Interprocedural Constant Propagation using Dependence Graphs and a Data-Flow Model

David Binkley
Loyola College in Maryland
4501 North Charles Street, Baltimore Maryland 21210-2699, USA.
binkley@cs.loyola.edu

Abstract. Aggressive compilers employ a larger number of well understood optimizations in the hope of improving compiled code quality. Unfortunately, these optimizations require a variety of intermediate program representations. A first step towards unifying these optimizations to a common intermediate representation is described. The representation chosen is the program dependence graph, which captures both control-flow and data-flow information from a program.

The optimization of (interprocedural) constant propagation is studied. The algorithm developed combines a program dependence graph called the system dependence graph (SDG) with the ideas of data-flow computing and graph rewriting. The algorithm safely finds the classes of constants found by other intraprocedural and intraprocedural constant propagation algorithms. In addition, the SDG allows constants to propagate *through* procedures. This enables the algorithm to discover constants in a calling procedure even thought no constants exist in the called procedure.

1. Introduction

Optimizing compilers employ a larger number of well understood optimizations in the hope of improving object code quality. These include, for example, code motion, call inlining, constant propagation, dead code elimination, loop interchanging, and register allocation [1, 2]. Unfortunately, these optimizations often require different intermediate representations of the program.

This paper describes the first step toward reducing the number of intermediate representations by unifying optimizations onto a common intermediate representation. The representation chosen is a variation of the program dependence graph [12, 17] called the *system dependence graph* (SDG) [15]. Program dependence graphs have been successfully used as an intermediate representation in parallelizing and vectorizing compilers to perform loop interchanging, strip mining, loop skewing, and other optimizations [2, 17]. A key feature of the SDG is its representation for programs with procedures and procedure calls. Recent trends toward programs with small procedures and a high proportion of procedure calls (typical of object oriented programs) have intensified the need for better interprocedural optimizations.

The optimization considered in this paper is interprocedural constant propagation [8, 19]. The goal of any constant propagation algorithm is to identify variables whose values are constant throughout all possible executions of the program. If a compiler can identify such variables it can improve the quality of the object code by replacing the variable with the constant and by propagating this constant to other expressions in the program. *Interprocedural* constant propagation concerns the propagating of constants to, from, and *through* called procedures.

The interprocedural constant propagation algorithm developed in this paper uses a *data-flow interpretation* of the SDG, incorporates *symbolic execution*, and uses *live code analysis*. The algorithm *safely* finds the same classes of constants found by traditional interprocedural and intraprocedural constant propagation algorithms [16, 19].

(A safe algorithm may miss some constants but all the variables it determines to be constant are guaranteed to be constant on all possible executions of the program.)

The data-flow interpretation of the SDG treats the SDG as a data-flow graph in which values "flow" on the dependence edges of the graph. This makes (constant) values available at uses as soon as they are determined (they do not have to propagate through unrelated nodes as in a control-flow-graph based algorithm).

The algorithm also employs symbolic execution of the SDG. This idea is a modification of the graph rewriting semantics for program dependence graphs given by Selke in [18]. It allows, for example, the optimization of a while-loop that sums the numbers from 1 to 10: while constant propagation alone would determine that a non-constant value is produced by the statement "$sum = sum + i$" in the body of the loop (because *sum* is assigned the values values 1, 3, 6, $\cdots$, 55), symbolic execution replaces the loop with the assignment statement "$sum = 55$."

Live-code analysis is used to increase the number of constants found. Non-live (*i.e.*, dead) code includes unreachable code and useless code. Unreachable code is never executed on any execution of the program, while useless code has no effect on the output of the program. Removing unreachable code may improve constant propagation by reducing the number of definitions that reach a use. For example, if a constant and a non-constant value reach a use of x, then x's value is non-constant. If, however, the source of the non-constant value reaching the use of x is determined to be dead code and eliminated, then x's value is constant. Removing useless code does not improve constant propagation, but does reduce the size of the optimized code.

The remainder of this paper is organized as follows: Section 2 reviews the system dependence graph, Selke's graph rewriting semantics, and the constant propagation lattice. Section 3 presents the interprocedural constant propagation algorithm based on the SDG as an intermediate representation. Section 4 compares this algorithm with existing constant propagation algorithms and Section 5 presents conclusions.

