GCC TREE 최적화
“Constant Folding” 과 “Algebraic Simplification”

정원교

2004년 6월 2일

목 차

제 1 절 25 주 문서를 시작하며

개념에 대한 부분은 Advanced Compiler Degisn Implementation by Steven S.Muchnick 을 참고하였습니다.

제 2 절 개념

이번 주에 다룰 “Constant Folding” 과 “Algebraic Simplification” 에 대한 개념을 간단히 언급하고 넘어가도록 하자.

2.1 Constant Folding

Constant Folding 는 Constant-expression evaluation 이라고 불리기도 하는데, 컴파일시 표현식의 operand 가 상수 (constant) 일 경우 이를 처리하는 것을 말한다. 이것은 대부분의 경우, 수행하는데 간단한 형태이며 가장 간단한 형태는 모든 값들이 상수로 이루어진 것을 말할 수 있다.

$$x = 1 + 2 * 4$$

예를 들어 설명한다면, 위와 같은 표현식이 존재할 때, 이를 아래와 같이 계산되어 컴파일된 바이너리가 최소한의 수행만 하도록 하는 것을 말한다.

$$x = 9$$

정수의 경우, 대부분 문제없이 수행할 수 있는데, 0 으로 나눈다거나 overflow 와 같은 exception 부분만 아닐 경우 대해서 말이다. 컴파일러에서는 이것이 실행시 문제없이 돌아갈 수 있느냐를 검사해 주는 것이 가장 중요할 것이다.

실수의 경우, 좀 더 상황이 복잡한데, 그 이유는 compiler 의 floating-point 연산이 실제 이 프로그램을 돌릴 프로세서의 결과값이 정확하게 같은지 따져보아야 한다. 그렇지 않을 경우, 제대로된 simulation 이 컴파일러에서 구현되어야 할 것이다. 제대로 구현이 안되어 있을 경우, 결과적으로 잘못된 operation 을 발생시킬 수 있다. 또한 ANSI/IEEE-754 규약에 따르면, exception 에 대한 type 과 exception 이 가능한 값에 대한 정의가 많이 지정되어 있다는 것인데, NaN 과 같은 infinite 들을 포함하고 있다.

2.2 Algebraic Simplification

Algebraic Simplification 는 표현식을 간단하게 할 수 있는 특정한 대수학 표현식이라고 할 수 있다. 위의 Constant Folding 처럼 컴파일러내에서 가장 잘 표현되어야 하는 것이라고 할 수 있다. 대부분의 Algebraic Simplification 들은 binary operator 의 조합을 가지며, 변수나 상수의 operator 에서 치환이 발표한 것을 적당한 표현식으로 대체한다.

$$i + 0 = 0 + i = i - 0 = i$$
$$0 - i = -i$$
$$i * 1 = 1 * i = i/1 = i$$
$$i * 0 = 0 * i = 0$$

위와 같이 binary operator 에 대한 간단화를 할 수 있을 뿐만 아니라 unary operator 에 대해서도 해당 될 수 있다.

$$-(-i) = i$$
$$i + (-j) = i - j$$

또한 Boolean 이나 bit-field type 들에 대해서도 해당될 수 있는데,

$$b \vee true = true \vee b = true$$
$$b \vee false = flase \vee b = b$$

위와 같은 표현식으로 간단화할 수 있다. 물론 위의 경우들 뿐만 아니라, Shift 연산이나, Relational operator 들에 대해서도 컴파일되는 architecture 에 따라 구현이 가능하다.

제 3 절 보조 함수 및 매크로

void **encode** (HOST_WIDE_INT *words, unsigned HOST_WIDE_INT low, HOST_WIDE_INT hi)

two-word integer 를 4 word 로 unpack 한다. LOW 와 HI 는 정수이며 두 'HOST_WIDE_INT' 조각이다. WORDS 는 HOST_WIDE_INT 들의 배열을 가르키고 있다.

void **decode** (HOST_WIDE_INT *words, unsigned HOST_WIDE_INT *low, HOST_WIDE_INT *hi)

4 word 들의 배열을 two-word integer 로 pack 한다. WORDS 는 word 들의 배열을 가르킨다. 정수는 두 'HOST_WIDE_INT' 조각으로 각각 *LOW 와 *HI 에 저장된다.

int **force_fit_type** (tree t, int overflow)

Type 에 속하지 않는 constant 내 모든 bit 를 0 혹은 1 로 설정함으로써 정수 constant T 의 type 에 대해 T 가 유효하도록 만든다.

만약 signed overflow 가 발생하면 1 을 그렇지 않으면 0 을 반환한다. 만약 OVERFLOW 가 0 이 아니면, signed overfow 는 T 를 계산하는 동안 이미 발생한 것이다. 그래서 그것을 단순히 늘린다.

(존재할 경우) CHECK_FLOAT_VALUE 를 호출함으로써 type 이 real constant T 에 대해 유효하도록 만든다.

int **add_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, unsigned HOST_WIDE_INT l2, HOST_WIDE_INT h2, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv)

두 doubleword 정수들을 doubleword 결과로 더한다. 각 argument 는 두 'HOST_WIDE_INT' 조각들로 주어진다. 한 argument 는 L1 과 H1; 다른 하나는 L2 와 H2 이다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

int **neg_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv)

두 doubleword 정수들을 doubleword 결과로 negate 한다. 그것이 sign 으로 가정했을 때 만약 operation 이 overflow 이면, 0 이 아닌 값을 반환한다. 각 argument 는 L1 과 H1 내 두 'HOST_WIDE_INT' 조각들로 주어진다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

int **mul_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, unsigned HOST_WIDE_INT l2, HOST_WIDE_INT h2, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv)

두 doubleword 정수들을 doubleword 결과로 곱한다. 그것이 sign 으로 가정했을 때 만약 operation 이 overflow 이면, 0 이 아닌 값을 반환한다. 각 argument 는 두 'HOST_WIDE_INT' 조각들로 주어진다. 한 argument 는 L1 과 H1; 다른 하나는 L2 와 H2 이다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

void **lshift_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, HOST_WIDE_INT count, unsigned int prec, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv, int arith)

결과의 PREC bit 들만 유지한체, L1, H2 내 Doubleword 정수를 COUNT 자리로 왼쪽 shift 한다. 만약 COUNT 가 음수이면 오른쪽 shift 한다. ARITH nonzero 는 arithmetic shift 을 의미하며 그렇지 않을 땐 logical shift 를 사용한다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

void **rshift_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, HOST_WIDE_INT count, unsigned int prec, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv, int arith)

결과의 PREC bit 들만 유지한체, L1, H2 내 Doubleword 정수를 COUNT 자리로 오른쪽 shift 한다. COUNT 는 반드시 양수여야 한다. ARITH nonzero 는 arithmetic shift 을 의미하며 그렇지 않을 땐 logical shift 를 사용한다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

void **lrotate_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, HOST_WIDE_INT count, unsigned int prec, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv)

결과의 PREC bit 들만 유지한체, L1, H2 내 Doubleword 정수를 COUNT 자리로 왼쪽 rotate 한다. 만약 COUNT 가 음수일 경우 오른쪽 rotate 한다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

void **rrotate_double** (unsigned HOST_WIDE_INT l1, HOST_WIDE_INT h1, HOST_WIDE_INT count, unsigned int prec, unsigned HOST_WIDE_INT *lv, HOST_WIDE_INT *hv)

결과의 PREC bit 들만 유지한체, L1, H2 내 Doubleword 정수를 COUNT 자리로 오른쪽 rotate 한다. COUNT 는 반드시 양수여야 한다. 값은 *LV 와 *HV 내 두 'HOST_WIDE_INT' 조각으로 저장된다.

int **div_and_round_double** (enum tree_code code, int uns, unsigned HOST_WIDE_INT lnum_orig, HOST_WIDE_INT hnum_orig, unsigned HOST_WIDE_INT lden_orig, HOST_WIDE_INT hden_orig, unsigned HOST_WIDE_INT *lquo, HOST_WIDE_INT *hquo, unsigned HOST_WIDE_INT *lrem, HOST_WIDE_INT *hrem)

doubleword integer LNUM, HNUM 를 doubleword integer LDEN, HDEN 로 나누서 몫은 *LQUO, *HQUO 에, 나머지는 *LREM, *HREM 에 저장한다. CODE 는 division 의 종류인 tree code 로써, TRUNC_DIV_EXPR 혹은 FLOOR_DIV_EXPR, CEIL_DIV_EXPR, ROUND_DIV_EXPR, EXACT_DIV_EXPR 중 하나이다. 몫이 어떻게 integer 로 round 되어지는지 제어하고, 만약 operation 이 overflow 난다면 0 이 아닌 값을 반환한다. UNS 의 값이 0 이 아닐 경우, unsigned division 을 수행함을 의미한다.

