An Implementation of Sparse Conditional Constant Propagation for Machine SUIF

Laurent Rolaz

Swiss Federal Institute of Technology
Processor Architecture Laboratory
Lausanne
28th March 2003

1 Introduction

The goal of constant propagation is to discover values that are constants on all possible executions of a program and to propagate these constants values as far through the program as possible. Expressions whose operands are all constants can be evaluated at compile time and the result propagated further. Constant propagation serves several purposes in compilers:

- Constant expressions evaluated at compile time need not be evaluated at execution time. If such expression are inside loops, it can save many evaluations at execution time.
- Unreachable code can be discovered by identifying conditional branches that always take one of the possible branch paths. Code that is never executed can be deleted.

Sparse Conditional Constant Propagation (SCCP) is an algorithm developed by Wegman and Zadeck [1]. This variant of constant propagation is a powerful algorithm for determining constants in a single procedure. It can discovers all constants that can be found by evaluating all conditional branches with all constant operands. SCCP operates on particular form of the intermediate representation called the *static single assignment* form.

Section 2 will describe the SSA form. The section 3 will be a detailed description based on the implementation of the SCCP algorithm as a Machine SUIF optimization pass.

2 Static Single Assignment Form

Many optimization algorithms need to know the relationship between uses of temporaries (or variables) and the points where they are evaluated. We wants to know either the set of points in the flow graph where the value computed by an instruction is used or the set of evaluations whose value might be used at this point in the flow graph. The static single assignment form (called SSA form). is a compact representation of these facts.

The basic idea used in constructing SSA form is to give unique names to the targets of all assignments in a procedure, and then overwrite uses of the assignment with the new name. More complicated programs have branch and join nodes. At the join nodes, we must add

a special form of assignment called a *ϕ-function* (also called *ϕ-node*). A ϕ-function at the entrance to a node B has the form

$$x \leftarrow \phi(x_1, x_2, \ldots, x_N)$$

where N is the number of control flow predecessors of B.

The semantics of the ϕ-functions are simple. It selects the value of the sources x_i that corresponds to the the block from which control is transferred and assigns this value to the destinations x. If control reaches B from its jth predecessors, then x is assigned the value of x_j. Any ϕ-function at B are executed before the ordinary statements in the node. The figure 1 shows an example.

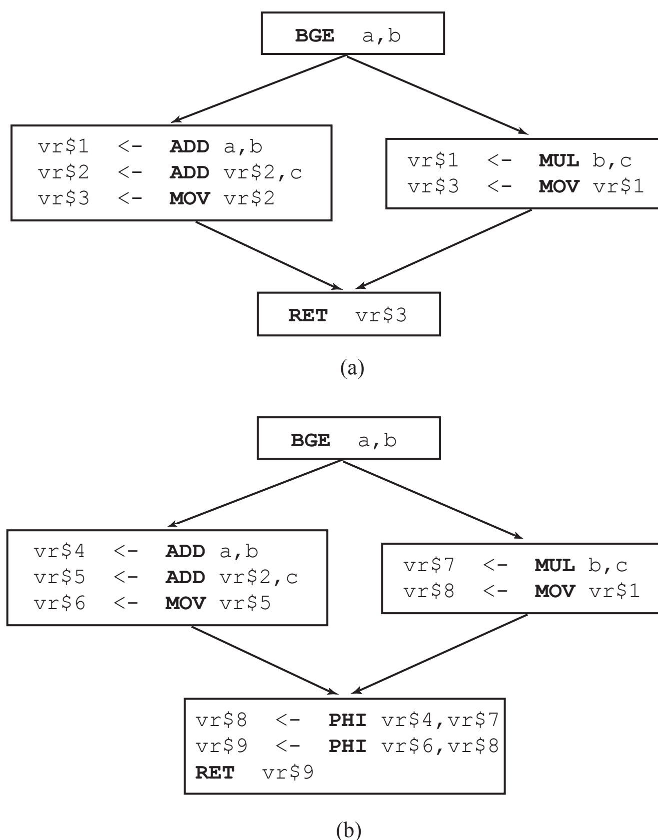

Figure 1: Example of a graph in SSA form (b) and its corresponding orginal CFG form (a). Remark that each temporary is assigned only once.

Hopefully the Machine SUIF infrastructure gives a static single assignment library, and a translation into the SSA form needn't be implemented.

3 Description of the Algorithm

Ideally, the algorithm would be simulate all possible executions of the flow graph and observe the temporaries that are constants. Of course, this is impractical, so an approximation to

this simulation need to be created. What can the the pass do at compile time ?

- If the single definition that defines the temporary is a load constant instruction, the pass knows that the temporary holds that constant.
- If all the operands of an instruction are constants or an algebraic applies indicates that a constant value results, then the target temporary is a constant.
- If all the operands of a ϕ-node are the same constant, the pass can deduce that the value of the target temporary is the same constant.
- If the pass can determine that certain paths are not possible because of other constants occurring in branching instructions, the pass can ignore those paths.

