In *Proceedings of the 1996 International Conference on Compiler Construction* (CC '96), Linköping, Sweden, April 24-26, 1996. LNCS 1060, Springer-Verlag.

Generalized Constant Propagation A Study in C

Clark Verbrugge[⋆] and Phong Co and Laurie Hendren[⋆⋆]

{clump, phaedrus, hendren}@cs.mcgill.ca
School of Computer Science
McGill University
Montréal, Québec, Canada H3A 2A7

Abstract. Generalized Constant Propagation (GCP) statically estimates the ranges of variables throughout a program. GCP is a top-down compositional compiler analysis in the style of abstract intepretation. In this paper we present an implementation of both intraprocedural and interprocedural GCP within the context of the C language. We compare the accuracy and utility of GCP information for several versions of GCP using experimental results from an actual implementation.

1 Introduction

Generalized Constant Propagation (GCP) is a top-down compositional compiler analysis based on the style of abstract interpretation [CC77]. A GCP analysis statically approximates the possible values each variable could take at each point in the program. As an extension of constant propagation (CP), GCP estimates *ranges* for variables rather than their precise value: each variable at each point is associated with a minimum and maximum value.

We have implemented GCP for the full C language, in both intraprocedural and interprocedural forms. We have tested the accuracy of our method both by assessing the quality of GCP information, and by measuring the amount of information which could be useful to subsequent analyses. Our experiments have show GCP to be efficient and viable; programs with many procedures obviously benefit more from an interprocedural analysis, but surprisingly high accuracy can be achieved with just an intraprocedural analysis. The use of read/write sets and points-to analysis also enhance the accuracy of GCP, particularly when constants cross procedure calls.

GCP information has several uses. Since it is an extension of CP, the same reasons for using it apply: elimination of dead code, arithmetic optimizations, static range-checking, etc. Naturally, GCP will tend to be more powerful in these respects; the value of a variable may not be constant, but its range might still allow a conditional to be statically evaluated. Ranges can also be useful for subsequent analyses, such as various loop transformations. However, it is also true that GCP locates more exact constants than CP, making GCP valuable even if range information is not needed.

[⋆] Research supported in part by an NSERC Graduate Fellowship and FCAR.

[⋆⋆] Research supported in part by NSERC and FCAR

In examining the ouput of GCP analysis, we also noticed that the information could be useful for program understanding. For example, variables determined by GCP to be bounded can also be transformed to more tightly typed variables, such as booleans or subranges. This information could be added to the program, making the program easier to understand and simplifying further analyses.

1.1 Related Work

Constant propagation is a popular analysis that has appeared in numerous forms over the years [CCKT86, CH95, GT93, MS93, WZ91]. Generalized constant propagation, however, has not enjoyed as much attention: Harrison's [Har77] article on the *range propagation* problem is one of the few papers to address this type of analysis, albeit in a more 'traditional' setting based on precomputation of control-flow diagrams and *def* and *use* sets. GCP information also bears some resemblance to the attempts by software engineers to produce provably correct programs. Analyses developed for this goal propagate symbolic information in an attempt to establish loop and/or program invariants, either for annotation, or mechanical verification [DM81]. This is a large body of literature with very different methods, but GCP can in some sense be considered a spawn of such efforts. In fact, a paper by Bourdoncle [Bou93] describes a method called *abstract debugging*, which propagates range information top-down and bottom-up in order to locate potential program bugs.

More recently, Patterson described an analysis similar to GCP used for static branch prediction [Pat95]. Here, ranges are augmented with a probability that the actual value lies uniformly distributed within that range, and sometimes with symbolic information as well; this allows the results of conditionals to be estimated with greater accurracy. As with Wegman and Zadeck's constant propagation, this is based on a Static Single Assignment (SSA) representation.

In any range analysis, computing the values within loops will tend to be expensive —the domain of ranges tends to be quite large, and it can take a long time for ranges to converge. Patterson deals with this problem by heuristically matching templates to loop carried expressions, resorting to brute force for the unrecognized cases. The approach we develop in section 3.3 is the notion of *stepping,* or artificially moving the range up in the domain. This has also been addressed by Bourdoncle, and is similar to the *widening/narrowing* tactic of Cousot and Cousot [CC92].

1.2 Overview

Our analysis relies on an *Abstract Syntax Tree* intermediate representation of the program. In Sect. 2 we describe this structure, and how we can augment it to support interprocedural analyses as well as intraprocedural ones. Section 3 explains the semantic basis for our analysis and gives rules for the intraprocedural version. Using the framework of Sect. 2, we extend our analysis to a full interprocedural GCP in Sect. 4. We then give experimental results in Sect. 5, comparing GCP with CP and contrasting the effects of intraprocedural GCP, GCP with Read/Write sets, and the complete interprocedural GCP.

2 Background and Setting

The GCP analysis has been implemented within the context of the McCAT optimizing/parallelizing compiler[HDE$^+$92]. As illustrated in Fig. 1, the first phase of the compiler takes multiple C files as input and creates a SIMPLE intermediate representation of the complete program[Sri92]. This phase consists of a symbolic linker that combines all of the input C files into one complete representation of the program, a simplification phase that translates the higher-level intermediate representation to SIMPLE, and a restructuring phase that eliminates `goto` statements[EH94]. The overall objective is to create an intermediate form that is structured and simple to analyze. Typical transformations performed in the simplification phase include: compiling complex statements into a series of basic statements, simplifying all conditional expressions in `if` and `while` statements to simple expressions with no side-effects, structuring `switch` statements, simplifying procedure arguments to either constants or variable names, and moving variable initializations from declarations to statements in the body of the appropriate procedure. Fig. 2(a) gives an example C program, while Fig. 2(b) gives the equivalent simplified program.

