute((constructor))` 来修饰相关函数，实现该函数在 `main()` 函数执行之前被执行。
- 还有一种方案是利用全局变量的构造函数在 `main()` 函数执行之前执行实现该效果。

实际上述两种方案在汇编的角度来看是一样的，都是通过 `.init_array` section 来实现的。

Reference

1. http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html
2. https://maskray.me/blog/2021-11-07-init-ctors-init-array

- genindex
- modindex
- search

7.3 How Sanitizer Interceptor Works

7.3.1 How Sanitizer Interceptor Works

我们在前面的文章中提到，所有的 Sanitizer 都由编译时插桩 (compile-time instrumentation) 和运行时库 (run-time library) 两部分组成。

那么 sanitizer 的运行时库中做了哪些事情呢?

以 ASan 为例：

- ASan 编译时会在每一处内存读写语句之前插入代码，根据每一次访问的内存所对应的影子内存 (shadow memory，就是使用额外的内存记录常规内存的状态）的状态来检测本次内存访问是否合法。还会在栈变量和全局变量附近申请额外内存作为危险区用于检测内存溢出。
- ASan 运行时库中最主要的就是替换了 malloc/free, new/delete 的实现，这样应用程序的内存分配都由 ASan 实现的内存分配器负责。ASan 内存分配器会在它分配的堆内存附近申请额外内存用于检测堆内存溢出，还会将被释放的内存优先放在隔离区 (quarantine) 用于检测像 heap-use-after-free, double-free 这样的堆内存错误。

ASan 运行时库中实际上不止替换了 malloc/free, new/delete 的函数实现，还替换了非常多的库函数的实现，如：memcpy, memmove, strcpy, strcat, pthread_create 等。

那么 sanitizer 是如何做到替换 malloc, free, memcpy 这些函数实现的呢？答案就是 sanitizer 中的 interceptor 机制。

本文以 ASan 为例，分析在 Linux x86_64 环境下 sanitizer interceptor 的实现原理。

Symbol interposition

在讲解 sanitizer interceptor 的实现原理之前，我们先来了解一下前置知识：symbol interposition。

首先我们考虑这样一个问题：如何在我们的应用程序中替换 libc 的 malloc 实现为我们自己实现的版本？

1. 一个最简单的方式就是在我们的应用程序中定义一个同名的 malloc 函数
2. 还有一种方式就是将我们的 malloc 函数实现在 libmymalloc.so 中，然后在运行我们的应用程序之前设置环境变量 `LD_PRELOAD=/path/to/libmymalloc.so`

那么为什么上述两种方式能生效呢？答案是 symbol interposition。

ELF specfication 在第五章 Program Loading and Dynamic Linking 中提到：

> When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the `DT_NEEDED` entries (in order), and then at the second level `DT_NEEDED` entries, and so on.

动态链接器 (dynamic linker/loader) 在符号引用绑定 (binding symbol references) 时，以一种广度优先搜索的顺序来查找符号：executable, needed0.so, needed1.so, needed2.so, needed0_of_needed0.so, needed1_of_needed0.so, ⋯

如果设置了 LD_PRELOAD，那么查找符号的顺序会变为：executable，preload0.so，preload1.so needed0.so, needed1.so, needed2.so, needed0_of_needed0.so, needed1_of_needed0.so, ⋯

如果一个符号在多个组件（executable 或 shared object）中都存在定义，那么动态链接器会选择它所看到的第一个定义。

我们通过一个例子来理解该过程：

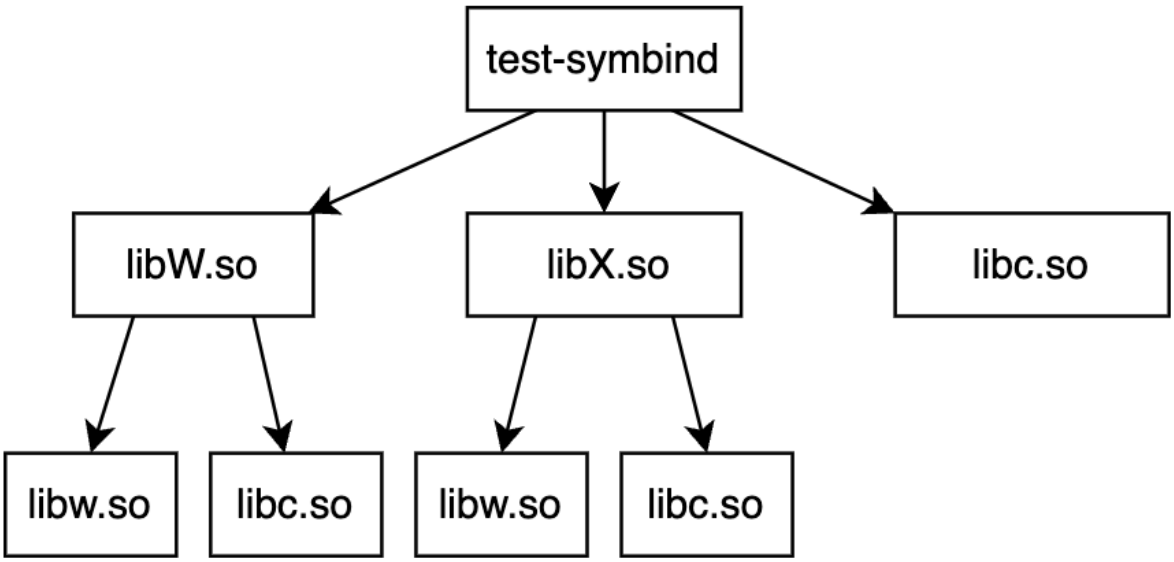
```
$ cat main.c
extern int W(), X();

int main() { return (W() + X()); }

$ cat W.c
extern int b();

int a() { return (1); }
int W() { return (a() - b()); }

$ cat w.c
int b() { return (2); }
```

(continues on next page)

(continued from previous page)

```
$ cat X.c
extern int b();

int a() { return (3); }
int X() { return (a() - b()); }

$ cat x.c
int b() { return (4); }

$ gcc -o libw.so -shared w.c
$ gcc -o libW.so -shared W.c -L. -lw -Wl,-rpath=.
$ gcc -o libx.so -shared x.c
$ gcc -o libX.so -shared X.c -L. -lx -Wl,-rpath=.
$ gcc -o test-symbind main.c -L. -lW -lX -Wl,-rpath=.
```

该例子中可执行文件与动态库之间的依赖关系如下图所示：



按照我们前面所说，本例中动态链接器在进行符号引用绑定时，是按照广度优先搜索的顺序，即：test-symbind, libW.so, libX.so, libc.so, libw.so, libx.so 的顺序查找符号定义的。

动态链接器提供了环境变量 LD_DEBUG 来输出一些调试信息，我们可以通过设置环境变量 LD_DEBUG= “symbols:bindings” 看下 test-symbind 的 symbol binding 的过程：

```
$ LD_DEBUG="symbols:bindings" ./test-symbind
   1884890:        symbol=a;  lookup in file=./test-symbind [0]
   1884890:        symbol=a;  lookup in file=./libW.so [0]
   1884890:        binding file ./libW.so [0] to ./libW.so [0]: normal symbol `a'
```

(continues on next page)

(continued from previous page)

```
   1884890:        symbol=b;  lookup in file=./test-symbind [0]
   1884890:        symbol=b;  lookup in file=./libW.so [0]
   1884890:        symbol=b;  lookup in file=./libX.so [0]
   1884890:        symbol=b;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
   1884890:        symbol=b;  lookup in file=./libw.so [0]
   1884890:        binding file ./libW.so [0] to ./libw.so [0]: normal symbol `b'
   1884890:        symbol=a;  lookup in file=./test-symbind [0]
   1884890:        symbol=a;  lookup in file=./libW.so [0]
   1884890:        binding file ./libX.so [0] to ./libW.so [0]: normal symbol `a'
   1884890:        symbol=b;  lookup in file=./test-symbind [0]
   1884890:        symbol=b;  lookup in file=./libW.so [0]
   1884890:        symbol=b;  lookup in file=./libX.so [0]
   1884890:        symbol=b;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
   1884890:        symbol=b;  lookup in file=./libw.so [0]
   1884890:        binding file ./libX.so [0] to ./libw.so [0]: normal symbol `b'
```

- 函数 a 在 libW.so 和 libX.so 中都有一份定义，但因为是按照 test-symbind, libW.so, libX.so, libc.so, libw.so, libx.so 的顺序查找符号定义的，所以最终所有对函数 a 的引用都绑定到 libW.so 中函数 a 的实现
- 函数 b 在 libw.so 和 libx.so 中都有一份定义，但因为是按照 test-symbind, libW.so, libX.so, libc.so, libw.so, libx.so 的顺序查找符号定义的，所以最终所有对函数 b 的引用都绑定到 libw.so 中函数 b 的实现

这样我们就理解为什么上述两种替换 malloc 的方式能生效了：

- 方式一：在我们的应用程序中定义一个同名的 malloc 函数。动态链接器在查找符号时 executable 的顺序在 libc.so 之前，因此所有对 malloc 的引用都会绑定到 executable 中 malloc 的实现。
- 方式二：将我们的 malloc 函数实现在 libmymalloc.so 中，然后在运行我们的应用程序之前设置环境变量 LD_PRELOAD=/path/to/libmymalloc.so。动态链接器在查找符号时 libmymalloc.so 的顺序在 libc.so 之前，因此所有对 malloc 的引用都会绑定到 libmymalloc.so 中 malloc 的实现。

实际上 sanitizer 对于 malloc/free 等库函数的替换正是利用了 symbol interposition 这一特性。下面我们以 ASan 为例来验证一下。

考虑如下代码：

```
// test.cpp
#include <iostream>
int main() {
    std::cout << "Hello AddressSanitizer!\n";
}
```

我们首先看下 GCC 的行为。

使用 GCC 开启 ASan 编译 test.cpp，`g++ -fsanitize=address test.cpp -o test-gcc-asan` 得到编译产物 test-gcc-asan。因为 GCC 默认会**动态链接** ASan 运行时库，所以我们可以使用 `objdump -p test-gcc-asan | grep NEEDED` 查看 test-gcc-asan 依赖的动态库 (shared objects)：

```
$ objdump -p test-gcc-asan | grep NEEDED
  NEEDED               libasan.so.5
  NEEDED               libstdc++.so.6
  NEEDED               libm.so.6
  NEEDED               libgcc_s.so.1
  NEEDED               libc.so.6
```

可以清楚的看到在 test-gcc-asan 依赖的动态库中 libasan.so 的顺序是在 libc.so 之前的。实际上链接时参数 `-fsanitize=address` 会使得 libasan.so 成为程序的第一个依赖库。

然后我们再通过环境变量 LD_DEBUG 看下 test-gcc-asan 的 symbol bindding 的过程：

```
$ LD_DEBUG="bindings" ./test-gcc-asan
   3309213:        binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /usr/lib/x86_
↪64-linux-gnu/libasan.so.5 [0]: normal symbol `malloc' [GLIBC_2.2.5]
   3309213:        binding file /lib64/ld-linux-x86-64.so.2 [0] to /usr/lib/x86_64-
↪linux-gnu/libasan.so.5 [0]: normal symbol `malloc' [GLIBC_2.2.5]
   3309213:        binding file /usr/lib/x86_64-linux-gnu/libstdc++.so.6 [0] to /usr/
↪lib/x86_64-linux-gnu/libasan.so.5 [0]: normal symbol `malloc' [GLIBC_2.2.5]
```

可以看到动态链接器将 libc.so, ld-linux-x86-64.so 和 libstdc++.so 中对 malloc 的引用都绑定到了 libasan.so 中的 malloc 实现。

下面我们看下 Clang，因为 Clang 默认是**静态链接** ASan 运行时库，所以我们就不看 test-clang-asan 所依赖的动态库了，直接看 symbol binding 的过程：

```
$ clang++ -fsanitize=address test.cpp -o test-clang-asan
$ LD_DEBUG="bindings" ./test-clang-asan
   3313022:        binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to ./test-clang-
↪asan [0]: normal symbol `malloc' [GLIBC_2.2.5]
   3313022:        binding file /lib64/ld-linux-x86-64.so.2 [0] to ./test-clang-asan␣
↪[0]: normal symbol `malloc' [GLIBC_2.2.5]
   3313022:        binding file /usr/lib/x86_64-linux-gnu/libstdc++.so.6 [0] to ./
↪test-clang-asan [0]: normal symbol `malloc' [GLIBC_2.2.5]
```

同样可以看到动态链接器将 libc.so, ld-linux-x86-64.so.2 和 libstdc++.so 中对 malloc 的引用都绑定到了 test-clang-asan 中的 malloc 实现（因为 ASan 运行时库中实现了 malloc，并且 clang 将 ASan 运行时库静态链接到 test-clang-asan 中）。

Sanitizer interceptor

下面我们来在源码的角度，学习下 sanitizer interceptor 的实现。

阅读学习 LLVM 代码的一个非常有效的方式就是结合对应的测试代码来学习。

Sanitizer interceptor 存在一个测试文件 interception_linux_test.cpp，llvm-project/interception_linux_test.cpp at main · llvm/llvm-project · GitHub

```
#include "interception/interception.h"
#include "gtest/gtest.h"

static int InterceptorFunctionCalled;

DECLARE_REAL(int, isdigit, int);

INTERCEPTOR(int, isdigit, int d) {
  ++InterceptorFunctionCalled;
  return d >= '0' && d <= '9';
}

namespace __interception {

TEST(Interception, Basic) {
  EXPECT_TRUE(INTERCEPT_FUNCTION(isdigit));

  // After interception, the counter should be incremented.
  InterceptorFunctionCalled = 0;
  EXPECT_NE(0, isdigit('1'));
  EXPECT_EQ(1, InterceptorFunctionCalled);
  EXPECT_EQ(0, isdigit('a'));
  EXPECT_EQ(2, InterceptorFunctionCalled);

  // Calling the REAL function should not affect the counter.
  InterceptorFunctionCalled = 0;
  EXPECT_NE(0, REAL(isdigit)('1'));
  EXPECT_EQ(0, REAL(isdigit)('a'));
  EXPECT_EQ(0, InterceptorFunctionCalled);
}

}  // namespace __interception
```

这段测试代码基于 sanitizer 的 interceptor 机制替换了 `isdigit` 函数的实现，在我们实现的 `isdigit` 函数中，每次 `isdigit` 函数被调用时都将变量 `InterceptorFunctionCalled` 自增 1。然后通过检验变量 `InterceptorFunctionCalled` 的值来测试 interceptor 机制的实现是否正确，通过 `REAL(isdigit)` 来调用真正的 `isdigit` 函数实现。

上述测试文件 interception_linux_test.cpp 中实现替换 `isdigit` 函数的核心部分是如下代码片段：

```
INTERCEPTOR(int, isdigit, int d) {
  ++InterceptorFunctionCalled;
  return d >= '0' && d <= '9';
}

INTERCEPT_FUNCTION(isdigit);

DECLARE_REAL(int, isdigit, int);
REAL(isdigit)('1');
```

- `INTERCEPTOR(int, isdigit, int d) { ... }` 用于将函数 `isdigit` 的实现替换为 { ⋯ } 的实现
- 在代码中调用 `isdigit` 之前，需要先调用 `INTERCEPT_FUNCTION(isdigit)`。如果 `INTERCEPT_FUNCTION(isdigit)` 返回为 true，则说明成功替换了将 libc 中 `isdigit` 函数的实现。
- `REAL(isdigit)('1')` 用于调用真正的 `isdigit` 实现，不过在调用 `REAL(isdigit)('1')` 之前需要先 `DECLARE_REAL(int, isdigit, int)`。

这部分代码在宏展开后的内容如下：

```
// INTERCEPTOR(int, isdigit, int d) 宏展开
typedef int (*isdigit_type)(int d);
namespace __interception { isdigit_type real_isdigit; }
extern "C" int isdigit(int d) __attribute__((weak, alias("__interceptor_isdigit"),␣
↪visibility("default")));
extern "C" __attribute__((visibility("default"))) int __interceptor_isdigit(int d) {
  ++InterceptorFunctionCalled;
  return d >= '0' && d <= '9';
}

// INTERCEPT_FUNCTION(isdigit) 宏展开
::__interception::InterceptFunction(
    "isdigit",
    (::__interception::uptr *) & __interception::real_isdigit,
    (::__interception::uptr) & (isdigit),
    (::__interception::uptr) & __interceptor_isdigit);

// DECLARE_REAL(int, isdigit, int) 宏展开
typedef int (*isdigit_type)(int);
namespace __interception { extern isdigit_type real_isdigit; };

// REAL(isdigit)('1') 宏展开
__interception::real_isdigit('1');
```

- 我们首先看下 INTERCEPTOR 宏做了哪些事情
  - 首先在 __interception namespace 中定义了一个函数指针 real_isdigit，该函数指针实际上后续会被设置为指向真正的 `isdigit` 函数地址。
  - 然后将 `isdigit` 函数设置为 weak，并且将 `isdigit` 设置成 `__interceptor_isdigit` 的 alias 别名
  - 最后将我们自己版本的 `isdigit` 函数逻辑实现在 `__interceptor_isdigit` 函数中

  根据 symbol interposition 这一节的内容，我们知道：要想替换 libc.so 中某个函数的实现（不妨把该函数称作 `foo`），只需要在 sanitizer runtime library 中定义同名 `foo` 函数，然后让 dynamic loader 在查找符号时 sanitizer runtime library 的顺序先于 libc.so 即可。

  那为什么这里要将我们的 `isdigit` 函数逻辑实现在函数 `__interceptor_isdigit` 中，并且将 `isdigit` 设置成 `__interceptor_isdigit` 的 alias 别名呢?

  考虑如下场景：假设用户代码中也替换了 `isdigit` 函数的实现，添加了自己的逻辑，那么最终 dynamic loader 选择的是用户代码中的 `isdigit` 的实现，而不是 sanitizer runtime library 中的 `isdigit` 的实现，这样的话 sanitizer 的功能就不能正常工作了。（实际上 sanitizer runtime library 中并没有替换 `isdigit` 的实现，这里只是用 `isdigit` 举例子便于说明）。

  但是如果我们在 sanitizer runtime library 中将 `isdigit` 设置成 `__interceptor_isdigit` 的 alias 别名，那么在用户代码中自己替换 `isdigit` 实现时就可以显示调用 `__interceptor_isdigit` 了。这样既不影响用户自行替换库函数，也不影响 sanitizer 功能的正确运行：

```
extern "C" int __interceptor_isdigit(int d);
extern "C" int isdigit(int d) {
  fprintf(stderr, "my_isdigit_interceptor\n");
  return __interceptor_isdigit(d);
}
```

  那在 sanitizer runtime library 中为什么将被替换的函数设置为 weak 呢?

  这是因为如果不设置为 weak ，在静态链接 sanitizer runtime library 时就会因为 multiple definition 而链接失败。

- 接着我们看下 INTERCEPT_FUNCTION 宏做了哪些事情

  INTERCEPT_FUNCTION 宏展开后就是对 __interception::InterceptFunction 函数的调用。`InterceptFunction` 函数的定义在 https://github.com/llvm/llvm-project/blob/main/compiler-rt/lib/interception/interception_linux.cpp：

```
namespace __interception {
static void *GetFuncAddr(const char *name, uptr wrapper_addr) {
  void *addr = dlsym(RTLD_NEXT, name);
  if (!addr) {
    // If the lookup using RTLD_NEXT failed, the sanitizer runtime library is
    // later in the library search order than the DSO that we are trying to
```

(continues on next page)

(continued from previous page)

```
    // intercept, which means that we cannot intercept this function. We still
    // want the address of the real definition, though, so look it up using
    // RTLD_DEFAULT.
    addr = dlsym(RTLD_DEFAULT, name);

    // In case `name' is not loaded, dlsym ends up finding the actual wrapper.
    // We don't want to intercept the wrapper and have it point to itself.
    if ((uptr)addr == wrapper_addr)
      addr = nullptr;
  }
  return addr;
}

bool InterceptFunction(const char *name, uptr *ptr_to_real, uptr func,
                       uptr wrapper) {
  void *addr = GetFuncAddr(name, wrapper);
  *ptr_to_real = (uptr)addr;
  return addr && (func == wrapper);
}
}  // namespace __interception
```

其实 `InterceptFunction` 函数的实现很简单：首先通过函数 `GetFuncAddr` 获得原本的名为 name 的函数地址，然后将该地址保存至指针 `ptr_to_real` 指向的内存。

函数 `GetFuncAddr` 的代码实现也很简单，核心就是 dlsym：

```
RTLD_DEFAULT
    Find the first occurrence of the desired symbol using the
    default shared object search order.  The search will
    include global symbols in the executable and its
    dependencies, as well as symbols in shared objects that
    were dynamically loaded with the RTLD_GLOBAL flag.

RTLD_NEXT
    Find the next occurrence of the desired symbol in the
    search order after the current object.  This allows one to
    provide a wrapper around a function in another shared
    object, so that, for example, the definition of a function
    in a preloaded shared object (see LD_PRELOAD in ld.so(8))
    can find and invoke the "real" function provided in
    another shared object (or for that matter, the "next"
    definition of the function in cases where there are
    multiple layers of preloading).
```

这也是为什么在函数 `GetFuncAddr` 中，先用 `dlsym(RTLD_NEXT, name)` 去寻找被 intercepted 函数

的真实地址，因为 sanitizer runtime library 是先于 name 函数真正所在的 shared object。

- 最后我们看下 DECLARE_REAL 宏和 REAL 宏做了哪些事情

  DECLARE_REAL 展开后就是声明了在 __interception namespace 中存在一个指向被替换函数真正实现的函数指针，REAL 宏就是通过这个函数指针来调用被替换函数的真正实现。

  例如，在测试用例中，`DECLARE_REAL(int, isdigit, int);` 就是在声明 __interception namespace 中存在一个函数指针 `real_isdigit`，该函数指针指向真正的 `isdigit` 函数地址，通过 `REAL(isdigit)` 来调用真正的 `isdigit` 函数。

P.S.

`__attribute__((alias))` 很有意思：

> Where a function is defined in the current translation unit, the alias call is replaced by a call to the function, and the alias is emitted alongside the original name. Where a function is not defined in the current translation unit, the alias call is replaced by a call to the real function. Where a function is defined as static, the function name is replaced by the alias name and the function is declared external if the alias name is declared external.

在 ASan runtime library 中 malloc 是 weak 符号，并且 malloc 和 __interceptor_malloc 实际指向同一个地址。

也就是说 `extern "C" void *malloc(size_t size) __attribute__((weak, alias("__interceptor_malloc"), visibility("default")));` 使得在 ASan runtime library 中造了一个弱符号 malloc，然后指向的和 __interceptor_malloc 是同一个地址。

```
$ readelf -sW --dyn-syms $(clang -print-file-name=libclang_rt.asan-x86_64.a) | grep␣
↪malloc
  ...
  99: 0000000000001150   606 FUNC    GLOBAL DEFAULT    3 __interceptor_malloc
  102: 0000000000001150   606 FUNC    WEAK   DEFAULT    3 malloc

$ readelf -sW --dyn-syms $(clang -print-file-name=libclang_rt.asan-x86_64.so) | grep␣
↪malloc
  ...
  3008: 00000000000fd600   606 FUNC    WEAK   DEFAULT   12 malloc
  4519: 00000000000fd600   606 FUNC    GLOBAL DEFAULT   12 __interceptor_malloc
```

P.S.2

熟悉在 Linux 下 sanitizer interceptor 机制的底层原理后，就很容易明白使用 sanitizer 时遇到的一些问题或坑为什么会是这样的。例如：

- Address Sanitizer fails to intercept function in shared library opened with RTLD_DEEPBIND · Issue #611 · google/sanitizers · GitHub

- ASan runtime does not come first in initial library list; you should either link runtime to your application or manually preload it with LD_PRELOAD. · Issue #796 · google/sanitizers · GitHub
- address sanitizer - Is it okay if ASAN runtime loaded as second library? - Stack Overflow

References

1. ELF interposition and -Bsymbolic | MaskRay
2. dlsym(3) - Linux manual pagedlsym(3) - Linux manual page
3. asan/tsan: weak interceptors · llvm/llvm-project@7fb7330 · GitHub

---

- genindex
- modindex
- search

7.4 How Sanitizer Get Stack Trace

7.4.1 How Sanitizer Get Stack Trace

Sanitizer 非常好用的一个原因就是报告的内容非常详细。例如 ASan 检测到一个 heap-use-after-free 类型的 bug，在报告中不仅会给出执行哪行代码时触发了 heap-use-after-free，还会给出这块堆内存是在哪里被申请的，又是在哪里被释放的。

例如下面这个 heap-use-after-free 的例子：

- “READ of size 4 at 0x603e0001fc64 thread T0” 给出的是触发 heap-use-after-free 的 stack trace
- “freed by thread T0 here:” 给出的是堆内存被释放时的 stack trace
- “previously allocated by thread T0 here:” 给出的是堆内存被申请时的 stack trace

```
// clang -O0 -g -fsanitize=address test.cpp && ./a.out
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

```
=================================================================
==6254== ERROR: AddressSanitizer: heap-use-after-free on address 0x603e0001fc64 at pc␣
↪0x417f6a bp 0x7fff626b3250 sp 0x7fff626b3248
READ of size 4 at 0x603e0001fc64 thread T0
```

(continues on next page)

(continued from previous page)

```
    #0 0x417f69 in main test.cpp:5
    #1 0x7fae62b5076c (/lib/x86_64-linux-gnu/libc.so.6+0x2176c)
    #2 0x417e54 (a.out+0x417e54)
0x603e0001fc64 is located 4 bytes inside of 400-byte region [0x603e0001fc60,
↪0x603e0001fdf0)
freed by thread T0 here:
    #0 0x40d4d2 in operator delete[](void*) llvm/projects/compiler-rt/lib/asan/asan_
↪new_delete.cc:61
    #1 0x417f2e in main test.cpp:4
previously allocated by thread T0 here:
    #0 0x40d312 in operator new[](unsigned long) llvm/projects/compiler-rt/lib/asan/
↪asan_new_delete.cc:46
    #1 0x417f1e in main test.cpp:3
Shadow bytes around the buggy address:
  0x1c07c0003f30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f50: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f60: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003f70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x1c07c0003f80: fa fa fa fa fa fa fa fa fa fa fa fa[fd]fd fd fd
  0x1c07c0003f90: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c07c0003fa0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c07c0003fb0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fa fa
  0x1c07c0003fc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c07c0003fd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap righ redzone:     fb
  Freed Heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==6254== ABORTING
```

本文以 ASan 为例分析 sanitizer runtime 是如何获取 stack trace 的。

Stack unwinding

在分析 sanitizer runtime 关于 stack trace 的具体实现之前，我们先来学习下 stack unwinding。

关于 stack unwinding，maskray 这篇文章 Stack unwinding | MaskRay 写的非常好。本节的内容都是从 maskray 这篇文章习来的。

Stack unwinding 主要用于获取 stack trace 或实现 C++ exception。

Stack unwinding 可以分为两类：

- synchronous: 由程序自身触发的，只发生在函数调用处（在 function body 内，不会出现在 prologue/epilogue 处）。
- asynchronous: 由 garbage collector, signal 或外部程序触发，这类 stack unwinding 可以发生在函数 prologue/epilogue 处。

Sanitizer 的 stack unwinding 就是 synchronous stack unwinding，由 sanitizer runtime 自身触发。例如 sanitizer runtime 在 malloc/free 时会通过 stack unwinding 获取 stack trace。

因此本文中我们只讨论 synchronous stack unwinding（而且我也不了解 asynchronous stack unwinding，以后有时间可以学习下）。

Frame pointer

最朴素的 stack unwinding 就是基于 frame (base) pointer [%rbp] 来实现的。如果编译时添加了选项 `-fno-omit-frame-pointer`，那么在函数 prologue/epilogue 处会有如下指令：

