INTERPROCEDURAL OPTIMIZATIONS

DCC 888

Warm up Example

```
int arith(int b, int e) {
  int sum = 0;
  int counter = b;
  while (counter < e) {
    sum += counter;
    counter++;
  }
  return sum;
}

int main() {
  int N0 = 0, N1 = 10;
  int N3 = arith(N0, N1);
}
```

1) How could we optimize this program?
2) Which knowledge would we need to optimize this program?
3) How can we obtain this knowledge?

Interprocedural Analyses and Optimizations

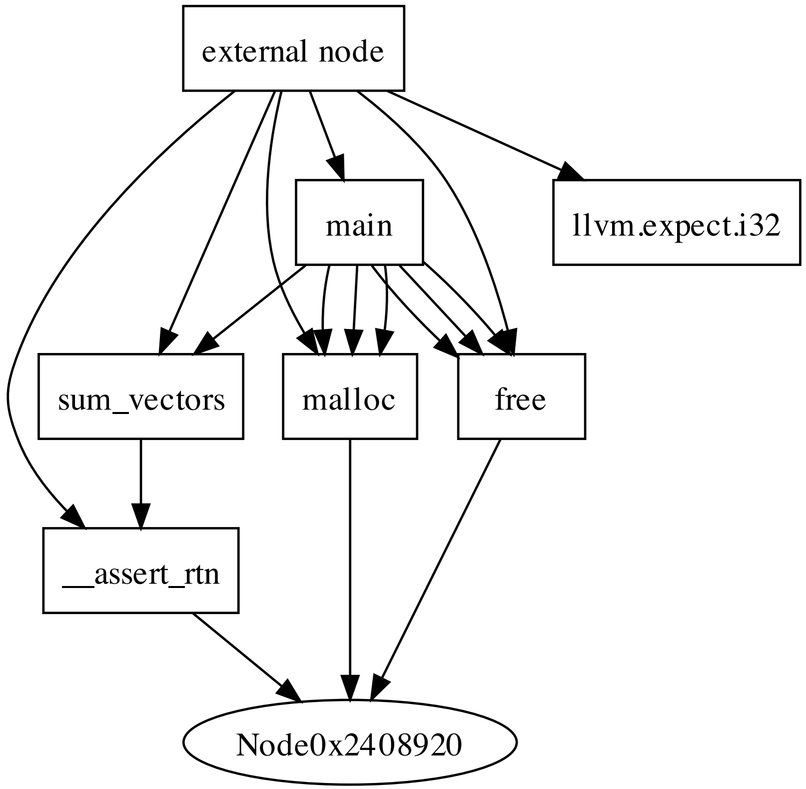
- Often a function does not contain all the information necessary to understand and optimize some aspect of it.
- Interprocedural analysis (or optimizations) go beyond the boundaries of functions.
- LLVM provides some interprocedural tools:
  - Call graphs
  - Module passes
  - Plus a vast API to deal with functions

How do we know that N is the size of src and dst in the program below?

```
int sort(int N, int* src, int* dst) {
  int i, j, k;
  for (i = 0; i < N; i++) {
    dst[i] = src[i];
  }
  for (j = 0; j < N - 1; j++) {
    for (k = j + 1; k < N; k++) {
      if (dst[j] < dst[k]) {
        int tmp = dst[j];
        dst[j] = dst[k];
        dst[k] = tmp;
      }
    }
  }
}

int main(int argc, char** argv) {
  int *a, *b;
  int S = atoi(argv[1]);
  a = (int*) malloc(sizeof(int) * S);
  b = (int*) malloc(sizeof(int) * S);
  readVec(a);
  sort(S, a, b);
  printVec(b);
}
```