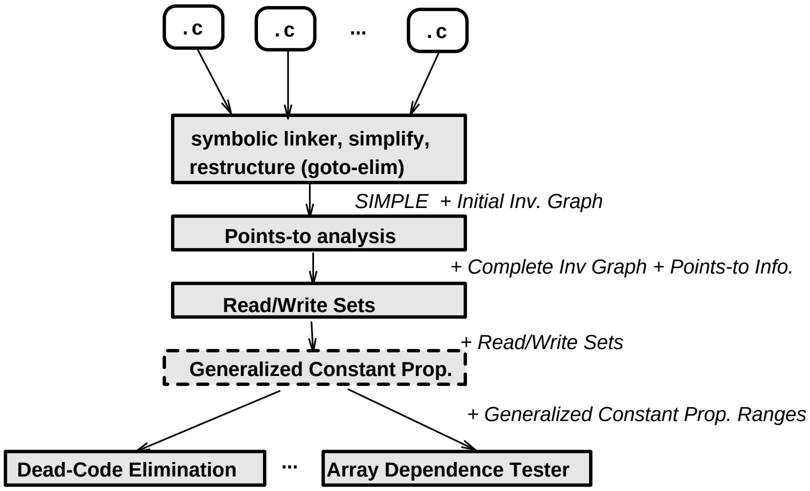

.c
.c
...
.c
symbolic linker, simplify, restructure (goto-elim)
SIMPLE + Initial Inv. Graph
Points-to analysis
+ Complete Inv Graph + Points-to Info.
Read/Write Sets
+ Read/Write Sets
Generalized Constant Prop.
+ Generalized Constant Prop. Ranges
Dead-Code Elimination
...
Array Dependence Tester

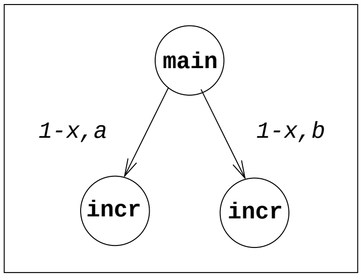
Fig. 1. Overview

GCP analysis also uses the output of points-to analysis and read/write set analysis. Points-to analysis is a context-sensitive interprocedural analysis that approximates the points-to relationships at each program point [EGH94]. For each indirection of the form `*x`, points-to information can be used to find the set of named locations pointed-to by `x`. Named locations include globals, parameters, locals, and special symbolic names that represent names that are not visible within the scope of a function, but are accessible via pointers. An example of a symbolic name is the name[3] `1-x` that is used to represent the location accessed via `*x` in the procedure `incr` in Fig. 2(b).

[3] We use `1-x` to denote the first dereference of `x`, `2-x` for the second dereference, etc.

```
int g1,g2;
void init_glob()
{ g1 = 1; }
void incr(int *x, int delta)
{ *x = *x + delta; }
main()
{ int a,b=3,c=4;
  g2 = 2;
  init_glob();
  scanf("%d",&a);
  incr(&a,c);
  incr(&b,g1);
  printf("%d %d %d %d %d \n",
             a,b,c,g1,g2);
}
```

(a) Example C Program



(c) invocation graph and mapping information

```
int g1,g2;
void init_glob()
{ g1 = 1; }          {g1}
void incr(int *x, int delta)
{ int t0;
  t0 = *x;            {t0}
  *x = t0 + delta;    {1-x}
}       t0:[-oo..oo] delta:[1..4]
main()
{ int a,b,c,*t1,*t2,*t3;
  b = 3;                  {b}
  c = 4;                  {c}
  g2 = 2;                 {g2}
  init_glob();            {g1}
  t1 = &a;                {t1}
  scanf("%d",t1);         {a}
  t2 = &a;                {t2}
  incr(t2,c);c:[4..4]     {a}
  t3 = &b;                {t2}
  incr(t3,g1);g1:[1..1]   {b}
 printf("%d %d %d %d %d \n",
           a,b,c,g1,g2);   {}
}        a:[-oo..oo] b:[4..4] c:[4..4]
            g1:[1..1] g2:[2..2]
```

(b) SIMPLE represenation with Write Sets and interprocedural GCP ranges

Fig. 2. Example Program, Write Sets and Invocation Graph

Points-to analysis also computes a complete invocation graph which captures all invocation contexts and the mapping information that is used to map location names in calling context to location names in called contexts. Fig. 2(c) shows the invocation graph for the example program. The root of the invocation graph is always `main`, and all calling chains are explicitly represented.[4] This is completely general; in the case of recursion (even mutual recursion), implicit cycles are introduced between matching *recursive* node and *approximate* nodes for the same function in order to represent all possible unwindings of the recursion; in the case of indirect function calls via function pointers, the list of functions pointed-to by the function pointer is given. Note that calls to library functions such as `scanf` and `printf` are not included in the invocation graph. The arcs in the invocation graph store the mapping information that was computed by points-to analysis. In our example, the first invocation of `incr` is represented

[4] There are several strategies for reducing the actual size of the invocation graph by sharing subtrees. However, it is conceptually simpler to think of the full unfolding of the invocation graph.

by left arc, and the mapping information indicates that the location name `a` in `main` corresponds to the symbolic name `1-x` in `incr`. Whereas, in the second invocation (right arc), the name `b` in `main` corresponds to `1-x` in `incr`. This mapping allows us to use one name within `incr` without losing the context-specific information from each invocation site. A more detailed description of the interprocedural environment, including the invocation graph can be found in [HEGV93].