REAL_VALUE_TYPE **real_value_truncate** (enum machine_mode mode, REAL_VALUE_TYPE arg)

Narrower mode 에서 가장 근접하게 가능한 값을 표현하는 real value 를 효율적으로 truncate 한다. 결과는 실제로 argument 와 같은 data type 으로 표현되지만 그것의 값은 보통 다르다.

Trap 이 FP operation 동안 발생할 수 있으며, 만들어진 handler 를 가지는 것은 calling function 의 책임이다.

int **target_isinf** (REAL_VALUE_TYPE x)

IEEE double precision number 에서 infinity 를 검사한다.

int **target_isnan** (REAL_VALUE_TYPE x)

IEEE double precision number 가 NaN 인지 아닌지를 검사한다.

int **target_negative** (REAL_VALUE_TYPE x)

Negative IEEE double precision number 를 검사한다.

void **exact_real_inverse_1** (PTR p)

적당한 설명이 존재하지 않는다.

int **exact_real_inverse** (enum machine_mode mode, REAL_VALUE_TYPE *r)

적당한 설명이 존재하지 않는다.

REAL_VALUE_TYPE **real_hex_to_f** (const char *s, enum machine_mode mode)

C99 hexadecimal floating point string constant S 를 변환한다. Mode MODE 인 실수값 type 을 반환한다. 이 함수는 REAL_ARITHMETIC 가 존재하지 않을 경우 host computer 의 floating point arithmetic 를 사용한다.

tree **negate_expr** (tree t)

주어진 표현식 T 의 negation 를 반환한다. case 가 null 을 반환할 경우 T 가 null 이 되도록 허락한다.

tree **split_tree** (tree in, enum tree_code code, tree *conp, tree *litp, tree *minus_litp, int negate_p)

Tree IN 을 constant 와 literal, variable 부분으로 잘라낸다. 이것은 CODE 를 사용하여 IN 으로 다시 조합될 수 있는 형태이다. “constant” 는 TREE_CONSTANT 를 가진 표현식을 의미하지 실제 constant 를 의지하지 않는다. CODE 는 반드시 commutative arithmetic operation 이여야 한다. Constant part 는 *CONP 에 저장하고 *LITP 에 literal 를 variable 부분을 반환한다. 만약 조각이 존재하지 않는다면, NULL 로 설정한다. 만약 tree 가 이러한 방식으로 분해되지 않는다면, variable part 는 전체 tree 를 반환하고 다른 부분은 NULL 로 반환된다.

만약 CODE 가 PLUS_EXPR 이면, 우리는 또한 MINUS_EXPR 를 사용한 tree 들로 분해할 수 있다. 해당 경우, 우리는 뺄셈을 하는 operand 를 negate 함으로써 할 수 있는데, 대신 우리가 literal 을 위해 *MINUS_LITP 를 대신 사용하는 경우는 예외 이다.

만약 NEGATE_P 가 true 이면, 우리는 IN 의 모든 것을 negate 하는데 여전히 MINUS_LITP 를 대신 사용하는 literal 의 경우 예외이다.

만약 IN 자신이 literal 혹은 constant 이면, 적당하게 그것을 반환한다.

우리는 세개의 값 중 모든 것이 IN 와 같은 type 이라는 보장을 하지 못하지만, 그들이 같은 signedness 와 mode 를 가질 것이라는 건 할 수 있다.

tree **associate_trees** (tree t1, tree t2, enum tree_code code, tree type)

위의 함수에 의해 절단된 tree 를 재결합한다. T1 과 T2 는 결합할 표현식이거나 null 이다. 새로운 표현식을 반환한다. 만약 우리가 operation 을 build 할 경우 그것은 TYPE 과 CODE 를 가지고 있다.

tree **int_const_binop** (enum tree_code code, tree arg1, tree arg2, int notrunc)

새로운 상수를 생성하기 위해 정수 상수들 ARG1 과 ARG2 를 수행 CODE 하에서 결합시킨다.

만약 NOTRUNC 가 0 이 아니면, Data type 을 맞추기 위해 결과를 truncate 하지 않는다.

void **const_binop_1** (PTR data)

Float overflow handler 에 의해 보호되는 동안 const_binop 를 위한 real arithmetic 를 수행한다.

static tree **const_binop** (enum tree_code code, tree arg1, tree arg2, int notrunc)

새로운 constant 를 생성하기 위해 operation CODE 하에 두 constant ARG1 와 ARG2 를 조합한다. 우리는 ARG1 와 ARG2 가 같은 data 을 가진다고 가정하거나, 적어도 같은 constant 조유와 같은 machine mode 를 가진다고 가정한다.

NOTRUNC 값이 0 이 아닐 경우, data type 을 맞추기 위해 결과를 truncate 하지 않는다.

hashval_t **size_htab_hash** (const void *x)

Hash code code X, INTEGER_CST 를 반환한다.

int **size_htab_eq** (const void *x, const void *y)

*X (INTEGER_CST tree node) 에 의해 표현되는 값이 *Y 에 의해 나타나는 값과 동일할 경우 0 이 아닌 값을 반환한다.

tree **size_int_wide** (HOST_WIDE_INT number, enum size_type_kind kind)

값의 low-order HOST_BITS_PER_WIDE_INT bit 가 NUMBER 이고, KIND 로 표현되는 sizetype 을 가진 INTEGER_CST 를 반환한다.

tree **size_int_type_wide** (HOST_WIDE_INT number, tree type)

거의 같지만, 요청된 type 이 분명하게 지정된다.

tree **size_binop** (enum tree_code code, tree arg0, tree arg1)

산술 연산 CODE 로 operand 들 OP1 과 OP2 를 결합한다. CODE 는 tree code 이다. 결과의 type 은 operand 들로부터 취해진다. 양쪽 모두 반드시 같은 type 정수 type 이고 size type 이여야 한다. 만약 operand 들이 상수이면, 결과도 그렇다.

tree **size_diffop** (tree arg0, tree arg1)

주어진 두 값으로, 둘다 sizetype 이거나 둘다 bitsizetype, 두 값사이의 차이점을 계산한다. Operand 들의 type 에 맞는 signed type 형태의 값을 반환한다.

void **fold_convert_1** (PTR data)

floating-point constant 들을 변환하기 위한 함수. floating point exception handler 에 의해 보호된다.

tree **non_lvalue** (tree x)

X 와 같은 expr 을 반환하지만, lvalue 로 전혀 유효하지 않은 것을 반환한다.

tree **pedantic_non_lvalue** (tree x)

pedantic 일 경우, X 와 같은 expr 을 반환하지만, pedantic lvalue 로써 확실히 유효하지 않을 수 있다. 그렇지 않을 경우 X 를 반환한다.

enum tree_code **invert_tree_comparison** (enum tree_code code)

주어진 tree comparison code 와 논리적인 반대인 code 를 반환한다. NE_EXPR 와 EQ_EXPR 를 제외한 floating-point comparison 에 대해 이 함수를 수행하는 것은 안전하지 못하다.

enum tree_code **swap_tree_comparison** (enum tree_code code) tree_code code)

비슷하지만, operand 들이 swap 되어진 결과를 비교에 대해 반환한다. 이것은 floating-point 에 대해 안전하다.

int **truth_value_p** (enum tree_code code)

만약 CODE 가 truth value 를 표현하는 tree code 일 경우 0 이 아닌 값을 반환한다.

int **operand_equal_p** (tree arg0, tree arg1, int only_const)

두 operand 들이 결과적으로 같다면 0 이 아닌 값을 반환한다. 만약 ONLY_CONST 가 non-zero 라면, 오직 상수에 대해서만 0 이 아닌 값을 반환한다. 이 함수는 operand 들이 구별이 불가능한지 아닌지를 검사하는데, 이것은 C 의 == operation 을 사용하는지를 구분하는것이 아니다. 이러한 구별은 IEEE floating point 에서는 중요한데, 이유는 아래와 같다.