Using the Call Graph

```c
void sum_vectors(int *src1, int *src2, int *dest, size_t
  unsigned i;
  assert(src1 != 0 && src2 != 0 && dest != 0);
  for (i = 0; i < n; ++i)
    dest[i] = src1[i] + src2[i];
}

#define SIZE 20

int main() {
  int *v1, *v2, *v3;
  unsigned i;
  v1 = (int*) malloc(SIZE);
  v2 = (int*) malloc(SIZE);
  v3 = (int*) malloc(SIZE);
  for (i = 0; i < SIZE; ++i) {
    v1[i] = i * 2;
    v2[i] = i * 3;
  }
  sum_vectors(v1, v2, v3, SIZE);
  free(v1);
  free(v2);
  free(v3);
  return 0;
}
```




```
$> clang -c -emit-llvm file.c -o file.bc

$> opt -view-callgraph file.bc
```

A Real-World Example

The LLVM intermediate representation provides us with the noalias attribute which we can use to mark the arguments of functions, indicating that these arguments do not alias each other. In what follows, we shall design an analysis to add this special attribute to the arguments of functions, whenever possible.

```
define void @sum_vectors(
    i32* noalias %src1,
    i32* noalias %src2,
    i32* noalias %d
    est, i32 %n) #0 {
    . . .
}
```

1) Why does LLVM provide this attribute?

2) What can the compiler do with a function if it "knows" that its arguments do not alias each other?

Pointer-Based Transformations

```
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}

void sum1(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    int tmp = a[i];
    if (!b[i]) {
      tmp = b[i];
    }
    r[i] = tmp;
  }
}
```

1) Which of these two functions is faster, sum0 or sum1?
2) Can the compiler change one into the other?
3) Which information is necessary to ensure that this modification is safe?

The Power of Information

```
#define SIZE 2000
#define LOOP 1000000
int main(int argc, char** argv) {
  int* a = (int*) malloc(SIZE * 4);
  int* b = (int*) malloc(SIZE * 4);
  int* c = (int*) malloc(SIZE * 4);
  int i;
  for (i = 0; i < SIZE; i++) {
    a[i] = i;
    b[i] = i%2;
  }
  if (argc % 2) {
    printf("sum0\n");
    for (i = 0; i < LOOP; i++)
      sum0(a, b, c, SIZE);
  } else {
    printf("sum1\n");
    for (i = 0; i < LOOP; i++)
      sum1(a, b, c, SIZE);
  }
}
```

```
$> gcc –O1 noalias.c

$> time ./a.out
sum0

real 0m12.930s
user 0m12.797s
sys  0m0.053s

$> time ./a.out a
sum1

real 0m2.710s
user 0m2.680s
sys  0m0.013s
```

Mac OS X, v10.5.8
2.26 GHz Intel Core 2 Duo
2 GB 1067 MHz DDR3

Add_No_Alias.cpp

Modifying Function Arguments

```
namespace {
 struct Add_No_Alias : public ModulePass {
   static char ID;
   Add_No_Alias() : ModulePass(ID) {}
   virtual bool runOnModule(Module &M) {
     . . .
   }
 };
}

char Add_No_Alias::ID = 0;
static RegisterPass<Add_No_Alias> X
   ("addnoalias", "Add no alias to function attributes");
```

We are defining a Module Pass; hence, we can iterate over all the functions in the program. Notice that for this problem we could also use a Function Pass.

We want to add the noalias attribute to the argument of functions. How do you think our **code** would be?

Can you guess the command line to use this pass?

Quick Look into the API

```
virtual bool runOnModule(Module &M) {
 for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F) {
  if (!F->isDeclaration()) {
   Function::arg_iterator Arg = F->arg_begin(), ArgEnd = F->arg_end();
   while (Arg != ArgEnd) {
    if (Arg->getType()->isPointerTy()) {
     AttrBuilder noalias(Attribute::get(Arg->getContext(), Attribute::NoAlias));
     int argNo = Arg->getArgNo() + 1;
     Arg->addAttr(AttributeSet::get(Arg->getContext(), argNo, noalias));
    }
    ++Arg;
   }
  }
 }
 return true;
}
```

1) How do we iterate over the arguments of a function?
2) Why do we have **this** test?
3) And why do we have **this other** test?