Read/write set analysis uses the points-to information to calculate the locations read and written by each basic and compositional statement.[5] For the purposes of GCP, we only use the write sets calculated for procedure calls. Note that read/write sets include all local, symbolic, and global locations written by a procedure call.[6] Figure 2(b) gives the `write` set for each statement in our example program (shown in bold italics). Note that the second assignment in `incr` shows that the symbolic location `1-x` is written. Also note that the write sets for procedure calls are quite precise as both points-to and read/write set analysis are context-sensitive interprocedural analyses.

Figure 2(b) also gives the range information that is collected for the program. Each direct use of a variable has been decorated with the appropriate GCP information as computed by our interprocedural algorithm.

3 Intraprocedural GCP

Within a procedure, GCP is a straightforward top-down semantic analysis of the SIMPLE AST. The semantic domain is first specified; in our case we will be concerned with the domain of scalar *ranges.* A corresponding semantic function is then developed for every possible type of node, and the semantic analysis proceeds by pattern matching on the AST node type and branching to the appropriate function. This is complicated somewhat by the presence of indirection; whenever a pointer is dereferenced, the ranges for every possible target variable have to be merged. Information is further diluted by function calls, which can side-effect not just global variables, but local ones indirectly referred to by pointers. The success of GCP would thus seem to hinge on the accuracy of how it handles procedure calls, and on the accuracy of the points-to analysis.

3.1 Semantic Domains

GCP estimates the value(s) a variable can assume at each point in the program by estimating the minimum and maximum values each variable can reach. Our semantic domain is then the domain of *ranges*: closed (scalar) intervals, partially-ordered by inclusion with both a smallest element ($\perp = []$, the empty range) and a largest ($\top = [-\infty \ldots \infty]$, where by ∞ we mean the largest machine representable scalar). Note that most every data type in C fits comfortably into this paradigm; `chars`, `shorts`, `ints`, `longs` (`signed` and `unsigned`) of course, but also `floats` and `doubles`, as discrete approximations to real numbers, `structs`

[5] This is similar to MOD/REF analysis.

[6] The actual sets calculated are divided into the *definite* write set for those locations definitely written and the *possible* write set for those locations which may be written.

as aggregate scalars, and even pointers as unsigned integers. Arrays are approximated as the merge of their contents.

Ranges form a nice semantic domain. If $[a, b]$ and $[c, d]$ are (inclusive, closed) ranges, one can consider $[a, b] \sqsubseteq [c, d]$ if $a \geq c$ and $b \leq d$. The meet of two ranges $[a, b]$ and $[c, d]$ is then the range which includes them both: $[\min(a, c), \max(b, d)]$.

Our ranges are also discrete, even for the representation of real numbers; every element is finite, and there exist least upper bounds for arbitrary sets of elements. In other words, the domain is a Scott Domain. The existence of fixed-points for monotonic functions, and closure under cross-product are then guaranteed.

This domain does have one unpleasant property—it is quite 'tall,' i.e., one can form very long chains. This will have implications for how fixed points are computed during the analysis of loops and recursion, as will be seen later.

3.2 Semantic Functions

As mentioned, every type of statement in the SIMPLE AST (both expression and control) needs a semantic analogue; we need separate methods for determining how GCP information is altered by assignments, the various arithmetic operations, sequencing, conditionals, loops, ...etc. In order to ensure convergence, it is also necessary that each semantic function be monotonic in its domain.

We cannot describe all the semantic functions here, but in Figs. 3 and 4 we show the semantic operations corresponding to a few different kinds of statements in a compositional form. We hope that this will give the reader some idea of the flavour of the effort. Note that the structured nature of our SIMPLE representation lends itself to concise and relatively compact analysis rules.

Most of these functions are quite obvious; assignment requires locating the range for the right hand side value, and storing it as the range for the left hand side variable (the effect of pointers is discussed below). Semantic plus is paradigmatic of most arithmetic functions. The largest range which could result from the operation being applied to any combination of values in the operand ranges is computed and returned as the result.

Most semantic functions dilute information, due to the necessity of being conservative. Conditionals, though, can generate information. When control passes through a conditional, it is necessary that the condition be satisfied (then-part), or unsatisfiable (else-part), and this can be reflected in the range sets passed into the corresponding statements. For example, a statement such as `if(i<0)` implies that `i` must be less than `0` when entering the affirmative branch, and that `i` must be at least `0` within the negative branch. Every conditional we encounter therefore splits the input into two constrained sets (line 13, Fig. 4), which must be merged after the conditional is completed.

