Fortran Optimization

Roger Sayle

NextMove Software
Santa Fe, New Mexico, USA

16th May 2010

Efficient Array Initialization (Part 1)

Often fortran array initialization, assigning the value zero, can be efficiently implemented for contiguous arrays using memset. Previously, the scalarizer in gfortran generated a loop (or nested loops) for such assignments.

```
integer :: a(20)
a(:) = 0
```

Previously gfortran generated the following [with -fdump-tree-original]

```
int8 S.0;

S.0 = 1;
while (1)
  {
    if (S.0 > 20) goto L.1; else (void) 0;
    (*a)[NON_LVALUE_EXPR <S.0> + -1] = 0;
    S.0 = S.0 + 1;
  }
L.1:;
```

Once the optimization was implemented, gfortran generated:

```
(void) __builtin_memset ((void *) a, 0, 80);
```

Counter Example #1

```
program t
 real :: arr(4)
 call a(arr(1:4:2))
 print *, arr
contains
subroutine a(b)
  real :: b(:)
  b(:) = 0
end subroutine
end program
```

TODO: Recognize reverse order initialization “a(20:1:-1) = 0”
TODO: Recognize subsequence initialization “a(20:40) = 0”
TODO: Recognize variable length initialization “a(1:n) = 0”

http://gcc.gnu.org/ml/fortran/2006-12/msg00305.html

Efficient Array Initialization (Part 2)

This patch improves the way that gfortran expands constant array constructors to GIMPLE. It's probably best explained via a simple example; consider the following code:

```
program v
  integer :: x(4)
  x(:) = (/ 3, 1, 4, 1 /)
end program
```

Currently, this is expanded by the gfortran front-end as:

```
MAIN__ ()
{
  int4 x[4];

  _gfortran_set_std (70, 127, 0);
  {
    int4 A.1[4];
    struct array1_int4 atmp.0;

    atmp.0.dtype = 265;
    atmp.0.dim[0].stride = 1;
    atmp.0.dim[0].lbound = 0;
    atmp.0.dim[0].ubound = 3;
    atmp.0.data = (void *) &A.1;
    atmp.0.offset = 0;
    {
      static int4 data.3[4] = {3, 1, 4, 1};

      __builtin_memcpy (&(*(int4[0:] *) atmp.0.data)[0], &data.3, 16);
    }
    {
      int8 S.4;

      S.4 = 0;
      while (1)
        {
          if (S.4 > 3) goto L.1;
          x[NON_LVALUE_EXPR <S.4>] = (*(int4[0:]*)atmp.0.data)[NON_LVALUE_EXPR <S.4>];
          S.4 = S.4 + 1;
        }
      L.1:;
    }
  }
}
```

Notice that we create a temporary array A, that requires an array descriptor, that we then populate via a memcpy call, and then use this in the scalarization of the assignment. Hence we require three arrays; "A", "data" and "x", and two copies.

With the patch below, we now generate the much simpler:

```
MAIN__ ()
{
 int4 x[4];

 _gfortran_set_std (70, 127, 0);
 {
   static int4 A.0[4] = {3, 1, 4, 1};

   {
     int8 S.1;

     S.1 = 1;
     while (1)
       {
         if (S.1 > 4) goto L.1;
         x[S.1 + -1] = A.0[S.1 + -1];
         S.1 = S.1 + 1;
       }
     L.1:;
   }
 }
}
```

where the array A, is initialized as a "static const", and then used directly during the scalarization. This requires only two arrays and a single copy. When the array constructor is large, this can result in significant time and space savings, for the common case.

The optimization is that when the array constructor consists entirely of constant elements, that specify it's entire size, it's possible to pre-initialize "A" if we use gfortran's notion of a non-descriptor array type.

http://gcc.gnu.org/ml/fortran/2007-01/msg00174.html

Efficient Array Initialization (Part 3)

The following minor tweak modifies two places in the gfortran front-end where we generate constant arrays. For example, in code such as:

```
program v
  integer :: x(4)
  x = (/ 3, 1, 4, 1 /)
end program
```

We internally generate the gimple:

```
{
  static int4 data.3[4] = {3, 1, 4, 1};
  __builtin_memcpy (&(*(int4[0:] *) atmp.0.data)[0], &data.3, 16);
}
```

where the array is marked "static" but not "const". This disables some middle-end optimizations that can take advantage of the fact that the array never changes, and causes the compiler to output it in a ".data" instead of a ".rodata" section.

This trivial patch causes gfortran to set the TREE_READONLY field on the DECL for "data.3" which allows good things to happen, and places these potentially large constant arrays in .rodata.

http://gcc.gnu.org/ml/fortran/2007-01/msg00114.html

Efficient Array Initialization (Part 4)

The following patch is a generalization of my recent constant array constructor improvements to avoid creating/copying temporary arrays when using EXPR_ARRAYs with rank greater than one. Previously, we'd avoid the need for a temporary only for one dimensional arrays, but the with a little effort this can be generalized to matrices, grids, etc.. that result from using the RESHAPE intrinsic.

This enhancement greatly simplifies the code we generate for:

```
integer :: x(2,2)
if (any(x(:,:) .ne. reshape ((/ 3, 1, 4, 1 /), (/ 2, 2 /))))
  call abort ()
```

[Whilst I was there I also noticed the above ANY intrinsic was generating generic of the form "(x[i] != A[i]) != 0", which is caused by not folding the comparison against zero. This is also fixed in the patch below.]

http://gcc.gnu.org/ml/fortran/2007-02/msg00228.html

Efficient Array Initialization (Part 5)

The following patch to the gfortran front-end improves the way that array constructors are passed as function arguments. The current implementation copies the entire array constructor into a temporary array to allow the temporary array's descriptor to be passed to the function. In the patch below we improve this for constant array constructors (of non-character types) by creating a descriptor for the array constructor (much like we do for regular array variables) and pass that. As with regular array variables this avoids any copying loops.

When combined with my previous patch to correctly detect dependencies on array constructors we can now expand the following code without any array temporaries.

```
integer :: x(6)
x = cshift((/ 1, 2, 3, 4, 5, 6/), 2)
end
```

With these two pending patches we get:

```
int4 x[6];
static int4 C.1362 = 2;
struct array1_int4 parm.2;
static int4 A.1[6] = {1, 2, 3, 4, 5, 6};
struct array1_int4 parm.0;

parm.0.dtype = 265;
parm.0.dim[0].lbound = 1;
parm.0.dim[0].ubound = 6;
parm.0.dim[0].stride = 1;
parm.0.data = (void *) &x[0];
parm.0.offset = 0;
parm.2.dtype = 265;
parm.2.dim[0].lbound = 1;
parm.2.dim[0].ubound = 6;
parm.2.dim[0].stride = 1;
parm.2.data = (void *) &A.1[0];
parm.2.offset = 0;
_gfortran_cshift0_4 (&parm.0, &parm.2, &C.1362, 0B);
```