3.1 Representing Arithmetic

While simulating the execution of the program, the pass will assign some symbolic value to each temporary. There are only three classes of values that the pass need to record :

undefined represents the value of a temporary when no value has yet been assigned to it.

overdefined represents the value of a temporary that the pass has determined might not be a constant. The value seams to be varying.

constant(c**)** represents the value of a temporary that the pass has determined as a known constant and c is its value.

This representation is implemented as a small class that encapsulate all these properties:

```
class InstVal {
  enum {
    undefined,          // This instruction has no known value, undef.
    constant,           // This instruction has a constant value, const(c).
    overdefined         // This instruction has an unknown value, var.
  } LatticeValue;
  // If constant value, the current value as a const operand
  Opnd ConstantVal;
public:
  inline InstVal() : LatticeValue(undefined), ConstantVal() {}

  inline bool isUndefined()   const;
  inline bool isConstant()    const;
  inline bool isOverdefined() const;

  // change value as varying - Return true if this is a new status to be in...
  inline bool InstVal::markOverdefined();
  // change value as constant(val) - Return true if this is a new status for us...
  inline bool InstVal::markConstant(Opnd val);
  // get the constant value if applicable
  inline Opnd getConstant() const;
}; // end of class InstVal
```

All of the temporaries except the formal parameters and the global variables are initialized to have the value `undef`. The formal parameters and the global variables are initialized to have the value `overdef` since the algorithm performs no inter-procedural analysis. As the algorithm progress it will either mark a temporary as having a constant value or the varying value. Later the pass may determine if that what it thought was a constant is really varying. Once a value becomes `overdef`, it is never changed back to a constant value.

3.2 Simulating the Arithmetic

The pass needs then a function to evaluate the effect of each instruction given the values of the operands in the arithmetic system describe in the precedent section. What are the arithmetic rules in this extended arithmetic ?

3.2.1 Binary Instructions

For two constant values, the arithmetic is target-machine arithmetic (conventional arithmetic). If one of the operands is undef, then the whole value is undef. If one of the operands has a varying value then the whole operation has a varying value. These rules are presented in the following table:

`dst <- OP src0,src1`	`undef`	`constant(`c_1`)`	`overdef`
`undef`	`undef`	`undef`	`undef`
`constant(`c_2`)`	`undef`	`constant(`c_1 `OP` c_2`)`	`overdef`
`overdef`	`undef`	`overdef`	`overdef`

3.2.2 Unary Instructions

For unary expressions, the rules are very simple and they are presented in the following table:

`dst <- OP src`	
`undef`	`undef`
`constant(`c`)`	`constant(OP` c`)`
`overdef`	`overdef`

3.2.3 ϕ-nodes

The rules for ϕ-nodes are not so complicated even though their number of source operands is variable. First if any of the operands is varying then the value of the ϕ-node is varying. If any two operands of the ϕ-node are distinct constant, then the value of the ϕ-node is varying. If there is at least one constant operand with no varying operand and all the constants are the same, then the value of the ϕ-node is that constant. Otherwise, all the arguments of the ϕ-node are undefined, so the value of the ϕ-node is undefined. These rules for ϕ-nodes with two source operands are presented in the following table:

`dst <- PHI src0,src1`	`undef`	`constant(`c_1`)`	`overdef`
`undef`	`undef`	`constant(`c_1`)`	`overdef`
`constant(`c_2`)`	`constant(`c_2`)`	`constant(`$c_1 == c_2$? c_1 : `overdef)`	`overdef`
`overdef`	`overdef`	`overdef`	`overdef`

3.2.4 Implementation

To apply these rules, a function is implemented which takes an instruction as an argument and update the value of the target operand. It returns a boolean value indicating whether the target operand has changes value. This function is structured with a large switch statement with one entry for each opcode. Then for each opcode we compute the effect of the instruction on the target operands and return the value true if it changes value.

```
// instruction evaluation to simulate arithmetic
bool SparseCondConstProp::sccp_evaluate(Instr *I) {
  // state changed
  bool value_changed = false;
  // get the instruction opcode
  int op_code = get_opcode(I);
  // dst operand if any
  Opnd dst;
  // srcs operand
  Opnd src0, src1;

  // case the op_code and evaluate the instruction
  switch (op_code) {

  case suifvm::LDC:
    // set dst as constant
    ....
    break;

  case suifvm::LDA:
  case suifvm::LOD:
    // adresse argument are overdefined
    ....
    break;

  case suifvm::CAL:
    // function return are overdefined
    ....
    break;

  case suifvm::ADD: case suifvm::SUB:
  case suifvm::MUL: case suifvm::DIV:
  case suifvm::SEQ: case suifvm::SLE:
  case suifvm::SNE: case suifvm::SL:
  case suifvm::AND: case suifvm::IOR:
  case suifvm::XOR:
    // implement rule table for binary operation
    ....
    break;

  case suifvm::CVT: case suifvm::MOV:
  case suifvm::ABS: case suifvm::NOT:
  case suifvm::NEG:
    // implement rule table for unary operation
    ....
  } // end switch
  return value_changed;
}
```