Running the first pass

```
$> clang -c -emit-llvm file.c -o file.bc

$> opt -load dcc888.dylib -addnoalias file.bc -o file.na.bc

$> llvm-dis < file.bc -o file.ll

$> llvm-dis < file.na.bc -o file.na.ll

$> diff file.ll file.na.ll

10c10

< define void @sum_vectors(i32* %src1 ...

---

> define void @sum_vectors(i32* noalias %src1 ...
```

1) Our pass is not exactly safe. Why?
2) What should we do to ensure that it is correct?

A More Sensible Approach

If a function $f(a_0, ..., a_n)$ has **two or more** formal parameters that are pointers, then insert it in the set of candidates.

If the program contains a call $f(p_0, ..., p_n)$ such that actual parameters p_i and p_j may alias each other, then remove f from the set of candidates.

For every function that remains in the set of candidates, add noalias to its formal parameters.

Why do we have **this criterion** to define candidates?

How do we know if the parameters alias each other?

The Interface of our Optimization

```
#ifndef CALLSITEALIAS_H_
#define CALLSITEALIAS_H_

using namespace llvm;
class Collect_Args_No_Alias: public ModulePass {
 public:
  static char ID;
  Collect_Args_No_Alias() : ModulePass(ID) {}
  ~Collect_Args_No_Alias() {}
  virtual bool runOnModule(Module &M);
  virtual void getAnalysisUsage(AnalysisUsage &AU) const;
 private:
  AliasAnalysis* AA;
  bool argsMayAlias(const CallInst* CI) const;
  bool isCandidate(const CallInst* CI) const;
  void addNoAlias(Function* F) const;
};

#endif
```

1) What is the role of the ifndef/define/endif tags?
2) Why do we mark methods with the **const** modifier?
3) What do you think this object does?

Alias Analysis

- Alias analysis is a static analysis that tries to determine, for each pair of pointers, if these pointers may point to the same memory region or not.

```
bool Collect_Args_No_Alias::argsMayAlias(const CallInst* CI) const {
 unsigned n_operands = CI->getNumArgOperands();
 bool mayAlias = false;
 for (unsigned i = 0; i < n_operands - 1; ++i) {
  const Value *pi = CI->getArgOperand(i);
  for (unsigned j = i + 1; j < n_operands; ++j) {
   const Value *pj = CI->getArgOperand(j);
   if (AA->alias(pi, pj) != AliasAnalysis::NoAlias) {
    mayAlias = true;
   }
  }
 }
 return mayAlias;
}
```

LLVM provides a number of alias analyses, which we can use to disambiguate pointers. We invoke the alias analysis as a pass, like in the example below:

```
bool Collect_Args_No_Alias::runOnModule(Module &M) {
 AA = &getAnalysis<AliasAnalysis>();
 . . .
}
```

The Alias Analysis Interface

```
const Value *pi = ...;
const Value *pj = ...;
switch(AA.alias(pi, pj)) {
  case AliasAnalysis::NoAlias:
    errs() << " do not alias.\n";
    break;
  case AliasAnalysis::MustAlias:
    errs() << " must alias.\n";
    break;
  case AliasAnalysis::PartialAlias:
    errs() << " alias partially.\n";
    break;
  case AliasAnalysis::MayAlias:
    errs() << " may alias.\n";
    break;
}
```

The alias analysis framework recognizes different ways through which variables can be related. The four classifications are shown in the program on the left.