Compare this to current mainline where we generate:

```
int4 x[6];
int4 A.5[6];
struct array1_int4 atmp.4;
static int4 C.1364 = 2;
int4 A.2[6];
struct array1_int4 atmp.1;
static int4 A.0[6] = {1, 2, 3, 4, 5, 6};
```

```
atmp.1.dtype = 265;
atmp.1.dim[0].stride = 1;
atmp.1.dim[0].lbound = 0;
atmp.1.dim[0].ubound = 5;
atmp.1.data = (void *) &A.2;
atmp.1.offset = 0;
{
  int8 S.3;

  S.3 = 0;
  while (1)
    {
      if (S.3 > 5) goto L.1;
      (*(int4[0:] *) atmp.1.data)[S.3] = A.0[S.3];
      S.3 = S.3 + 1;
    }
  L.1:;
}
atmp.4.dtype = 265;
atmp.4.dim[0].stride = 1;
atmp.4.dim[0].lbound = 0;
atmp.4.dim[0].ubound = 5;
atmp.4.data = (void *) &A.5;
atmp.4.offset = 0;
_gfortran_cshift0_4 (&atmp.4, &atmp.1, &C.1364, 0B);
{
  int8 S.6;

  S.6 = 0;
  while (1)
    {
      if (S.6 > 5) goto L.2;
      x[NON_LVALUE_EXPR <S.6>] = (*(int4[0:] *) atmp.4.data)[S.6];
      S.6 = S.6 + 1;
    }
  L.2:;
}
```

http://gcc.gnu.org/ml/fortran/2007-02/msg00030.html

Efficient Array Copies (Part 1)

Suitable full array assignments of the form “a(:) = b(:)” may be efficiently implemented using memcpy. This optimization is currently only implemented for arrays that don't require a descriptor (i.e. whose bounds are known at compile-time). Additionally, gfortran supports “arrays of derived types containing allocatable components” which can't be directly copied using a block move, but require what the C++ front-end might call a copy constructor. Hence the memcpy optimization applies only to arrays of numeric/logical types.

http://gcc.gnu.org/ml/fortran/2007-01/msg00113.html

Efficient Array Copies (Part 2)

The following patch implements an extension/enhancement of my original __buitin_memcpy patch that was suggested/recommended by Paul Thomas. This extends the set of types considered "copyable" with memcpy to additionally include user derived types that don't contain allocatable components. This allows us to use memcpy in the following testcase.

```
type t
  logical valid
  integer :: x, y
end type
type (t) :: s(5)
type (t) :: d(5)

d(:) = s(:)
```

The trick is to use the same logic as in gfc_trans_scalar_assign, and check the ts.derived->alloc_comp field, which indicates whether the type contains any allocatable fields that require special handling. Presumably, the front-end sets this field correctly for derived types containing derived types, otherwise both the existing gfc_trans_scalar_assign and the new code below contain potential bugs. Likewise, for derived types containing character fields.

Efficient Array Copies (Part 3)

The following patch builds upon the two previous patches to (1) use statically initialized arrays to implement array constructors and (2) use __builtin_memcpy when copying one array to another. With this installment we reuse the functionality of the previous two patches, to now use __builtin_memcpy when assigning an array from a constant array constructor.

Currently, the testcase below

```
integer :: x(4)
x(:) = (/ 3, 1, 4, 1 /)
```

is expanded as

```
static int4 A.0[4] = {3, 1, 4, 1};

{
  int8 S.1;

  S.1 = 1;
  while (1)
    {
      if (S.1 > 4) goto L.1;
      (*x)[S.1 + -1] = A.0[S.1 + -1];
      S.1 = S.1 + 1;
    }
  L.1:;
}
```

but with this patch we now generate

```
static int4 A.0[4] = {3, 1, 4, 1};
(void) __builtin_memcpy ((void *) x, (void *) &A.0, 16);
```

Array Assignment Dependency Checking (Part 1)

An interesting observation/insight made in Roth2001 is that because scalarization only cares about detection of “loop-carried true dependencies”, all scalar subscripts can be ignored. Implementing this turned out to be a one line change, but a subtle one.

http://citeseer.ist.psu.edu/roth01evaluation.html
"Evaluation of Array Syntax Dependence Analysis"
Gerald Roth, Gonzaga University,
Proceedings of the International Conference on Parallel and
Distributed Processing Techniques and Applications (PDPTA 2001),
Las Vegas, Nevada, June 2001.

Example #1
```
subroutine foo(a,i,j)
  integer, dimension (4,4) :: a
  integer :: i, j

  where (a(i,:) .ne. 0)
    a(j,:) = 1
  endwhere
end subroutine
```

In this example, the is no loop carried dependence between a(i,:) and a(j,:).

Example #2
```
subroutine foo(a,x,y)
  integer, dimension (4,4,4) :: a
  integer :: i, j

  where (a(x,2,:) .ne. 0)
    a(y,3,:) = 1
  endwhere
end subroutine
```

In this example, there is not loop dependence because the second index, where “2” != “3” guarantees independence.

http://gcc.gnu.org/ml/fortran/2006-03/msg00039.html
http://gcc.gnu.org/ml/gcc-patches/2006-03/msg00209.html

Array Assignment Dependency Checking (Part 2)

I've been looking into futher improving some of gfortran's dependency analysis, and whilst looking through which routines could be reused or adapted, I discovered a curious omission in gfc_is_same_range. When comparing ranges, we currently test the stride and the lower bound, but curiously not the upper bound! Hence a(:5) is always considered the same range as a(:4)??

http://gcc.gnu.org/ml/fortran/2006-02/msg00560.html

Array Assignment Dependency Checking (Part 3)

The following patch makes some improvements to the gfortran front-end's dependency analysis. Currently gfortran is transitioning from dependency analysis based in the scalarizer (checking gfc_ss*) to analysis based on expressions (checking gfc_expr*) [see the comment above trans-array.c:gfc_conv_resolve_dependencies]. The old API uses gfc_conv_resolve_dependencies (which calls gfc_dep_resolver), whilst the new API uses dependency.c:gfc_check_dependency.

The significant bit of the patch below is to hook the intelligence of the array reference dependency checking in gfc_dep_resolver into the preferred gfc_check_dependency API.

As an example of the improvement this provides, gfc_check_dependency could previously determine that array refs "a" and "a" were equal and therefore hadn't a dependency, but couldn't tell that "a(:)" and "a(:)" or that "a(1:3)" and "a(1:3)" were equal. A convenient model system for gfc_check_dependency is the F90 WHERE statement generation, where its easy to construct tests that check whether a dependency was identified by scanning the output for the allocation of a temporary array.

Whilst double checking this hook-up, I discovered a potential problem in gfc_check_element_vs_element. Where we didn't check for a -2 return value from gfc_dep_compare_expr, meaning that we couldn't determine the equality or ordering of the operands at compile-time (for example, with two distinct variables). In this eventually, its unsafe to assume that array refs have no dependency. By assuming that these variables are not equal, we current skip checking the later dimensions, which may have a conflict.