Another function implments the rules for ϕ-nodes:

```
// Phi-node evaluation with Phi-node arithmetic
bool SparseCondConstProp::sccp_evaluate(PhiNode *PN) {
  // state changed
  bool value_changed = false;

  // implement rule table for PHI-nodes
  ....
  return value_changed;
}
```

3.3 Simulating Conditional Branches

The algorithm can eliminate the code that cannot be executed. To do this, there is a set of rules for conditional branches that will simulate the effect of the destinations of only the branches that the compiler can determine might be executed. If an operand of a conditional branch is undefined, it indicates that no possible paths out of this instruction are yet known, so the pass will stop evaluating instructions at this point. If any of the operands in a conditional branch is varying, then any possible destination of the branch is assumed to be executable, so both the true and false alternative are simulated. If the operands are constants, then these constants are used to determine which single destinations are simulated. A single function has been implemented to return the possible destination of a control-transfer instruction:

```
// getFeasibleSuccessors - Returns a vector of boolean to indicate which
// successors are reachable from a given CTI
void SparseCondConstProp::getFeasibleSuccessors(Instr *I,
std::vector<bool> &Succs) {
  Opnd src0;
  Opnd src1;
  int opcode = get_opcode(I);
  switch(opcode) {
  case suifvm::JMP:
    // the destination is always taken
    Succs[0] = true;
    break;
  case suifvm::BTRUE:
  case suifvm::BFALSE:
    // implement unary comparison rules and branch rules
    ....
    break;

  case suifvm::BEQ:
  case suifvm::BNE:
  case suifvm::BGE:
  case suifvm::BGT:
  case suifvm::BLE:
  case suifvm::BLT:
  // implement binary comparison rules and branch rules
    ....
    break;
  } // end switch
}
```

3.4 Simulating the Flow Graph

The pass uses a simplified technique for simulating the flow graph in SSA form. Instead of following each possible path through the graph, the pass need only compute the effect of a temporary changing value. These condition suggest a work-list algorithm. The pass need only revaluate an instruction when at least one of its operands changes. So when the evaluation of an instruction changes, all others instruction that use this result are inserted in the work list to be revaluated.

How does the compiler know when an instruction may be executed ? When a block is executed, some of the successor blocks will be executed, depending on the value of the temporaries controlling the conditional branch. Since one conditional branch can introduce multiple possible destination blocks, a work-list algorithm is again suggested. So we have two work lists: one for the instructions needing reevaluation and another for blocks that have become executable.

First let's see the data structure needed:

```
class SparseCondConstProp {
  public:
    ....
  protected:
    ....
    // Visited is the set of blocks that have been visited.
    std::set<CfgNode*>    Visited;
    // InstrWorkList contains the set of instructions that need reevaluation.
    std::vector<Operation*>   InstrWorkList;
    // BBWorkList is a work list of blocks. A block enters the list whenever it is
    // possible that the block might be executed, that is, each time the conditional
    // branch in one of its predecessor indicates that this block has become executable.
    std::vector<CfgNode*>   BBWorkList;
    // the state each operand is in ...
    std::map<Opnd, InstVal, CmpOpnd> ValueState;
    // a matrix of boolean to store edge between block transition
    std::vector< std::vector<bool> > Executable;
    ....
```

Then we need a function to simulate a block. Each block is simulated again when one of its predecessor edge becomes executable, and this changes the value of the ϕ-nodes at the beginning of the block. When a new edge is present, a new operand becomes relevant in each ϕ-node. The First time a block is processed, all of the other instructions in the block are evaluated.

```
// block simulation - each block is simulated again when one of its predecessor edge
// becomes executable. This changes the value of the Phi-nodes.
void SparseCondConstProp::sccp_block(CfgNode *BB) {
  ....
  // get all Phi-nodes in the current Basic Block
  List<PhiNode*> phi_nodes_list = ssa_form->phi_nodes(BB);
  // and simulate them
  for (List<PhiNode*>::iterator It = phi_nodes_list.begin();
       It != phi_nodes_list.end(); ++It) {
    sccp_instruction(*It);
  }
  // check if BB has already been visited
  if (Visited.find(BB) != Visited.end()) {
    // the BB has already been visited, don't simulate it
    return;
  } else {
    // simulate this Basic Block for the first time
    // we have to simulate all the instruction in it
    for (InstrHandle h = start(BB); h != end(BB); ++h) {
      Instr *instr = *h;
      sccp_instruction(instr);
    } // end for
  } // end if
  ....
}
```

