The Machine-SUIF Control Flow Graph Library

Release version 2.02.07.15

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

July 15, 2002

Abstract

The Machine-SUIF CFG library provides an abstraction of control flow graphs with nodes containing lists of machine instructions. Such a CFG can be used to replace the linear instruction-list representation of the body of a procedure being compiled. The library supports translation of the linear representation to and from CFG form, transformation of a CFG by adding, reconnecting, and removing nodes, and fine-grained control over individual program instructions within CFG nodes. It allows precise control of program layout, to ensure that re-linearization leaves instructions in the desired order.

The CFG library is lean by design. We have factored tools for control-flow and data-flow analysis into separate libraries, based on this one. Users can pick the functionality they need, and they can build other CFG-based tools without incurring overhead for features they don't need.

Contents

1 Introduction

This document discusses the Machine-SUIF control flow graph (CFG) library. It discusses design goals, the interface to the library, implementation decisions, current status, and planned updates.

The Machine-SUIF CFG library presents an abstraction of control flow graphs built on the basic `instr_list` structure of the Machine-SUIF system. It allows you to build CFGs and to transform them by rearranging and reconnecting nodes. Machine SUIF provides companion libraries based on this one that support control-flow and data-flow analysis.

The CFG library consists of files in the `cfg` subdirectory of the *machsuif* distribution package. This document describes its design and use. It assumes that the reader is familiar with the SUIF and Machine-SUIF systems and has read the `machine` library documentation.

The following section discusses the development and research goals that drove the design of our CFG library. Section 3 describes our view of how the library will be used. Sections 4 through 6 describe the programming interface provided by the CFG library, and they provide some notes on its implementation.

2 Goals for the CFG Library

This section lists our goals for the CFG library.

- Analysis. Many compiler analysis passes assume a CFG as a basic data structure. Supporting them amounts to supplying a CFG abstract data type, including methods to traverse and examine the CFG and its nodes. Such methods include enumerating all CFG nodes and edges, enumerating and traversing links to and from a node, and iterating over the instructions that make up a node. These basic abstractions are necessary for analyses that use data or control flow (e.g. live variable analysis, reaching definitions, dead code identification, and natural loop detection).

- Transformation. We wanted to build a CFG library that would support not just construction and analysis of CFGs but also transformations of the CFG that change the underlying program. Many compiler optimizations are essentially CFG manipulations, rearranging and/or duplicating the basic blocks of the program, but leaving the internals of the basic blocks the same. Examples of such optimizations include loop peeling, loop unrolling, and static correlated branch prediction [4]. To support such tasks, the data structures of the CFG library need to be linked to the underlying compiler IR, so that changing graph connections or duplicating CFG nodes results in corresponding changes to the program being compiled.

- Ordering. One possible limitation of the CFG approach is that CFG nodes can have many possible orderings, but actual program code must occupy numbered locations in program memory. On many modern microprocessors, the ordering of CFG blocks in memory can drastically affect performance due to negative branch and cache effects. Optimizations such as code layout [2] and branch alignment [1] reorder the basic blocks of a program to minimize these effects. While these optimizations do not change the graph structure of the program, their effect on performance makes them useful to support in the CFG library. So in addition to supporting graph representations and transformations of programs, we also wanted the CFG library to allow users to specify the layout order of CFG nodes.

- Scheduling. Beyond CFG transformations and reorderings, optimizations such as partial redundancy elimination (PRE) and global instruction scheduling move instructions between basic blocks in the program. The CFG library should allow programs to insert instructions into nodes, to move instructions around, and to add new nodes to the program and CFG. Primitives that support such transformations are provided, but advanced instruction schedulers are expected not to violate library conventions.

The next section gives an overview of how we expect the library to be used.

3 Philosophy

One of the things that makes the control-flow graph such a useful compile-time abstraction is that it abstracts away from sequential code. It enables graph-based transformations and allows them to be expressed without reference to a linear code layout. By parceling a procedure's instruction list into basic blocks, we make it efficient to reorder the blocks without having to splice their instructions into a new linear order.

The question arises whether to manage the control instructions within a CFG so that at all times they correctly reflect *some* linear order. Take the code for a simple `if ... then ... else ...` statement. Suppose the code generator has produced a conditional branch instruction to the `else` statement, and has place the `then` statement after the `if` test. The `then` code ends with a `jump` around the `else` statement to the join point. Breaking basic blocks after control instructions will put a `jump` at the end of the node for the `then` statement, and none at the end of that for the `else` statement, which just falls through. If a transformation now inverts the sense of the conditional branch and adjusts CFG flow edges accordingly in the CFG, what should be done to the instruction lists in the `then` and `else` nodes? The natural re-linearization algorithm would remove the `jump` from the end of the `then` node and insert one at the end of the `else` node. Should such `jump` adjustments be made every time the CFG changes?

For many kinds of optimization, this isn't important, either because they change no control flow at all, or because they are not concerned with the precise schedule of instructions. For these, it's fine to leave the selection of a final layout and the adjustment of `jump` instructions to the library method that re-linearizes the flow graph.

For other optimizations, like instruction scheduling and code placement, it's essential to know exactly what instructions will appear where, and in fact, to be able to control such decisions precisely.

