sparse Documentation

Release 0.6.0

sparse's development community

Oct 13, 2019

Contents

CHAPTER 1

User documentation

1.1 __nocast vs __bitwise

__nocast warns about explicit or implicit casting to different types. HOWEVER, it doesn't consider two 32-bit integers to be different types, so a __nocast int type may be returned as a regular int type and then the __nocast is lost.

So __nocast on integer types is usually not that powerful. It just gets lost too easily. It's more useful for things like pointers. It also doesn't warn about the mixing: you can add integers to __nocast integer types, and it's not really considered anything wrong.

__bitwise ends up being a *stronger integer separation*. That one doesn't allow you to mix with non-bitwise integers, so now it's much harder to lose the type by mistake.

So the basic rule is:

- __nocast on its own tends to be more useful for *big* integers that still need to act like integers, but you want to make it much less likely that they get truncated by mistake. So a 64-bit integer that you don't want to mistakenly/silently be returned as int, for example. But they mix well with random integer types, so you can add to them etc without using anything special. However, that mixing also means that the __nocast really gets lost fairly easily.
- __bitwise is for *unique types* that cannot be mixed with other types, and that you'd never want to just use as a random integer (the integer 0 is special, though, and gets silently accepted - it's kind of like NULL for pointers). So gfp_t or the safe endianness types would be __bitwise: you can only operate on them by doing specific operations that know about *that* particular type.

Generally, you want __bitwise if you are looking for type safety. __nocast really is pretty weak.

1.1.1 Reference:

- Linus' e-mail about __nocast vs __bitwise:

  https://marc.info/?l=linux-mm&m=133245421127324&w=2

CHAPTER 2

Developer documentation

2.1 Test suite

Sparse has a number of test cases in its validation directory. The test-suite script aims at making automated checking of these tests possible. It works by embedding tags in C comments in the test cases.

2.1.1 Tag's syntax

`check-name:` *name*

Name of the test. This is the only mandatory tag.

`check-description:` *description ...*

A description of what the test checks.

`check-command:` *command arg ...*

There are different kinds of tests. Some can validate the sparse preprocessor, while others will use sparse, cgcc, or even other backends of the library. check-command allows you to give a custom command to run the test-case. The `$file` string is special. It will be expanded to the file name at run time. It defaults to `sparse $file`.

`check-arch-ignore:` *arch[|...]*

`check-arch-only:` *arch[|...]*

Ignore the test if the current architecture (as returned by `uname -m`) matches or not one of the archs given in the pattern.

`check-assert:` *condition*

Ignore the test if the given condition is false when evaluated as a static assertion (`_Static_assert`).

`check-cpp-if:` *condition*

Ignore the test if the given condition is false when evaluated by sparse's pre-processor.

`check-exit-value:` *value*

The expected exit value of check-command. It defaults to 0.

`check-timeout: timeout`

The maximum expected duration of check-command, in seconds. It defaults to 1.

`check-output-start` / `check-output-end`

The expected output (stdout and stderr) of check-command lies between those two tags. It defaults to no output.

`check-output-ignore` / `check-error-ignore`

Don't check the expected output (stdout or stderr) of check-command (useful when this output is not comparable or if you're only interested in the exit value). By default this check is done.

`check-known-to-fail`

Mark the test as being known to fail.

`check-output-contains: pattern`

Check that the output (stdout) contains the given pattern. Several such tags can be given, in which case the output must contains all the patterns.

`check-output-excludes: pattern`

Similar than the above one, but with opposite logic. Check that the output (stdout) doesn't contain the given pattern. Several such tags can be given, in which case the output must contains none of the patterns.

`check-output-pattern(nbr): pattern`

`check-output-pattern(min,max): pattern`

Similar to the contains/excludes above, but with full control of the number of times the pattern should occur in the output. If *min* or *max* is − the corresponding check is ignored.

2.1.2 Using test-suite

The test-suite script is called through the check target of the Makefile. It will try to check every test case it finds (`find validation -name '*.c'`). It can be called to check a single test with:

```
$ cd validation
$ ./test-suite single preprocessor/preprocessor1.c
     TEST     Preprocessor #1 (preprocessor/preprocessor1.c)
preprocessor/preprocessor1.c passed !
```

2.1.3 Writing a test

The test-suite comes with a format command to make a test easier to write:

```
test-suite format [-a] [-l] [-f] file [name [cmd]]
```

name: **check-name value** If no name is provided, it defaults to the file name.

cmd: **check-command value** If no cmd is provided, it defaults to `sparse $file`.