```
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret
```

函数 prologue 处 `pushq %rbp` 将 caller 的 frame pointer 值压栈，`movq %rsp, %rbp` 将寄存器%rbp 的值更新为保存 caller frame pointer 的栈地址。这样，一旦我们获取了当前函数的 frame pointer 的值，将其解引用后就可以得到 caller 的 frame pointer 的值，不停地解引用就能获取到所有栈帧的 frame pointer。

我们可以将 stack frame 抽象为如下结构体：

```
struct stack_frame {
  stack_frame* nextFrame;
  void* returnAddress;
};
```

以如下汇编代码为例进行说明：在 x86_64 下，caller 会在执行 call 指令时会将当前函数的下一条指令地址压栈，然后跳转到 callee 的入口处继续执行，接着在 callee 的第一条指令就是 `pushq %rbp` 将寄存器%rbp 的值压栈。这两条指令合作将 struct stack_frame 的内容填充好。

```
main:                                   # @main
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movl    $2, %edi
        callq   foo(int) # Pushes address of next instruction onto stack,
                         # populating 'returnAddress' member of 'stack_frame'.
                         # Then jumps to 'callee' address(i.e. function foo).
        addl    $1, %eax
        addq    $16, %rsp
        popq    %rbp
        retq

foo:                                    # @foo(int)
        pushq   %rbp       # Push rbp (stack_frame ptr) onto stack (populates
↪'nextFrame' member)
        movq    %rsp, %rbp # Update rbp to point to new stack_frame
        subq    $16, %rsp. # Reserve an additional 16 bytes of stack-space
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edi
        callq   bar(int)
        addl    $2, %eax
        addq    $16, %rsp. # Restore rsp
        popq    %rbp       # Pop rbp from stack
        retq               # Pop return address from top of stack and jump to it
```

因此我们可以通过 `_builtin_frame_address(0)` 得到当前函数的 frame pointer 值，解引用 frame pointer 即可得到 nextFrame 和 returnAddress，不断重复，这样我们就能获取到 stack trace 了。

一个简单的 unwinding 代码示例实现 test_unwind.cpp 如下：

```
1  #include <stdio.h>
2
3  __attribute__((noinline)) void fast_unwind() {
4    unsigned long *frame = (unsigned long *)__builtin_frame_address(0);
5    for (;;) {
6      printf("frame pointer is: %p\n", frame);
7      unsigned long *pc = (unsigned long*)frame[1];
8      printf("pc is: %p\n", pc);
9      unsigned long *new_frame = (unsigned long *)(*frame);
10      if (*new_frame <= *frame) break;
11      frame = new_frame;
12    }
13  }
```

(continues on next page)

(continued from previous page)

```
14
15  __attribute__((noinline)) int bar(int n) {
16    if (n <= 0)
17      return 0;
18    if (n == 1)
19      return 1;
20    fast_unwind();
21    return bar(n-1) + bar(n-2);
22  }
23
24  __attribute__((noinline)) int foo(int n) {
25    return bar(n)+2;
26  }
27
28  int main() {
29    return foo(2)+1;
30  }
```

```
$ clang++ test_unwind.cpp -g -no-pie -fno-omit-frame-pointer && ./a.out
frame pointer is: 0x7ffcbde94c10
pc is: 0x4011ec
frame pointer is: 0x7ffcbde94c30
pc is: 0x401233
frame pointer is: 0x7ffcbde94c50
pc is: 0x401259

$ llvm-symbolizer -iCfe a.out 0x4011ec 0x401233 0x401259
bar(int)
test-stack-trace/test_unwind.cpp:21:14
foo(int)
test-stack-trace/test_unwind.cpp:25:16
main
test-stack-trace/test_unwind.cpp:29:16
```

但是这种基于 frame pointer 实现 stack unwinding 的方式有很大的局限性：编译器默认只有在 O0 优化等级下会添加 `-fno-omit-frame-pointer` 编译选项，并且预留一个寄存器用于存储 frame pointer 与不预留一个寄存器存储 frame pointer 相比会有额外的性能开销。

DWARF Call Frame Information

另一种 stack unwinding 的实现方式就是基于 DWARF Call Frame Information 来实现的，由 compiler/assembler/linker/libunwind 提供相应支持。

我们还是以例子进行说明：

```
$ cat test.cpp
__attribute__((noinline)) int bar(int n) {
  if (n <= 0)
    return 0;
  if (n == 1)
    return 1;
  return bar(n-1) + bar(n-2);
}

__attribute__((noinline)) int foo(int n) {
  return bar(n)+2;
}

int main() {
  return foo(2)+1;
}

# 生成 test_unwind.s
$ clang++ -O1 test.cpp -S
```

我们可以在 test.s 中看到.cfi_def_cfa_offset, .cfi_offset 这样的 CFI directives，assembler/linker 会根据这些 CFI directives 生成.eh_frame section，最终用于 stack unwinding。

例如在汇编文件 test.s 中函数 bar 对应的汇编代码中有如下内容：

```
_Z3bari:                                # @_Z3bari
    .cfi_startproc
# %bb.0:                                # %entry
    pushq    %rbp
    .cfi_def_cfa_offset 16
    pushq    %rbx
    .cfi_def_cfa_offset 24
    pushq    %rax
    .cfi_def_cfa_offset 32
    .cfi_offset %rbx, -24
    .cfi_offset %rbp, -16
    testl    %edi, %edi
    jle    .LBB0_1
```

我们手动将 `.cfi_offset %rbp, -16` 修改为 `.cfi_offset %rbp, -24`，然后将 test.s 编译为可执行文件，并用 gdb 调试看下会有什么影响：

```
$ clang test.s
$ gdb ./a.out
(gdb) b bar
(gdb) r
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) disassemble
Dump of assembler code for function _Z3bari:
   0x0000000000401110 <+0>:    push   %rbp
   0x0000000000401111 <+1>:    push   %rbx
   0x0000000000401112 <+2>:    push   %rax
=> 0x0000000000401113 <+3>:    test   %edi,%edi
(gdb) i r rbx rbp
rbx            0x0                 0
rbp            0x401170            0x401170 <__libc_csu_init>
(gdb) bt
#0  0x0000000000401110 in bar(int) ()
#1  0x0000000000401156 in foo(int) ()
#2  0x000000000040116b in main ()
(gdb) f 1
#1  0x0000000000401156 in foo(int) ()
(gdb) i r rbx rbp
rbx            0x0                 0
rbp            0x0                 0x0
```

我们让程序停在 bar 函数中 `test %edi,%edi` 处，然后运行程序。可以看到在断点处 rbp 的值是 0x401170，rbx 的值是 0，跳转至 frame 1 后，再次看 rbp 的值，此时变为了 0。然而应当跳转至 frame 1 处和 rbp 的值也应该是 0x401170，变成 0 是因为我们将汇编文件 test.s 中 `.cfi_offset %rbp, -16` 修改为了 `.cfi_offset %rbp, -24`。而 cfi_offset -24 处存储的是%rbx 的值，%rbx 的值是 0，所以 gdb 将 frame 1 的%rbp 的值恢复为了 0。

这样我们通过这样一个简单的例子管中窥豹了解了基于 DWARF Call Frame Information 的 stack unwinding。

Sanitizer stack trace

Sanitizer runtime 在 stack unwinding 时，有两种策略：fast unwind 和 slow unwind，sanitizer runtime 优先使用 fast unwind。

在 sanitizer runtime 中有很多地方都需要收集 stack trace，例如在 malloc/free 被调用时收集 stack trace。在 ASan runtime library 中 intercptor malloc/free 函数实现中就通过宏 `GET_STACK_TRACE_MALLOC` 和 `GET_STACK_TRACE_FREE` 来获取 stack trace 保存至 BufferedStackTrace 类型的变量 stack 中，然后将 stack 作为参数传给函数 `asan_malloc`, `asan_free` 保存起来。

```
// compiler-rt/lib/asan/asan_malloc_linux.cpp

INTERCEPTOR(void*, malloc, uptr size) {
  if (DlsymAlloc::Use())
    return DlsymAlloc::Allocate(size);
  ENSURE_ASAN_INITED();
  GET_STACK_TRACE_MALLOC;
  return asan_malloc(size, &stack);
}

INTERCEPTOR(void, free, void *ptr) {
  if (DlsymAlloc::PointerIsMine(ptr))
    return DlsymAlloc::Free(ptr);
  GET_STACK_TRACE_FREE;
  asan_free(ptr, &stack, FROM_MALLOC);
}
```

`GET_STACK_TRACE_MALLOC` 和 `GET_STACK_TRACE_FREE` 宏展开后经过一系列的调用，最终执行的是 `BufferedStackTrace::Unwind()`：

`BufferedStackTrace::Unwind()` 函数的各个参数含义如下：

- max_depth：设置 unwind 最大回溯的深度。
- pc: the pc will be in the position 0 of the resulting stack trace. 即 unwind 起始处的 pc，是通过 `__builtin_return_address(0)` 得到的。
- bp: the bp may refer to the current frame or to the caller's frame. 即 unwind 起始处的 frame address，是通过 `__builtin_frame_address(0)` 得到的。
- context: 通常为 nullptr。在 Android lollipop 版本之前，从 signal handler 中 unwind 要基于 libcorkscrew.so，需要用到 signal handler 提供的 context 参数。
- stack_top, stack_bottom: unwind 起始处所在线程的线程栈底和线程栈顶，主要用于判断采取 fast unwind 时 unwind 过程何时终止。
- request_fast_unwind: 表示是使用 fast unwind 还是 slow unwind。在环境变量 ASAN_OPTIONS 中可以设置运行时参数 fast_unwind_on_check, fast_unwind_on_fatal, fast_unwind_on_malloc，sanitizer 会根据这些参数

的设置以及运行环境是否支持选择使用 fast unwind 还是 slow unwind。

- fast_unwind_on_check: If available, use the fast frame-pointer-based unwinder on internal CHECK failures. Defaults to false.
- fast_unwind_on_fatal: If available, use the fast frame-pointer-based unwinder on fatal errors. Defaults to false.
- fast_unwind_on_malloc: If available, use the fast frame-pointer-based unwinder on malloc/free. Defaults to true.

```
// compiler-rt/lib/sanitizer_common/sanitizer_stacktrace_libcdep.cpp

void BufferedStackTrace::Unwind(u32 max_depth, uptr pc, uptr bp, void *context,
                                uptr stack_top, uptr stack_bottom,
                                bool request_fast_unwind) {
  // Ensures all call sites get what they requested.
  CHECK_EQ(request_fast_unwind, WillUseFastUnwind(request_fast_unwind));
  top_frame_bp = (max_depth > 0) ? bp : 0;
  // Avoid doing any work for small max_depth.
  if (max_depth == 0) {
    size = 0;
    return;
  }
  if (max_depth == 1) {
    size = 1;
    trace_buffer[0] = pc;
    return;
  }
  if (!WillUseFastUnwind(request_fast_unwind)) {
#if SANITIZER_CAN_SLOW_UNWIND
    if (context)
      UnwindSlow(pc, context, max_depth);
    else
      UnwindSlow(pc, max_depth);
    // If there are too few frames, the program may be built with
    // -fno-asynchronous-unwind-tables. Fall back to fast unwinder below.
    if (size > 2 || size >= max_depth)
      return;
#else
    UNREACHABLE("slow unwind requested but not available");
#endif
  }
  UnwindFast(pc, bp, stack_top, stack_bottom, max_depth);
}
```

UnwindFast

UnwindFast 其实就是基于 frame pointer 的 unwind，具体实现如下：

```
// llvm-project/compiler-rt/lib/sanitizer_common/sanitizer_stacktrace.cpp
void BufferedStackTrace::UnwindFast(uptr pc, uptr bp, uptr stack_top,
                                    uptr stack_bottom, u32 max_depth) {
  // TODO(yln): add arg sanity check for stack_top/stack_bottom
  CHECK_GE(max_depth, 2);
  const uptr kPageSize = GetPageSizeCached();
  trace_buffer[0] = pc;
  size = 1;
  if (stack_top < 4096) return;  // Sanity check for stack top.
  uhwptr *frame = GetCanonicFrame(bp, stack_top, stack_bottom);
  // Lowest possible address that makes sense as the next frame pointer.
  // Goes up as we walk the stack.
  uptr bottom = stack_bottom;
  // Avoid infinite loop when frame == frame[0] by using frame > prev_frame.
  while (IsValidFrame((uptr)frame, stack_top, bottom) &&
         IsAligned((uptr)frame, sizeof(*frame)) &&
         size < max_depth) {
    uhwptr pc1 = frame[1];
    // Let's assume that any pointer in the 0th page (i.e. <0x1000 on i386 and
    // x86_64) is invalid and stop unwinding here.  If we're adding support for
    // a platform where this isn't true, we need to reconsider this check.
    if (pc1 < kPageSize)
      break;
    if (pc1 != pc) {
      trace_buffer[size++] = (uptr) pc1;
    }
    bottom = (uptr)frame;
    frame = GetCanonicFrame((uptr)frame[0], stack_top, bottom);
  }
}

static inline uhwptr *GetCanonicFrame(uptr bp,
                                      uptr stack_top,
                                      uptr stack_bottom) {
  CHECK_GT(stack_top, stack_bottom);
  return (uhwptr*)bp;
}

// Check if given pointer points into allocated stack area.
static inline bool IsValidFrame(uptr frame, uptr stack_top, uptr stack_bottom) {
  return frame > stack_bottom && frame < stack_top - 2 * sizeof (uhwptr);
```

(continues on next page)

(continued from previous page)

```
}
```

UnwindSlow

UnwindSlow 就是基于 libunwind 提供的接口 `_Unwind_Backtrace` 来实现的 unwind：

```
// llvm-project/compiler-rt/lib/sanitizer_common/sanitizer_unwind_linux_libcdep.cpp
void BufferedStackTrace::UnwindSlow(uptr pc, u32 max_depth) {
  CHECK_GE(max_depth, 2);
  size = 0;
  UnwindTraceArg arg = {this, Min(max_depth + 1, kStackTraceMax)};
  _Unwind_Backtrace(Unwind_Trace, &arg);
  // We need to pop a few frames so that pc is on top.
  uptr to_pop = LocatePcInTrace(pc);
  // trace_buffer[0] belongs to the current function so we always pop it,
  // unless there is only 1 frame in the stack trace (1 frame is always better
  // than 0!).
  // 1-frame stacks don't normally happen, but this depends on the actual
  // unwinder implementation (libgcc, libunwind, etc) which is outside of our
  // control.
  if (to_pop == 0 && size > 1)
    to_pop = 1;
  PopStackFrames(to_pop);
  trace_buffer[0] = pc;
}

struct UnwindTraceArg {
  BufferedStackTrace *stack;
  u32 max_depth;
};

_Unwind_Reason_Code Unwind_Trace(struct _Unwind_Context *ctx, void *param) {
  UnwindTraceArg *arg = (UnwindTraceArg*)param;
  CHECK_LT(arg->stack->size, arg->max_depth);
  uptr pc = Unwind_GetIP(ctx);
  const uptr kPageSize = GetPageSizeCached();
  // Let's assume that any pointer in the 0th page (i.e. <0x1000 on i386 and
  // x86_64) is invalid and stop unwinding here.  If we're adding support for
  // a platform where this isn't true, we need to reconsider this check.
  if (pc < kPageSize) return UNWIND_STOP;
  arg->stack->trace_buffer[arg->stack->size++] = pc;
  if (arg->stack->size == arg->max_depth) return UNWIND_STOP;
```

(continues on next page)

(continued from previous page)

```
  return UNWIND_CONTINUE;
}
```

_Unwind_Backtrace 的函数原型如下。

```
// _Unwind_Backtrace() is a gcc extension that walks the stack and calls the
// _Unwind_Trace_Fn once per frame until it reaches the bottom of the stack
// or the _Unwind_Trace_Fn function returns something other than _URC_NO_REASON.
typedef _Unwind_Reason_Code (*_Unwind_Trace_Fn)(struct _Unwind_Context *,
                                                void *);
extern _Unwind_Reason_Code _Unwind_Backtrace(_Unwind_Trace_Fn, void *);
```

_Unwind_Backtrace 在 stack unwinding 时，对于每一个 frame 都会调用回调函数 Unwind_Trace 将此 frame 的 PC/IP 指令地址保存至 BufferedStackTrace 的成员变量 trace_buffer 中。

StackDepot

本节我们来看下 sanitizer runtime 是如何保存 stack trace 的。

我们在本文前面提到 intercetpr malloc 调用 asan_malloc 来进行内存分配，而 asan_malloc 就是对 __asan::Allocator::Allocate 函数的一层包装。

```
// compiler-rt/lib/asan/asan_allocator.cpp
void *asan_malloc(uptr size, BufferedStackTrace *stack) {
  return SetErrnoOnNull(instance.Allocate(size, 8, stack, FROM_MALLOC, true));
}

void *Allocate(uptr size, uptr alignment, BufferedStackTrace *stack,
               AllocType alloc_type, bool can_fill) {
  ...
  m->alloc_context_id = StackDepotPut(*stack);
  ...
}
```

注意到 __asan::Allocator::Allocate 函数是通过调用函数 StackDepotPut 将 stack unwinding 获取的 stack trace 保存起来。StackDepotPut 会返回一个 context_id，维护 context_id 与 stack trace 的映射关系，后续可以通过 context_id 找到对应的 stack trace。

存储 stack trace 的核心数据结构就是 StackDepot，代码位于：

- compiler-rt/lib/sanitizer_common/sanitizer_stackdepotbase.h
- compiler-rt/lib/sanitizer_common/sanitizer_stackdepot.h
- compiler-rt/lib/sanitizer_common/sanitizer_stackdepot.cpp

```
// FIXME(dvyukov): this single reserved bit is used in TSan.
typedef StackDepotBase<StackDepotNode, 1, StackDepotNode::kTabSizeLog>
    StackDepot;
static StackDepot theDepot;

u32 StackDepotPut(StackTrace stack) {
  StackDepotHandle h = theDepot.Put(stack);
  return h.valid() ? h.id() : 0;
}

StackTrace StackDepotGet(u32 id) {
  return theDepot.Get(id);
}
```

- `StackDepotPut` 函数，参数类型是 StackTrace（BufferedStackTrace 是 StackTrace 的子类），返回值类型是 u32。存储 stack trace，返回一个 id，id 与 stack trace 是一一对应关系。
- `StackDepotGet` 函数，参数类型是 u32，返回值类型是 StackTrace。根据 id 返回对应的 stack trace。

StackDepotNode 和 StackDepotBase/StackDepot 的成员变量如下所示。

- StackDepotNode。成员变量 StackDepotNode *link; 存储指向下一个 StackDepotNode 的指针，即多个 StackDepotNode 组成一个链表。成员变量 id 用于标识该 StackDepotNode/StackTrace。成员变量 tag 的可能取值是 TAG_UNKNOWN(0), TAG_ALLOC(1), TAG_DEALLOC(2), TAG_CUSTOM(100) 表示 stack trace 的来源。成员变量 size 就是用于表示 stack trace 的深度，成员变量 stack 是个数组，数组每个元素用于存储 stack trace 每一帧的 pc。
- StackDepot/StackDepotBase。StackDepot 通过 hash table 来存储 StackDepotNode，hash table 维护了 1<<20 个 tab，每 1<<12 个 tab 又组成了一个 part。每个 tab 存储的是指向 StackDepotNode 链表第一个元素的指针。

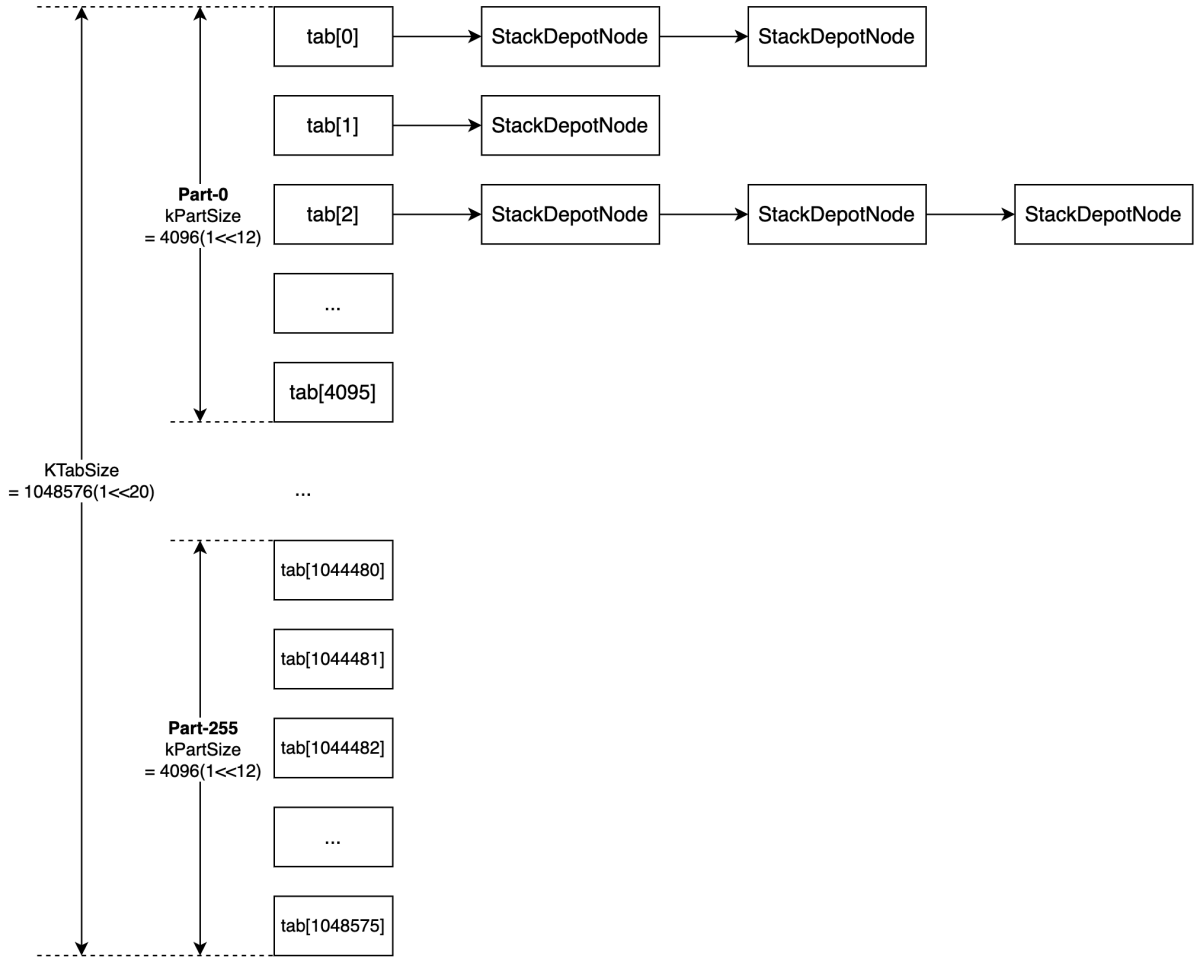
```
struct StackDepotNode {
  StackDepotNode *link;
  u32 id;
  atomic_uint32_t hash_and_use_count; // hash_bits : 12; use_count : 20;
  u32 size;
  u32 tag;
  uptr stack[1];  // [size]
  ...
};

template <class Node, int kReservedBits, int kTabSizeLog>
class StackDepotBase {
  atomic_uintptr_t tab[kTabSize];   // Hash table of Node's.
  atomic_uint32_t seq[kPartCount];  // Unique id generators.
  ...
```

(continues on next page)

(continued from previous page)

```
};
```

StackDepot hash table 示意图如下：



- 对于一个给定的 stack trace，首先计算出该 stack trace 的 hash 值记作 h，然后计算 `h % kTabSize` 找到存储该 stack trace 的 tab。判断当前给定的 stack trace 是否已经在 `tab[h % kTabSize]` 对应的链表中。如果不在，就申请一块内存，在这块内存上根据给定的 stack trace 构造 StackDepotNode，然后将该 StackDepotNode 插入到 `tab[h % kTabSize]` 对应的链表开头。那么 StackDepotNode 的 id 是怎么计算的呢？每 1<<12 个 tab 组成了一个 part，数组 `seq[kPartCount]` 存储的是每个 part 中当前已经存储了多少个 StackDepotNode。通过计算 `h % kTabSize / kPartSize` 找到存储该 stack trace 的 tab 所在的 part。对于一个新的 StackDepotNode 对应的 id 就是 `(seq[part] + 1) | (part << kPartShift)`。
- 对于一个给定的 id，首先通过 `uptr part = id >> kPartShift;` 找到该 id 对应的 StackDepotNode 位于哪个 part。遍历该 part 中的 1 <<12 个 tab，在每个 tab 链表中寻找是否存在与给定 id 相等的 StackDepotNode。

关于 sanitizer runtime 是如何保存 stack trace 的，可以仔细阅读下 StackDepot 的代码实现，这部分代码非常值

得学习，这里就不一一贴代码了。

References

1. Stack unwinding | MaskRay
2. https://developers.facebook.com/blog/post/2021/09/23/async-stack-traces-folly-synchronous-asynchronous-stack-traces/

- genindex
- modindex
- search

7.5 ThreadSanitizer

7.5.1 Prologue

据我所知，目前中文互联网上介绍 ThreadSanitizr(AKA TSan) 原理的文章很少，只有 ThreadSanitizer——跟 data race 说再见 写的很清晰。

但是实际上这篇知乎文章是基于 **ThreadSanitizer V1** 的论文 ThreadSanitizer: data race detection in practice (WBIA ’09) 来讲解的。ThreadSanitzer V1 是基于 Valgrind 实现的，基于 happen-before 和 lockset 算法。

而目前集成在 GCC/Clang 中的 ThreadSanitizer 实际上已经是 V2 版本了，底层算法也与 V1 不同，根据 AddressSanitizer, ThreadSanitizer, and MemorySanitizer: Dynamic Testing Tools for C++ (GTAC’2013)，TSan V2 使用的 fast happens-before 算法，**类似**于 FastTrack(PLDI’09) 中提出的算法。

所以本节关于 TSan 的学习笔记包含两篇文章：

1. 第一篇是对 FastTrack: efficient and precise dynamic race detection (PLDI’09) 这篇论文的学习笔记
2. 第二篇是从 TSan 代码实现的角度，理解 TSan 背后的检测算法

P.S.1 TSan runtime 最新是 V3 版本，截止本文撰写时 LLVM14 还没有发布，应该从 LLVM14 开始 TSan 默认使用 V3 runtime 。

P.S.2 我就 TSan 的底层算法请教了 TSan 作者 Dmitry Vyukov，得到回复是：TSan V2/V3 使用的算法都是类似于作者的另外一个 data race 检测工具 Relacy’08 ，但是该工具没有论文，只有源码。

P.S.3 Konstantin Serebryany 诚不欺我，TSan V2 使用算法确实**类似** FastTrack(PLDI’09) 算法，学习 FastTrack(PLDI’09) 对于理解 TSan V2 的底层算法非常有帮助。

7.5.2 Dissecting ThreadSanitizer Algorithm

本文深入剖析 ThreadSanitizer(V2) 检测 Data Race 背后的算法原理。

Introduction

ThreadSanitizer(AKA TSan) 是一个集成在 GCC 和 Clang 中的动态分析工具，能够检测 C++ 代码中大多数的数据竞争 (data race) 。它由编译时插桩和运行时库两部分组成，通过编译和链接时添加参数 `-fsanitize=thread`，就可以在运行时检测 data race 。

Data Race

TSan 是检测 data race 的动态分析工具。我们先看下 data race 指的是什么?

Data Race：两个线程 **concurrently** 访问了**同一个内存位置 (memory location)**，并且两个线程的访存操作中**至少一个是写操作**。

注：关于 race condition 和 data race 的区别，见 Race Condition vs. Data Race –Embedded in Academia

例如下述代码：两个线程并发地修改整型全局变量 Global 存在 data race。两个线程执行结束后，全局变量 Global 的值可能是 1 也可能是 2。如果是在读写 STL 容器时存在 data race，则可能导致更严重的后果，比如内存破坏、程序崩溃。

