SimpleArgumentPromotion.cpp

```cpp
#include "llvm/CallGraphSCCPass.h"
#include "llvm/DerivedTypes.h"
#include "llvm/Module.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/CallGraph.h"
#include "llvm/Target/TargetData.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/Statistic.h"
#include <set>

namespace (llvm) {

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

//===−− SimpleArgumentPromotion.cpp − Promote by-reference arguments −−−−−−===//

// Support for transforming functions from 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()) continue;

    // Check to see which arguments are promotable.  If an argument is not
    // promotable, remove it from the PointerArgs vector.
    for (CallSite::arg_iterator AI = CS.arg_begin(), E = CS.arg_end(); AI != E; ++AI) {
        if (!isSafeToPromoteArgument(PointerArgs[ai])) {
            PointerArgs.pop_back();
            Changed |= LocalChange;
        }
    }

    // Attempt to promote arguments from all functions in this SCC.
    for (unsigned i = 0; i < SCC.size(); i += 1) {
        LocalChange |= PromoteArguments(SCC[i]);
    }
    Changed |= LocalChange;
    // Remember that we changed something.
}

bool SimpleArgPromotion::runOnSCC(const std::vector<CallGraphNode> &SCC) {
    bool Changed = false, LocalChange;
    do {
        // Attempt to promote arguments from all functions in this SCC.
        for (unsigned i = 0; i < SCC.size(); i += 1) {
            LocalChange |= PromoteArguments(SCC[i]);
        }
        Changed |= LocalChange;
        // Remember that we changed something.
    } while (LocalChange);

    return Changed;
}

// This pass promotes "by reference" arguments to "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 allocs (especially in C++ template
// code like the STL).

// This pass is a simplified version of the LLVM argrpromotion pass (it
// invalidates alias analysis instead of updating it, and can not promote
// pointers to aggregates).
retry {
    while (LocalChange);
    return false;
}

bool SimpleArgPromotion::isSafeToPromoteArgument(Argument *Arg) {
    if (Arg->isVolatile()) return false;

    // First check: see if there are any pointer arguments!  If not, quick exit.
    if (!value::isArgumentOrAddressesArg(ARG)) return false;

    for (Function::iterator F = F->begin(), E = F->end(); F != E; ++F) {
        if (isa<PointerType>(F->getReturnType()))
            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()) continue;

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

        // Check to see which arguments are promotable.  If an argument is not
        // promotable, remove it from the PointerArgs vector.
        if (value::isArgumentOrAddressesArg(ARG))
            return false;

        for (unsigned i = 0; i < SCC.size(); i += 1) {
            if (!isSafeToPromoteArgument(PointerArgs[i]))
                PointerArgs.pop_back();
            Changed |= LocalChange;
            // Remember that we changed something.
        }
    }

    // We can only promote this argument if all of the uses are loads.
    for (Value::use_iterator UI = Arg->use_begin(), E = Arg->use_end();
        UI != E || !UI->isVolatile()) return false;

    // Not a load.
    return true;
}

bool SimpleArgPromotion::isSafeToPromoteArgument(Argument *Arg) {
    if (Arg->empty()) return false;

    // : OKay, promote all of the arguments are rewrite the callees!
    for (Function *F = DoPromotion(F, PointerArgs);
        !F->empty();
        F = DoPromotion(F, PointerArgs);)
        return true;
}

// SimpleArgPromotion : public CallGraphSCCPass {
virtual void getAnalysisUsage(AnalysisUsage &AU) const {
    AU.addRequired<AliasAnalysis>();
    AU.addRequired<CallGraph>();
    AU.addRequired<CallGraphSCCPass>();
}

virtual bool runOnSCC(const std::vector<CallGraphNode> &SCC) {
    bool Changed = false, LocalChange;
    do {
        // Attempt to promote arguments from all functions in this SCC.
        for (unsigned i = 0; i < SCC.size(); i += 1) {
            LocalChange |= PromoteArguments(SCC[i]);
        }
        Changed |= LocalChange;
        // Remember that we changed something.
    } while (LocalChange);

    return Changed;
}
```
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. 

```
std::vector<Value*> Args;
```

// 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 *IntPtr = cast<PointerType>(I->getType())

```
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 *IntPtr = cast<PointerType>(I->getType());
    for (pred_iterator PI = pred_begin(BB), E = pred_end(BB); PI != E; ++PI)
        if (AA.canInstructionRangeModify(BB->front(), *Load, Arg, LoadSize))
            return false; // Pointer is invalidated!
```

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.

```
std::vector<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<Function*> Params;
```

for (Function::iterator I = F->begin(), E = F->end(); I != E; ++I)
    if (!I->use_empty())
        ++NumArgumentsDead;
    else if (I->isVarArg())
        Params.push_back(cast<PointerType>(I->getType())->getElementType());
    else if (!I->use_empty())
        Params.push_back(cast<PointerType>(I->getType()));
    ++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());
```

```
while (!I->use_empty()) {
    LoadInst *LI = cast<LoadInst>(I->load());
    I2->setName(I->getName()+".*val");
    LI->getParent()->getInstList().erase(LI);
    DEBUG(std::cerr << "*** Promoted load of argument " << I->getName() << ", in function " << F->getName() << ", at " << LI->getName() << " in " << LI->getParent()->getName() << "")
    while (!I->use_empty()) {
        LoadInst *LI = cast<LoadInst>(I->load());
        I2->setName(I->getName()+".*val");
        LI->getParent()->getInstList().erase(LI);
    }
```

```
// If this is an unmodified argument, move the name and users over to the new version.
```

```
NF->getBasicBlockList().splice(NF->begin(), F->getBasicBlockList());
```

```
// Loop over the argument list, transferring uses of the old arguments over to the new arguments, also transferring over the names as well.
```

```
for (Function::iterator I = F->begin(), E = F->end(); I != E; ++I)
    if (!I->use_empty())
        I->replaceAllUsesWith(I2);
```

```
return true; // Transformation!
```

```
while (!F->use_empty()) {
    CallSite CS = CallSite::getCallee(argc);  
    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::iterator I = F->begin(), E = F->end(); I != E; ++I)
```

```
if ((ArgsToPromote.count(I)) // Unmodified argument.
    else if (!I->use_empty()) // Non-dead argument: insert the load.
```

```
new CallInst(NF, Args, ", Call);
```

```
// Push any varargs arguments on the list
```

```
for (; AI != CS.arg_end(); ++AI)
```

```
(new LoadInst(*AI, (*AI)->getName(), Call);
```

```
// Push any varargs arguments on the list
```

```
(new LoadInst(*AI, (*AI)->getName()+".*val", Call));
```

```
// Finally, remove the old call from the program, reducing the use-count of 
```

```
F;
```

```
return true; // Transformation!
```

```
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.
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Loading the pass into ‘opt’ ...

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

-mem2reg            - Number of pointer arguments promoted
-stats                   - Enable statistics output from program

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

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

... Statistics Collected ...

248 bytecode writer     - Number of bytecode bytes written
3 simpleargpromotion - Number of pointer arguments promoted
3 internal int %test(int *%Y.val, int) {
%C = add int %Y.val, %Y.val
ret int %C
}

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
}

... 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.

194 bytecode writer     - Number of bytecode bytes written
2 mem2reg - Number of alloca’s promoted
3 simpleargpromotion - Number of pointer arguments promoted
3 internal int %test(int *%Y.val, int) {
%C = add int %Y.val, %Y.val
ret int %C
}

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
}

... Statistics Collected ...

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

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

... compiles to this LLVM code:

```cpp
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.

```cpp
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
}
```

...