An approach for data propagation from Tree SSA to RTL

Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin
ISP RAS
{dm, ssg, amonakov, zhur}@ispras.ru

Abstract

This paper describes an approach for propagation of information collected during Tree SSA optimization passes to RTL level to enable RTL-passes to take advantage of the information available on Tree SSA, which is generally more precise than that available on RTL. At the moment the propagated information includes data dependence and may alias information. In the paper we propose an approach for such propagation and discuss problems that arise during the implementation.

1. Introduction

There are two major intermediate languages (*ILs*) in GCC to perform optimization passes on the program code: *Tree SSA* and *RTL*. *RTL (register transfer language)* representation is a low level representation, which can be either independent of the target machine or hardware specific. Initially there was only RTL implemented in GCC, and all optimization passes were written to work only with this IL. Tree SSA form was introduced in GCC version 4.0. This representation provides convenient infrastructure to implement middle-end optimizations, and since it was introduced many new Tree SSA optimizations have been written, and some RTL optimizations were replaced with their Tree SSA versions. However, there are still optimizations that need to be done with RTL. Many of these RTL optimization passes need the information that is available on Tree SSA but is hard to compute at RTL level, since RTL is low-level representation and it lacks significant information about the source code. Unfortunately, the information computed on Tree SSA becomes unavailable after trees have been translated into RTL during the expand pass. In this paper we discuss the approaches for propagating optimization information collected at Tree SSA level to make it available for RTL-level passes.

1.1 Problem formulation

Throughout this article, we will refer to expression at Tree SSA level as *original tree*, the information we need to propagate as *attributes*, and the same expression expanded to RTL as *expanded rtx*, or simply *rtx*.

Hence the general problem of propagating information can be formulated as follows. Given a set of original trees T, set of attributes A, set of expanded rtxes R, mappings $\varphi : T \to A$ and $e : T \to R$, construct a mapping

$$\varphi' : R \to A$$

such that

$$\forall t \in T : \varphi(t) \neq \emptyset \Rightarrow \varphi'(e(t)) = \varphi(t).$$

Though from a mathematical point of view the task may seem to be straightforward, it has some difficulties being implemented in the existing GCC framework. These include how do we organize $e(t)$ mapping and preserve it's correctness throughout all RTL optimization passes, where do we store the mapping tables and associated attributes, how we deal with flow sensitive information and others. We will address all of these issues in Section 4.

The rest of the paper is organized as follows. In Sections 2 and 3 we discuss the particular cases of attribute propagation – *may aliases* for improving instruction scheduler and *data dependencies* for improving modulo scheduling. We generalize the approach and discuss propagation of Tree SSA attributes to RTL in the general case in Section 4. In Section 5 we discuss current results and future work, and give the conclusion of the work and summarize the results in Section 6.

2. Propagation of alias information

Precise *may alias* information is very important for many back end optimizations. When information is missing for the memory references, the RTL optimizers will conservatively assume that `x[i]` overlaps with `y[j]`, and won't allow to move `y[j]` across the `x[i]` location. This puts limitations on the instruction scheduler's ability to rearrange instructions, preventing it from intermixing different array accesses.

There is aliasing information that is available on RTL level, but it's type-based and flow-insensitive (Tree SSA may alias analysis is flow-sensitive). The problem of imprecise alias information is even worse for architectures that have no displacement in address which is important for GCC RTL alias analysis. This means that heuristics that work for architectures like x86 for disambiguating memory references of the form `[bp+4]` and `[bp+32]` won't work on machines without base+offset addressing mechanism (e.g. IA-64). The approach for overcoming this has been proposed in [Gupta2002].