The Machine-SUIF CFG library allows you to control the layout relationship between any pair of nodes, but it also allows you to leave the layout of part or all of a CFG unspecified. At the time two nodes become constrained to be adjacent in memory order, the library may remove or insert a trailing `jump` instruction to achieve consistency with this constraint. But in the absence of layout constraints, it doesn't insert or remove `jump`s as the graph is transformed.

To implement this policy efficiently, the library must have quick access to the control-transfer instruction (CTI), if any, in each node. It relies on there being at most one CTI, and it keeps a record of its position. As a client, you're free to modify any of the instructions in a block, but when you change the CTI, you're expected to call a library method to reflect the change.[1] Common operations on nodes, such as replacing one of its control successors, update the CTI directly, without requiring the programmer to make changes at both the graph level and the instruction level.

We have tried to keep the CFG library, and particularly its class interfaces, lean. Though Machine SUIF provides flow analysis libraries based on our CFG abstraction, users may want to write their own. We've tried to make it easy to do so.

4 Class `Cfg`

The two main classes used in the CFG library are `Cfg` and `CfgNode`. A `Cfg` represents an entire control flow graph for a procedure. The nodes within a `Cfg` have class `CfgNode`. The functions operating on class `Cfg` handle the graph overall: graph expansion and simplification, graph validation and printing, and the enumeration of nodes and edges. Those operating on `CfgNode` are about managing a node's relations with its control and layout neighbors and out keeping its code consistent with those relations.

[1] The library doesn't assume that CTI instruction, will be the last in the node; that would make it difficult, for example, to support scheduling for delayed-branch architectures by placing instructions after the control instruction of a block. The library doesn't provide full support for such architectures; e.g., when building a CFG, it always breaks basic blocks at control transfer instructions. Completing the support for target code with delay slots following branch instructions would require subclassing to perform the necessary construction-time adjustments.

4.1 CFG construction

To create a CFG or to normalize the form of an existing CFG, use the following functions:

6a ⟨Cfg *creation and normalization* 6a⟩≡ (8b)
```
Cfg* new_cfg(InstrList*, bool keep_layout    = false,
                         bool break_at_call  = false,
                         bool break_at_instr = false);
void canonicalize(Cfg*,  bool keep_layout    = false,
                         bool break_at_call  = false,
                         bool break_at_instr = false);
void remove_layout_info(Cfg*);
void fix_layout(Cfg*);
```

Creation function `new_cfg` parses a linear list of instructions into basic blocks. There are options for determining precisely what constitutes a basic block and whether the original linear layout of the instructions should be reflected by initial layout constraints. Given an existing CFG, you may want to ensure that it has the same form as if built from scratch with a given option set. The `canonicalize` function serves this purpose.

Each of the flags `keep_layout`, `break_at_call`, and `break_at_instr` is optional. When `keep_layout` is `true`, the CFG gets node-layout constraints reflecting the program's initial linear layout. When `break_at_call` is `true`, each procedure-call instruction is treated as a CTI, so that it terminates the CFG node that contains it. `break_at_instr` means that each node of the graph contains only one instruction.

The `remove_layout_info` function clears all layout constraints in a given graph.

The `fix_layout` function, on the other hand, imposes layout constraints wherever they don't already exist. In some cases, it inserts additional insert unconditional branch instructions. That happens when a potential fall-through edge in the CFG cannot actually be realized by fall-through because of a conflicting layout constraint. Likewise, `fix_layout` may eliminate an unconditional branch if it turns out to be equivalent to fall-through in the newly-imposed layout.

4.2 CFG inspection

6b ⟨Cfg *inspectors* 6b⟩≡ (8b) 7b ▷
```
CfgNode* get_node(Cfg*, int pos);
CfgNode* get_node(Cfg*, CfgNodeHandle);
CfgNode* node_at(Cfg*, LabelSym*);

int nodes_size(Cfg*);
CfgNodeHandle nodes_start(Cfg*);
CfgNodeHandle nodes_last(Cfg*);
CfgNodeHandle nodes_end(Cfg*);

CfgNode* get_entry_node(Cfg*);
CfgNode* get_exit_node(Cfg*);

void set_entry_node(Cfg*, CfgNode*);
void set_exit_node(Cfg*, CfgNode*);
```

Their capsule summaries:

> `get_node(cfg, pos)` returns the node at `pos` in `cfg`'s node list, where `pos` is a position in the list (either a zero-based integer or a handle).
> `get_node(cfg, label)` returns the node of `cfg` at `label`.
>
> `nodes_size(cfg)` returns the number of nodes in `cfg`.

`nodes_start(cfg)` returns a handle on the first element of `cfg`.
`nodes_last(cfg)` returns a handle on the last element of `cfg`.
`nodes_end(cfg)` returns the sentinel handle for `cfg`.

`get_entry_node(cfg)` returns `cfg`'s entry node.
`get_exit_node(cfg)` returns `cfg`'s exit node.
`set_entry_node(cfg, entry)` sets `cfg`'s entry node to `entry`.
`set_exit_node(cfg, exit)` sets `cfg`'s exit node to `exit`.

Type `CfgNodeHandle` is an abbreviation for an STL iterator.