Consider,

```
a(i,1:3) = a(j,0:2)
```

This has a dependency, because if i == j, we need to confirm that the second dimensions don't overlap. This is fixed below by cleaning up gfc_check_element_vs_element. I also changed the code to assume both arguments are ARRAY_REFs to consistently match the implementations of both gfc_check_element_vs_section and gfc_check_section_vs_section.

Finally, because I'm paranoid, I added an extra test or two to gfc_dep_resolver to ensure that both reference chains have the same depth, and assume a dependency if they don't match up.

http://gcc.gnu.org/ml/fortran/2006-03/msg00028.html

Array Assignment Dependency Checking (Part 4)

There's currently an interesting difference between the front-end dependencies detected on 32-bit and 64-bit platforms. Consider the following code (taken from dependency_10.f90 below).

```
integer, dimension (4) :: a
integer :: n

n = 3
where (a(:n) .ne. 0)
  a(:n) = 1
endwhere
```

One i686-pc-linux-gnu, we correctly detect that "a(:n)" is the same as "a(:n)", however on x86_64-unknown-linux-gnu, we don't, and instead allocate a temporary array!

The problem is that array indices are different widths/types on different targets, so on 64-bit machines the gfortran front-end inserts a call to __gfc_convert_i4_i8 in the upper bound of the array section. This unfortunately is enough to trip up the routine gfc_dep_compare_expr, which assumes the worst, and we therefore treat these refs as overlaping.

The patch below extends gfc_dep_compare_expr, to be able to test the equality of intrinsic operators/functions depending upon whether their operands/arguments are considered equal. Hence we assume __gfc_convert_i4_i8(x) is the equal to __gfc_convert_i4_i8(y) if we can show that x is equal to y. Likewise, because it's easy/common we also handle unary and binary operators, so that we can tell "n + 1" == "n + 1", when "n" == "n" and "1" == "1".

Hopefully, this should benefit real-world code, as the idiom "array(:n)" is commonly used for dealing with variable numbers of elements in fixed sized arrays. For example, polyhedron's protein.f90 uses "array(:natoms)" in its' WHERE statements.

http://gcc.gnu.org/ml/fortran/2006-03/msg00151.html

Array Assignment Dependency Checking (Part 5)

The following patch improves the gfortran front-end's dependency analysis of array expressions containing pointer variables. The first change is to shuffle around the tests of "pointerness" such that we only check whether expr1 is a pointer when comparing EXPR_VARIABLEs. At the moment, we'll mistakenly consider that "p = 0" contains a dependency, even though the RHS (expr2) is constant, just because the LHS (expr1) is "pointer" variable.

The second change is that we now allow pointer variables to be considered equal, so that "p" and "p" (variable expressions of the same symbol) are treated as referencing the same object, even when "p" is a pointer type.

Finally, the third change is that a pointer variable "p" and another variable "v" can not reference the same object, if the base type of "p" is not the same as the base type of "v". To demonstrate this consider the example code below extracted from the polyhedron benchmark capacita.f90:

```
integer, private, pointer, dimension(:,:) :: Grid
real, allocatable, dimension(:,:) :: Y

where (Grid == 0)
  Y = 0
end where
```

The WHERE statement looks quite trivial; we test one variable Grid as a mask, to conditionally set another variable Y. Unfortunately, mainline currently allocates a temporary LOGICAL*1 mask as it believes that writes to Y may affect the values of Grid!? The problem is that Grid is defined as a pointer which currently pessimizes the dependency checking in gfc_check_dependency. Of course, looking closer there couldn't possibly be a dependency using type-based aliasing, as Grid points to an array of integers and Y points to an array of reals. The Fortran standards guarantee that these two references can't alias, as is supported by the comments in symbol.c:gfc_symbols_could_alias.

http://gcc.gnu.org/ml/fortran/2006-03/msg00238.html

Array Assignment Dependency Checking (Part 6)

Yet another patch to improve the gfortran front-end's dependency analysis. The workhorse of gfc_check_dependency is the function gfc_dep_compare_expr which compares two expressions returning 0 for equality, 1 or -1 if they are ordered and -2 otherwise. Currently, ordering is only supported for integer constants, so this routine can determine "2 < 3". The patch below extends this functionality with some minimal symbolic comparison functionality. This can be used to determine that "N < N+1" and that "N+1 < N+2". The middle-end assumes that pointer arithmetic doesn't overflow, which I believe is also reasonable in fortran array index expressions (perhaps a language lawyer can comment). Alternatively, (worst case) we could add a new return value -3, indicating that the two expressions can't be equal, but not specifying their relative ordering.

Two minor difficulties in the development of this patch. Once again I was bitten by __convert_i4_i8 on 64bit platforms (which isn't an issue on 32bit targets), so I added special support for integral extensions which are "unary, constant, increasing functions", such that extend(A) op extend(B) holds if A op B does, where op is either ==, < and >. The other gotcha was that we now have to be even more careful about forall indices (as revealed the fiendish forall_5.f90).

I think some of this analysis is much better performed by the middle-end using trees, however (i) the middle-end currently doesn't have much symbolic range analysis, (ii) even simplistic front-end analysis can have a significant performance impact [for example, yesterday's 20-25% improvement in polyhedron's channel.f90]. Ultimately, its a front-end design decision whether to use trees or even canonicalize "X + -C" vs "X - C" or simplify "(X + C1) + C2". This would simplify testing that "X - 1 < X + 1" which we still can't do. Fortunately, I believe the simplistic analysis below is sufficient to catch the remaining low hanging fruit in polyhedron.

For example, in nf.f90 we've the following array assignment (in both NF2DPrecon and NF3DPrecon):

```
x(i:i+nx-1) = x(i:i+nx-1) - c*x(i-nx:i-1)
```

As long as we can show that i-1 < i, we can prove there's no dependency. Excercise left to reader. Patch in preparation.

http://gcc.gnu.org/ml/fortran/2006-03/msg00482.html

Array Assignment Dependency Checking (Part 7)

Element vs. Section

A small mistake in the current mainline implementation of dependency.c's gfc_check_element_vs_section means that gfortran currently assumes that any array section/range may potentially alias/overlap any array index. Currently, this code only attempts to handle integer range bounds and indices, but a logic error means that it can't even disambiguate simple cases such as "a(2,...)" vs "a(4:8,...)".

The patch below pretty much rewrites gfc_check_element_vs_section (and in the process obsoletes gfc_is_inside_range) extending the functionality and at the same time fixing the above (safe though pessimistic) bug.

The new code attempts to avoid the previous constraint that the range bounds and element must be integer constants. This allows us to disambiguate "2" from "4:N", "10" from "N:8", and "2" from "N:4:-1", and even "3" from "5:9:N". Using the pending symbolic expression comparison patch, this change should also handle things such as "N-1" vs "N:N+6".

http://gcc.gnu.org/ml/fortran/2006-03/msg00502.html

Array Assignment Dependency Checking (Part 8)

Section vs. Section

Simplistic section vs. section dependency checking can only handle simple cases of overlap where the start, end and strides of both sections are all compile-time integer constants. This can be improved with symbolic range bounds testing.

