15-745 Lecture 2

Programming in SUIF

15-745 Lecture 2

Background (SUIF1)

- SUIF: Stanford University Intermediate Format
- Goal: Develop a research compiler infrastructure
  - Easy to modify
  - Easy to augment
  - (subgoal) Enforce modularity
  - Speed not an issue
- Reality:
  - Sometimes easy, sometimes hard
    - Extending the IR was VERY hard!
  - Even for research, speed is an issue

SUIF2

- Goals:
  - Better support for modifying the IR
  - Faster
  - Reflective IR
  - Tools for adding to IR
  - Support modules & performance
    - i.e., no disk accesses between passes

The SUIF System

PGI Fortran  EDG C  EDG C++  Java

Interprocedural Analysis
Parallelization
Locality Opt

OSUIF

SUIF2

C  MachSUIF

Alpha  x86

* C++ OSUIF to SUIF is incomplete

From Lam, PLDI workshop
MACHSUIF

- Machine SUIF developed at Harvard
- A backend for SUIF2
- Goals:
  - Machine specific optimizations
  - Support architecture research
  - Easy to extend
  - Support multiple targets
  - 1-to-1 correspondence
  - Portable across different compilation environments

Compiler Phases

SUIF2 - Modular

- Executable
- suifdriver
- IR
- suifnodes basicnodes
- Kernel
- suifkernel iokernel

SUIF2 - 2 Methods

- A series of stand-alone programs
  - Suif-file1
  - driver+module1
  - Suif-file2
  - driver+module2
  - Suif-file3
  - driver+module3
  - Suif-file4

- A driver that imports & applies modules to program in memory
- Suifdriver
- imports executes
- module1 module2
- module3
- Suif-file4

From Lam, PLDI workshop
IR Structure

- **FileSetBlock**: the main container for files and external symbol tables
  - **FileBlock**: represents a source file: file scope symbol tables
  - **ProcedureDefinition**

Logistics

- Can run on Linux (2.2) or Alpha (color machine)
- To get setup you should execute:
  - `cs745=/afs/cs/academic/classes/15745-s03`
  - `eval `$/cs745/public/bin/setup-suif-env -sh``
- Now a ton of environment vars are set
- You can cross-compile
  - which may be useful when debugging new passes

Logistics

- Running the Front-end/Generating SUIF2 IR
  - Linux: `c2s foo.c`
  - Alpha: `c2sbyl foo.c`
  - Run this on the target machine for cpp
- **Converting High-SUIF to Low-SUIF**
  - `do_lower foo.sui foo.lsf`
- **Converting to MACHSUIF IR/SUIFVM**
  - `do_svm foo.lsf foo.svm`
- ...
  - Eventually,
    - `gcc -o foo foo.s`

Different Targets

- Different target machines have
  - different opcodes
  - different reg files, conventions, etc.
- MACHSUIF approach
  - early code generation
  - parameterize passes
  - suifvm
- Logistics
  - `do_gen -target_lib X`
  - `MACHSUIF_TARGET_LIB=X`
  - `X can currently be: alpha or x86`
Different Compiler Environments

- Support different compiler substrates without having to rewrite code
  - SUIF static compilation
  - DECO dynamic compilation

- Optimization Programming Interface (OPI)
  - Defines an interface that can manipulate IR in a target-machine specific manner.
  - Defines containers:
    - lists of instructions
    - control flow graph
  - Relies on substrate for I/O, etc.

Existing Passes

- s2m: convert suif to suifvm
- gen: convert suifvm to target dialect
- raga: register allocation
- dce: dead code elimination
- fin: finalize (frame layout), proc entry/exit
- il2cfg: instruction lists to cfg
- cfg2il: cfg to instruction lists
- m2a: create .s file
- m2c: create .c file

Ordering

```
  s2m
    |   |
    +---+---+---+
    |   |   |   |
    +---+---+---+
        |   |   |
        +---+---+
          |   |
          +---+
            |
            il2cfg

  peep
  dce
  m2a
  m2c

  gen  raga  fin
```

Using OPI to Create a Pass

- Two parts to every pass:
  - Substrate independent part
  - Wrapper
- Independent part performs the optimization
- Wrapper
  - Binds the pass to the target (if necessary)
  - Links pass to substrate
Independent Part

- Each substrate-independent part is a class
- **Class defines** at least three methods:
  - `initialize`: code run before the pass
  - `do_opt_unit`: perform the pass
  - `finalize`: code to run afterwards
- Utilizes OPI functions and libraries
  - `CFG`: control flow graphs
  - `CFA`: control flow analysis
  - `BVD`: bit vector based dataflow

---

Wrapper

- Wraps the stand-alone class in a subclass of `PipelineablePass`

- `PipelineablePass` deals with:
  - command line parsing
  - file I/O
  - iterating on files, procedures, etc.