7a ⟨`CfgNodeHandle` *definition* 7a⟩≡ (8b)
```
typedef list<CfgNode*>::iterator CfgNodeHandle;
```

Nicknames The `nodes_` functions are so named because the sequence of node pointers in a `Cfg` is called `nodes`. These functions are frequently used; since a `Cfg` instance contains only one such sequence, we can provide shorter names for these handle-related functions without ambiguity.

7b ⟨`Cfg` *inspectors* 6b⟩+≡ (8b) ◁6b
```
inline int
size(Cfg *cfg)
{
    return nodes_size(cfg);
}

inline CfgNodeHandle
start(Cfg *cfg)
{
    return nodes_start(cfg);
}

inline CfgNodeHandle
end(Cfg *cfg)
{
    return nodes_end(cfg);
}
```

4.3 Graph simplification

The following functions perform graph simplifications. Each returns `true` if it succeeds in changing the graph. Since one kind of simplification can create the opportunity for others, you may want to repeat these transformations until no progress is made by any of them.

7c ⟨`Cfg` *simplification* 7c⟩≡ (8b)
```
bool remove_unreachable_nodes(Cfg*);
bool merge_node_sequences(Cfg*);
bool optimize_jumps(Cfg*);
```

The function `remove_unreachable_nodes(cfg)` removes nodes unreachable from the entry of `cfg`; it renumbers the CFG and thus may invalidate node numbers that you may have saved. The function `merge_node_sequences(cfg)` merges sequences of control-equivalent nodes in `cfg`, while `optimize_jumps(cfg)` eliminates jumps to jumps in `cfg`.

4.4 Graph printing

When debugging code that uses a CFG, it is often useful to be able to print an ASCII representation of the graph.

8a ⟨`Cfg` *printing* 8a⟩≡ (8b)
```
void fprint(FILE*, Cfg*);
void fprint(FILE*, Cfg*, bool follow_layout, bool show_code);
```

The optional flag `follow_layout` causes the order of the printed nodes to follow the layout constraints of any nodes that have such constraints. The optional flag `show_code` causes the instructions in each node to be printed. Both flags default to `false`.

4.5 Header file `graph.h`

Module `graph` has the following header file.

8b ⟨*cfg/graph.h* 8b⟩≡
```
/* file "cfg/graph.h" -- Control Flow Graph */

⟨Machine-SUIF copyright 22⟩

#ifndef CFG_GRAPH_H
#define CFG_GRAPH_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "cfg/graph.h"
#endif

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

⟨CfgNodeHandle definition 7a⟩

⟨Cfg creation and normalization 6a⟩

⟨Cfg inspectors 6b⟩

⟨Cfg simplification 7c⟩

⟨Cfg printing 8a⟩

#endif /* CFG_GRAPH_H */
```

5 Class `CfgNode`

The class `CfgNode` is the data structure used for the individual nodes in the CFG.

5.1 Node Creation

New CFG nodes are created with respect to a particular CFG and are immediately included in the graph's roster of nodes, even though they may have no connection to other nodes.

9a ⟨CfgNode *creation* 9a⟩≡ (15b)
```
CfgNode* new_empty_node(Cfg*);
CfgNode* insert_empty_node(Cfg*, CfgNode *tail, CfgNode *head);
CfgNode* insert_empty_node(Cfg*, CfgNode *tail, int succ_pos);
CfgNode* clone_node(Cfg*, CfgNode*);
```

The function `new_empty_node(cfg)` returns a new empty and isolated node of `cfg`. The function `insert_empty_node(cfg, tail, head)` inserts a new node along edge (`tail`, `head`), while `insert_empty_node(cfg, tail, succ_pos)` inserts a new node between `tail` and successor number `succ_pos`. Note that the sequence of successors of a node is an ordered multiset: there may be several edges from node `tail` to some node `head`. The second form of `insert_empty_node` allows you to say exactly which edge to split with a new node. Finally, `clone_node(cfg, node)` replicates `node` in `cfg`, leaving it disconnected.

5.2 Node Contents

The main content of a CFG node is a list of instructions. You can scan and edit this list using the same function calls that you'd use for the contents of an `InstrList`:

9b ⟨CfgNode *bodily functions* 9b⟩≡ (15b)
```
int instrs_size(CfgNode*);
InstrHandle instrs_start(CfgNode*);
InstrHandle instrs_last(CfgNode*);
InstrHandle instrs_end(CfgNode*);
InstrHandle prepend(CfgNode*, Instr*);
InstrHandle append(CfgNode*, Instr*);
void replace(CfgNode*, InstrHandle, Instr*);
InstrHandle insert_before(CfgNode*, InstrHandle, Instr*);
InstrHandle insert_after(CfgNode*, InstrHandle, Instr*);
Instr* remove(CfgNode*, InstrHandle);
```

where

> `instrs_size(node)` returns the number of elements in `node`.
> `instrs_begin(node)` returns a handle on the first element of `node`.
> `instrs_end(node)` returns the sentinel handle for `node`.
> `prepend(node, instr)` inserts `instr` at the beginning of `node`.
> `append(node, instr)` inserts `instr` at the end of `node`.
> `replace(node, handle, instr)` replaces the element at `handle` in `node` by `instr`.
> `insert_before(node, handle, instr)` inserts `instr` before `handle` in `node`.
> `insert_after(node, handle, instr)` inserts `instr` after `handle` in `node`.
> `remove(node, handle)` removes and returns the instruction at `handle` in `node`.

