15-745 Lecture 2

Programming in SUIF

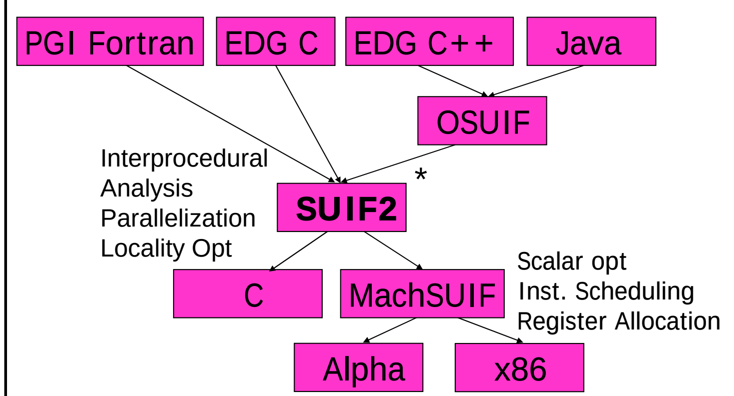
Background (SUIF1)

- 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



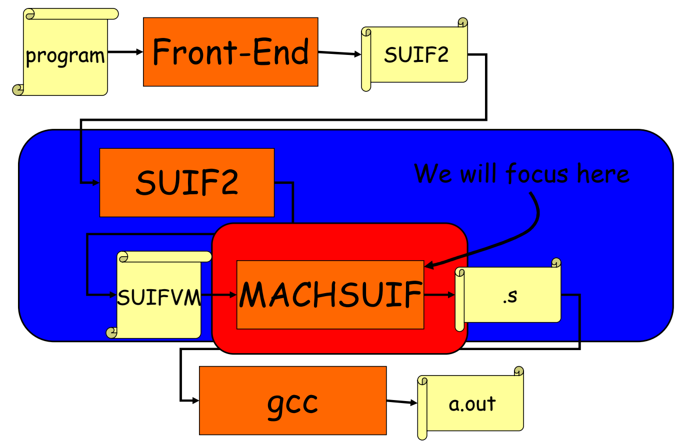
* C++ OSUIF to SUIF is incomplete

From Lam, PLDI workshop

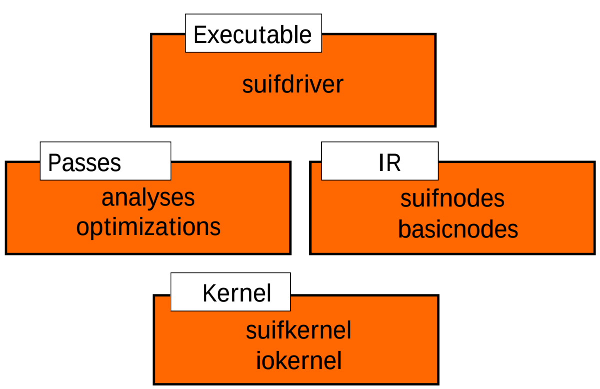
MACHSUIF

- 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

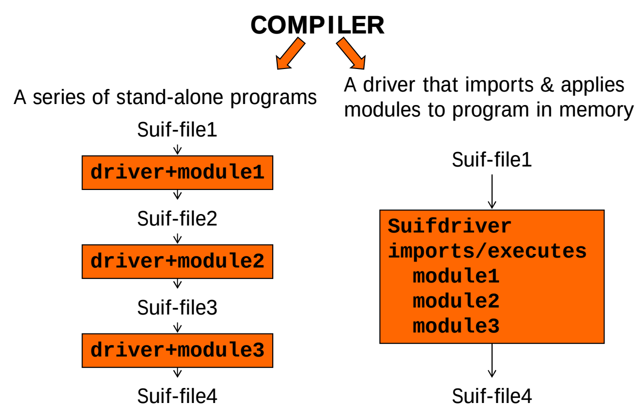


SUIF2 - Modular



From Lam, PLDI workshop

SUIF2 – 2 Methods



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
- To get setup you should execute:
  - cs745=/afs/cs/academic/classes/15745-f03
  - 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` } Creates `foo.suif`
  - Alpha: `c2sby1 foo.c` } Creates `foo.suif`
  - Run this on the target machine for cpp
- Converting High-SUIF to Low-SUIF
  - `do_lower foo.suif 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`

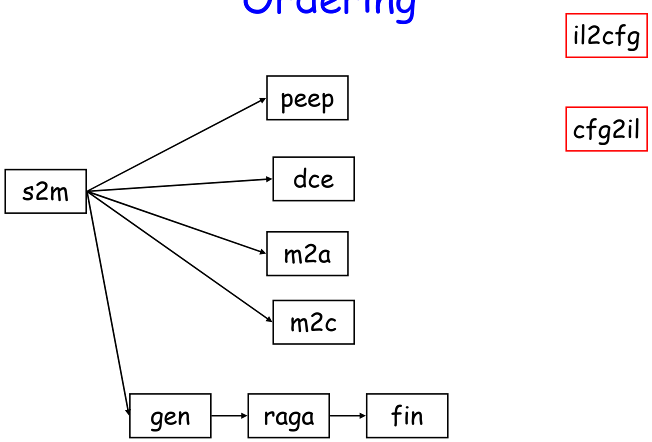
Different Compiler Environments

- 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



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**


```
void
PeepSuifPass::do_file_set_block(FileSetBlock *fsb) {
    set_opi_predefined_types(fsb);
}

PeepSuifPass::do_file_block(FileBlock *fb) {
    claim(has_note(fb, k_target_lib),
          "expected target_lib annotation on file block");

    focus(fb);
    peep.initialize();
}

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

PeepSuifPass::finalize() {
    peep.finalize();
}
```


Defining Pass Peep

```
class Peep {
  public:
    Peep() { }

    void initialize();
    void do_opt_unit(OptUnit*);
    void finalize();

    …
}
```

OptUnit (in SUIF environment) is a ProcedureDefinition

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

- **`Instr*`** is type of every instruction.
- All contain an Opcode.
  - Opcode value is target specific
- Classes of instructions:
  - Active
    - alm (arith, logical, memory)
    - cti (control transfer)
  - Inactive
    - label
    - dot (pseudo-ops)
- Constructors (**`new_instr_alm`**)
- Predicates (**`is_alm`**)

Operands

- Instructions can contain operands of type **`Opnd`**.
- All Operands
  - have a type (`get_type(opnd)`)
  - a kind (`get_kind(opnd)`, `is_reg(opnd)`)
  - can be copied (`clone(opnd)`)
  - support `==` and `!=`
  - can be hashed (`hash(opnd)`)
  - can be printed (`fprint(opnd)`)
- Operands come in many flavors:

Null	is_null
Variable symbols	is_var
Registers	is_reg
Immediate	is_immed
Address	is_addr_sym
Address expression	is_addr_exp

Register Operands

- Hard or Virtual
- Virtual Register:
  - Creation: **`opnd_reg(type)`**
  - Testing: **`is_virtual_reg(opnd)`**
- Hard Register
  - Creation: **`opnd_reg(num, type)`**
  - Testing: **`is_hard_ref(opnd)`**
  - Getting Number: **`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

```
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

```
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

```
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