1. −0.0 와 0.0 는 구별될 수 있지만, −0.0==0.0 는 아니다.
2. 두 NaN 은 구별할 수 없을 지 몰라도 NaN!=NaN 는 아니다.

int **operand_equal_for_comparison_p** (tree arg0, tree arg1, tree other)

operand_equal_p 와 비슷하지만, ARG1 이 OTHER 와 비교되어질 때 ARG1 로부터 shorten_compare 에 의해 ARG0 이 구성되어질 수 있는지 살펴본다.
아직 알 수 없을 경우, 0 을 반환한다.

int **twoval_comparison_p** (tree arg, tree *cval1, tree *cval2, int *save_p)

ARG 가 comparison 이거나 comparison 에 대한 arithmetic 을 수행하는 표현식인지를 살펴본다. comparison 들은 반드시 두개의 다른 값들에 대해 비교 해야 하며, 각각 *CVAL1 와 *CVAL2 저장되어져 있을 것이다. 만약 두개의 값이 0 이 아니라면 몇몇 operand 가 이미 발견되었다는 것을 의미한다. 표현식에서 variable 들은 comparison 을 제외한 다른 곳에서는 전혀 사용하지 않을 것이다. 만약 SAVE_P 가 true 일 경우, 이것은 표현식 주변에 있는 SAVE_EXPR 를 제거하고 save_expr 가 CVAL1 와 CVAL2 를 인자로 호출될 필요가 있음을 의미한다.
만약 이것이 true 이면 1 을 반환하고 그렇지 않을 경우 0 을 반환한다.

tree **eval_subst** (tree arg, tree old0, tree new0, tree old1, tree new1)

ARG 는 단순히 arithmetic operation 들과 comparison 들을 포함하는 것으로 알려진 tree 이다. Comparison 의 operand 로써 OLD0 가 나타날 경우 NEW0 으로 대체함으로써 tree 내 operation 들을 evaluate 한다. NEW1 과 OLD1 에 대해서도 마찬가지이다.

tree **omit_one_operand** (tree type, tree result, tree omitted)

expression 의 결과가 TYPE 으로 변환된 RESULT 인 경우에 대한 tree 를 반환한다. OMITTED 는 이전 표현식의 operand 였지만, 이제 필요하지 않다. (예를 들면, 우리는 OMITTED * 0 를 fold 한 경우를 들 수 있겠다.)
만약 OMITTED 가 side effect 를 가지고 있다면, 우리는 반드시 그것을 재평가 하여야 한다. 그렇지 않다면 단순히 TYPE 으로 RESULT 를 변환한다.

tree **pedantic_omit_one_operand** (tree type, tree result, tree omitted)

비슷하지만, non_lvalue 대신에 pedantic_non_lvalue 를 호출한다.

tree **invert_truthvalue** (tree arg)

ARG 의 truth-negation 에 대해 간단화된 tree 를 반환한다. 이것은 절대 ARG 자체를 변경하지 않는다. 우리는 ARG 가 truth value (0 혹은 1) 을 반환하는 operation 으로 가정한다.

tree **distribute_bit_expr** (enum tree_code code, tree type, tree arg0, tree arg1)

ARG0 와 ARG1 에 적용된 주어진 bit-wise operation CODE 는 두 operand 가 공통된 input 을 가지는 서로 다른 bit-wise operation 인지 살펴본다. 그래서 만약 그렇다면 operation 을 절약하기 위해 bit operation 들을 분리하며, 상수 들이 포함되어 있다면 둘로 가능한 분리한다. 예를 들어

$(A|B)$ & $(A|C)$ 를 $A|(B$ & $C)$

로 변환한다. 좀 더 진전된 simplification 는 B 와 C 가 상수일 경우 발생할 것이다.
만약 이 최적화가 수행될 수 없다면, 0 이 반환될 것이다.

tree **make_bit_field_ref** (tree inner, tree type, int bitsize, int bitpos, int unsignedp)

BITPOS 에서 사작하는 INNER 의 BITSIZE bit 들을 참조하는 type TYPE 인 BIT_FIELD_REF 를 반환한다. field 는 UNSIGNEDP 가 0 이 아닐 경우 unsigned 이다.

tree **optimize_bit_field_compare** (enum tree_code code, tree compare_type, tree lhs, tree rhs)

Bit-field compare 를 최적화한다.

두 경우가 존재하는데, 첫번째는 constant 에 대한 비교이며, 두번째는 field 가 chunk (byte, halfword, word) 의 시작과 관련하여 같은 bit 위치인 두 item 의 비교이다. 이러한 경우 우리는 bitfield extraction 에 따른 shift implicit 를 피할 수 있다.

Constant 에 대해, 우리는 mask 의 BIT_AND_EXPR 를 가지는 shifted constant 와 비교되는 operand 의 byte 혹은 halfword, word 의 비교를 emit 한다. 같은 위치 의 두 field 에 대해서는 비슷한 mask 와 AND 를 수행하고 AND 들의 결과를 비교한다.

CODE 는 comparison code 인데, NE_EXPR 혹은 EQ_EXPR 중 하나일 것이다. COMPARE_TYPE 는 comparison 의 type 이며, LHS 와 RHS 는 각각 comparison 의 왼쪽과 오른쪽 operand 들이다.

만약 위에서 설명한 최적화가 수행될 수 있을 경우, 결과 tree 를 반환하고, 그렇지 않을 경우 0 을 반환한다.

tree **decode_field_reference** (tree exp, HOST_WIDE_INT *pbitsize, HOST_WIDE_INT *pbitpos, enum machine_mode *pmode, int *punsignedp, int *pvolatilep, tree *pmask, tree *pand_mask)

fold_truthop 의 하위루틴: field reference 를 해석한다.

만약 EXPR 가 comparison reference 라면, 우리는 innermost reference 를 반환한다.

*PBITSIZE 는 reference 내 bit 의 number 로 설정되고, *PBITPOS 는 시작 bit number 로 설정된다.

만약 innermost field 가 완전히 mode-sized unit 에 포함될 수 있다면, *PMODE 에 해당 mode 가 설정된다. 그렇지 않다면, VOIDmode 가 설정된다.

*PVOLATILEP 는 만약 지금까지 만난 어떤 표현식이 volatile 일 경우 1 로 설정 되고 그렇지 않다면 변경되지 않는다.

*PUNSIGNEDP 는 field 의 signedness 로 설정된다.

*PMASK 는 사용된 mask 로 설정된다. 이것은 BIT_AND_EXPR 에 포함되거나 field 의 width 로부터 유도된다.

*PAND_MASK 는 발견될 경우 BIT_AND_EXPR 에서 발견된 mask 로 설정된다.

만약 이것이 component reference 가 아니거나 우리가 처리할 수 있는 것이 아니라면 0 을 반환한다.

int **all_ones_mask_p** (tree mask, int size)

만약 MASK 가 low-order bit position 인 것의 SIZE 의 mask 를 나타낼 경우 0 이 아닌 값을 반환한다.

int **simple_operand_p** (tree exp)

fold_truthop 의 하위루틴: operand 가 무조건적으로 evaluate 되어질 수 있을 정도로 간단한지를 결정한다.

다음 함수들은 fold_range_test 의 하위루틴이며 comparison 의 logical combination 을 range test 로 변경을 시도하려고 한다.

예를 들면, 아래의 두 예제,

X == 2 || X == 3 || X == 4 || X == 5

와

X >= 2 && X <= 5

는 아래와 같이 변환될 것이다.

(unsigned) (X − 2) ¡= 3

우리는 comparison 의 각각의 집합을 range 의 inside 와 outside 로 표현하는데 IN_P 와 같이 명명된 variable 를 사용하여, lower 와 upper bound 를 가지는 range 를 나타낸다. 만약 bound 들의 하나가 생략된다면, type 의 가장 높은 값과 가장 낮은 값을 나타낸다.

아래의 comment 에서, 우리는 range 를 괄호와 두 숫자로 표현하는데, 표현하는데 있어서 해당 range 의 inside 를 표현하기 위해서 "+" 를 사용하고 해당 range 의 outside 를 나타내기 위해 "−" 르 사용한다. 그래서 이러한 조건은 prefix 에 서로 다른 기호를 사용함으로써 반대로 만들 수 있다. 생략된 bound 는 "−" 로 나타낸다. 예를 들어, "− [−, 10]" 는 가장 낮은 값에서 시작하고 10 에서 끝나는 범위의 바깥쪽을 의미하는데, 다른 말로 풀이하면 10 보다 큰 것을 의미한다. Range "+ [−, −]" 는 항상 true 이기 때문에 range "− [−, −]" 는 항상 false 이다.