```
int Global;

void Thread1() {
  Global = 1;
}

void Thread2() {
  Global = 2;
}
```

根据 data race 的定义，判断代码中是否存在 data race 需要考虑 3 个条件：

1. 两个线程访问的是否为**同一个 memory location**
2. 两个线程的访存操作中**至少有一个是写操作**
3. 两个线程的访存操作是否 **concurrent**

其中前两个条件很容易判断，所以检测 data race 的要解决的关键问题就是怎么判断两个访存操作是否 **concurrent！**

Happen-Before & Concurrent

在介绍如何判断两次访问操作是否是 concurrent 之前，我们需要先引入 happen-before 的定义。

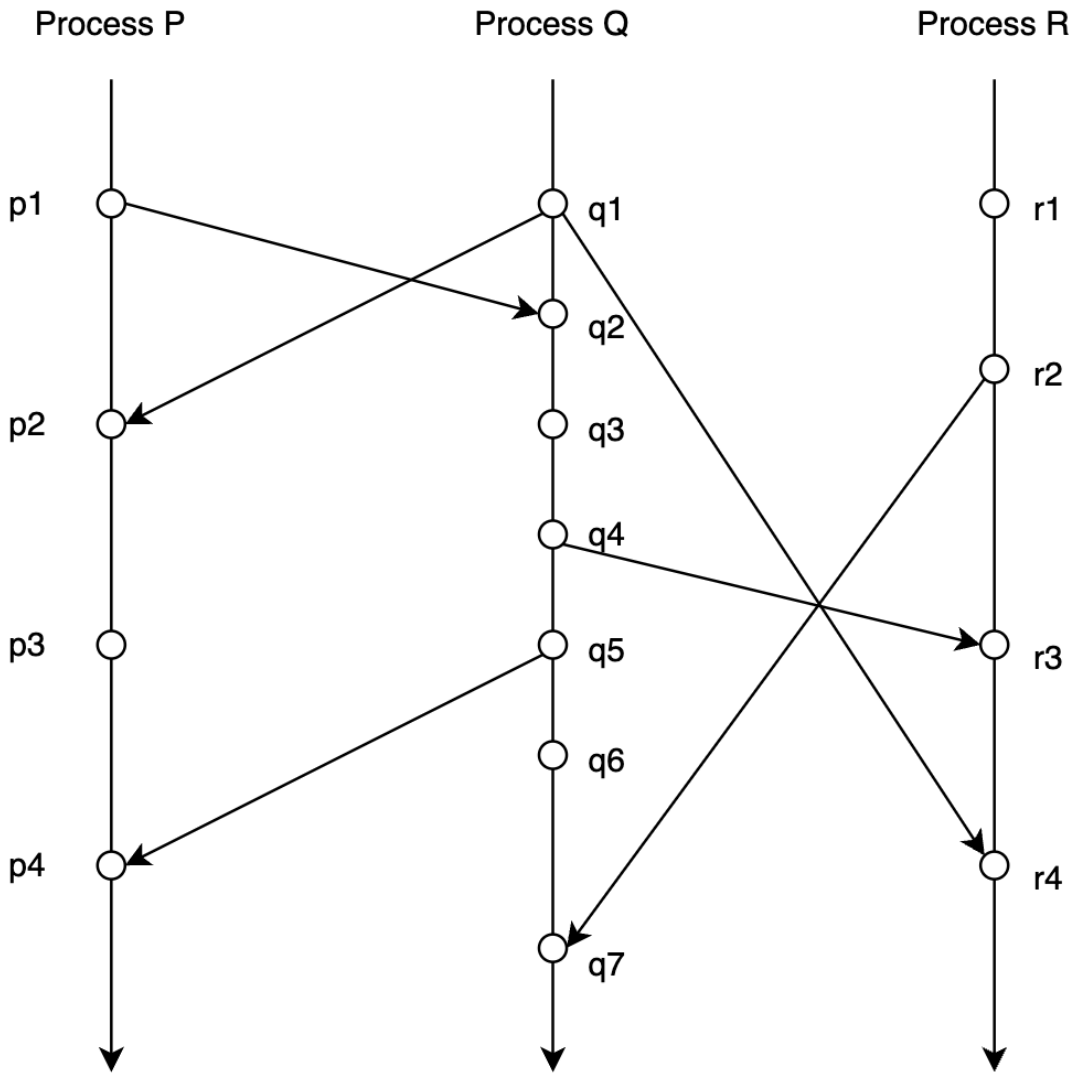
Happen-before 的定义最开始是在 Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system 中给出的，描述的是分布式系统中事件之间的一种偏序关系。

一个分布式系统是由一系列 processes 组成的，每个 process 又由一系列事件组成，不同的 process 之间是通过收发消息进行通信的。

Happen-before 关系（记作 $\rightarrow$）的定义：

1. 如果事件 a 和事件 b 是在同一个 process 中的事件，并且 a 早于 b 发生，那么 $a \rightarrow b$
2. 如果事件 a 和事件 b 是不同 process 中的事件，且 b 是 a 发送的消息的接收者，那么 $a \rightarrow b$
3. **Happen-before** 关系是一种严格**偏序**关系 (strict partial order)，即满足 transitive, irreflexive and antisymmetric
    1. Transitive。对于任意事件 a, b, c，如果 $a \rightarrow b$ 且 $b \rightarrow c$，那么有 $a \rightarrow c$
    2. Irreflexive。对于任意事件 a，都有 $a \nrightarrow a$
    3. Antisymmetric。对于任意事件 a, b，如果 $a \rightarrow b$，那么有 $b \nrightarrow a$

下面通过一个例子对 happen-before 进行说明：



上图是对一个分布式系统的某一次 trace：

- 3 条垂直线分别表示 3 个 process: P, Q, R
- 垂直线上的点表示事件，在同一条垂直线上纵坐标小的事件发生的时间早于纵坐标大的事件发生的时间。例如事件 p_1 早于事件 p_2 发生
- 连接 process 之间的线表示 process 之间通过收发消息进行通信，$p_1 \rightarrow q_2$ 表示 process P 于事件 p_1 向 process Q 发送消息，这个消息被 process Q 于事件 q_2 接收到

那么对于上图分布式系统 trace：

- 根据 **happen-before** 的定义能得出：$p_1 \rightarrow r_4$，这是因为 $p_1 \rightarrow q_2, q_2 \rightarrow q_4, q_4 \rightarrow r_3, r_3 \rightarrow r_4$，所以 $p_1 \rightarrow r_4$。即事件 $p1$ 一定是先于事件 $r4$ 发生，不管上述分布式系统事件运行多少次
- 尽管根据本次 trace 看来在时间上事件 q_3 是早于事件 p_3 发生的，但是 $q_3 \nrightarrow p_3$ 且 $p_3 \nrightarrow q_3$，即事件 q_3 和事件 p_3 之间是没有 happen-before 关系的。所以不能保证每一次运行，事件 q_3 都是早于事件 p_3 发生

的，也有可能在某一次 trace 中事件 p_3 是早于事件 q_3 发生的

理解了 happen-before 的定义后，我们给出 **concurrent** 的定义：如果 $a \nrightarrow b$ 且 $b \nrightarrow a$，那么称 a 和 b 是 **concurrent** 的。

这样我们就能够将判断两次访存操作之间是否 concurrent 转化为了判断两次访存操作之间是否存在 happen-before 关系。

那么如何判断两次访存操作之间是否存在 happen-before 关系呢？答案是 Vector Clock。在介绍 Vector Clock 之前，我们需要先了解下 Lamport Logical Clock。

Lamport Logical Clock

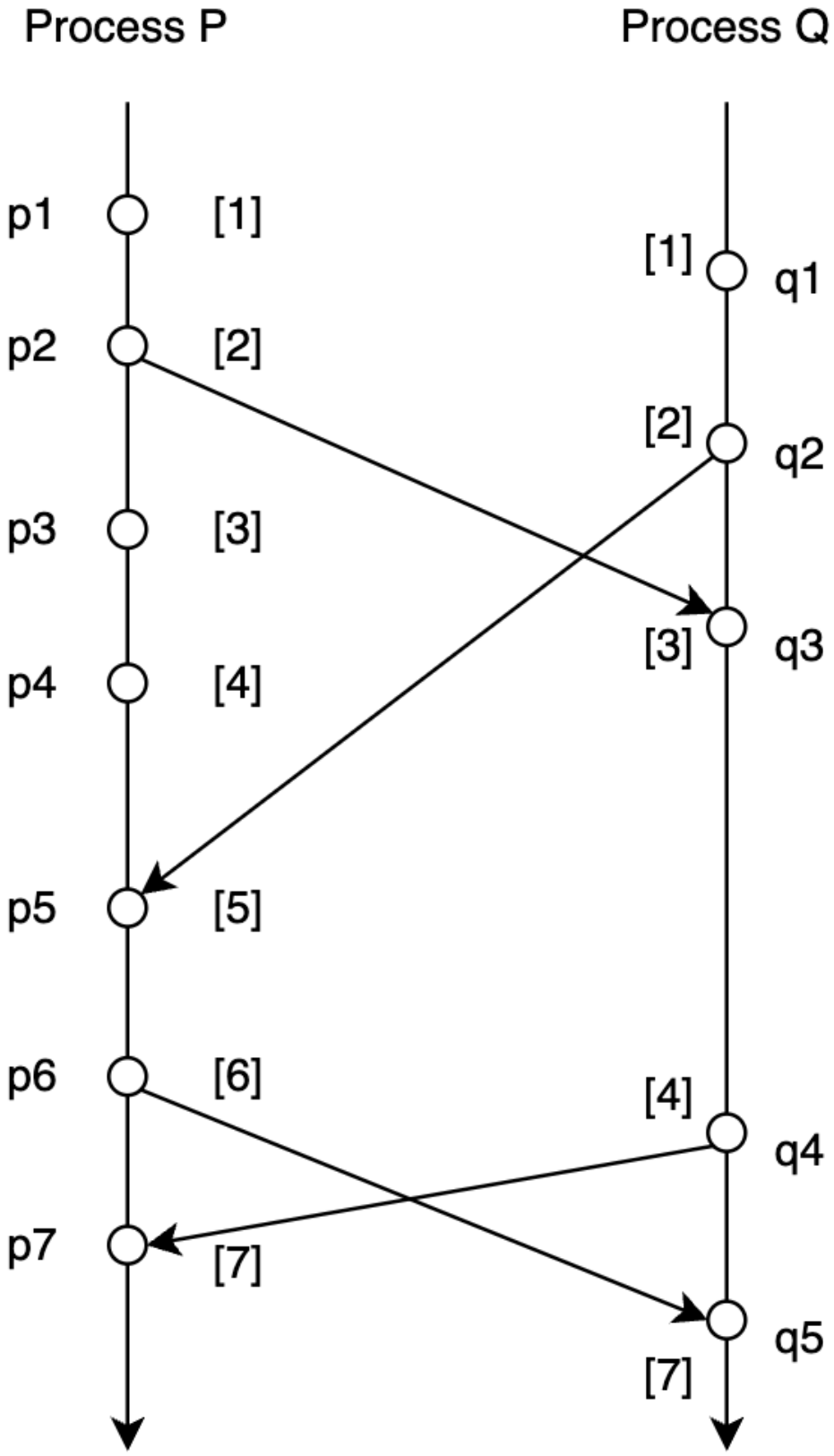
Lamport logical clock 算法是由 Leslie Lamport 在 Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system 中提出的一种简单的逻辑时钟算法，用于描述分布式计算机系统中事件的偏序关系。

算法如下：

1. 每个 Process P_i 都持有一个逻辑时钟 $Clock_i$，process P_i 在每次本地事件发生之前，都将 $Clock_i$ 自增 1
2. Process P_i 向其他 Process 发送消息时，先执行步骤 1，然后将 $Clock_i$ 的值包含在消息中一并发送出去
3. Process P_j 接收到 Process P_i 发送来的消息时，获取消息中携带的 $Clock_i$ 的值，与自身的 $Clock_j$ 取最大值，然后在认为收到消息之前将 $Clock_j$ 自增 1

根据 lamport logical clock 算法流程和 happen-before 定义易得：对于任意两个事件，如果事件 a **happen-before** 事件 b，那么 $Clock(a) < Clock(b)$。

下面通过例子来说明 lamport logical clock 算法流程：


Process P
Process Q
p1
[1]
[1]
q1
p2
[2]
[2]
q2
p3
[3]
[3]
q3
p4
[4]
p5
[5]
p6
[6]
[4]
q4
p7
[7]
q5
[7]

- 初始时，Process P 和 Process Q 的逻辑时钟的值都为 0
- Process P 发生本地事件 p_1，逻辑时钟 $Clock_p$ 的值由 0 更新为 1。Process Q 发生本地事件 q_1，逻辑时钟 $Clock_q$ 的值由 0 更新为 1
- Process P 于事件 p_2 向 Process Q 发送消息，先自增逻辑时钟 $Clock_p$ 的值，由 1 更新为 2。然后将此时 $Clock_p$ 的值（即 2）包含在消息中一并发送出去
- Process Q 接收到 Process P 发送来的消息，获取消息中携带的逻辑时钟的值（即 2），首先将逻辑时钟 $Clock_q$ 的值更新为消息中携带的逻辑时钟的值与此时自身逻辑时钟 $Clock_q$ 的值的最大值（即 $Clock_q$ 更新为 2 和 2 的最大值，还是 2），然后再将 $Clock_q$ 自增 1（即 $Clock_q$ 的值由 2 更新为 3 ）
- ...

但是 lamport logical clock 是存在局限性的：

- 只能保证：如果 $a \rightarrow b$，那么 $Clock(a) < Clock(b)$
- 但是 $Clock(a) < Clock(b)$ 不意味着 $a \rightarrow b$，即我们不能基于 $Clock(a) < Clock(b)$ 来判定事件 a **happen-before** 事件 b

例如 $Clock_p(p1)= 1 < 2 = Clock_q(q2) $，但是实际上 p_1 **happen-before** p_2 是不成立！也就是说，事件 p_1 和事件 p_2 之间谁都可能早于另外一个事件发生。

我们前面将 data race 检测问题转化为了判断两次访存操作之间是否存在 happen-before 关系的问题。但是由于 lamport logical lock 的局限性，我们不能直接将 lamport logic clock 应用于 data race 的检测。

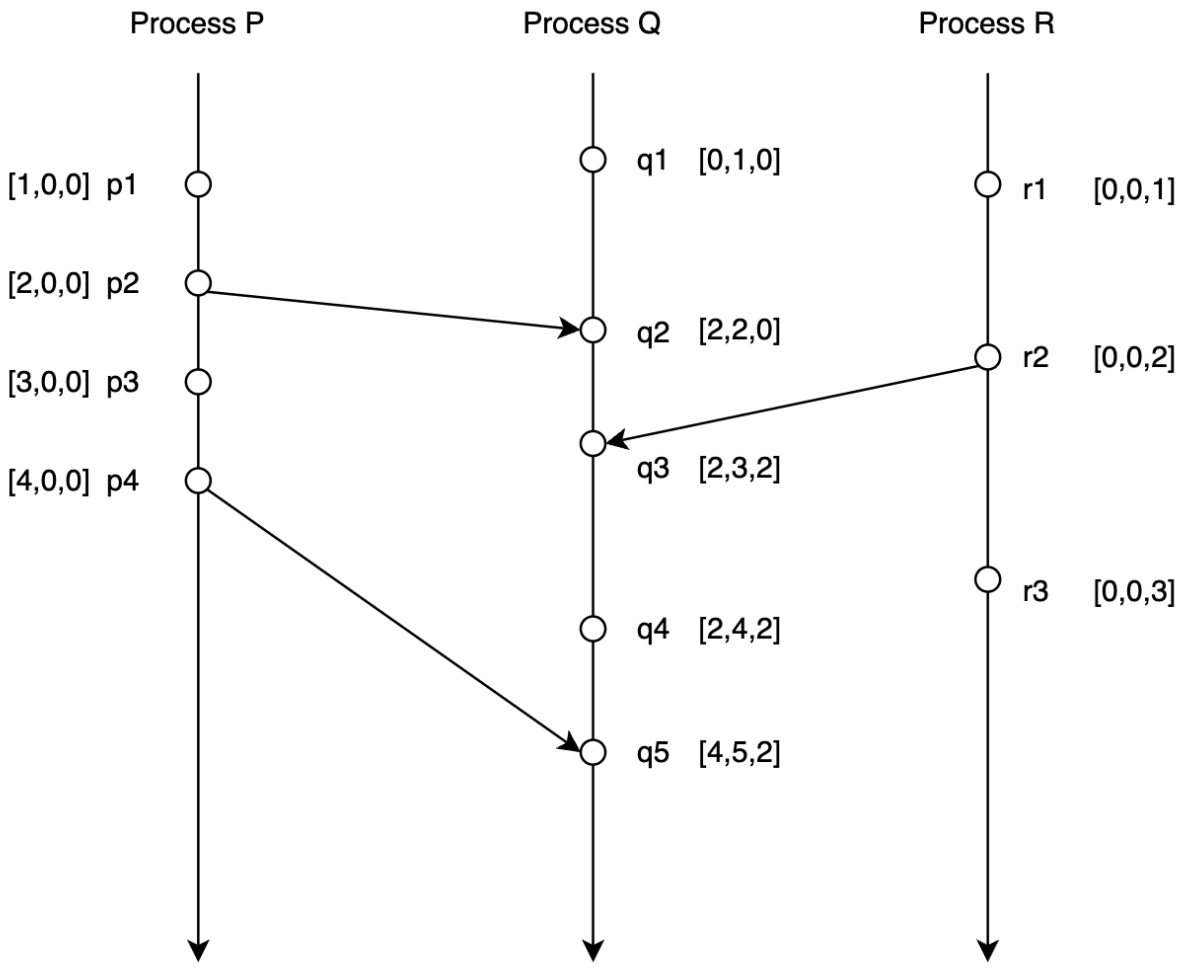
Vector Clock

Colin Fidge 和 Friedemann Mattern 提出的 vector clock 解决了 lamport logic clock 的上述局限性

vector clock 算法如下：

- 每一个 process P_i 都对应一个 vector clock VC_i，VC_i 是由 n 个元素组成的向量，n 是分布式系统中 process 的数量。每个 process P_i 的 VC_i 都被初始化为 0
- 每当 process P_i 发生本地事件之前，更新 vector clock：$VC_i[i] = VC_i[i] + 1$
- Process P_i 向其他 Process 发送消息时，先更新 vector clock：$VC_i[i] = VC_i[i] + 1$，然后将 VC_i 的值包含在消息中
- process P_j 接收由 process P_i 发送来的 message，更新 vector clock：$VC_j[j] = VC_j[j] + 1, VC_j[k] = max(VC_j[k], VC_i[k])$ for all process k

下面还是通过一个例子来说明 vector clock 的算法流程：



- 初始时 $VC_1 = VC_2 = VC_3 = [0, 0, 0]$
- Process P 发生内部事件 p_1，更新 vector clock：$VC_1 = [0 + 1, 0, 0] = [1, 0, 0]$
- Process Q 发生内部事件 p_2，更新 vector clock：$VC_2 = [0, 0 + 1, 0] = [0, 1, 0]$
- Process R 发生内部事件 p_3，更新 vector clock：$VC_3 = [0, 0, 0 + 1] = [0, 0, 1]$
- process Q 于事件 q_2 接收由 process P 于事件 p_2 发送的消息，更新 vector clock：
  - $VC_1[1] = 1 + 1 = 2, VC_1 = [2, 0, 0]$
  - $VC_2[2] = 1 + 1 = 2, VC_2 = [0, 2, 0]$
  - $VC_2 = max(VC_1, VC_2) = [max(2, 0), max(0, 2), max(0, 0)] = [2, 2, 0]$
- ...

Vector clock 解决了 lamport logical clock 的局限性，满足如下性质：

- 如果事件 a **happen-before** 事件 b，那么 $VC(a) < VC(b)$
- 如果 $VC(a) < VC(b)$，那么事件 a **happen-before** 事件 b

即 $p_a \to q_b \quad \text{iff} \quad VC_p(a) < VC_q(b)$

Vector clock 上的偏序关系如下：

- $VC_p = VC_q \quad \text{iff} \ \forall k, \ VC_p[k] = VC_q[k]$
- $VC_p \neq VC_q \quad \text{iff} \ k, \ VC_p[k] \neq VC_q[k]$
- $VC_p \leq VC_q \quad \text{iff} \ \forall k, \ VC_p[k] \leq VC_q[k]$
- $VC_p < VC_q \quad \text{iff} \ (VC_p \leq VC_q \ \text{and} \ VC_p \neq VC_q)$

根据 $p_a \to q_b \quad \text{iff} \quad VC_p(a) < VC_q(b)$ 这个性质，我们就能使用 vector clock 来判断两次访存操作之间是否存在 happen-before 关系，即能够基于 vector clock 算法来检测多线程程序中的 data race。

Data Race Detection

我们前面在介绍 lamport logic clock 和 vector clock 时都是以分布式系统中的事件之间序关系为背景进行介绍的。

实际上多线程程序也可以看作是一个分布式系统。我们对上述 vector clock 算法稍加修改，就可以应用于检测多线程程序中的 data race：

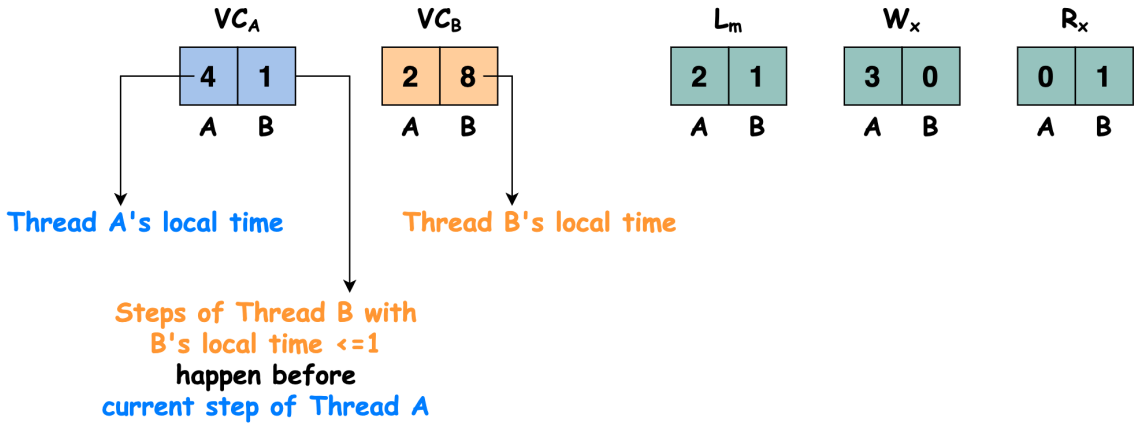
- 符号定义：
  - $\mathbb{VC}_{\not\models} \sqsubseteq \mathbb{VC}_{\not\models} \quad \textbf{iff} \quad \forall t. \, \mathbb{VC}_{\not\models}(t) \leq \mathbb{VC}_{\not\models}(t)$
  - $\mathbb{VC}_{\not\models} \sqcup \mathbb{VC}_{\not\models} \quad = \quad \lambda t. \, max(\mathbb{VC}_{\not\models}(t), \mathbb{VC}_{\not\models}(t))$
- 每个线程 t 都对应一个 vector clock $\mathbb{VC}_t$，初始值为 0。对于任意一个线程 u，$\mathbb{VC}_t(u)$ 记录的其实是与线程 t 的当前操作满足 happen-before 关系的线程 u 的上一次操作的 clock。如果把线程 u 的上一次操作记为 O_u，把线程 t 的当前操作记为 O_t，那么有 O_u happen-before O_t
- 每一个锁 m 都对应一个 vector clock $\mathbb{L}_m$
- 每一个变量 x 都对应两个 vector clock: $\mathbb{W}_x$ 和 $\mathbb{R}_x$。对于任意一个线程 t，$\mathbb{W}_x[t]$ 和 $\mathbb{R}_x[t]$ 记录了线程 t 对变量 x 的最后一次读/写的 clock
  - 线程 t 对变量 x 的读时，会将 $\mathbb{R}_x[t]$ 的值更新为 $\mathbb{VC}_t[t]$ 的值
  - 程序 t 对变量 x 的写时，会将 $\mathbb{W}_x[t]$ 的值更新为 $\mathbb{VC}_t[t]$ 的值
- 程序中执行同步和线程操作时，算法会更新相应的 vector clock：
  - $rel(t, m)$。线程 u 释放了锁 m，先将 $\mathbb{L}_m$ 的值更新为 $\mathbb{VC}_u$ 的值，再将 $\mathbb{VC}_u[u] + +$
  - $acq(t, m)$。线程 t 获取了锁 m，将 $\mathbb{VC}_t$ 的值更新为 $\mathbb{VC}_t \sqcup \mathbb{L}_m$ 的值
  - $fork(t, u)$。先将 $\mathbb{VC}_u$ 的值更新为 $\mathbb{VC}_u \sqcup \mathbb{VC}_t$，再将 $\mathbb{VC}_t[t] + +$
  - $join(t, u)$。先将 $\mathbb{VC}_t$ 的值更新为 $\mathbb{VC}_t \sqcup \mathbb{VC}_u$，再将 $\mathbb{VC}_u[u] + +$
- 判断是否存在 data race：

- 假设当前线程 u 读变量 x，如果满足 $\mathbb{W}_{\curvearrowright} \sqsubseteq \mathbb{VC}_{\approx}$，那么当前线程 u 对变量 x 的读与之前其他线程对变量 x 的写不存在 data race
- 假设当前线程 u 写变量 x，如果 $\mathbb{W}_{\curvearrowright} \sqsubseteq \mathbb{VC}_{\approx}$ 且 $\mathbb{R}_{\curvearrowright} \sqsubseteq \mathbb{VC}_{\approx}$ 那么当前线程 u 对变量 x 的写与之前其他线程对变量 x 的写和读不存在 data race

---

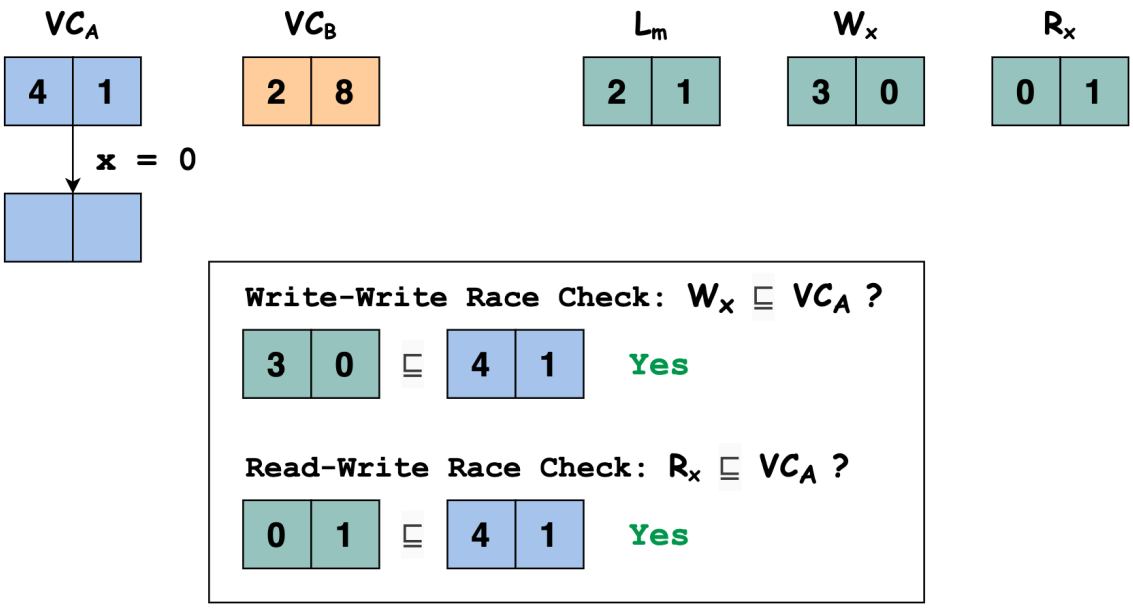
下面还是通过一个例子来说明如何应用 vector clock 检测多线程程序中的 data race：

- 考虑多线程程序中有两个线程 A 和 B，有一个锁 m，还有一个变量 x
- 线程 A 对应的 vector clock 记作 $\mathbb{VC}_A$，线程 B 对应的 vector clock 记作 $\mathbb{VC}_B$，锁 m 对应的 vector clock 记作 $\mathbb{L}_m$，变量 x 对应的写/读 vector clock 记作 $\mathbb{W}_x$ 和 $\mathbb{R}_x$
- 假设多线程程序某一个时刻的运行状态如下图所示：

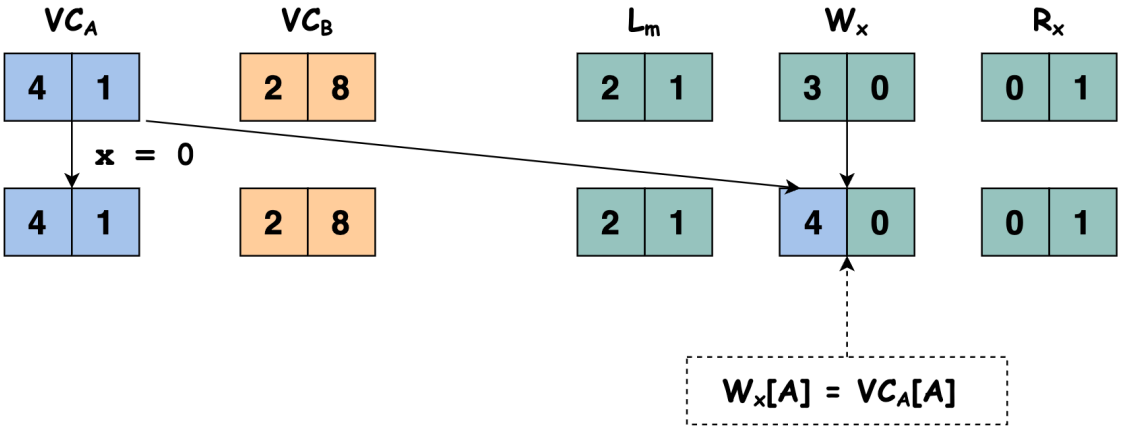


- 此时线程 A 执行语句 `x = 0` 写变量 x，我们需要检查当前线程对变量 x 的写与之前其他线程对变量 x 的写和读之间是否存在 data race，即判断 $\mathbb{W}_{\curvearrowright} \sqsubseteq \mathbb{VC}_{\approx}$ 和 $\mathbb{R}_{\curvearrowright} \sqsubseteq \mathbb{VC}_{\approx}$ 是否满足。

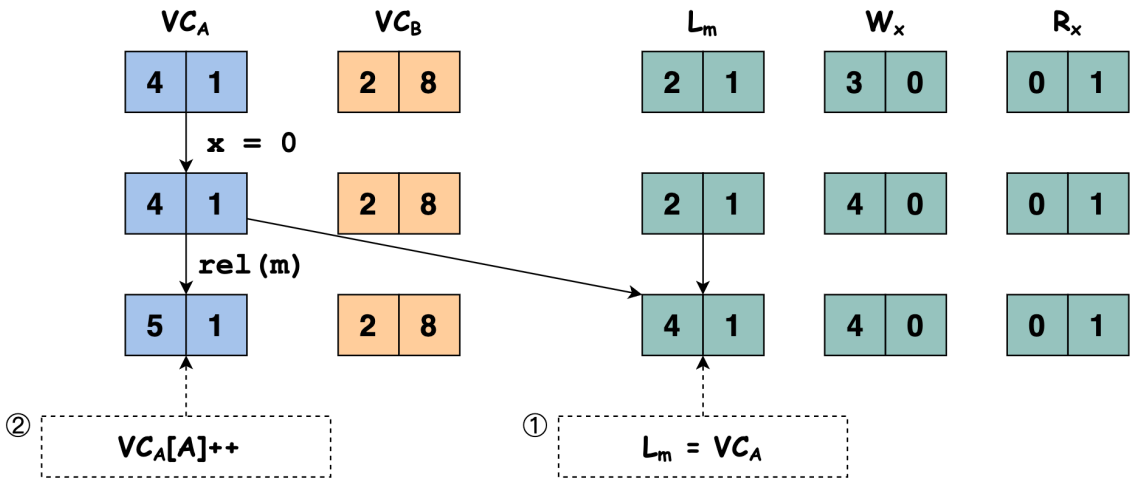
  显然 $[3,0] \sqsubseteq [4,1]$ 和 $[0,1] \sqsubseteq [4,1]$ 都满足，即本次线程 A 执行语句 `x = 0` 写变量 x 与之前其他线程对变量 x 的写和读之间不存在 data race



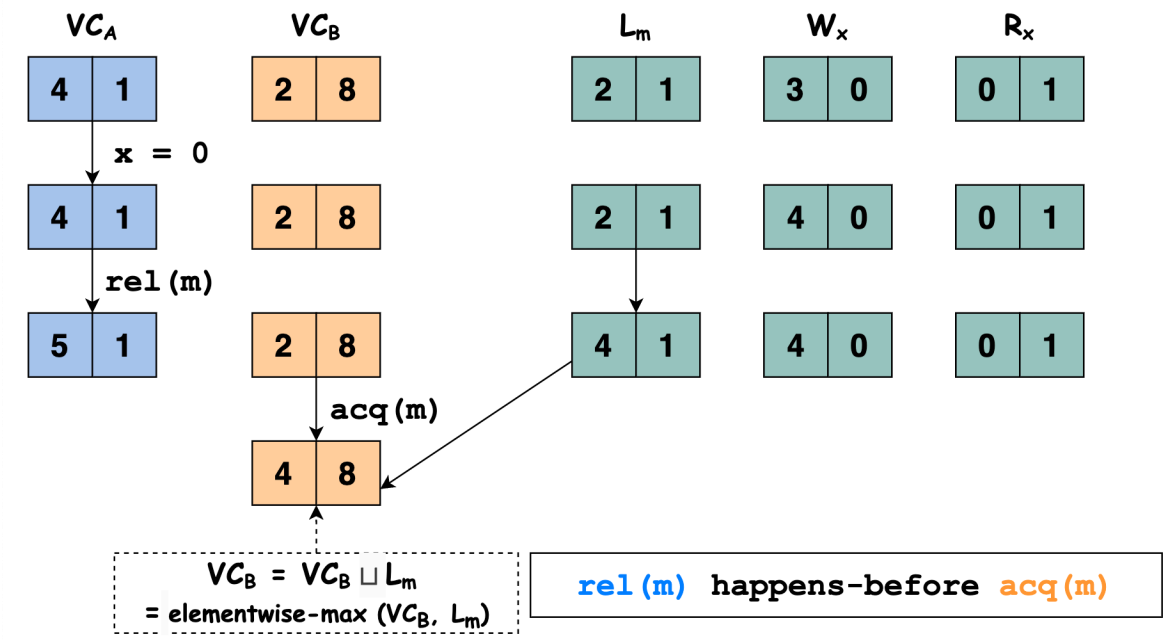
- 因为线程 A 执行语句 `x = 0` 写了变量 x，我们要更新 $\mathbb{W}_x$ 的值，将 $\mathbb{W}_x[A]$ 的值更新为 $\mathbb{VC}_A[A]$ 的值：



- 程序继续执行，此时线程 A 执行语句 `rel(m)` 释放锁 m，先将 $\mathbb{L}_m$ 的值更新为 $\mathbb{VC}_A$ 的值，再将 $\mathbb{VC}_A[A]$++



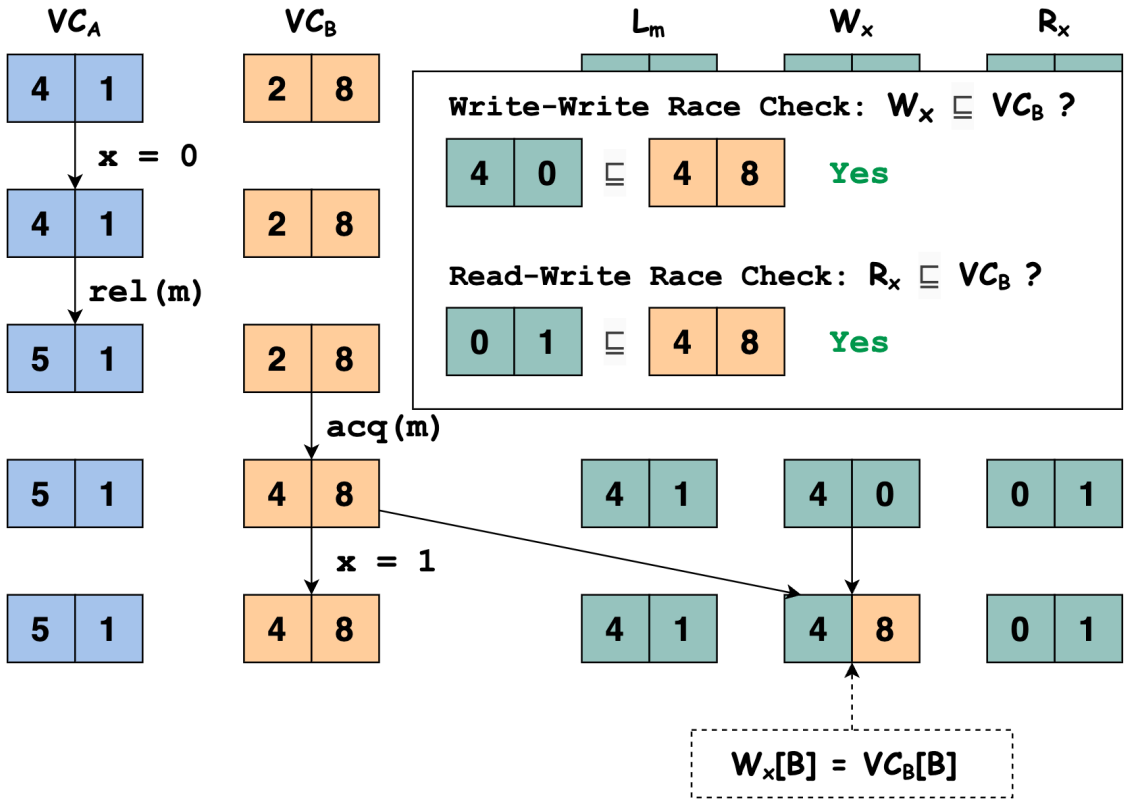
- 程序继续执行，此时线程 B 执行语句 `acq(m)` 申请锁 m，将 $\mathbb{VC}_B$ 的值更新为 $\mathbb{VC}_B \sqcup \mathbb{L}_m$ 的值



- 线程 B 执行语句 `x = 1` 写变量 x，我们需要检查当前线程对变量 x 的写与之前其他线程对变量 x 的写和读之间是否存在 data race，即判断 $\mathbb{W}_x \sqsubseteq \mathbb{VC}_{\approx}$ 和 $\mathbb{R}_x \sqsubseteq \mathbb{VC}_{\approx}$ 是否满足。

  显然 $[4,0] \sqsubseteq [4,8]$ 和 $[0,1] \sqsubseteq [4,8]$ 都满足，即本次线程 B 执行语句 `x = 1` 写变量 x 与之前其他线程对变量 x 的写和读之间不存在 data race。

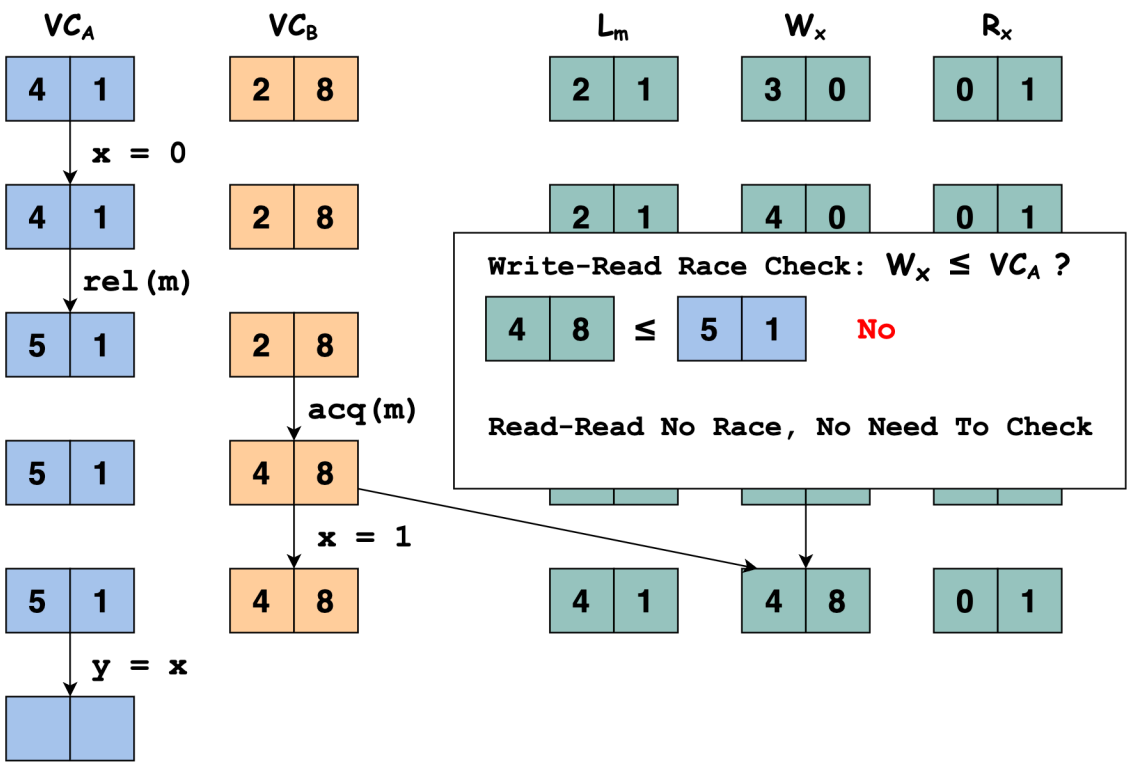
  因为线程 B 执行语句 `x = 1` 写变量 x，我们要更新 $\mathbb{W}_x$ 的值，将 $\mathbb{W}_x[B]$ 的值更新为 $\mathbb{VC}_B[B]$ 的值：



- 线程 A 执行语句 `y = x` 读变量 x，我们需要检查当前线程对变量 x 的读与之前其他线程对变量 x 的写之间是否存在 data race，即判断 $\mathbb{W}_{\curvearrowleft} \sqsubseteq \mathbb{VC}_{\approxeq}$ 是否满足。

  显然 $[4,8] \sqsubseteq [5,1]$ **不满足**，即本次线程 B 执行语句 `x = 1` 写变量 x 与之前其他线程对变量 x 的读之间**存在 data race**。

  即我们检测到线程 A 执行语句 `y = x` 读变量 x 与之前其他线程对变量 x 的写和读之间**存在 data race**。更具体地说是线程 A 执行语句 `y = x` 读变量 x 与之前线程 B 执行语句 `x = 1` 读变量 x 存在 data race。



ThreadSanitizer Internals

ThreadSanitizer 检测 data race 的思想其实就是基于 vector clock 算法的，只不过在实现时做了一些取舍。比如对于每一个变量 x，ThreadSanitizer 不会记录所有线程最近一次对变量 x 的读写，ThreadSanitizer 只会记录最近 4 次对变量 x 的读写。

ThreadSanitizer 由编译时插桩和运行时库两部分组成。

- 编译时插桩：对于每一次 memory access (load, store)，都会在此次 acccess 之前插入一个函数调用 (`__tsan_read`, `__tsan_write`)，该函数调用是在运行时库中实现的。
- 运行时库：在 `__tsan_read`, `__tsan_write` 中实现 data race 检测的核心逻辑，判断本次访问是否存在 data race。劫持了很多函数实现如 libc 和 pthread，在申请锁、释放锁，fork/join 线程时更新 vector clock 。

例如，本文最开始提到的全局变量数据竞争的代码片段使用 ThreadSanitizer 插桩后的代码变为如下所示：

