Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Interprocedural Constant Propagation and Method Versioning in GCC

Razya Ladelsky
IBM Research Lab in Haifa
razya@il.ibm.com

Mircea Namolaru
IBM Research Lab in Haifa
namolaru@il.ibm.com

Abstract

In recent years interprocedural dataflow optimization and analysis became a standard part of optimizing compilers. We implemented such an optimization in GCC—interprocedural constant propagation (IPCP), which tries to determine which parameters of given methods are known to be constant at compile-time. In order to implement IPCP we used the interprocedural framework being developed in GCC. We also extended this framework to support versioning (a.k.a. specialization or cloning) of methods, to serve IPCP and other interprocedural optimizations. In this paper we describe both the IPCP algorithm and implementation, as well the versioning support. In addition, we outline future plans for further extensions of IPCP.

1 Introduction

Interprocedural data flow analysis propagates information across a program's methods. The compiler can take advantage of this information later when optimizing individual methods. IPCP is an effective interprocedural data flow optimization that extends constant propagation beyond the single method boundary.

IPCP can be devised as either a context-insensitive or a context-sensitive optimization. The context-insensitive form determines for each method in the program, the formal parameters that have the same constant value in every invocation of the method. The context-sensitive form determines the parameters that have constant values each time a specific method is called from a specific callsite.

Callahan et al. [Cal86] use "jump functions" and "return jump functions" to perform IPCP. The computation of these functions is done by intraprocedural analysis that may be either flow-sensitive (depends on the control flow) or flow-insensitive (independent of the control flow).

Based on their work, we chose a flow-insensitive, context-insensitive implementation and pass-through-parameter jump functions, which we refer to simply as jump functions. Its aim is finding out whether a certain formal parameter receives the same constant value from all of the method's callsites. In such a case, we can propagate this constant to the method for the benefit of later optimizations.

The versioning utility can generate different versions of a method. This utility is generic and can be used by many interprocedural optimizations. IPCP uses this utility to create a new version annotated with IPCP results, the optimized version, usually invoked instead of the original one.

2 Interprocedural Analysis Framework in GCC

Initially, GCC compiled one method at a time. Each method was translated into an intermediate representation, the optimizations were performed and then its object representation was created.

Interprocedural analysis (IPA) requires information about multiple methods within the application program, therefore, the previous approach had to be modified. The compiler has to collect information (intermediate representation, cfg, profiling info, etc.) for every method. After this information is accumulated for all methods, the whole program can analyzed. Then the rest of compilation can occur, one method at a time.

There are different ways to store this information. One way is storing it on a disk. Another is to keep it in memory, as GCC currently does.

GCC IPA framework is evolving. When started implementing IPCP, a "high gimple tree" representation of each method was available. In order to support profile based inlining, this representation was replaced by "low-level gimple tree" representation that includes control flow graph information for each method. This is the current representation used by IPCP and other interprocedural analyses and optimizations. There are plans to replace it with a tree SSA representation. This will allow already existing tree SSA analyses and optimizations to be used by IPA.

One of the important data structures for IPA is the callgraph. This is a directed graph, where each node corresponds to a method and each directed edge from node1 to node2 corresponds to a direct call from the method represented by node1 to the method represented by node2. Indirect calls (called via pointer) are not represented at the moment in GCC's callgraph. Any method whose address is taken is considered to be a potential target of an invocation, which makes further analyses and optimizations very conservative.

3 IPCP Algorithm

IPCP algorithm requires an intraprocedural stage whose results are propagated through the callgraph by the interprocedural stage.

3.1 Intraprocedural stage

The intraprocedural stage of IPCP is done using a simple flow-insensitive analysis of the method. A flow-sensitive analysis can be performed, but experimental results have shown that the simpler analysis succeeds in most of the cases. At each callsite, the compiler constructs a "jump function" that represents the value passed by the callsite to each actual argument. These values include the following:

$(Formal, id)$ – the caller's formal parameter *id* is passed as an actual argument. This is called "pass through parameter."

$(Constant, val)$ – a constant is passed as an actual argument, and its value is *val*.