이렇게 설정함으로써 missing bound 들은 consistent manner 로써 다루어지게 되는데, 이것은 missing bound 혹은 "true" 와 "false" 가 특별한 경우를 사용하여 다루어질 필요가 없도록 한다.

tree **range_binop** (enum tree_code code, tree type, tree arg0, tree arg1, int upper0_p, int upper1_p)

CODE 를 ARG0 과 ARG1 에 적용한 결과값을 반환하는데, 생략되어진 ARG0 과/혹은 ARG1 의 경우 또한 다룬다. 즉 무제한 range 를 다룬다는 것이다. UPPER0_P 와 UPPER1_P 는 만약 각각의 argument 가 upper bound 이고 lower 에 대해서는 0 일 경우 0 이 아닌 값이다. TYPE 은, 만약 0 이 아니면, 결과의 type 이다. comparison 에 대해서는 반드시 지정되어야 한다. ARG1 은 만약 둘다 지정되어 있을 경우, ARG0 의 type 으로 변환될 것이다.

tree **make_range** (tree exp, int *pin_p, tree *plow, tree *phigh)

주어진 EXP, logical expression, 는 PIN_P 와 PLOW, PHIGH 에 의해 나타난 variable 들을 테스트하여 range 를 설정한다. 실제로 검사되어진 표현식을 반환한다. *PLOW 와 *PHIGH 는 반환되는 표현식과 같은 type 으로 구성될 것이다. 만약 EXP 가 comparison 이 아니라면, 우리는 대부분의 경우, 유용한 값과 range 를 반환하지 못할 것이다.

tree **build_range_check** (tree type, tree exp, int in_p, tree low, tree high)

주어진 range 인 LOW 와 HIGH, IN_P 그리고 표현식 EXP, 마지막으로 결과 type 인 TYPE 을 사용하여, EXP 가 range 의 내부에 있는지를 (혹은 외부에 있는지, 이는 IN_P 에 의존한다.) 검사하기 위한 표현식을 반환한다.

int **merge_ranges** (int *pin_p, tree *plow, tree *phigh, int in0_p, int in1_p, tree low0, tree high0, tree low1, tree high1)

주어진 두 range 들을 사용하여 우리가 그것을 하나로 병합할 수 있는지를 살펴본다. 만약 우리가 할 수 있다면 1 을 없다면 0 을 반환한다. 지정된 parameter 내에 output range 를 설정한다.

tree **fold_range_test** (tree exp)

EXP 는 몇몇 boolean test 들의 logical combination 이다. 우리는 몇몇 range test 를 통해서 그것을 병합할 수 있는지를 살피며, 가능할 경우 새 tree 를 반환한다.

tree **unextend** (tree c, int p, int unsignedp, tree mask)

fold_truthop 의 하위루틴: C 는 P bit 값으로 해석된 INTEGER_CST 이다. 이것을 정리하여 만약 C 가 그것의 전체 width 에 대해 signed-extended 일 경우에 여분의 bit 를 0 으로 설정할 것이다. 만약 MASK 가 0 이 아닌 값이면, 여분의 bit 로 AND 연산이 되어진 INTEGER_CST 이다.

tree **fold_truthop** (enum tree_code code, tree truth_type, tree lhs, tree rhs)

LHS 와 RHS 의 logical expression 들을 fold 할 방법을 찾는다: 같은 innermost item 에 두 comparison 을 병합하려고 한다. "ch >= '0' && ch <= '9'" 와 같은 range test 들을 찾는다. 비용이 많이 드는 branch 들을 가지는 machine 상에서 간단한 형태의 조합을 찾아 무조건적으로 RHS 를 evaluate 한다.

예를 들어, 만약 우리가 p− >a == 2 && p− >b == 4 를 가지고 있고 A 와 B 를 늘리기 충분히 큰 object 를 만들 수 있다면, mask 로 AND 연산이 수행된 object 에 대한 comparison 으로 이것을 수행할 수 있다.

만약 우리가 p− >a == q− >a && p− >b == q− >b 를 가지고 있다면, 한 comparison 으로 이것을 하기 위해 bit masking operation 들을 사용할 수 있을 것이다.

우리는 두 normal comparison 들과 위에서 언급한 함수로 만들어진 BIT_AND_EXPR 들을 검사한다.

CODE 는 수행된 logical operation 이다. 이것은 TRUTH_ANDIF_EXPR 혹은 TRUTH_AND_EXPR, TRUTH_ORIF_EXPR, TRUTH_OR_EXPR 일 수 있다.

TRUTH_TYPE 은 logical operand 의 type 이고 LHS 와 RHS 은 그것의 두 operand 들이다.

우리는 간단화된 tree 를 반환하거나 최적화가 불가능하면 0 을 반환한다.

tree **optimize_minmax_comparison** (tree t)

상수와 MIN_EXPR 혹은 MAX_EXPR 비교를 하는 T 를 최적화한다.

tree **extract_muldiv** (tree t, tree c, enum tree_code code, tree wide_type)

T 는 constant C 를 몫 (CODE 가 어떤 종류의 나누기 혹은 곱셈인지 말한다.) 으로 하여 나눠지거나, 곱해지는 integer expression 이다. T 속에 이미 다른 operation 들로 fold 된 것을 제거할 수 있는지 살펴본다. WIDE_TYPE 는 (만약 null 이 아니라면) 우리의 type 보다 더 넓은지 계산하는데 사용 되는 type 이다.

예를 들어, 우리가 (X * 8) + (Y * 16) 를 4 로 나눈다면, 우리는 (X * 2) + (Y * 4) 를 되돌려 줄 수 있다. 하지만 원래의 표현식이 overflow 하지 않아야 된다거나, 해당 overflow 가 질의를 하는 언어에 type 으로 정의 되어 있지 않아야 한다는 것을 정확하게 해야한다.

우리는 또한 (X + 7) * 4 를 X * 4 + 28 로 변경할 수 있는데, machine 이 multiply-accumulate insn 를 가지고 있거나, 이것이 addressing calculation 의 부분일 경우에 해당한다.

만약 우리가 non-null expression 를 반환한다면, 원래 계산과 동일한 형태인 것 이지만, 원래의 type 은 아닐 수 있다.

tree **strip_compound_expr** (tree t, tree s)

만약 T 가 COMPOUND_EXPR (이것은 S, 즉 SAVE_EXPR, 를 evaluate 위해서 거의 포함된다.) 를 포함한다면 실제 evaluate 되어진 표현식을 반환한다. 때때로 tree 를 우리가 수정할 수도 있음을 참고하라.

tree **constant_boolean_node** (int value, tree type)

지정된 constant VALUE 를 가지는 (0 혹은 1 일 것이다.), 또한 지정된 TYPE 을 가지는 node 를 반환한다.

int **count_cond** (tree expr, int lim)

COND_EXPR 들의 nesting 이 얼마나 복잡할 수 있는지 보기 위한 유틸리티 함수이다. EXPR 은 표현식이고 LIMIT 는 우리가 고려하지 않을 정도의 count 이다. (이것은 complex 표현식에서 너무 많은 시간을 보내는 것을 피해기 위해서이다.)

tree **fold_binary_op_with_conditional_arg** (enum tree_code code, tree type, tree cond, tree arg, int cond_first_p)

'a + (b ? x : y)' 를 'x ? (a + b) : (a + y)' 로 변형시킨다. 'a + (x < y)' 를 '(x < y) ? (a + 1) : (a + 0)' 로 변형시킨다. 이 CODE 는 '+' 에, '(b ? x : y)' 혹은 '(x < y)' 와 같은 것은 COND 표현식, 'a' 와 같은 것은 ARG 에 대응한다. 만약 COND_FIRST_P 가 0 이 아닐 경우, COND 는 CODE 에 대한 첫번째 argument 이며, 그렇지 않을 경우 (여기 주어진 예제처럼) 두번째 argument 이다. TYPE 은 원래 표현식의 type 이다.

tree **fold** (tree expr)

EXPR 의 constant folding 과 다른 관련된 simplification 을 수행한다. 관련된 simplification 는 x*1 =¿ x, x*0 =¿ 0, 기타 등등을 포함하고 associative law 의 application 도 포함한다. NOP_EXPR conversion 은 (전체적인 C type 표현식을 변경하지 않고 주의하는 한) 자유롭게 제거될 수 있을 것이다. 우리는 CONVERT_EXPR 혹은 FIX_EXPR, FLOAT_EXPR 를 통해서는 간단화 할 수 없지만, 만약 그들이 constant operand 들을 가지고 있다면, constant-fold 할 수 있다.

int **multiple_of_p** (tree type, tree top, tree bottom)

첫번째 argument 가 두번째 argument 의 곱인지를 결정한다. 만약 그렇지 않다면, 0 을 반환하거나, 우리는 쉽게 그것이 그러한지 결정할 수 없을 것이다.

