The Machine SUIF Bit-Vector Data-Flow-Analysis Library

Release version 2.00.11.30

Glenn Holloway and Allyn Dimock
{holloway,dimock}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

November 30, 2000

Abstract

The bit-vector data-flow (BVD) library of Machine SUIF [3] is a framework for iterative, bit-vector-based data-flow analyzers. It uses Machine SUIF's control-flow graph (CFG) library [4] to parse the program being analyzed into basic blocks, and it associates data-flow results with these CFG nodes.

Each specific data-flow solver is derived from an abstract class called `bvd`, which supplies the generic machinery. It is quite easy to develop solvers for problems that fit the classical paradigm.

At present, the BVD library contains two concrete solvers, one that computes liveness information and another that does reaching-definitions analysis. Others will follow as the need for them arises.

Contents

1 Introduction

Data-flow analysis (DFA) is a textbook example of code reuse [1, 2, 5]. Available expressions, live variables, reaching definitions, or other useful sets of properties can be computed for all points in a program using a generic algorithmic framework. Typically, the property sets are represented as bit vectors, and the generic algorithm propagates them iteratively over a flow graph, transforming them monotonically to reflect the effects of basic blocks and the confluence of edges, until they converge at all points. Different problems have different initial conditions for the bit vectors (empty or full), different confluence transformations (union or intersection), different directions of traversal (with control flow or against it), and different rules for expressing the effects of a basic block. But the fact that their solvers fit a common pattern is useful, because once the framework is in place, it enables new analyzers to be added quickly and correctly.

The bit-vector data-flow (BVD) library of Machine SUIF [3] is a framework for iterative, bit-vector-based data-flow analyzers. It uses Machine SUIF's control-flow graph (CFG) library [4] to parse the program being analyzed into basic blocks, and it associates data-flow results with these CFG nodes.

Each specific data-flow solver is derived from an abstract class called `bvd`, which supplies the generic machinery. Section 2 describes this class, and Section 3 describes data structures used by `bvd` for capturing data flow that is local to a basic block.

For now, the BVD library contains a solver for liveness information and a reaching-definitions analyzer. The `liveness` subclass of `bvd` is described in Section 4. Section 5 develops the map from operands to bit-vector positions used in liveness analysis.

2 Class `Bvd`

The classical iterative, bit-vector-based algorithm for data-flow analysis is covered in many general-purpose texts on optimizing compilation. We assume you are familiar with the basic approach, and we concentrate on describing how to use the BVD library to develop instances of this useful paradigm. Our running example will be the liveness analyzer, the details of which are described in Sections 4 and 5.

Terminology. The data-flow analyzers covered here do two things: they identify an interesting class (or *universe*) of syntactic or semantic program elements, and for each such element, they identify the points in the program to which the element, or some aspect of the element, "flows". For an available-expressions problem, the elements are syntactic expressions evaluated by the program, and an expression e flows to point p in the program if e is computed (*generated*) on every path to p without being invalidated (*killed*) by an assignment to some variable in e before p is reached. For the reaching-definitions problem, the universe of interesting elements consists of the statements that assign to, or otherwise side-affect storage locations, and these definitions flow to all the program points reached by paths that don't contain an overriding assignment. For a liveness problem, the universe consists of storage cells, such as variables and registers. The liveness of a variable is generated by a use of the variable, and it (the liveness) flows backward along control paths until killed by a definition of (e.g., an assignment to) the variable.

In the examples just mentioned, the data-flow behavior of each element in the chosen universe is independent of the others. Iterative data-flow analysis, as implemented in the BVD library, exploits this fact by dealing with all the elements in parallel. For each element e and each program point p, it wants to produce one bit of information: true if e flows to p and false if it does not. The parallelism is achieved by packing these bits into bit vectors, with one bit position, or *slot*, per universe element, and then solving the data-flow equations for all elements at once by computing on the bit vectors instead of the individual slots.

