Design and Implementation of Tree SSA

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

Tree SSA is a new optimization framework for GCC that allows the implementation of machine and language independent transformations. This paper describes the major components of Tree SSA, how they interact with the rest of the compiler and, more importantly, how to use the framework to implement new optimization passes.

1 Introduction

The main goal of the Tree SSA project is to evolve GCC's optimization infrastructure to allow more powerful analyses and transformations that had traditionally proven difficult or impossible to implement in RTL. Though originally started as a one person hobby project, other developers in the community expressed interest in it and a development branch off the main GCC repository was started. Soon thereafter, Red Hat began sponsoring the project and, over time, other organizations and developers in the community also started contributing. Presently, about 30 developers are actively involved in it[1], and work is underway to implement vectorization and loop optimizations based on the Tree SSA framework. We expect Tree SSA to be included in the next major release of GCC.

Although Tree SSA represents a significant change in the internal structure of GCC, its main design principle has been one of evolution, not revolution. As much as possible, we tried to make Tree SSA a "drop-in" module. In particular, we decided to keep the `tree` and `rtl` data structures so that neither front ends nor back ends needed to be re-implemented from scratch. This was an important engineering decision that (a) allowed us to reach to a working system in a relatively short period of time, but (b) it exposed a few weak spots in the existing data structures that we will need to address in the future (Section 8).

This paper describes Tree SSA from a programmer's point of view. Emphasis is placed on how the different modules work together and what is necessary to implement a Tree SSA pass in GCC. Section 2 provides an overview of the new files and compiler switches added to GCC. Section 3 describes the GENERIC and GIMPLE intermediate representations. Section 4 describes the control flow graph (CFG), block and statement manipulation functions. Section 5 describes how optimization passes are scheduled and declared to the pass manager. Section 6 describes the basic data flow infrastructure: statement operands and the SSA form as implemented on GIMPLE. Section 7 describes alias analysis. Conclusions and future work are in Section 8.

[1] This is a rough estimate based only on ChangeLog entries.

2 Overview

2.1 Command line switches

Most of the new command line options added by Tree SSA are only useful to GCC developers. They fall into two broad categories: debugging dumps and individual pass manipulation.

All the debugging dumps are requested with `-fdump-tree-`*pass*`-`*modifier*. By default, the tree representation is emitted in a syntax resembling C. Passes can be individually selected, but the most common usage is to enable all of them using `-fdump-tree-all`. When enabled, each pass dumps all the functions in the input program to a separate file. Dump files are numbered in the same order in which passes are applied. Therefore, to see the effects of a single pass, one can just run *diff* between the N and $N + 1$ dumps.

Modifiers affect the format of the dump files and/or the information included in them[2]. Currently, the following modifiers can be used:

- `raw`: Do not pretty-print expressions. Use the traditional tree dumps instead.
- `details`: Request detailed debugging output from each pass.
- `stats`: Request statistics from each pass.
- `blocks`: Show basic block boundaries.
- `vops`: Show virtual operands (see Section 6 for details).
- `lineno`: Show line numbers from the input program.
- `uid`: Show the unique ID (i.e., `DECL_UID`) for every variable.

We currently support enabling and disabling most individual SSA passes. Although, it is not clear whether that will be always supported, it is sometimes useful to disable passes when debugging GCC. Note, however, that even if the bug goes away when disabling an individual pass, it does not mean that the pass itself is faulty. The bug may exist somewhere else and is exposed at this point.

All the new command line switches are described in detail in the GCC documentation.

2.2 New files

All the necessary API and data structure definitions are in *tree-flow.h* and *tree-passes.h*. The remaining files can be loosely categorized as basic infrastructure, transformation passes, analysis passes and various utilities.

2.2.1 Basic infrastructure

tree-optimize.c is the main driver for the tree optimization passes. In particular, it contains `init_tree_optimization_passes`, which controls the scheduling of all the tree passes, and `tree_rest_of_compilation` which performs all the gimplification, optimization and expansion into RTL of a single function.

tree-ssa.c, *tree-into-ssa.c* and *tree-outof-ssa.c* implement SSA renaming, verification and various functions needed to interact with the SSA form.

tree-ssanames.c and *tree-phinodes.c* implement memory management mechanisms for re-using `SSA_NAME` and `PHI_NODE` tree nodes after they are removed.