2.1 Approach for may alias propagation

May aliases for the memory reference are defined as a set of variables that are accessible through this reference. If the exact set of aliased variables can not be determined, we consider that the reference *aliases anything* and denote its may alias set with $\top$.
Tree SSA `may_aliases` pass provides flow-sensitive *may alias* information, and it's assigned to each SSA generation of the dereferenced pointer. Since one variable on RTL generally corresponds to many SSA generations, original trees (as well as their may alias attributes, which also are SSA trees) should be converted out of the SSA form first:

GREPS '07 September 16, 2007, Brasov, Romania.

```
unsigned long a[N], b[N];
unsigned long *p, *q;
int i;

p = a; q = b;

for (i = 0; i < N; i++) {
  *p++ = *q++;
}
```

(a) Sample code: memory copy loop

```
...

.mmi
st8 [r16] = r11, 8
ld8 r10 = [r15]
nop
;;
.mmi
st8 [r16] = r10
ld8 r9 = [r23]
nop
;;
.mmi
st8 [r24] = r9
ld8 r8 = [r25]
nop
;;
.mmi
st8 [r26] = r8
ld8 r3 = [r18]
nop

...
```

(b) Unrolled loop body on IA-64 without *may aliases* propagation

```
...

.mmi
st8 [r15] = r3, 8
ld8 r19 = [r9]
nop 0
.mmi
ld8 r14 = [r2]
ld8 r18 = [r23]
nop 0
;;
.mmi
ld8 r16 = [r27]
st8 [r15] = r14
nop 0
.mmi
st8 [r25] = r19
st8 [r26] = r18
nop 0

...
```

(c) Unrolled loop body on IA-64 with *may aliases* propagation

Figure 1. Code optimization with exported may alias information on IA-64

$$c(t_i^{ssa}) = t,\ c : T^{ssa} \rightarrow T$$

$$\varphi(t) = \bigcup_i \{c(a^{ssa}) : a^{ssa} \in \varphi^{ssa}(t_i^{ssa})\},\ \forall i : c(t_i^{ssa}) = t$$

The second part of the expression gives the mapping from SSA to non-SSA trees, the first – mapping of corresponding attribute sets. Here $c(t^{ssa})$ is a coalescing function, which maps SSA variables with non-overlapping live ranges into the one memory partition. Besides the different SSA-generations of the same variable, separately defined variables may get coalesced, e.g. if they are defined in different blocks on the same level.

With adjusting the coalescing function $c(t^{ssa})$ we may get either flow-sensitive or flow-insensitive export (or combination of both, coalescing only data the least critical for disambiguations). If we put

$$c(t_i^{ssa}) = t_i,\ \ t_i \neq t_j \Leftrightarrow t_i^{ssa} \neq t_j^{ssa}$$

we'll get flow sensitive may alias export. Even flow-insensitive may alias propagation should deliver better results than original type-based, because the type-based can never disambiguate pointers of the same type.

To propagate attributes (may aliases) for the given tree t to RTL, we need to assign them to appropriate rtx r, such that $r = e(t)$. The *tree* to rtx mapping can be done during the expand pass, which implements $e(t)$. The problem is that between the expand pass and the pass where alias information will be actually used (in our case, instruction scheduler, which is one of the final RTL passes) there are many other optimization passes. Each of them may alter or remove existing rtxes, or introduce new, so if there were any mapped attributes available after the expand pass, they already might be either lost or become incorrect for those rtxes available in later passes. Therefore, if we'd like allow to modify rtxes, we should provide an interface for adjusting attributes and mapping in accordance with changes made. With the propagation of *may aliases* this problem was addressed as follows. Every rtx that represents a memory reference in GCC has attached standard memory attributes, accessed through the MEM_ATTRS macro. These include type-based alias set number (MEM_ALIAS_SET), part of original expression (MEM_EXPR[1]), and other attributes. These attributes are known to be propagated correctly from expand till the final RTL pass, so we added new MEM_ORIG_EXPR field to standard memory attributes and propagated it in the way similar to how original MEM_EXPRs were propagated, copying it along with the rtx it's assigned to, and clearing original expression when its rtx is changed.