2. Background

2.1. The System Dependence Graph

This section summarizes a minor extension to the definition of the SDG presented in [15]. The SDG models a language with the following properties:

1. A complete system is a single main procedure and a set of auxiliary procedures.
2. Parameters are passed by value-result.[1]
3. Input and output are modeled as streams; thus, for example, $print(x)$ is treated as the assignment statement "$output\_stream = concatenate(output\_stream, x)$."

We make the further assumption that there are no calls of the form $P(x, x)$ or of the form $P(g)$ for global variable g. The former restriction sidesteps potential copy-back conflicts. The latter restriction permits global variables to be treated as additional parameters to each procedure; thus, we do not discuss global variables explicitly.

The SDG for system S contains one procedure dependence graph (PDG) for each procedure in S connected by interprocedural control- and flow-dependence edges. The PDG for procedure P contains vertices, which represent the components of P, and edges, which represent the dependence between these components. With the

---

[1] Techniques for handling reference parameter passing and aliasing are discussed in [15] and [6].

exception of call statements, a single vertex represents the predicates of if and while statements, assignment statements, input statements, and output statements of P; in addition, there is a distinguished vertex called the *entry vertex*, and an *initial-definition vertex* for each variable that may be used before being defined. Initial-definition vertices represent the assignment of the value 0 to these variables.

A call statement is represented using a *call* vertex and four kinds of *parameter vertices* that represent value-result parameter passing: on the calling side, parameter passing is represented by *actual-in* and *actual-out* vertices, which are control dependent (see below) on the call-site vertex; in the called procedure parameter passing is represented by *formal-in* and *formal-out* vertices, which are control dependent on the procedure's entry vertex. Actual-in and formal-in vertices are included for every parameter and global variable that may be used or modified as a result of the call; formal-out and actual-out vertices are included for every parameter and global variable that may be modified as a result of the call. (Interprocedural data-flow analysis is used to determine which parameters and globals may be used and/or modified as a result of a procedure call [3, 4].)

PDGs include three kinds of intraprocedural edges: *control dependence edges, data dependence edges,* and *summary edges*. The source of a control dependence edge is either the entry vertex, a predicate vertex, or a call-site vertex. Each edge is labeled either **true** or **false**. A control dependence edge $v \longrightarrow_c u$ from vertex v to vertex u means that during execution, whenever the predicate represented by v is evaluated and its value matches the label on the edge to u, then the program component represented by u will eventually be executed provided the program terminates normally (edges from entry and call-site vertices are always labeled **true**; these vertices are assumed to always evaluate to **true**). Note that for the block structured language studied here each vertex has a single incoming control edge.

There are two kinds of data dependence edges, *flow dependence edges* and *def-order dependence edges*. Flow dependence edge $v \longrightarrow_f w$ runs from a vertex v that represents an assignment to a variable x to vertex a w that represents a use of x reached by that assignment. The set of flow dependence edges is divided into two subsets: *loop-independent* and *loop-dependent*: loop-dependent edges represent values passed from one loop iteration to another; all other flow dependence edges are loop-independent. A def-order edge $v \longrightarrow_{do(u)} w$ runs between vertices v and w, where both v and w represent assignments to x, the definitions at v and w reach a common use at u, and v lexically precedes w (*i.e.*, v is to the left of w in the system's abstract syntax tree).

Summary edges represent transitive dependences due to calls. A summary edge $v \longrightarrow_s u$ connects actual-in vertex v at a call-site to actual-out vertex u at the same call site if there is a path in the SDG from v to u that *respects calling context* by matching calls with returns.

Connecting PDGs to form the SDG is involves, the addition of three kinds of interprocedural edges: (1) a *call* edge connects each call vertex to the corresponding procedure-entry vertex; (2) a *parameter-in* edge connects each actual-in vertex to the corresponding formal-in vertex in the called procedure; (3) a *parameter-out* edge connects each formal-out vertex to the corresponding actual-out vertex at all call sites. (A call edge is an interprocedural control edge; parameter-in and -out edges are interprocedural data dependence edges.)

Example. Figure 1 shows the PDG of a system without call statements. Figure 3(b) shows part of the SDG for a system with a call statement. In Fig. 3(b) at the call-site, the actual-in vertex for *a* is labeled "$x_{in} = a$" and the actual-out vertex "$a = x_{out}$." In procedure P, the formal-in vertex for *x* is labeled "$x = x_{in}$" and the formal-out vertex "$x_{out} = x$." In these figures as well as the remaining figures in the paper, actual-in, actual-out, formal-in, and formal-out vertices for the variables *output_stream* and *input_stream* are omitted for clarity.