3.3 Loops and Stepping

Loops in a semantic analysis require the computation of a fixed point in the semantic domain. The process is illustrated starting from line 27 in Fig. 4 (other loop structures are similar); the GCP information coming out of the body of the loop is merged with the information entering the loop, and the process is

```
 /* Given statement S, an input GCP set, returns the output GCP set */
fun process_stmt (S,Input) =
  if basic_stmt(S)
    return(process_basic_stmt(S,Input))
  else
    case S of
      < SEQ(S1,S2) > =>
        return(process_stmt(S2,process_stmt(S1,Input)))
      < IF(cond,thenS,elseS) > =>
        return(process_if(cond,thenS,elseS,Input))
      < WHILE(cond,bodyS) > =>
        return(process_while(cond,bodyS,Input))
      ...

fun process_basic_stmt(S,Input) =
  case S of
    < x = y > =>
      /* RangeOf returns the range for the given variable in the given input set */
      return(Input − x:RangeOf(x,Input) + x:RangeOf(y,Input))
    < x = *y > =>
      /* MergeRanges returns the smallest range ⊒ every range in a given list.
         Dereference returns a list of variables pointed to by the given variable
         at the given statement. RangesOf is the list−version of RangeOf */
      [a..b] = MergeRanges(RangesOf(Dereference(y,S)),Input)
      return(Input − x:RangeOf(x,Input) + x:[a..b])
      /* Merge pair−wise merges two lists of ranges for the same set of variables.
         DefinitelyPointsTo returns true if the given variables have definite points-to
         relationship at the given program point */
    < *x = y > =>
      derefx = Dereference(x,S)
      if (DefinitelyPointsTo(x,derefx,S))
        return(Input − derefx:RangeOf(derefx,Input) + derefx:RangeOf(y,Input))
      /* If x does not definitely point to a single variable, then the
         strategy is to merge the range from the right−hand−side of the
         statement with the range of all variables x can possibly point to */
      foreach temp in derefx
        Input = Input − temp:Rangeof(temp,Input) +
                        temp:Merge(RangeOf(temp,Input),Rangeof(y,Input))
      return(Input)
    < x = y + z > =>
      [a..b] = semantic_plus(RangeOf(y,Input),RangeOf(z,Input))
      return(Input − x:RangeOf(x,Input) + x:[a..b])
      ...
```

Fig. 3. Compositional intraprocedural rules for GCP (continued on next figure)

```
fun semantic_plus([a..b],[c..d]) =
  if (a+c $< -\infty$) e = $-\infty$ else e = a+c
  if (b+d $> \infty$) f = $\infty$ else f = b+d
  return([e..f])

  /* Given a simple conditional cond, the statements in the then and
     else part, and GCP input, return the gcp output set. */
fun process_if(cond,thenS,elseS,Input) =
  /* ConstrainConditional splits the input set into two sets; one consistent
     with the conditional (to serve as input to the then statement), and one
     inconsistent (for the else statement). If either set is empty, then
     the input is such that the conditional can only have one possible outcome */
  (consistentInput,inconsistentInput) = ConstrainConditional(cond,Input)
  /* Do not process the then statement if we've determined that it
     cannot be executed! */
  if (consistentInput != {})
    thenOutput = process_stmt(thenS,consistentInput)
  else
    thenOutput = {}
  if (inconsistentInput != {})
    elseOutput = process_stmt(elseS,inconsistentInput)
  else
    elseOutput = {}
  return(Merge(thenOutput,elseOutput))

fun process_while(cond,bodyS,Input) =
  iterations = 0
  Output = oldBodyInput = bodyOutput = {}
  do
    Input = Merge(Input,bodyOutput)
    (consistentInput,inconsistentInput) = ConstrainConditional(cond,Input)
    converged = (consistentInput == oldBodyInput)
    if (not converged and iterations > maxiterations)
      /* StepUp artificially moves a non−converging range up in the
         semantic domain */
      StepUp(consistentInput,oldBodyInput)
      iterations = 0
    oldBodyInput = consistentInput
    bodyOutput = process_stmt(bodyS,consistentInput)
    Output = Merge(Output,inconsistentInput)
    iterations = iterations + 1
  while (not converged)
  return(Output)
```

Fig. 4. Compositional intraprocedural rules for GCP (continued from previous figure)

repeated until convergence. The output of the loop is gathered as the merged result of all sets of ranges which do not satisfy the conditional; in the actual implementation we also merge the results of `break` and `continue` statements with the Output and Input sets respectively.

Our semantic functions are all monotonic in their individual range-domains, so we will reach a fixed point eventually. Unfortunately, the domain of ranges can be quite "tall;" there are monotonic chains of *very* long length: e.g., $[0..0] \leq [0..1] \leq \ldots \leq [-\infty..\infty]$ where ∞ is typically 2^{31}. In the worst case then, our monotonicity requirement only ensures convergence after 2^{32} steps per variable, which is clearly unacceptable.

We can, however, speed up this process by sacrificing the quality of information. Ranges for variables that refuse to converge after some fixed number of iterations can be artificially "stepped up" (raised in the semantic domain). If each range can be stepped only so many times before reaching $[-\infty..\infty]$, then the monotonicity of our semantic operations guarantees convergence in much less time. By using some heuristics to guide the choice of which variable to step (e.g., choosing the loop index first), we can achieve a reasonable compromise between efficiency and accuracy. In our implementation, for instance, we have 2 non-converging iterations for each stepping operation. The first four steppings individually push the non-converging ends of the variables in the loop conditional to the loop bounds (if known), or to ∞ (or $-\infty$). The next step is to push some ($n = 40$ in our case) and then all the non-converging ends of all variables to ∞, then stepping all non-converging ranges to $[-\infty, \infty]$, and finally stepping all variables to $[-\infty, \infty]$. Thus, each fixed-point requires at most $14 + 2n$ iterations.