Examples #3

```
        a(1:5:2) = a(8:6:-1)

        a(1:8) = a(2:9)

        a(4:7) = a(4:1:-1)
```

Examples with compile-time constant array sections.

Examples #4

```
        a(i:i+2) = a(i+4:i+6)

        a(j:1:-1) = a(j:5)

        a(k:k+2) = a(k+1:k+3)
```

Examples of resolvable dependences with symbolic section bounds.

Improvements to section vs. section handling allow use to resolve dependencies such as the following example from Polyhedron's capacita.f90, where, for example, the NAG fortran compiler requires a temporary.

```
        t(Ng1:2*Ng1-1,:) = t(Ng1:1:-1,:)  ! Remaining half, using symmetry
```

TODO: There are simple examples of array sections that are independent, even though the bounds overlap. An example is the assignment "a(1:5:2) = a(2:6:2)".

http://gcc.gnu.org/ml/fortran/2006-04/msg00069.html

Array Assignment Dependency Checking (Part 9)

Array Constructors

Originally, gfortran's dependency.c doesn't know that constant array constructors, such /(1, 2, 3 /), aren't affected by assignments to variables. This then pessimizes code such as "x(:) = cshift((/ 1, 2, 3, 4, 5/), 2)" or "reshape(x,(/2,2/))" where our paranoia causes us to use temporary destination arrays to avoid the imagined dependencies.

Example

```
  integer :: a(4)

  where (a(:) .ne. 0)
    a(:) = (/ 1, 2, 3, 4 /)
  endwhere
end
```

The simple/obvious fix is to actually implement the missing dependency analysis for EXPR_ARRAY in gfc_check_dependency, by looping over the values of the array constructor.

http://gcc.gnu.org/ml/fortran/2007-01/msg00709.html

FORALL Statement improvements (Part 1)

At first I thought the issue was just a minor memory leak that the info structure that we allocate with gfc_getmem at the start of that function was not being freed by the end of that routine. On closer analysis, it looks like the strange decision to implement the nested_forall_info as a doubly-linked list with the outermost FORALL at the head, complicates the task of popping the current FORALL off of the list, which was then never implemented, making it unsafe to free that memory. The consequence is that if ever we have multiple FORALLs inside a FORALL, we contine to accumulate loops/scopes, leading to incorrect code/behaviour on the following testcase.

```
 integer :: a(10,10)
 integer :: tot
 a(:,:) = 0
 forall (i = 1:10)
   forall (j = 1:10)
     a(i,j) = 1
   end forall
   forall (k = 1:10)
     a(i,k) = a(i,k) + 1
   end forall
 end forall
 tot = sum(a(:,:))
 print *, tot
 if (tot .ne. 200) call abort ()
end
```

Instead of the assignment to one and increment loops each being executed 100 (i.e. 10*10) times, we expand the second "increment" FORALL as a triply nested loop of i,j,k which then gets executed a 1000 times. This then leads to the result that instead of tot being the expected 200, mainline gfortran actually produces the result 1100!

Investigating in subversion, it turns out that like the previous FORALL mask bug, this also dates back to the addition of trans-stmt.c to mainline, and is therefore in the strictest sense not a regression.

The fix below both simplifies the nested_forall_info data structure, speeds up its processing/manipulation and resolves the correctness bug. Instead of having the current quadratic behaviour where we represent the head of nested_forall_info as the outermost loop, and therefore have to append new scopes to the tail, and then traverse forward to the end of the list and then backwards as we generate the actual loops, things can be much simplified by reversing the datastructure and considering the head to be the innermost FORALL. This then only requires a singly linked list, O(1) cons as we enter a scope and no need to unlink the head as we pop out of a scope. Much like other "scopes" are handled in GCC.

http://gcc.gnu.org/ml/fortran/2007-01/msg00618.html

FORALL Statement improvements (Part 2)

The following patch resolves PR fortran/30404, a wrong code bug in all versions of gfortran triggered by nested FORALL loops containing conditional execution masks.

This is a little tricky, where to begin...

In order to preserve the specified "parallel execution" semantics of fortran9x's FORALL statements, the compiler has to ensure that the effects of the loop body don't effect/modify any specified conditional execution mask. In the most general case, and without doing any dependency analysis, this requires that we construct a temporary array to hold the execution mask, and populate it in a loop before performing the actual loop itself.

i.e.

```
FORALL(i=1:10,cond(i))
  foo();
```

is expanded as

```
bool temp[10];
for (i=0; i<10; i++)
  temp[i] = cond(i+1);
for (i=0; i<10; i++)
  if (temp[i])
    foo();
```

Although this is inefficient, if it can be shown that foo cannot affect future versions of cond, for the purposes of this patch/explanation we'll concentrate on this general/fallback case.

The problem in the PR involves how handle conditional execution masks in nested FORALL statements. Consider the slightly more complicated:

```
FORALL(i=1:3,cond1(i))
  FORALL(j=1:5,cond2(i,j))
    foo();
```

The problem now becomes what is the intended semantics, and how can it be implemented. The key element involves the conditional execution mask for invoking "foo". Conceptually, we potentially require a mask to cover the iteration space of foo(). Given that foo may affect the evaluation of cond1 and cond2, we need to all the required evaluations of cond1 and cond2 are performed before foo. In the worst case, this obviously requires (up to) 15 elements in the mask.

The cause of this bug is that the possibility of such worst case dependencies weren't fully considered or fully implemented when nested FORALL lowering was first first written. As a result, the execution mask currently used by gfortran doesn't fully take the nesting into account and only allocates an array of size 5, which is then populated without taking into account the influence of "i" on cond2.

The fix is to significantly restructure the way we allocate and populate conditional execution masks in forall statements. Firstly, the population loop (e.g. for cond2) needs to be executed over the full forall nest (iterating over i and j). Previously, gfc_trans_nested_forall_loop had a special "nest_flag" used solely for this purpose, to generate just the innermost loop of the forall nest, for the purpose of populating the conditional mask. Alas, as explained above, this is incorrect, so one set of changes in the patch below is to remove this flag, and update all callers.

The immediate fallout of traversing the full iteration space to populate the execution mask, is that we now need to know in advance how large that space is. For easy cases, like the one above it's easy enough to use the constant array bounds and simply multiply 3*5 to get a worst case bound. However, when the bounds are constants, or even loop invariant, the task becomes more complicated. In the general case, we now have to expand the full loop nest, and count the number of times the body is executed.

[Aside: In theory, we could dynamically reallocate the mask extending it as we go, which is an equally valid but slightly more complex implementation. For the time being, we'll use the counting method].

Conveniently, trans-stmt.c already has a function for expanding all the loops in a FORALL nest to determine the size of the iteration space called "compute_overall_iter_number", which is used to determine the size required by a WHERE mask in a FORALL nest, amongst other things. So we can reuse this function, via allocate_temp_for_forall_nest to determine the size of the mask we need.