2.2. A Graph Rewriting Semantics for the SDG

This section first overviews Selke's graph rewriting semantics for (single procedure) program dependence graphs and then extends this to a graph rewriting semantics for the SDG. Graph rewriting first chooses a vertex that has no incoming control or loop-independent dependence edges. It then fires the vertex: the vertex is removed along with its outgoing edges. In addition, it may add, remove, or update other vertices and edges in the graph. These additional actions for each kind of vertex are given informally in the following table; formal definitions are given in [18].

Vertex Kind	Rule
Enter:	No additional actions.
Assignment:	For an assignment vertex labeled "$x = exp$", modify the targets of all outgoing edges as follows: replace *x* with *c tagged* by *x*, written c^x, where *c* is the constant value of *exp* (*exp*'s value is constant because the assignment vertex has no incoming data dependence edges). Tags allow subsequent rewritings to overwrite previous ones. Their need is illustrated in Fig. 2.
If-predicate:	Assume the predicate's value is **true** (the **false** case is symmetric). Remove all vertices representing statements in the **false** branch of the if statement along with their incoming and outgoing edges.
While-predicate:	If the predicate evaluates to **false** then all vertices representing the loop are removed along with their incoming and outgoing edges. Otherwise, if the predicate evaluates to **true** then copies of the vertices that represent the body of the loop are added to the graph. Copies are added as if one iteration of the loop had been unrolled and place before the original while-loop. (The difficult part of this rewriting is correctly determining the dependence edges associated with the new vertices).

In addition to modifying the graph, the rewriting of an input or output vertex effects the global input or output stream: input statements consume values from the head of the input stream, while output statements append values to the output stream.

Example. For the program in Fig. 1, Fig. 2(a) shows the graph after rewriting the entry vertex and then the vertices labeled "$x = 0$" and "$y = 1$" (in either order). The rewriting of "$x = 2$" illustrates the need for tags: because 0^x in the output vertex has the tag *x*, its value is replaced as a result of rewriting "$x = 2$" (Fig. 2(c)). Without tags, there is no way of knowing that the 0 was a rewritten *x*. Finally, rewriting the output vertex produces a 2 on the output stream (the tag is stripped when the value is output) and leaves the empty graph; thus, the evaluation terminates.

Rewriting the SDG

Rewriting the SDG requires adding a rule for call-site vertices. An information statement of this rule is given below, a formalization can be found in [5], which also indirectly extends Selke's semantics to the SDG. Like the while-loop rule, the call-site rule introduces copies of vertices. These new vertices are copies of the vertices

```
main ()
{
    x = 0;
    y = 1;
    if (y > 0)
        x = 2;
    print (x);
}
```

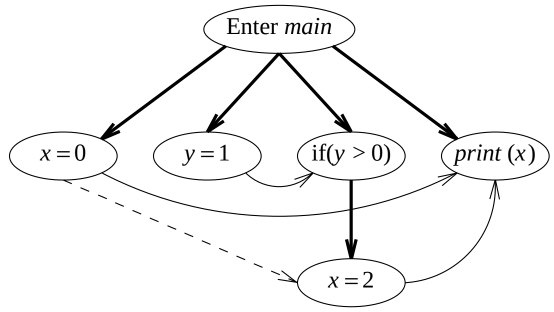

Figure 1. An example program and its program dependence graph. Bold arrows represent control edges, arcs represent flow-dependence edges, and dashed lines represent def-order edges. (In all the figures in this paper, unlabeled control edges are assumed to be labeled **true**.)

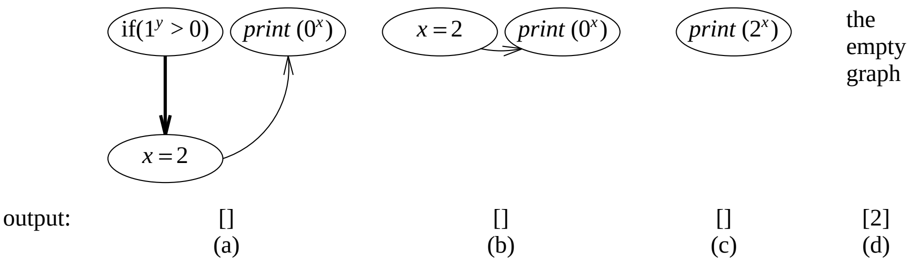

Figure 2. The rewriting of the program dependence graph from Fig. 1.

from the called procedure. Unlike the while-loop rule, computing the edges for the copies is trivial because of the interface (parameter vertices and edges) between a call site and the called procedure.