3.4 Considerations for C

Almost all languages have loops, and the difficulties they present to semantic analyses are not unique to C. The C language though does have two distinct features which greatly impact the efficacy of GCP: pointers and an abundance of procedure calls.

Pointers. Whenever a dereferenced pointer is encountered in the code, it is essential to know which variables might be accessed in order to compute the correct range information. When a dereferenced pointer appears on the right hand side of an expression, as an *R-value,* the semantic function computes the least upper-bound of all ranges which might be referred to as the result of the dereference (see the semantic $x = *y$ in Fig. 3). No matter which variable is actually accessed during runtime, it is then guaranteed to be included in the range GCP reports for the dereference. When a dereference occurs on the left hand side of an assignment, as an *L-value,* correctness requires that the range to be stored be merged with the existing range values for every variable which might be indicated by the dereference (see $*x = y$ in Fig. 3). This sort of conservative estimation can result in very poor information. If the set of variables accessed by an arbitrary pointer dereference is not known, all referenced variables must be assumed accessible.

Points-to analysis limits this sort of conservative dilution. By identifying target variables for each dereference, it is possible to restrict the number of ranges which have to be merged, or merged into. In a language like C, where pointers are ubiquitous, this sort of information is essential for reasonable accuracy.

Procedure Calls. Each time a procedure call is encountered, intraprocedural GCP must discard all information about any variable which might be altered by the function call. In the absence of information about where pointers might be directed, the most naive conservative approach is just to push every range up to $\top$; a slightly more clever tactic is to just raise all globals and any local variable which has had its address taken.

Points-to information allows GCP to more precisely determine which variables could be accessed by a function call. By computing the transitive closure of the possibly-points-to relation starting from the function call parameters and globals, the set of all variables which could be accessed can be determined.

Even with points-to this is still overly conservative. Pointers are often passed in C procedures to avoid copying information onto the stack, and not just to facilitate side-effects; pushing all accessible variables to $\top$ is clearly overkill. If read/write sets are available, though, it is possible to identify which variables might actually be written to during a function call. By just raising the variables in this latter set, the number of variables needlessly raised to $\top$ can be reduced.

4 Interprocedural GCP

The approximations used for intraprocedural GCP information over procedure calls are clearly suboptimal; in order to be surely correct, we seem to be forced to throw away a great deal of information. Even with the more accurate identification of altered or aliased variables possible with points-to and read/write sets, we still have to discard all information about the range of an altered variable. We cannot know exactly what the function does to the variables it changes, so it is necessary to assume the variables could be anything after a call.

An interprocedural analysis does not suffer from this limitation. By knowing the effect of each call on both local and global variables, we can determine, for instance, that a given procedure simply increments its value rather than computing an arbitrary function. Moreover, by using the invocation graph framework developed in section 2, we can compute interprocedural information in a context-sensitive way, avoiding the generalizations (and hence dilution of information) produced by the *calling context problem.*

4.1 Using the Invocation Graph

Making GCP interprocedural requires just two functions, *map* and *unmap*. As each function call is recursively traversed, the actual parameters passed to the callee are mapped to the formal parameters. The mapping information calculated by points-to analysis is used to map between names in the caller and symbolic names in the callee. As the function body is processed, the ranges computed from the current input set are merged with the existing ranges imbedded in the program from previous calls to the same function. Once the input set has completed the body, the values it contains are unmapped back to the caller's variables using the original map information. The ranges stored within the callee will then represent the merged input of all calls to that function, while the ranges returned after processing a function call represent the result of the call given the current input set from the caller (i.e. these values are context-sensitive). The process is shown functionally in Fig. 5 as a three-way branch on the invocation graph node (`ign`) type; non-recursive computations are illustrated in the first case, and map and unmap rules for GCP are shown in Fig. 6.

In the absence of recursion, this process is straighforward. When a recursive call appears, however, we are required to compute a fixed-point for the call representing all possible unrollings for the recursive call. This is indicated in our traversal of the invocation graph by a matching recursive and approximate node pair (linked by a backedge).

At each recursive node we store an input, an output, and a list of pending inputs. The input and output pair can be thought of as approximating the effect of the call associated with the recursive function (let us call it `f`), and the pending list accumulates input information which has not yet been propagated through the function. The fixed-point computation generalizes the stored input until it finds an input that summarizes all invocations of `f` in any unrolled call tree starting at the recursive node for `f`. Similarly, the output is generalized to find a summary for the output for any unrolling of the call tree starting in the recursive node for `f`. The generalizations of the input and output may alternate, with a new generalization of the output causing the input to change.

Consider the rule for the approximate node in Fig. 5; in this case, the current input is compared to the the stored input of the matching recursive node. If the current input is contained in the stored input, then we use the stored output as the result. Otherwise, the result is not yet known for this input, so the input is put on the pending list, and bottom ($\perp$) is returned as the result. Note that an approximate node never evaluates the body of a function, it either uses the stored result, or returns $\perp$.

Now consider the recursive rule. In this case we have an iteration that only terminates when the input is sufficiently generalized (the pending list of inputs is empty) and the output is sufficiently generalized (the result of evaluating the call doesn't add any new information to the stored output).