우리가 고려하는 (이 시점에서 이 routine 은 좀 더 확실히 일반화되어질 수 있고, 수행할 *_DIV_EXPR 의 fold 경우들이 무엇이건 확생할 수 있을 것이다.) 어떤 분류에 대한 예제는 다음과 같은 것을 발견하는 것이다.

SAVE_EXPR (I) * SAVE_EXPR (J * 8)

는 SAVE_EXPR (J * 8) node 들이 같은 node 이다는 것을 알

SAVE_EXPR (J * 8)

의 곱니다.

이 code 는 또한 다음의 것도 발견하는데 다뤄질 수 있을 것이다.

SAVE_EXPR (I) * SAVE_EXPR (J * 8)

는 8 의 배수이기 때문에 우리는 가능할 수 있는 나머지에 대해 신경을 쓸 필요가 없다.

우리는 오직 SAVE_EXPR 를 어떻게 계산되어질 수 있는지 결정하는데만 내부를 살펴본다는 것을 알기 바란다; SAVE_EXPR 의 내부들을 가지고 다른 어떤것과 fold 하는 것은 안전 하지 않는다. 실행시 evaluate 되어질 것이라는 것을 알 수 없기 때문이다. 예를 들어 위의 후 예제는 SAVE_EXPR (I) * J 혹은 그것에 관한 어떤 variant 로써 수행되어질 수 없는데, 원래 SAVE_EXPR 의 evaluation time 에서 J 의 값은 새 표현식이 evaluate 되어질 시간과 같을 필요가 없기 때문이다. 이 분류에서 유효할 수 있는 최선의 최적화는 는

SAVE_EXPR (I) * SAVE_EXPR (SAVE_EXPR (J) * 8)

를 8 로 나누어 아래와 같이 변화시키는 것이다.

SAVE_EXPR (I) * SAVE_EXPR (J)

(같은 SAVE_EXPR (J) 가 원래의 것과 변형된 버전에서 사용된 곳)

int **tree_expr_nonnegative_p** (tree t)

만약 't' 가 non-negative 로 알려질 경우, true 를 반환한다.

int **rtl_expr_nonnegative_p** (rtx r)

만약 'r' 이 non-negative 로 알려져 있을 경우, true 를 반환한다. 오직 그 순간에서의 constant 들만 다룬다.

제 4 절 구현

이제 아래부터는 GCC 의 fold () 함수가 각 node 에 대해서 어떻게 처리하는지에 대해 살펴보도록 하자.

아래의 설명중 fold () 함수내에서 사용되는 지역 변수가 나올 수 있는데, t, arg0, arg1, wins 가 나타날 수 있다. t 는 현재 folding 을 하려고 하는 node 이다. arg0 은 constant folding 을 실행할 node 의 첫번째 operand, arg1 은 두번째 operand 가 존재할 경우 설정된다. 마지막 wins 는 arg0 과 arg1 둘 다 INTEGER_CST 혹은 REAL_CST node 일 경우 1 의 값을 그렇지 않을 경우 0 을 갖게된다. 즉 operand 가 모두 상수이면 1 을 가지게 될 것이다.

- INTEGER_CST

  REAL_CST

  VECTOR_CST

  STRING_CST

  COMPLEX_CST

  CONSTRUCTOR

  위 TREE node 들은 constant folding 이 필요없는 node 들이다.

- CONST_DECL

  DECL_INITIAL (t) 를 fold 시킨 node 를 반환한다.

- NOP_EXPR

  FLOAT_EXPR

  CONVERT_EXPR

  FIX_TRUNC_EXPR

  이 node 들의 경우, Type 의 변환이 이루어질 수도 있고, Type 의 변환이 이루어진 후 이에 대한 constant tree 를 반환할 수 있다. 여기서 이루어지는 constant fold 는 arg0 의 type 에 따라 그의 precision 을 변환해 주기 위해서 존재하기도 하고, arg0 이 constant 이느냐에 따라 반환되는 값이 달라지게 된다. 일반적인 Type 변환을 위해서는 convert () 함수를 사용하지만, arg0 가 상수일 경우, 그리고 이중으로 걸쳐진 conversion 이 아닌 이상 fold_constant () 함수를 이용하여 변환되게 된다.

- VIEW_CONVERT_EXPR

  이 node 의 arg0 또한 VIEW_CONVERT_EXPR 일 경우, 이에 대한 중복을 제거한다. 즉 build () 함수를 이용하여 arg0 의 arg0 을 operand 로 새 node 를 작성하게 된다.

- COMPONENT_REF

  arg0 이 CONSTRUCTOR 일 경우, arg0 의 CONSTRUCTOR_ELTS 중 arg1 과 같은 것이 있는지 찾아보고, 있을 경우에는 arg1 의 TREE_VALUE 를 반환하게 된다.

- RANGE_EXPR

  wins 값을 TREE_CONSTANT (t) 에 기록하게 되는데, 앞에서 설명했듯이, 내부 operand 들이 constant 인지에 따라 wins 값이 결정되게 된다.

- NEGATE_EXPR

이 node 의 경우, -5 혹은, $-(-a)$, $-(a-b)$ 와 같은 표현식을 다루게 된다. arg0 가 상수일 경우, 그 수를 실제 node 에 적용을 하여, INTEGER_CST 혹은 REAL_CST node 를 새로 구성하게 된다. 이 부분에서는 arg0 가 정수 상수인지 실수 상수인지가 중요한 문제가 될 수 있는데, GCC 에서 이를 어떻게 처리하는지는 앞의 "정수와 실수" 섹션에서 이미 언급하였으니 그것을 참고하기 바란다. arg0 가 NEGATE_EXPR 일 경우는 $-(-a)$ 와 같은 표현식이 나온 경우이며, 이를 a 를 나타내는 node, 즉 arg0 의 operand 를 반환한다. 마지막으로 $-(a-b)$ 표현식을 $(b-a)$ 표현식으로 변환하게 된다.

- ABS_EXPR

  절대값 |1| (내부가 변수가 아닌 상수일 경우) 은 TREE_UNSIGNED 가 정의되어 있거나, INT_CST_LT 로 검사할 수 있는 non-negative 일 경우, 절대값 수행을 할 필요가 없지만, 값이 $|-2|$ 와 같이 음수일 경우, 이 절대값은 negation 이 이루어져야 한다. 정수의 경우 neg_double () 함수를 통해 해당 operation 이 실행되며, 실수의 경우
  REAL_VALUE_NEGATIVE 를 통해서 시행된다. ||1|| 혹은 $|-a|$ 와 같은 표현일 경우, 적당한 ABS_EXPR 을 생성하여, 이에 대한 표현식을 간단하게 한다. 그 외에 수행할 부분이 없을 경우, 입력받은 표현식을 그대로 돌려준다.

- CONJ_EXPR

  arg0 의 TYPE 이 COMPLEX_TYPE 이 아닐 경우, convert () 함수를 이용하여 type 변환을 한다. arg0 가 COMPLEX_EXPR 일 경우, arg0 의 첫번째 인자와 negate_expr () 함수를 거친 두번째 인자로 COMPLEX_EXPR 를 새로이 생성한다. arg0 가 COMPLEX_CST 일 경우, build_complex 를 이용하여 arg0 의 첫번째 인자와 negate_expr () 함수를 거친 두번째 인자로 COMPLEX_EXPR 를 새로이 생성한다. arg0 가 PLUS_EXPR 혹은 MINUS_EXPR 일 경우, arg0 의 첫번째 인자로 CONJ_EXPR node 를 생성하고, arg0 의 두번째 인자로 CONJ_EXPR node 를 생성한 후, 새롭게 생긴 이 두 node 를 PLUS_EXPR 혹은 MINUS_EXPR 로 새롭게 묶는다. arg0 가 CONJ_EXPR 일 경우, arg0 의 첫번째 인자를 반환한다.

- BIT_NOT_EXPR

  (~1) 과 같이 arg0 가 constant 일 경우, build_int_2 () 함수를 이용하여 실제 계산한 값을 반환하게 되며, (~(~1)) 과 같은 표현식의 경우, 결과적으로 arg0 의 첫번째 operand 인 1 이 된다. 나머지 constant 가 아닐 경우, constant folding 이 발생하지 않는다.