At this point, there are a number of improvements that can be made. The first is that in compute_overall_iter_number, if we've an unnested FORALL loop with a constant bound, we can use that directly. Allowing use to use a local array to hold the mask, rather than calling malloc. Secondly, if we have to go to the effort of traversing the iteration space, we might as well optimize the presence of outer conditions, to prune the size of the mask (or temporary arrays we need to hold).

In the example above, if cond1 is (/true,false,true/), don't need to allocate all 3*5=15 elements, but only 2*5=10. i.e. only allocate the space we need, for sparse execution masks and/or deep forall nests this can result in a significant saving. The secret however is that our mask indices are now global to the forall loop nest, and incremented each time we reach a test. This required a minor restructuring of gfc_trans_forall_loop which currently resets "maskindex" to zero within the forall loop nest.

With these major changes, the trans-stmt.c reorganization below resolves the PR and gets the correct result. In fact, the improvements to compute_overall_iter_number mean that we now frequently allocate less memory for in conditional FORALL statements. They also cause harmless failures of gfortran.dg/dependency_8.f90 and gfortran.dg/dependency_13.f90. These two testcase check that the dependency analyzer spots a hazard by scanning the .original tree-dump for a "malloc". Now that we do a better job of determining constant mask sizes, we now don't call malloc but allocate the masks on the stack as local variables. The patch below tweaks these testcases to search for the use of a temporary, "temp", indicating that a dependency was found by the analyzer.

For those that have managed to follow all this, the code we now generate for the second example above is:

```
  bool temp1[3];
  for (i=0; i<3; i++)
    temp1[i] = cond1(i+1);
```

```
count = 0;
for (i=0; i<3; i++)
  if (temp1[i])
    count += 5;
temp2 = malloc(count);

mi = 0;
for (i=0; i<3; i++)
  if (temp1[i])
    for (j=0; j<5; j++)
      temp2[mi++] = cond2(i+1,j+1);

mi = 0;
for (i=0; i<3; i++)
  if (temp1[i])
    for (j=0; j<5; j++)
      if (temp2[mi++])
        foo();

free(temp2);
```

Naturally, there are a number of follow-up patches I intend to submit to start improving this, mainly to use the dependency analyzer to avoid the use of a temporary mask if possible, perform simple loop fusion and failing that improvements to compute_overall_iter_number to improve the way we determine the size of the iteration space. However, I'll keep those optimizations/enhancements independent of this correctness fix.

http://gcc.gnu.org/ml/fortran/2007-01/msg00298.html

FORALL Statement improvements (Part 3)

The following patch is a further improvement to the way the gfortran front-end translates nested FORALL statements. One aspect of this translation is the calculation of the number of times the body of a loop nest will be executed. This is done by trans-stmt.c:compute_overall_iter_number which contains a TODO comment with the words "optimize the computing process". By default, the current implementation is to expand a complete copy of the loop nest, incrementing a counter in the body to determine the size at run-time.

The patch below tackles the common/simple case of unconditional loop nests with constant bounds. In this case, the body is executed bound1*bound2*... boundN times.

```
integer a(100,100)
outer: forall (i=1:100)
  inner: forall (j=1:100,i.ne.j)
    a(i,j) = a(j,i)
  end forall inner
end forall outer
end
```

In this matrix transpose calculation we need to determine the number of times "i .ne. j" is executed, to allocate a suitable execution mask for it. With the patch below, we can now establish at compile-time (in the front-end) that the answer is 10,000 times, and allocate a suitable stack temporary, rather than expand loops and call malloc.

http://gcc.gnu.org/ml/fortran/2007-01/msg00388.html

FORALL Statement improvements (Part 4)

The following patch is another improvement to the way gfortran translates nested FORALL statements into gimple. Now that the data-structure used to represent forall loop nests has been simplified, it's now possible for trans-stmt.c to use gfc_trans_nested_forall_loop to expand just the "N" outermost loops instead of the whole forall nest. This allows another generalization of the code in compute_overall_iter_number which determines the number of times a loop body will be executed.

As explained in one of my previous patches to this function
http://gcc.gnu.org/ml/fortran/2007-01/msg00388.html
It's possible to determine the iteration number at compile-time when the loops are unconditional with constant bounds. We can use a similar strategy to simplify the calculation when the innermost loops are unconditional with constant bounds, and eliminate/peel some of the inner loops from the loop stack.

So in an example code such as:

```
  integer :: it(3,2)
  forall (i = 1:2, i < 3)
    forall (j = 1:2)
      it(i+1,j) = it(i,j)
    end forall
  end forall
end
```

we no longer need to loop over both i and j to determine how many times the assignment is executed. We only need to loop over i, evaluating the conditional, and then multiply the result by two.

FORALL Statement improvements (Part 5)

The following patch fixes PR fortran/30400 which is a rejects-valid and ICE-on-valid problem using ANY, ALL or any logical intrinsic function as the mask in a FORALL statement.

The problem is in the parser (match.c's match_forall_iterator) where the parser when analysing a statement such as the one below speculatively considers the identifiers after each comma to be the name of an iteration variable to be followed by "=", then <start>:<end>[:<stride>].

```
forall (i = 1:5, j = 1:5, k = 1:5, any (a (i, j, k, :) .gt. 6))
```

Hence it's not until the "(" after the "any" that syntactically we can tell that "any" isn't another iteration variable such as "i", "j" or "k". However it turns out that the speculative call to primary.c's match_variable can occassionally modify the symbol it finds, for example updating it's "flavor" from FL_UNKNOWN to FL_VARIABLE given the context. To undo this side-effect when we roll-back the parse, the code following the "cleanup:" label in match_forall_iterator *always* resets the identifier's symbol flavour back to FL_UNKNOWN.

Unfortunately, in the case of this PR this "undo" mechanism is far too aggressive. For our "ANY" symbol, the match_variable fails because we already know that the flavor is FL_PROCEDURE. Reseting this to FL_UNKNOWN even when it wasn't modified, causes the front-end to loose track of the intrinsic's properties including logical return type, etc...

The fix below is to guard the parser roll-back/undo such that we only revert the symbol flavour in a failed parse, if the flavor is currently FL_VARIABLE. This avoids corrupting the built-in intrinsics that are FL_PROCEDURE, and better limits the modification to those symbols that were actually/likely modified. I don't think this is perfect, there's a possibility that a symbol was FL_VARIABLE before we tried treating/interpreting it as an forall iterator variable, in which case we'll anonymize it back to FL_UNKNOWN. However, this patch is sufficient to resolve the current class of failures.

http://gcc.gnu.org/ml/fortran/2007-02/msg00357.html

FORALL Statement improvements (Part 6)

The following simple patch tweaks the way that we expand conditional FORALL statements in gfortran, to improve the case where the conditional expression (mask) is a compile-time constant, such as .true. or .false. Although unlikely to occur commonly in real code, these are obvious optimizations that are easily implemented before more complex dependency-based transformations.