As with `InstrList`, the `instr_` functions have nicknames because the are so frequently used.

9c ⟨CfgNode *function nicknames* 9c⟩≡ (15b)
```
inline int
size(CfgNode *node)
{
    return instrs_size(node);
}

inline InstrHandle
start(CfgNode *node)
{
    return instrs_start(node);
}

inline InstrHandle
last(CfgNode *node)
{
    return instrs_last(node);
}

inline InstrHandle
end(CfgNode *node)
{
    return instrs_end(node);
}
```

5.3 Graph Node Identification

These OPI functions help identify a given CFG node, by giving a code-label symbol representing the start of the code within it or by printing an ASCII representation of it for debugging.

10 ⟨CfgNode *identification* 10⟩≡ (15b)
```
Cfg* get_parent(CfgNode*);
int get_number(CfgNode*);
LabelSym* get_label(CfgNode*);
void fprint(FILE*, CfgNode*, bool show_code = false,
            bool show_addrs = false, bool no_header = false);
```

Function `get_parent(node)` returns the CFG that owns `node`. Function `get_number(node)` returns a non-negative integer identifier that is less than `nodes_size(get_parent(node))`. The function `get_label(node)` returns the label of `node`'s leading label instruction, inserting a new label instruction if none was present. The `fprint` function is used to support printing of the overall CFG. It prints a one-line node header containing the node number and the nature of its control-termination instruction, if any, and then its lists of successor and predecessor numbers. If the Boolean argument `show_code` is true, then it prints the node's instructions as well. If `show_addrs` is true as well, each instruction is preceded by its hexadecimal address. Finally, if the `no_header` flag argument is true, `fprint` omits the header line, presumably to allow code to be listed by itself.

5.4 Control-Flow Relations

When a CFG is created from a list of instructions, its edges reflect the control paths implied by the control-flow instructions of that list. We call these *normal* edges. The CFG library also supports the creation of two other kinds of edges: *exceptional* and *impossible*.

The assumption usually made when the CFG creator encounters a procedure call instruction is that it returns exactly once for each time the call is executed. In many languages, it is possible to write procedures that may not return in the normal way, or that may return more than once for each call. To reflect such exceptional control paths, we provide a special kind of successor relation, used to describe flow of control from a call site directly to the exit node, or from the entry node to the point immediately after a call. We call such flow *exceptional*, since it is not the usual thing, and since it is often associated with the raising of an exception. We provide functions for creating and recognizing exceptional edges in a CFG.

Though unusual, exceptional edges are at least possible control paths. On the other hand, there are important analysis techniques that insert completely *impossible* edges in the CFG [3]. They may require that every node lies on a path to the exit node, for example. To handle an infinite loop, they insert an impossible edge from one of the loop nodes to the exit. We also provide functions to create and recognize impossible edges.

When the CFG is constructed, impossible edges are included to be sure that every node is reachable from the entry, and every node has a path to the exit. However, no exceptional edges appear in the CFG when it is first constructed. To add them, you call `set_exceptional_succ` explicitly, as described below.

5.5 Finding Node Neighbors

Each `CfgNode` keeps a list of predecessors and a list of successors reachable by any kind of edge. You access these lists through the following functions.

11 ⟨`CfgNode` *control-neighbor functions* 11⟩≡ (15b) 12a ▷

```
int succs_size(CfgNode*);
CfgNodeHandle succs_start(CfgNode*);
CfgNodeHandle succs_end(CfgNode*);

int preds_size(CfgNode*);
CfgNodeHandle preds_start(CfgNode*);
CfgNodeHandle preds_end(CfgNode*);

CfgNode *fall_succ(CfgNode*);
CfgNode *taken_succ(CfgNode*);
```

`succs_size(node)` and `preds_size(node)` return the number of `nodes`'s successor and predecessor nodes, respectively.
`succs_start(node)` and `preds_start(node)` return a handle on `nodes`'s first successor and predecessor, respectively.
`succs_end(node)` and `preds_end(node)` return the sentinel handle for `nodes`'s successor and predecessor sequences, respectively.
`fall_succ(node)` and `taken_succ(node)` return `node`'s first and second successors, respectively.

The sequence of predecessors represents an unordered set, while the sequence of successors corresponds to the target(s) of the node's terminating instruction. The predecessor list is unordered because we could not imagine a useful ordering to implement; it is a set (and not a multiset) because having a set seems a simpler abstraction. The successors sequence is ordered; positions 0 and 1 in the list have special meaning depending on the kind of control instruction that terminates the node. The same target can appear multiple times in the successors list, so that different cases of a multiway branch can jump to the same label. The number of successors of a `CfgNode` depends on the control instruction that ends the node:

- Unconditional jumps and fall-through nodes (nodes that have no terminating control instruction) have just one successor, which is always successor number 0.
- Call instructions also have just one successor, the node to which the call will return.

- Conditional branches have two successors. The fall-through path is always successor number 0, while the taken path is successor number 1.
- Multiway branches have as many branches as there are slots in the branch dispatch table. Changing the jth successor (counting from 0) of a multiway branch modifies the jth dispatch table entry. (There is no "default" target label.)

All exceptional and impossible successors occur after the normal successors.