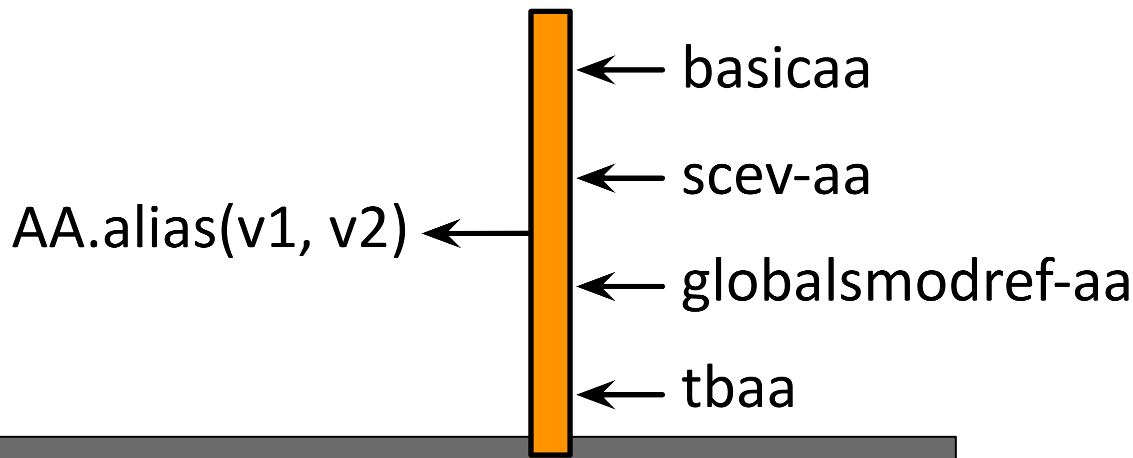
1) What do you think is the meaning of each one of **these** qualifiers?

2) How do you think the alias analysis pass determines the classification of each pair of pointers?

Invoking Alias Analysis

Alias analyses are implemented as an analysis group. An analysis group is a set of LLVM passes that use the same interface. In our example, we are interested in discovering if two values alias, e.g., AA.alias(v1, v2). The simplest way to answer this query is to say "may" for everything. Yet, we can use much fancier algorithms, trading time for extra precision.

The alias analysis interface




```
$> opt -basicaa -scev-aa -globalsmodref-aa -tbaa ...
```

Indicating the Intention to Use

We must indicate explicitly our intention to use the AliasAnalysis. We do this in the getAnalysisUsage method, which every LLVM pass can overwrite. In this way, we can get a pointer to the AliasAnalysis object, which contains the results of LLVM's alias analyses.

```
void Collect_Args_No_Alias::getAnalysisUsage(AnalysisUsage &AU) const {
  // AU.setPreservesAll();
  AU.addRequired<AliasAnalysis>();
}

bool Collect_Args_No_Alias::runOnModule(Module &M) {
  AA = &getAnalysis<AliasAnalysis>();
  …
}
```

Why we cannot say that our pass preserves the other analyses?

```
$> opt -basicaa -scev-aa -globalsmodref-aa -tbaa ...
```

Searching Promising Calls

```
bool Collect_Args_No_Alias::runOnModule(Module &M) {
  AA = &getAnalysis<AliasAnalysis>();

  // 1) Find promising calls

  // 2) Add the noalias tags whenever possible

  return ...;
}
```

1) How do you think we should implement our algorithm?

2) What should we return from our runOnModule method?

Remember the problem: we want to design an analysis to add the noalias attribute to the arguments of functions, whenever possible.

1) Finding Promising Calls

```
SmallPtrSet<const Function*, 32> candidateCalls;
SmallPtrSet<const Function*, 32> mayAliasCalls;
// Check all the calls in the program.
for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F) {
 if (!F->isDeclaration()) {
  for (inst_iterator I = inst_begin(&*F), E = inst_end(&*F); I != E; ++I) {
   if (const CallInst *CI = dyn_cast<CallInst>(&*I)) {
    if (isCandidate(CI)) {
     candidateCalls.insert(CI->getCalledFunction());
     if (argsMayAlias(CI)) {
      mayAliasCalls.insert(CI->getCalledFunction());
     }
    }
   }
  }
 }
}
```

Why do we have to guard against **declarations**?

What is this iterator giving us?

We need to implement two new methods: **isCandidate**, and **argsMayAlias**. We shall do it in a while, but before, we must implement the rest of the runOnModule method.