To avoid the storage cost of allocating a different bit vector for every point in the program, it is customary to associate them only with the entry and/or exit points of nodes in the CFG of the program. It is easy to propagate from one of these node boundaries through the linear sequence of instructions of the node when necessary.

We take exactly this classical approach. We assume that every data-flow solver determines the universe of interesting elements and assigns a small non-negative number as the identifier of each element. We call this number the *slot* of the element. Instead of using the bit-vector or bit-string metaphor when describing flow-analysis results, we use sets of slots. That is, instead of describing computations in terms of bitwise boolean operations on vectors, we use set operations on sets of small integers. The two metaphors are conceptually equivalent, and of course, we make sure that the sets used for data-flow analysis have the complexity properties that have made "bit-vector-based" data-flow analysis effective and popular. The advantage of the set metaphor is simply that much of the machinery that works for sets based on bit vectors carries over smoothly to sets with other performance properties.

The slot-set types used in this library are derived from `NatSet`, a natural-number set class defined in the Utilities section of the `machine` library document [3]. You should familiarize yourself with the properties of those set types if you haven't already. For data-flow analysis results, we use the class `NatSetDense`, which has bit-vector-like complexity traits. Where a sparse representation is more suitable, we use the list-based set class `NatSetSparse`. The operations on all the `NatSet` classes are the same. There is an iterator class for these set types called `NatSetIter`.

The base class for data-flow solvers. The abstract base class from which you derive a data-flow solver is `Bvd`:

4 ⟨*class* `Bvd` 4⟩≡ (7c)

```
class Bvd {
  public:
    virtual ~Bvd() { }

    const NatSet* in_set(CfgNode*)  const;
    const NatSet* out_set(CfgNode*) const;

    virtual void find_kill_and_gen(Instr *) = 0;

    virtual const NatSet* kill_set() const = 0;
    virtual const NatSet* gen_set() const = 0;

    virtual int num_slots() const = 0;
    void print(CfgNode*, FILE* = stdout) const;

  protected:
    enum direction { forward, backward };
    enum confluence_rule { any_path, all_paths };

    Bvd(Cfg*,
        direction = forward,
        confluence_rule = any_path,
        FlowFun *entry_flow = NULL,
        FlowFun *exit_flow = NULL,
        int num_slots_hint = 0);

    bool solve(int iteration_limit = INT_MAX);

⟨Bvd details 7a⟩
};
```

2.1 Accessing data-flow results

Note that most methods of `Bvd` are protected from public use. Apart from its destructor, the public methods are all about accessing the results of data-flow analysis.

The `num_slots` method (which must defined by a concrete subclass) returns the total number of allocated slots once the solver has been run. In other words, it is the size of the universe of items found in the program during data-flow analysis. At present, this and the `print` method are mainly used for debugging. The essential public methods are those that access the sets associated by `Bvd` with each CFG node.

`in_set(`n`)`	Returns the slot set for the (control) entry point of node n.
`out_set(`n`)`	Returns the slot set for the (control) exit point of node n.

The sets returned are owned by the `Bvd` object; you don't need to worry about deleting them.

To use liveness analysis as an example, `in_set(`*node*`)` indicates which items are live on entry to *node* and `out_set(`*node*`)` indicates those live at its exit. (Note that, even though liveness is a "backward" data-flow problem, the `in_set` method refers to a node's control-flow entry, not its exit.)

2.2 Constructing a data-flow analyzer

The concrete subclass that derives a specific data-flow solver from `Bvd` supplies the following parameters to its constructor.

- The CFG of the procedure to be analyzed.
- The *direction* of the data-flow problem: either `forward`, following edges in the direction of control flow from the entry node, or `backward`, following edges in reverse from the exit node.

  Liveness, for example, is a backward problem. Liveness information is *generated* by use occurrences of variables (say), and propagates backward against the direction of control flow until it is *killed* by definition occurrences (e.g., assignments).
- The *confluence rule* to be used when data flow from multiple nodes is combined: either `all_paths` or `any_paths`.

