The Machine-SUIF Control Flow Analysis Library

Release version 2.00.11.30

Glenn Holloway and Michael D. Smith
{holloway,smith}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

November 30, 2000

Abstract

The Machine-SUIF control flow analysis (CFA) library builds on the control flow graph (CFG) library. It currently provides dominator analysis and natural-loop analysis. The `DominanceInfo` class computes dominator sets, the dominator tree, and dominance frontiers in either the forward or reverse graphs. Given dominance information, the `NaturalLoopInfo` class finds natural loops, computes the loop depth of each CFG node, and provides predicates that identify loop boundaries.

Contents

1 Introduction

The Machine-SUIF control flow analysis (CFA) library builds on the control flow graph (CFG) library. It currently provides dominator analysis and natural loop analysis. The `DominanceInfo` class, described in Section 2, computes dominator sets, the dominator tree, and dominance frontiers in either the forward or reverse graphs. Given dominance information, the `NaturalLoopInfo` class, described in Section 3, finds natural loops, computes the loop depth of each CFG node, and provides predicates that identify loop boundaries.

2 Dominator Analysis

2.1 Class `DominanceInfo`

Class `DominanceInfo` can compute and cache dominance information about a CFG. For each node n, it can produce:

- the immediate dominator of n;
- the set of dominators of n; and
- the dominance frontier of n.

It can analyze the CFG either in the forward or the reverse direction. Forward analysis views each edge as oriented "normally", i.e., in the direction of control flow, and it starts from the entry node. Reverse analysis views each edge as reversed, and it starts from the exit node. By using the reverse graph, methods of `DominanceInfo` can produce for each node n:

- the immediate postdominator of n;
- the set of postdominators of n; and
- the reverse dominance frontier of n.

Because only some of the possible information is needed in a typical application, everything is computed on demand. You call a `find_`... method to perform each analysis, then use other methods to access the results.

3 ⟨*class* `DominanceInfo` 3⟩≡ (5b)

```
class DominanceInfo {
  public:
    DominanceInfo(Cfg *graph);
    virtual ~DominanceInfo();

    Cfg *graph() const { return _graph; }

    // Perform control-flow analysis
    void find_dominators();
    void find_postdominators();
    void find_dom_frontier();
    void find_reverse_dom_frontier();

    // Access analysis results
    bool dominates(int n_dominator, int n_dominatee) const;
    bool dominates(CfgNode *dominator, CfgNode *dominatee) const;
    bool postdominates(int n_dominator, int n_dominatee) const;
    bool postdominates(CfgNode *dominator, CfgNode *dominatee) const;
```

```
    const NatSet *dominators(int n) const;
    const NatSet *dominators(CfgNode *n) const;
    const NatSet *postdominators(int n) const;
    const NatSet *postdominators(CfgNode *n) const;
    CfgNode *immed_dom(int n) const;
    CfgNode *immed_dom(CfgNode *n) const;
    CfgNode *immed_postdom(int n) const;
    CfgNode *immed_postdom(CfgNode *n) const;
    const NatSet *dom_frontier(int n) const;
    const NatSet *dom_frontier(CfgNode *n) const;
    const NatSet *reverse_dom_frontier(int n) const;
    const NatSet *reverse_dom_frontier(CfgNode *n) const;

    void print(FILE * = stdout) const;

  ⟨DominanceInfo protected parts 5a⟩
  };
```

When constructing a `DominanceInfo` object, supply a pointer to the CFG to be analyzed. To recover this underlying CFG, use method `graph`.

To see the state of a `DominanceInfo` object during debugging, use the `print` method. The information is indexed by the CFG node numbers. Use the `print` method of the CFG to learn more about the nodes.

The remaining `DominanceInfo` methods are for generating or accessing analysis results.

Call `find_dominators` to compute the dominator relation in the forward graph. Then access this information through the following methods.

`immed_dom(`n`)`	Returns `NULL` if n is the entry node. Otherwise, returns the immediate dominator of node n, i.e., its parent in the dominator tree.
`immed_dom(`i`)`	Returns `immed_dom(`n_i`)`, where n_i is the node with number i.
`dominates(`$n1$`, `$n2$`)`	Returns `true` when node $n1$ dominates node $n2$.
`dominates(`$i1$`, `$i2$`)`	Returns `dominates(`n_{i1}`, `n_{i2}`)`, where n_{i1} (n_{i2}) is the node with number $i1$ ($i2$).
`dominators(`n`)`	Returns a set of node numbers representing the dominators of node n.
`dominators(`i`)`	Returns `dominators(`n_i`)`, where n_i is the node with number i.