For our implementation of may aliases export we define $\varphi(t)$ (see Section 1.1) as a triplet

$$\langle\, t, Pts_{deref}(t), Subvars(t) \,\rangle$$

where t is an original tree itself, $Pts_{deref}(t)$ is a points-to set for t, if t is a *dereferenced ptr* or $\top$ otherwise; $Subvars(t)$ is a set of subvariables[2] for t, if t contain subvariables or is assigned one. We define V_{ops} set as a set of variables that are accessed with t reference:

$$V_{ops}(t) = \begin{cases} Subvars(t), & Subvars(t) \neq \top \\ Pts_{deref}(x), & base\_addr(t) = *x \\ & \wedge\, x\ is\ DECL \\ \{x\} \cup Subvars(x), & base\_addr(t) = x \\ & \wedge\, x\ is\ DECL \\ \top, & otherwise \end{cases}$$

[1] MEM_EXPR() contains stripped down original memory expression, e.g. `x` for the reference `x[i]` and `<null>.c` for COMPONENT_REF `a[j].c`. This expression is used by the `alias.c` heuristics to disambiguate references like `a.b` and `a.c`.

[2] Some structure and array variables may have subvariables, each represented with *structure field tags (SFTs)*. E.g. and `x` is an instance of `struct s` that has two fields, then `x` will have two subvariables, and `x.a` will be assigned *SFT.1* tag. Also if some pointer points to `x`, then its points-to set will also contain all `x` subvars' tags. Please refer to [DNovillo2006] for the details.

where

$$base\_addr(x) = \begin{cases} x, & x\ is\ DECL \\ & \vee\, x = *p \\ base\_addr(t), & x = t[i] \\ & \vee\, x = t.field \\ \top, & otherwise \end{cases}$$

E.g. for the dereference of pointer $V_{ops}(*p) = Pts_{deref}(p)$, for the variable declaration $V_{ops}(x) = \{x\} \cup Subvars(x)$.

To reduce memory consumption, $V_{ops}(t)$ is computed dynamically upon requests to `alias.c *_dependence()` functions. So instead of saving $V_{ops}(t)$ for every tree in the program, points-to sets and subvars are saved only for those trees that actually have them.

To determine whether two memory references t_1 and t_2 alias each other, first $V_{ops}(t_1)$ and $V_{ops}(t_2)$ are computed. If $V_{ops}(t_1) \cap V_{ops}(t_2) = \emptyset$, then memory reference t_1 is *not aliased* with t_2. Otherwise t_1 may alias t_2.

2.2 Example

Figure 1 shows how the lack of good aliasing information prevents the instruction scheduler from performing effectively on the sample memory copy loop. Figure 1(a) shows original loop C code, Figure 1(b) shows the fragment of the unrolled and scheduled loop body, and the same fragment scheduled using propagated *may alias* information is shown in Figure 1(c). The speedup for this sample loop of the code in (c) comparing to that in (b) evaluates to 4x, assuming that there is still 4-cycle latency on memory loads between instruction bundles.

2.3 May alias export with data speculation

In addition to conservative propagation of *may alias* sets, the propagation of less precise *"most likely alias"* sets sometimes is also desirable. While information about some dereferenced pointers is propagated with *may aliases* as has been described earlier, some points-to sets may not be propagated with flow-insensitive method because few of the SSA generations may not have useful points-to sets ($\varphi(t) = \top$), while other actually may have finite points-to sets. When coalescing variables, unification of attributes by all SSA generations will result in *aliases anything* ($\top$) set. Sometimes pointers lose their finite points-to on pointer arithmetics. E.g. for the loop sequentially accessing `a[i]` the following code is generated:

```
  D.2970_17 = malloc (D.2969_16);
  a_2 = (int *) D.2970_17;
  ...
L1:
  D.2972_24 = (long unsigned int) i_28;
  D.2973_25 = D.2972_24 * 4;
  D.2974_26 = (int *) D.2973_25;
  D.2975_27 = D.2974_26 + a_2;
  ...
  goto L1;
```