```
int global;

void Thread1() {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(&global);
  global = 1;
```

(continues on next page)

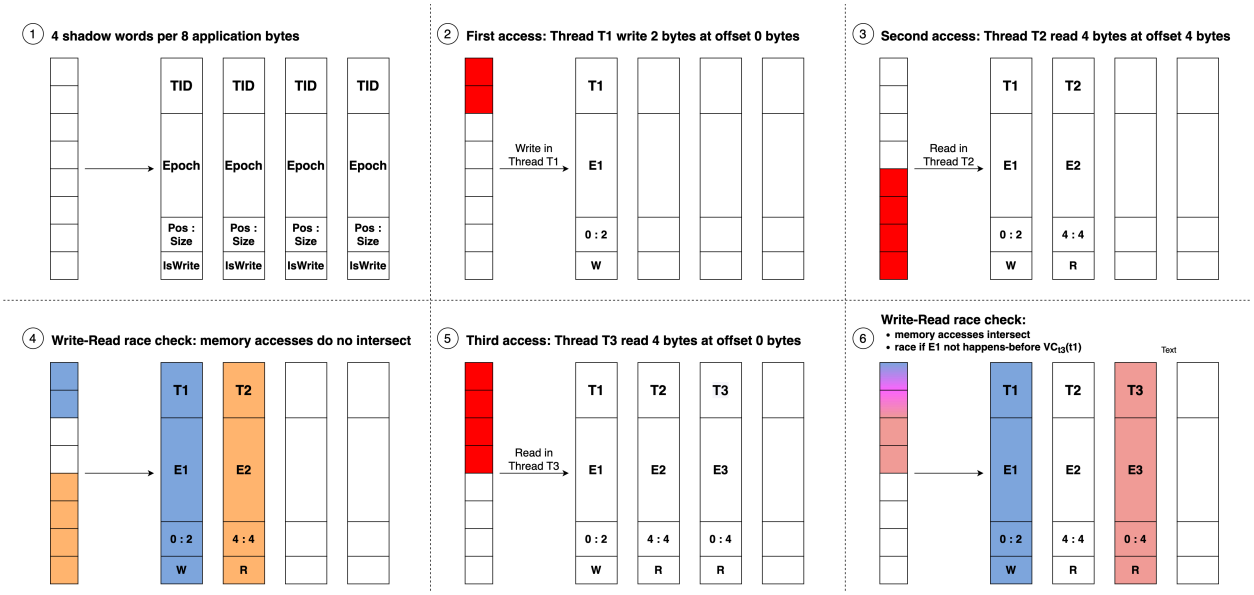
(continued from previous page)

```
  __tsan_func_exit();
}

void Thread2() {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(&global);
  global = 2;
  __tsan_func_exit();
}
```

注意到：在 `global = 1` 和 `global = 2` 之前都插入了对 `__tsan_write4` 的函数调用。

启用 ThreadSanitizer 后，在程序运行过程中，每一个线程都会保存一个 vector clock，每 8-bytes 的应用程序内存都对应有 4 个 8-bytes 的 shadow word。每个 shadow word 都用于记录一次访问操作，记录 TID（线程 id）、Epoch（访存操作发生时线程 TID 此时的 local time）、Pos:Size（标识本次访存访问的是当前 8-bytes 的哪几个 bytes）、IsWrite（标识本次访存操作是读还是写）。

每次读写变量 x 时，由于程序被 ThreadSanitizer 插桩，所以在执行读写操作之前，都会调用函数 `__tsan_read` 或 `__tsan_write`，在 `__tsan_read` 和 `__tsan_write` 的函数实现中，首先找到变量 x 所在的 8-bytes 内存区域。然后找到这 8-bytes 内存所对应的 4 个 shadow word，检查当前这一次对变量 x 的读写与 shadow word 中记录的最近 4 次读写是否存在 data race。最后更新 shadow word 的内容，记录本次对变量 x 的读写，保证 shadow word 记录的是最近 4 次对变量 x 的读写。



完整的 ThreadSanitizer 算法的伪代码如下所示：

```
def HandleMemoryAccess(thread_state, tid, pc, addr, size, is_write, is_atomic):
  shadow_mem = MemToShadow(addr) # the type of shadow_mem is uint64_t*
```

(continues on next page)

(continued from previous page)

```
  IncrementThreadClock(tid)
  LogEvent(tid, pc)
  new_shadow_word = ShadowWord(tid, CurrentClock(tid), addr, size, is_write, is_
↪atomic)
  stored = false
  for i in range(0, 4):
    raced = UpdateOneShadowState(thread_state, shadow_mem, i, new_shadow_word, stored)
    if raced:
      return
  if not stored:
    # Evict a random Shadow Word
    shadow_mem[Random(4)] = store_word # Atomic

def UpdateOneShadowState(shadow_mem, idx, new_shadow_word, stored):
 old_shadow_word = shadow_mem[idx] # Atomic
 # The old state is empty
 if old_shadow_word == 0:
   if not stored:
     StoreIfNotYetStored(shadow_mem[idx], store_word)
     stored = true
   return false
 # Is the memory access equal to the previous?
 if AccessedSameRegion(old_shadow_word, new_shadow_word):
   if SameThreads(old_shadow_word, new_shadow_word): # same thread
     if IsRWWeakerOrEqual(old_shadow_word, new_shadow_word):
       StoreIfNotYetStored(shadow_mem[idx], store_word)
       stored = true
     return false
   if HappensBefore(old_shadow_word, thread_state):
       return false
   if IsBothReadsOrAtomic(old_shadow_word, new_shadow_word):
     return false
   # race!
   ReportRace(old_shadow_word, new_shadow_word)
   return true
 # Do the memory access intersect?
 if AccessedIntersectingRegions(old_shadow_word, new_shadow_word):
   if SameThreads(old_shadow_word, new_shadow_word):
     return false
   if IsBothReadsOrAtomic(old_shadow_word, new_shadow_word):
     return false
   if HappensBefore(old_shadow_word, thread_state):
     return false
```

(continues on next page)

(continued from previous page)

```
   # race!
   ReportRace(old_shadow_word, new_shadow_word)
   return true
 # The accesses do not intersect, do nothing
 return false

def StoreIfNotYetStored(shadow_mem, store_word):
  *shadow_mem = store_word # Atomic
  store_word = 0

def IsRWWeakerOrEqual(old_shadow_word, new_shadow_word):
 return (old_shadow_word.is_atomic > new_shadow_word.is_atomic) ||
        (old_shadow_word.is_atomic == new_shadow_word.is_atomic &&
         !old_shadow_word.is_write >= !new_shadow_word.is_write)

def HappensBefore(old_shadow_word, thread_state):
  return thread_state.vector_clock.get(old_shadow_word.tid) >= old_shadow_word.epoch
```

References

1. Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system.Communications of the ACM,21(7), pp.558-565.
2. Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. (PLDI ’09)
3. Finding races and memory errors with GCC instrumentation. (GNU Tools Cauldron 2012)
4. AddressSanitizer, ThreadSanitizer and MemorySanitizer –Dynamic Testing Tools for C++. (Google Test Automation Conference)
5. https://en.wikipedia.org/wiki/Lamport_timestamp
6. https://en.wikipedia.org/wiki/Vector_clock

7.5.3 DataRace 检测算法之 FastTrack

根据 AddressSanitizer, ThreadSanitizer, and MemorySanitizer: Dynamic Testing Tools for C++ (GTAC’2013)，TSan V2 使用的 fast happens-before 算法，**类似**于 FastTrack(PLDI’09) 中提出的算法。

本文是对 FastTrack: efficient and precise dynamic race detection (PLDI’09) 这篇论文的学习笔记。

Preliminaries

Data Race

在 FastTrack(PLDI'09) 论文中，全文都用的是 race condition，根据 Race Condition vs. Data Race. EMBEDDED IN ACADEMIA 中给出的关于 race condition 和 data race 的定义，该论文中应该实际指代的应该是 data race。本文全都使用“data race”。

首先看 data race 的定义：

- A race condition occurs when a program's execution contains two accesses to the same memory location that are not ordered by the happens-before relation, where at least one of the accesses is a write. ——from FastTrack: efficient and precise dynamic race detection (PLDI'09)
- A data race is a situation when two threads concurrently access a shared memory location and at least one of the accesses is a write. ——from ThreadSanitizer: data race detection in practice (WBIA '09)

即，如果两个线程访问同一个内存位置 (memory location)，至少有一个访问是写操作，并且两个线程访问内存位置的访问顺序是不确定，则说明存在 data race。

举一个非常简单的 data race 的例子：