Vertex Kind	Rule
Call-site	Replace the call-site vertex with copies of the vertices for the called procedure (except its entry vertex). Change the interprocedural data dependence edges into (intraprocedural) flow dependence edges and the parameter vertices into assignment vertices.

Example. Figure 3 shows a program with a call and part of its rewriting. Figure 3(b) shows the SDG for the program shown in Fig. 3(a) after rewriting the enter vertex. Figure 3(c) shows the result of rewriting the call-site vertex "*call P*" where the flow dependence edges from "$x_{in} = a$" to "$x = x_{in}$" and from "$x_{out} = x$" to "$a = x_{out}$" have replaced the two interprocedural data dependence edges in Fig. 3(b).

2.3. The Constant Propagation Lattice

This section, based largely on [19], discusses the lattice used for constant propagation and introduces the notion of constant *confidence*. As shown in Fig. 4, the constant-propagation lattice is composed of a top element $\top$, an infinite collection of constants c_i, and a bottom element $\bot$. The value $\top$ represents "no information" or

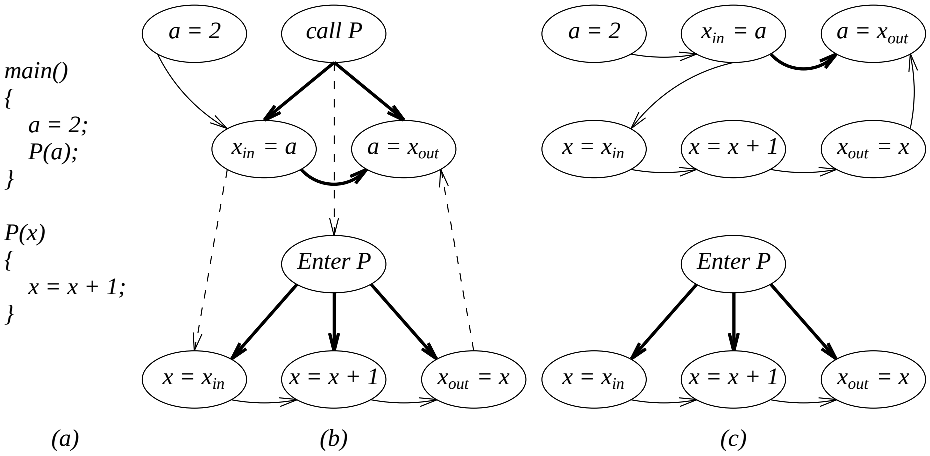

Figure 3. The call-site rewrite rule.

optimistically some yet undetermined constant. The value $\perp$ represents "non-constant", and is used, for example, in the case of a read statement.

Following the data-flow model, each flow dependence edge in the SDG is labeled by a lattice element, which represents the current best approximation to the value "flowing" down the edge (*i.e.*, the value produced by the assignment statement at the source of the edge). Initially, all edges are labeled $\top$, which represents the optimistic assumption that they will be labeled by some constant.

When a use of a variable is reached by multiple definitions, the definitions are combined by the lattice *meet* operator $\sqcap$ defined by the following table:

Rules for $\sqcap$	
$x \sqcap \top = x$	$c_i \sqcap c_j = c_i$ if $i = j$
$x \sqcap \perp = \perp$	$c_i \sqcap c_j = \perp$ if $i \neq j$
where x represents any of $\top$, $\perp$, or c_i	

The evaluation of expressions is strict: if any of the variables in the expression has the value $\perp$, then the expression has value $\perp$. (In some cases, this can be softened by taking special attributes of certain operators into account. For example, 0 times anything, including $\perp$, is 0.)

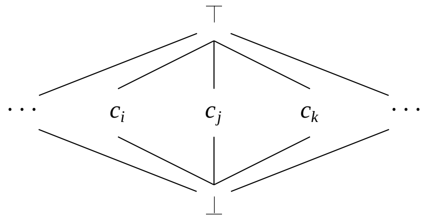

Figure 4. The three-level constant propagation lattice.

Confidence

Each constant in the program is assigned a confidence: *high* or *low*. High confidence is assigned to constants from the original program and all constants derived solely from high-confidence constants. Low confidence is assigned to all constants produced by operations involving low-confidence constants and $\top$).