[2] Note that not all passes are affected by these modifiers. A pass that does not support a specific modifier will silently ignore it.

tree-cfg.c contains routines to build and manipulate the CFG.

tree-dfa.c implements general purpose routines for dealing with program variables and data flow queries like immediate use information.

tree-ssa-operands.c contains routines for scanning statement operands (`get_stmt_operands`).

tree-iterator.c contains routines for inserting, removing and iterating over GIMPLE statements. Two types of iterators are provided, those that do not stop at basic block boundaries (known as *tree statement iterators*) and those that do (known as *block statement iterators*). Most optimization passes use the latter.

c-gimplify.c, *gimplify.c* and *tree-gimple.c* contain the routines used to rewrite the code into GIMPLE form. They also provide functions to verify GIMPLE expressions.

2.2.2 Transformation passes

gimple-low.c removes binding scopes and converts the clauses of conditional expressions into explicit gotos. This is done early before any other optimization pass as it greatly simplifies the job of the optimizers.

tree-ssa-pre.c, *tree-ssa-dse.c*, *tree-ssa-forwprop.c*, *tree-ssa-dce.c*, *tree-ssa-ccp.c*, *tree-sra.c* and *tree-ssa-dom.c* implement some commonly known scalar transformations: partial redundancy elimination, dead store elimination, forward propagation, dead code elimination, conditional constant propagation, scalar replacement of aggregates and dominator-based optimizations.

tree-ssa-loop.c is currently a place holder for all the optimizations being implemented in the LNO (Loop Nest Optimizer) branch [1]. Presently, it only implements loop header copying, which moves the conditional at the bottom of a loop to its header (benefiting code motion optimizations).

tree-tailcall.c marks tail calls. The RTL optimizers will make the final decision of whether to expand calls as tail calls based on ABI and other conditions.

tree-ssa-phiopt.c tries to replace PHI nodes with an assignment when the PHI node is at the end of a conditional expression.

tree-nrv.c implements the named return value optimization. For functions that return aggregates, this optimization may save a structure copy by building the return value directly where the target ABI needs it.

tree-ssa-copyrename.c tries to reduce the number of distinct SSA variables when they are related by copy operations. This increases the chances of user variables surviving the out of SSA transformation.

tree-mudflap.c implements pointer and array bound checking. This pass re-writes array and pointer dereferences with bound checks and calls to its runtime library. Mudflap is enabled with `-fmudflap`.

tree-complex.c, *tree-eh.c* and *tree-nested.c* rewrite a function in GIMPLE form to expand operations with complex numbers, exception handling and nested functions.

2.2.3 Analysis passes

tree-ssa-alias.c implements type-based and flow-sensitive points-to alias analysis.

tree-alias-type.c, *tree-alias-ander.c* and *tree-alias-common.c* implement flow-insensitive points-to alias analysis (Andersen analysis).

2.2.4 Various utilities

tree-ssa-copy.c contains support routines for performing copy and constant propagation.

domwalk.c implements a generic dominator tree walker.

tree-ssa-live.c contains support routines for computing live ranges of SSA names.

tree-pretty-print.c implements `print_generic_stmt` and `print_generic_expr` for printing GENERIC and GIMPLE tree nodes.

tree-browser.c implements an interactive tree browsing utility, useful when debugging GCC. It must be explicitly enabled with `--enable-tree-browser` when configuring the compiler.

3 Intermediate Representation

Although Tree SSA uses the `tree` data structure, the parse trees coming out of the various front ends cannot be used for optimization because they contain language dependencies, side effects and can be nested in arbitrary ways. To address these problems, we have implemented two intermediate representations: GENERIC and GIMPLE [4].

GENERIC provides a way for a language front end to represent entire functions in a language-independent way. All the language semantics must be explicitly represented, but there are no restrictions in how expressions are combined and/or nested. If necessary, a front end can use language-dependent trees in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE. In particular, a front end need not emit GENERIC at all. For instance, in the current implementation, the C and C++ parsers do not actually emit GENERIC during parsing.

GIMPLE is a subset of GENERIC used for optimization. Both its name and the basic grammar are based on the SIMPLE IR used by the McCAT compiler at McGill University [3]. Essentially, GIMPLE is a 3 address language with no high-level control flow structures.

1. Each GIMPLE statement contains no more than 3 operands (except function calls) and has no implicit side effects. Temporaries are used to hold intermediate values as necessary.

2. Lexical scopes are represented as containers.

3. Control structures are lowered to conditional gotos.

4. Variables that need to live in memory are never used in expressions. They are first loaded into a temporary and the temporary is used in the expression.