The information for `D.2975_27` indicating that it may point only inside array `a` is lost at the assignment to `D.2975_27`. If we replace the unification rule $\alpha \cup \top = \top$ for may alias sets used in propagation of may aliases with $\alpha \cup \top = \alpha$, we may then retain useful may alias set { `a` } in this example, but with this rule the correctness of *may alias* set can't be guaranteed. We call the may alias set computed using unification rule above as *most likely alias*. Then, V'_{ops} are computed in the same way earlier V_{ops} were computed, but this time based on *most likely alias* set instead of *may alias*. We say that memory references t_1 and t_2 most likely do not alias, if $V'_{ops}(t_1) \cap V'_{ops}(t_2) = \emptyset$. Data dependence between t_1 and t_2 is considered *weak*, if t_1 may alias t_2, and t_1 most likely doesn't alias t_2.

The example of optimization that benefits from such information is data speculation [Belevantsev06]. With lack of good aliasing information data speculation may generate too many speculative instructions, which will result in performance regression. We can speculate more effectively, if we'll give a priority to those speculations, which dependences with instruction we speculate against are *weak*.

2.4 Related work

In [Gupta2002] a flow-sensitive method for disambiguating memory references by tracing the pointer arithmetic operations is proposed. Each RTL pseudo register r is assigned an address descriptor at each program point p. The address descriptor consists of a pair $A(r) = (I, Z)$, where Z denotes the set of mod-k residues of the value computed at program point p relative to the value computed by instruction I. Usually, Z is represented as a bitmap integer value. E.g. if $k = 32$ and given the instructions

I_1: `r1 = r0 + c`
I_2: `r2 = r1 + 40`

then $A(r2) = (I_1, 100\text{h})$, which means that `r2` has an offset m from the value defined at I_1, $m\ mod\ k = 8$, and $Z = 100\text{h}$ has it's 8-th bit set. Then, pointer arithmetic operations are defined for address descriptors sets and iterative data flow analysis is performed, computing the address descriptors for each pseudo register at each program point. After address descriptors has been computed, two memory references `[`r_1`]` and `[`r_2`]` can be disambiguated, if corresponding address descriptors $A(r_1) = (I_1, Z_1)$ and $A(r_2) = (I_2, Z_2)$ are such that $I_1 = I_2$ and $Z_1 \cap Z_2 = \emptyset$. However, this approach has a few drawbacks, such as high overhead on iterative computing of address descriptors and loss of analysis precision when the referenced registers were computed using different base registers, e.g. when the referenced memory was allocated dynamically on the heap. Also it may experience problems with disambiguating references to structures allocated on stack, if their size is a multiple of k – the size of mod-k residues sets used in the algorithm for tracing pointer arithmetics.

3. Data dependency export

In this section we discuss the export of data dependencies from Tree SSA to RTL and how it's used to improve modulo scheduling pass in GCC. This pass implements the *Swing Modulo Scheduling (SMS)* algorithm for software pipelining [Hagog2004, Llosa2001]. To perform effectively, it needs precise data dependency information, including memory reference dependences and distances between them in the iteration space.

Data dependence analysis determines whether two statements modify the same data and thus should be executed in a certain order. Dependencies restrict the possibilities for reordering or parallelizing statements. Dependence analysis within loop bodies usually deals with array references. Let f and g be two access functions to array a, and x and y be two iteration vectors. A dependence between two statements accessing a via f and g exists if and only if $f(x) = g(y)$. The dependence analysis aims to prove the independence, i.e. two statements never access the same element, or, when the former is impossible, to characterize the relationship between the accesses, namely distance and direction vectors. This information could be used by a number of loop transformations, including parallelization, vectorization, and software pipelining.