2) Add the noalias Tag Whenever Possible

```
bool wasModified = false;
for (Module::iterator I = M.begin(), E = M.end(); I != E; ++I) {
  if (!I->isDeclaration()) {
    if (candidateCalls.count(I) > 0 && mayAliasCalls.count(I) == 0) {
      addNoAlias(I);
      wasModified = true;
    }
  }
}
return wasModified;
```

What is the semantics of the value returned by runOnModule? When should it return true or false?

```
bool Collect_Args_No_Alias::runOnModule(Module &M) {
  AA = &getAnalysis<AliasAnalysis>();
  SmallPtrSet<const Function*, 32> candidateCalls;
  SmallPtrSet<const Function*, 32> mayAliasCalls;
  for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F) {
    if (!F->isDeclaration()) {
      for (inst_iterator I = inst_begin(&*F), E = inst_end(&*F); I != E; ++I) {
        if (const CallInst *CI = dyn_cast<CallInst>(&*I)) {
          if (isCandidate(CI)) {
            candidateCalls.insert(CI->getCalledFunction());
            if (argsMayAlias(CI)) {
              mayAliasCalls.insert(CI->getCalledFunction());
            }
          }
        }
      }
    }
  }
  bool wasModified = false;
  for (Module::iterator I = M.begin(), E = M.end(); I != E; ++I) {
    if (!I->isDeclaration()) {
      if (candidateCalls.count(I) > 0 && mayAliasCalls.count(I) == 0) {
        addNoAlias(I);
        wasModified = true;
      }
    }
  }
  return wasModified;
}
```

The whole runOnModule method

Back to the Search of Promising Calls

```
SmallPtrSet<const Function*, 32> candidateCalls;
SmallPtrSet<const Function*, 32> mayAliasCalls;
// Check all the calls in the program.
for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F) {
 if (!F->isDeclaration()) {
  for (inst_iterator I = inst_begin(&*F), E = inst_end(&*F); I != E; ++I) {
   if (const CallInst *CI = dyn_cast<CallInst>(&*I)) {
    if (isCandidate(CI)) {
     candidateCalls.insert(CI->getCalledFunction());
     if (argsMayAlias(CI)) {
      mayAliasCalls.insert(CI->getCalledFunction());
     }
    }
   }
  }
 }
}
```

How would you implement these two functions?

We say that a function is a candidate to be optimized whenever it **(i)** has two or more parameters which are pointers, and **(ii)** none of these parameters alias each other in any invocation of the function throughout the program code. We need to implement two functions now: **isCandidate** and **argsMayAlias**.

Looking for Good Candidates

```
bool Collect_Args_No_Alias::isCandidate(const CallInst* CI) const {
  unsigned n_operands = CI->getNumArgOperands();
  unsigned numPointerArgs = 0;
  for (unsigned i = 0; i < n_operands; ++i) {
    if (CI->getArgOperand(i)->getType()->isPointerTy()) {
      numPointerArgs++;
    }
  }
  return numPointerArgs > 1;
}
```

Now, to finish everything, we just need to implement a function that add noalias to all the pointer arguments of functions. How to do it?

1) Can you write a header comment explaining what isCandidate does?

2) An unrelated, yet cool question: imagine that we had this loop instead – *for (i = 0; i < CI->getNumArgOperands(); ++i)* – do you think the compiler would move the call to outside the loop? Does it do it always?

Adding the noalias Tag to Arguments

```
void Collect_Args_No_Alias::addNoAlias(Function* F) const {
  Function::arg_iterator Arg, ArgEnd;
  for (Arg= F->arg_begin(), ArgEnd = F->arg_end(); Arg != ArgEnd; ++Arg) {
    if (Arg->getType()->isPointerTy()) {
      AttrBuilder noalias(Attribute::get(Arg->getContext(), Attribute::NoAlias));
      int argNo = Arg->getArgNo() + 1;
      Arg->addAttr(AttributeSet::get(Arg->getContext(), argNo, noalias));
    }
  }
}
```

We had seen a code a bit like this one[§], so it does not require much explanation. Notice that this function is changing the program, e.g., it changes the declaration of functions. We must remember to indicate, through the return value of the runOnModule method, that the program has been modified.