The output of the test-suite format command can be redirected into the test case to create a test-suite formatted file.:

```
$ ./test-suite format bad-assignment.c Assignment >> bad-assignment.c
$ cat !$
cat bad-assignment.c
/*
 * check-name: bad assignment
 *
```

(continues on next page)

(continued from previous page)

```
 * check-command: sparse $file
 * check-exit-value: 1
 *
 * check-output-start
bad-assignment.c:3:6: error: Expected ; at end of statement
bad-assignment.c:3:6: error: got \
 * check-output-end
 */
```

The same effect without the redirection can be achieved by using the `-a` option.

You can define the check-command you want to use for the test.:

```
$ ./test-suite format -a validation/preprocessor2.c "Preprocessor #2" \
                "sparse -E \$file"
$ cat !$
cat validation/preprocessor2.c
/*
 * This one we happen to get right.
 *
 * It should result in a simple
 *
 *      a + b
 *
 * for a proper preprocessor.
 */
#define TWO a, b

#define UNARY(x) BINARY(x)
#define BINARY(x, y) x + y

UNARY(TWO)
/*
 * check-name: Preprocessor #2
 *
 * check-command: sparse -E $file
 * check-exit-value: 0
 *
 * check-output-start

a + b
 * check-output-end
 */
```

2.2 sparse - extra options for developers

2.2.1 SYNOPSIS

`tools` [*options*]... *file.c*‘

2.2.2 DESCRIPTION

This file is a complement of sparse’s man page meant to document options only useful for development on sparse itself.

2.2.3 OPTIONS

-fdump-ir=pass,[pass]
Dump the IR at each of the given passes.

The passes currently understood are:

- `linearize`
- `mem2reg`
- `final`

The default pass is `linearize`.

-f<name-of-the-pass>[-disable|-enable|=last]
If `=last` is used, all passes after the specified one are disabled. By default all passes are enabled.

The passes currently understood are:

- `linearize` (can't be disabled)
- `mem2reg`
- `optim`

-vcompound
Print all compound global data symbols with their sizes and alignment.

-vdead
Add `OP_DEATHNOTE` annotations to dead pseudos.

-vdomtree
Dump the dominance tree after its calculation.

-ventry
Dump the IR after all optimization passes.

-vpostorder
Dump the reverse postorder traversal of the CFG.

2.3 Sparse API

2.3.1 Utilities

Pointer list manipulation

int **ptr_list_size**(struct ptr_list **head*)
Get the size of a ptrlist.

Parameters

- **head** – the head of the list

Returns the size of the list given by **head**.

bool **ptr_list_empty**(const struct ptr_list **head*)
Test if a list is empty.

Parameters

- **head** – the head of the list

Returns `true` if the list is empty, `false` otherwise.

bool **ptr_list_multiple**(const struct ptr_list **head*)
Test is a list contains more than one element.

Parameters

- **head** – the head of the list

Returns `true` if the list has more than 1 element, `false` otherwise.

void ***first_ptr_list**(struct ptr_list **head*)
Get the first element of a ptrlist.

Parameters

- **head** – the head of the list

Returns the first element of the list or `NULL` if the list is empty

void ***last_ptr_list**(struct ptr_list **head*)
Get the last element of a ptrlist.

Parameters

- **head** – the head of the list

Returns the last element of the list or `NULL` if the list is empty

void ***ptr_list_nth_entry**(struct ptr_list **list*, unsigned int *idx*)
Get the nth element of a ptrlist.

Parameters

- **head** – the head of the list

Returns the nth element of the list or `NULL` if the list is too short.

int **linearize_ptr_list**(struct ptr_list **head*, void ***arr*, int *max*)
Linearize the entries of a list.

Parameters

- **head** – the list to be linearized
- **arr** – a `void*` array to fill with **head**'s entries
- **max** – the maximum number of entries to store into **arr**

Returns the number of entries linearized.

Linearize the entries of a list up to a total of **max**, and return the nr of entries linearized.

The array to linearize into (**arr**) should really be `void *x[]`, but we want to let people fill in any kind of pointer array, so let's just call it `void **`.

void **pack_ptr_list**(struct ptr_list **listp*)
Pack a ptrlist.

Parameters

- **listp** – a pointer to the list to be packed.

When we've walked the list and deleted entries, we may need to re-pack it so that we don't have any empty blocks left (empty blocks upset the walking code).

void **split_ptr_list_head**(struct ptr_list *head)
Split a ptrlist block.