I'm wondering what the gfortran maintainers think about implementing these sort of source-to-source transformations earlier, perhaps during parsing or a simplification pass? It'd would simplify translation if "where (.true.) a(:) = b(:)" were previously lowered to "a(:) = b(:)" or if "elsewhere (.false.)" were eliminated, etc... I'm not sure where best it would be appropriate to canonicalize "forall (..., .true.)" into "forall (...)", but prior to this patch we'd generate significantly different code for each.

http://gcc.gnu.org/ml/fortran/2007-02/msg00120.html

WHERE Statement improvements (Part 1)

The following patch improves the code generated by gfortran for the common case of simple (Fortran-90) WHERE statements and constructs.

Consider the following test case:

```
program where_1
  integer :: a(5)
  integer :: b(5)

  a = (/1, 2, 3, 4, 5/)
  b = (/1, 0, 1, 0, 1/)

  where (b .ne. 0)
    a = a + 10
  endwhere
  print *,a
end program
```

Currently, the code generated for the WHERE portion of this program looks like the following C code:

```
int *mask = malloc(5*sizeof(int));
for (i=0; i<5; i++)
  mask[i] = (b[i] != 0);
for (i=0; i<5; i++)
  if (mask[i])
    a[i] += 10;
free(mask);
```

Likewise the slightly more complicated example:

```
program where_2
  integer :: a(5)
  integer :: b(5)

  a = (/1, 2, 3, 4, 5/)
  b = (/1, 0, 1, 0, 1/)

  where (b .ne. 0)
    a = a + 10
  elsewhere
    a = 0
  endwhere
  print *,a
end program
```

generates the even more curious:

```
  int *mask1 = malloc(5*sizeof(int));
  int *mask2 = malloc(5*sizeof(int));
  for (i=0; i<5; i++)
  {
    int flag = (b[i] != 0);
    mask1[i] = flag;
    mask2[i] = !flag;
  }
  for (i=0; i<5; i++)
    if (mask1[i])
      a[i] += 10;
  for (i=0; i<5; i++)
    if (mask2[i])
      a[i] = 0;
  free(mask1);
  free(mask2);
```

The patch below is the main part of a series of patches to improve things. This installment identifies the common cases of a where containing a single assignment without internal dependencies, possibly with a single unconditional elsewhere also containing a single assignment without internal dependencies.

With this patch we now generate, the following code for the first case:

```
  for (i=0; i<5; i++)
    if (b[i] != 0)
      a[i] += 10;
```

and the following code for the second case:

```
  for (i=0; i<5; i++)
    if (b[i] != 0)
      a[i] += 10;
    else
      a[i] = 0;
```

Unfortunately, this transformation revealed some minor limitations in gfortran's dependency infrastructure, so the patch below address four separate but related issues. The first and most trivial is that the prototype for gfc_check_dependency is duplicated in both dependency.h and trans-array.h. When the implementation was broken out of trans-array.c into dependency.c the prototype was duplicated rather than moved. This patch removes the trans-array.h definition, and tweaks trans-stmt.c to include "dependency.h" to get this declaration (updating the build dependenices in Make-lang.in appropriately).

One subtle issue concerns the semantics of gfc_check_dependency. This routine was originally intended to analyse WHERE and FORALL statements to determine whether array references contained a dependency, i.e. in something like "x = y" whether the write to x would affect the "future" values of y. In such cases, we need a temporary array to preserve the original y during the assignment.

The complication involves this "TODO" comment from the current code:
/* Identical ranges return 0, overlapping ranges return 1. */

This is the semantics required by this patch, that identical ranges have no dependency. i.e. in the assignment "a = a" we don't need a temporary, or in "WHERE (a /= 0) a = LOG(a)", we don't need a temporary. Unfortunately, implementing this TODO revealed that gfc_check_dependency has other callers that required identical array references/ranges to have a dependency. The callers where for the non-copying intrinsics TRANSPOSE, CSHIFT and EOSHIFT.

Consider the difference between "A = SQRT(A)" and "A = TRANSPOSE(A)", clearly the first doesn't require a temporary as each element can be processed order-independently, whilst in the second we do require a temporary, as the source "A" and destination "A" have been silently permuted. To allow both behaviours, I introduce an "identical" argument to gfc_check_dependency to indicate where identical references should be considered dependent or not.

The final aspect of this patch is the new gfc_trans_where_3 to handle simple-enough WHERE statements. Currently the dependency checks are extremely strict, and the restriction to single assignments, no nested WHEREs, no WHEREs in FORALL, no more than one ELSEWHERE and only ELSEWHEREs without conditions may seem draconian but probably catches a significant fraction of WHEREs in practice.

I've follow up patches to only use LOGICAL*1 for masks in the WHERE statements that still need them, and optimizations to avoid the need for complementary true/false masks, and to reduce the number of mask allocations in nested WHEREs/ELSEWHEREs.

http://gcc.gnu.org/ml/fortran/2006-02/msg00063.html
http://gcc.gnu.org/ml/gcc-patches/2006-02/msg00316.html

WHERE Statement improvements (Part 2)

The following patch is a simple change to reduce the run-time memory footprint of gfortran's WHERE statements and constructs by using a array of LOGICAL*1 for the execution mask, instead of the current LOGICAL*4. I must acknowledge Jakub Jelinek's previous efforts to apply a similar optimization to FORALL masks: http://gcc.gnu.org/ml/fortran/2005-06/msg00253.html

http://gcc.gnu.org/ml/fortran/2006-02/msg00084.html

WHERE Statement Improvements (Part 3)

It turns out that code generation of WHERE statements was worse than I thought. On Saturday, I claimed that the following source code:

```
where (b .ne. 0)
  a = a + 10
endwhere
```

generated the following internal representation:

```
int *mask = malloc(5*sizeof(int));
for (i=0; i<5; i++)
  mask[i] = (b[i] != 0);
for (i=0; i<5; i++)
  if (mask[i])
    a[i] += 10;
free(mask);
```

Using only a single mask array. On closer inspection, at the time it actually generated something like:

```
int *mask1 = malloc(5*sizeof(int));
int *mask2 = malloc(5*sizeof(int));
for (i=0; i<5; i++)
{
  int flag = (b[i] != 0);
  mask1[i] = flag;
  mask2[i] = !flag;
}
for (i=0; i<5; i++)
  if (mask1[i])
    a[i] += 10;
free(mask1);
free(mask2);
```

i.e. we always allocate and populate two complementary masks, even when the second one (mask2 above) is completely unused. Unfortunately, GCC's tree-level and RTL optimizers are currently unable to optimize away this second allocation, meaning that simple WHERE loops allocate twice as much memory as they need.

The patch below fixes this by avoiding the allocation, population and freeing of the complementary "pending mask" when it is not used, i.e. at the end of a WHERE chain (i.e. when the last conditional WHERE/ELSEWHERE expression isn't followed by an ELSEWHERE). This is signaled by passing a NULL pointer as the NMASK argument to gfc_evaluate_where_mask.