  The `all_paths` rule means that to obtain the slot describing data flow entering node n, we use *intersection* over the sets that represent flow out of n's data-flow predecessors:

  $$\mathit{before}[n] = \bigcap_{p \in \mathit{pred}(n)} \mathit{after}[p]$$

  The `any_path` rule is the same, except that it uses *union* instead of intersection:

  $$\mathit{before}[n] = \bigcup_{p \in \mathit{pred}(n)} \mathit{after}[p]$$

  The set of data-flow predecessors $\mathit{pred}(n)$ depends on the direction of the problem. In a forward problem, they are the control-flow predecessors; in a backward problem, the control-flow successors. In a forward problem, $\mathit{before}[n]$ is the information at the control-flow entry of n and $\mathit{after}[n]$ is that at its exit. In a backward problem, it's the other way around.

  For all nodes n, the initial value of $\mathit{before}[n]$ depends on the confluence rule. For an any-path problem, these sets start empty and accrue information through set union. With an all-paths (intersection) problem, the $\mathit{before}[n]$ start as the universal set and the final solution is carved out by intersection.

  The liveness example is an any-path problem: a variable is live at the exit of node n (i.e., the point before data flow propagates backward through n) if it is live at the entry of any of n's control-flow successors (i.e., its data-flow predecessors).

- Special flow functions for the entry (`entry_flow`) and exit (`exit_flow`) nodes. Since these nodes contain no code, their local data flow is by default represented using the identity function on slot sets (Section 3). If there are special boundary conditions for a particular data-flow problem, however, use (`entry_flow`) and/or (`exit_flow`) to express them.
- An optional estimate, `num_slots_hint`, of the number of slots in each slot set of the analysis. This can be used by the implementation to preallocate storage for data-flow results.

2.3 Solving a data-flow problem

In addition to its constructor, class `Bvd` defines the protected method `solve` for use by its subclasses, typically in their own constructor methods. `solve` initializes the slot sets associated with nodes and then pre-computes the net effect of each node as a flow function from slot sets to slot sets.

Next, `solve` computes the local flow functions that capture the effects of the individual nodes. A flow function is a slot-set transformer; it captures the total effect of a CFG node n in one data structure that is applied to *before*$[n]$, yielding *after*$[n]$. Starting with the plain identity function as the flow function for the node, `solve` scans its instructions in forward (backward) order for a forward (backward) problem. It first uses the *kill*$[i]$ set for instruction i to alter the flow function so that it removes the corresponding slots from the argument set. Then it uses *gen*$[i]$ to make the flow function insert the generated slots. For the entry (exit) node, it uses the `entry_flow` (`exit_flow`) parameter to accommodate boundary conditions, as discussed above.

Finally, `solve` runs the iterative propagation algorithm on the slot sets until it converges at a fixed point or until a caller provided iteration limit is reached. In the latter case, `solve` returns `false` to indicate abnormal termination.

2.4 Analyzing individual instructions

The remaining public member functions are pure virtual methods to be defined by the subclass for a specific data-flow problem. They handle the analysis of single instructions.

`find_kill_and_gen(`i`)`	Prepares to enumerate the *kill* and *gen* sets of instruction i.
`kill_set()`	Returns the *kill* set of the instruction analyzed by `find_kill_and_gen`.
`gen_set()`	Returns the *gen* set of the instruction analyzed by `find_kill_and_gen`.

When the flow function that represents the net effect of a node is being constructed, the solver calls `find_kill_and_gen` once on each instruction in the node. It then uses the `kill_set` and `gen_set` methods to obtain the results that `find_kill_and_gen` has found, i.e., to scan the slots killed by and those generated by the current instruction.

In the liveness example, the slots killed by an instruction are those for variables or registers that the instruction defines, i.e., writes to. The slots generated by an instruction are those for variables or registers that it uses. `find_kill_and_gen` determines the definition set and the use set for the instruction and `kill_set` and `gen_set` allow the solver to access them.