Parameters

- **head** – the ptrlist block to be splitted

A new block is inserted just after **head** and the entries at the half end of **head** are moved to this new block. The goal being to create space inside **head** for a new entry.

void ****__add_ptr_list**(struct ptr_list **listp*, void *ptr)
Add an entry to a ptrlist.

Parameters

- **listp** – a pointer to the list
- **ptr** – the entry to add to the list

Returns the address where the new entry is stored.

Note code must not use this function and should use `add_ptr_list()` instead.

void ****__add_ptr_list_tag**(struct ptr_list **listp*, void *ptr, unsigned long *tag*)
Add a tagged entry to a ptrlist.

Parameters

- **listp** – a pointer to the list
- **ptr** – the entry to add to the list
- **tag** – the tag to add to **ptr**

Returns the address where the new entry is stored.

Note code must not use this function and should use `add_ptr_list_tag()` instead.

bool **lookup_ptr_list_entry**(const struct ptr_list *head, const void *entry)
Test if some entry is already present in a ptrlist.

Parameters

- **list** – the head of the list
- **entry** – the entry to test

Returns `true` if the entry is already present, `false` otherwise.

int **delete_ptr_list_entry**(struct ptr_list **list*, void *entry, int *count*)
Delete an entry from a ptrlist.

Parameters

- **list** – a pointer to the list
- **entry** – the item to be deleted
- **count** – the minimum number of times **entry** should be deleted or 0.

int **replace_ptr_list_entry**(struct ptr_list **list*, void *old_ptr, void *new_ptr, int *count*)
Replace an entry in a ptrlist.

Parameters

- **list** – a pointer to the list
- **old_ptr** – the entry to be replaced
- **new_ptr** – the new entry
- **count** – the minimum number of times **entry** should be deleted or 0.

void * **undo_ptr_list_last** (struct ptr_list **head*)
Remove the last entry of a ptrlist.

Parameters

- **head** – a pointer to the list

Returns the last elemant of the list or NULL if the list is empty.

Note this doesn't repack the list

void * **delete_ptr_list_last** (struct ptr_list **head*)
Remove the last entry and repack the list.

Parameters

- **head** – a pointer to the list

Returns the last elemant of the list or NULL if the list is empty.

void **concat_ptr_list** (struct ptr_list **a*, struct ptr_list ***b*)
Concat two ptrlists.

Parameters

- **a** – the source list
- **b** – a pointer to the destination list.

The element of **a** are added at the end of **b**.

void **copy_ptr_list** (struct ptr_list ***listp*, struct ptr_list **src*)
Copy the elements of a list at the end of another list.

Parameters

- **listp** – a pointer to the destination list.
- **src** – the head of the source list.

void **__free_ptr_list** (struct ptr_list ***listp*)
Free a ptrlist.

Parameters

- **listp** – a pointer to the list

Each blocks of the list are freed (but the entries themselves are not freed).

Note code must not use this function and should use the macro `free_ptr_list()` instead.

Miscellaneous utilities

void ***xmemdup** (const void **src*, size_t *len*)
Duplicate a memory buffer in a newly allocated buffer.

Parameters

- **src** – a pointer to the memory buffer to be duplicated
- **len** – the size of the memory buffer to be duplicated

Returns a pointer to a copy of **src** allocated via `__alloc_bytes()`.

char *__xstrdup__(const char *_src_)
Duplicate a null-terminated string in a newly allocated buffer.

Parameters

- **src** – a pointer to string to be duplicated

Returns a pointer to a copy of **str** allocated via `__alloc_bytes()`.

char *__xasprintf__(const char *_fmt_, ...)
Printf to allocated string.

Parameters

- **fmt** – the format followed by its arguments.

Returns the allocated & formatted string.

This function is similar to asprintf() but the resulting string is allocated with __alloc_bytes().

char *__xvasprintf__(const char *_fmt_, va_list _ap_)
Vprintf to allocated string.

Parameters

- **fmt** – the format
- **ap** – the variadic arguments

Returns the allocated & formatted string.

This function is similar to asprintf() but the resulting string is allocated with __alloc_bytes().

2.3.2 Parsing

Constant expression values

int __is_zero_constant__(struct expression *_expr_)
Test if an expression evaluates to the constant `0`.

Returns `1` if **expr** evaluate to `0`, `0` otherwise.

int __expr_truth_value__(struct expression *_expr_)
Test the compile time truth value of an expression.

Returns

- `-1` if **expr** is not constant,
- `0` or `1` depending on the truth value of **expr**.

2.3.3 Typing

struct symbol *__evaluate_expression__(struct expression *_expr_)
Evaluate the type of an expression.