```
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
  Global=1;
  return NULL;
}

void *Thread2(void *x) {
  Global=2;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
  printf("%d", Global);
}
```

执行上述代码，输出是不确定的：有时候是 1，有时候是 2。

Concepts

一个程序中有很多个线程，每一个线程用 $t \in Tid$ 表示，这些线程读写的变量用 $x \in Var$ 表示，锁用 $m \in Lock$ 表示。

线程 t 能执行的**操作**包括：

- $rd(t,x)$ 和 $wr(t,x)$ 分别表示线程 t 读 x，线程 t 写 x
- $acq(t,m)$ 和 $rel(t,m)$ 分别表示获取锁 m，释放锁 m
- $fork(t,u)$ 表示线程 t fork 一个新的线程 u
- $join(t,u)$ 表示线程 t 阻塞直至线程 u 终止执行

happen-before 的定义（尝试用中文表述总觉得词不达意，摆烂直接用英文原文）：

A trace α captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads.

The happens-before relation $<_\alpha$ for a trace α is the smallest transitively-closed relation over the operations in α such that the relation $a <_\alpha b$ holds whenever a occurs before b in α and one of the following holds:

- Program order: The two operations are performed by the same thread.
- Locking: The two operations acquire or release the same lock.
- Fork-join: One operation is $fork(t,u)$ or $join(t,u)$ and the other operation is by thread u.

如果 a happens before b，那么 b happens after a

如果两个线程访问同一个内存位置 (memory location)，至少有一个访问是写操作，并且这两个访问操作之间没有 happen-before 关系，那么说明这两个访存操作之间存在 data race。

Vector Clock and $DJIT^+$ Algorithm

Vector Clock

假设程序中有 n 个线程，每个线程都对应一个 n 个元素的 vector，称为 vector clock。

Vector clock 之间是存在偏序关系 $\sqsubseteq$ 的。

$\sqcup$ 表示两个 vector clock 之间的交汇 (join) 操作。

$\bot_C$ 表示最小的 vector clock 。

inc_t 表示递增 vector clock 的表示线程 t 的那个元素。

形式化表示如下：

- $C_1 \sqsubseteq C_2$ **iff** $\forall t.\ C_1(t) \leq C_2(t)$
- $C_1 \sqcup C_2 = \lambda t.\ max(C_1(t), C_2(t))$

- $\bot_C \quad = \quad \lambda t.\ 0$
- $inc_t(C) \quad = \quad \lambda u.$ **if** $u = t$ **then** $C(u) + 1$ **else** $C(u)$

举个例子帮忙理解 vector clock 。

假设程序中有 2 个线程 t_1 和 t_2 ， 假设 t_1 的 vector clock C_1 为 $< 4, 0 >$， t_2 的 vector clock C_2 为 $< 5, 8 >$ 。

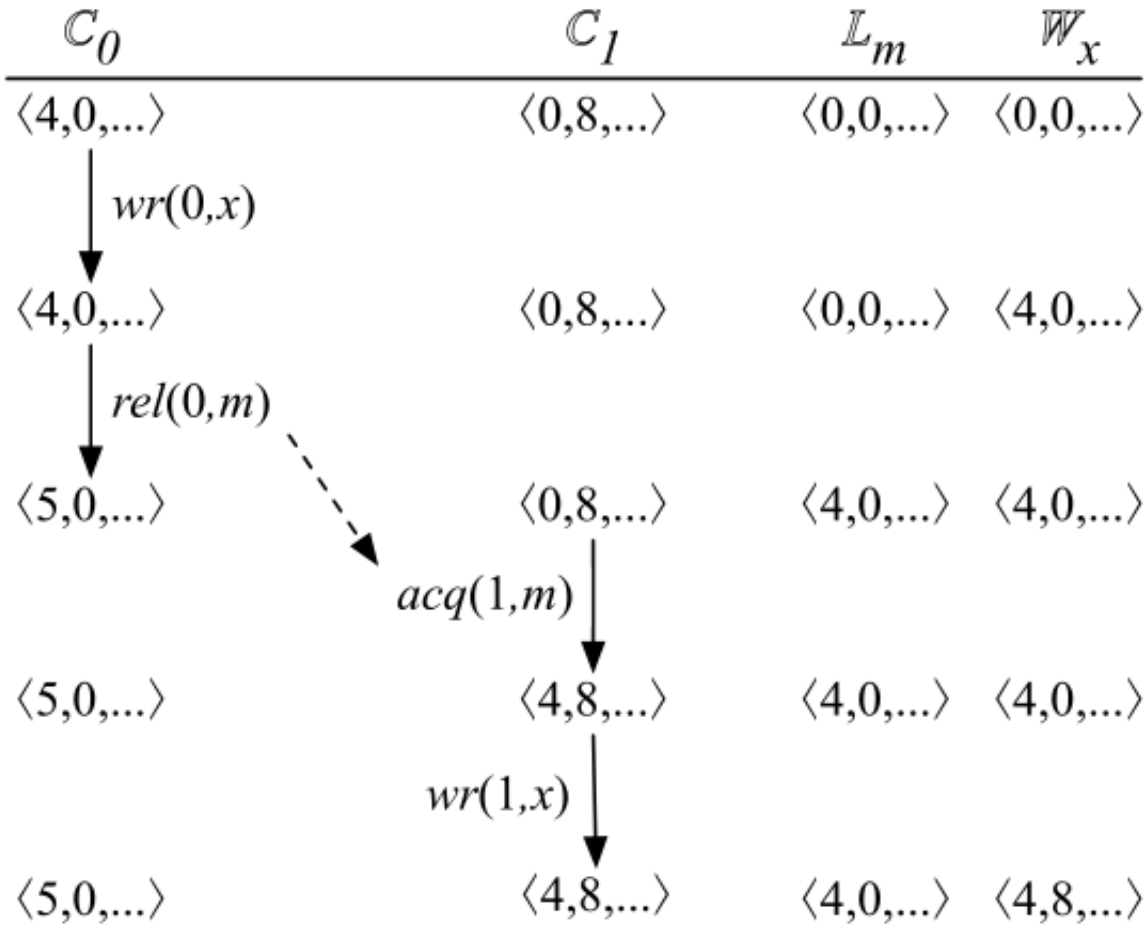
- 因为 $4 \leq 5$ ， $0 \leq 8$ 所以 $C_1 \sqsubseteq C_2$
- $C_1 \sqcup C_2 \ = < max(4, 5), max(0, 8) > = < 5, 8 >$
- 因为此例中只有 2 个线程所以 $\bot_C = < 0, 0 >$
- $inc_{t1}(C_1) = < 4 + 1, 0 > = < 5, 0 >$， $inc_{t2}(C_1) \ = < 4, 0 + 1 > = < 4, 1 >$， $inc_{t1}(C_2) \ = < 5 + 1, 8 > = < 6, 8 >$， $inc_{t2}(C_2) \ = < 5, 8 + 1 > = < 5, 9 >$

$DJIT^+$ Algorithm

$DJIT^+$ Algorithm 就是基于 vector clock 来检测 data race 的：

- 每个线程 t 都对应一个 vector clock $\mathbb{C}_t$，对于任意一个线程 u，$\mathbb{C}_t(u)$ 记录了与线程 t 的当前操作满足 happen-before 关系的线程 u 的上一次操作的 clock（如果把线程 u 的上一次操作记为 O_u，把线程 t 的当前操作记为 O_t，那么有 O_u happen-before O_t）
- 每一个锁 m 也对应一个 vector clock $\mathbb{L}_m$
- 每一个变量 x 对应两个 vector clock $\mathbb{W}_x$ 和 $\mathbb{R}_x$。对于任意一个线程 t ， $\mathbb{W}_x$ 和 $\mathbb{R}_x$ 记录了线程 t 对变量 x 的最后一次读/写的 clock
  - 线程 t 对变量 x 的读时，会将 $\mathbb{R}_x[t]$ 的值更新为 $\mathbb{C}_t(t)$ 的值
  - 程序 t 对变量 x 的写时，会将 $\mathbb{W}_x[t]$ 的值更新为 $\mathbb{C}_t(t)$ 的值
- 程序中执行同步和线程操作时，算法会更新相应的 vector clock：
  - 线程 u 释放了锁 m，$DJIT^+$ 会先将 $\mathbb{L}_m$ 的值更新为 $\mathbb{C}_u$ 的值，再将 $\mathbb{C}_u$ 的值更新为 $inc_u(\mathbb{C}_u)$
  - 线程 t 获取了锁 m，$DJIT^+$ 会将 $\mathbb{C}_t$ 的值更新为 $\mathbb{C}_t \sqcup \mathbb{L}_m$ 的值。
  - $fork(t, u)$，$DJIT^+$ 会先将 $\mathbb{C}_u$ 的值更新为 $\mathbb{C}_u \sqcup \mathbb{C}_t$，再将 $\mathbb{C}_t$ 的值更新为 $inc_t(\mathbb{C}_t)$
  - $join(t, u)$，$DJIT^+$ 会先将 $\mathbb{C}_t$ 的值更新为 $\mathbb{C}_t \sqcup \mathbb{C}_u$，再将 $\mathbb{C}_u$ 的值更新为 $inc_u(\mathbb{C}_u)$
- 如何判断是否存在 data race：
  - 假设当前线程 u 读变量 x ， 如果 $\mathbb{W}_x \sqsubseteq \mathbb{C}_u$ 那么当前线程 u 对变量 x 的读则与之前其他线程对变量 x 的写不存在 data race
  - 假设当前线程 u 写变量 x ， 如果 $\mathbb{W}_x \sqsubseteq \mathbb{C}_u$ 且 $\mathbb{R}_x \sqsubseteq \mathbb{C}_u$ 那么当前线程 u 对变量 x 的写则与之前其他线程对变量 x 的写和读不存在 data race

我们用如下例子来理解 $DJIT^+$ 是如何检测 data race 的：



如上图所示，程序中有两个线程，线程 0 和线程 1。线程 0 对应的 vector clock 为 $\mathbb{C}_0$，线程 1 对应的 vector clock 为 $\mathbb{C}_1$，锁 m 对应的 vector clock 为 $\mathbb{L}_m$，并且我们用 vector clock $\mathbb{W}_x$ 来记录前一次对变量 x 的写操作。

1. 初始状态时，$\mathbb{C}_0$ 为 <4, 0>，$\mathbb{C}_1$ 为 <0, 8>，$\mathbb{L}_m$ 为 <0, 0>（即 $\perp_C$），$\mathbb{W}_x$ 为 <0, 0>（即 $\perp_C$）
2. 线程 0 写变量 x，vector clock $\mathbb{W}_x[0]$ 的值更新为 $\mathbb{C}_0[0]$ 的值，所以 $\mathbb{W}_x$ 的值由 <0, 0> 更新为 <4, 0>，其余 vector clock 的值不变
3. 线程 0 释放锁 m，vector clock $\mathbb{L}_m$ 的值更新为 $\mathbb{C}_0$ 的值 <4, 0>，然后 vector clock $\mathbb{C}_0$ 的值更新为 $inc_0(\mathbb{C}_0)$ 即 <5, 0>，其余 vector clock 的值不变
4. 线程 1 获取锁 m，vector clock $\mathbb{C}_1$ 的值更新为 $\mathbb{C}_1 \sqcup \mathbb{L}_m$ 即 <0, 8> $\sqcup$ <4, 0> = <4, 8>，其余 vector clock 的值不变
5. 线程 1 写变量 x，由于 $\mathbb{C}_1$ 为 <4, 8>，$\mathbb{W}_x$ 为 <4, 0>，所以 $\mathbb{W}_x \sqsubseteq \mathbb{C}_1$，也就是说 $\precapprox_\smallsetminus(\nVdash, \curvearrowleft)$ happen-before $\precapprox_\smallsetminus(\nVdash, \curvearrowleft)$，所以线程 1 写变量 x 与线程 0 写变量 x 之间没有 data race。最后还要更新 $\mathbb{W}_x[1]$ 为 $\mathbb{C}_1[1]$，即 $\mathbb{W}_x$ 的值由 <4, 0> 变为 <4, 8>，其余 vector clock 的值不变

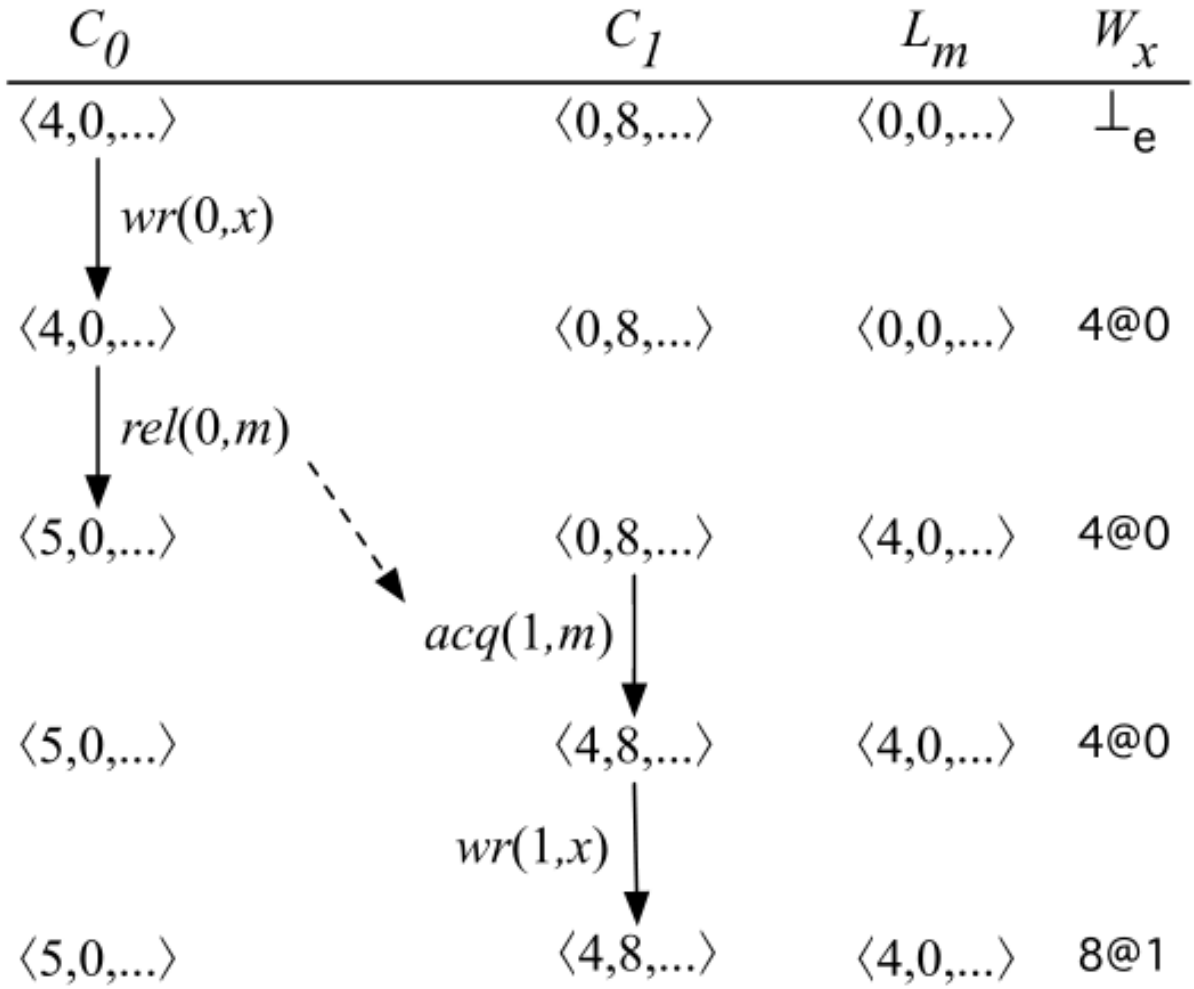
FastTrack Algorithm

上述基于 vector clock 的 $DJIT^+$ Algorithm 的缺点就是性能开销。如果程序中有 n 个线程，那么每一个 vector clock 都需要 $O(n)$ 的空间，并且对 vector clock 的操作 (copying, comparing, joining, etc) 都要花费 $O(n)$ 的时间。

key observation：作者通过收集大量的 Java 程序信息发现：在所有需要被 data race detector 监测的操作（rd, wr, acq, rel, $fork$, $join$, $\cdots$）中，同步 (synchronization) 操作 (lock acquires and releases, forks, joins, waits, notifies, etc) 出现的次数只占很少的比例；而对数组和对象字段的读写则占了被监测操作的 96% 。

key insight：超过 99% 的读写操作，我们并不需要使用 vector clock 来表示其 happen-before 关系，只需使用一种更轻量级 happen-before 表示方式即可，只在必要时 fallback 为 vector clock 。

还是用上面这个例子来进行说明，这次我们不再使用 vector clock 来记录每个线程对变量 x 的写操作，只记录上一次是哪个线程写了变量 x：



我们把 clock c 和 thread t 组成的对组 (pair) 叫作 $epoch$，记作 $c@t$。epoch 与 vector clock 存在如下关系：$c@t \preceq C$ **iff** $c \leq C(t)$ 。

在上面这个例子中，对于 $\mathbb{W}_x$ 我们只需要使用 $epoch$ 记录上一次是哪个线程写了变量 x：

1. 初始状态时，$\mathbb{C}_0$ 为 <4, 0>，$\mathbb{C}_1$ 为 <0, 8>，$\mathbb{L}_m$ 为 <0, 0>（即 $\perp_C$），$\mathbb{W}_x$ 为 $\perp_e$
2. 线程 0 写变量 x，将 epoch $\mathbb{W}_x$ 的值更新为 $4@0$，表示线程 0 在 clock 为 4 时写了变量 x
3. 线程 0 释放锁 m，vector clock $\mathbb{L}_m$ 的值更新为 $\mathbb{C}_0$ 的值 <4, 0>，然后 vector clock $\mathbb{C}_0$ 的值更新为 $inc_0(\mathbb{C}_0)$ 即 <5, 0>，其余 vector clock 的值不变
4. 线程 1 获取锁 m，vector clock $\mathbb{C}_1$ 的值更新为 $\mathbb{C}_1 \sqcup \mathbb{L}_m$ 即 <0, 8> $\sqcup$ <4, 0> = <4, 8>，其余 vector clock 的值不变
5. 线程 1 写变量 x，由于 $\mathbb{C}_1$ 为 <4, 8>，$\mathbb{W}_x$ 为 $4@0$，所以有 $\mathbb{W}_x = 4@0 \preceq < 4, 8 > = \mathbb{C}_1$，也就是说 $\precsim\!\!\!\!\diagdown(\nVdash, \curvearrowright)$ happen-before $\precsim\!\!\!\!\diagdown(\Vdash, \curvearrowright)$，所以线程 1 写变量 x 与线程 0 写变量 x 之间没有 data race。最后还要更新 $\mathbb{W}_x$ 为 $8@1$，表示线程 1 在 clock 为 8 时写了变量 x，其余 vector clock 的值不变

在这个例子中，我们用 epoch 来代替 vector clock $\mathbb{W}_x$ 后，在判断 $wr(1, x)$ 是否与 $wr(0, x)$ 之间存在 data race 时，将 $O(n)$ 的 vector clock 之间的比较操作 $\sqsubseteq$ 替换优化为了 $O(1)$ 的 epoch 与 vector clock 之间的比较操作 $\preceq$。

下面我们系统地学习一下 FastTrack Algorithm 是如何检测 data race 的。

Type of Data Race

data race 可以根据观测到的读写操作的先后顺序可以分为以下三类：

- **write-write** data race。

  对于 write-write 这种 data race，我们没有必要使用完整的 vector clock 记录所有线程对变量 x 的写操作，只记录上一次是哪个线程写了变量 x 就足够了。假设程序执行至某一时刻，对变量 x 的读写操作还没有出现过 data race，那么所有对变量 x 的写操作都是按照 happen-before 关系排序好的，所以为了检测后续对变量 x 的写与之前对变量 x 的写之间是否存在 write-write data race，我们需要记录的关键信息就是最近的一次对变量 x 的写是在哪一 clock 由哪一 thread 完成的。如前所述，我们把 clock c 和 thread t 组成的对组 (pair) 叫作 $epoch$，记作 $c@t$。epoch 与 vector clock 存在如下关系：$c@t \preceq C$ **iff** $c \leq C(t)$ 。所以检测是否存在 write-write data race，我们只需花费 $O(1)$ 的时间开销比较 epoch 与 vector clock 之间是否满足 $\preceq$ 关系即可。

- **write-read** data race

  一旦我们用 epoch 来记录上一次对变量 x 的写，检测是否存在 write-read data race 也变得非常简单。假设线程 t 读变量 x 时的 vector clock 是 $\mathbb{C}_t$，我们只需花费 $O(1)$ 的时间开销比较 $\mathbb{W}_x \preceq \mathbb{C}_t$ 是否满足，就可以判断上一次对变量 x 的写操作是否 happen-before 这一次对变量 x 的读操作，以检测是否存在 write-read data race 。

- **read-write** data race

  与 write-write data race 和 write-read data race 的检测相比，read-write data race 的检测则相对复杂一些。因为在没有 data race 的程序中，对变量 x 的多个读操作也可能是 concurrent 的。假设程序执行至某一个时刻，对变量 x 的两个读操作 $rd(0, x)$ 和 $rd(1, x)$ 是 concurrent 的，即有可能 $rd(0, x)$ 先执行，也有可能 $rd(1, x)$ 先执行，那么就算我们知道 $wr(\_, x)$ happen-before $rd(0, x)$ ，我们也无法判断 $wr(\_, x)$ happen-before $rd(1, x)$ 是否成立。因此，我们需要使用完整的 vector clock R_x 来记录对变量 x 的读操作。

FastTrack 使用一种自适应的 (adaptive) 方式来记录对变量 x 的读：如果当前对变量 x 的读操作 happen-after 所有之前对变量 x 的读操作，那么我们只需要使用 epoch 来记录这一次最新的对变量 x 的读即可。如果对变量 x 的读操作之间是 concurrent 的，我们转而使用 vector clock 来记录对变量 x 的读操作。

Analysis Detail

FastTrack 是一个 online algorithm，对于被测程序，FastTrack 会维护一个程序状态 σ，每当程序执行一个操作 a 时，FastTrack 会相应地更新状态：$\sigma \Rightarrow^a \sigma'$ 。

$\sigma = (C, L, R, W)$ 是一个四元组：

- C_t 表示当前线程 t 的 vector clock
- L_m 表示上一次释放锁 m 对应的 vector clock
- R_x 表示上一次对变量 x 读对应的 epoch 或 vector clock
- W_x 表示上一次对变量 x 写对应的 epoch

状态 σ 的初始值为：

$\sigma_0 = (\lambda t.inc_t(\bot_V), \lambda m.\bot_V, \lambda x.\bot_e, \lambda x.\bot_e)$

$E(t)$ 表示线程 t 的当前 epoch $c@t$，其中 $c = C_t(t)$ 即线程 t 的当前 clock 。

R 是一个函数，R_x 是 $R(x)$ 的缩写，$R[x := V]$ 表示将 $R(x)$ 修改为 V，其余部分不变。

下图详细给出了针对不同的操作，FastTrack 是如何更新程序状态 σ（在每种操作的右边同时给出了作者在 benchmarks 中观察到的不同操作的出现占比）：

Reads: 82.3% of all Operations

[FT READ SAME EPOCH] 63.4% of reads

$$\frac{R_x = E(t)}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R, W)}$$

[FT READ SHARED] 20.8% of reads

$$\frac{R_x \in VC \quad W_x \preceq C_t \quad R' = R[x := R_x[t := C_t(t)]]}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

[FT READ EXCLUSIVE] 15.7% of reads

$$\frac{R_x \in Epoch \quad R_x \preceq C_t \quad W_x \preceq C_t \quad R' = R[x := E(t)]}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

[FT READ SHARE] 0.1% of reads

$$\frac{R_x = c@u \quad W_x \preceq C_t \quad V = \bot_V[t := C_t(t), u := c] \quad R' = R[x := V]}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

Writes: 14.5% of all Operations

[FT WRITE SAME EPOCH] 71.0% of writes

$$\frac{W_x = E(t)}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W)}$$

[FT WRITE EXCLUSIVE] 28.9% of writes

$$\frac{R_x \in Epoch \quad R_x \preceq C_t \quad W_x \preceq C_t \quad W' = W[x := E(t)]}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W')}$$

[FT WRITE SHARED] 0.1% of writes

$$\frac{R_x \in VC \quad R_x \sqsubseteq C_t \quad W_x \preceq C_t \quad W' = W[x := E(t)] \quad R' = R[x := \bot_e]}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R', W')}$$

Other: 3.3% of all Operations

[FT ACQUIRE]

$$\frac{C' = C[t := (C_t \sqcup L_m)]}{(C, L, R, W) \Rightarrow^{acq(t,m)} (C', L, R, W)}$$

[FT RELEASE]

$$\frac{L' = L[m := C_t] \quad C' = C[t := inc_t(C_t)]}{(C, L, R, W) \Rightarrow^{rel(t,m)} (C', L', R, W)}$$

[FT FORK]

$$\frac{C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, R, W) \Rightarrow^{fork(t,u)} (C', L, R, W)}$$

[FT JOIN]

$$\frac{C' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, L, R, W) \Rightarrow^{join(t,u)} (C', L, R, W)}$$

Read Operations

Read Operations 又细分为 4 条规则：

- **[FT READ SAME EPOCH]**

  此时程序执行的操作是 $rd(t, x)$，即线程 t 读变量 x。如果 $R_x = E(t)$，即前一次对变量 x 读与这一次对变量 x 读，是同一个线程在同一 clock 时刻对变量 x 读，那么不用更新程序状态 σ

- **[FT READ SHARED]**

  此时程序执行的操作是 $rd(t, x)$，即线程 t 读变量 x。如果此时 R_x 已经是用 vector clock 表示的 ($R_x \in VC$)，并且前一次对变量 x 的写 happen-before 此时线程 t 对变量 x 的读 ($W_x \preceq C_t$)，那么我们只需要把

vector clock R_x 中线程 t 的那部分更新为 $C_t(t)$ 即可，形式化表示 $R' = R[x := R_x(t := C_t(t))]$

- **[FT READ EXCLUSIVE]**

  此时程序执行的操作是 $rd(t, x)$，即线程 t 读变量 x。如果此时 R_x 是用 epoch 表示的 ($R_x \in Epoch$)，并且前一次对变量 x 的读 happen-before 此时线程 t 对变量 x 的读 ($\$R_x \preceq C_t$)，前一次对变量 x 的写 happen-before 此时线程 t 对变量 x 的读 ($W_x \preceq C_t$)，那么我们只需更新 epoch R_x 为 $E(t)$ 即可。$E(t)$ 表示线程 t 的当前 epoch $c@t$，其中 $c = C_t(t)$ 即线程 t 的当前 clock

- **[FT READ SHARE]**

  此时程序执行的操作是 $rd(t, x)$，即线程 t 读变量 x。如果此时 R_x 是用 epoch 表示的 ($R_x = c@u$)，并且前一次对变量 x 的写 happen-before 此时线程 t 对变量 x 的读 ($W_x \preceq C_t$)，但是前一次对变量 x 的读与此时线程 t 对变量 x 的读没有 happen-before 关系 ($\$R_x \preceq C_t$)，那么我们需把 epoch R_x 转换为 vector clock R_x，线程 u 的 clock 是 c，线程 t 的 clock 是 $C_t(t)$。形式化表示 $V = \bot_V[t := C_t(t), u := c], R' = R[x := V]$

Write Operations

- **[FT WRITE SAME EPOCH]**

  此时程序执行的操作是 $wr(t, x)$，即线程 t 写变量 x。如果 $W_x = E(t)$，即前一次对变量 x 写与这一次对变量 x 写，是同一个线程在同一 clock 时刻对变量 x 写，那么不用更新程序状态 σ。

- **[FT WRITE EXCLUSIVE]**

  此时程序执行的操作是 $wr(t, x)$，即线程 t 写变量 x。如果此时 R_x 是用 epoch 表示的 ($R_x \in Epoch$)，并且前一次对变量 x 的读 happen-before 此时线程 t 对变量 x 的写 ($R_x \preceq C_t$)，前一次对变量 x 的写 happen-before 此时线程 t 对变量 x 的写 ($W_x \preceq C_t$)，那么我们只需要把 epoch W_x 中更新为 $E_t(t)$ 即可，形式化表示 $W' = W[x := E(t)]$

- **[FT WRITE SHARED]**

  此时程序执行的操作是 $wr(t, x)$，即线程 t 写变量 x。如果此时 R_x 已经是用 vector clock 表示的 ($R_x \in VC$)，并且前面所有对变量 x 的读 happen-before 此时线程 t 对变量 x 的写 ($R_x \sqsubseteq C_t$)，前一次对变量 x 的写 happen-before 此时线程 t 对变量 x 的写 ($W_x \preceq C_t$)，那么我们把 epoch W_x 中更新为 $E(t)$ ，并且把 R_x 更新为 $\bot_e$ 。

  将 R_x 更新为 $\bot_e$ 是因为后续对变量 x 的写操作不可能与此时 R_x 中记录的对变量 x 的写操作有 data race 了，所以我们无需再记录之前对变量 x 的写了。

  形式化表示 $W' = W[x := E(t)], R' = R[x := \bot_e]$

Other Operations

Other operations 包括 acquire, release, fork 和 join，FastTrack algorithm 对这些操作的处理与 $DJIT^+$ algorithm 类似：

- **[FT ACQUIRE]**

  此时程序执行的操作是 $acq(t, m)$，线程 t 获取了锁 m 。将 C_t 的值更新为 $C_t \sqcup L_m$ 的值

- **[FT RELEASE]**

  此时程序执行的操作是 $rel(t, m)$，线程 t 释放了锁 m 。将 L_m 的值更新为 C_t 的值，再将 C_t 的值更新为 $inc_t(C_t)$

- **[FT FORK]**

  此时程序执行的操作是 $fork(t, u)$ 。先将 C_u 的值更新为 $C_u \sqcup C_t$，再将 C_t 的值更新为 $inc_t(C_t)$

- **[FT JOIN]**

  此时程序执行的操作是 $join(t, u)$ 。先将 C_t 的值更新为 $C_t \sqcup C_u$，再将 C_u 的值更新为 $inc_u(C_u)$

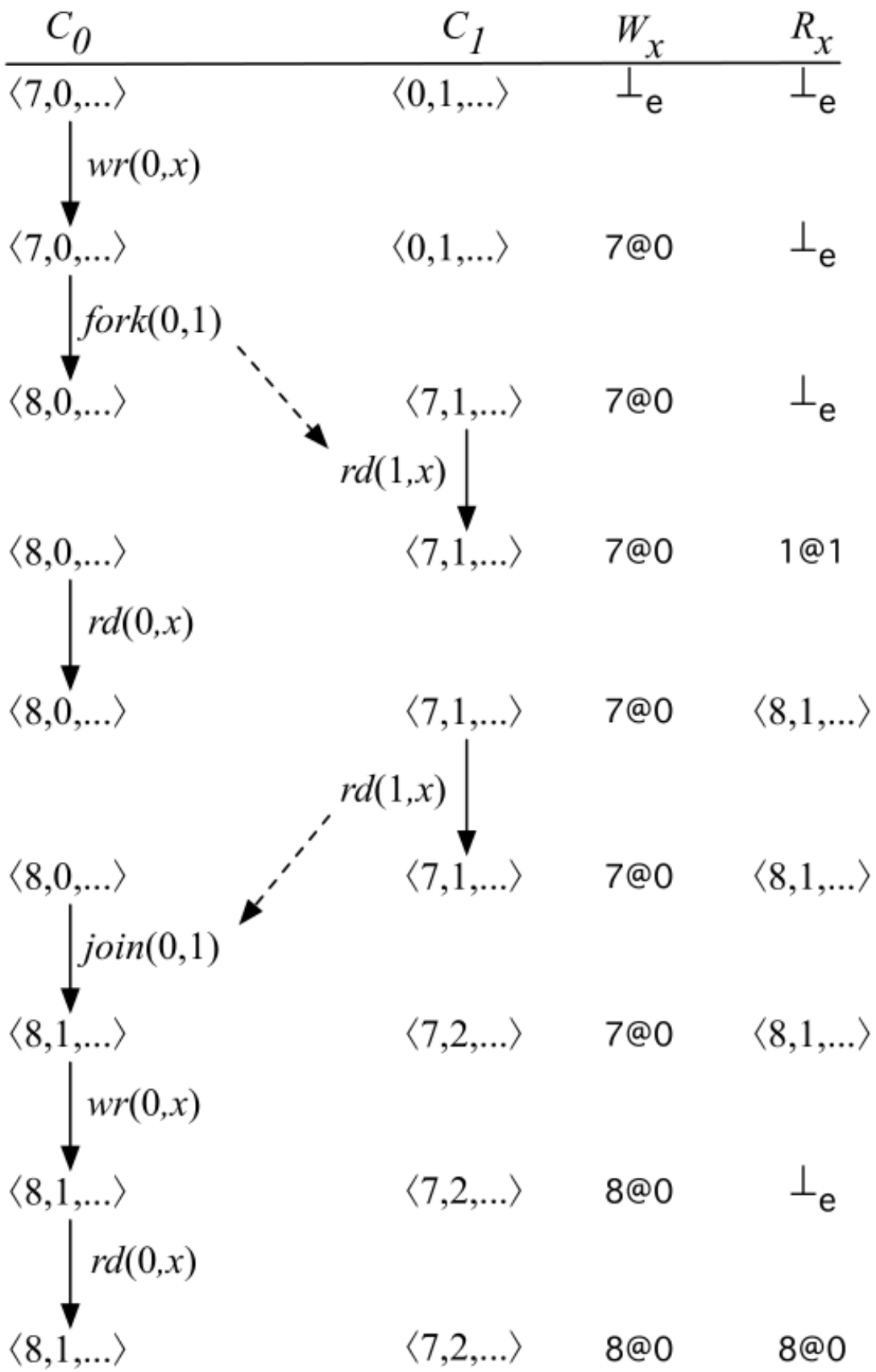
Algorithm Pseudo Code

FastTrack Algorithm 的伪代码实现如下，就是对上一节 Analysis Detail 的实现，此处不再赘述：