$(Unknown, \_)$ – neither of the above.

Figure 1 illustrates the jump functions for the callsites in `f` and `f1`, where `J(caller, callee, callsite, formal_of_callee)` is used for describing a jump function. In `f1`, `a` is modified, therefore, the value reaching `cs4` is not the same as the value of the formal parameter `a`. Thus, `J(f1, g, cs4, c) = Unknown`.

```
f (int a, int b) {
  cs1:  g (b,a);
  cs2:  g (a,1);
}

f1 (int a, int b) {
 cs3:  g (b,a);
 a = .... ;      // a modified
 cs4:  g (a,1);
}

g (int c, int d) {
 . . .
}

J (f, g, cs1 ,c) = b (Formal)
J (f, g, cs1, d) = a (Formal)
J (f, g, cs2, c) = a (Formal)
J (f, g, cs2, d) = 1 (Constant)

J (f1, g, cs3 ,c) = b (Formal)
J (f1, g, cs3, d) = Unknown
J (f1, g, cs4, c) = Unknown
J (f1, g, cs4, d) = 1 (Constant)
```

Figure 1: Jump functions example

Since we chose a flow-insensitive implementation, the modification of `a` affects all appearances of `a`, therefore `J(f1, g, cs3, d) = Unknown` as well. This example shows that modify information is needed for each formal parameter and method.

3.2 Interprocedural stage

The interprocedural stage takes all jump functions and computes a value, referred to as cval, for each formal parameter of each method. The possible values for cval include `(TOP, _)`, `(BOTTOM, _)`, and `(CONSTANT, val)`. Each cval is initialized to `(TOP, _)`, meaning that this formal has not yet been analyzed. If cval for formal f is `(CONSTANT, val)` it means that all callsites to this method visited so far have the same constant value "val" passed to f. Otherwise, the value is `(BOTTOM, _)`. Figure 2 outlines the interprocedural propagation.

There are three main methods referred:

`cval_compute` – given type and type_info of the actual argument *i* (returned by the jump function), it computes a new cval for formal *i* of callee as described in Figure 2.

`method_cval` – returns current cval of formal *i* of callee.

`cval_meet` – computes a new cval as described in Figure 2

Formals that have `(CONSTANT, val)` cvals at the end of the algorithm are those formals that are called with the same constant val in all invocations of this method.

3.3 IPCP algorithm example

We simplify the example in Figure 1, and consider only the methods f and g. We trace the computation of cvals for formals of method g, as function of f's formals cvals.

Initial cvals for formals of method g:

$$cval(g,c) = \texttt{(TOP, \_)}$$
$$cval(g,d) = \texttt{(TOP, \_)}$$

We assume that the order of processing of the calls to g is cs1, cs2.

For cs1:

cval (g, c) =

$$meet(cval(f,b), \texttt{(TOP, \_)}) = cval(f,b)$$

cval (g, d) =

$$meet(cval(f,a), \texttt{(TOP, \_)}) = cval(f,a)$$

For cs2:

cval (g, c) =

$$meet(cval(f,a), cval(g,c)) =$$

```
IPCP_PROP

 worklist = all methods
 while (worklist_not_empty)
   mt = remove method from worklist
   for (every callsite cs of mt)
     callee = callsite_callee (cs)
     for (every actual argument i of cs)
       (type, type_info) = J (mt, callee, cs, i)
       cval_new = cval_compute (type, type_info, i, callee)
       cval_old = method_cval (callee, i)
       cval_result = cval_meet (cval_new, cval_old)
       if (cval_changed (cval_old, cval_result))
         cval_set (callee, i, cval_result);
         add_method (worklist, callee);
       endif
     endfor
   endfor
 endwhile

end

cval_compute (Constant, val, i, callee) = (CONSTANT, val)
cval_compute (Formal, id, i, callee) = cval (id)
cval_compute (Unknown, _, i, callee) = (BOTTOM, _)

cval_meet ((BOTTOM, _), x) = (BOTTOM, _)
cval_meet ((TOP, _), x) = x
cval_meet ((CONSTANT, val1), (CONSTANT, val2))
                                  = (BOTTOM, _), if val1 != val2
                                  = (CONSTANT, val1), if val1 == val2
```