The functions returning a `CfgNodeHandle` permit individual nodes to be visited in the usual manner. The `fall_succ` and `taken_succ` functions are convenient ways to access successors 0 and 1, respectively.

Making new neighbors. You set the normal successors of a node using one of the following.

12a ⟨`CfgNode` *control-neighbor functions* 11⟩+≡ (15b) ◁11 12b▷

```
CfgNode* get_pred(CfgNode*, int pos);
CfgNode* get_succ(CfgNode*, int pos);
void set_succ(CfgNode*, int pos, CfgNode *succ);
void set_fall_succ(CfgNode*, CfgNode *succ);
void set_taken_succ(CfgNode*, CfgNode *succ);
```

The function `get_pred` returns the `n`-th predecessor of the node to which it's applied (recall that the predecessors are not ordered). `get_succ` returns the node corresponding to the `n`-th successor edge, and different values of `n` may yield the same successor node.

Calling `set_succ` makes `succ` the new `n`-th successor of `node`. It disconnects the former `n`-th successor (if any), and updates the predecessor lists of the old and new successors. If necessary, it also modifies `node`'s CTI, which should be consistent with the existing successor sequence, to reflect the change. For example, if `node` ends with a conditional branch, then

```
set_succ(node, 1, succ);
```

unlinks any previous "taken" successor, sets it to `succ`, and replaces the target symbol of `node`'s branch instruction with the first label of node `succ`.

Functions `set_fall_succ` and `set_taken_succ` are provided for mnemonic value; they are equivalent to calling `set_succ` with second argument `0` and `1`, respectively.

Note that making node e a successor of the entry node has the effect of making e an entry point of the procedure.

Creating exceptional and impossible edges. For exceptional and impossible edges, there are special ways of setting a successor node.

12b ⟨`CfgNode` *control-neighbor functions* 11⟩+≡ (15b) ◁12a 12c▷

```
void set_exceptional_succ(CfgNode*, int n, CfgNode *succ);
void set_impossible_succ(CfgNode*, int n, CfgNode *succ);
```

If necessary, each of the above functions extends the successor sequence of the node to accommodate at least `n`+1 elements. An edge created by `set_exceptional_succ` (`set_impossible_succ`) is marked as exceptional (impossible). Detecting whether a particular numbered successor represents a normal, exceptional, or impossible edge is handled by the following predicates.

12c ⟨CfgNode *control-neighbor functions* 11⟩+≡ (15b) ◁12b 13a▷
```
bool is_normal_succ(CfgNode*, CfgNode *succ);
bool is_normal_succ(CfgNode*, int pos);
bool is_exceptional_succ(CfgNode*, CfgNode *succ);
bool is_exceptional_succ(CfgNode*, int pos);
bool is_possible_succ(CfgNode*, CfgNode *succ);
bool is_possible_succ(CfgNode*, int pos);
bool is_impossible_succ(CfgNode*, CfgNode *succ);
bool is_impossible_succ(CfgNode*, int pos);
bool is_abnormal_succ(CfgNode*, CfgNode *succ);
bool is_abnormal_succ(CfgNode*, int pos);
```

The versions of the above predicates that take an integer `pos` as their second argument test whether the `pos`-th successor edge is of the indicated kind. A "possible" successor is one that is either normal or exceptional, but not impossible. An "abnormal" one is either exceptional or impossible.

To eliminate an abnormal edge of either kind, use one of the following functions.

13a ⟨CfgNode *control-neighbor functions* 11⟩+≡ (15b) ◁12c
```
void remove_abnormal_succ(CfgNode*, CfgNode *succ);
void remove_abnormal_succ(CfgNode*, int pos);
```

Testing the control-transfer instruction. It's often necessary to test whether a node is terminated by a CTI, and if so, what kind.

13b ⟨CfgNode *control-instruction functions* 13b⟩≡ (15b) 13c▷
```
bool ends_in_cti(CfgNode*);     // has CTI
bool ends_in_ubr(CfgNode*);     // has CTI satisfying is_ubr()
bool ends_in_cbr(CfgNode*);     // has CTI satisfying is_cbr()
bool ends_in_mbr(CfgNode*);     // has CTI satisfying is_mbr()
bool ends_in_call(CfgNode*);    // has CTI satisfying is_call()
bool ends_in_return(CfgNode*);  // has CTI satisfying is_return()
```

Adjusting the control-transfer instruction. To this point, we have not described any way of changing the number of successors of a node. The `set_succ` function diverts an existing edge, and it updates the CTI accordingly. But sometimes, there's no substitute for altering the control instruction yourself, and then asking the library to adjust edges accordingly. Here are the relevant functions.

13c ⟨CfgNode *control-instruction functions* 13b⟩+≡ (15b) ◁13b
```
Instr* get_cti(CfgNode*);
InstrHandle get_cti_handle(CfgNode*);

void invert_branch(CfgNode*);

void reflect_cti(CfgNode*, Instr *cti, CfgNode *implicit_succ = NULL);
```

If the node has a control-transfer instruction, then it is returned by `get_cti`. Function `get_cti_handle` is similar, but returns a handle on the CTI's position in the instruction list, or a sentinel handle is there is no CTI.

Function `invert_branch` changes the polarity of a branch by changing its opcode to the opposite opcode (e.g. changing `beq` to `bne`). If the branch has a `branch_edge_weights` annotation, giving the edge frequencies of the node's two out-edges, then `invert_branch` swaps these as well.