High confidence constants will not be changed by further constant propagation. This allows more aggressive optimization to be performed. Low confidence constants may change as a result of updates in other parts of the graph. Thus, low confidence is assigned to a constant to prevent the unsafe removal of graph components.

Formally, confidence is modeled using the following lattice

$$\begin{array}{c}\top_{confidence} = \text{high confidence} \\ | \\ \bot_{confidence} = \text{low confidence}\end{array}$$

where the meet operation $\sqcap_{confidence}$ is defined as follows:

$$\top_{confidence} \sqcap_{confidence} \bot_{confidence} = \bot_{confidence}.$$

In what follows, we omit the subscript *confidence*, when it is clear from context.

3. Interprocedural Constant Propagation

This section describes the interprocedural constant propagation algorithm based on the SDG in which constants “flow” through the SDG. The algorithm is enhanced by concurrently performing symbolic execution and live code analysis. Symbolic execution is accomplished using a variation of the rules from Sect. 2.2. Live-code analysis is performed by initially labeling all vertices as non-live and then marking vertices as live only as they are encountered during constant propagation. When the algorithm terminates remaining non-live vertices represent dead code that can be removed from the SDG.

Before presenting the interprocedural constant propagation algorithm in Sect. 3.2, Sect. 3.1 presents a data-flow based intraprocedural constant propagation algorithm. The algorithm in Sect. 3.2 will work with this or any other intraprocedural constant propagation algorithm. Both the intraprocedural and interprocedural algorithms have three steps: first the program’s dependence graph is constructed. The graph is then rewritten according to a set of rewrite rules. Finally, the rewritten graph is used as an intermediate form for code generation, or, as discussed in Sect. 3.3, to produce optimized source.

3.1. Intraprocedural Constant Propagation

The intraprocedural constant propagation algorithm is essentially a data-flow interpreter. The interpreter propagates constants by continually *firing* (rewriting) vertices and propagating values along edges. Because no run-time information is available, the conditions under which a vertex can be fired *differ* from the rewriting discussed in Sect. 2.2. These differences are given in Rule 1. The remaining rewrite rules push information along the edges of the PDG.

1. The rewriting rules from Sect. 2.2 are modified as follows:
   (a) A vertex representing an action with a side effect (*e.g.*, printing) is never fired and is therefore removed from the graph only if it represents dead code.

(b) Since the rewriting of a while loop increases code size, two limits are placed on the number of such rewritings. These limits are controlled by the constants: *size_increase* and *max_times*. *Size_increase* limits the final code size of the loop (after optimization) to some multiple of the original loop code size. Thus, the number of unrollings is dependent on the effectiveness of optimization on the copy of the loop body. Since optimization may remove the entire body, *max_times* is used to limit the maximum number of times a loop can be unrolled. For example, *max_times* is needed to stop the optimization of "*while* $(n > 0)\ \ n = 1$."

2. A def-order edge $a \longrightarrow_{do(u)} b$ in which a and b have the same control predecessor implies the previously conditional assignment at b has been made unconditional with respect to the definition at b by an application of Rule 5. In this case, only the second definition at b actually reaches u, so the edges $a \longrightarrow_{do(u)} b$ and $a \longrightarrow_f u$ are removed from the graph.

3. If all incoming flow dependence edges for variable v have the same constant value c on them and these constants have high confidence then, the edges are removed and v is replaced by c^v. (If a previous rewrite has replaced v with c^v then all remaining incoming flow dependence edges must have c on them.) Any vertex left with no outgoing edges represents dead code and can be removed.

4. For an assignment vertex labeled "$x = e$", the expression e is evaluated by first using the meet operation to determine a value for each variable v used in e, then computing the value for e. Finally, this value is placed on all outgoing flow dependence edges of the assignment vertex. The confidence of this value is the meet of the confidences of all the input values. (Edges from non-live vertices contain the value of $\top$ with low confidence.)

5. A predicate vertex p, labeled with expression e is evaluated in the same manner as an assignment vertex. The resulting value v may cause certain control successors of p to become live: if v is $\top$ then *none* of the control successors become live; if v is **true** then the **true** successors become live; if v is **false** then the **false** successors become live; finally, if v is $\perp$ then *all* successors become live. Furthermore, if v is **true** or **false** with high confidence then all targets of outgoing control edges from p whose labels match v are made control dependent on the p's control predecessor. All remaining control successors of p along with their (transitive) control successors and p are removed from the graph.