Figure 2: IPCP propagation

$$meet(cval(f,a), cval(f,b))$$

cval (g, d) =

$$meet(\texttt{(CONSTANT, 1)}, cval(g,d)) =$$

$$meet(\texttt{(CONSTANT, 1)}, cval(f,a))$$

Let's assume that there's a single call to f in the program *f (1, 1)*; in this case:

$$cval(f,a) = \texttt{(CONSTANT, 1)}$$

$$cval(f,b) = \texttt{(CONSTANT, 1)}$$

Therefore,

$$cval(g,c) = \texttt{(CONSTANT, 1)}$$

$$cval(g,d) = \texttt{(CONSTANT, 1)}$$

Let's assume now that the single call to f is *f (1, 2)*; in this case:

$$cval(f,a) = \texttt{(CONSTANT, 1)}$$

$$cval(f,b) = \texttt{(CONSTANT, 2)}$$

Therefore,

$$cval(g,c) = \texttt{(BOTTOM, \_)}$$

$$cval(g,d) = \texttt{(CONSTANT, 1)}$$

Let's assume now that the single call to f is *f (2, 2)*; in this case:

$$cval(f,a) = \texttt{(CONSTANT, 2)}$$

$$cval(f,b) = \texttt{(CONSTANT, 2)}$$

Therefore,

$$cval(g,c) = \texttt{(CONSTANT, 2)}$$

$$cval(g,d) = \texttt{(BOTTOM, \_)}$$

4 IPCP Implementation Overview

IPCP has been developed in the tree-profiling branch and is scheduled to be integrated in mainline 4.1. IPCP is one of the IPA passes, and is enabled by the `-fipa-cp` flag. The option `-fipa-dump-cp` dumps IPCP data structures and results, as well as the callgraph after IPCP transformation. The IPCP code can be found in two files, `ipa_prop.c` and `ipa_prop.h`. IPCP implementation is divided into three stages:

Intraprocedural stage—performs a local analysis of the method that computes the values passed at the method's callsite (construction of the jump functions).

Interprocedural propagation stage—uses this local information to solve the interprocedural problem.

Transformation stage—propagates the information about the constant-valued formals to the methods.

The first two stages implement the algorithm (described in section 3). The third stage relies heavily on versioning and will be explained in details in later sections.

4.1 Data Structures

Similar to any other IPA optimization, IPCP needs to annotate the callgraph with additional information. This information is divided into two main components: `ipa_node` and `ipa_edge` structures. `ipa_node` stores IPCP information related to a method and its formal parameters, and is pointed to by a field in the callgraph node (a cgraph node). It contains the following fields:

`ipcp_orig_node` – this field is non-null only for a versioned method. It points to the original node from which the method was versioned.

`ipcp_mod` – an array with an entry for each formal parameter of the method that indicates whether or not this formal parameter is modified inside this method.

`ipcp_cval` – an array in which each entry holds the cval for each formal parameter of this method. Each cval is implemented as a couple (type, value). Types of cval currently supported by the implementation are all integer and real constants, and also references to such constants (as in Fortran).

The `ipa_edge` structure stores IPCP information related to a callsite and its arguments, and is pointed to by a field in callgraph edge (a cgraph edge). It contains:

`ipcp_jump_func` – an array with an entry for each actual parameter representing the jump function for the argument. Each jump function is implemented as a couple (type, value). Types of jump function currently supported by the implementation are all integer and real constants, also references to such constants (as in Fortran), and pass-through parameters.

4.2 IPCP functions

The driver of IPCP is `ipcp_driver()`. The three stages of IPCP are implemented by the following functions:

`ipcp_init_stage()` – performs an intraprocedural analysis of all the methods, computing modify information and jump functions.

`ipcp_iterate_stage()` – performs the interprocedural stage based on the information computed in `ipcp_init_stage()`, by iterating over all methods and propagating IPCP information across the call graph. It stores its results in the cval data structure.