```
class ThreadState {
  int tid;
  int C[];
  int epoch;  // invariant: epoch == C[tid]
}

class VarState {
  int W, R;
  int Rvc[]; // used iff R == READ_SHARED
}

class LockState {
  int L[];
}

void read(VarState x, ThreadState t)
  if (x.R == t.epoch) return;                 // Same Epoch 63.4%

  // write-read race?
  if (x.W > t.C[TID(x.W)]) error;

  // update read state
  if (x.R == READ_SHARED) {                   // Shared     20.8%
    x.Rvc[t.tid] = t.epoch;
  } else {
    if (x.R <= t.C[TID(x.R)]) {               // Exclusive  15.7%
      x.R = t.epoch;
    } else {                                  // Share       0.1%
      if (x.Rvc == null)
        x.Rvc = newClockVector();                  // (SLOW PATH)
      x.Rvc[TID(x.R)] = x.R;
      x.Rvc[t.tid] = t.epoch;
      x.R = READ_SHARED;
    }
  }
}

void write(VarState x, ThreadState t)
  if (x.W == t.epoch) return;                 // Same Epoch 71.0%

  // write-write race?
  if (x.W > t.C[TID(x.W)]) error;

  // read-write race?
  if (x.R != READ_SHARED) {                   // Shared     28.9%
    if (x.R > t.C[TID(x.R)]) error;
  } else {                                    // Exclusive   0.1%
    if (x.Rvc[u] > t.C[u] for any u) error;       // (SLOW PATH)
  }
  x.W = t.epoch; // update write state
}
```

Example

最后，我们再用一个例子来理解 FastTrack 是如何检测 data race 的：


C_0
C_1
W_x
R_x
⟨7,0,...⟩
⟨0,1,...⟩
$\perp_e$
$\perp_e$
$wr(0,x)$
⟨7,0,...⟩
⟨0,1,...⟩
7@0
$\perp_e$
$fork(0,1)$
⟨8,0,...⟩
⟨7,1,...⟩
7@0
$\perp_e$
$rd(1,x)$
⟨8,0,...⟩
⟨7,1,...⟩
7@0
1@1
$rd(0,x)$
⟨8,0,...⟩
⟨7,1,...⟩
7@0
⟨8,1,...⟩
$rd(1,x)$
⟨8,0,...⟩
⟨7,1,...⟩
7@0
⟨8,1,...⟩
$join(0,1)$
⟨8,1,...⟩
⟨7,2,...⟩
7@0
⟨8,1,...⟩
$wr(0,x)$
⟨8,1,...⟩
⟨7,2,...⟩
8@0
$\perp_e$
$rd(0,x)$
⟨8,1,...⟩
⟨7,2,...⟩
8@0
8@0

初始状态时，W_x 和 R_x 都是 $\perp_e$ ，表示变量 x 还没有被写和读过。

1. 线程 0 写变量 x，epoch W_x 的值更新为线程 0 的当前 epoch $c@t$，即 7@0
2. $fork(0,1)$ 。先将 C_1 的值更新为 $C_1 \sqcup C_0$ 即 <7,1>，再将 C_0 的值更新为 $inc_0(C_0)$ 即 <8, 0>
3. 线程 1 读变量 x，因为前一次对变量 x 的写 happen-before 此时线程 1 对变量 x 的读 ($W_x = 7@0 \preceq < 7, 1 > = C_1$)，所有没有 write-read data race。因为之前 R_x 是 $\perp_e$，所以只需要用 epoch 来表示 R_x 即可，将 R_x 的值更新为线程 1 的当前 epoch $c@t$，即 1@1
4. 线程 0 读变量 x，因为前一次对变量 x 的写 happen-before 此时线程 0 对变量 x 的读 ($W_x = 7@0 \preceq < 8, 0 > = C_0$)，所有没有 write-read data race。但是前一次对变量 x 的读与此时线程 0 对变量 x 的读没有 happen-before 关系 ($R_x = 1@1 \not\preceq < 8, 0 > = C_0$) 。所以需要用 vector clock 替代 epoch 来表示 R_x ，将 R_x 的值更新为 <8, 1>
5. 线程 1 读变量 x，因为前一次对变量 x 的写 happen-before 此时线程 1 对变量 x 的读 ($W_x = 7@0 \preceq < 7, 1 > = C_1$)，所以没有 write-read data race。因为 R_x 已经是是 vector clock 表示了，所以只需要把 vector clock R_x 中线程 1 的那部分更新为 $C_1(1)$ 即可。因为 $R_x(1)$ 是 1，$C_1(1)$ 也是 1，vector clock R_x 更新前后都是 <8, 1>
6. $join(0,1)$，先将 C_0 的值更新为 $C_0 \sqcup C_1$ 即 <8,1>，再将 C_1 的值更新为 $inc_1(C_1)$ 即 <7, 2>
7. 线程 0 写变量 x，此时 R_x 已经是用 vector clock 表示的，并且前面所有对变量 x 的读 happen-before 此时线程 0 对变量 x 的写，没有 read-write data race ($R_x = < 8, 1 > \sqsubseteq < 8, 1 > = C_0$)，前一次对变量 x 的写 happen-before 此时线程 t 对变量 x 的写，没有 write-write data race ($W_x = 7@0 \preceq < 8, 1 >= C_0$)，我们把 W_x 中更新为线程 0 的当前 epoch 即 8@0 ，并且把 R_x 更新为 $\perp_e$ 。
8. 线程 0 读变量 x，因为前一次对变量 x 的写 happen-before 此时线程 0 对变量 x 的读 ($W_x = 8@0 \preceq < 8, 1 > = C_0$)，所以没有 write-read data race。因为之前 R_x 是 $\perp_e$，所以只需要用 epoch 来表示 R_x 即可，将 R_x 的值更新为线程 0 的当前 epoch $c@t$，即 8@0

最终 FastTrack 发现程序中没有 data race 。

Conclusions

本文是对知名的 data race 检测算法 FastTrack 的学习笔记。

FastTrack 算法是对 $DJIT^+$ 算法的一个优化，本质上都是基于 vector clock 检测 data tace 的，FastTrack 通过（在某些情况下）使用 epoch 代替 vector clock 来获得更好的空间复杂度和时间复杂度。

最后说一下，由于手工验证 data race detector 检测出来的 data race 是不是误报是非常困难的，所以对于 data race 的检测，我们希望 data race detector 报告出来的 data race 都是真的，而不是误报。在实践中，我们通常会用动态分析来做 data race 的检测，保证没有误报，比如 TSan 就是 data race 的动态分析工具。

理解了 FastTrack 算法后，再去阅读 TSan 源码，理解 TSan 背后的算法就会变得容易些。下一篇文章就是 ThreadSanitizer 底层算法的基本原理。

7.5.4 A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

本文是对 A Tree Clock Data Structure for Causal Orderings in Concurrent Executions (ASPLOS'22) 这篇论文的学习笔记。

个人感觉这篇文章非常精妙，提出了 tree clock 这种数据结构来替代目前 Data Race 检测算法中广泛使用的数据结构 vector clock，极大的改善了计算 happen-before 关系时的时间复杂度。

Contribution

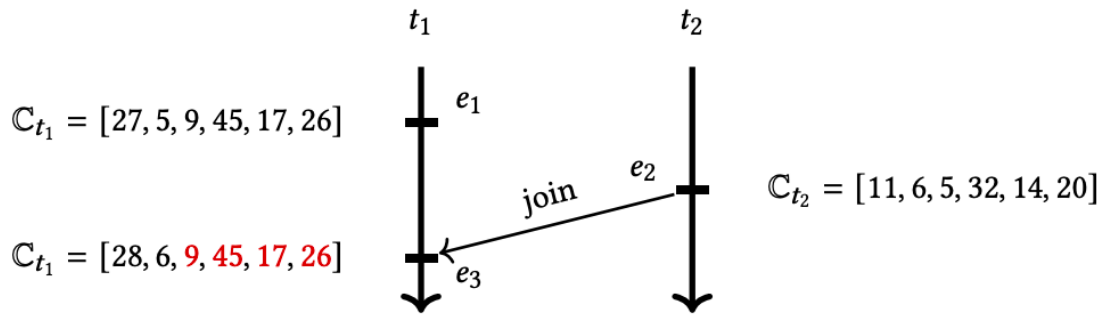
首先来看 tree clock 解决的是什么问题：

假设程序中有 k 个线程，那么 vector clock 的大小就是 k，那么 vector clock 在做 join 和 copy 操作时，就是 $\Theta(k)$ 的时间复杂度。因此，当 k 很大时，join 和 copy 就是 bottleneck。而 tree clock 可以优化该 linear 的时间复杂度为 sublinear。

Key Insight

vector clock 最基本的操作就是 join 操作，即 $\mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2} = \lambda t.\ max(\mathbb{C}_{t_1}(t), \mathbb{C}_{t_2}(t))$

考虑如下例子：



- 当前程序中一共有 6 个线程，vector clock 的大小为 6。线程 t_2 和线程 t_1 于事件 e_2 和事件 e_3 进行同步 (synchronization)。可以理解为事件 e_2 释放了锁 m，然后事件 e_3 获取了锁 m，通过锁来实现同步。所以更新 $\mathbb{C}_{t_1}$ 为 $\mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2}$，即 $\mathbb{C}_{t_1} = \mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2} = [27, 5, 9, 45, 17, 26] \sqcup [11, 6, 5, 32, 14, 20]$
  $= [max(27, 11), max(5, 6), max(9, 5), max(45, 32), max(17, 14), max(26, 20)]$
  $= [28, 6, 9, 45, 17, 26]$
- 现在我们**假设**线程 t_2 的 vector clock $\mathbb{C}_{t_2}$ 中 t_3, t_4, t_5, t_6 的值都是 $\mathbb{C}_{t_2}$ 之前和 $\mathbb{C}_{t_3}$ 做 join 操作时从 t_3 处获取到的。注意到 $\mathbb{C}_{t_1}$ 中 t_3 的值（即 $\mathbb{C}_{t_1}(t_3) = 9$）比 $\mathbb{C}_{t_2}$ 中 t_3 的值（即 $\mathbb{C}_{t_2} = 5$）**大**，这意味着**之前**在某一时刻线程 t_2 与 t_3 之间先进行同步，t_2 从 t_3 获取到了 t_3, t_4, t_5, t_6 的值，**然后**某一时刻线程 t_1 与 t_3 之间再进行同步。也就是说，在 e_2 这个时候线程 t_2 的 $\mathbb{C}_{t_2}$ 中 t_3, t_4, t_5, t_6 的值**一定不会**比 e_3 时线程 t_1 的 $\mathbb{C}_{t_1}$ 中持有的 t_3, t_4, t_5, t_6 的值**更**新。所以在更新 $\mathbb{C}_{t_1}$ 为 $\mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2}$ 时，计算 t_3, t_4, t_5, t_6 的值是**冗余操作**!

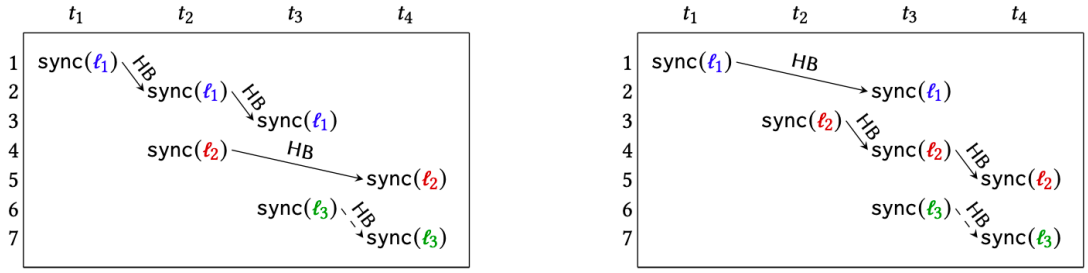
- 由于 vector clock 中没有保存“线程 t_2 的 vector clock $\mathbb{C}_{t_2}$ 中 t_3, t_4, t_5, t_6 的值都是 $\mathbb{C}_{t_2}$ 之前和 $\mathbb{C}_{t_3}$ 做 join 操作时从 t_3 处获取到的”这个信息，所以不能避免 $\mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2}$ 时分别计算 t_3, t_4, t_5, t_6 的最大值这种冗余操作。而 tree clock 就是通过将这一信息编码在一种树数据结构中来避免这种冗余计算的。

Tree Clock Data Structure

Tree clock 每一个节点对应 vector clock 中的一个 entry，每个节点 u 都由三个元素组成 (tid, clk, aclk)，每个节点的子节点 Chld(u) 都是按照 aclk 的大小降序排序的。

我们举例来说明节点的每一个元素所代表的意义：

$sync(l)$ 表示同步操作，如图 2(a) 中线程 t_1 和线程 t_2 于事件 e_1 和 e_2 处进行同步，即线程 t_1 在 e_1 释放了锁 l_1，线程 t_2 在 e_2 获取了锁 l_1，事件 e_1 和事件 e_2 满足 happen-before 关系。



(a) Direct monotonicity. (b) Indirect monotonicity.

Figure 2: Illustration of the two insights behind tree clocks. An event sync(ℓ) represents two events acq(ℓ), rel(ℓ).

Figure 3: The tree clock of t_4 after processing the event e_7 in the traces of Figure 2a (left) and Figure 2b (right).

图 3 左边的树对应图 2(a) 中线程 t_4 在事件 e_7 后的 tree clock：

- 因为是 t_4 的 tree clock，所以根节点 $(t_4, 2, \perp)$ 的 tid 就是 t_4 ，根节点的 clk 值是 2，这是因为在事件 e_7 后线程 t_4 的 local time 就是 2，根节点的 aclk 的值一定是 $\perp$
- 通过树的结构可以看出来，此时 t_4 持有的 t_3 的信息是通过之前和 t_3 做 join 操作获得的，t_4 持有的 t_2 的信息是通过之前和 t_2 做 join 操作获得的，而 t_4 持有的 t_1 的信息则是通过 t_2 传递过来获得的。

- 根节点的左子节点是 $(t_3, 2, 2)$ 。$(t_3, 2, 2)$ 的 aclk 的值是 2 表示线程 t_4 此时持有的 t_3 的信息是在 local time 为 2 时与线程 t_3 做的 join 获取的，clk 值为 2 表示当时线程 t_3 的 local time 是 2 。
- 根节点的右子节点是 $(t_2, 2, 1)$ 。线程 t_4 此时持有的 t_2 的信息是在 local time 为 1 时与线程 t_2 做的 join 获得的，并且当时线程 t_2 的 local time 是 2 。
- 节点 $(t_2, 2, 1)$ 只有一个子节点 $(t_1, 1, 1)$。对于线程 t_4 来说，持有的 t_1 的信息是通过 t_2 传递获得的。线程 t_2 是在 local time 为 1 时与线程 t_1 做的 join，并且当时线程 t_1 的 local time 就是 1 。

图 3 右边的树对应图 2(b) 中线程 t_4 在事件 e_7 后的 tree clock：

- 因为是 t_4 的 tree clock，所以根节点 $(t_4, 2, \perp)$ 的 tid 就是 t_4 ，根节点的 clk 值是 2，这是因为在事件 e_7 后线程 t_4 的 local time 就是 2，根节点的 aclk 的值一定是 $\perp$
- 通过树的结构可以看出来，此时 t_4 持有的 t_1, t_2, t_3 的信息都是通过之前和 t_3 做 join 操作获得的，而 t_4 持有的 t_1, t_2 的信息则是通过 t_3 传递过来获得的。
- 根节点只有一个子节点 $(t_3, 3, 2)$ 。表示线程 t_4 此时持有的 t_3 的信息是在 local time 为 2 时与线程 t_3 做的 join 获取的，当时线程 t_3 的 local time 是 3 。
- 节点 $(t_3, 3, 2)$ 的左子节点是 $(t_2, 1, 2)$ 。线程 t_3 此时持有的 t_2 的信息是在 local time 为 2 时与线程 t_2 做的 join ，并且当时线程 t_2 的 local time 是 1 。
- 节点 $(t_3, 3, 2)$ 的右子节点是 $(t_1, 1, 1)$。线程 t_3 是在 local time 为 1 时与线程 t_1 做的 join，并且当时线程 t_1 的 local time 就是 1 。

Algorithm

Tree clock 详细的数据结构及算法如下图所示：

Algorithm 2: The tree clock data structure.

```
// Initialize a tree clock for thread t
function Init(t)
    Let u ← (t, 0, ⊥)
    Make u the root of T
    Let ThrMap(t) ← u

// Get the clock for thread t
function Get(t)
    if TC.ThrMap(t) ≠ ⊥ then
        Let u ← ThrMap(t)
        return u.clk
    return 0

// Increment the clock of the root thread
function Increment(i)
    Let z ← T.root
    z.clk ← z.clk + i

// True iff ⊑ TC'
function LessThan(TC')
    Let z ← T.root
    return z.clk ≤ TC'.Get(z.tid)

// Update with ⊔ TC'
function Join(TC')
    Let z' ← TC'.T.root
    if z'.clk ≤ Get(z'.tid) then
        return
    Let S ← an empty stack
    getUpdatedNodesJoin (S, z')
    detachNodes (S)
    attachNodes (S)
    // Place the updated subtree under the root of T
    Let w ← ThrMap(z'.tid)
    Let z ← T.root
    Assign w.aclk ← z.clk
    pushChild (w, z)

// Monotone copy, assumes that this ⊑ TC'
function MonotoneCopy(TC')
    Let z' ← TC'.T.root
    Let z ← T.root
    Let S ← an empty stack
    getUpdatedNodesCopy (S, z', z)
    detachNodes (S)
    attachNodes (S)
    // New root has the same tid as the root of TC'.T
    Assign T.root ← ThrMap(z'.tid)

// Populate S with a pre-order traversal of the subtree rooted at u'
    with nodes whose clock has progressed
routine getUpdatedNodesJoin(S, u')
    foreach v' in Chld(u') do
        if Get(v'.tid) < v'.clk then getUpdatedNodesJoin (S, v');
        else if v'.aclk ≤ Get(u'.tid) then break;
    Push u' in S

// Detach from T the nodes with tid that appears in S
routine detachNodes(S)
    foreach v' in S do
        if ThrMap(v'.tid) ≠ ⊥ then
            Let v ← ThrMap(v'.tid)
            if v ≠ T.root then
                Let x ← Prnt(v)
                Remove v from Chld(x)

// Re-attach the nodes of T with tid that appears in S to obtain the
    shape corresponding to TC'.T
routine attachNodes(S)
    while S is not empty do
        Let u' ← pop S
        if ThrMap(u'.tid) ≠ ⊥ then
            Let u ← ThrMap(u'.tid)
        else
            Let u ← (u'.tid, 0, ⊥)
            Let ThrMap(u.tid) ← u
        Assign u.clk ← u'.clk
        Let y' ← Prnt(u')
        if y' ≠ ⊥ then
            Assign u.aclk ← u'.aclk
            Let y ← ThrMap(y'.tid)
            pushChild (u, y)

// Similar to getUpdatedNodesJoin
routine getUpdatedNodesCopy(S, u', z)
    foreach v' in Chld(u') do
        if Get(v'.tid) < v'.clk then
            getUpdatedNodesCopy (S, v', z)
        else
            if z ≠ ⊥ and v'.tid = z.tid then Push v' in S;
            if v'.aclk ≤ Get(u'.tid) then break;
    Push u' in S