The process of lowering GENERIC into GIMPLE, known as *gimplification*, works recursively, replacing complex statements with sequences of statements in GIMPLE form. A front end which wants to use the tree optimizers needs to

1. have a whole-function tree representation,

2. provide a definition of `LANG_HOOKS_GIMPLIFY_EXPR`,

3. call `gimplify_function_tree` to lower to GIMPLE, and,
4. hand off to `tree_rest_of_compilation` to compile and output the function.

The GCC internal documentation includes a detailed description of GENERIC and GIMPLE that an implementor of new language front ends will find useful.

4 Control Flow Graph and IR manipulation

Data structures for representing basic blocks and edges are shared between GIMPLE and RTL. This allows the GIMPLE CFG to use all the functions that operate on the flow graph independently of the underlying IR (e.g., dominance information, edge placement, reachability analysis). For the cases where IR information is necessary, we either replicate functionality or have introduced hooks.

The flow graph is built once the function is put into GIMPLE form and is only removed once the tree optimizers are done[3].

Traversing the flow graph can be done using `FOR_EACH_BB`, which will traverse all the basic blocks sequentially in program order. This is the quickest way of going through all basic blocks. It is also possible to traverse the flow graph in dominator order using `walk_dominator_tree`.

Each basic block has a list of all the statements that it contains. To traverse this list, one should use a special iterator called *block statement iterator* (BSI). For instance, the code fragment in Figure 1 will display all the statements in the function being compiled (`current_function_decl`).

It is also possible to do a variety of common operations on the flow graph and statements: edge insertion, removal of statements and insertion of statements inside a block. Detailed information about the flow graph can be found in GCC's internal documentation.

5 Pass manager

Every SSA pass must be registered with the pass manager and scheduled in `init_tree_optimization_passes`. Passes are declared as instances of `struct tree_opt_pass`, which declares everything needed to run the pass, including its name, function to execute, properties required and modified and what to do after the pass is done.

In this context, properties refer to things like dominance information, the flow graph, SSA form and which subset of GIMPLE is required. In theory, the pass manager would arrange for these properties to be computed if they are not present, but not all properties are presently handled. Each pass will also declare which properties it destroys so that it is recomputed after the pass is done.

To add a new Tree SSA pass, one should

1. create a global variable of type `struct tree_opt_pass`,
2. create an `extern` declaration for the new pass in `tree-pass.h`, and,
3. sequence the new pass in `tree-optimize.c:init_tree_optimization_passes` by calling `NEXT_PASS`. If the pass needs to be applied more than once, use `DUP_PASS` to duplicate it first.

[3] It may be advantageous to keep the CFG all the way to RTL, so this may change in the future.

```
{
  basic_block bb;
  block_stmt_iterator si;

  FOR_EACH_BB (bb)
    for (si = bsi_start (bb); !bsi_end_p (si); bsi_next (&si))
      {
        tree stmt = bsi_stmt (si);
        print_generic_stmt (stderr, stmt, 0);
      }
}
```

Figure 1: Traversing all the statements in the current function.

6 SSA form

Most of the tree optimizers rely on the data flow information provided by the Static Single Assignment (SSA) form [2]. The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable.

Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment.

This section describes how the compiler recognizes and classifies statement operands recognized, the process of renaming the program into SSA form and how is aliasing information incorporated into the SSA web.

6.1 Statement operands

Tree SSA implements two types of operands: *real* and *virtual*. Real operands are those that represent a single, non-aliased, memory location which is atomically read or modified by the statement (i.e., variables of non-aggregate types whose address is not taken). Virtual operands represent either partial or aliased references (i.e., structures, unions, pointer dereferences and aliased variables).

Since the SSA form uses a versioning scheme on variable names, in principle it would not be possible to assign version numbers to virtual operands. So, the compiler associates a symbol name to the operand and provides SSA versioning for that symbol. Symbols for virtual operands are either created or derived from the original operand:

- For pointer dereferences, a new symbol called a *memory tag* (MT) is created. Memory tags represent the memory location pointed-to by the pointer. For instance, given a pointer `int *p`, the statement `*p = 3` will contain a virtual definition to `p`'s memory tag (more details in Section 7).
- For references to variables of non-aggregate types, the base symbol of the reference is used. For instance, the statement `a.b.c = 3`, is considered a virtual definition for `a`. Other terms to refer to virtual definitions include "*may-defs*," when they refer to aliased stores, and

"*non-killing defs*" when they refer to partial stores to an object of a non-aggregate type. Similarly, virtual uses are known as "*may-uses*."