2.5 Implementation

Our implementation uses the following storage:

- A `FlowFun` for each node in the CFG, reclaimed after `solve`.
- A `NatSetDense` for the entry of each node.
- A `NatSetDense` for the exit of each node.

The non-public members of Bvd are declared as follows.

7a ⟨Bvd *details* 7a⟩≡ (4)

```
  protected:
    Cfg* _graph;
    FlowFun* _entry_flow;
    FlowFun* _exit_flow;
    int _num_slots_hint;

    direction _dir;
    confluence_rule _rule;

    Vector<FlowFun> _flow;             // slot set transformer for each node
    Vector<NatSetDense> _in;           // node-entry slot set for each node
    Vector<NatSetDense> _out;          // node-exit  slot set for each node

    Vector<NatSetDense> *_before;      // &_in  if forward, &_out if backward
    Vector<NatSetDense> *_after;       // &_out if forward, &_in  if backward

    void compute_local_flow(int);      // subroutines ...
    void combine_flow(CfgNode *);      //   ... of method solve
```

2.6 Header file solve.h

Class Bvd is defined in header file solve.h, which also declares the initialization and finalization functions for the BVD library:

7b ⟨*BVD library initialization* 7b⟩≡ (7c 15c) 15b▷

```
extern "C" void enter_bvd(int *argc, char *argv[]);
extern "C" void exit_bvd(void);
```

7c ⟨*solve.h* 7c⟩≡

```
/* file "bvd/solve.h" -- Iterative bit-vector data-flow solver */

⟨Machine-SUIF copyright 16c⟩

#ifndef BVD_SOLVE_H
#define BVD_SOLVE_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "bvd/solve.h"
#endif

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

#include <bvd/flow_fun.h>


⟨class Bvd 4⟩

⟨BVD library initialization 7b⟩

#endif /* BVD_SOLVE_H */
```

3 Flow functions

Muchnick [5] and other authors present the theoretical basis for data-flow problems in terms of *flow functions* (or sometimes *transfer functions*).

Here we define the three monotone *boolean* $\rightarrow$ *boolean* functions, extended pointwise to vectors of booleans, i.e., bit vectors. As in Section 2 we use the slot-set metaphor instead of talking in terms of bit vectors, and so these flow functions operate on values of type `NatSetDense`. Class `FlowFun` has a method `apply` that applies the flow function that it represents to a `NatSetDense` value. This allows us to encode operations on `NatSetDense`s directly from the theory.

The reason that we get away with using the theory in practice is that there are only three monotone boolean functions (the identity function, the function that always returns the constant $\top$ ("top", *true*, 1), and the function that always returns the constant $\perp$ ("bottom", *false*, 0). So we only need two bits to represent a monotone *boolean* $\rightarrow$ *boolean* function. Thus the pointwise extension of boolean functions to slot sets can be represented in only twice as much space as the sets that they manipulate.

3.1 Class FlowFun

8a ⟨*class* FlowFun 8a⟩≡ (10d)

```
class FlowFun {
  public:
    FlowFun(int num_slots_hint = 0);

    FlowFun(const FlowFun&);
    void operator=(const FlowFun&);

    void set_to_top();
    void set_to_top(int slot);
    void set_to_bottom();
    void set_to_bottom(int slot);
    void set_to_id();
    void set_to_id(int slot);

    void apply(NatSetDense &updated);

    void print(int bound, FILE* = stdout);

  ⟨FlowFun details 8b⟩
  };
```

When you create an instance of `FlowFun`, it is initialized to the identity function. The constructor takes an integer that the implementation can use as an estimate of slot-set sizes.

