```cpp
//===-- SimpleArgumentPromotion.cpp - Promote by-reference arguments -------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file was developed by the LLVM research group and is distributed under
// the University of Illinois Open Source License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This pass promotes "by reference" arguments to be "by value" arguments.  In
// practice, this means looking for internal functions that have pointer
// arguments.  If we can prove, through the use of alias analysis, that an
// argument is *only* loaded, then we can pass the value into the function
// instead of the address of the value.  This can cause recursive simplification
// of code and lead to the elimination of allocas (especially in C++ template
// code like the STL).
//
// This pass is a simplified version of the LLVM argpromotion pass (it
// invalidates alias analysis instead of updating it, and can not promote
// pointers to aggregates).
//
//===----------------------------------------------------------------------===//

#include "llvm/CallGraphSCCPass.h"
#include "llvm/DerivedTypes.h"
#include "llvm/Instructions.h"
#include "llvm/Module.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/CallGraph.h"
#include "llvm/Target/TargetData.h"
#include "llvm/Support/CallSite.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/Debug.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/Statistic.h"
#include <set>
using namespace llvm;

namespace {
  Statistic<> NumArgumentsPromoted("simpleargpromotion",
                                   "Number of pointer arguments promoted");
  Statistic<> NumArgumentsDead("simpleargpromotion",
                               "Number of dead pointer args eliminated");

  /// SimpleArgPromotion - Convert 'by reference' arguments to 'by value'.
  ///
  struct SimpleArgPromotion : public CallGraphSCCPass {
    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
      AU.addRequired<AliasAnalysis>();
      AU.addRequired<TargetData>();
      CallGraphSCCPass::getAnalysisUsage(AU);
    }

    virtual bool runOnSCC(const std::vector<CallGraphNode*> &SCC);
  private:
    bool PromoteArguments(CallGraphNode *CGN);
    bool isSafeToPromoteArgument(Argument *Arg) const;
    Function *DoPromotion(Function *F, std::vector<Argument*> &ArgsToPromote);
  };

  RegisterOpt<SimpleArgPromotion> X("simpleargpromotion",
                              "Promote 'by reference' arguments to 'by value'");
}

bool SimpleArgPromotion::runOnSCC(const std::vector<CallGraphNode*> &SCC) {
  bool Changed = false, LocalChange;

  do {  // Iterate until we stop promoting from this SCC.
    LocalChange = false;
    // Attempt to promote arguments from all functions in this SCC.
    for (unsigned i = 0, e = SCC.size(); i != e; ++i)
      LocalChange |= PromoteArguments(SCC[i]);
    Changed |= LocalChange;               // Remember that we changed something.
```

```cpp
  } while (LocalChange);

  return Changed;
}

/// PromoteArguments - This method checks the specified function to see if there
/// are any promotable arguments and if it is safe to promote the function (for
/// example, all callers are direct).  If safe to promote some arguments, it
/// calls the DoPromotion method.
///
bool SimpleArgPromotion::PromoteArguments(CallGraphNode *CGN) {
  Function *F = CGN->getFunction();

  // Make sure that it is local to this module.
  if (!F || !F->hasInternalLinkage()) return false;

  // First check: see if there are any pointer arguments!  If not, quick exit.
  std::vector<Argument*> PointerArgs;
  for (Function::aiterator I = F->abegin(), E = F->aend(); I != E; ++I)
    if (isa<PointerType>(I->getType()))
      PointerArgs.push_back(I);
  if (PointerArgs.empty()) return false;

  // Second check: make sure that all callers are direct callers.  We can't
  // transform functions that have indirect callers.
  for (Value::use_iterator UI = F->use_begin(), E = F->use_end();
       UI != E; ++UI) {
    CallSite CS = CallSite::get(*UI);
    if (!CS.getInstruction())       // "Taking the address" of the function
      return false;

    // Ensure that this call site is CALLING the function, not passing it as
    // an argument.
    for (CallSite::arg_iterator AI = CS.arg_begin(), E = CS.arg_end();
         AI != E; ++AI)
      if (*AI == F) return false;   // Passing the function address in!
  }

  // Check to see which arguments are promotable.  If an argument is not
  // promotable, remove it from the PointerArgs vector.
  for (unsigned i = 0; i != PointerArgs.size(); ++i)
    if (!isSafeToPromoteArgument(PointerArgs[i])) {
      std::swap(PointerArgs[i--], PointerArgs.back());
      PointerArgs.pop_back();
    }

  // No promotable pointer arguments.
  if (PointerArgs.empty()) return false;

  // Okay, promote all of the arguments are rewrite the callees!
  Function *NewF = DoPromotion(F, PointerArgs);

  // Update the call graph to know that the old function is gone.
  getAnalysis<CallGraph>().changeFunction(F, NewF);
  return true;
}


/// isSafeToPromoteArgument - As you might guess from the name of this method,
/// it checks to see if it is both safe and useful to promote the argument.
bool SimpleArgPromotion::isSafeToPromoteArgument(Argument *Arg) const {
  // We can only promote this argument if all of the uses are loads.
  std::vector<LoadInst*> Loads;

  for (Value::use_iterator UI = Arg->use_begin(), E = Arg->use_end();
       UI != E; ++UI)
    if (LoadInst *LI = dyn_cast<LoadInst>(*UI)) {
      if (LI->isVolatile()) return false;        // Don't modify volatile loads.