`ipcp_insert_stage()` – passes on the information computed by the algorithm for use by later optimizations. For safety reasons, specialization/versioning utility is required. Versioning and this last stage, `ipcp_insert_stage()`, are referred to later on.

5 Versioning

The versioning utility creates a full duplication of the method; the cgraph node and the gimple tree representation (including the cfg and the function struct) of the method are duplicated. The callgraph is updated as required. Versioning has been developed in the tree-profiling branch and is scheduled to be integrated in mainline 4.1.

The versioning utility resides in `cgraphunit.c`, `tree-inline.c`, and `gimplify.c`. It receives the original cgraph node to be duplicated, varray of cgraph edges representing the callers of the new version, and varray of `ipa_replace_map` structures representing a tree replacement (explained later). It returns a new cgraph node for the new version. The driver is `cgraph_function_versioning()`. It consists of three main functions:

`cgraph_copy_node_for_versioning()` creates a new cgraph node, whose contents are duplicated from the original cgraph node, and integrates the new node properly in the callgraph.

The callees (i.e., exiting) cgraph edges of the original node are duplicated—all methods called by the original version are also called by the new version.

The callers (i.e., entering) cgraph edges passed by the user are redirected to point to the new node.

A special case is an edge representing a recursive invocation, for example g→g. When duplicated, this edge's caller is g_versioned and its callee is g. The edge is redirected to point to the versioned node (g_versioned→g_versioned).

`tree_function_versioning()` duplicates the low-level gimple tree representing the method. It receives two FUNCTION_DECL tree nodes, one for the original method and one for the new copied method. It uses existing functions available in tree-inline.c to copy the method's arguments, cfg, and body.

Initially, these functions were written for inlining. They were adapted to support versioning. This makes sense because of the similarity between versioning and inlining. Inlining can be thought of as versioning the callee and then integrating it into the caller.

During the duplication it is possible to replace a subtree with a new one as specified

by the user. This information is contained in ipa_replace_map structs. This utility is very useful for IPCP, enabling constant propagation during duplication.

update_call_expr() After the body is copied, it modifies the call expr tree nodes of all callers of the versioned node with the new name of the versioned method.

When versioning a method, a unique new name must be created for it. We use the original name to create the new one, as follows: `new_name = 'original_name.<number>'`. The `'.'` makes sure the user won't create such a name (it is not a source code valid method name). The number is generated uniquely by the compiler. When the original name includes characters that are not alpha-numeric, such as operator<< , the name created might not be a valid assembly name. This is why we check the name and any non alpha-numeric characters are transformed to `'_'`.

When there are multiple compilation units, the same number might be generated for more than one compilation unit and then the name might not be unique. We define the version to be local (not externally visible outside the compilation unit), therefore, this problem will not interfere.

Versioning the same method in the same way (propagating the same constant) in multiple translation units, could have used the "link once" idea. This requires an ABI support for versioned names. We might consider this in the future.

6 Transformation Stage Implementation

This section describes in details IPCP's transformation stage, implemented in `ipcp_insert_stage()`.

6.1 Propagating the constant to the method

Due to the current callgraph implementation the only calls analyzed are direct calls present in our compilation unit. There could be other calls that are not analyzed: externally visible methods used outside our compilation unit and methods called via pointer. Thus propagating the formal-is-a-constant information into the original method is not safe. The purpose of using versioning is having two versions: the original one that stays unchanged for safety reasons, and the versioned method is annotated with the information regarding the formal being a constant value.

IPCP uses three ways to propagate a constant to the versioned method:

Building a new assignment statement `formal = constant` and inserting it at the beginning of the versioned method (this is the usual case).

Replacing all the uses of the formal with the constant in the body of the versioned method. This can be done for a read-only formal (known to have no definitions) with no address taken uses. A similar approach is used by the inliner for read-only formals. This is done by versioning that has the capability to replace a sub-tree with another.