3.1 Data dependence analysis in GCC

There are two analyses of array data dependencies in GCC, which are on the Tree SSA and the RTL level, respectively. The Tree SSA data dependence analysis is located in `tree-data-ref.[ch]` files, also using parts of `tree-chrec.c`. For a given loop, the analysis builds a vector of data references (represented as `struct data_reference`) and a vector of dependence relations (represented as `struct data_dependence_relation`). A data reference contains links to a memory reference and a container statement, a first accessed location, a base object, and other memory attributes. A dependence relation contains the data references it links, its type, distance vector, direction vector, and subscripts information. The Tree SSA dependence analysis is used by the vectorizer and by the loop linear transformations (`tree-vect*.[ch]` and `tree-loop-linear.c`, respectively). The RTL array data dependence analyzer is located in `ddg.[ch]` files and was written specifically for swing modulo scheduling (SMS) implementation in GCC. The analyzer builds a data dependence graph (DDG) for a given basic block. The DDG is represented as a vector of nodes. Each DDG node contains vectors of incoming and outgoing dependence edges, sets of successors and predecessors of the node in the DDG, and the containing instruction. Each DDG edge, analogously to the Tree SSA analysis, contains source and destination nodes of the edge, a dependence type, an edge latency, and a distance. Additionally, the edges that are going to/from the same node form a linked list analogously to control flow edges. The analyzer uses scheduler dependence analysis (located in `sched-deps.c`) to build intra-loop dependencies and the data flow engine (located in `df-*.c`) to build inter-loop dependencies.

The RTL analyzer has the following deficiencies:

- A DDG is built only for a single basic block loops. This is because the current SMS implementation only supports such loops. The problem not only puts additional constraint on the SMS but also prevents using the dependence information in other passes.
- A distance of inter-loop dependencies is not calculated and is set to one conservatively. This limits the SMS implementation to interleaving instructions from neighbor iterations.
- Intra-loop dependencies are calculated using RTL alias analysis, which is not always able to disambiguate array references (especially on the architectures that lack address displacement, like IA-64).

All these problems may be avoided if the data dependency information is propagated from the Tree SSA to the RTL level.

3.2 The approach for data dependency propagation

To bring precise data dependences to Modulo Scheduling pass, we save `struct data_reference` data for each memory reference as it is computed on Tree SSA in function `compute_data_dependences_for_loop`. The new special Tree SSA optimization pass has been written, which traverses all loops within the current function and saves the data dependency attributes for each memory reference found in a hashtable indexed by the memory reference's tree address. The data dependency attributes saving pass is executed before `iv_opts` pass, because loop dependency analyzer can not handle correctly `TARGET_MEM_REFS` which are introduced by the `iv_opts` pass.

Similarly as with *may alias* export, we should take care of preserving consistency of the propagated information and its mapping. To preserve correct mapping, we need to take care of all places where exported memory references are modified, deleted or new memory references are created within the function loops. All these operations should be tracked in Tree SSA optimization passes located from after `iv_opts` to `out_of_ssa` passes inclusive. Fortunately, there're very few passes in the specified interval, and there is only one place where the mapping correctness may be broken: it is when the memory reference being replaced with TARGET_MEM_REF in `iv_opts` pass, and its TMR_ORIGINAL attribute receives the unshared and rewritten out of the SSA form value, thus the tree address, which is the key for attribute lookup table, is being changed. We handle it, replacing the affected key tree with appropriate new tree, rewritten out of the SSA. On the RTL level we use `MEM_ORIG_EXPR` of the rtx to obtain a key for attributes hashtable lookup, so the task of preserving the mapping correctness on RTL is equivalent to that in may alias export (see Section 2).

If the rtx was derived from the original SSA tree only with such transformations that the resulting rtx still represents the same memory reference the dependence data was computed for, then that rtx has it's data dependence attributes attached and we consider that rtx to attributes mapping is correct. The consistency of the dependency data itself puts another problem. Data dependency attributes are relevant only within the loop for which they were computed. Some optimization passes (e.g. VRP, basic block reordering) may modify control flow and transform the loops. In some cases nested loops may be converted even into two sequential loops with common shared block, that is a latch for the first loop and header for the second. If data dependency attributes were computed for the original loop, they will become irrelevant after such transformation.