Parameters

- **expr** – the expression to be evaluated

Returns the type of the expression or `NULL` if the expression can't be evaluated

struct symbol *__evaluate_statement__(struct statement *_stmt_)
Evaluate the type of a statement.

Parameters

- **stmt** – the statement to be evaluated

Returns the type of the statement or `NULL` if it can't be evaluated

void **evaluate_symbol_list** (struct symbol_list **list*)
Evaluate the type of a set of symbols.

Parameters

- **list** – the list of the symbol to be evaluated

2.3.4 Optimization

Instruction simplification

Notation

The following conventions are used to describe the simplications:

- Uppercase letters are reserved for constants:
  - *M* for a constant mask,
  - *S* for a constant shift,
  - *N* for a constant number of bits (usually other than a shift),
  - *C* or 'K' for others constants.
- Lowercase letters *a*, *b*, *x*, *y*, ... are used for non-constants or when it doesn't matter if the pseudo is a constant or not.
- Primes are used if needed to distinguish symbols (*M*, *M'*, ...).
- Expressions or sub-expressions involving only constants are understood to be evaluated.
- *$mask(N)* is used for *((1 << N) -1)*
- *$trunc(x, N)* is used for *(x & $mask(N))*
- Expressions like *(-1 << S)*, *(-1 >> S)* and others formulae are understood to be truncated to the size of the current instruction (needed, since in general this size is not the same as the one used by sparse for the evaluation of arithmetic operations).
- *TRUNC(x, N)* is used for a truncation *to* a size of *N* bits
- *ZEXT(x, N)* is used for a zero-extension *from* a size of *N* bits
- *OP(x, C)* is used to represent some generic operation using a constant, including when the constant is implicit (e.g. *TRUNC(x, N)*).
- *MASK(x, M)* is used to respresent a 'masking' instruction:
  - *AND(x, M)*
  - *LSR(x, S)*, with *M* = (-1 << S)
  - *SHL(x, S)*, with *M* = (-1 >> S)
  - *TRUNC(x, N)*, with *M* = $mask(N)
  - *ZEXT(x, N)*, with *M* = $mask(N)
- *SHIFT(x, S)* is used for *LSR(x, S)* or *SHL(x, S)*.

Utilities

static struct basic_block ***phi_parent** (struct basic_block **source*, pseudo_t *pseudo*)
Find the trivial parent for a phi-source.

static int **get_phisources**(struct instruction *sources[], int *nbr*, struct instruction **insn*)
Copy the phi-node's phisrcs into to given array.

Returns 0 if the the list contained the expected number of element, a positive number if there was more than expected and a negative one if less.

Note we can't reuse a function like linearize_ptr_list() because any VOIDs in the phi-list must be ignored here as in this context they mean 'entry has been removed'.

static pseudo_t **trivial_phi**(pseudo_t *pseudo*, struct instruction **insn*, struct pseudo_list ***list*)
Detect trivial phi-nodes.

Parameters

- **insn** – the phi-node
- **pseudo** – the candidate resulting pseudo (NULL when starting)

Returns the unique result if the phi-node is trivial, NULL otherwise

A phi-node is trivial if it has a single possible result:

- all operands are the same
- **the operands are themselves defined by a chain or cycle of phi-nodes** and the set of all operands involved contains a single value not defined by these phi-nodes

Since the result is unique, these phi-nodes can be removed.

int **kill_insn**(struct instruction **insn*, int *force*)
Kill an instruction.

Parameters

- **insn** – the instruction to be killed
- **force** – if unset, the normal case, the instruction is not killed if not free of possible side-effect; if set the instruction is unconditionally killed.

The killed instruction is removed from its BB and the usage of all its operands are removed. The instruction is also marked as killed by setting its ->bb to NULL.

static int **dead_insn**(struct instruction **insn*, pseudo_t **src1*, pseudo_t **src2*, pseudo_t **src3*)
Kill trivially dead instructions.

static inline int **replace_pseudo**(struct instruction **insn*, pseudo_t **pp*, pseudo_t *new*)
Replace the operand of an instruction.

Parameters

- **insn** – the instruction
- **pp** – the address of the instruction's operand
- **new** – the new value for the operand

Returns REPEAT_CSE.

static unsigned int **operand_size**(struct instruction **insn*, pseudo_t *pseudo*)
Try to determine the maximum size of bits in a pseudo.

Right now this only follow casts and constant values, but we could look at things like AND instructions, etc.

Simplifications

static int **simplify_mask_or_and**(struct instruction **insn*, unsigned long long *mask*, pseudo_t *ora*, pseudo_t *orb*)
Try to simplify MASK(OR(AND(x, M'), b), M).