Note that there is no corresponding `set_cti` function. To change the CTI, you must change the instruction list directly and then call `reflect_cti`. This function records the new CTI, which must already have been put into the node and changes the node's flow successors to reflect the new control instruction.

Obviously, if that instruction can fall through, the function needs to know what to use as its implicit successor. In that case, pass it a second argument giving the implicit successor node.

`reflect_cti` preserves the exceptional and impossible successors of the node that it operates on, but it completely replaces the normal successors.. If its second argument is `NULL`, rather than a CTI instruction, it clears the node's CTI association and makes `implicit_succ` (its third argument) the node's only normal successor.

5.6 Layout relations

The CFG library supports basic block placement and layout by allowing the user to specify an ordering (partial or total) of the nodes in a CFG and its associated procedure. The ordering is specified by a doubly-linked list that runs through the `CfgNode`s of the graph. Null pointers in this linked list indicate "don't care" orderings, while connected components (this document calls them "clumps") of the list will be laid out in order. This gives you complete flexibility between specifying no order at all (all layout links null) to a total order on nodes.

Checking and changing layout constraints. To read and modify layout information, the library provides the following functions.

14 ⟨`CfgNode` *layout functions* 14⟩≡ (15b)

```
CfgNode *get_layout_pred(CfgNode*);
CfgNode *get_layout_succ(CfgNode*);

void clear_layout_succ(CfgNode*);
bool set_layout_succ(CfgNode*, CfgNode *succ);
bool set_layout_succ(CfgNode*, LabelSym *succ_label);
```

A nodes's layout predecessor is returned by `get_layout_pred`; its layout successor, by `get_layout_succ`.

All of the functions that change a layout constraint between two nodes are invoked on the first of the two in layout order. To remove a layout constraint, use `clear_layout_succ`. To establish a layout constraint, use `set_layout_succ`, supplying the new layout successor as its argument. (For convenience, you can supply a label of the successor node instead.)

The `set_layout_succ` function is careful about branch polarity and explicit jumps, inverting branches and inserting unconditional jumps when necessary. When it needs to invert a branch, this function calls `invert_branch` and then returns `true`.

Effects of layout constraints on code. When a `Cfg` is constructed, no instructions are added or removed, whether or not layout constraints are specified. Thus a procedure can be transformed into CFG form and back to instruction-list form with no changes, whether or not the layout-retention option is selected.

When no layout links have been specified, the library treats the CFG as a general graph. You are free to transform it without having to ensure that there is a linearization of its current instructions that is in valid executable order.

On the other hand, the library insists that nodes with layout successors be valid standard basic blocks. This makes `set_layout_succ` picky about the conditions under which it allows a link to be set.

- It is always legal to set the layout successor of a fall-through node or a node ending in an unconditional branch. If the layout successor is also the control-flow successor, then an unneeded unconditional branch instruction will be removed. If not, a goto will be created to be the terminating control instruction in the node (i.e. it will be returned by `get_cti`). These constraints ensure that any necessary explicit goto's become visible for timing analysis and instruction scheduling.

- It is legal to set the layout successor of a conditional branch node only if the layout successor is either the fall-through successor or the taken successor of the node. The CFG library sets the branch polarity so that the layout successor becomes the fall-through successor of the node. If you actually want the layout successor of a conditional branch node not to be a control-flow successor, then you are responsible for creating a new unconditional branch node below the conditional branch. This can be done easily using the `split_critical_edge` function.

- It is legal to set the layout successor of a node ending in a call instruction only if the layout successor is also the control flow successor.

- It is always legal to set the layout successor of a multiway branch node.

- It is never legal to set a layout successor if adding that link to the layout list would create a layout cycle. Moreover, it is never legal to change the fall-through successor of a node after its layout successor has been set. (The library will allow you to set the fall-through successor to the same value, though).

Bounding a node's computation. It is sometimes useful to find the first "real" instruction in a node, i.e., the first one other than a label or some other kind of marker. It's occasionally useful to identify the last non-control-transfer instruction in a node. The following functions identify such "boundary" instructions within a node.

15a ⟨`CfgNode` *boundary functions* 15a⟩≡ (15b)
```
Instr *first_non_label(CfgNode*);
Instr *first_active_instr(CfgNode*);
Instr *last_non_cti(CfgNode*);
```

Function `first_non_label` returns the first instruction in a node that isn't a label. `first_active_instr` is similar, but skips pseudo-ops and null instructions in addition to labels. Function `last_non_cti` returns the last instruction before the terminating control instruction in the node. Each of these boundary-instruction functions returns the null pointer if there is no qualifying instruction.

5.7 Header file for module `node.h`

The `CfgNode` class and support data structures are defined in the module `node`, which has the following header file:

15b ⟨*cfg/node.h* 15b⟩≡
```
/* file "cfg/node.h" -- Control Flow Graph Nodes */

⟨Machine-SUIF copyright 22⟩

#ifndef CFG_NODE_H
#define CFG_NODE_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "cfg/node.h"
#endif

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

⟨CfgNode creation 9a⟩

⟨CfgNode identification 10⟩
```

```
⟨CfgNode bodily functions 9b⟩

⟨CfgNode function nicknames 9c⟩

⟨CfgNode control-neighbor functions 11⟩

⟨CfgNode control-instruction functions 13b⟩

⟨CfgNode layout functions 14⟩

⟨CfgNode boundary functions 15a⟩

#endif /* CFG_NODE_H */
```

6 Utilities

This section lists a number of helper functions that don't fall into the traditional class hierarchies. These helper functions live in the `util.h` header file.