      Loads.push_back(LI);
    } else {
      return false;  // Not a load.
    }

  // Okay, now we know that the argument is only used by load instructions.  Use
  // alias analysis to check to see if the pointer is guaranteed to not be
  // modified from entry of the function to each of the load instructions.
  Function &F = *Arg->getParent();

  // Because there could be several/many load instructions, remember which
  // blocks we know to be transparent to the load.
  std::set<BasicBlock*> TranspBlocks;

  AliasAnalysis &AA = getAnalysis<AliasAnalysis>();
  TargetData &TD = getAnalysis<TargetData>();

  for (unsigned i = 0, e = Loads.size(); i != e; ++i) {
    // Check to see if the load is invalidated from the start of the block to
    // the load itself.
    LoadInst *Load = Loads[i];
    BasicBlock *BB = Load->getParent();

    const PointerType *LoadTy =
      cast<PointerType>(Load->getOperand(0)->getType());
    unsigned LoadSize = TD.getTypeSize(LoadTy->getElementType());

    if (AA.canInstructionRangeModify(BB->front(), *Load, Arg, LoadSize))
      return false;  // Pointer is invalidated!

    // Now check every path from the entry block to the load for transparency.
    // To do this, we perform a depth first search on the inverse CFG from the
    // loading block.
    for (pred_iterator PI = pred_begin(BB), E = pred_end(BB); PI != E; ++PI)
      for (idf_ext_iterator<BasicBlock*> I = idf_ext_begin(*PI, TranspBlocks),
             E = idf_ext_end(*PI, TranspBlocks); I != E; ++I)
        if (AA.canBasicBlockModify(**I, Arg, LoadSize))
          return false;
  }

  // If the path from the entry of the function to each load is free of
  // instructions that potentially invalidate the load, we can make the
  // transformation!
  return true;
}

/// DoPromotion - This method actually performs the promotion of the specified
/// arguments, and returns the new function.  At this point, we know that it’s
/// safe to do so.
Function *SimpleArgPromotion::DoPromotion(Function *F,
                                          std::vector<Argument*> &Args2Prom) {
  std::set<Argument*> ArgsToPromote(Args2Prom.begin(), Args2Prom.end());

  // Start by computing a new prototype for the function, which is the same as
  // the old function, but has modified arguments.
  const FunctionType *FTy = F->getFunctionType();
  std::vector<const Type*> Params;

  for (Function::aiterator I = F->abegin(), E = F->aend(); I != E; ++I)
    if (!ArgsToPromote.count(I)) {
      Params.push_back(I->getType());
    } else if (I->use_empty()) {
      ++NumArgumentsDead;
    } else {
      // Add a parameter to the function for each element passed in.
      Params.push_back(cast<PointerType>(I->getType())->getElementType());
      ++NumArgumentsPromoted;
    }

  // Create the new function body and insert it into the module.
  FunctionType *NFTy = FunctionType::get(FTy->getReturnType(), Params,
                                         FTy->isVarArg());
  Function *NF = new Function(NFTy, F->getLinkage(), F->getName());
  F->getParent()->getFunctionList().insert(F, NF);

  // Loop over all of the callers of the function, transforming the call sites
  // to pass in the loaded pointers.
  //
```

```
  std::vector<Value*> Args;
  while (!F->use_empty()) {
    CallSite CS = CallSite::get(F->use_back());
    Instruction *Call = CS.getInstruction();

    // Loop over the operands, inserting the loads in the caller as needed.
    CallSite::arg_iterator AI = CS.arg_begin();
    for (Function::aiterator I = F->abegin(), E = F->aend(); I != E; ++I, ++AI)
      if (!ArgsToPromote.count(I))      // Unmodified argument.
        Args.push_back(*AI);
      else if (!I->use_empty())         // Non-dead argument: insert the load.
        Args.push_back(new LoadInst(*AI, (*AI)->getName()+".val", Call));

    // Push any varargs arguments on the list
    for (; AI != CS.arg_end(); ++AI)
      Args.push_back(*AI);

    Instruction *New;  // Create the new call or invoke instruction.
    if (InvokeInst *II = dyn_cast<InvokeInst>(Call)) {
      New = new InvokeInst(NF, II->getNormalDest(), II->getUnwindDest(),
                           Args, "", Call);
    } else {
      New = new CallInst(NF, Args, "", Call);
    }
    Args.clear();

    if (!Call->use_empty()) {
      Call->replaceAllUsesWith(New);
      New->setName(Call->getName());
    }

    // Finally, remove the old call from the program, reducing the use-count of
    // F.