Parameters

- **insn** – the masking instruction
- **mask** – the associated mask (M)
- **ora** – one of the OR's operands, guaranteed to be PSEUDO_REG
- **orb** – the other OR's operand

Returns 0 if no changes have been made, one or more REPEAT_* flags otherwise.

static int **simplify_mask_or**(struct instruction **insn*, unsigned long long *mask*, struct instruction **or*)
Try to simplify MASK(OR(a, b), M).

Parameters

- **insn** – the masking instruction
- **mask** – the associated mask (M)
- **or** – the OR instruction

Returns 0 if no changes have been made, one or more REPEAT_* flags otherwise.

static int **simplify_mask_shift_or**(struct instruction **sh*, struct instruction **or*, unsigned long long *mask*)
Try to simplify MASK(SHIFT(OR(a, b), S), M).

Parameters

- **sh** – the shift instruction
- **or** – the OR instruction
- **mask** – the mask associated to MASK (M):

Returns 0 if no changes have been made, one or more REPEAT_* flags otherwise.

static int **simplify_memop**(struct instruction **insn*)
Simplify memops instructions.

Note We walk the whole chain of adds/subs backwards. That's not only more efficient, but it allows us to find loops.

static int **simplify_cond_branch**(struct instruction **br*, struct instruction **def*, pseudo_t *newcond*)
Simplify SET_NE/EQ $0 + BR.

2.4 Sparse's Intermediate Representation

2.4.1 Instructions

This document briefly describes which field of struct instruction is used by which operation.

Some of those fields are used by almost all instructions, some others are specific to only one or a few instructions. The common ones are:

- .src1, .src2, .src3: (pseudo_t) operands of binops or ternary ops.
- .src: (pseudo_t) operand of unary ops (alias for .src1).
- .target: (pseudo_t) result of unary, binary & ternary ops, is sometimes used otherwise by some others instructions.
- .cond: (pseudo_t) input operands for condition (alias .src/.src1)
- .type: (symbol*) usually the type of .result, sometimes of the operands

Terminators

OP_RET

Return from subroutine.

- .src : returned value (NULL if void)
- .type: type of .src

OP_BR

Unconditional branch

- .bb_true: destination basic block

OP_CBR

Conditional branch

- .cond: condition
- .type: type of .cond, must be an integral type
- .bb_true, .bb_false: destination basic blocks

OP_SWITCH

Switch / multi-branch

- .cond: condition
- .type: type of .cond, must be an integral type
- .multijmp_list: pairs of case-value - destination basic block

OP_COMPUTEDGOTO

Computed goto / branch to register

- .src: address to branch to (void*)
- .multijmp_list: list of possible destination basic blocks

Arithmetic binops

They all follow the same signature:

- .src1, .src1: operands (types must be compatible with .target)
- .target: result of the operation (must be an integral type)
- .type: type of .target

OP_ADD

Integer addition.

OP_SUB

Integer subtraction.

OP_MUL

Integer multiplication.

OP_DIVU

Integer unsigned division.

OP_DIVS

Integer signed division.

OP_MODU

Integer unsigned remainder.

OP_MODS

Integer signed remainder.

OP_SHL

Shift left (integer only)

OP_LSR

Logical Shift right (integer only)

OP_ASR

Arithmetic Shift right (integer only)

Floating-point binops

They all follow the same signature:

- .src1, .src1: operands (types must be compatible with .target)
- .target: result of the operation (must be a floating-point type)
- .type: type of .target

OP_FADD

Floating-point addition.

OP_FSUB

Floating-point subtraction.

OP_FMUL

Floating-point multiplication.

OP_FDIV

Floating-point division.

Logical ops

They all follow the same signature:

- .src1, .src2: operands (types must be compatible with .target)
- .target: result of the operation
- .type: type of .target, must be an integral type

OP_AND

Logical AND

OP_OR

Logical OR

OP_XOR

Logical XOR

Integer compares

They all have the following signature:

- .src1, .src2: operands (types must be compatible)
- .target: result of the operation (0/1 valued integer)
- .type: type of .target, must be an integral type

OP_SET_EQ

Compare equal.

OP_SET_NE

Compare not-equal.

OP_SET_LE

Compare less-than-or-equal (signed).

OP_SET_GE

Compare greater-than-or-equal (signed).

OP_SET_LT

Compare less-than (signed).

OP_SET_GT

Compare greater-than (signed).

OP_SET_B

Compare less-than (unsigned).

OP_SET_A

Compare greater-than (unsigned).