- If the pass is parameterized (i.e., must treat different targets differently)
  - call `focus()` for each `FileBlock` and `ProcedureDefinition`

---

```cpp
class Peep {
public:
  Peep() {
    void initialize();
    void do_opt_unit(OptUnit*);
    void finalize();
  ...

  OptUnit (in SUIF environment) is a ProcedureDefinition

  Peep::do_file_set_block(FileSetBlock *fsb) {
    set_opti_predefined_types(fsb);
  }

  Peep::do_file_block(FileBlock *fb) {
    claim(has_note(fb, k_target_lib),
      "expected target_lib annotation on file block");
    focus(fb);
    peep.initialize();
  }

  Peep::do_procedure_definition(ProcedureDefinition *pd) {
    focus(pd);
    peep.do_opt_unit(pd);
    defocus(pd);
  }

  Peep::finalize() {
    peep.finalize();
  }
};
```
OPI Data Structures

- Reference or Value Semantics?
  - It is made explicit
    - (Be safe and do explicit copies)
- Basics:
  - constants, symbols, and types
- Instructions
- Operands
- Containers
- Annotations

Instructions

- \texttt{Instr*} is type of every instruction.
- All contain an \texttt{Opcode}.
  - \texttt{Opcode} value is target specific
- Classes of instructions:
  - Active
    - \texttt{alm (arith, logical, memory)}
    - \texttt{cti (control transfer)}
  - Inactive
    - \texttt{label}
    - \texttt{dot (pseudo-ops)}
- Constructors (\texttt{new\_instr\_alm})
- Predicates (\texttt{is\_alm})

Operands

- Instructions can contain operands of type \texttt{Opnd}.
- All Operands
  - have a type (\texttt{get\_type(opnd)})
  - a kind (\texttt{get\_kind(opnd), is\_reg(opnd)})
  - can be copied (\texttt{clone(opnd)})
  - support \texttt{==} and \texttt{!=}
  - can be hashed (\texttt{hash(opnd)})
  - can be printed (\texttt{print(opnd)})
- Operands come in many flavors:
  - \texttt{Null} \hspace{1cm} \texttt{is\_null}
  - Variable symbols \hspace{1cm} \texttt{is\_var}
  - Registers \hspace{1cm} \texttt{is\_reg}
  - Immediate \hspace{1cm} \texttt{is\_immed}
  - Address \hspace{1cm} \texttt{is\_addr\_sym}
  - Address expression \hspace{1cm} \texttt{is\_addr\_exp}

Register Operands

- Hard or Virtual
  - Virtual Register:
    - Creation: \texttt{opnd\_reg(type)}
    - Testing: \texttt{is\_virtual\_reg(opnd)}
  - Hard Register
    - Creation: \texttt{opnd\_reg(num, type)}
    - Testing: \texttt{is\_hard\_ref(opnd)}
    - Getting Number: \texttt{get\_reg(r)}
Variable Symbols

- `get_var(v)` returns the `VarSym*` for this operand
- `VarSym*` contains information about the symbol:
  - scope
  - definition
  - type
  - whether address is taken, etc.

Example

```c
bool is_mortal(Opnd opnd)
{
  if (is_reg(opnd))
    return true;
  if (is_var(opnd)) {
    VarSym *vs = get_var(opnd);
    return (!is_addr_taken(vs) && is_auto(vs));
  }
  return false;
}
```

Sequences

- `InstrList`, `CfgNode` (contain instructions)
- `Instr` (contains operands)
- Many functions based on position and handles
  - `size`
  - `start`, `last`, `end`
  - `get_x(container, int)`
  - `get_x(container, handle)`
  - `++`, `--`
  - prepend, append, ...

CFG

- `nodes_size(cfg)`
- `nodes_start(cfg)`
- `nodes_last(cfg)`
- `nodes_end(cfg)`
- `get_node(cfg, integer)`
- `get_node(cfg, handle)`
- `get_entry_node(cfg)`
- `get_exit_node(cfg)`
- Adding nodes is handled separately
  - e.g., `insert_empty_node(cfg, tail, head)`
Iterating Example

```c
void Peep::do_opt_unit(OptUnit *unit)
{
    claim(is_kind_of<Cfg>(get_body(unit)),
        "Body is not in CFG form");
    unit_cfg = static_cast<Cfg*>(get_body(unit));
    for (int i = 0; i < nodes_size(unit_cfg); ++i) {
        CfgNode *b = get_node(unit_cfg, i);
        ...
    }
}
```

What's important about a CFG Node?

- Instructions it contains
  - e.g., instrs_size(node)
- Predecessors and Successors
  - e.g., preds(node)
- How it ends?
  - get_cti_handle(node)
  - ends_in_ubr(node)

Iterating Example

```c
void Peep::do_opt_unit(OptUnit *unit)
{
    claim(is_kind_of<Cfg>(get_body(cur_unit)),
        "Body is not in CFG form");
    unit_cfg = static_cast<Cfg*>(get_body(unit));
    for (int i = 0; i < nodes_size(unit_cfg); ++i) {
        CfgNode *b = get_node(unit_cfg, i);
        Instr *inst;
        InstrHandle ih = last(b);
        int bsize = size(b);
        for (int i = 0; i < bsize; ) {
            inst = *ih;
            ++i, --ih;  // scan instructions backwards
        }
    }
}
```

Some Predefined Optimizations

- remove_unreachable_nodes
- optimize_jumps
Annotations

- Allow any information to be added to (almost) any node.
- They are persistent
  - key, value pair
    - key is of type NoteKey
- Kinds of annotations:
  - flag annotations
    - existence is what matters; no key
  - singleton annotations
    - value is of type: long, Integer, IdString, IrObject*
      (IrObject are all ptr types. NOT: Opnd)
  - list annotations, custom annotations

There Is More

- Read: Overview and OPI users guide.
- Do HW1