[§]: See slide 9

Looking into an example

```
void Fft (int n, float z[], float w[], float e[]) {
 int i, j, k, l, m, index;
 m = n / 2;
 l = 1;
 do {
  k = 0;
  j = l;
  i = 1;
  do {
   do {
    w[i + k] = z[i] + z[m + i];
    w[i + j] = e[k + 1] * (z[i] - z[i + m])
     - e[k + 1] * (z[i] - z[i + m]);
    w[i + j] = e[k + 1] * (z[i] - z[i + m])
     + e[k + 1] * (z[i] - z[i + m]);
    i = i + 1;
   } while (i <= j);
   k = j;
   j = k + l;
  } while (j <= m);
  l++;
 } while (l <= m);
}
```

Jean-Baptiste-Joseph Fourier
(1768-1830)

```
float* genVec(unsigned n) {
 float* f = (float*)malloc(n * sizeof(float));
 int i;
 for (i = 0; i < n; i++) {
  f[i] = 2.5 + i;
 }
 return f;
}

int main(int argc, char** argv) {
 int n = atoi(argv[1]);
 float* z = genVec(n);
 float* w = genVec(n);
 float* e = genVec(n);
 Fft(n, z, w, e);
 return 0;
}
```

The function Fft, on the right, is an adaptation of the classic Fast Fourier Transform, typically seen in high-performance computing.

Running the Example

```
$> clang -c -emit-llvm fft.c -o file.bc
$> opt -mem2reg -instnamer file.bc -o file.rbc
$> opt -load dcc888.dylib -pointerdis file.rbc -o file.na.rbc
$> llvm-dis < file.rbc -o file.ll
$> llvm-dis < file.na.rbc -o file.na.ll
$> diff file.ll file.na.ll
6c6
< define void @Fft(i32 %n, float* %z, ...
---
> define void @Fft(i32 %n, float* noalias %z, ...
$> clang -c -emit-llvm file.ll -o file.rbc
$> clang -c -emit-llvm file.na.ll -o file.na.rbc
$> opt -O2 file.rbc -o file.opt.rbc
$> opt -O2 file.na.rbc -o file.opt.na.rbc
$> llc file.opt.rbc -o file.opt.s
$> gcc file.opt.s -o file.opt.exe
$> time ./file.opt.exe 30000
real    0m1.054s
user    0m1.039s
sys     0m0.006s
$> llc file.opt.na.rbc -o file.opt.na.s
$> gcc file.opt.na.s -o file.opt.na.exe
$> time ./file.opt.na.exe 30000
real    0m0.680s
user    0m0.668s
sys     0m0.005s
```

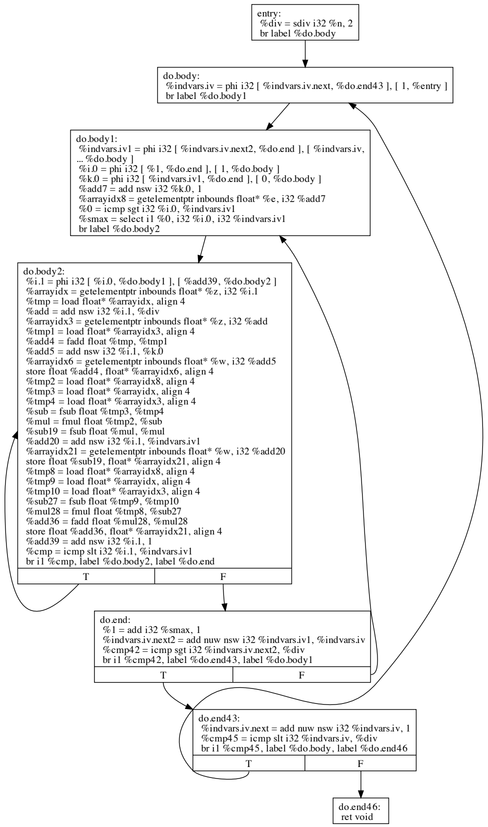
This experiment was performed on a Mac OS X, v10.5.8 2.26 GHz Intel Core 2 Duo 2 GB 1067 MHz DDR3. As we can see, the optimized program is about 36% faster than the original program.