Replacing the all indirect reference trees with the constant tree, for cases where the formal is a reference to a constant allocated in a read-only area (as in Fortran—see explanation further in Fortran section).

6.2 Updating the callgraph

IPCP iterates over all the methods and versions the ones having constant-valued cval(s) for its

```
f1 (x)
{
  g (5);
}

f2 (x)
{
   g (x);
}
```

```
f1 (x)
{
  g_versioned (5)
}

f2 (x)
{
  g_versioned (x)
}

f1_versioned (x)
{
  x = 5;
  g_versioned (5)
}

f2_versioned(x)
{
  x = 5;
  g_versioned (x)
}
```

Before versioning | After versioning

Figure 3: f1, f2 and g before and after versioning.

formal(s). No order of iteration is assumed, when a particular method is being versioned, its callers are the original callers and their versions generated up to that point (if any). IPCP versioning redirects all these callers to the new created version of this method. After all required methods are versioned, the callgraph has the property that if a method is versioned, the original method has no callers in the callgraph; the versioned method is being called instead. As illustrated below, the resulting callgraph may not be correct at this point.

In the example in Figure 3, suppose IPCP analysis found that f1, f2, and g (not shown) are invoked from all callsites with the constant 5. All three methods are versioned and the assignment `x=5` is inserted in `f1_versioned`, `f2_versioned`, and `g_versioned` (not shown). After versioning, all callers of `g (f1, f2, f1_versioned, f2_versioned)` call `g_versioned`.

The resulting callgraph is incorrect, as it is possible to invoke f2 with a parameter different than constant 5 (for instance an invocation via a pointer or outside the compilation unit). This parameter is passed through to `g_versioned`, but `g_versioned` is a special version of g that assumes the constant 5 as its formal parameter.

The problem described above may occur only for pass-through parameters. In such cases, the original method should call the original callee (and not the versioned one). When a method is called with a constant as argument (as in f1), it is correct to call the versioned method (annotated with that constant).

We fix the callgraph by redirecting all edges corresponding to callsites which do not pass the constant directly, to the original callees. The call exprs are modified accordingly.

6.3 Update profile information

Adding versioned methods and changing the callgraph breaks the profiling information. `ipcp_update_profiling()` updates profiling information for both the original methods and the versioned ones. The profiling information uses several counts relevant for versioning: cgraph node count, cgraph edge count, cfg bb count, and cfg edge count.

The cgraph node count represents the number of times the method was invoked. The cgraph edge count represents the number of invocations done via the corresponding call site. For a cgraph node, summing up the counts of entering cgraph edges and dividing by its count yields the “count scale.” The count scale describes which part of the calls to the method were direct calls (as indirect calls are not represented by a cgraph edge).

All counts of the versioned method (cgraph node count, exiting cgraph edges counts, cfg

bbs counts, and cfg edges counts) need to be multiplied by the count scale. All counts for the original method need to be multiplied by the complementary value `(1 - count_scale)`.

6.4 Minimizing IPCP versioning

One of the problems caused by versioning is the increased space and increased compile-time required for the application. In some cases we could eliminate versioning for methods local to the compilation unit. An example of local methods are static methods whose address doesn't escape the file unit. The IPCP algortihm could cause the versioning of all methods on a given path in the callgraph. The first method in the path is the one that receives the constant directly from the callsite; the rest of the methods receive the constant as pass-through parameter.

For example: f→g→h→k. Assuming f passes a constant to g, g and h pass-through the constant. At the end of the algorithm g, h, and k are versioned.

In order to decrease the number of versioned methods, we can trace these paths. If all methods on such a path are local to the compilation unit, we can avoid versioning of these methods and propagate the constants to the original ones. Another way to minimize versioning is profile-driven IPCP. It could restrict versioning to methods belonging to hot paths only.

7 Fortran

This section describes the special issues of IPCP enhancement for Fortran.

7.1 The Fortran callgraph

In Fortran each top-level program unit (i.e., function/subroutine) is an independent module. The Fortran front-end in GCC isn't type-safe between modules, and creates multiple declarations for the same method. In other words, each module has its own copy of any external declaration.