Call `find_postdominators` to compute the dominator relation in the reverse graph. The results are available through methods exactly analogous to those just described:

`immed_postdom(`n`)`	Returns `NULL` if n is the exit node. Otherwise, returns the immediate postdominator of node n.
`immed_postdom(`i`)`	Returns `immed_postdom(`n_i`)`, where n_i is the node with number i.
`postdominates(`$n1$`, `$n2$`)`	Returns `true` when node $n1$ postdominates node $n2$.
`postdominates(`$i1$`, `$i2$`)`	Returns `postdominates(`n_{i1}`, `n_{i2}`)`, where n_{i1} (n_{i2}) is the node with number $i1$ ($i2$).
`postdominators(`n`)`	Returns a set of node numbers representing the postdominators of node n.
`postdominators(`i`)`	Returns `postdominators(`n_i`)`, where n_i is the node with number i.

The `find_dom_frontier` and `find_reverse_dom_frontier` methods compute the dominance frontiers and postdominance frontiers of each node in the graph. Access the results of the former through:

`dom_frontier(`n`)`	Returns a set of node numbers representing the dominance frontier of node n in the forward flow graph.
`dom_frontier(`i`)`	Returns `dom_frontier(`n_i`)`, where n_i is the node with number i.

Access the results of calling `find_reverse_dom_frontier` through the methods:

`reverse_dom_frontier(`n`)`	Returns a set of node numbers representing the dominance frontier of node n in the reverse flow graph.
`reverse_dom_frontier(`i`)`	Returns `reverse_dom_frontier(`n_i`)`, where n_i is the node with number i.

2.2 Implementation

For finding dominator sets, we use an iterative algorithm similar to Algorithm 10.16 in Aho, Sethi, and Ullman [1]. In a future release, we expect to switch to the algorithm of Lengauer and Tarjan [4], which should be more efficient on most flow graphs.

The immediate dominator of a node n is obtained by choosing the strict dominator of n whose dominator set differs from that of n only by removal of n.

We compute dominance frontiers using the algorithm of Cytron *et al* [2].

The privy parts of class `DominanceInfo` include methods that produce: dominator sets for each node, immediate dominators for each node, and the dominance-frontier set for each node. These methods each take a flag indicating whether to operate on the forward or the reverse graph (`true` means forward).

An instance of the class holds a pointer to the underlying CFG, as well as pointers that are either `NULL` (before the corresponding analysis has been performed) or point to an array of analysis results, with one entry per CFG node, in order of the node number.

5a ⟨`DominanceInfo` *protected parts* 5a⟩≡ (3)

```
protected:
  NatSetDense *do_dominators(bool forward) const;
  CfgNode **do_immed_dominators(bool forward) const;
  void do_dom_frontiers(CfgNode *, bool forward);

private:
  Cfg *_graph;

  NatSetDense *_doms;          // bit vector per node
  NatSetDense *_pdoms;         // bit vector per node
  CfgNode **_idom;             // node per node
  CfgNode **_ipdom;            // node per node
  NatSetDense *_df;            // bit vector per node
  NatSetDense *_rdf;           // bit vector per node
```

2.3 Header file for module `dom.h`

The `DominanceInfo` class is defined in module `dom`, which has the following header file.

5b ⟨*dom.h* 5b⟩≡

```
/* file "cfa/dom.h" -- Dominance Analysis */

⟨Machine-SUIF copyright 8c⟩

#ifndef CFA_DOM_H
#define CFA_DOM_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "cfa/dom.h"
#endif

#include <machine/machine.h>
#include <cfg/cfg.h>

⟨class DominanceInfo 3⟩

#endif /* CFA_DOM_H */
```

3 Natural loop analysis

3.1 Class NaturalLoopInfo

Class `NaturalLoopInfo` computes and caches information about the natural loops in a CFG.

6 ⟨*class* `NaturalLoopInfo` 6⟩≡ (7b)

```
class NaturalLoopInfo {
  public:
    NaturalLoopInfo(DominanceInfo *dom_info)
        : _dom_info(dom_info), _depth(NULL), _loop(NULL) { }
    virtual ~NaturalLoopInfo() { delete [] _depth; }

    DominanceInfo *dom_info() const { return _dom_info; }

    void find_natural_loops();

    const NatSet* loop_at(int n) const;
    const NatSet* loop_at(CfgNode *n) const;
    int loop_depth(int n) const;
    int loop_depth(CfgNode *n) const;
    void set_loop_depth(CfgNode *n, int d);
    void set_loop_depth(int n, int d);

    bool is_loop_begin(int n) const;    // true if block is loop entry
    bool is_loop_begin(CfgNode *cn) const;
    bool is_loop_end(int n) const;      // true if block jumps to loop entry
    bool is_loop_end(CfgNode *cn) const;
    bool is_loop_exit(int n) const;     // true if block is a loop exit
    bool is_loop_exit(CfgNode *cn) const;

    void print(FILE* = stdout) const;

⟨NaturalLoopInfo protected parts 7a⟩
};
```

An instance of `NaturalLoopInfo` is constructed from a pointer to `DominanceInfo` because dominators are used for the loop analysis. You can recover the `DominanceInfo` object by using the `dom_info` method.