Using this scheme, the compiler can now rename both real and virtual operands into SSA form. Every symbol that complies with SSA_VAR_P will be renamed. This includes VAR_DECL, PARM_DECL and RESULT_DECL nodes. To determine whether an SSA_VAR_P will be renamed as a real or virtual operand, the predicate is_gimple_reg is used. If it returns true the variable is added as a real operand, otherwise it is considered virtual.

Every statement has 4 associated arrays representing its operands: DEF_OPS and USE_OPS hold definitions and uses for real operands, while VDEF_OPS and VUSE_OPS hold potential or partial definitions and uses for virtual operands. These arrays are filled in by get_stmt_operands. The code fragment in Figure 2 shows how to print all the operands of a given statement. Operands are stored inside an auxiliary data structure known as *statement annotation* (stmt_ann_t). That's a generic annotation mechanism used throughout Tree SSA to store optimization-related information for statements, variables and SSA names.

6.2 SSA Renaming Process

We represent variable versions using SSA_NAME nodes. The renaming process in *tree-into-ssa.c* wraps every real and virtual operand with an SSA_NAME node which contains the version number and the statement that created the SSA_NAME. Only definitions and virtual definitions may create new SSA_NAME nodes.

Sometimes, flow of control makes it impossible to determine what is the most recent version of a variable. In these cases, the compiler

```
void
print_ops (tree stmt)
{
  vuse_optype vuses;
  vdef_optype vdefs;
  def_optype defs;
  use_optype uses;
  stmt_ann_t ann;
  size_t i;

  get_stmt_operands (stmt);
  ann = stmt_ann (stmt);

  defs = DEF_OPS (ann);
  for (i = 0; i < NUM_DEFS (defs); i++)
    print_generic_expr (stderr,
                        DEF_OP (defs, i), 0);

  uses = USE_OPS (ann);
  for (i = 0; i < NUM_USES (uses); i++)
    print_generic_expr (stderr,
                        USE_OP (uses, i), 0);

  vdefs = VDEF_OPS (ann);
  for (i = 0; i < NUM_VDEFS (vdefs); i++)
    print_generic_expr (stderr,
                        VDEF_OP (vdefs, i), 0);

  vuses = VUSE_OPS (ann);
  for (i = 0; i < NUM_VUSES (vuses); i++)
    print_generic_expr (stderr,
                        VUSE_OP (vuses, i), 0);
}
```

Figure 2: Accessing the operands of a statement.

inserts an artificial definition for that variable called *PHI function* or *PHI node*. This new definition merges all the incoming versions of the variable to create a new name for it. For instance,

```
if (...)
  a_1 = 5;
else if (...)
  a_2 = 2;
else
  a_3 = 13;

# a_4 = PHI <a_1, a_2, a_3>
return a_4;
```

Since it is not possible to statically determine which of the three branches will be taken at runtime, we don't know which of a_1, a_2 or a_3 to use at the return statement. So, the SSA renamer creates a new version, a_4, which is assigned the result of "merging" all three other versions. Hence, PHI nodes mean "one of these operands. I don't know which."

Previously we had described virtual definitions as non-killing definitions, this means that given a sequence of virtual definitions for the same variable, they should all be related somehow. To this end, virtual definitions are considered read-write operations. So, the following code fragment

```
...
# a = VDEF <a>
a[i] = f ();
...
# a = VDEF <a>
a[j] = g ();
...
# a = VDEF <a>
a[k] = h ();
...
```

is transformed into SSA form as

```
...
# a_2 = VDEF <a_1>
a[i] = f ();
...
# a_3 = VDEF <a_2>
a[j] = g ();
...
# a_4 = VDEF <a_3>
a[k] = h ();
...
```

Notice how every VDEF has a data dependency on the previous one. This is used mostly to prevent errors in scalar optimizations like code motion and dead code elimination. Passes that want to manipulate statements with virtual operands should obtain additional information (e.g., by building an array SSA form, or value numbering as is currently done in the dominator optimizers). The SSA form for virtual operands is actually a factored use-def (FUD) representation [5]. When taking the program out of SSA form, the compiler will not insert the copies needed to resolve the overlap. Virtual operands are simply removed from the code.