The problem may be approached by giving optimization passes an interface to adjust the attributes as the memory references or the control flow are modified. In most cases, we're unable to adjust the attributes so they remain correct after they have been exported. The task of maintaining the correctness of propagated attributes might be as hard as computing them from scratch in place. E.g. trying to maintain flow-sensitive *may alias* attributes correctness through the RTL optimization that moves the instruction between basic blocks will result in rewriting the alias analysis on the intermediate representation that is RTL annotated with non-SSA trees. Although sometimes writing ad hoc pass that computes the attributes right in the place they are needed might be an option, in general the attributes propagation from the Tree SSA should be considered as a cheap way for improving the quality of optimization information on the RTL: we will bring everything we can to the RTL, and invalidate anything we can't.

So, if the attributes lose their consistency in the intermediate optimization passes that are performed after the attributes are exported and before they were used, we need to delete them, so they won't lead to generation of the incorrect code. For example, while propagating memory dependences, their consistency may be broken by the loop unrolling pass. Consider the loop that has two memory references, `x[i]` and `x[i+4]`. Within the original loop, the distance between these iterations is 4. After the loop body has been unrolled, say, with unrolling factor 4, the distance will be equal to 1. This way, while performing loop unrolling, the exported distance attribute should be adjusted by the loop unrolling factor, or simple invalidated.

3.3 Data verifier

To aid the correct propagation of the data dependency information a special verifier pass has been written. This pass runs after each of the intermediate optimization passes (either Tree-SSA or RTL) and performs the following tasks:

- for each loop within the current function all memory references within the loop are traversed, and if the current intermediate representation is RTL, then key k is assigned the value of `MEM_ORIG_EXPR(mem)`, otherwise k is a memory reference tree itself;

```
uint8 a[N], b[N];

for (i=0; i < N; i++) {
  a[i] = b[i] = 0;
}
```

(a) Sample code: memset loop

```
...

.L48:
    .mmi
    add r15 = r14, r17
    st4 [r16] = r0
    add r16 = r14, r18
    .mmi
    nop 0
    adds r14 = 4, r14
    nop 0
    ;;
    .mib
    st4 [r15] = r0
    nop 0
    br.cloop.sptk.few .L48
    ;;

...
```

(b) Loop body on IA-64 scheduled with SMS, no data dependence export

```
...

.L48:
    .mmi
    st4 [r16] = r0
    st4 [r15] = r0
    add r16 = r14, r17
    .mmb
    add r15 = r14, r18
    adds r14 = 4, r14
    br.cloop.sptk.few .L48
    ;;

...
```

(c) Loop body on IA-64 scheduled with SMS, with data dependences export

Figure 2. Code optimization with exported data dependences on IA-64

- for the key k obtained on previous step, the existence of an attributes $\varphi(k)$ is checked in the hashtable, and if the key is found, k is marked as *seen* on this verifier pass, otherwise the warning is given;
- for each k' previously marked as *seen* on this pass, we build a pair (k, k') and check whether the corresponding `struct data_dependence_relation` can be found for this pair, and if not, the warning is given. This may happen if the loops got transformed.

Although the verifier can't guarantee the consistency of the propagated attributes (e.g. if loop unrolling was applied), it's main job is to identify those passes that break either the tree to attribute mapping or consistency of the data dependence information. After these passes has been identified, if that is possible, they get fixed so that the attributes are propagated correctly. If the fix for correct propagation is not possible, then the correspondent attributes are invalidated.

3.4 Example

The work on data dependences propagation is still in progress, but currently we're able to export Tree SSA dependences for simple loops (Figure 2). For the example loop, Tree SSA dependence analysis is able to figure that the `a[i]` and `b[i]` references are independent, and spurious dependence created by imprecise RTL dependence analyzer is eliminated with the export. This leads to 2x speedup of the example loop.

4. A general approach for propagation

In this section we generalize the approach for data propagation from Tree-SSA to RTL level based on our experience with two projects for data propagation – *may aliases* and *data dependences* export.