6.1 Enumeration

6.1.1 Depth-first walk

The following utility walks a subgraph of the CFG in depth-first order, performing a given action after all successors of a node have already been visited. It can walk either the forward or reverse graph. Its arguments are: the node at which to start; the graph orientation (`true` for forward); the set of nodes already visited; the node action to be performed; and an optional flag that causes impossible edges to be ignored.

16a ⟨*node enumeration utilities* 16a⟩≡ (18c)

```
extern void
    depth_first_walk(CfgNode *start, bool forward, NatSet *visited,
                     DepthFirstWalkAction&, bool not_impossible = false);
```

The action applied to each node in the walk is expressed using a subclass of the following little class:

16b ⟨*class* `DepthFirstWalkAction` 16b⟩≡ (18c)

```
class DepthFirstWalkAction {
  public:
    virtual ~DepthFirstWalkAction() { }
    virtual void operator()(CfgNode*) { }
};
```

This class can be used as it is, if no action is needed at each node. Otherwise you specialize it to perform whatever task is needed. The utility in the next section is implemented using a `DepthFirstWalkAction` that accumulates a list of nodes in reverse postorder.

6.1.2 Reverse postorder sequence

The following class presents the nodes of a CFG in reverse postorder [3].

16c ⟨*class* `CfgNodeListRpo` 16c⟩≡ (18c)
```
class CfgNodeListRpo
{
  public:
    CfgNodeListRpo(Cfg *graph, bool forward = true);
    virtual ~CfgNodeListRpo();

    int size();
    CfgNodeHandle start();
    CfgNodeHandle end();

    void prepend(CfgNode*);
    void append(CfgNode*);
    void replace(CfgNodeHandle, CfgNode*);
    void insert_before(CfgNodeHandle, CfgNode*);
    void insert_after(CfgNodeHandle, CfgNode*);
    CfgNode* remove(CfgNodeHandle);

⟨CfgNodeListRpo protected part 17a⟩
};
```

The interface of this class is exactly that of any other sequence in Machine SUIF. Its constructor takes a CFG and an optional boolean indicator of graph orientation. The constructor computes the node sequence; you can inspect and even modify the sequence using the usual set of sequence manipulation actions. By default, or when its second argument is `true`, the constructor operates on the normal forward graph; a `false` second argument causes it to use the reverse graph instead.

Typically, you bind a reverse-postorder it to a local variable and use the handle abstraction to scan it. The sequence is automatically reclaimed when the scope of the local variable ends. For example:

```
{
    CfgNodeListRpo rpo(the_cfg);
    for (CfgNodeHandle h = rpo.start(); h != rpo.end(); ++h) {
        ...                 // use *h to access current node
    }
}                           // list rpo is reclaimed here
```

The implementation of class `CfgNodeListRpo` uses a C++ list of node pointers:

17a ⟨`CfgNodeListRpo` *protected part* 17a⟩≡ (16c)
```
  protected:
    list<CfgNode*> nodes;
```

6.2 Node Identification

The `cfg_node` annotation connects a label symbol back to the corresponding CFG node. The following helper function uses that note to return the node for a given label.

17b ⟨*function* `label_cfg_node` 17b⟩≡ (18c)
```
CfgNode *label_cfg_node(Sym *);
```

To reach the CFG node containing a given instruction, apply this function:

17c ⟨*function* `get_parent_node` 17c⟩≡ (18c)
```
CfgNode* get_parent_node(Instr*);
```

It returns the containing node, if there is one, or else NULL.

6.3 Visualization

The generate_vcg function will write a graph description in a format suitable for the VCG (Visualization for Compiler Graphs) tool to read and display graphically. The FILE * argument should be an open file descriptor to which to write. See the documentation for the VCG tool for more information about the VCG file format.

18a ⟨*visualization support* 18a⟩≡ (18c)
```
extern void generate_vcg(FILE *, Cfg *);        // Dump vcg format file
```

6.4 Node-Sequence Utilities

Sometimes it is useful to inquire if a node pred precedes another node in the CFG; has_pred provides this functionality.

18b ⟨*node-sequence utilities* 18b⟩≡ (18c)
```
extern bool has_pred(CfgNode *node, CfgNode *pred);
```

6.5 Header file for module util.h

We declare module util with the following header file.

18c ⟨*cfg/util.h* 18c⟩≡
```
/* file "cfg/util.h" -- Control Flow Graph Utilities */

⟨Machine-SUIF copyright 22⟩

#ifndef CFG_UTIL_H
#define CFG_UTIL_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "cfg/util.h"
#endif

#include <machine/machine.h>

#include <cfg/cfg_ir.h>
#include <cfg/graph.h>

⟨class DepthFirstWalkAction 16b⟩

⟨node enumeration utilities 16a⟩

⟨class CfgNodeListRpo 16c⟩

extern IdString k_cfg_node;

⟨function label_cfg_node 17b⟩

⟨function get_parent_node 17c⟩

⟨visualization support 18a⟩
```

```
⟨node-sequence utilities 18b⟩

#endif /* CFG_UTIL_H */
```

7 Library initialization

Before you can start using the facilities of the `cfg` library, the library must initialize some parts of itself. In SUIF, this is performed by defining an `init_`*libname* routine.