Like dominance analysis, natural loops are computed only on demand. Call `find_natural_loops` to run (or rerun) the analysis. Before doing so, you must have called `find_dominators` on the underlying `DominanceInfo` structure.

The `loop_at()` method takes a node n (or its node number) and returns the set of node numbers comprising the natural loop with header n. (If n is not a loop header, the result set is empty.)

The `loop_depth()` method returns the number of natural loops that contain a node. This can be set using the `set_loop_depth()` methods, in case you need to override the results of `find_natural_loops()`.

The `is_loop_begin()` method returns `true` if the specified node dominates any of its predecessors. The `is_loop_end()` method returns `true` if the specified node is dominated by any of its successors. And the `is_loop_exit()` method returns `true` if the specified node has any successor with a different loop depth. Note that this may not be the right loop exit criterion to use in all cases.

3.2 Implementation

For finding natural loops, we use a method based on Algorithm 10.1 in Aho, Sethi, and Ullman [1].

An instance of `NaturalLoopInfo` stores the dominance information provided at its construction in its `_dom_info` field. When a method needs the CFG, it goes though `_dom_info` to reach it. An instance of `NaturalLoopInfo` also contains a pointer `_depth` that is `NULL` until `find_natural_loops` is called and thereafter points to an array of loop depths, one for each node in the CFG.

7a ⟨`NaturalLoopInfo` *protected parts* 7a⟩≡ (6)

```
  private:
    DominanceInfo *_dom_info;
    int *_depth;                       // int per node
    NatSetDense *_loop;                // bit vector per node
```

3.3 Header file for module `loop.h`

The `NaturalLoopInfo` class is defined in module `loop`, which has the following header file.

7b ⟨*loop.h* 7b⟩≡

```
/* file "cfa/loop.h" -- Natural Loop Analysis */

⟨Machine-SUIF copyright 8c⟩

#ifndef CFA_LOOP_H
#define CFA_LOOP_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "cfa/loop.h"
#endif

#include <machine/machine.h>
#include <cfg/cfg.h>
#include <cfa/dom.h>

⟨class NaturalLoopInfo 6⟩

#endif /* CFA_LOOP_H */
```

4 Library Boilerplate

4.1 Initialization and finalization

Every SUIF library has a pair of routines for initialization and finalization. For the CFA library, these are declared in the `cfa` module.

8a ⟨*cfa library initialization* 8a⟩≡ (8b)
```
extern "C" void init_cfa(SuifEnv* suif_env);
```

4.2 Header file for the CFA library

The following is the header file is for use by other libraries and passes that depend upon the CFA library. It is never included in any implementation file within the `machsuif/cfa` directory. We use comments to indicate dependences among the header files.

8b ⟨*cfa.h* 8b⟩≡
```
/* file "cfa/cfa.h" */

⟨Machine-SUIF copyright 8c⟩

#ifndef CFA_CFA_H
#define CFA_CFA_H

#include <machine/copyright.h>

#include <cfa/dom.h>
#include <cfa/loop.h>

⟨cfa library initialization 8a⟩

#endif /* CFA_CFA_H */
```

5 Copyright

All of the code is protected by the following copyright notice.

8c ⟨*Machine-SUIF copyright* 8c⟩≡ (5b 7b 8b)

6 Summary

The CFA library was originally part of the Machine-SUIF CFG library, which supports not only analysis of programs, but also CFG-level transformations, code layout, and fine-grained code motion. At Harvard, the analysis facilities described in this document have been useful in transformation to SSA form, in dead code elimination, and in register allocation. For Machine SUIF version 2, we segregated these analysis functions to make the basic CFG library simpler.

7 Acknowledgments

This document is heavily based on the CFG library in Machine SUIF version 1.1.2 that was done by Cliff Young. Cliff traced the roots of this library all the way back to a CFG library that was written by Mike Smith (one of the co-authors of the current document). A version of this original library was modified by Bob Wilson at Stanford. Tony DeWitt brought the library to Harvard and helped Cliff to adapt the data structure constructors to the Machine-SUIF version 1.1.2 library. Gang Chen wrote the loop analysis code, and Mike built the unified data-flow analysis routines. Other members of the HUBE research group at Harvard contributed useful suggestions to the design. Tim Callahan of Synopsis, Inc. and Berkeley helped to uncover and fix several bugs.

This work is supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225), a NSF Young Investigator award (CCR-9457779), and a NSF Research Infrastructure award (CDA-9401024). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, Compaq, Digital Equipment, Hewlett-Packard, International Business Machines, Intel, and Microsoft.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, pp. 451-490, October 1991.

[3] Glenn Holloway and Michael D. Smith. *The Machine-SUIF Control Flow Graph Library.* The Machine-SUIF compiler documentation set, Harvard University, 1998.

[4] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Languages and Systems*, pp. 121-141, July 1979.