In general case, a propagation of any attributes from Tree SSA to RTL consists of the following tasks:

1. **Attributes mapping.** This includes how the propagated attributes will be assigned to rtxes. There are a few ways for doing this: the first is to introduce additional fields into rtx data structures themselves (like `MEM_ATTRS`), the second is to create on the side data structures (e.g. hashtables, indexed by the rtx). We've chosen the first approach and added a new `MEM_ORIG_EXPR` field to `MEM_ATTRS`, as described in Section 2. The main reason for choosing this option is its transparency: the mechanisms for preserving `MEM_ATTRS` during code transformation already exist, only minor changes are required to enable propagation of the newly introduced fields. In both our projects we just needed to propagate the attributes only for the small quantity of program's rtxes, i.e. dereferenced pointers for may alias export and memory references for data dependence export. For these tasks putting a link to attributes into the `MEM_ATTRS` structure is feasible (increase in memory usage on `cc1-i-files` is negligible), but for propagation of some other attributes if it will require assigning them to some rtxes other than *MEM*s one may consider creating an external structure to implement mapping of the attributes.
2. **Flow sensitivity.** Since on Tree SSA the attributes exported are flow sensitive, it should be decided whether it's important to preserve their flow sensitivity on the RTL. E.g. for may alias export we've counted the number of times the different versions of dereferenced pointers were coalesced, at it turned out that such situations are very rare – only a few cases for the bootstrap, so we think that for the may alias export flow insensitive information is sufficient. If the flow sensitive export was chosen, the `add_coalesce` function should be fixed in such way that different SSA generations of the pointer will receive different memory locations. This may result in code blowout and should be done only if flow sensitive information is really necessary for target optimization.
3. **Data storage.** First, it should be decided what data is propagated as the attributes. There is the usual memory/speed tradeoff between exporting all attributes that can be computed at Tree SSA or exporting only the minimal set of attributes so the rest can be computed at the RTL level upon request. For may alias export we've chosen the latter, exporting only points-to sets for dereferenced pointers, and computing points-to for requested trees upon calls to `true_dependence`. The overhead measured on compiling `cc1-i-files` was less than 1%. However, we might have computed V_{ops} for each memory reference at the Tree SSA level, sacrificing the memory usage, but significantly simplifying the computational part of may alias propagation. In contrary, data dependences export doesn't use any additional computation at the RTL, just exporting all results of loop dependence analysis at Tree SSA.

   Second, the representation of the exported attributes should be chosen. With our implementation, the attributes representation is not significantly different from that on Tree SSA level with

	Total	New	New, %
Bootstrap	61004	2844	4.66
SPECint2000	100363	5185	5.17

Table 1. The number of disambiguations on bootstrap and SPECint2000 compilation.

major exception that every tree is rewritten out of its SSA form (see Sections 2-3). To reference these attributes it requires a mapping from RTL expressions to tree expressions. However, it is undesirable to have this mapping for all instructions, as this will effectively keep in memory both representations. At the moment, such mapping is maintained only for memory references and can be used for propagating any information regarding memory, such as aliasing or data dependencies. For other expressions, the better way could be to rewrite the data to refer to RTL expressions instead of tree expressions.

4. **Correctness of the rtx to attributes mapping.** This determines whether trees will preserve link to their attributes through tree optimization passes until the expand pass, and similarly whether rtxes will retain their mapping to attributes from expand until the point where exported attributes actually will be used. If we are exporting attributes for memory references, correctness of this mapping will ensure that attributes will be the same as for the original tree after transformations of the memory reference, such as conversion to `TARGET_MEM_REF` and adjusting addresses with frame pointer.
5. **Consistency of the exported data.** While the mapping may be correct, the exported attributes themselves may become irrelevant after the optimizations performed. This type of consistency determines the correctness of attributes with respect to external code transformations. The one example of such transformations that affects the consistency of attributes for data dependency export is loop unrolling (see Section 2).
6. **Data verification.** To ensure that the propagated attributes are alive and valid, we propose writing a verifier to catch the cases when the exported data is missing or invalid. When the problem cases are identified, there are two ways to fix them. First, all offending optimization passes should be fixed to preserve the data. This way is error prone as it potentially requires modifications in many unrelated places, but it is the only way at the moment. Second, an API for modifying the IL should be developed, and all passes should use just this API. In this case, only interface functions should be fixed to update the exported data. This way is preferable, and there are examples of projects that followed it, namely preserving loop structures (only control-flow API functions were modified to update the loop information) and the dataflow project (only functions manipulating with RTL were modified to update the dataflow stuff).