In a related clean-up, the patch below also removes the "pmask" argument to gfc_trans_where_2. All callers of this function currently pass NULL for this parameter, so removing it makes understanding the rest of this change easier, as "pmask" is local to gfc_trans_where_2.

http://gcc.gnu.org/ml/fortran/2006-02/msg00099.html

WHERE Statement Improvements (Part 4)

The following patch is the next installment in the series of patches to improve gfortran's code generation for F95's WHERE construct. One issue with the current implementation is the large number of execution mask arrays that are concurrently allocated.

Consider the following F95 source code:

```
program where_1
  integer :: a(4)
  integer :: b(4)

  a = (/1, 2, 3, 4/)
  b = (/0, 0, 0, 0/)
  where (a .eq. 1)
    b = 1
  elsewhere (a .eq. 2)
    b = 2
  elsewhere (a .eq. 3)
    b = 3
  elsewhere
    b = 4
  endwhere
  print *,b
end program
```

Currently, trans-stmt.c allocates true masks and false masks (termed execution masks and pending execution masks in the relevant standards) for each conditional in a where construct.

Hence the above code is generated as:

```
temp1 = (bool*)malloc(4);
temp2 = (bool*)malloc(4);
for (int i1=0; i1<4; i1++)
{
  temp1[i1] = (a[i1] == 1);
  temp2[i1] = !temp1[i1];
}
for (int i2=0; i2<4; i2++)
  if (temp1[i2])
    b[i2] = 1;
temp3 = (bool*)malloc(4);
temp4 = (bool*)malloc(4);
for (int i3=0; i3<4; i3++)
{
  temp3[i3] = (a[i3] == 2);
  temp4[i3] = !temp3[i3];
}
```

```
  for (int i4=0; i4<4; i4++)
    if (temp2[i4] && temp3[i4])
      b[i4] = 2;
  temp5 = (bool*)malloc(4);
  temp6 = (bool*)malloc(4);
  for (int i5=0; i5<4; i5++)
  {
    temp5[i5] = (a[i5] == 3);
    temp6[i6] = !temp5[i5];
  }
  for (int i6=0; i6<4; i6++)
    if (temp2[i6] && temp4[i6] && temp5[i6])
      b[i6] = 3;
  for (int i7=0; i7<4; i7++)
    if (temp2[i7] && temp4[i7] && temp6[i7])
      b[i7] = 4;
  free(temp1);
  free(temp2);
  free(temp3);
  free(temp4);
  free(temp5);
  free(temp6);
```

Notice both the large number of mask allocations (two per conditional) and the inefficiency that these are repeatedly ANDed together in later where assignments (i.e. the temp2, temp4 and temp6 references in the COND_EXPR for the final elsewhere's b = 4 masked assignment).

The patch below attempts to clean all of this up, such that we only dynamically allocate a single pair of execution masks for each WHERE construct. Nested WHEREs allocate their own masks, but ELSEWHEREs require no allocation. The trick is to perform the ANDing as execution proceeds, and always maintain a single execution mask and pending execution mask for the current position in the WHERE construct.

For the example code above, we now generate:

```
  temp1 = (bool*)malloc(4);
  temp2 = (bool*)malloc(4);
  for (int i1=0; i1<4; i1++)
  {
    temp1[i1] = (a[i1] == 1);
    temp2[i1] = !temp1[i1];
  }
  for (int i2=0; i2<4; i2++)
    if (temp1[i2])
      b[i2] = 1;
  for (int i3=0; i3<4; i3++)
  {
    bool cond = (a[i3] == 2);
  temp1[i3] = temp2[i1] && cond;
  temp2[i3] = temp2[il] && !cond;
 }
 for (int i4=0; i4<4; i4++)
  if (temp1[i4])
   b[i4] = 2;
 for (int i5=0; i5<4; i5++)
 {
  bool cond = (a[i5] == 3);
  temp1[i5] = temp2[i5] && cond;
  temp2[i5] = temp2[i5] && !cond;
 }
 for (int i6=0; i6<4; i6++)
  if (temp1[i6])
   b[i6] = 3;
 for (int i7=0; i7<4; i7++)
  if (temp2[i7])
   b[i2] = 4;
 free(temp1);
 free(temp2);
```

Peak memory usage is also reduced as nested masks are now deallocated when the execution of their WHERE construct is completed. Previously all of the masks in all of the branches a WHERE construct were live until the top-level WHERE finished.

This change also simplifies the code internally. Temporary mask arrays are now allocated and deallocated in gfc_trans_where_2. This means there's no longer a need to maintain a list of arrays to deallocate in a temporary_list structure, or to loop over it after calling gfc_trans_where_2.

There are still a number of follow-up clean-ups and optimizations that can be made once this patch is in, but this change is already a significant hunk to review. The two obvious improvements are that we can now remove the TRUTH_AND_EXPR from gfc_trans_where_assign (as the mask list now always contains only a single element), and avoiding the updating of "cmask" for empty where/elsewhere clauses. i.e.

```
    WHERE ((1/a) .ne. 0)
    ENDWHERE
```

need not allocate any memory, but must perform the potentially trapping divisions.

http://gcc.gnu.org/ml/fortran/2006-02/msg00376.html

WHERE Statement Improvements (Part 5)

This patch optimizes some corner cases in the code generation of WHERE statements. Rule R739 of section 7.5.3.1 of the F95 standard allows all "where-body-constructs" to be empty. These empty bodies result in us currently populating/allocating temporary masks when they're not needed.

The simplest change is in the call to gfc_evaluate_where_mask, where if the current WHERE/conditional ELSEWHERE clause has any empty body, there's no need to update "cmask". Subsequent clauses only require the pending execution mask.

Then related to this change is that we can now tweak the conditions under which we allocate the temporary "cmask" and "pmask". It doesn't matter how many clauses there are in WHERE construct, if all of the bodies are empty we won't need either of pmask or cmask. However, these corner cases are rare, and there's little benefit of looking at more than the first two or three clauses. This is sufficient for us to completely optimize F90's WHERE construct (which didn't allow conditional ELSEWHERE clauses).

The logic is that if the construct contains three or more clauses (i.e. must be an F95 construct), just allocate both temporary masks as always, (tested via cblock->block && cblock->block->block). Otherwise if we only need a "cmask", if there are executable statements in either of the first two clauses, and we only need a "pmask" if there are executabe statements in the second clause (if one exists).