    Call->getParent()->getInstList().erase(Call);
  }

  // Since we have now created the new function, splice the body of the old
  // function right into the new function, leaving the old rotting hulk of the
  // function empty.
  NF->getBasicBlockList().splice(NF->begin(), F->getBasicBlockList());

  // Loop over the argument list, transfering uses of the old arguments over to
  // the new arguments, also transfering over the names as well.
  //
  for (Function::aiterator I = F->abegin(), E = F->aend(), I2 = NF->abegin();
       I != E; ++I, ++I2)
    if (!ArgsToPromote.count(I)) {
      // If this is an unmodified argument, move the name and users over to the
      // new version.
      I->replaceAllUsesWith(I2);
      I2->setName(I->getName());
    } else if (!I->use_empty()) {
      // Otherwise, if we promoted this argument, then all users are load
      // instructions, and all loads should be using the new argument that we
      // added.
      while (!I->use_empty()) {
        LoadInst *LI = cast<LoadInst>(I->use_back());
        I2->setName(I->getName()+".val");
        LI->replaceAllUsesWith(I2);
        LI->getParent()->getInstList().erase(LI);
        DEBUG(std::cerr << "*** Promoted load of argument ’" << I->getName()
                        << "’ in function ’" << F->getName() << "’\n");
      }
    }

  // Now that the old function is dead, delete it.
  F->getParent()->getFunctionList().erase(F);
  return NF;
}
```

```
/************************ Loading the pass into 'opt' **************************

$ opt -load ~/llvm/lib/Debug/libsimpleargpromote.so -help
OVERVIEW: llvm .bc -> .bc modular optimizer

USAGE: opt [options] <input bytecode>

OPTIONS:
  Optimizations available:
...
    -sccp                  - Sparse Conditional Constant Propagation
    -simpleargpromotion    - Promote 'by reference' arguments to 'by value'
    -simplifycfg           - Simplify the CFG
...
  -load=<pluginfilename>   - Load the specified plugin
...
  -stats                   - Enable statistics output from program
...


************************ Simple LLVM Example ***********************************

--------- basictest.ll ---------
internal int %test(int *%X, int* %Y) {
        %A = load int* %X
        %B = load int* %Y
        %C = add int %A, %B
        ret int %C
}

internal int %caller(int* %B) {
        %A = alloca int
        store int 1, int* %A
        %C = call int %test(int* %A, int* %B)
        ret int %C
}

int %callercaller() {
        %B = alloca int
        store int 2, int* %B
        %X = call int %caller(int* %B)
        ret int %X
}
--------- basictest.ll ---------

*********************** Run with simpleargpromotion ****************************

$ llvm-as < basictest.ll | opt -load ~/llvm/lib/Debug/libsimpleargpromote.so \
                               -simpleargpromotion -stats | llvm-dis

===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

248 bytecodewriter     - Number of bytecode bytes written
  3 simpleargpromotion - Number of pointer arguments promoted


internal int %test(int %Y.val, int) {
        %C = add int %Y.val, %Y.val
        ret int %C
}

internal int %caller(int %B.val1) {
        %A = alloca int
        store int 1, int* %A
        %A.val = load int* %A
        %C1 = call int %test( int %A.val, int %B.val1 )
        ret int %C1
}

int %callercaller() {
        %B = alloca int
```

```
        store int 2, int* %B
        %B.val = load int* %B
        %X1 = call int %caller( int %B.val )
        ret int %X1
}

*********************** Run with simpleargpromotion & mem2reg ******************

$ llvm-as < basictest.ll | opt -load ~/llvm/lib/Debug/libsimpleargpromote.so \
                               -simpleargpromotion -mem2reg -stats | llvm-dis

===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

194 bytecodewriter     - Number of bytecode bytes written
  2 mem2reg            - Number of alloca's promoted
  3 simpleargpromotion - Number of pointer arguments promoted

internal int %test(int %Y.val, int) {
        %C = add int %Y.val, %Y.val
        ret int %C
}

internal int %caller(int %B.val1) {
        %C1 = call int %test( int 1, int %B.val1 )
        ret int %C1
}

int %callercaller() {
        %X1 = call int %caller( int 2 )
        ret int %X1
}

****************************** Simple C++ Example ******************************

void test(std::vector<int> &V) {
  V.push_back(7);
}

... compiles to this LLVM code:

void %_Z4testRSt6vectorIiSaIiEE("std::vector<int>"* %V) {
        %mem_tmp = alloca int
        store int 7, int* %mem_tmp
        call void %_ZNSt6vectorIiSaIiEE9push_backERKi("std::vector<int>"* %V,
                                                      int* %mem_tmp)
        ret void
}

... arg promotion and mem2reg result in this, eliminating the stack allocation
and simplifying the code.

void %_Z4testRSt6vectorIiSaIiEE("std::vector<int>"* %V) {
        call void %_ZNSt6vectorIiSaIiEE9push_backERKi("std::vector<int>"* %V,
                                                      int 7)
        ret void
}

*/
```