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;
            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))
            {
                ++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
        // 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();
            }

            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);
    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)
```

```
{
    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)
                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)
            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
/// 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();
```

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);
        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:
  %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。