- PLUS_EXPR

  $A+(-B)$ 표현식의 경우, $A-B$ 로 변환하고, $(-A)+B$ 표현식의 경우, $B-A$ 로 변환한다. $(A+0)$ 과 같은 표현식의 경우나 $(A+(-0)$ 과 같은 표현식의 경우, arg0 을 t 의 type 으로 convert () 함수를 이용하여 변환후, non_lvalue () 함수를 적용한다. (a & 2) + (b & 5) 표현식과 같이 arg0 과 arg1 의 표현식이 BIT_AND_EXPR 이고 arg0 의 두번째 operand 와 arg1 의 두번째 operand 가 공통인 bit 를 가지고 있지 않는 constant 일 경우, 좀 더 효율적인 간단화를 시키기 위해 BIT_IOR_EXPR 로 취급을 한다. 이를 위해, bit_ior 라벨로 goto 하게 된다. $(((2*4)+a)+(1*3))$ 과 같이 (plus (plus (mult) (foo)) (mult)) 표현식을 (((2*4)+(1*3)) + a)) 과 같이 (plus (plus (mult) (mult)) (foo)) 로 재결합함으로써 factoring 경우에 이득을 얻을 수 있도록 한다. $(A*C)+(B*C)$ 와 같은 표현식을 $(A+B)*C$ 와 같이 재구성한다. 실제 계산을 하는 값이 정수인지 실수인지에 따라 위의 변환이 적용될 수도 안될 수도 있다는 것을 알기 바란다. 지금까지 해당 규칙이 존재하지 않았다면, fold () 함수에 선언되어 있는 라벨 bit_rotate 로 넘어가게 된다. 이 부분에서는 다음과 같은 operation 이 일어나게 된다.

  $(A << C1)+(A >> C2)$ 와 같은 표현식이고, A 가 unsigned 이고 $C1+C2$ 가 A 의 size 일 경우 A 를 $C1$ bit 만큼 rotate 시키는 것이므로 이를 간단화한다.

  $(A << B)+(A >> (Z-B))$ 와 같은 표현식이고, A 가 unsigned 이고 Z 가 A 의 size 일 경우, A 를 B bit 만큼 rotate 시키는 것으므로 이를 간단화한다.

이제 associate 라벨의 수행을 하게 되는데, arg0 과 arg1 을 변수, 상수, literal 로 분리해서 다시 재결합함으로써 literal 이 뒤에서 재결합할 수 있는 기회를 증가 시키거나 상수 혹은 literal 의 합에 대한 relocatable 표현식들을 생성하는 기회를 증가시키기 위해 표현식을 처리한다. (2 + 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- MINUS_EXPR

  $A - (-B)$ 인 경우, $A + B$ 로 변환한다. $(-A) - CST$ 인 경우, 여기서 CST 는 constant 를 가르킴, $(-CST) - A$ 로 변환한다. $0 - A$ 인 경우, negate_expr () 함수를 이용하여 해당 표현식의 negation node 를 생성한다. $A - 0$ 인 경우, arg0 을 t 의 type 으로 변환 후, non_lvalue () 함수를 적용시킨다. $(A * C) - (B * C)$ 와 같은 표현식을 $(A - B) * C$ 로 반환한다. &x − &x 와 같은 표현식을 t 의 type 을 사용하여 integer_zero_node 로 변환시킨다. 이 표현식의 경우, 결과적으로 0 이기 때문이다. 이제 associate 라벨의 수행을 하게 되는데, arg0 과 arg1 을 변수, 상수, literal 로 분리해서 다시 재결합함으로써 literal 이 뒤에서 재결합할 수 있는 기회를 증가 시키거나 상수 혹은 literal 의 합에 대한 relocatable 표현식들을 생성하는 기회를 증가시키기 위해 표현식을 처리한다. (2 − 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- MULT_EXPR

  $(-A) * (-B)$ 표현식의 경우, $A * B$ 로 변경한다. $a * 0$ 일 경우, omit_one_operand () 함수를 이용하여 0 을 반환하게 된다. $a * 1$ 일 경우, arg0 의 type 을 변경후, non_lvalue () 함수를 적용해 반환한다. $(a * (1 << b))$ 일 경우 혹은 $((1 << b) * a)$ 일 경우, 각각 $(a << b)$ 와 $(b << a)$ 로 변경한다. $(X + 7) * 4$ 와 같은 표현식은 $X * 4 + 28$ 형태와 같이 간단화 되어지는데, 이러한 operation 은 extract_muldiv () 함수에서 이루어진다. $x * 2$ 는 $x + x$ 이기 때문에, 이와 같이 변환시켜준다. 이제 associate 라벨의 수행을 하게 되는데, arg0 과 arg1 을 변수, 상수, literal 로 분리해서 다시 재결합함으로써 literal 이 뒤에서 재결합할 수 있는 기회를 증가 시키거나 상수 혹은 literal 의 합에 대한 relocatable 표현식들을 생성하는 기회를 증가시키기 위해 표현식을 처리한다. (2 ∗ 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- BIT_IOR_EXPR

  (a | 0xffffffff) 인 경우, omit_one_operand () 함수를 적용해 arg1 을 t 의 type 으로 변환 후 non_lvalue () 함수를 적용해 반환하게 되는데, arg0 이 TREE_SIDE_EFFECTS 를 가지고 있을 경우, 우리는 반드시 그것을 재평가 해야하기 때문에 arg0 와 arg1 을 이용하여 새로운 COMPOUND_EXPR node 를 만들어 반환하게 된다. (a | 0x0) 인 경우, arg0 을 t 의 type 으로 변환 후, non_lvalue () 함수를 적용해 반환한다. $(A|B)$ & $(A|C)$ 와 같은 표현식일 경우, distribute_bit_expr () 함수를 이용하여 A | (B & C) 로 변환한다. (~a) | (~b) 의 경우, ~(a & b) 로 변환한다. NAND instruction 이 없는 machine 들에 대해 좀 더 효율적인 code 를 만들어낸다. 지금까지 해당 규칙이 존재하지 않았다면, fold () 함수에 선언되어 있는 라벨 bit_rotate 로 넘어가게 된다. 이 부분에서는 다음과 같은 operation 이 일어나게 된다.

  $(A << C1) + (A >> C2)$ 와 같은 표현식이고, A 가 unsigned 이고 $C1 + C2$ 가 A 의 size 일 경우 A 를 $C1$ bit 만큼 rotate 시키는 것이므로 이를 간단화한다.

  $(A << B) + (A >> (Z - B))$ 와 같은 표현식이고, A 가 unsigned 이고 Z 가 A 의 size 일 경우, A 를 B bit 만큼 rotate 시키는 것으므로 이를 간단화한다.

  이제 associate 라벨의 수행을 하게 되는데, arg0 과 arg1 을 변수, 상수, literal 로 분리해서 다시 재결합함으로써 literal 이 뒤에서 재결합할 수 있는 기회를 증가 시키거나 상수 혹은 literal 의 합에 대한 relocatable 표현식들을 생성하는 기회를 증가시키기 위해 표현식을 처

리한다. (2 | 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- BIT_XOR_EXPR

  (a ^ 0x0) 와 같은 표현식일 경우, arg0 을 t 의 type 으로 convert () 함수를 이용하여 변환한 후, non_lvalue () 함수를 적용한다. (a ^ 0xffffffff) 표현식일 경우, arg0 을 t 의 type 을 이용하여, BIT_NOT_EXPR node 를 생성하여, 새로 folding 한다. (a & 2) ^ (b & 5) 표현식과 같이 arg0 과 arg1 의 표현식이 BIT_AND_EXPR 이고 arg0 의 두번째 operand 와 arg1 의 두번째 operand 가 공통인 bit 를 가지고 있지 않는 constant 일 경우, 좀 더 효율적인 간단화를 시키기 위해 BIT_IOR_EXPR 로 취급을 한다. 이를 위해, bit_ior 라벨로 goto 하게 된다. 지금까지 해당 규칙이 존재하지 않았다면, fold () 함수에 선언되어 있는 라벨 bit_rotate 로 넘어가게 된다. 이 부분에서는 다음과 같은 operation 이 일어나게 된다.

  $(A << C1) + (A >> C2)$ 와 같은 표현식이고, A 가 unsigned 이고 $C1 + C2$ 가 A 의 size 일 경우 A 를 $C1$ bit 만큼 rotate 시키는 것이므로 이를 간단화한다.