OP_SET_BE

Compare less-than-or-equal (unsigned).

OP_SET_AE

Compare greater-than-or-equal (unsigned).

Floating-point compares

They all have the same signature as the integer compares.

The usual 6 operations exist in two versions: 'ordered' and 'unordered'. These operations first check if any operand is a NaN and if it is the case the ordered compares return false and then unordered return true, otherwise the result of the comparison, now guaranteed to be done on non-NaNs, is returned.

OP_FCMP_OEQ

Floating-point compare ordered equal

OP_FCMP_ONE

Floating-point compare ordered not-equal

OP_FCMP_OLE

Floating-point compare ordered less-than-or-equal

OP_FCMP_OGE

Floating-point compare ordered greater-or-equal

OP_FCMP_OLT

Floating-point compare ordered less-than

OP_FCMP_OGT

Floating-point compare ordered greater-than

OP_FCMP_UEQ

Floating-point compare unordered equal

OP_FCMP_UNE

Floating-point compare unordered not-equal

OP_FCMP_ULE

Floating-point compare unordered less-than-or-equal

OP_FCMP_UGE

Floating-point compare unordered greater-or-equal

OP_FCMP_ULT

Floating-point compare unordered less-than

OP_FCMP_UGT

Floating-point compare unordered greater-than

OP_FCMP_ORD

Floating-point compare ordered: return true if both operands are ordered (none of the operands are a NaN) and false otherwise.

OP_FCMP_UNO

Floating-point compare unordered: return false if no operands is ordered and true otherwise.

Unary ops

OP_NOT

Logical not.

- .src: operand (type must be compatible with .target)
- .target: result of the operation
- .type: type of .target, must be an integral type

OP_NEG

Integer negation.

- .src: operand (type must be compatible with .target)
- .target: result of the operation (must be an integral type)
- .type: type of .target

OP_FNEG

Floating-point negation.

- .src: operand (type must be compatible with .target)
- .target: result of the operation (must be a floating-point type)
- .type: type of .target

OP_SYMADDR

Create a pseudo corresponding to the address of a symbol.

- .src: input symbol (must be a PSEUDO_SYM)
- .target: symbol's address

OP_COPY

Copy (only needed after out-of-SSA).

- .src: operand (type must be compatible with .target)
- .target: result of the operation
- .type: type of .target

Type conversions

They all have the following signature:

- .src: source value
- .orig_type: type of .src
- .target: result value
- .type: type of .target

Currently, a cast to a void pointer is treated like a cast to an unsigned integer of the same size.

OP_TRUNC

Cast from integer to an integer of a smaller size.

OP_SEXT

Cast from integer to an integer of a bigger size with sign extension.

OP_ZEXT

Cast from integer to an integer of a bigger size with zero extension.

OP_UTPTR

Cast from pointer-sized unsigned integer to pointer type.

OP_PTRTU

Cast from pointer type to pointer-sized unsigned integer.

OP_PTRCAST

Cast between pointers.

OP_FCVTU

Conversion from float type to unsigned integer.

OP_FCVTS

Conversion from float type to signed integer.

OP_UCVTF

Conversion from unsigned integer to float type.

OP_SCVTF

Conversion from signed integer to float type.

OP_FCVTF

Conversion between float types.

Ternary ops

OP_SEL

- .src1: condition, must be of integral type
- .src2, .src3: operands (types must be compatible with .target)
- .target: result of the operation
- .type: type of .target

OP_RANGE

Range/bounds checking (only used for an unused sparse extension).

- .src1: value to be checked
- .src2, src3: bound of the value (must be constants?)
- .type: type of .src[123]?

Memory ops

OP_LOAD

Load.

- .src: base address to load from
- .offset: address offset
- .target: loaded value
- .type: type of .target

OP_STORE

Store.

- .src: base address to store to
- .offset: address offset
- .target: value to be stored
- .type: type of .target

Others

OP_SETFVAL

Create a pseudo corresponding to a floating-point literal.

- .fvalue: the literal's value (long double)
- .target: the corresponding pseudo
- .type: type of the literal & .target

OP_SETVAL

Create a pseudo corresponding to a string literal or a label-as-value. The value is given as an expression EXPR_STRING or EXPR_LABEL.

- .val: (expression) input expression
- .target: the resulting value
- .type: type of .target, the value

OP_PHI

Phi-node (for SSA form).

- .phi_list: phi-operands (type must be compatible with .target)
- .target: "result"
- .type: type of .target

OP_PHISOURCE

Phi-node source. Like OP_COPY but exclusively used to give a defining instructions (and thus also a type) to *all* OP_PHI operands.