// Push u in the front of head of Chld(v)
routine pushChild(u, v)
    Assign Prnt(u) ← v
    Push u to the front of Chld(v)
```

我们还是举例说明 tree clock 执行 Join 和 MonotoneCopy 这两个关键操作的算法流程。

Join

我们用 $TC_2.Join(TC_1)$ 表示 joins the tree clock TC1 to TC2。

以图 4 说明，$TC_2.Join(TC_1)$ 的流程：

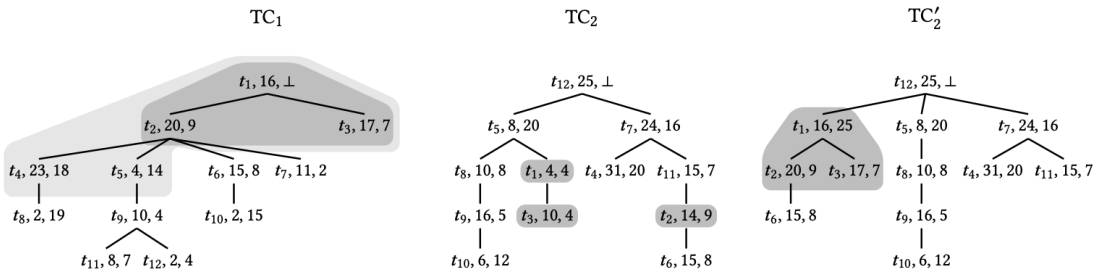


Figure 4: Illustration of TC_2 .Join(TC_1). Light gray marks the nodes of TC_1 whose time is compared to the time of the respective thread in TC_2 (i.e., the total iterations in Line 37). Dark gray marks the nodes that are updating/being updated (i.e., the size of $\mathcal{S}$). TC_2' is the result of the join, where dark gray marks the sub-tree updated by Join.

- `Let z' = TC'.T.root`

  z’ 即 TC_1 的根节点 $(t_1, 16, \perp)$

- `if z'.clk <= Get(z'.tid) then return`

  z’．clk 为 16，z’．tid 为 t_1；Get(z’．tid) 即 TC_2 中 tid 为 t_1 的节点的 clk 值，TC_2 中 tid 为 t_1 的节点为 $(t_1, 4, 4)$，其 clk 值为 4

  z’．clk = 16 <= Get(z’．tid) = 4 不成立，所以需要继续执行 Join 剩余流程。（这里 z’．clk <= Get(z’．tid) 就是判断 TC_2 中持有的关于 t_1 的信息是否比 TC_1 更新）

- `Let S = an empty stack; getUpdatedNodesJoin(S, z')`

  执行完 `getUpdatedNodesJoin(S, z')` 后，S 的内容如下：

  $[(t_1, 16, \perp), (t_3, 17, 7), (t_2, 20, 9)]$，注意 S 是栈

- `detachNodes(S)` 就是将 TC_2 中与 S 中节点 tid 相同的节点从 TC_2 中“拆卸”下来，即将 TC_2 中的 $(t_1, 4, 4)$、$(t_3, 10, 4)$ 和 $(t_2, 14, 9)$ “拆卸”下来

- `attachNodes(S)` 就是将 `detachNodes(S)` 中”拆卸”下来的节点再“附加”到恰当的位置：

  - 第一个处理的 S 中的节点是 $(t_1, 16, \perp)$，记作 u’，对应的 TC_2 中的节点就是 $(t_1, 4, 4)$，记作 u。更新 u 的 clk 为 u’ 的 clk，更新 u 的 aclk 为 u’ 的 aclk，即 $(t_1, 4, 4)$ 变为 $(t_1, 16, \perp)$，由于 u’ 是 TC_1 的根结点，我们暂时先把 u 先放一放，等下再将其“附加”到 TC_2 中恰当的位置。

  - 第二个处理的 S 中的节点是 $(t_3, 17, 7)$，记作 u’，对应的 TC_2 中的节点就是 $(t_3, 10, 4)$，记作 u。更新 u 的 clk 为 u’ 的 clk，更新 u 的 aclk 为 u’ 的 aclk，即 $(t_3, 10, 4)$ 变为 $(t_3, 17, 7)$，在 TC_1 中 u’ $(t_3, 17, 7)$ 的父节点是 $(t_1, 16, \perp)$，所以我们将 u “附加”到 TC_2 中 tid 为 t_1 的节点上，作为第一个子节点。

  - 第三个处理的 S 中的节点是 $(t_2, 20, 9)$，流程与 $(t_3, 17, 7)$ 类似不再赘述，需要注意的是，我们在“拆卸”和“附加”时原本节点的子节点也会随着父节点一起动。

- `Let w = ThrMap(z'.tid); Let z = T.root; Assign w.aclk = z.clk; pushChild(w, z)`

  最后将我们在 `attachNodes(S)` 中遗留下来的 $(t_1, 16, \perp)$ 进行处理，首先将 aclk 的值设置为 TC_2 的根节点的 clk 值，即 $(t_1, 16, \perp)$ 变为 $(t_1, 16, 25)$，然后“附加”到根结点上，作为根结点的第一个子节点。

MonotoneCopy

我们用 $TC_2.MonotoneCopy(TC_1)$ 表示 copy TC1 to TC2 when we know that TC2 $\sqsubseteq$ TC1 。

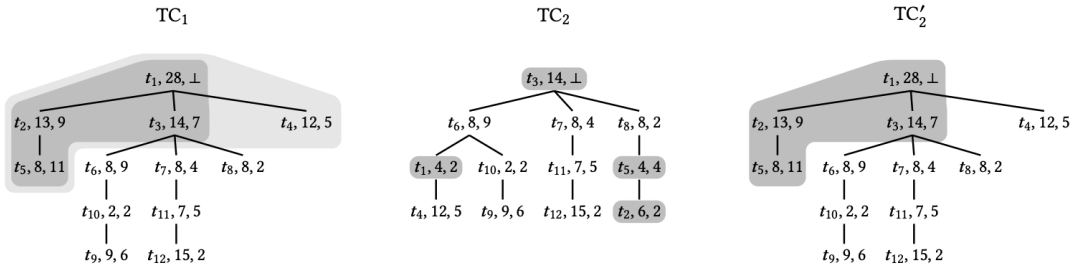


Figure 5: Illustration of TC_2 .MonotoneCopy(TC_1). Light gray marks the nodes of TC_1 whose time is compared to the time of the respective thread in TC_2 (i.e., the total iterations in Line 63). Dark gray marks the nodes that are updating/being updated (i.e., the size of $\mathcal{S}$). TC'_2 is the result of the copy, where dark gray marks the sub-tree updated by MonotoneCopy. Node $(t_3, 14, \perp)$ (i.e., the root) of TC_2 is updated although t_3 has not progressed in TC_1, as it is placed under the new root $(t_1, 28, \perp)$ in TC'_2.

$TC_2.MonotoneCopy(TC_1)$ 的流程就不再详细解释了，这里以图 5 为例简单描述下：

- `Let z' = TC'.T.root; Let z = T.root`

  z’ 即 TC_1 的根结点 $(t_1, 28, \perp)$，z 即 TC_2 的根节点 $(t_3, 14, \perp)$

- `getUpdatedNodesCopy(S, z', z)`

  执行完 `getUpdatedNodesCopy(S, z', z)` 后，S 的内容为：$[(t_1, 28, \perp), (t_3, 14, 7), (t_2, 13, 9), (t_5, 8, 11)]$

- `detachNodes(S)` 和 `attachNodes(S)` 的流程与 Join 是一样的，最后 `Assign T.root = ThrMap(z'.tid)` 将 $(t_1, 28, \perp)$ 设置为新的 TC_2 的根节点。

Conclusion

本文提出了 tree clock 这种数据结构，一种用于在并发执行中维护逻辑时间的新数据结构。与 vector clock 相比，核心思想就是 tree clock 额外保存了“当前 tree clock 中每一个 thread 的 clock 的值是何时从何处获取到的”这一信息，利用这一信息 tree clock 可以在 sublinear 时间内执行 join 和 copy 操作，从而尽可能避免了由于冗余操作造成的时间开销。

原本中还有很多部分都涉及到一些复杂的数学证明，我就没有涉猎了。

P.S. 这篇论文获得 ASPLOS 2022 的 Best Paper Awards 我觉得当之无愧。

- genindex
- modindex
- search

7.6 GWP-ASan

7.6.1 GWP-ASan 原理剖析

序言

GWP-ASan 是一个概率性内存错误检测工具，是以内存分配器的方式实现的。概率性是指随机保护某些堆分配，在性能和捕获内存错误之间有一个 tradeoff。

Address sanitizer, thread sanitizer 等由编译时插桩和运行时库两部分组成，需要从源码重新编译程序。而 GWP-ASan 则不需要从源码重新编译，因为 GWP-ASan 是以内存分配器的方式实现的。

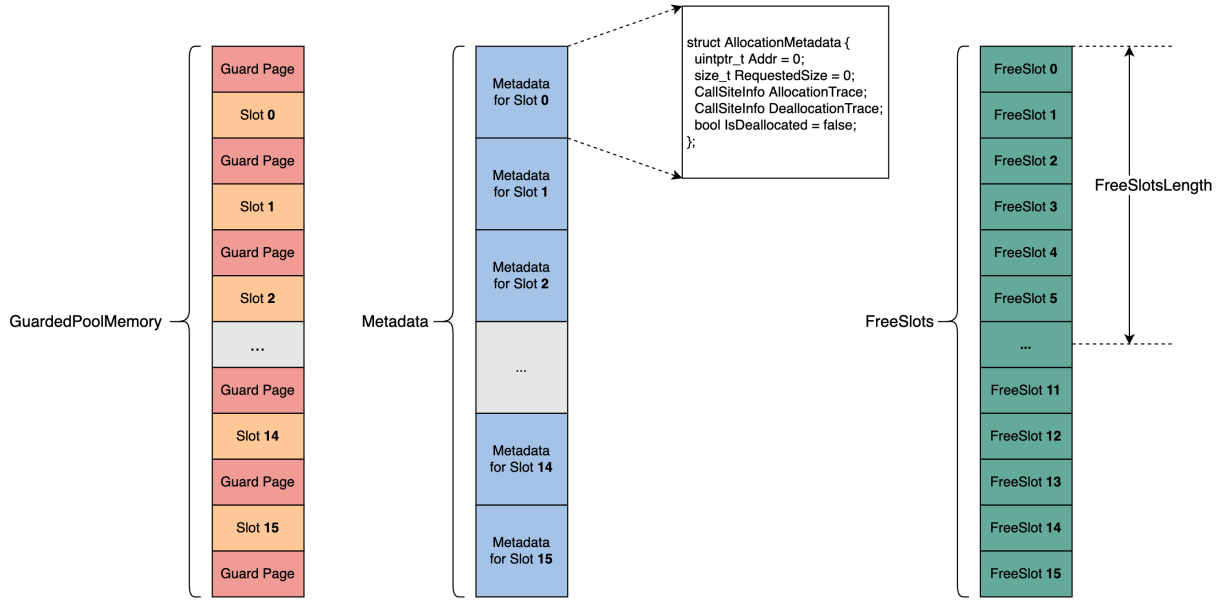
GWP-ASan 实际上是有多个实现的：

- TCMalloc
  - https://google.github.io/tcmalloc/gwp-asan.html
  - https://github.com/google/tcmalloc/blob/master/tcmalloc/guarded_page_allocator.h
- Chromium
  - https://chromium.googlesource.com/chromium/src/+/lkgr/docs/gwp_asan.md
  - https://chromium.googlesource.com/chromium/src/+/refs/heads/main/components/gwp_asan/
- Android
  - https://developer.android.com/ndk/guides/gwp-asan
- llvm-project
  - https://github.com/llvm/llvm-project/tree/main/compiler-rt/lib/gwp_asan

本文 GWP-ASan 原理剖析是基于的 llvm-project 中的 GWP-ASan，llvm-project 版本是 cef07169ec9f46fd25291a3218cf12bef324ea0c

了解 GWP-ASan 原理建议先看下 https://sites.google.com/a/chromium.org/dev/Home/chromium-security/articles/gwp-asan 有一个基本的了解。

原理



GWP-ASan 的核心数据结构就是上图中的 GuardedPoolMemory, Metadata, FreeSlots。

- GuardedPoolMemory
  - 包含 16 个 Slot 和 16+1 个 GuardPage。Slot 和 GuardPage 的大小都是 PageSize。Slot 的数量可以由参数 MaxSimultaneousAllocations (Number of simultaneously-guarded allocations available in the pool. Defaults to 16) 通过环境变量进行设置
  - 每一次由 GWP-ASan 分配出去的内存，都是位于在 GuardedPoolMemory Slot 中的内存
    * 每一次可分配的内存大小，最大为 PageSize
    * 每一个 Slot 用于一次内存分配后，该 Slot 不会再用于另外的内存分配，直至这块 chunk 被释放。假设上一次分配的内存位于 Slot 0 中，并且分配的内存还没有被释放，就算 Slot 0 中还有足够的大小可供这一次分配，这一次分配的内存还是会去其他的 Slot 中分配
- Metadata
  - 每一个 GuardedPoolMemory Slot 对应 Metadata 中的一个 AllocationMetadata
  - Metadata 中记录了 Addr, RequestedSize, AllocationTrace, DeallocationTrace, IsDeallocated 的信息
- FreeSlots
  - FreeSlot 中记录的是之前分配出去的且当前已经被释放了的 GuardedPoolMemory Slot Index。如果 GuardedPoolMemory 中的 16 个 Slot 都已经分配出去过了（注，这 16 个 Slot 当前有可能已经被释放、也有可能没有释放），此时需要再分配内存时，是随机去 FreeSlots 选择一个 FreeSlot 对应的 GuardedPoolMemory Slot 进行分配。如果 FreeSlots 中没有 FreeSlot，则分配失败
  - FreesSlotsLength 记录的是当前 FreeSlots 的有效长度

- 检测到内存错误立即 Crash
  - 整个 GuardedPoolMemory 是通过 mmap 申请的，初始时都是 PROT_NONE 权限。只有在将 Slot 的一部分大小分配出去时，才将该 Slot 的权限通过 mprotect 设置为 PROT_READ | PROT_WRITE，等到释放这块内存时，又将 Slot 权限设置为 PROT_NONE
  - 这样一旦有 heap buffer overflow, heap buffer underflow 或者 use after free 这样的错误，就会因为没有权限访问而 crash

检测 heap buffer overflow

例子

```cpp
#include <cstdlib>

int main() {
  char *Ptr = reinterpret_cast<char *>(malloc(4000));
  volatile char x = *(Ptr + 4016);
  volatile char y = *(Ptr + 4096);
  return 0;
}
```

```
$ clang++ -fsanitize=scudo heap_buffer_overflow.cpp -o heap_buffer_overflow
$ GWP_ASAN_OPTIONS='SampleRate=1' ./heap_buffer_overflow
```

注意到上述例子 heap buffer overflow 的例子，有时候能够检测到 `*(Ptr + 4016)` 这次越界访问，有时候则检测不到 `*(Ptr + 4016)` 这次越界访问（只检测到了 `*(Ptr + 4096)` 的越界访问）。多次运行./heap_buffer_overflow 会得到两种报告：

- Case1, Buffer Overflow at 4096 bytes to the right of a 4000-byte allocation

```
*** GWP-ASan detected a memory error ***
Buffer Overflow at 0x7bfff7a41000 (4096 bytes to the right of a 4000-byte␣
↪allocation at 0x7bfff7a40000) by thread 235485 here:
  #0 heap_buffer_overflow(+0x290b9) [0x55555557d0b9]
  #1 heap_buffer_overflow(+0x293a5) [0x55555557d3a5]
  #2 heap_buffer_overflow(+0x295e9) [0x55555557d5e9]
  #3 /lib/x86_64-linux-gnu/libpthread.so.0(+0x12730) [0x7ffff7ca0730]
  #4 heap_buffer_overflow(main+0x2e) [0x5555555a115e]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7ffff7aac09b]
  #6 heap_buffer_overflow(_start+0x2a) [0x55555555a53a]

0x7bfff7a41000 was allocated by thread 235485 here:
  #0 heap_buffer_overflow(+0x290c9) [0x55555557d0c9]
```

(continues on next page)

(continued from previous page)

```
  #1 heap_buffer_overflow(+0x26d25) [0x55555557ad25]
  #2 heap_buffer_overflow(+0x27edb) [0x55555557bedb]
  #3 heap_buffer_overflow(+0x41f82) [0x555555595f82]
  #4 heap_buffer_overflow(main+0x19) [0x5555555a1149]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7ffff7aac09b]
  #6 heap_buffer_overflow(_start+0x2a) [0x55555555a53a]

*** End GWP-ASan report ***
```

- Case2, Buffer Overflow at 4016 bytes to the right of a 4000-byte allocation

```
*** GWP-ASan detected a memory error ***
Buffer Overflow at 0x7bfff7a41010 (4016 bytes to the right of a 4000-byte␣
↪allocation at 0x7bfff7a40060) by thread 235483 here:
  #0 heap_buffer_overflow(+0x290b9) [0x55555557d0b9]
  #1 heap_buffer_overflow(+0x293a5) [0x55555557d3a5]
  #2 heap_buffer_overflow(+0x295e9) [0x55555557d5e9]
  #3 /lib/x86_64-linux-gnu/libpthread.so.0(+0x12730) [0x7ffff7ca0730]
  #4 heap_buffer_overflow(main+0x21) [0x5555555a1151]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7ffff7aac09b]
  #6 heap_buffer_overflow(_start+0x2a) [0x55555555a53a]

0x7bfff7a41010 was allocated by thread 235483 here:
  #0 heap_buffer_overflow(+0x290c9) [0x55555557d0c9]
  #1 heap_buffer_overflow(+0x26d25) [0x55555557ad25]
  #2 heap_buffer_overflow(+0x27edb) [0x55555557bedb]
  #3 heap_buffer_overflow(+0x41f82) [0x555555595f82]
  #4 heap_buffer_overflow(main+0x19) [0x5555555a1149]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7ffff7aac09b]
  #6 heap_buffer_overflow(_start+0x2a) [0x55555555a53a]

*** End GWP-ASan report ***
```

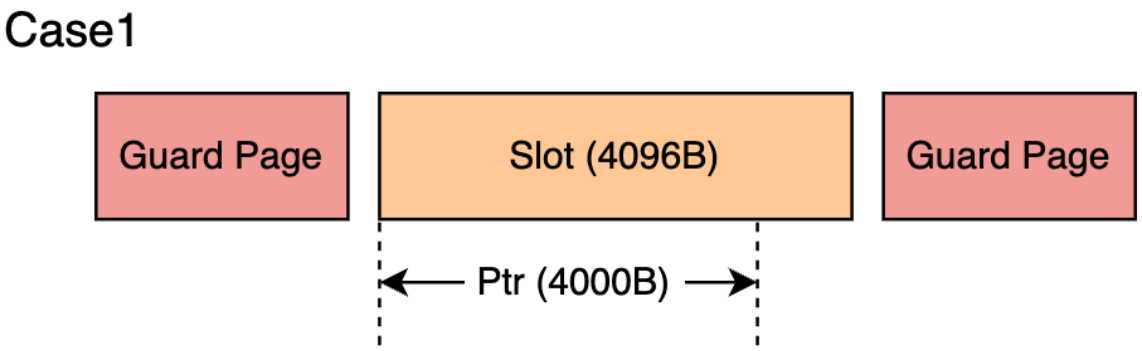
检测原理

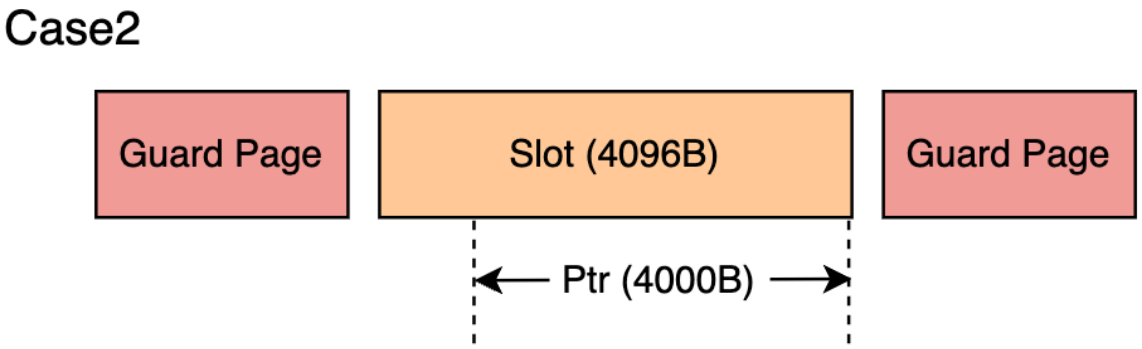
为什么会出现上述现象，原因见下图：

- GWP-ASan 是对于每一次 allocate 都是随机选择 left-align or right-align 的
- 对于 case1 来说，返回给用户的 Ptr 地址就是 Slot 的起始地址，所以当访问 `*(Ptr + 4016)` 时，Ptr + 4016 还是在该 Slot 中，也就是有权限访问的，也就不会 crash，所以检测不到 `*(Ptr + 4016)` 此处溢出。
- 对于 case2 来说，返回给用户的 Ptr 地址就是 SlotEnd - Size，所以当访问 `*(Ptr + 4016)` 时，Ptr +

4016 是位于该 GuardPage 中的，没有权限访问，故能检测出 `*(Ptr + 4016)` 此处溢出。

- 对于 case1 和 case2 来说，都能检测 `*(Ptr + 4096)` 到这次溢出，这是因为 case1 和 case2 中，Ptr + 4096 都位于 Guard Page 中，因此都能检测到这里的溢出。





检测 use after free

例子

```
#include <cstdlib>

int main() {
  char *Ptr = reinterpret_cast<char *>(malloc(10));

  for (unsigned i = 0; i < 10; ++i) {
    *(Ptr + i) = 0x0;
  }

  free(Ptr);
```

(continues on next page)

(continued from previous page)

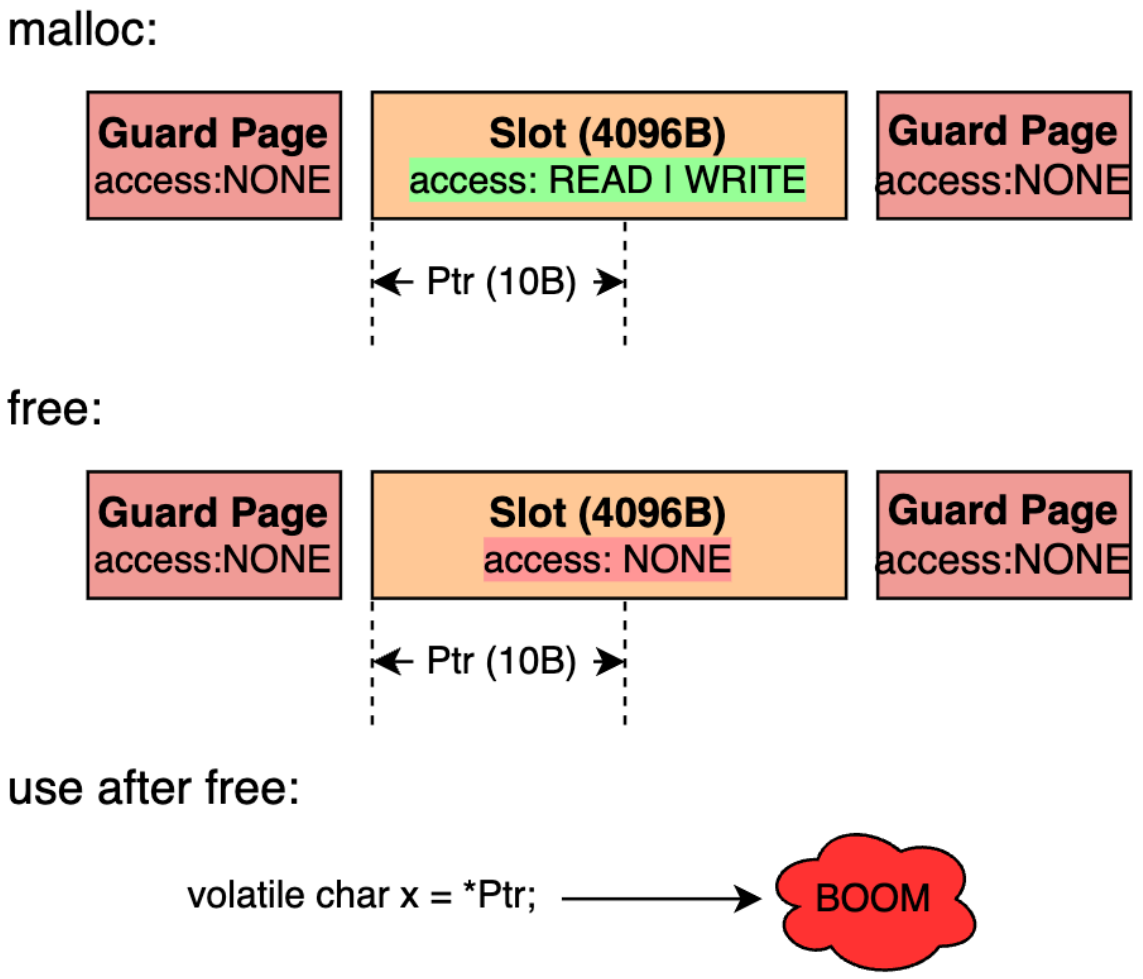
```
  volatile char x = *Ptr;
  return 0;
}
```

```
$ clang++ -fsanitize=scudo use_after_free.cpp -o use_after_free
$ GWP_ASAN_OPTIONS='SampleRate=1' ./use_after_free
*** GWP-ASan detected a memory error ***
Use After Free at 0x7b287e319000 (0 bytes into a 10-byte allocation at␣
↪0x7b287e319000) by thread 307939 here:
  #0 ./use_after_free(+0x290b9) [0x5603b0fe40b9]
  #1 ./use_after_free(+0x293a5) [0x5603b0fe43a5]
  #2 ./use_after_free(+0x295e9) [0x5603b0fe45e9]
  #3 /lib/x86_64-linux-gnu/libpthread.so.0(+0x12730) [0x7f287e579730]
  #4 ./use_after_free(main+0x54) [0x5603b1008184]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7f287e38509b]
  #6 ./use_after_free(_start+0x2a) [0x5603b0fc153a]

0x7b287e319000 was deallocated by thread 307939 here:
  #0 ./use_after_free(+0x290c9) [0x5603b0fe40c9]
  #1 ./use_after_free(+0x26d25) [0x5603b0fe1d25]
  #2 ./use_after_free(+0x280eb) [0x5603b0fe30eb]
  #3 ./use_after_free(+0x44dc4) [0x5603b0fffdc4]
  #4 ./use_after_free(main+0x50) [0x5603b1008180]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7f287e38509b]
  #6 ./use_after_free(_start+0x2a) [0x5603b0fc153a]

0x7b287e319000 was allocated by thread 307939 here:
  #0 ./use_after_free(+0x290c9) [0x5603b0fe40c9]
  #1 ./use_after_free(+0x26d25) [0x5603b0fe1d25]
  #2 ./use_after_free(+0x27edb) [0x5603b0fe2edb]
  #3 ./use_after_free(+0x41f82) [0x5603b0ffcf82]
  #4 ./use_after_free(main+0x19) [0x5603b1008149]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7f287e38509b]
  #6 ./use_after_free(_start+0x2a) [0x5603b0fc153a]

*** End GWP-ASan report ***
```

检测原理

为什么能检测到此处 use after free，原理见下图：

- GWP-ASan 在分配内存时，会将分配出去的 chunk 所在的 Slot 的权限通过 mprotect 设置为 PROT_READ | PROT_WRITE，等到释放这块内存时，又将 Slot 权限设置为 PROT_NONE。这样当访问已经释放的内存时会就是 crash



如果仔细思考下，GWP-ASan 检测 use-after-free 是有局限性的，考虑如下代码：

- 第 27 行的 use-after-free 就不会被检测出来，因为 underlying slot 又被用于分配了
- 第 33 行的 use-after-free 虽然会被检测出来，但是 allocation/deallocation stack traces 实际上并不匹配，报告中给出的是 Ptr17 的 allocation/deallocation stack traces

```
#include <cstdlib>

int main() {
```

(continues on next page)

(continued from previous page)

```
  // fill GuardedPoolMemory 16 slots
  char *Ptr1 = reinterpret_cast<char *>(malloc(10));
  char *Ptr2 = reinterpret_cast<char *>(malloc(10));
  char *Ptr3 = reinterpret_cast<char *>(malloc(10));
  char *Ptr4 = reinterpret_cast<char *>(malloc(10));
  char *Ptr5 = reinterpret_cast<char *>(malloc(10));
  char *Ptr6 = reinterpret_cast<char *>(malloc(10));
  char *Ptr7 = reinterpret_cast<char *>(malloc(10));
  char *Ptr8 = reinterpret_cast<char *>(malloc(10));
  char *Ptr9 = reinterpret_cast<char *>(malloc(10));
  char *Ptr10 = reinterpret_cast<char *>(malloc(10));
  char *Ptr11 = reinterpret_cast<char *>(malloc(10));
  char *Ptr12 = reinterpret_cast<char *>(malloc(10));
  char *Ptr13 = reinterpret_cast<char *>(malloc(10));
  char *Ptr14 = reinterpret_cast<char *>(malloc(10));
  char *Ptr15 = reinterpret_cast<char *>(malloc(10));
  char *Ptr16 = reinterpret_cast<char *>(malloc(10));
  // use and free Ptr1
  for (unsigned i = 0; i < 10; ++i) {
    *(Ptr1 + i) = 0x0;
  }
  free(Ptr1);
  // reuse Ptr1's underlying GuardedPoolMemory slot
  char *Ptr17 = reinterpret_cast<char *>(malloc(10));
  // use after free, false negative
  volatile char x = *Ptr1;
  // free Ptr17
  free(Ptr17);
  // use after free, wrong allocation/deallocation stack traces.
  volatile char y = *Ptr1;
  return 0;
}
```

```
*** GWP-ASan detected a memory error ***
Use After Free at 0x7b7f2fad5ff0 (0 bytes into a 10-byte allocation at␣
↪0x7b7f2fad5ff0) by thread 1005136 here:
  #0 ./use_after_free_dumb(+0x30c26) [0x55cc332dac26]
  #1 ./use_after_free_dumb(+0x31107) [0x55cc332db107]
  #2 ./use_after_free_dumb(+0x30e37) [0x55cc332dae37]
  #3 /lib/x86_64-linux-gnu/libpthread.so.0(+0x12730) [0x7f7f2fd1a730]
  #4 ./use_after_free_dumb(main+0x168) use_after_free_dumb.cpp:33:21  [0x55cc332ef3c8]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7f7f2fb4009b]
  #6 ./use_after_free_dumb(_start+0x2a) [0x55cc332c68da]
```

(continues on next page)

(continued from previous page)

```
0x7b7f2fad5ff0 was deallocated by thread 1005136 here:
  #0 ./use_after_free_dumb(+0x30c26) [0x55cc332dac26]
  #1 ./use_after_free_dumb(+0x2fbb4) [0x55cc332d9bb4]
  #2 ./use_after_free_dumb(+0x306d2) [0x55cc332da6d2]
  #3 ./use_after_free_dumb(main+0x162) use_after_free_dumb.cpp:31:3 [0x55cc332ef3c2]
  #4 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7f7f2fb4009b]
  #5 ./use_after_free_dumb(_start+0x2a) [0x55cc332c68da]

0x7b7f2fad5ff0 was allocated by thread 1005136 here:
  #0 ./use_after_free_dumb(+0x30c26) [0x55cc332dac26]
  #1 ./use_after_free_dumb(+0x2fbb4) [0x55cc332d9bb4]
  #2 ./use_after_free_dumb(+0x30547) [0x55cc332da547]
  #3 ./use_after_free_dumb(+0x3ee02) [0x55cc332e8e02]
  #4 ./use_after_free_dumb(+0x3e935) [0x55cc332e8935]
  #5 ./use_after_free_dumb(main+0x143) use_after_free_dumb.cpp:27:9 [0x55cc332ef3a3]
  #6 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7f7f2fb4009b]
  #7 ./use_after_free_dumb(_start+0x2a) [0x55cc332c68da]