Look at the program again. Which reasons do you think could explain this huge speedup?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC-UFMG

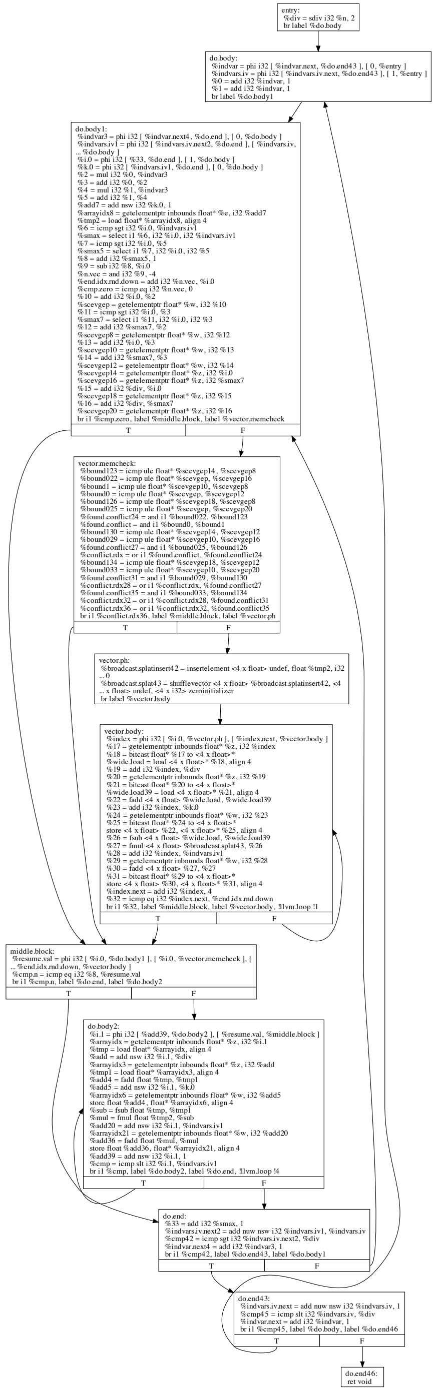
Looking Into the Example

In this example, unrolling is the hero: the noalias tag lets LLVM unroll the most internal loop of the example. Unrolling, by itself, has enabled more aggressive optimizations inside basic blocks as well. The optimized code, on the right, is a bit larger, but it is much faster than the original code, on the left.



CFG for 'Fft' function

Optimization



CFG for 'Fft' function

Which kind of optimizations do you think we can get by telling the compiler that arrays do not alias each other?

Final Remarks

- LLVM provides different categories of passes:
  - Basic block passes; Loop passes; Function passes; Module passes; etc
- Module Passes let us see the program as a whole.
  - This view enables much more extensive analyses and optimizations.
- To find out more about interprocedural optimizations, you can take a look into the LLVM sources, e.g.:

```
$> llvm/lib/Transforms/IPO$ ls
ArgumentPromotion.cpp          InlineSimple.cpp
BarrierNoopPass.cpp            Inliner.cpp
CMakeLists.txt                 Internalize.cpp
ConstantMerge.cpp              LLVMBuild.txt
DeadArgumentElimination.cpp    LoopExtractor.cpp
Debug+Asserts                  Makefile
ExtractGV.cpp                  MergeFunctions.cpp
FunctionAttrs.cpp              PartialInlining.cpp
GlobalDCE.cpp                  PassManagerBuilder.cpp
GlobalOpt.cpp                  PruneEH.cpp
IPConstantPropagation.cpp      StripDeadPrototypes.cpp
IPO.cpp                        StripSymbols.cpp
InlineAlways.cpp
```