Finally, there's one further refinement to the above logic that we can omit the "cmask" in where_15.f90 [which has an empty initial WHERE followed by a non-empty unconditional ELSEWHERE], where the "unconditional ELSEWHERE simply uses pmask" optimization obviates the need for cmask.

http://gcc.gnu.org/ml/fortran/2006-02/msg00435.html

WHERE Statement Improvements (Part 6)

The following patch is the final installment of my set of patches to improve code generation of gfortran's WHERE construct. The gist of this final optimization is to change

```
WHERE (cond)
  stmt1
ELSEWHERE
  stmt2
ENDWHERE
```

from being implemented as

```
cmask = malloc(...);
pmask = malloc(...);
for (i=0; i<...; i++)
{
  bool tmp = cond(i);
  cmask[i] = tmp;
  pmask[i] = !tmp;
}
for (i=0; i<...; i++)
  if (cmask[i])
    stmt1(i);
for (i=0; i<...; i++)
  if (pmask[i])
    stmt2(i);
free(pmask);
free(cmask);
```

which uses two temporary masks, into the equivalent:

```
cmask = malloc(...);
for (i=0; i<...; i++)
  cmask[i] = cond(i);
for (i=0; i<...; i++)
  if (cmask[i])
    stmt1(i);
for (i=0; i<...; i++)
  if (!cmask[i])
    stmt2(i);
free(cmask);
```

i.e. instead of using two temporary mask arrays, where one mask holds the exact complement of the other, we teach trans-where to "invert" the sense of a conditional execution mask. It turns out this optimization is only useful for the first WHERE/ELSEWHERE of an unnested WHERE construct. This is detected in the code below as "mask" will be NULL_TREE.

This is mostly a mechanical change, such that whereever we used to pass a "mask" expression tree, we now also pass a bool flag called invert, which specifies if the "mask" should be used as is or inverted. During this clean-up, I spotted a few more places where we still traversed a linked list of MASKs and ANDed them together. This dead-code is cleaned up below.

Finally, the logic for deciding whether to allocate neither, either or both of cmask and pmask in gfc_trans_where_2 was getting confusing (to me), so I decided to split it out with the use of need_cmask and need_pmask, and then tackled the different cases we can optimize sequentially. My apologies for restructuring my own code, but the case-by-case coding style should be easier to maintain, with so many interacting optimizations. Hopefully, the comments I've added fully describe what's going on, but with luck no-one will need to touch this code again for some time.

As a minor benefit of need_cmask and need_pmask, we can now avoid calculating the size of the array to allocate (potentially at run-time), when we don't have to allocate either.

http://gcc.gnu.org/ml/fortran/2006-02/msg00469.html

Integer SIGN Intrinsic Improvement

I was recently looking over the fortran Sweep3D benchmark and noticed that in the inner most loops it was making heavy use of the SIGN intrinsic. Looking at the gimple we generate for the integer variants of the gfortran's SIGN intrinsic, we can not only do slightly better but at the same time fix a latent bug.

The result of SIGN(X,Y) is the absolute value of X but with the sign of Y. For floating point types this conveniently maps to __builtin_copysign, but for integral types we currently lower this to the tree:

```
return (a >= 0) ^ (b >= 0) ? -a : a
```

which on x86_64 expands to something like:

```
movl   %edi, %eax
notl   %esi
movl   %edi, %edx
notl   %eax
shrl   $31, %esi
negl   %edx
shrl   $31, %eax
xorl   %esi, %eax
cmovne  %edx, %edi
movl   %edi, %eax
ret
```

It turns out that its possible to do this more conveniently without branching or conditional moves using the alternate sequence:

```
int tmp = (a ^ b) >> 31;
return (a + t) ^ t;
```

The keen eyed will recognize this as very similar to the branchless "abs" that we generate in the middle-end, but tweaked such that we perform the negation if the sign bits differ. Interested readers can also find this implementation suggested in section 2-9 "Transfer of Sign" in Henry Warren's book "Hacker's Delight".

With this alternate implementation, we now generate the shorter and faster:

```
xorl   %edi, %esi
sarl   $31, %esi
leal   (%rsi,%rdi), %eax
xorl   %esi, %eax
ret
```

I tried quantifying how much better this was by comparing the times to call the above two implementations several million times vs. a control of calling a function that just returned zero. Unfortunately, this approach is severely flawed with the short sequence overlapping considerably with the call/loop overhead; the resulting times were 20.949s for a null implementation, 20.954 for the new implementation and 24.586 for the old implementation resulting in a apparent speed-up of several hundred fold!?

In theory, many of the above transformations should be implemented in fold-const.c and the various if-conversion passes (and I'll submit those patches as follow-ups), but whilst investigating this I noticed that there's currently a bug in the current SIGN intrinsic expansion, in that it doesn't protect against multiple evaluation of it's arguments.

So the FORTRAN code:

```
  function foo()
    integer :: foo
    foo = sign(ia(),ib())
  end function
end
```

on mainline expands to the tree

```
foo ()
{
  int4 __result_foo;

  __result_foo = ia () >= 0 ^ ib () >= 0 ? -ia () : ia ();
  return __result_foo;
}
```

where we clearly call the function ia(), the X argument, multiple times. When this function has side-effects, we can end up generating incorrect results. In the middle-end, we'd normally use a SAVE_EXPR to handle this, but the gfortran scalarizer conveniently provides a gfc_evaluate_now function for this purpose. If I could write FORTRAN, I might even have attempted a new testcase for the testsuite.

http://gcc.gnu.org/ml/fortran/2007-01/msg00458.html

FORMAT Extension

A common extension supported by many fortran compilers, including AIX f77 and IRIX f77, is to allow the X format to be specified without an integer prefix. Unfortunately, this extension is often found in the source code of many scientific packages (such as the semi-empirical quantum mechanics program MOPAC) even though the fortran-77 standard indicates it should formally be written 1X. The patch below adds support for this feature/extension to g77.

It turns out this behaviour is already supported by g77's run-time library, as the run-time parsing of fortran FORMAT statements in libI77/fmt.c's ne_d routine already treats 'X' as '1X'. All that's needed is a tweak to the front-end to avoid issuing an error, except when the -fpedantic flag has been specified.

Example #1

```
        PRINT 10, 2, 3
   10   FORMAT (I1, X, I1)
        END
```

http://gcc.gnu.org/ml/gcc-patches/2003-03/msg01465.html

Operator .xor. Extension

The following patch adds support for ".xor." as an intrinsic operator. This is supported by several vendor compilers including IBM's XL fortran, DEC fortran, SUN fortran and even, I've discovered, GNU f77! See http://www.llnl.gov/icc/lc/asci/fpe/fpe.operators.html

My original goal was to allow .xor. to support integral types in addition to logicals, however, once I discovered that f77 supported logical .xor., the lesser first installment to treat .xor. as a synonym .neqv. seemed a reasonable split, and potentially a regression fix.

http://gcc.gnu.org/ml/fortran/2007-09/msg00070.html

Logical/Integer Interconversion Extension

The following patch adds the ability to implicitly convert between integer and logical types to the gfortran front-end as a GNU extension. This functionality is commonly supported by several compilers including IRIX's f77 and g77 when using the -fugly-logint command line option. Without this functionality, gfortran is unable to build some dusty-deck codes, such as the semi-empirical quantum mechanics program MOPAC, which compiles without warnings using SGI's MIPSPro f77 compiler.

One possibility was to add support for g77's -fugly-logint to gfortran, however a much cleaner solution is to treat it as an extension using gfortran's standard handling mechanism. This allows us to automatically generate errors when compiling against the f95 and f2003 standards, and issue warnings when the user specifies "-pedantic".

http://gcc.gnu.org/ml/gcc-patches/2005-02/msg00213.html