`set_to_top(n)`	Makes `this` a function that sets bit `n` to 1 ($\top$).
`set_to_top()`	Makes `this` a function that sets all bits to 1 ($\top$).
`set_to_bottom(n)`	Makes `this` a function that sets bit `n` to 0 ($\perp$).
`set_to_bottom()`	Makes `this` a function that sets all bits to 0 ($\perp$).
`set_to_id(n)`	Makes `this` a function that passes the value of bit `n` unchanged.
`set_to_id()`	Makes `this` a function that passes the values of all bits unchanged.
`apply(b)`	Applies `this` to set `b`, overwriting `b` with the result.

3.2 Implementation

As hinted earlier, we use two slot sets to represent a flow function.

8b ⟨FlowFun *details* 8b⟩≡ (8a)
```
  private:
    NatSetDense _id;
    NatSetDense _cs;
```

Roughly speaking, the presence or absence of s in set `_id` determines whether the flow function is the identity at slot s or always produces a constant ($\top$ or $\bot$) at that slot. In the latter case, the particular constant is determined by whether or not `_cs` contains s. Here are the precise rules:

	$s \notin$ `_cs`	$s \in$ `_cs`
$s \notin$ `_id`	$\bot$ at s	$\top$ at s
$s \in$ `_id`	*identity* at s	*(disallowed)*

Inlined methods. For efficiency, we define the methods that compute and apply flow functions as `inline` members.

9a ⟨FlowFun *inlines* 9a⟩≡ (10d) 9b ▷
```
inline void
FlowFun::set_to_top()
{
    // set function to "constant 1" on all bits
    _id.remove_all();
    _cs.insert_all();
}
```

9b ⟨FlowFun *inlines* 9a⟩+≡ (10d) ◁ 9a 9c ▷
```

inline void
FlowFun::set_to_top(int slot)
{
    claim(slot >= 0, "FlowFun::set_to_top - negative slot %d", slot);
    _id.remove(slot);
    _cs.insert(slot);
}
```

9c ⟨FlowFun *inlines* 9a⟩+≡ (10d) ◁ 9b 9d ▷
```

inline void
FlowFun::set_to_bottom()
{
    // set function to "constant 0" on all bits
    _id.remove_all();
    _cs.remove_all();
}
```

9d ⟨FlowFun *inlines* 9a⟩+≡ (10d) ◁ 9c 10a ▷
```

inline void
FlowFun::set_to_bottom(int slot)
{
    claim(slot >= 0, "FlowFun::set_to_bottom - negative slot %d", slot);
    _id.remove(slot);
    _cs.remove(slot);
}
```

10a ⟨FlowFun *inlines* 9a⟩+≡ (10d) ◁9d 10b▷

```
inline void
FlowFun::set_to_id()
{
    // set function to "identity" on all bits
    _id.insert_all();
    _cs.remove_all();
}
```

10b ⟨FlowFun *inlines* 9a⟩+≡ (10d) ◁10a 10c▷

```
inline void
FlowFun::set_to_id(int slot)
{
    _id.insert(slot);
    _cs.remove(slot);
}
```

To apply a flow function to a slot set, we subtract away any slots at which the function is constant and then union in the new constant slots. We rely on the representation invariant that `_cs` $\cap$ `_id` $= \emptyset$.

10c ⟨FlowFun *inlines* 9a⟩+≡ (10d) ◁10b

```
inline void
FlowFun::apply(NatSetDense &updated)
{
    updated *= _id;
    updated += _cs;
}
```

3.3 Header file `flow_fun.h`

Class `FlowFun` is defined in the following header file.

10d ⟨*flow_fun.h* 10d⟩≡

```
/* file "bvd/flow_fun.h" -- Flow functions */

⟨Machine-SUIF copyright 16c⟩

#ifndef BVD_FLOW_FUN_H
#define BVD_FLOW_FUN_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "bvd/flow_fun.h"
#endif

#include <machine/machine.h>

⟨class FlowFun 8a⟩

⟨FlowFun inlines 9a⟩

#endif /* BVD_FLOW_FUN_H */
```

4 Class Liveness

As mentioned in Section 2 the only elements needed to build a liveness analyzer on top of class Bvd are a mechanism for mapping the "interesting" operands of a program to slot numbers, and a way of enumerating the slots of operands defined and/or used by a particular instruction. The map from operands to slots is called an operand catalog. It has class OpndCatalog, and it is the subject of Section 5. The "def/use" analyzer for a single instruction is described later in this section.