Another function need to be implemented to get the instruction needing reevaluation because the evaluation of another function has been changed. This function is used collectively with `sccp_evaluate(Instr*)`. This function get the modified operand in argument. First it find all the instructions which use this operand and then add them into the work list.

```
inline void SparseCondConstProp::updateInstrUses(Opnd UsedOpnd) {
  // get all instructions or Phi-nodes which use the usedOpnd
  List<Occurrence> occ_list = ssa_form->use_chain(UsedOpnd);
  for (List<Occurrence>::iterator it = occ_list.begin(); it != occ_list.end(); ++it) {
    Occurrence *use = &(*it);
    // add the operation into the work list
```

```
    if (use->is_instr_handle()) {
      Instr *instr = *use->get_instr_handle();
      InstrWorkList.push_back(new Operation(use->get_instr_handle()));
    } else {
      InstrWorkList.push_back(new Operation(use->get_phi_node()));
    } // end if
  } // end for
}
```

We can now implement a function that simulate an arithmetic instruction, and schedule reevaluation of instructions that need it:

```
// instruction simulation
void SparseCondConstProp::sccp_instruction(Instr *I) {
  ....
    // evaluate the instruction
    if (sccp_evaluate(I)) {
      // iterate over each targets of the instruction
      for (int i = 0, m = dsts_size(I); i < m; i++) {
        Opnd target = get_dst(I,i);
        updateInstrUses(target);
      } // end for
    } // end if
  ....
```

And the same one simulates a control transfer instruction:

```
// instruction simulation
void SparseCondConstProp::sccp_instruction(Instr *I) {
  ....
    std::vector<bool> succ_feasible(cur_block->get_succ_count());
    ....
    // get feasible blocks from the CTI instruction
    getFeasibleSuccessors(I, succ_feasible);
    // get all the possible destination Block for this instruction
    for(int i = 0, m = cur_block->get_succ_count(); i < m; i++) {
      // get the target block
      CfgNode *target_block = cur_block->get_succ(i);
      // see if target_block is feasible from the current CTI
      if (! succ_feasible[target_block->get_number()])
         continue; // target_block not feasible thus we continue to the next one
      else { // target block is feasible !!
        // set the execution edge as truly executable
        if (! Executable[this_block_id][target_block_id]) {
          Executable[this_block_id][target_block_id] = true;
          // add the target block to BlockList
          BBWorkList.push_back(target_block);
        } // end if
      } // end if
    } // end for It
  } // end if
}
```

At last, we can now describe the main function. To start the whole algorithm, the entry block is placed in the work list (this is done in the function `sccp_initialize()`. The algorithm works around the the two work lists and completes when both are empty, it means that no more instructions are changing value and no more blocks are becoming executable:

```
void SparseCondConstProp::do_opt_unit(OptUnit *unit) {
  ....
  // first initialize the algorithm
  sccp_initialize();
  // process till both work list are empty
  while ((! InstrWorkList.empty()) || (! BBWorkList.empty())) {
    // process the instruction work list
```

```
    while (! InstrWorkList.empty()) {
      // take O from WorkList
      Operation *O = InstrWorkList[0];
      InstrWorkList.erase(InstrWorkList.begin());
      // simulate instruction O
      sccp_instruction(O);
    } // end while
    // process the Basic Block WorkList
    while (! BBWorkList.empty()) {
      CfgNode *BB = BBWorkList.back();
      BBWorkList.pop_back();
      // simulate block BB
      sccp_block(BB);
    } // end while
  } // end while
  ....
```

3.5 Improving the Flow Graph

The constant propagation algorithm described above does not change the flow graph. It computes information about the flow graph. The pass can now use this information to improve the graph in three ways:

- The instructions whose destination temporary has been evaluated as constant are modified to be a load constant instructions.
- The instructions of which some operands has been evaluated as constant are modified to use the constants instead of temporaries.
- An edge that has not become executable is eliminated, and the conditional branching instruction representing that edge is modified to be a simpler instruction. The ϕ-node at the head of the edge must be modified to have one less operand.

Let's remember that the algorithm described above compute for each temporaries in a procedure its symbolic value.

Notes that after all these transformations on a control flow graph, many dead code might occur. There are two reasons:

- Many load constants to temporaries have been inserted in the code but some of these constants might have been propagated further through the code. And thus, it often happens that these load constants are useless.
- When an edge has been eliminated, it may be possible that the destination block becomes unexecutable.

Thus, it is a good idea to run the dead code elimination pass just after the sparse conditional constant propagation optimization.

References

[1] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 291–299. ACM Press, 1985.