Because of this, the callgraph built for Fortran programs is not informative at all. Several cgraph nodes created for the same method are unrelated to one another. Because the connection is lost between the method definitions and the method calls, the IPCP analysis is pointless. In order to perform IPCP analysis for Fortran, a temporary local hack was used to merge the nodes according to the name of the method. IPCP supports Fortran constants, so as soon as a proper callgraph is available, IPCP analysis can be effective for Fortran as well.

7.2 Supporting Fortran

In Fortran, passing a constant to a method is actually passing an address of a temporary holding that constant value. Uses of the formal in a method are actually uses of the de-reference of the parameter. The temporary created by GCC for the constant is defined to be in a read-only memory area. See example in Figure 4. Support for Fortran constants (both real and integral constants) was added into IPCP.

The transformation stage, where the constant found by the algorithm should be passed on to the method, required special handling.

It is impossible to insert an assignment statement (see subsection 6.1) at the beginning of the method, as it would assign a value to a read-only area. However, since we know it is a read-only area, it cannot be changed, which gives

```
program main
 call foo (8)
end


subroutine foo(n)
 i=n
 print *, "i=", i
end
```

(a)

```
MAIN__ () {
  int4 C.479 = 8;
  foo (&C.479);
}

foo (n) {
  int4 i;
  int4 D.484;

  D.484 = *n;
  i = D.484;
  /* print */
}
```

(b)

Figure 4: Fortran example and its gimple representation; C.479 is a read-only memory allocated by the compiler.

us the ability to replace all de-references of the formal with the constant.

In the example presented in Figure 4, we cannot insert `*n = 8` at the beginning of `foo`, as `*n` is a read-only area. All uses of `*n` will be replaced with `8` in `foo`.

8 Status

The first implementation of IPCP in GCC did not include versioning and in order to ensure safety was very restricted. Since then, versioning was added and IPCP was extended to use it. IPCP has been enhanced to support all integer types, real types, and references to constants (Fortran).

IPCP efficiency was measured on SPEC2000. Because it has been developed on an evolving branch, the results have been rather unstable. Analysis of the results show that versioning has a significant impact—in some cases the versioned method gets inlined where the original didn't. Therefore, IPCP affects inlining decisions. As a future extension, the inliner could be taught to use IPCP results.

Regarding the applicability of IPCP, analysis of SPEC2000 found several instances where IPCP could help the performance. For example, in SPECFP wupwise benchmark, there's a method called 48 times with two constant-valued arguments. Due to IPCP, an optimized version of this method is produced.

9 Future

Below are potential enhancements to IPCP:

Using IPCP infrastructure as a propagation engine to propagate various properties for formal parameters in the callgraph. Some examples of this include:

- Detecting 'arrays passed as pointers.' This can be used e.g. by the vectorizer.
- Interprocedural range propagation.
- Inferring that a formal's value is divisible by constant, could be helpful e.g. for loop unrolling.

Reducing calling overhead. If a formal has no definition in the method, and all its uses were replaced by the constant found by IPCP then it is redundant and can be eliminated.

Multiple versioning for IPCP. We intend to support multiple versioning, which would support the case in which a formal receives more than one constant value. This may help improve run-time performance.

Profile-guided IPCP. In order to reduce the number of versions, IPCP will use profile information to make versioning decisions. This will be useful, especially for multiple versioning.

10 Acknowledgements

We would like to thank Jan Hubička and Steven Bosscher who reviewed the patches and helped going over design and implementation issues. We would also like to thank Revital Eres who contributed the major part of the versioning utility, the IBM Haifa team for helpful discussions, and all GCC contributors who offered help and comments.

References

[Cal86] David Callahan, Keith D. Cooper, Ken Kennedy, Linda Torczon. *Interprocedural Constant Propagation*. ACM SIGPLAN '86 Symposium on Compiler Construction.

[Grove] Grove, Dan, Linda Torczon. *Interprocedural Constant Propagation: A study of Jump Function Implementation*. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.

[Morgan] Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

[Muchnick] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Wolfe] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.