4.1 Class Liveness

To perform liveness analysis, you simply construct an instance of class Liveness, passing a CFG, an operand catalog, and (optionally) a def/use analyzer as arguments to the constructor.

11 ⟨*class* Liveness 11⟩≡ (13)

```
class Liveness : public Bvd {
  public:
    Liveness(Cfg *graph, OpndCatalog *catalog, DefUseAnalyzer *analyzer)
        : Bvd(graph, backward, any_path)
    {
        _catalog = catalog;

        if (analyzer) {
            _analyzer_own = NULL; _analyzer = analyzer;
        } else
            _analyzer_own = _analyzer = new DefUseAnalyzer(catalog);

        solve();
    }

    virtual ~Liveness()                        { delete _analyzer_own; }

    virtual void find_kill_and_gen(Instr *mi)
        { _analyzer->analyze(mi); }

    virtual const NatSet* kill_set() const    { return _analyzer->defs_set(); }
    virtual const NatSet* gen_set()  const    { return _analyzer->uses_set(); }

    virtual int num_slots() const             { return _catalog->num_slots(); }

  protected:
    OpndCatalog* catalog()                    { return _catalog; }
    DefUseAnalyzer* analyzer()                { return _analyzer; }

  private:
    OpndCatalog *_catalog;
    DefUseAnalyzer *_analyzer;
    DefUseAnalyzer *_analyzer_own;            // default analyzer
};
```

The operand catalog, of type pointer to OpndCatalog, serves two purposes. It screens out the uninteresting operands, and it assigns slot numbers to the interesting one. You don't need to have enrolled any operands in the catalog before invoking the Liveness constructor (although you are free to do so). In the course of scanning the CFG, the solver enrolls interesting operands as needed. Then, after analysis is complete, you use the catalog to map operands to slot numbers so that you can access the data-flow results.

Note that the constructor and destructor and num_slots are the only public methods defined by Liveness. (Recall from Section 2 that num_slots returns number of slots in the final catalog after liveness has been computed.) To get at the analysis, you use the inherited methods `in_set` and `out_set`.

The protected methods are principally those that any subclass of `Bvd` must supply:

`find_kill_and_gen`(*i*)	Analyzes instruction *i* prior to use of `kill_iter` and `gen_iter`. For `Liveness`, this extracts the slots of operands defined (written) and operands used (read) by *i*.
`kill_iter()`	Produces an iterator over the *kill* set for the instruction most recently analyzed instruction by `find_kill_and_gen`. For `Liveness`, the *kill* set holds the slots of operands *defined.*
`gen_iter()`	Produces an iterator over the *gen* set for the instruction most recently analyzed instruction by `find_kill_and_gen`. For `Liveness`, the *gen* set holds the slots of operands *used.*

Since liveness is a backward data-flow problem, the solver built into class `Bvd` applies `find_kill_and_gen` to the instructions in a basic block by starting at the last one and working backward to the entry of the block. It starts with a "blank" flow function (one that would do nothing if applied to a slot set) and accumulates the block's net effect on liveness (for all slots) as it goes through the block. For each instruction, the operands defined are considered first; their liveness is "killed" because just before their definitions, their old values cannot be useful. The operands used by the instruction are considered next; they become live (their liveness is "generated") from the point of view of code preceding the current instruction. The flow function that results from this one pass over a node in the CFG captures its effect on liveness. The solver therefore has no further need to examine instructions as is solves the liveness problem for the whole procedure.

The `catalog` and `analyzer` methods of class `Liveness` simply return pointers to the operand catalog and the instruction def/use analyzer associated with an instance of the class.

