Sparse API

Utilities

Pointer list manipulation

The data structure handled here is designed to hold pointers but two special cases need to be avoided or need special care:

- NULL is used by {PREPARE,NEXT}_PTR_LIST() to indicate the end-of-list. Thus, NULL can't be stored in lists using this API but is fine to use with FOR_EACH_PTR() and its variants.
- VOID is used to replace a removed pseudo 'usage'. Since phi-nodes (OP_PHI) use a list to store their operands, a VOID in a phi-node list must be ignored since it represents a removed operand. As consequence, VOIDs must never be used as phi-node operand. This is fine since phi-nodes make no sense with void values but VOID is also used for invalid types and in case of errors.

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 in the list.

Linearize the entries of a list up to a total of **max**, and return the number of entries in the list.

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 split

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 element 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 element 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

unsigned int **hexval**(unsigned int *c*)

Return the value coresponding to an hexadecimal digit.

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().

Utilities for flowgraphs

int `cfg_postorder`(struct entrypoint *ep*)

Set the BB's reverse postorder links.

Each BB will also have its 'order number' set.

void `domtree_build`(struct entrypoint *ep*)

Build the dominance tree.

> **Each BB will then have:**
> - a link to its immediate dominator (::idom)
> - the list of BB it immediately dominates (::doms)
> - its level in the dominance tree (::dom_level)

bool `domtree_dominates`(struct basic_block *a*, struct basic_block *b*)

Test the dominance between two basic blocks.

Parameters:
- **a** – the basic block expected to dominate
- **b** – the basic block expected to be dominated

Returns: `true` if **a** dominates **b**, `false` otherwise.

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**.

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

---

int **evaluate_arguments**(struct symbol_list *argtypes , struct expression_list *args)

Evaluate the arguments of a function.

Parameters:
- **argtypes** – the list of the types in the prototype
- **args** – the list of the effective arguments

Optimization

Optimization main loop

---

void **optimize**(struct entrypoint *ep)

Optimization main loop.

Flow simplification

int **remove_phisources**(struct basic_block *par , struct basic_block *old)

Remove phi-sources from a removed edge.

Note: It's possible to have several edges between the same BBs. It's common with switches but it's also possible with branches. This function will only remove a single phi-source per edge.

static int **remove_other_phisources**(struct basic_block *bb , struct multijmp_list *list , struct basic_block *target)

Remove all phisources but the one corresponding to the given target.

static bool **bb_is_forwarder**(struct basic_block *bb)

Does the BB contains ignorable instructions but a final branch?

Note: something could be done for phi-sources but ... we'll see.

static bool **phi_check**(struct instruction *node)

Check if the sources of a phi-node match with the parent BBs.

int **convert_to_jump**(struct instruction *insn , struct basic_block *target)

Change a switch or a conditional branch into a branch.

static int **merge_bb**(struct basic_block *top , struct basic_block *bot)

Merge two BBs.

Parameters:
- **top** – the first BB to be merged
- **bot** – the second BB to be merged

int **simplify_cfg_early**(struct entrypoint *ep)

Early simplification of the CFG.

> **Three things are done here:**
>
> # inactive BB are removed # branches to a 'forwarder' BB are redirected to the forwardee. # merge single-child/single-parent BBs.

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 inline bool **is_pow2**(pseudo_t *src*)

Check if a pseudo is a power of 2.

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 ptr_list_to_array() for the phi-sources 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 inline bool **is_signed_constant**(long long _val_ , unsigned _osize_ , unsigned _nsize_)

Is this same signed value when interpreted with both size?

---

static inline pseudo_t **is_same_op**(pseudo_t _src_ , int _op_ , unsigned _osize_)

Is @src generated by an instruction with the given opcode and size?

---

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 inline int **replace_with_unop**(struct instruction *_insn_ , int _op_ , pseudo_t _src_)

Replace a binop with an unop.

Parameters:
- **insn** – the instruction to be replaced
- **op** – the instruction's new opcode
- **src** – the instruction's new operand

Returns: REPEAT_CSE

---

static inline int **replace_binop_value**(struct instruction *insn , int op , long long val)

Replace rightside's value.

Parameters:
- **insn** – the instruction to be replaced
- **op** – the instruction's new opcode
- **src** – the instruction's new operand

Returns: REPEAT_CSE

---

static inline int **replace_binop**(struct instruction *insn , int op , pseudo_t *pa , pseudo_t a , pseudo_t *pb , pseudo_t b)

Replace binop's opcode and values.

Parameters:
- **insn** – the instruction to be replaced
- **op** – the instruction's new opcode

Returns: REPEAT_CSE

---

static inline int **replace_opcode**(struct instruction *insn , int op)

Replace the opcode of an instruction.

Returns: REPEAT_CSE

---

static int **replace_insn_pair**(struct instruction *out , int op_out , struct instruction *in , int op_in , pseudo_t a , pseudo_t b , pseudo_t c)

Create an instruction pair OUT(IN(a, b), c).

---

static inline int **swap_insn**(struct instruction *out , struct instruction *in , pseudo_t a , pseudo_t b , pseudo_t c)

Create an instruction pair OUT(IN(a, b), c) with swapped opcodes.

---

static int **swap_select**(struct instruction *out , struct instruction *in , pseudo_t a , pseudo_t b , pseudo_t c , pseudo_t d)

Create an instruction pair OUT(SELECT(a, b, c), d).

---

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 **canonical_order**(pseudo_t *p1* , pseudo_t *p2*)

Check if the given pseudos are in canonical order.

The canonical order is VOID < UNDEF < PHI < REG < ARG < SYM < VAL The rationale is:

- VALs at right (they don't need a definition)
- REGs at left (they need a defining instruction)
- SYMs & ARGs between REGs & VALs
- REGs & ARGs are ordered between themselves by their internal number
- SYMs are ordered between themselves by address
- VOID, UNDEF and PHI are uninteresting (but VOID should have type 0)

---

static bool **insn_before**(struct basic_block *bb , struct instruction *x , struct instruction *y)

Test if, in the given BB, the ordering of 2 instructions.

---

static inline bool **can_move_to**(pseudo_t *src* , struct instruction *dst)

Check if it safe for a pseudo to be used by an instruction.

---

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.