*** End GWP-ASan report ***
```

GWP-ASan 与 ASan 的对比

1. ASan 能检测栈、堆、全局变量的内存错误，而 GWP-ASan 只能检测堆上的内存错误，并且 GWP-ASan 的内存错误检测能力是概率性的 (probabilistic)
2. ASan 的额外性能开销和内存开销远高于 GWP-ASan，ASan 通常会增加 2-3 倍的性能和内存开销，而 GWP-ASan 的额外开销则基本可以忽略不计。这是这样 GWP-ASan 可以在生产环境/版本中使用，比如 GWP-ASan 在 Chrome 浏览器中发现了很多内存错误
3. GWP-ASan 发现的错误中大约有 90% 都是 use-after-frees，剩下的则是 out-of-bounds reads and writes

总结

实际上 GWP-ASan 的原理非常简单，很早以前就在 ElectricFence or PageHeap 中就有所应用。而这种通过概率采样的方式去处理问题的思路还是非常有意思的，虽然采样的方式会牺牲一定的准确性与能力，但是另一方面就可以在基本不影响应用的环境下去发现问题。不止是 GWP-ASan，AutoFDO 也是通过随机采样而不是程序插桩的方式，在不影响原本程序性能的情况下，收集程序运行时信息指导反馈编译时优化。

- genindex
- modindex
- search

- genindex
- modindex
- search

CHAPTER

EIGHT

MISC

8.1 Exploring C++ Undefined Behavior Using Constexpr

本文是对 Exploring Undefined Behavior Using Constexpr 这篇文章的简单翻译学习。

实际上本文的内容和 LLVM/Clang 不是那么的相关。

在 C++ 中有很多 undefined behavior (即未定义行为，常简写为 UB)，一般来讲，我们应该尽可能的避免 UB。常见的 UB 包括 overflow (溢出)，out-of-bounds memory access (内存越界访问) 等等。现有的很多工具都可以检测出 UB，可以把这些工具分成两类：基于静态分析（编译器警告，clang-tidy 等）和基于动态分析 (基于 LLVM/Clang 的 UndefinedBehaviorSanitizer 和 AddressSanitizer 等)。

作为一个优秀的程序员，我们应该了解 UB，以便在编程时避免 UB，在 code review 时及时发现 UB。在编写代码时，我们可能觉得某部分代码是 UB，所以我们想 google 来查阅资料来看这部分代码到底是不是 UB，但是通常我们并不知道这种 UB 对应的术语是什么，或者我们找到了一篇相关文章，但文章中没有谈到我们正在处理的这种 UB 的特定情况。

这篇文章中提出了一种方法，通过 C++ 中的 constant expressions 配合 godbolt 来迅速地探究某段代码是否为 UB。

8.1.1 Constant Expressions

C++ 中的 constant expressions 有很多需要满足的限制，其中一个就是：一个 constant expression 中不允许出现 UB (undefined behavior is not allowed in a constant expression)。

C++ 标准在 [expr.const]p4 和 [expr.const]p4.6 对 *constant expressions* and *undefined behavior* 有如下的说明：

> An expression e is a core constant expression unless the evaluation of e, following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:
>
> - ...
> - an operation that would have undefined behavior as specified in [intro] through [cpp] of this document [Note: including, for example, signed integer overflow ([expr.prop]), certain pointer arithmetic ([expr.add]), division by zero, or certain shift operations — end note];

所以如果我们有一个操作，这个操作是 UB，那么我们就不能在 constant expression 的上下文中实现这个操作，根据 C++ 标准，编译器必须以警告或者错误的形式报告这种行为，事实上，目前编译器是以报错的形式来报告这种行为的。这样我们就能够让编译器自己告诉我们，什么是 UB，什么不是。

godbolt 是一个以网页形式提供服务的交互型编译器，我们可以通过 godbolt 来测试一些简单的代码片段。

由此 godbolt + undefined behavior is not allowed in a constant expression 为我们提供了一种交互式的快速的探究什么样的代码是 UB，什么样的代码不是 UB 的方法。但是也同样因为 constant expression 需要满足很多限制，所以有一些 UB 不能以这种方式被测试。例如，堆内存分配和 reinterpret_cast 都不能出现在 constant expression 中，因此我们就不能以这种方式来探究 use after free 和 strict aliasing violations 这类 UB。

需要注意 C++ 中有很多特殊的例外情况，通过这种方式得到的结果也不一定是正确的，有很多具体的例子需要深入探究学习。

8.1.2 An Example with Arithmetic Overflow

首先以算术溢出为例子，讲一下具体的操作。

```
#include <limits>

void f1()
{
    unsigned int x1 =
        std::numeric_limits<unsigned int>::max() + 1;   // Overflow one over the max
    unsigned int x2 = 0u - 1u;                          // Wrap one below the min
    int y1 = std::numeric_limits<int>::max() + 1;       // Overflow one over the max
    int y2 = std::numeric_limits<int>::min() - 1;       // Underflow one below the min
}
```

上述代码片段中，有 *unsigned int* 的 overflow (上溢)，有 *unsigned int* 的下溢 (underflow)，有 *signed int* 的上溢，有 *signed int* 的下溢。究竟哪些是 UB 哪些不是 UB?

我们先回过头看一下 *constexpr*，C++ 标准中要求 [dcl.constexpr]p9：被 *constexpr* 说明符所声明的变量必须是 literal types，必须被初始化，并且必须使用 constant expressions 初始化。

> A constexpr specifier used in an object declaration declares the object as const. Such an object shall have literal type and shall be initialized. In any constexpr variable declaration, the full-expression of the initialization shall be a constant expression (7.7). ···

我们将上面的代码片段加上 *constexpr* 说明符，这样用于初始化被声明的变量的表达式就必须是 constant expressions。也就是说 `std::numeric_limits<unsigned int>::max()+1`, `0u-1u`, `std::numeric_limits<int>::max()+1`, `std::numeric_limits<int>::min()-1` 必须是 constant expressions。如果它们中存在 UB，那么它们就不是 constant expressions，编译器就会报错！我们将上述代码复制到 godbolt 中进行编译，查看编译器的输出结果 https://godbolt.org/z/0vx_j6

x86-64 clang (trunk) 的输出如下：

```
<source>:7:19: error: constexpr variable 'y1' must be initialized by a constant␣
↪expression
    constexpr int y1=std::numeric_limits<int>::max()+1;
                  ^  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:7:53: note: value 2147483648 is outside the range of representable values of␣
↪type 'int'
    constexpr int y1=std::numeric_limits<int>::max()+1;
                                                    ^
<source>:8:19: error: constexpr variable 'y2' must be initialized by a constant␣
↪expression
    constexpr int y2=std::numeric_limits<int>::min()-1;
                  ^  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:8:53: note: value -2147483649 is outside the range of representable values␣
↪of type 'int'
    constexpr int y2=std::numeric_limits<int>::min()-1;
                                                    ^
2 errors generated.
Compiler returned: 1
```

x86-64 gcc (trunk) 的输出如下：

```
<source>: In function 'void f()':
<source>:7:53: warning: integer overflow in expression of type 'int' results in '-
↪2147483648' [-Woverflow]
    7 |     constexpr int y1=std::numeric_limits<int>::max()+1;
      |                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~
<source>:7:54: error: overflow in constant expression [-fpermissive]
    7 |     constexpr int y1=std::numeric_limits<int>::max()+1;
      |                                                      ^
<source>:7:54: error: overflow in constant expression [-fpermissive]
<source>:8:53: warning: integer overflow in expression of type 'int' results in
↪'2147483647' [-Woverflow]
    8 |     constexpr int y2=std::numeric_limits<int>::min()-1;
      |                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~
<source>:8:54: error: overflow in constant expression [-fpermissive]
    8 |     constexpr int y2=std::numeric_limits<int>::min()-1;
      |                                                      ^
<source>:8:54: error: overflow in constant expression [-fpermissive]
Compiler returned: 1
```

根据编译器的输出可知：**signed int 的 overflow 和 underflow 都是 UB，unsigned int 的 overflow 和 underflow 都不是 UB。**

C++ 标准的 [expr]p4 和 [basic.fundamental]p2中有相应的说明：

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of

> representable values for its type, the behavior is undefined. [Note: Treatment of division by zero, forming a remainder using a zero divisor, and all floating-point exceptions vary among machines, and is sometimes adjustable by a library function. — end note]
>
> ···The range of representable values for the unsigned type is 0 to 2^(N−1) (inclusive); arithmetic for the unsigned type is performed modulo 2^N. [Note: Unsigned arithmetic does not overflow. Overflow for signed arithmetic yields undefined behavior ([expr.pre]). — end note]

Arithmetic Overflow 的另一种有趣的情况是：

```
constexpr int x = std::numeric_limits<int>::min() / -1;
```

如果我们将上述代码放到 godbolt 中 https://godbolt.org/z/Ecp88m，我们会发现上述代码也是 UB。假设 int 是 64 bit，那么 int 能表示的最大值是 2147483647，而 `std::numeric_limits<int>::min() / -1` 的结果是 2147483648，超出了 int 所能表示的返回，所以这是一个 UB。

通过这个例子我们可以看到 `undefined behavior is not allowed in a constant expression` 为我们提供了一种强大的方法来探究并识别 UB (尽管虽然这种方法不能处理所有的 UB)。

下面我们具体分析所有的能用这种方式来捕获的 UB。

8.1.3 Conversions and values that can not be represented

下面我们看一下，把一个 integral or floating-point type 的变量转换成一个 smaller sized type 是不是 UB。

```
constexpr unsigned int u = std::numeric_limits<unsigned int>::max();  // 1
constexpr int i = u;                                                   // Line 6

constexpr double d = static_cast<double>(std::numeric_limits<int>::max()) + 1;  // 2
constexpr int x = d;  // Line 10

constexpr double d2 = std::numeric_limits<double>::max();  // 3
constexpr float f = d2;                                    // Line 13
```

将上述代码放到 godbolt 中 https://godbolt.org/z/2ZfPKt，查看编译输出。

x86-64 clang (trunk) 的输出如下：

```
<source>:10:16: error: constexpr variable 'x' must be initialized by a constant␣
↪expression
 constexpr int x = d;
               ^   ~
<source>:10:20: note: value 2147483648 is outside the range of representable values␣
↪of type 'const int'
 constexpr int x = d;
                   ^
```

(continues on next page)

(continued from previous page)

```
<source>:13:18: error: constexpr variable 'f' must be initialized by a constant␣
↪expression
 constexpr float f = d2;
                 ^   ~~
<source>:13:22: note: value 1.797693134862316E+308 is outside the range of␣
↪representable values of type 'const float'
 constexpr float f = d2;
                     ^
2 errors generated.
Compiler returned: 1
```

根据 x86-64 clang (trunk) 编译器的输出可知：case1 是 well-defined，case2 和 case3 不是 well-defined。值得注意的是 x86-64 gcc (trunk) 只对 case2 报错。

根据 C++ 标准的 Integral conversions 部分 [conv.integral]p3 (this changes in C++20 it modulo 2^N)，case 1 是 impelmentation defined。

> If the destination type is signed, the value is unchanged if it can be represented in the destination type; otherwise, the value is implementation-defined.

根据 C++ 标准的 Floating-point conversions 部分 [conv.dobule]p1 和 Floating-integral conversions 部分 [conv.fpint]p1，case2 和 case3 是 UB。

> A prvalue of floating-point type can be converted to a prvalue of another floating-point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.
>
> A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [Note: If the destination type is bool, see [conv.bool]. — end note]

8.1.4 Division by zero

很多人都知道整型变量除以零是 UB，但是对浮点型变量除以零是否为 UB 则不确定。将下面的代码通过 goldbot 编译 https://godbolt.org/z/tRM6oF

```
constexpr int x = 1/0;         // Line 2
constexpr double d = 1.0/0.0; // Line 3
```

x86-64 clang (trunk) 的输出如下：

```
<source>:2:18: error: constexpr variable 'x' must be initialized by a constant␣
↪expression
   constexpr int x = 1/0;
```

(continues on next page)

(continued from previous page)

```
                  ^   ~~~
<source>:2:23: note: division by zero
   constexpr int x = 1/0;
                      ^
<source>:3:21: error: constexpr variable 'd' must be initialized by a constant␣
↪expression
   constexpr double d = 1.0/0.0;
                     ^   ~~~~~~~
<source>:3:28: note: floating point arithmetic produces an infinity
   constexpr double d = 1.0/0.0;
                            ^
<source>:2:23: warning: division by zero is undefined [-Wdivision-by-zero]
   constexpr int x = 1/0;
                      ^~
```

可以看到，整型变量除以零、浮点型变量除以零都是 UB。

8.1.5 Shifty characters

关于移位运算，我们可能想知道下面这些操作哪些是 UB：

1. Shifting greater than the bit-width of the type?
2. Shifting by a negative shift?
3. Shifting a negative number?
4. Shifting into the sign bit?

编写对应的测试代码如下：

```
void foo()
{
    static_assert(sizeof(int) == 4 && CHAR_BIT == 8 );
    constexpr int y1 = 1 << 32;   // Shifting greater than the bit-width
    constexpr int y2 = 1 >> 32;   // Shifting greater than the bit-width
    constexpr int y3 = 1 << -1;   // Shifting by a negative amount
    constexpr int y4 = -1 << 12;  // Shifting a negative number
    constexpr int y5 = 1 << 31;   // Shifting into the sign bit
}
```

查看编译输出 https://godbolt.org/z/p7onyC 发现：除了 Shifting into the sign bit 之外，其他的操作都是 UB。

前两种移位操作的情况 (Shifting greater than the bit-width of the type, Shifting by a negative shift) 在 C++ 标准 [expr.shift]p1 中有相关说明：

> The shift operators << and >> group left-to-right.

```
shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression
```

> The operands shall be of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. **The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand.**

第三种移位操作的情况 (Shifting a negative number) 在 C++20 之前是 UB [expr.shift]p2：

> The value of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are zero-filled. If E1 has an unsigned type, the value of the result is E1×(2^E2), reduced modulo one more than the maximum value representable in the result type. Otherwise, **if E1 has a signed type and non-negative value**, and E1×(2^E2) is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; **otherwise, the behavior is undefined**.

在 p0907r4 中被规定为 well-defined.

8.1.6 Everyones favorite pointer, nullptr

关于 nullptr，一些人可能很简单地认为：只要是涉及到 nullptr 的操作都是 UB。

下面是一段简单的示例代码：

```
constexpr int bar()
{
    constexpr int* p = nullptr;
    return *p;  // Unconditional UB
}

constexpr void foo()
{
    constexpr int x = bar();
}
```

上述代码的编译输出 https://godbolt.org/z/cyiVq9 与我们所预想的一致，确实是 UB。

```
<source>:1:15: error: constexpr function never produces a constant expression [-
↪Winvalid-constexpr]
constexpr int bar() {
              ^
<source>:3:12: note: read of dereferenced null pointer is not allowed in a constant␣
↪expression
    return *p;        // Unconditional UB
```

(continues on next page)

(continued from previous page)

```
          ^
<source>:7:19: error: constexpr variable 'x' must be initialized by a constant␣
↪expression
    constexpr int x = bar();
                  ^   ~~~~~
<source>:3:12: note: read of dereferenced null pointer is not allowed in a constant␣
↪expression
    return *p;        // Unconditional UB
           ^
<source>:7:23: note: in call to 'bar()'
    constexpr int x = bar();
```

下面，我们看一些比较复杂的情况。

通过 nullptr 来访问类的非静态成员，通过 nullptr 来访问类的静态成员，是否都是 UB?

```
struct A
{
    constexpr int g() { return 0;}
    constexpr static int f(){ return 1;}
};

static constexpr A* a=nullptr;

void foo()
{
    constexpr int x = a->f(); // case1
    constexpr int y = a->g(); // case2
}
```

case1 是通过 nullptr 访问类的静态成员，case2 是通过 nullptr 访问类的非静态成员。提交到 godbolt 编译后，发现 x86-64 clang(trunk) 和 x86-64 gcc(trunk) 的编译输出结果不一致 https://godbolt.org/z/7NrcFD。x86-64 clang(trunk) 认为 case2 是 UB，而 gcc 对 case1 和 case2 都没有报错。

事实上，虽然 case1 是 well-defined，但是 CWG defect report 315: Is call of static member function through null pointer undefined? 告诉我们，当通过 nullptr 访问静态成员时，没有 lvalue-to-rvalue 的转换。

8.1.7 More pointer fun

Incrementing pointer out of bounds

如果一个指针越界了，但是我们不使用该指针、不对该指针解引用，那么是否为 UB 呢?

```
static const int arrs[10]{};

void foo()
{
    constexpr const int* y = arrs + 11;
}
```

根据 https://godbolt.org/z/-E06pt，x86-64 clang(trunk) 报告了该错误，但是 x86-64 gcc(trunk) 并没有捕获到该错误。可以看到如果一个指针越界了，不过该指针后续是否用于其他操作中，都是 UB。但是，需要注意的是一个例外，如果一个指针只越界了一个元素，那么则不是 UB，如 std::end 就是用指向最后一个元素的后一个元素的指针来表示容器或者数组的结尾，https://en.cppreference.com/w/cpp/iterator/end。

Incrementing out of bounds but coming back in

```
constexpr int foo(const int *p)
{
    return *((p + 12) - 5); // ?
}

constexpr void bar()
{
    constexpr int arr[10]{};
    constexpr int x = foo(arr);
}
```

虽然中间的表达式 p + 12 越界，(p + 12) - 5 没有越界，但是这也是 UB。与上一种情况类似 x86-64 gcc(trunk) 同样没有捕获到该错误 https://godbolt.org/z/D4uayd。

C++ 标准 [expr.add]p4 中告诉我们什么样的索引是可接受的。

> When an expression J that has integral type is added to or subtracted from an expression P of pointer type, the result has the type of P. - If P evaluates to a null pointer value and J evaluates to 0, the result is a null pointer value. - Otherwise, if P points to an array element i of an array object x with n elements, the expressions P + J and J + P (where J has the value j) point to the (possibly-hypothetical) array element i + j of x if 0 ≤ i + j ≤ n and the expression P - J points to the (possibly-hypothetical) array element i - j of x if 0 ≤ i − j ≤ n. - Otherwise, the behavior is undefined.

Footnote 80:

> ···A pointer past the last element of an array x of n elements is considered to be equivalent to a pointer to a hypothetical element x[n] for this purpose; see [basic.compound].

上述标准涵盖了 incrementing out of bounds 和 incrementing out of bounds during an intermediate step 这两种情况。

在 [basic.compound]p3 说明了指向最后一个元素的后一个元素的指针是合法的 (one past the end is a valid pointer)：

> ···Every value of pointer type is one of the following:
>
> - a pointer to an object or function (the pointer is said to point to the object or function), or
> - a pointer past the end of an object ([expr.add]), or
>
> ···

Out of bounds access

没有什么好说的，越界访问是 UB。

```
constexpr int foo(const int *p)
{
    return *(p + 12);
}

constexpr void bar()
{
    constexpr int arr[10]{};
    constexpr int x = foo(arr);
}
```

https://godbolt.org/z/O2GqaX，clang 和 gcc 都报告除了该错误。

8.1.8 End of life

某个变量在其生命周期结束后被使用是一种很难被检测的 UB。因 constant expressions 中不允许内存分配，但是允许使用引用，所以下面用了一个函数返回对局部变量的引用的例子。

```
constexpr int& foo()
{
    int x = 0;

    return x;  // x will soon be out of scope
               // but we return it by reference
}  // bye bye x
```

(continues on next page)

(continued from previous page)

```
constexpr int bar()
{
    constexpr int x = foo();
    return x;
}
```

https://godbolt.org/z/hM607D gcc 和 clang 都报告了该 UB，clang 的错误报告可读性更好 (read of variable whose lifetime has ended) 。

8.1.9 Flowing off the end of a value returning function

```
constexpr int foo(int x)
{
    if (x)
        return 1;
    // Oppps we forgot the return 0;
}

void bar()
{
    constexpr int x = foo(0);
}
```

在函数 foo 中有两条可能被执行的路径，但只有一条路径上有 return 语句，如果 x == 0 则会发生未定义行为。

https://godbolt.org/z/AALpj5 gcc 和 clang 都报告了该 UB。

C++ 标准 [stmt.return]p2 对这种情况进行了说明：

> …Flowing off the end of a constructor, a destructor, or a non-coroutine function with a cv void return type is equivalent to a return with no operand. Otherwise, flowing off the end of a function other than main or a coroutine ([dcl.fct.def.coroutine]) results in undefined behavior.

8.1.10 Modifying a constant object

尝试修改一个 constant 对象是 UB。下面的代码通过 const_cast 去除了变量的 const 属性，然后对变量进行了修改。

```
struct B
{
    int i;
    double d;
};
```

(continues on next page)

(continued from previous page)

```
constexpr B bar()
{
    constexpr B b = { 10, 10.10 };
    B *p = const_cast<B *>(&b);

    p->i = 11;
    p->d = 11.11;

    return *p;
}

void foo()
{
    constexpr B y = bar();
}
```

编译器的输出结果：https://godbolt.org/z/AYulw0 ， clang 捕获到了该 UB，gcc 又没有…

C++ 标准在 [dcl.type.cv]p4 进行了说明：

> Except that any class member declared mutable ([dcl.stc]) can be modified, any attempt to modify ([expr.ass], [expr.post.incr], [expr.pre.incr]) a const object ([basic.type.qualifier]) during its lifetime ([basic.life]) results in undefined behavior …

值得注意的是，如果我们通过 const_case 去除了变量的 const 属性，但是并不修改变量，这种行为是 well-defined，例如下面的代码：

```
struct B
{
    int i;
    double d;
};

constexpr B bar()
{
    constexpr B b = { 10, 10.10 };
    B *p = const_cast<B *>(&b);

    int x = p->i;

    return *p;
}

void foo()
```

(continues on next page)

(continued from previous page)

```
{
    constexpr B y = bar();
}
```

https://godbolt.org/z/PaADef clang 和 gcc 都没有报错。

8.1.11 Accessing a non-active union member

```
union Y { float f; int k; };
void g() {
  constexpr Y y = { 1.0f }; // OK, y.x is active union member (10.3)
  constexpr int n = y.k;    // Line 4
}
```

https://godbolt.org/ clang 和 gcc 都报告了该 UB。

C++ 标准中与此相关的说明 [class.union]p1：

> …a non-static data member is active if its name refers to an object whose lifetime has begun and has not ended ([basic.life]). At most one of the non-static data members of an object of union type can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time …

8.1.12 Casting int to enum outside its range

C++ 标准中的相关说明：

[dcl.enum]p8：

> For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, the values of the enumeration are the values representable by a hypothetical integer type with minimal width M such that all enumerators can be represented. The width of the smallest bit-field large enough to hold all the values of the enumeration type is M. It is possible to define an enumeration that has values not defined by any of its enumerators. …

[expr.static.cast]p10：

> A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the enumeration type has a fixed underlying type, the value is first converted to that type by integral conversion, if necessary, and then to the enumeration type. If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values ([dcl.enum]), and otherwise, the behavior is undefined …

defect report 2338 应该是当前标准这么规定的原因。

在下面的例子中，enum A 需要 1 bit 来表示其所有的枚举变量，同时 enum A 没有 fixed underlying type，所以将任意需要超过 1 bit 来表示的数 cast 为 enum A 都是 UB。

```
enum A
{
    e1 = 0,
    e2
};

constexpr int foo()
{
    constexpr A a1 = static_cast<A>(4); // 4 requires 2 bit
    return a1;
}

constexpr int bar()
{
    constexpr int x = foo();
    return x;
}

int main()
{
    return bar();
}
```

查看 godbolt 的结果 https://godbolt.org/z/ZWI2xb，发现 gcc, clang, msvc 都没有报告编译错误，也就是没有捕获到该 UB。clang 通过 UBSan，能检测到与上例类似的一种情况 https://wandbox.org/permlink/s6judERNBKYIgspG，但是也不能检测到上例中的 UB。

8.1.13 Multiple unsequenced modifications

Stack Overflow 上的一个 infamous 的问题 Why are these constructs using pre and post-increment undefined behavior?

国内谭浩强的教材习题中也有类似的问题。

下面代码是该问题的简化版本，其中 x 被修改两次，而加法运算符的两个操作数的求值是无顺序的。

```
constexpr int f(int x)
{
    return x++ + x++;
}

int main()
{
```

(continues on next page)

(continued from previous page)

```
    constexpr int x = 2;
    constexpr int y = f(x);
}
```

通过 https://godbolt.org/z/Uai2P9，我们发现 clang, gcc, msvc 都没有报错。而事实上该行为是 UB。

8.1.14 One More Inconsistency, Guaranteed Copy Elision

TODO 见原文……

[class.copy.elision]p1 中指出：

> …Copy elision is not permitted where an expression is evaluated in a context requiring a constant expression ([expr.const]) and in constant initialization ([basic.start.static]). [Note: Copy elision might be performed if the same expression is evaluated in another context.— end note]

原文中举了一个 Richard Smith 分享的例子来说明：

```
struct B {B* self=this;};
extern const B b;
constexpr B f() {
    B b;                                // Line 4
    if(&b == &::b) return B();          // Line 5
    else return b;                      // Line 6
}
constexpr B b=f(); // is b.self == b   // Line 8
```

8.1.15 An Example, A Strong Integer Type

在原文的最后，作者举了一个例子说明本文提供的方法怎样被应用。该例演示了如何构造一个简单的强 int 类型，当该类在 constexpr 上下文中被使用时，它将捕获我们在使用 int 时可能遇到的所有常见的 UB。

```
struct Integer {
   constexpr Integer(int v){value = v;}
   constexpr Integer(double d){value = d;}
   constexpr Integer(const Integer&) = default;

   int Value() const {return value;}

   constexpr Integer operator+(Integer y) const {
       return {value + y.value};
   }
```

(continues on next page)

(continued from previous page)

```
    constexpr Integer operator-(Integer y) const {
        return {value - y.value};
    }

    constexpr Integer operator*(Integer y) const {
        return {value*y.value};
    }

    constexpr Integer operator/(Integer y) const {
        return {value/y.value};
    }

    constexpr Integer operator<<(Integer shift) const {
        return {value << shift.value};
    }

    constexpr Integer operator>>(Integer shift) const {
        return {value >> shift.value};
    }

    int value{};
};
```

一些本文中已讨论过的 UB 的操作：

```
constexpr Integer i_int_max{INT_MAX};
constexpr Integer i_int_max_plus_one{i_int_max+1}; // Overflow
constexpr Integer i_one{1};
constexpr Integer i_zero{0};
constexpr Integer i_divide_by_zero = i_one/i_zero;  // Divide by zero
constexpr Integer i_double_max{DBL_MAX}; // double value outside of range␣
↪representable by int
constexpr Integer i_int_min{INT_MIN};
constexpr Integer i_minus_one{-1};
constexpr Integer i_overflow_division = i_int_min/i_minus_one;  // Overflow
constexpr Integer i_shift_ub1 = i_one << 32;
constexpr Integer i_shift_ub2 = i_minus_one << 1;
constexpr Integer i_shift_ub3 = i_one << -1;
```

https://godbolt.org/z/ScpyN1 说明上述 UB 均被捕获到了。

8.1.16 Conclusion

本文中我们了解了常量表达式，并且学习到了在常量表达式上下文中是禁止未定义行为的。我们可以利用 constexpr 来捕获和探究未定义的行为，本文中已经探究了可以在常量表达式上下文中被研究的大部分未定义行为。

需要注意的是此方法依靠编译器为我们捕获未定义行为，但是编译器是有 bug 的，事实上在本文中也已经看到了一些编译器结果与标准不一致的情况，因此我们应尽可能使用多个编译器进行测试，以避免漏报和误报。

另一个需要注意的是 Guaranteed Copy Elision，*TODO*。

8.1.17 P.S.

Integral Promotions

> The implicit conversions that preserve values are commonly referred to as promotions. Before an arithmetic operation is performed, integral promotion is used to create int s out of shorter integer types. Similarly, floating-point promotion is used to create double s out of float s. Note that these promotions will not promote to long (unless the operand is a char16_t, char32_t, wchar_t, or a plain enumeration that is already larger than an int) or long double. This reflects the original purpose of these promotions in C: to bring operands to the ‘‘natural’’ size for arithmetic operations. The integral promotions are:
>
> - A char, signed char, unsigned char, short int, or unsigned short int is converted to an int if int can represent all the values of the source type; otherwise, it is converted to an unsigned int.
> - A char16_t, char32_t, wchar_t, or a plain enumeration type is converted to the first of the following types that can represent all the values of its underlying type: int, unsigned int, long, unsigned long, or unsigned long long.
> - A bit-field is converted to an int if int can represent all the values of the bit-field; otherwise, it is converted to unsigned int if unsigned int can represent all the values of the bit-field. Otherwise, no integral promotion applies to it.
> - A bool is converted to an int ; false becomes 0 and true becomes 1. Promotions are used as part of the usual arithmetic conversions.

Usual Arithmetic Conversions

> These conversions are performed on the operands of a binary operator to bring them to a common type, which is then used as the type of the result: 1. If either operand is of type long double, the other is converted to long double. - Otherwise, if either operand is double, the other is converted to double. - Otherwise, if either operand is float, the other is converted to float. - Otherwise, integral promotions (§10.5.1) are performed on both operands. 2. Otherwise, if either operand is unsigned long long, the other is converted to unsigned long long. - Otherwise, if one operand is a long long int and the other is an unsigned long int, then if a long long int can represent all the values of an unsigned long int, the unsigned long int is converted to a long

long int ; otherwise, both operands are converted to unsigned long long int. Otherwise, if either operand is unsigned long long, the other is converted to unsigned long long. - Otherwise, if one operand is a long int and the other is an unsigned int, then if a long int can represent all the values of an unsigned int, the unsigned int is converted to a long int ; otherwise, both operands are converted to unsigned long int. - Otherwise, if either operand is long, the other is converted to long. - Otherwise, if either operand is unsigned, the other is converted to unsigned. - Otherwise, both operands are int. These rules make the result of converting an unsigned integer to a signed one of possibly larger size implementation-defined. That is yet another reason to avoid mixing unsigned and signed integers.

- genindex
- modindex
- search

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search