Note that all of the code for `Liveness` is given in the above class declaration. All that's required to build a `Bvd`-based analyzer is to establish a map from "interesting data-flow items" to bit-vector slots, and to write a *gen*/*kill* analyzer for instructions.

4.2 Class `DefUseAnalyzer`

The task of identifying the data-flow items defined and used by a particular instruction is handled by an instance of class `DefUseAnalyzer`. This class handles explicit operands as well as implicit definitions and uses recorded in `regs_defd` and `regs_uses` annotations. The latter are used, for example, on call and return instructions for certain architectures to indicate registers used for argument and result transmission and also registers potentially defined because they are not protected by a callee-saves convention.

12 ⟨*class* `DefUseAnalyzer` 12⟩≡ (13)

```
class DefUseAnalyzer {
  public:
    DefUseAnalyzer(OpndCatalog *catalog) : _catalog(catalog) { }
    virtual ~DefUseAnalyzer() { }

    virtual void analyze(Instr *mi);

    NatSetIter defs_iter() const             { return _defs.iter(); }
    NatSetIter uses_iter() const             { return _uses.iter(); }

    const NatSet *defs_set() const           { return &_defs; }
    const NatSet *uses_set() const           { return &_uses; }

  protected:
    OpndCatalog *catalog()                   { return _catalog; }
    NatSet *defs()                           { return &_defs; }
    NatSet *uses()                           { return &_uses; }
```

```
  private:
    OpndCatalog *_catalog;
    NatSetSparse _defs;
    NatSetSparse _uses;
};
```

The only input needed to create a `DefUseAnalyzer` is the operand catalog. When the `analyze` method is applied to an instruction, it attempts to enroll each operand of the instruction in the catalog. This has the effects of excluding uninteresting operands, of enrolling interesting operands in the catalog, and of providing slot numbers that can be used to make up def/use sets. The `analyze` method builds these sets for the instruction. The the other public methods can be used to access them, either directly as sets (methods `defs_set` and `uses_set`) or as iterations (methods `defs_iter` and `uses_iter`).

13 ⟨*liveness.h* 13⟩≡

```
/* file "bvd/liveness.h" -- Liveness analyzer */

⟨Machine-SUIF copyright 16c⟩

#ifndef BVD_LIVENESS_H
#define BVD_LIVENESS_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "bvd/liveness.h"
#endif

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

#include <bvd/solve.h>


⟨class DefUseAnalyzer 12⟩

⟨class Liveness 11⟩

#endif /* BVD_LIVENESS_H */
```

5 Mapping operands to bit-vector slots

Solvers of bit-vector data-flow problems need to associate an integer identifier with each item in the universe whose data flow they calculate. As mentioned earlier, we call these unique integers *slots* because that is the term sometimes used for bit-vector positions. In the liveness problem, the items of interest are operands, so the liveness solver needs a map from operands to slots. The purpose of an *operand catalog* is to provide this map.

As the program is scanned during analysis, the catalog is used to assign slots to operands. Later, as operands are revisited, the catalog provides efficient lookup of their slot numbers so that data-flow information can be read out of the slot sets produced by analysis.

5.1 Class `RegSymCatalog`

The `machine` library (see *The SUIF Machine Library* document) defines class `OpndCatalog` which provides a common abstract interface for operand catalogs. One very useful concrete subclass of `OpndCatalog` implements a catalog for register and variable-symbol operands. This class, called `RegSymCatalog` is heavily used in register allocation, scalar optimization, and instruction scheduling.

14a ⟨*class* `RegSymCatalog` 14a⟩≡ (15a)

```
class RegSymCatalog : public OpndCatalog {
  public:
    typedef bool (*filter_f)(Opnd);

    RegSymCatalog(bool record = false, filter_f = NULL,
                  int hash_table_size = 1024);

    virtual int size() const { return _next_slot; }

    virtual bool enroll(Opnd, int *slot = NULL);
    virtual bool lookup(Opnd, int *slot = NULL) const;

⟨RegSymCatalog details 14b⟩
};
```

Constructing a `RegSymCatalog`. The constructor for class `RegSymCatalog` declared above takes the following arguments:

- An optional flag indicating whether operands enrolled should be remembered so that the catalog can be printed.
- An optional filter, a Boolean function taking an operand as argument. If the filter is provided, the catalog ignores any operand for which the filter returns `false`.
- An optional bucket count for the catalog's hash table. This must be a power of two. Overriding the default value may affect performance, but not the table's capacity.