  $(A << B) + (A >> (Z - B))$ 와 같은 표현식이고, A 가 unsigned 이고 Z 가 A 의 size 일 경우, A 를 B bit 만큼 rotate 시키는 것으므로 이를 간단화한다.

  이제 associate 라벨의 수행을 하게 되는데, arg0 과 arg1 을 변수, 상수, literal 로 분리해서 다시 재결합함으로써 literal 이 뒤에서 재결합할 수 있는 기회를 증가 시키거나 상수 혹은 literal 의 합에 대한 relocatable 표현식들을 생성하는 기회를 증가시키기 위해 표현식을 처리한다. (2 | 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- BIT_AND_EXPR

  (a & 0xffffffff) 인 경우, arg0 을 t 의 type 으로 변환 후, non_lvalue () 함수를 적용해 반환한다. (a & 0x0) 인 경우, omit_one_operand () 함수를 적용해 arg1 을 t 의 type 으로 변환 후 non_lvalue () 함수를 적용해 반환하게 되는데, arg0 이 TREE_SIDE_EFFECTS 를 가지고 있을 경우, 우리는 반드시 그것을 재평가 해야하기 때문에 arg0 와 arg1 을 이용하여 새로운 COMPOUND_EXPR node 를 만들어 반환하게 된다. $(A|B)$ & $(A|C)$ 와 같은 표현식일 경우, distribute_bit_expr () 함수를 이용하여 A | (B & C) 로 변환한다. ((int)c & 0x377) 와 같은 표현식에서 c 가 unsigned char 일 경우, $(int)c$ 로 간단화시킨다. (~a) & (~b) 의 경우, ~(a | b) 로 변환한다. 이것은 NOR instruction 이 없는 machine 에 대해 좀 더 효율적인 code 를 만들어낸다. 이제 associate 라벨의 수행을 하게 되는데, arg0 과 arg1 을 변수, 상수, literal 로 분리해서 다시 재결합함으로써 literal 이 뒤에서 재결합할 수 있는 기회를 증가 시키거나 상수 혹은 literal 의 합에 대한 relocatable 표현식들을 생성하는 기회를 증가시키기 위해 표현식을 처리한다. (2 & 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- BIT_ANDTC_EXPR

  arg0 가 0xffffffff 인 경우, arg1 의 type 을 변환 후, non_lvalue () 함수를 적용해 반환한다. arg0 이 0 인 경우, arg1 이 side effect 를 가지지 않는 이상, arg0 을 반환한다. arg1 이 INTEGER_CST 인 경우, arg1 을 BIT_NOT_EXPR node 로 만들고, code 를 BIT_AND_EXPR 로 변경후, 위의 BIT_AND_EXPR 처리 부분으로 goto 한다. 위의 모든 사항에 대해 해당되는 것이 없을 경우, binary 라벨로 goto 한다.

- RDIV_EXPR

  $a/0$ 과 같은 경우를 처리한다. $(-A)/(-B)$ 와 같은 표현식일 경우, A/B 로 변환을 한다. $a/1$ 과 같은 표현식일 경우, arg0 을 t 의 type 으로 변환 후, non_lvalue () 함수를 적용해 반환한다. $A/B/C$ 와 같은 표현식을 $A/(B * C)$ 로 변환한다. $A/(B/C)$ 와 같은 표현식을 $(A/B) * C$ 로 변환한다. (2 / 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직

접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- TRUNC_DIV_EXPR

  ROUND_DIV_EXPR

  FLOOR_DIV_EXPR

  CEIL_DIV_EXPR

  EXACT_DIV_EXPR

  $a/1$ 과 같은 표현식일 경우, arg0 을 t 의 type 으로 변환 후, non_lvalue () 함수를 적용해 반환한다. $a/0$ 과 같은 경우를 처리한다. 만약 arg0 가 arg1 의 곱일 경우, 가장 빠른 나누기 operation 인 EXACT_DIV_EXPR node 로 새롭게 쓴다. multiple_of_p () 함수를 통해서 arg0 가 arg1 의 곱인지를 결정한다. $((X*8)+(Y*16))/4$ 와 같은 표현식을 extract_muldiv () 함수를 이용하여 $(X*2)+(Y*4)$ 와 같은 표현식으로 변환한다. (2 / 5) 와 같이 둘다 상수일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- CEIL_MOD_EXPR

  FLOOR_MOD_EXPR

  ROUND_MOD_EXPR

  TRUNC_MOD_EXPR

  (a % 1) 와 같은 표현식일 경우, omit_one_operand () 함수를 이용하여, arg0 를 돌려주게 된다. (a % 0) 과 같은 표현식일 경우, t 를 그대로 돌려준다. arg1 이 INTEGER_CST 일 경우, extract_muldiv () 를 실행하여, arg0 에 대한 simplification 을 수행한다.

- LSHIFT_EXPR

  RSHIFT_EXPR

  LROTATE_EXPR

  RROTATE_EXPR

  C 언어의 경우, LROTATE_EXPR 와 RROTATE_EXPR node 를 다루지 않기 때문에 이 해당 node 를 위한 simplification 에 대해서는 언급하지 않겠다. LSHIFT_EXPR 와 RSHIFT_EXPR node 에 대해서는 arg0 와 arg1 가 둘 다 constant 일 경우, const_binop () 함수를 통해서 직접 계산한 값으로 새로운 node 를 만든 후, 원래 표현식과 같은 type 을 가지도록 convert () 시켜준다.

- MIN_EXPR

  MAX_EXPR

  이 node 의 경우, C++ 에서 사용되는 minimum 과 maximum operator 들이기 때문에 여기에서는 언급하지 않는다.

- TRUTH_NOT_EXPR

  arg0 를 invert_truthvalue () 함수를 거쳐, t 의 type 으로 변환 후 반환한다.

- TRUTH_ANDIF_EXPR

만약 arg0 가 constant zero 이면 그것을 반환한다. 그렇지 않을 경우, 아래의 TRUTH_AND_EXPR 부분으로 넘어간다.

- TRUTH_AND_EXPR

  arg0 이 INTEGER_CST 이고 arg0 이 0 이 아닐 때, arg1 을 t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg1 이 INTEGER_CST 이고 arg1 이 0 이 아닐 때, 그리고 code 가 TRUTH_ANDIF_EXPR 가 아니거나 arg0 가 side effect 를 가지고 있지 않을 때, arg0 을 t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg1 이 0 일 경우, arg0 이 side effect 를 가지지 않는 한, arg1 을 반환한다. 비슷하게 arg0 에 대해서도 마찬가지다. $(A||B)$ && $(A||C)$ 와 같은 표현식을 $A||(B$ && $C)$ 로 변환한다. 나머지 부분에서는 range comparison 를 생성할 수 있는지와 fold_truthop () 함수를 이용하여 lhs 가 rhs 와 비슷할 경우, 이를 merge 하려고 한다.

- TRUTH_ORIF_EXPR

  arg0 이 INTEGER_CST 이고 0 이 아닐 경우, arg0 를 type 변환후 반환한다. 아닐 경우, fold () 함수에서 TRUTH_OR_EXPR node 를 처리하는 부분으로 넘어간다.

- TRUTH_OR_EXPR

  arg0 이 INTEGER_CST 이고 arg0 이 0 일 때, arg1 을 t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg1 이 INTEGER_CST 이고 arg1 가 0 이며, t 의 TYPE_CODE 가 TRUTH_ORIF_EXPR 가 아니거나, arg0 가 TREE_SIDE_EFFECTS 를 가지지 않을 때, arg0 을 t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg1 이 INTEGER_CST 이고 arg1 이 0 이 아닐 경우, omit_one_operand () 함수를 호출한다. arg0 이 INTEGER_CST 이고 arg0 이 0 이 아닐 경우, omit_one_operand () 함수를 호출한다. $(A||B)$ && $(A||C)$ 와 같은 표현식을 $A||(B$ && $C)$ 로 변환한다. 나머지 부분에서는 range comparison 를 생성할 수 있는지와 fold_truthop () 함수를 이용하여 lhs 가 rhs 와 비슷할 경우, 이를 merge 하려고 한다.

- TRUTH_XOR_EXPR

  arg0 이 0 일 경우, arg1 을 t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg1 이 0 일 경우, arg0 을 t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg0 이 1 일 경우, arg1 을 invert_truthvalue () 함수에 적용한 후, t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다. arg1 이 1 일 경우, arg0 을 invert_truthvalue () 함수에 적용한 후, t 의 type 으로 변환하고 non_lvalue () 함수를 거친후 반환한다.