Such considerations are not necessary when dealing with real operands. SSA_NAMEs for real operands are considered distinct variables and can be moved around at will. When the program is taken out of SSA form (*tree-outofssa.c*), overlapping live ranges are handled by creating new variables and inserting the necessary copies between different versions of the same variable. For instance, given the GIMPLE program in SSA form in Figure 3a, optimizations will create overlapping live ranges for two different versions of variable b, namely b_3 and b_7 (Figure 3b). When the program is taken out of SSA form, prior to RTL expansion, the two different versions of b will be assigned different variables (Figure 3c).

```
foo (a, b, c)
{
  a_4 = b_3;
  if (c_5 < a_4)
    goto <L0>;
  else
    goto <L1>;

<L0>:
  b_7 = b_3 + a_4;
  c_8 = c_5 + a_4;

  # c_2 = PHI <c_5, c_8>;
  # b_1 = PHI <b_3, b_7>;
<L1>:
  return b_1 + c_2;
}
```

(a) Original SSA form.

```
foo (a, b, c)
{
  if (c_5 < b_3)
    goto <L0>;
  else
    goto <L1>;

<L0>:
  b_7 = b_3 + b_3;
  c_8 = c_5 + b_3;

  # c_2 = PHI <c_5, c_8>;
  # b_1 = PHI <b_3, b_7>;
<L1>:
  return b_1 + c_2;
}
```

(b) SSA form after optimization.

```
foo (a, b, c)
{
  if (c < b)
    goto <L0>;
  else
    goto <L1>;

<L0>:
  b.0 = b + b;
  c = c + b;
  b = b.0;

<L1>:
  return b + c;
}
```

(c) Resulting normal form.

Figure 3: Overlapping live ranges for different versions of the same variable.

```
foo (int *p)
{
  #   TMT.1_5 = VDEF <TMT.1_4>;
  *p_1 = 5;

  #   VUSE <TMT.1_5>;
  T.0_2 = *p_1;

  return T.0_2 + 1;
}
```

Figure 4: Representing pointer dereferences with memory tags.

7 Alias analysis

Aliasing information is incorporated into the SSA web using artificial symbols called *memory tags*. A memory tag represents a pointer dereference. Since there are no multi-level pointers in GIMPLE, it is not necessary for the compiler to handle more than one level of indirection. So, given a pointer p, every time the compiler finds a dereference of p (`*p`), it is considered a virtual reference to p's memory tag (Figure 4).

Given this mechanism, whenever the compiler determines that a pointer p may point to variables a and b (and p is dereferenced), a memory tag for p is created and variables a and b are added to p's memory tag.

The code fragment in Figure 5 illustrates this scenario. The compiler determines that p_2 may point to a or b, and so whenever p_2 is dereferenced, it adds virtual references to a and b. Also notice that since both a and b have their addresses taken, they are always considered virtual operands.

The compiler computes three kinds of aliasing information: type-based, flow-sensitive points-to and flow-insensitive points-to[4].

Going back to the code fragment in Figure 5, notice how the two different versions of pointer p have different alias sets. This is because p_1 is found to point to either c or d, while p_2 points to either a or b. In this case, the compiler is using flow-sensitive aliasing information and will create two memory tags, one for p_1 and another

[4]This one is currently not computed by default. It is enabled with `-ftree-points-to=andersen`

```
foo (int i)
{
  ...
<L0>:
  # p_1 = PHI <&d, &c>;

  #   c_5 = VDEF <c_7>;
  #   d_18 = VDEF <d_17>;
  *p_1 = 5;

  ...
<L3>:
  # p_2 = PHI <&b, &a>;

  #   a_19 = VDEF <a_4>;
  a = 3;

  #   b_20 = VDEF <b_6>;
  b = 2;

  #   VUSE <a_19>;
  #   VUSE <b_20>;
  T.0_8 = *p_2;
  ...

  #   a_21 = VDEF <a_19>;
  #   b_22 = VDEF <b_20>;
  *p_2 = T.1_9;
  ...
}
```

Figure 5: Using flow-sensitive alias information.

for p_2. Since these memory tags are associated with SSA_NAME objects, they are known as *name memory tags* (NMT).

In contrast, when the compiler cannot compute flow-sensitive information for each SSA_NAME, it falls back to flow-insensitive information which is computed using type-based or points-to analysis. In these cases, the compiler creates a single memory tag that is associated to **all** the different versions of a pointer (i.e., it is associated with the actual VAR_DECL or PARM_DECL node). Such memory tag is called *type memory tag* (TMT).