5. Current progress and future work

The alias export and data dependency export are implemented in GCC. The first is available in `alias-export` branch, the second is being developed as a part of contract with Intel Corp. The data dependency export patch has been sent as RFC to the GCC mailing list [DDGExport]. However, the work on both projects is still in progress and the results are preliminary.

Table 1 shows the number of disambiguations for the alias export branch on bootstrap and SPECint2000 compilation. Disambiguations are counted with the calls to `*_dependence` functions in `alias.c`, which are being called from the instruction scheduler and `gcse` pass. The alias export gives thousands of disambiguations at RTL level using information from Tree SSA, but only about 5% of them are new in respect to what can be obtained with old RTL alias analysis. Currently, performance gain on SPEC2000 is neutral, so further investigation will be needed to understand why the new disambiguations do not result in a performance increase or what should be done to disambiguate those memory references that are significant for code performance. To do this, we're extracting the code from performance tests that may benefit from alias export, and with the extracted sample we're fixing the issues that prevent the data from being propagated correctly.

Data dependency export performance has not yet been measured with SPEC, but it has shown positive results on sample loops (like the one shown in Section 3.4). Currently, we are working with the data verifier to ensure the correct data propagation on bootstrap, regression testing and SPEC compilation.

Since in the future we may be interested in exporting more information from Tree-SSA to RTL besides alias and data dependency, we plan to merge both projects and develop a general API for data storage and correct propagation from Tree SSA to RTL, taking into consideration all the issues discussed in Section 4.

6. Conclusion

We have presented an approach for data propagation from Tree SSA to RTL, identified the major problems of data propagation and proposed the solutions for each of them. These problems are: establishing a mapping from rtx to tree, maintaining flow sensitivity, storage of the propagated data and its correct propagation between the passes, and verification of the data correctness. The method described in the paper has been implemented in GCC for propagating alias and data dependency information. We also have presented examples of successful propagation of alias information and data dependences, and shown how this propagation can improve the performance of instruction scheduling and swing modulo scheduling passes. In the future, a general API for data propagation is to be developed that will allow us to propagate any new data with minimal code changes, using the common propagation framework.

References

[DNovillo2006] Diego Novillo. Presentation slides for CGO 2007, San Jose, California. `http://www.airs.com/dnovillo/Papers/#cgo2007`

[Belevantsev06] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, and Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In *Proceedings of GCC Developers' Summit 2006*, Ottawa, Canada, June 2006.

[GCCInternals] `http://gcc.gnu.org/onlinedocs/gccint`

[Gupta2002] Sanjiv K. Gupta Naveen Sharma. Alias Analysis for Intermediate Code. Proceedings of the GCC Developers Summit, 2003, pages 71–78.

[Hagog2004] Mostafa Hagog. Swing Modulo Scheduling for GCC. Proceedings of the GCC Developeres Summit, June 2004, Pages 55–64

[Llosa2001] Josep Llosa, Eduard Ayguade, Antonio Gonzalez, Mateo Valero, Jason Eckhardt. Lifetime-sensitive Modulo Scheduling in a Production Environment. IEEE Transactions on Computers, Vol 50, No. 3, 2001, pages 234–249.

[DDGExport] http://gcc.gnu.org/ml/gcc/2007-08/msg00342.html