19a ⟨*cfg library initialization* 19a⟩≡ (19c)

```
extern "C" void init_cfg(SuifEnv* suif_env);
```

The following string keys the annotation that connects a label to the corresponding CFG node.

19b ⟨*cfg string constants* 19b⟩≡ (19c)

```
extern IdString k_cfg_node;
```

The `cfg` library initialization header has the following layout:

19c ⟨*cfg/init.h* 19c⟩≡

```
/* file "cfg/init.h" */

⟨Machine-SUIF copyright 22⟩

#ifndef CFG_INIT_H
#define CFG_INIT_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "cfg/init.h"
#endif

⟨cfg library initialization 19a⟩

⟨cfg string constants 19b⟩

#endif /* CFG_INIT_H */
```

8 Header file for the CFG library

The following is the header file is for use by other libraries and passes that depend on the CFG library. It is never included in any implementation file within the `machsuif/cfg` directory.

19d ⟨*cfg/cfg.h* 19d⟩≡
```
/* file "cfg/cfg.h" */

⟨Machine-SUIF copyright 22⟩

#ifndef CFG_CFG_H
#define CFG_CFG_H

#include <machine/copyright.h>

#include <cfg/cfg_ir.h>
#include <cfg/cfg_ir_factory.h>
#include <cfg/graph.h>
#include <cfg/node.h>
#include <cfg/util.h>
#include <cfg/init.h>

#endif /* CFG_CFG_H */
```

9 Hoof Specification of CFG Classes

This section contains the Hoof code that defines our `Cfg` and `CfgNode` IR objects in SUIF. We refer you to the SUIF documentation if you want to learn more about Hoof and these definitions.

9.1 Class Cfg

20 ⟨*class* `Cfg` 20⟩≡ (21b)
```
concrete Cfg : AnyBody {
    list<CfgNode* owner> nodes;
    CfgNode* reference entry_node;
    CfgNode* reference exit_node;

    list<LabelSym* reference> noted_labels;

    bool keep_layout;
    bool break_at_call;
    bool break_at_instr;

    CPP_DECLARE
      public:
        list<CfgNode*>& nodes() { return _nodes; }
        InstrList* to_instr_list();
        list<LabelSym*>& noted_labels() { return _noted_labels; }
    CPP_DECLARE
    CPP_BODY
        extern InstrList* cfg_to_instr_list(Cfg*);
        InstrList*
        Cfg::to_instr_list()
        {
            return cfg_to_instr_list(this);
        }
    CPP_BODY
};
```

9.2 Class CfgNode

21a ⟨*class* CfgNode 21a⟩≡ (21b)

```
concrete CfgNode : ScopedObject {
    int number;
    Instr* reference cti;
    list<Instr* owner> instrs;
    list<CfgNode* reference> preds;
    list<CfgNode* reference> succs;
    CfgNode* reference layout_pred;
    CfgNode* reference layout_succ;
    IInteger exc_succs;
    IInteger imp_succs;

    CPP_DECLARE
      public:
        list<Instr*>& instrs() { return _instrs; }
        list<CfgNode*>& succs() { return _succs; }
        list<CfgNode*>& preds() { return _preds; }

        IInteger& exc_succs() { return _exc_succs; }
        IInteger& imp_succs() { return _imp_succs; }
    CPP_DECLARE
};
```

9.3 Overall hoof specification for module cfg

The combined hoof grammar for the cfg module has the following layout.

21b ⟨*cfg/cfg_ir.hoof* 21b⟩≡

```
# file "cfg_ir.hoof"
#
#    Copyright (c) 2000 The President and Fellows of Harvard College
#
#    All rights reserved.
#
#    This software is provided under the terms described in
#    the "machine/copyright.h" include file.


#include "machine/machine_ir.hoof"

module cfg_ir {

    include <bit_vector/bit_vector.h>;
    include <machine/machine.h>;

    import basicnodes;
    import machine;

⟨class Cfg 20⟩

⟨class CfgNode 21a⟩
}
```

10 Copyright

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

22 ⟨*Machine-SUIF copyright* 22⟩≡ (8b 15b 18 19)

11 Summary

We have developed a CFG library that supports not only analysis of programs, but also CFG-level transformations, code layout, and fine-grained code motion. At Harvard, we have used this machine library to build compiler passes including dead code elimination, loop peeling and unrolling, static correlated branch prediction, code layout, register allocation, and a number of instruction schedulers.

12 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] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. *Near-optimal Intraprocedural Branch Alignment.* Proc. ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, pp. 183-193, June 1997.

[2] Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. *Procedure Placement using Temporal Ordering Information.* Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture, pp. 303-313, December 1997.

[3] Robert Morgan. *Building an Optimizing Compiler.* Digital Press, 1998.

[4] Cliff Young and Michael D. Smith. "Improving the Accuracy of Static Branch Prediction Using Branch Correlation". *Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 232-241, October 1994.