5 Experimental Results

In order to examine the relative merits of the different flavours of GCP, we need a qualitative way of measuring the GCP information produced. This is provided by dividing the ranges GCP can produce into four categories, according to their potential utility: **Exact,** an actual constant, like [3..3]; **Bounded,** a finite subrange, like [1...10]; **Half-open,** one end of the range is a number, but the other is infinite, like $[1..\infty]$; and **Total,** the range is $\top$, like $[-\infty..\infty]$.

We have counted the number of ranges falling into each of the four categories, for each of three different variations on GCP:

Naive: Intraprocedural GCP only. No points-to information; a pointer dereference returns all variables which have had their address taken. A function call causes all globals and all variables which have had their address taken to be set to $\top$.

```
/* Given a list of input ranges, parameters (actuals and formals), an invocation
   graph node for the function, the function body, and mapping information,
   returns the list of ranges resulting from the function call */
fun process_call(Input,actualList,formalList,ign,funcBody,mapInfo) =
  funcInput = gcp_map(Input,formalList,actualList,mapInfo)
  case ign of
    < Ordinary > =>
      funcOutput = process_stmt(funcBody,funcInput)
      return(gcp_unmap(Input,funcOutput,mapInfo))

    < Approximate > =>
      recIgn = ign.backEdge /* get partner recursive node in invoc. graph */
      /* if this input is contained in  stored input, use stored ouput */
      if isSubsetOf(funcInput,recIgn.storedInput)
        return(gcp_unmap(Input,recIgn.storedOutput,mapInfo))
      else  /* put this input in the pending list, and return Bottom */
        addToPendingList(funcInput,recIgn.pendingList)
        return Bottom

    < Recursive > =>
      ign.storedInput   = funcInput   /* initial input estimate */
      ign.storedOutput = Bottom     /* initial output estimate */
      ign.pendingList   = {}            /* no unresolved inputs pending */
      done = false
      do
        /* process the body */
        funcOutput = process_stmt(funcBody,ign.storedInput)
        /* if there are unresolved inputs, merge inputs and restart */
        if (ign.pendingList != {})
           ign.storedInput = Merge(ign.storedInput,ign.pendingList)
           ign.pendingList = {}
           ign.storedOutput = Bottom
        /* check to see if the new output is included in old output */
        else if isSubsetOf(funcOutput,ign.storedOutput)
           done = true;
        else /* merge outputs and try again */
           ign.storedOutput = Merge(ign.storedOutput,funcOutput)
      while (not done)
      /* return the fixed−point after unmapping */
      return(gcp_unmap(Input,ign.storedOutput,mapInfo))
```

Fig. 5. Compositional interprocedural rules for GCP

```
fun gcp_map(Input,formalList,actualList,mapInfo) =
  funcInput = {}
  foreach formalI in (formalList) /* formals inherit the range from actuals */
    funcInput = funcInput + formalI:RangeOf(actualI,Input)
  foreach globalI in (globalVarList) /* the range of globals remains the same */
    funcInput = funcInput + globalI:RangeOf(globalI,Input)
  foreach x in (SymbolicVars(mapInfo))
    mappedVars = getMappedVars(x,mapInfo)
    funcInput = funcInput + x:MergeRanges(RangesOf(mappedVars,Input))
    /* symbolic vars receive the merged range of the variables they represent */
  return(funcInput)

fun gcp_unmap(Input,funcOutput,mapInfo) =
  Output = Input /* initialize the Output of the call to its Input */
  foreach globalI in (globalVarList)
    Output = (Output − globalI:RangeOf(globalI,Input) +
             globalI:RangeOf(globalI,funcOutput))
      /* each global gets the new range from the called function */
  foreach x in (SymbolicVars(mapInfo))
    mappedVars = getMappedVars(x,mapInfo)
    foreach var in (mappedVars)
      Output = (Output − var:RangeOf(var,Input) + var:RangeOf(x,funcOutput))
                 /* each variable represented by a symbolic variable in
                 the called function gets the range of the symbolic variable */
  return (Output)
```

Fig. 6. Map and unmap functions for interprocedural GCP

R/W: Intraprocedural GCP that uses interprocedural points-to information, and read/write sets. Pointer dereferences return just the variables indicated by points-to. Function calls set all variables in the write set of the call to $\top$.

I-R/W Interprocedural GCP that uses interprocedural points-to information, and read/write sets. Pointer dereferences return just the variables indicated by points-to. Calls to user functions are evaluated using the using our context-sensitive strategy, and calls to library functions are approximated using read/write sets to set all variables in the write set of the call to $\top$.

In each case we only count the "relevant" ranges, by which we mean ranges for references to non-pointer variables appearing on the right hand side of assignments, as arguments to a function call, or as expressions in a loop or conditional. These are the ranges which could be of interest to a subsequent analysis. Since each analysis is run on the same program, each of our three cases gives the same total number of relevant ranges. The only difference is in how many ranges fall into each of our four categories.

Figure 2(b) indicates the ranges that we would count for our example program under the **I-R/W** strategy. In this case there are 9 uses of non-pointer variables, so there is a total of 9 relevant ranges, of which 6 are exact, 1 is bounded and 2 are total. Note how the context-sensitive nature of our interprocedural analysis keeps the two different calls to `incr` distinct, while still merging the values of `delta` within the procedure. The results for this program under the **Naive** and **R/W** strategies are much less precise. With **Naive** there are only 2 exact ranges corresponding to the 2 uses of the local variable `c`. With the **R/W** strategy one more exact range is found for the use of the global `g2`. In this case the R/W sets are used to determine that no procedure call kills the constant value generated by the assignment `g2 = 2`.