- .phi_src: operand (type must be compatible with .target, alias .src)
- .target: the "result" PSEUDO_PHI
- .type: type of .target
- .phi_users: list of phi instructions using the target pseudo

OP_CALL

Function call.

- .func: (pseudo_t) the function (can be a symbol or a "register", alias .src))
- .arguments: (pseudo_list) list of the associated arguments
- .target: function return value (if any)
- .type: type of .target
- .fntypes: (symbol_list) list of the function's types: the first entry is the full function type, the next ones are the type of each arguments

OP_INLINED_CALL

Only used as an annotation to show that the instructions just above correspond to a function that have been inlined.

- .func: (pseudo_t) the function (must be a symbol, alias .src))
- .arguments: list of pseudos that where the function's arguments
- .target: function return value (if any)
- .type: type of .target

OP_SLICE

Extract a "slice" from an aggregate.

- .base: (pseudo_t) aggregate (alias .src)
- .from, .len: offet & size of the "slice" within the aggregate
- .target: result
- .type: type of .target

OP_ASM

Inlined assembly code.

- .string: asm template
- .asm_rules: asm constraints, rules

Sparse tagging (line numbers, context, whatever)

OP_CONTEXT

Currently only used for lock/unlock tracking.

- .context_expr: unused
- .increment: (1 for locking, -1 for unlocking)
- .check: (ignore the instruction if 0)

Misc ops

OP_ENTRY

Function entry point (no associated semantic).

OP_BADOP

Invalid operation (should never be generated).

OP_NOP

No-op (should never be generated).

OP_DEATHNOTE

Annotation telling the pseudo will be death after the next instruction (other than some other annotation, that is).

2.5 How to write sparse documentation

2.5.1 Introduction

The documentation for Sparse is written in plain text augmented with either reStructuredText (.rst) or MarkDown (.md) markup. These files can be organized hierarchically, allowing a good structuring of the documentation. Sparse uses Sphinx to format this documentation in several formats, like HTML or PDF.

All documentation related files are in the `Documentation/` directory. In this directory you can use `make html` or `make man` to generate the documentation in HTML or manpage format. The generated files can then be found in the `build/` sub-directory.

The root of the documentation is the file `index.rst` which mainly lists the names of the files with real documentation.

2.5.2 Minimal reST cheatsheet

Basic inline markup is:

- `*italic*` gives *italic*
- `**bold**` gives **bold**
- ``` ``mono`` ``` gives `mono`

Headings are created by underlining the title with a punctuation character; it can also be optionally overlined:

```
#############
Major heading
#############

Minor heading
-------------
```

Any punctuation character can be used and the levels are automatically determined from their nesting. However, the convention is to use:

- # with overline for parts
- * with overline for chapters
- = for sections
- - for subsections
- ^ for subsubsections

Lists can be created like this:

```
* this is a bulleted list
* with the second item
  on two lines
* nested lists are supported

        * subitem
        * another subitem

* and here is the fourth item

#. this is an auto-numbered list
#. with two items

1. this is an explicitly numbered list
2. with two items
```

Definition lists are created with a simple indentation, like:

```
term, concept, whatever
        Definition, must be indented and
        continue here.

        It can also have several paragraphs.
```

Literal blocks are introduced with `::`, either at the end of the preceding paragraph or on its own line, and indented text:

```
This is a paragraph introducing a literal block::

        This is the literal block.
        It can span several lines.

        It can also consist of several paragraphs.
```

Code examples with syntax highlighting use the *code-block* directive. For example:

```
.. code-block:: c

        int foo(int a)
        {
                return a + 1;
        }
```

will give:

```c
int foo(int a)
{
        return a + 1;
}
```

2.5.3 Autodoc

Sparse source files may contain documentation inside block-comments specifically formatted:

```
///
// Here is some doc
// and here is some more.
```

More precisely, a doc-block begins with a line containing only `///` and continues with lines beginning by `//` followed by either a space, a tab or nothing, the first space after `//` is ignored.

For functions, some additional syntax must be respected inside the block-comment:

```
///
// <mandatory short one-line description>
// <optional blank line>
// @<1st parameter's name>: <description>
// @<2nd parameter's name>: <long description
// <tab>which needs multiple lines>
// @return: <description> (absent for void functions)
// <optional blank line>
// <optional long multi-line description>
int somefunction(void *ptr, int count);
```

Inside the description fields, parameter's names can be referenced by using `@<parameter name>`. A function doc-block must directly precede the function it documents. This function can span multiple lines and can either be a function prototype (ending with `;`) or a function definition.

Some future versions will also allow to document structures, unions, enums, typedefs and variables.

This documentation can be extracted into a .rst document by using the *autodoc* directive:

```
.. c:autodoc:: file.c
```

For example, a doc-block like:

```
///
// increment a value
//
// @val: the value to increment
// @return: the incremented value
//
// This function is to be used to increment a
// value.
//
// It's strongly encouraged to use this
// function instead of open coding a simple
// ``++``.
int inc(int val)
```

will be displayed like this:

int **inc** (int *val*)

Parameters

- **val** – the value to increment

Returns the incremented value

This function is to be used to increment a value.

It's strongly encouraged to use this function instead of open coding a simple ++.

2.5.4 Intermediate Representation

To document the instructions used in the intermediate representation a new domain is defined: 'ir' with a directive:

```
.. op: <OP_NAME>
       <description of OP_NAME>
       ...
```

This is equivalent to using a definition list but with the name also placed in the index (with 'IR instruction' as descriptions).

CHAPTER 3

How to contribute

3.1 Submitting patches: the sparse version

Sparse uses a patch submit process similar to the Linux Kernel Submitting Patches

This document mostly focuses on the parts that might be different from the Linux Kernel submitting process.

1. Git clone a sparse repository:

   ```
   git clone git://git.kernel.org/pub/scm/devel/sparse/sparse.git
   ```

2. Coding Style remains the same.

3. Sign off the patch.

   The usage of the Signed-off-by tag is the same as Linux Kernel Sign your work.

   Notice that sparse uses the MIT License.

3.2 TODO

3.2.1 Essential

- SSA is broken by simplify_loads() & branches rewriting/simplification
- attributes of struct, union & enums are ignored (and possibly in other cases too).
- add support for bitwise enums

3.2.2 Documentation

- document the extensions
- document the API
- document the limitations of modifying ptrlists during list walking
- document the data structures
- document flow of data / architecture / code structure

3.2.3 Core

- if a variable has its address taken but in an unreachable BB then its MOD_ADDRESSABLE may be wrong and it won't be SSA converted.
  - let kill_insn() check killing of SYMADDR,
  - add the sym into a list and
  - recalculate the addressability before memops's SSA conversion
- bool_ctype should be split into internal 1-bit / external 8-bit
- Previous declarations and the definition need to be merged. For example, in the code here below, the function definition is **not** static:

```
static void foo(void);
void foo(void) { ... }
```

3.2.4 Testsuite

- there are more than 50 failing tests. They should be fixed (but most are non-trivial to fix).

3.2.5 Misc

- GCC's -Wenum-compare / clangs's -Wenum-conversion -Wassign-enum
- parse *_attribute*((fallthrough))
- add support for __builtin_unreachable()
- add support for format(printf()) (WIP by Ben Dooks)
- make use of UNDEFs (issues warnings, simplification, ... ?)
- add a pass to inline small functions during simplification.

3.2.6 Optimization

- the current way of doing CSE uses a lot of time
- add SSA based DCE
- add SSA based PRE
- Add SSA based SCCP
- use better/more systematic use of internal verification framework

3.2.7 IR

- OP_SET should return a bool, always
- add IR instructions for va_arg() & friends
- add a possibility to import of file in "IR assembly"
- dump the symtable
- dump the CFG

3.2.8 LLVM

- fix . . .

3.2.9 Internal backends

- add some basic register allocation
- add a pass to transform 3-addresses code to 2-addresses
- what can be done for x86?

3.2.10 Longer term/to investigate

- better architecture handling than current machine.h + target.c
- attributes are represented as ctypes's alignment, modifiers & contexts but plenty of attributes doesn't fit, for example they need arguments.
  - format(printf, . . .),
  - section(". . . ")
  - assume_aligned(alignment[, offsert])
  - error("message"), warning("message")
  - . . .
- should support "-Werror=. . . " ?
- All warning messages should include the option how to disable it. For example: "warning: Variable length array is used." should be something like: "warning: Variable length array is used. (-Wno-vla)"
- ptrlists must have elements be removed while being iterated but this is hard to insure it is not done.
- having 'struct symbol' used to represent symbols *and* types is quite handy but it also creates lots of problems and complications
- Possible mixup of symbol for a function designator being not a pointer? This seems to make evaluation of function pointers much more complex than needed.
- extend test-inspect to inspect more AST fields.
- extend test-inspect to inspect instructions.

CHAPTER 4

Indices and tables

- genindex

Index

Symbols

C

D

E

F

G

I

K

L

O

P

R

S

T

U

X