Figure 6 is similar to the previous example, but in this case the addresses of a, b, c and d escape the current function, something which the current implementation does not handle. This forces the compiler to assume that all versions of p may point to either of the four variables a, b, c and d. And so it creates a type memory tag for p and puts all four variables in its alias set.

The concept of 'escaping' is the same one used in the Java world. When a pointer or an ADDR_EXPR escapes, it means that it has been exposed outside of the current function. So, assignment to global variables, function arguments and returning a pointer are all escape sites.

We also use escape analysis to determine whether a variable is call-clobbered. If an ADDR_EXPR escapes, then the associated variable is call-clobbered. If a pointer P_i escapes, then all the variables pointed-to by P_i (and its memory tag) also escape.

In certain cases, the list of may aliases for a pointer may grow too large. This may cause an explosion in the number of virtual operands inserted in the code. Resulting in increased memory consumption and compilation time.

When the number of virtual operands needed

```
foo (int i)
{
  #   a_6 = VDEF <a_3>;
  #   b_4 = VDEF <b_5>;
  #   c_13 = VDEF <c_12>;
  #   d_15 = VDEF <d_14>;
  p_2 = baz (&a, &b, &c, &d);

  #   a_16 = VDEF <a_6>;
  #   b_17 = VDEF <b_4>;
  #   c_18 = VDEF <c_13>;
  #   d_19 = VDEF <d_15>;
  *p_2 = 5;

  #   a_20 = VDEF <a_16>;
  a = 3;

  #   b_21 = VDEF <b_17>;
  b = 2;

  #   VUSE <a_20>;
  #   VUSE <b_21>;
  #   VUSE <c_18>;
  #   VUSE <d_19>;
  T.1_7 = *p_2;
  ...
}
```

Figure 6: Using flow-insensitive alias information.

```
foo ()
{
  #   TMT.5_13 = VDEF <TMT.5_12>;
  p_2 = baz (&a, &b, &c, &d);

  #   TMT.5_14 = VDEF <TMT.5_13>;
  *p_2 = 5;

  #   TMT.5_15 = VDEF <TMT.5_14>;
  a = 3;

  #   TMT.5_16 = VDEF <TMT.5_15>;
  b = 2;

  #   VUSE <TMT.5_16>;
  T.1_7 = *p_2;
  ...
}
```

Figure 7: Effects of alias grouping.

to represent aliased loads and stores grows too large (configurable with `-param max-aliased-vops`), alias sets are grouped to avoid severe compile-time slow downs and memory consumption. The alias grouping heuristic essentially reduces the sizes of selected alias sets so that they are represented by a single symbol. This way, aliased references to any of those variables will be represented by a single virtual reference. Resulting in an improvement of compilation time at the expense of precision in the alias information.

Figure 7 shows the same example from Figure 6 compiled with `--param max-aliased-vops=1`. Notice how all four variables are represented by p's type memory tag, namely *TMT.5*. Even references to individual variables, like the assignment $a = 3$ are considered references to *TMT.5*.

8 Conclusions and future work

Tree SSA represents a useful foundation to incorporate more powerful optimizations and

analyses to GCC. Although the basic infrastructure is already functional and produces encouraging results there is still much work ahead of us. Over the next few months we will concentrate on the following areas:

Compile time performance. Currently, Tree SSA is in some cases slower than other versions of GCC. In particular, C++ programs seem to be the most affected. Preliminary findings point to memory management inside GCC and various data structures that are being stressed.

Another source of compile time slowness are the presence of RTL optimizations that have been superseded by Tree SSA. While some passes have already been disabled or simplified, there still remain some RTL passes which could be removed.

Run time performance. In general, Tree SSA produces similar or better code than other versions of GCC. However, there are still some missing optimizations. Most notably, loop optimizations, which we expect will be incorporated soon.

In the immediate future, most efforts will be focused on stabilizing the infrastructure in preparation for the next major release of GCC. Although the framework is tested daily on several different architectures, there are still some known bugs open against it and there are some architectures that have not been tested or have had limited exposure.

We expect the infrastructure to keep evolving, particularly as new optimizations are added, we will probably find design and/or implementation limitations that will need to be addressed. We have tried to make the basic design sufficiently flexible to permit such changes without overhauling the whole middle-end.

References

[1] D. Berlin, D. Edelsohn, and S. Pop. High-Level Loop Optimizations for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[3] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Lecture Notes in Computer Science, no. 457, Springer-Verlag, August 1992.

[4] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[5] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Reading, Mass.: Addison-Wesley, Redwood City, CA, 1996.