We have run the three kinds of GCP on the following benchmark set:

Asuite: Compiler test suite from Argonne National Labs.
Chomp: Solves a simple board game.
Circle: An $O(n^4)$ minimum spanning circle algorithm.
Clinpack: Numerical test routines.
Cluster: Two greedy graph clustering algorithms.
Dhrystone: Standard timing benchmark.
Frac: Computes rational representation of a real number.
Mersenne: Computes n digits of $2^p - 1$ for a given p and n.
Nrcode2-4: Another test suite for vectorizing C compilers.
Numerical: Complex number routines – zroots, laguer.
Stanford: Baby benchmarks suite – 10 small programs.
Tomcatv: A standard Fortran benchmark, ported to C.

Of course the data GCP collects will be greatly influenced by many factors. In the left columns of Table 1, we show the number of SIMPLE statements in the program, function definitions, function calls (includes calls to library functions), global variables, maximum loop nesting and total number of loops. It should be expected that programs having more functions, calls and globals will benefit more from using read/write sets, and from using an interprocedural GCP.

Benchmark	Stmts	Funcs	Calls	Globals	Nest	Loops	F-Ps	Avg iter	Intra(s)	Inter(s)
Asuite	1841	93	299	23	3	218	623	8.10	16.08	20.39
*Chomp	439	20	54	5	2	22	259	6.33	0.89	7.07
Circle	251	4	5	12	4	8	266	6.59	58.91	57.75
Clinpack	909	11	53	23	2	33	1701	8.56	11.13	96.16
Cluster	599	20	63	18	3	53	565	6.08	21.24	28.54
Dhrystone	242	14	20	65	2	7	71	17.30	9.98	55.81
Frac	103	2	3	0	2	3	9	7.00	0.33	0.50
Mersenne	117	8	17	3	2	7	102	7.52	0.18	1.87
Nrcode2-4	405	3	36	28	2	44	82	9.16	6.33	6.44
Numerical	319	11	18	11	2	11	33	3.97	1.29	5.70
*Stanford	998	47	84	67	3	88	4058	5.46	26.93	456.88
Tomcatv	333	2	15	7	3	19	310	4.89	46.09	50.24

Table 1. Benchmarks descriptions. Dynamic measurements are I-R/W. An "*" indicates the presence of recursion.

In Fig. 7, we illustrate the relative quality of the analyses based on how ranges are divided up into the four different kinds, and the relative quality of regular constant propagation as well. The length of the black bars show the percentage of exact ranges as found by GCP and CP, and the length of the dark gray bars shows the extra constants found just by GCP. The total length of the bars show the percentage of ranges that give at least some information (exact, bounded, or open). It is interesting to note that a suprisingly large fraction of ranges contain at least some useful information, and that in several cases (`stanford`, `mersenne`, `dhrystone`, and marginally `clinpack` and `asuite`) GCP locates more constants than CP. These extra constants are the result of merged information being subsequently reduced to a constant, such as a boolean (`b=`[0..1]) within "`if(b) {`...`}`." In comparing the effectiveness of **naive** versus **R/W** and **I-R/W**, however, the results seem to depend very much on the style of the benchmark under analysis.

The **naive** analyses of `circle`, `cluster`. `numerical` and `tomcatv2` give almost the same results as the more expensive **R/W** and **I-R/W** analyses. For the latter, there is only one large main function, all scalars are local variables, and there are no pointers to local variables. Thus, the **naive** scheme does not make any overly conservative assumptions, and the results are quite accurate. In the three former cases, there are almost no constants to speak of, regardless of the form of the analysis.

Most of the remaining benchmarks show improved results for the **I-R/W** method. In these cases constants are propagated through parameters, and so interprocedural results are substantially better. However, `dhrystone`, `nrcode` and `clinpack` all show benefits from using R/W sets; here, the reduced kill-sets provided by the R/W analysis (particularly with respect to library functions) allow more constants to be carried across procedure calls.

GCP is a semantic analysis on a tall domain, so the amount of time needed by GCP is important to consider. In Table 1 we also show dynamic measurements of the number of loop fixed-points, the average number of iterations for each fixed-point, and the total time (user time in seconds, on a Sun Sparc 20) consumed by both **R/W GCP** and **I-R/W GCP**. Note that these times are for our unoptimized code; a reduction in time by a constant factor would be easy to achieve with a less naive implementation of range sets, and, particularly in the **I-R/W** case, by including memoization. Relative relationships, however, are valid. Intraprocedural GCP takes time roughly proportional to nesting, but interprocedural GCP can take much longer due to procedure calls imbedded in loops, and recursion. Also note that, due to the stepping heuristics, the computation of each fixed-point actually requires very few iterations.

6 Conclusions and Further Work

We have demonstrated the efficacy of GCP in the context of the full C language. Moreover, we have shown that while a full interprocedural analysis certainly improves information for programs with constants passed via parameters, reasonable accuracy can be achieved with just a straightforward intraprocedural analysis for programs with simpler control structure. GCP also tends to locate more exact constants than CP.