- EQ_EXPR

  NE_EXPR

  LT_EXPR

  GT_EXPR

  LE_EXPR

  GE_EXPR

  (-a) CMP (-b) 와 같은 표현식일 경우, b CMP a 로 변환한다. 여기서 CMP 는 이 해당 TREE_CODE 를 대표하는 기호이다. (-a) CMP CST 와 같은 표현식일 경우, a swap(CMP) (-CST) 로 변환한다. 여기서 CST 는 constant 이며, swap () 은 기호 CMP 를 바꾸는 것을 의미한다. 예를 들어, CMP 가 GT_EXPR 일 경우, swap 되어지면 LT_EXPR 가 된다. a CMP (-0) 와 같은 표현식을 a CMP 0 로 변환한다. $foo++ == CONST$ 와 같은 표현식을 $++foo == CONST + INCR$ 와 같이 변환한다. $foo-- == CONST$ 와 같은 표

현식을 $--foo == CONST - INCR$ 와 같이 변환한다. $X >= CST$ 와 같은 표현식을 CST 가 양수일 경우, $X > (CST - 1)$ 와 같이 변환한다. $(X + 2) == 1)$ 와 같은 표현식을 $X == (1 + 2)$ 변환한다. $-$ 에 대해서 마찬가지로 $(X - 2) == 1)$ 일 경우, $X == 1 - 2)$ 로 변환한다. NEGATE_EXPR 에 대해서도 비슷한데, $-(X + 1) == 2$ 와 같은 표현식을 $(X + 1) == -2$ 로 변환하게 된다. 이것은 == 일 때 뿐만 아니라, != 일 경우에도 해당한다. $X - Y == 0$ 와 같은 표현식을 $X == Y$ 로 변환한다. != 표현식에 대해서도 마찬가지이다. 상수와 MIN_EXPR 혹은 MAX_EXPR 비교를 하는 T 가 존재할 경우 아래와 같은 operation 을 수행한다.

오직 EQ_EXPR 와 GT_EXPR 만 다루며, 나머지는 logical simplification 을 사용한 recursive call 을 통해 처리한다.
$MAX(X, 0) == 0$ 를 $X <= 0$ 로 변환한다.
$MAX(X, 0) == 5$ 를 $X == 5$ 로 변환한다.
$MAX(X, 0) == -1$ 를 $false$ 로 변환한다.
$MIN(X, 0) == 0$ 를 $X >= 0$ 로 변환한다.
$MIN(X, 0) == 5$ 를 $false$ 로 변환한다.
$MIN(X, 0) == -1$ 를 $X == -1$ 로 변환한다.
$MAX(X, 0) > 0$ 를 $X > 0$ 으로 변환한다.
$MAX(X, 0) > 5$ 를 $X > 5$ 로 변환한다.
$MAX(X, 0) > -1$ 를 $true$ 로 변환한다.
$MIN(X, 0) > 0$ 를 $false$ 로 변환한다.
$MIN(X, 0) > 5$ 를 $false$ 로 변환한다.
$MIN(X, 0) > -1$ 를 $X > -1$ 로 변환한다.

$(1 << foo)$ & bar 와 같은 표현식을 $(bar >> foo)$ & 1 로 변환한다. $(A \& C) == C$ 와 같은 표현식이고 C 가 2 의 배수이면, $(A \& C)! = 0$ 로 변환한다. $X < (1 << Y)$ 와 같은 표현식이고 X 가 unsigned 일 경우, $X >> Y == 0$ 와 같이 변환한다. 비슷하게 >= 는 != 로 변환한다. 각 node 가 서로 같은 node 들을 비교하고 있는지, 즉 arg0 과 arg1 이 같은지 검사한다. $(a > b) == 0$ 와 같은 표현식이나, $((x > y) - (y > x)) > 0$ 와 같은 표현식이 간단화될 수 있는지 확인하고, 간단화할 수 있을 경우, 간단화한다. COMPONENT_REF 나 BIT_FIELD_REF 에 대한 비교 구문일 경우, 간단화하는데, optimize_bit_field_compare () 함수를 통해 수행한다. $strlen(ptr) == 0$ 와 같은 표현식을 $*ptr == 0$ 로 변환한다. 비슷하게 $strlen(ptr)! = 0$ 와 같은 표현식의 경우 $*ptr! = 0$ 로 변환한다.

- COND_EXPR

CST ? A : B 와 같은 표현식일 경우, arg0 의 값에 따라 pedantic_non_lvalue () 함수를 거쳐 적당히 처리한다. A ? B : B 와 같이 arg1 과 arg2 의 표현식 node 가 같을 경우, pedantic_omit_one_operand () 함수를 이용하여 둘 중 하나를 반환한다. A op B ? A : C 와 같은 표현식의 경우, operation 과 B 와 C 의 값에 따라 간단화를 수행한다.

A op 0 ? A : -A 일 경우, 비교 operator 에 따라 A 혹은 -A, abs (A), -abs (A) 일 것이다. A != 0 ? A : 0 일 경우, 이것은 단순히 A 이고, == 일 경우 항상 0 이다. A op B ? A : B 일 경우, 이것은 operator 에 따라, A 혹은 B, min (A, B) max (A, B) 이다. A op C1 ? A : C2 이고, C1 과 C2 가 constant 정수일 경우, 간단화를 수행할 수 있는데, 예를 들어 C1 이 1 보다 작거나, C2 보다 1 이상 클 경우, MIN 혹은 MAX 로 변환되어질 수 있을 것이다. 이에 대해 자세히 살펴보면 아래와 같다.

operator 가 EQ_EXPR 일 경우, A 를 C1 으로 대체한다.
LT_EXPR node 이고 C1 이 C2 + 1 일 경우, 이것은 $min(A, C2)$ 이다.
LE_EXPR node 이고 C1 이 C2 - 1 일 경우, 이것은 $min(A, C2)$ 이다.
GT_EXPR node 이고 C1 이 C2 - 1 일 경우, 이것은 $max(A, C2)$ 이다.
GE_EXPR node 이고 C1 이 C2 + 1 일 경우, 이것은 $max(A, C2)$ 이다.

A ? 1 : 0 일 경우, A 로 간단히 한다. A & 2 ? 2 : 0 와 같은 표현식을 찾아, A & 2 로 간단화한다.

- COMPOUND_EXPR

  arg0 이 side effect 를 가지고 있거나, pedantic 일 경우, 간단화를 수행하지 않는다. arg1 이 0 일 경우, NOP_EXPR node 를 수행한다. 위의 경우 중 해당 사항이 없을 경우, arg1 을 t 의 type 으로 변환 후 반환한다.

- COMPLEX_EXPR

  이 tree node 에 대한 arg0 과 arg1 이 모두 상수일 경우, build_complex () 함수를 이용하여, COMPLEX_CST node 를 생성하게 된다.

- REALPART_EXPR

  arg0 이 COMPLEX_EXPR 일 경우, omit_one_operand () 함수를 호출한다. arg0 이 COMPLEX_CST 일 경우, arg0 의 TREE_REALPART 를 반환한다. arg0 이 PLUS_EXPR 혹은 MINUS_EXPR 일 경우, 이를 folding 한다.

- IMAGPART_EXPR

  arg0 의 type 이 COMPLEX_TYPE 이 아닐 경우, convert () 함수를 통해, integer_zero_node 를 반환한다. arg0 이 COMPLEX_EXPR 일 경우, omit_one_operand () 함수를 호출한다. arg0 이 COMPLEX_CST 일 경우, arg0 의 TREE_IMAGPART 를 반환한다. arg0 이 PLUS_EXPR 혹은 MINUS_EXPR 일 경우, 이를 folding 한다.

- CLEANUP_POINT_EXPR

  이에 대한 정확한 예제를 찾을 수 없어 완료할 수 없었습니다.

- CALL_EXPR

  Built-in function 함수들에 대한 constant folding 을 수행하는데, 이 수행은 fold_builtin () 함수에서 수행되며, __builtin_constant_p () built-in 함수와 __builtin_classify_type () 함수, __builtin_strlen () 함수에 대해 수행한다. 이러한 constant folding 을 위한 함수들은 $prefix/gcc/builtins.c 파일에 정의되어 있으며, argument 가 0 혹은 1, 혹은 다른 상수 형태인 것에 대해 처리를 한다.