5.2 Implementation details

5.2.1 Class `RegSymCatalog`

The private part of `RegSymCatalog` declares the user-provided operand filter and the running slot count.

14b ⟨`RegSymCatalog` *details* 14b⟩≡ (14a) 14c ▷

```
    filter_f _filter;
    int _next_slot;
```

It also records the hash map taking operands to slots:

14c ⟨`RegSymCatalog` *details* 14b⟩+≡ (14a) ◁ 14b 14d ▷

```
    HashMap<unsigned long, int> _hash_map;
```

The function member is simply a subroutine of the public methods.

14d ⟨`RegSymCatalog` *details* 14b⟩+≡ (14a) ◁ 14c

```
    bool get_slot(Opnd, int*, bool);
```

5.3 Header file for module catalog

Classes OpndCatalog and RegSymCatalog are defined in module catalog, which has the following header file:

15a ⟨*catalog.h* 15a⟩≡
```
/* file "bvd/catalog.h" -- Maps from operands to slot sets */

⟨Machine-SUIF copyright 16c⟩

#ifndef BVD_OPND_CATALOG_H
#define BVD_OPND_CATALOG_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "bvd/catalog.h"
#endif

#include <machine/machine.h>

⟨class RegSymCatalog 14a⟩

#endif /* BVD_OPND_CATALOG_H */
```

6 Library initialization

Before you can start using the facilities of the BVD library, the library must initialize some parts of itself. In SUIF, this is performed by defining an init_*libname* routine. The respective declarations are:

15b ⟨*BVD library initialization* 7b⟩+≡ (7c 15c) ◁7b
```
extern "C" void init_bvd(SuifEnv*);
```

The BVD library initialization header file has the following simple layout:

15c ⟨*init.h* 15c⟩≡
```
/* file "bvd/init.h" */

⟨Machine-SUIF copyright 16c⟩

#ifndef BVD_INIT_H
#define BVD_INIT_H

#include <machine/copyright.h>

#ifndef SUPPRESS_PRAGMA_INTERFACE
#pragma interface "bvd/init.h"
#endif

#include <machine/machine.h>

⟨BVD library initialization 7b⟩

#endif /* BVD_INIT_H */
```

When you extend the library, you may need to add initialization and or finalization actions to the corresponding implementation file `init.cpp`.

7 Header file for the BVD library

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

16a ⟨*bvd.h* 16a⟩≡
```
/* file "bvd/bvd.h" */

⟨Machine-SUIF copyright 16c⟩

#ifndef BVD_BVD_H
#define BVD_BVD_H

#include <machine/copyright.h>

⟨contents of bvd.h 16b⟩

#endif /* BVD_BVD_H */
```

16b ⟨*contents of* `bvd.h` 16b⟩≡ (16a)
```
#include <bvd/flow_fun.h>
#include <bvd/catalog.h>
#include <bvd/solve.h>
#include <bvd/liveness.h>
#include <bvd/reaching_defs.h>
#include <bvd/init.h>
```

8 Copyright

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

16c ⟨*Machine-SUIF copyright* 16c⟩≡ (7c 10d 13 15 16a)

9 Acknowledgments

This work is supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225) and a NSF Young Investigator award (CCR-9457779). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, 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] C. Fischer and R. LeBlanc. *Crafting a Compiler.* Benjamin/Cummings, 1988.

[3] Glenn H. Holloway and Michael D. Smith. *The SUIF Machine Library.* The Machine SUIF documentation set, Harvard University, 1998.

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

[5] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.