A common criticism leveled at analyses based on abstract interpretation is the exponential cost of computing fixed points. GCP has also shown itself to

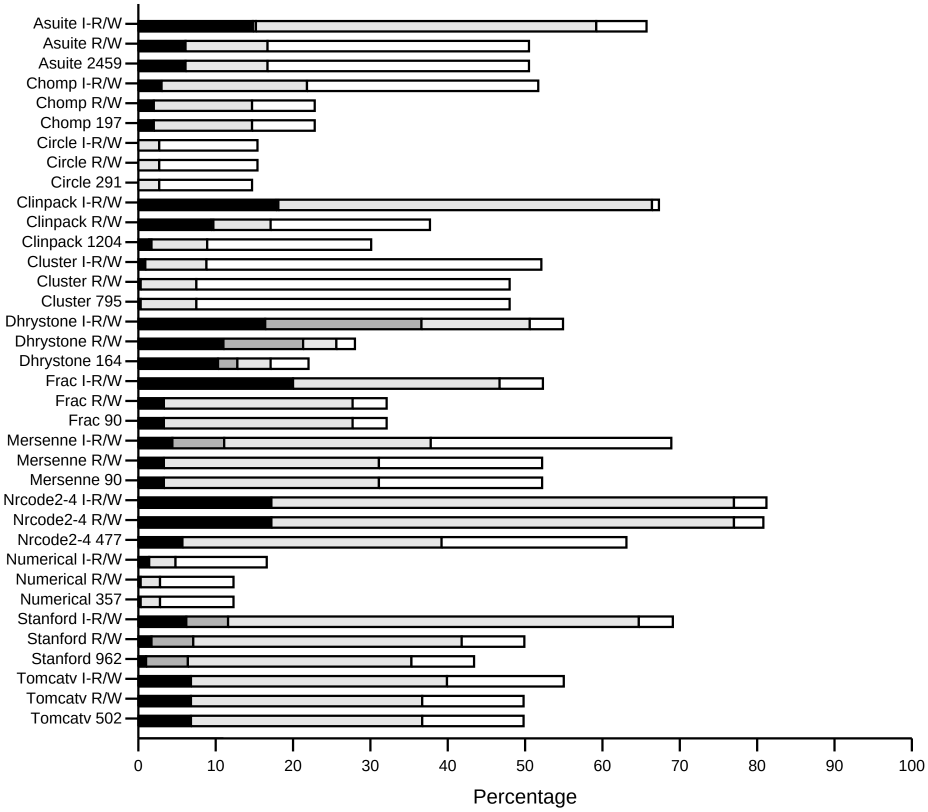


Fig. 7. Relevant ranges found for benchmarks. The percentages of the different types of ranges are show; black and dark gray are for exact constants (regular CP and extras found by GCP), light gray is bounded, and white for half-open. The remaining ranges are total. The number next to each benchmark name is the actual total number of relevant ranges in the program.

be quite reasonable in this respect; by stepping ranges that do not converge, fixed-points can be calculated in just a few iterations per loop. The inclusion of a simple heuristic, such as stepping the variable involved in the loop conditional first, permits rapid convergence without overly sacrificing quality of information.

We are also considering the effects of a few simple heuristics to enhance intraprocedural GCP. It should be possible to improve **naive** GCP by identifying the more common situations where GCP unnecessarily discards information in order to be safe, such as over selected library calls. The use of stepping also requires further examination; perhaps accuracy or speed can be improved with different heuristics. The effect of GCP on other analyses that use GCP information also remains to be examined: how often are ranges actually useful?

References

[Bou93] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. of SIGPLAN PLDI '93*, pages 46–55, Albuquerque, N. Mex., Jun. 1993.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Conf. Rec. of POPL-4*, pages 238–252, Los Angeles, Calif., Jan. 1977.

[CC92] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening / narrowing approaches to abstract interpretation. Technical Report LIX/RR/92/09, Ecole Polytechnique Laboratoire d'Informatique, 91128 Palaiseau Cedex, France, Juin 1992.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, pages 152–161, Palo Alto, Calif., Jun. 1986.

[CH95] Paul R. Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *Proc. of SIGPLAN PLDI '95*, pages 23–31, La Jolla, Calif., Jun. 1995.

[DM81] Nachum Dershowitz and Zohar Manna. Inference rules for program annotation. *IEEE Transactions on Software Engineering*, SE-7(2):207–222, Mar. 1981.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of SIGPLAN PLDI '94*, pages 242–256, Orlando, Flor., Jun. 1994.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proc. of the 1994 Intl. Conf. on Computer Languages*, pages 229–240, Toulouse, France, May 1994.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proc. of SIGPLAN PLDI '93*, pages 90–99, Albuquerque, N. Mex., Jun. 1993.

[Har77] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. on Software Eng.*, 3(3):243–250, May 1977.

[HDE+92] 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 *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Publ. in 1993.

[HEGV93] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural analysis framework for C compilers. ACAPS Tech. Memo 72, Sch. of Comp. Sci., McGill U., Montréal, Qué., Jul. 1993. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[MS93] Robert Metzger and Sean Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(1–4):213–232, Mar.–Dec. 1993.

[Pat95] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proc. of SIGPLAN PLDI '95*, pages 67–78, La Jolla, Calif., Jun. 1995.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill U., Montréal, Qué., Sep. 1992.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.