Example. Figure 5(a) shows the graph before the rewriting of the vertex labeled "$a = 2$." The subsequent rewriting of the if vertex produces Fig. 5(b). After rewriting the remaining assignment vertex, the graph cannot be further rewritten because printing represents an operation with a side-effect. (As example of dead-code removal, if the if-predicate vertex had an incoming control edge and thus could not have been re-written, the read vertex would still have been removed as unreachable-code.)

Example. Figure 6 illustrates the propagating of values along data-flow edges. Figure 6(a) shows a sample PDG (in the figure $\perp$ represents a value previously determined to be non-constant). Figure 6(b) shows the effect of Rule (4) on the statement "$x = 2$" from the body of the if statement. Figure 6(c) shows the result of firing the other vertex labeled "$x = 2$" using the assignment-vertex rewrite rule from Sect. 2.2. Finally, Fig. 6(d) shows the result of firing Rule 3: all the incoming edges of the vertex labeled "*print* (2^x)" have the constant 2 with high confidence on them.

```
P ()
{
   a = 2;
   if (a == 0)
      read (b);
   else
      b = a*2;
   print (b);
}
```

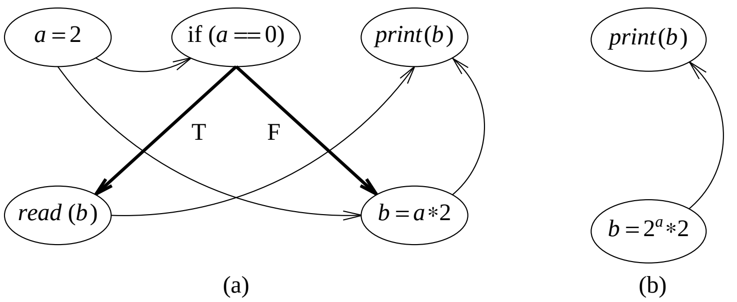

Figure 5. An intraprocedure constant propagation example.

```
P ()
{
   x = 2;
   while (⊥)
   {
      if (⊥)
      {
         x = 2;
      }
   }
   print (x);
}
```

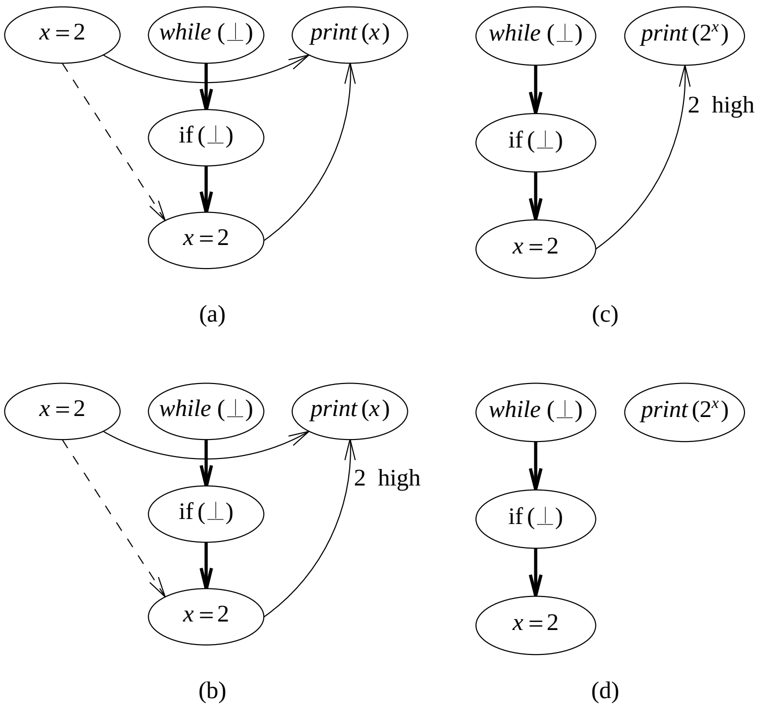

Figure 6. An example that propagates values along data-flow edges.

3.2. Interprocedural Constant Propagation

An interprocedural constant propagation algorithm can be obtained by adding the call-site rewrite rule from Sect. 2.2 to the intraprocedural algorithm developed in Sect. 3.1. However, this algorithm fails to take advantage of opportunities to partially propagate constants to, from, and through procedures when only some of the parameters are constants. For example, Rule 3 (see below) makes use of the fact that the incoming summary edges of an actual-out vertex indicate which actual-parameter's initial values must be constants in order to propagate constants through the called procedure to the actual-out vertex.

The following rules augment the intraprocedural constant propagation rules from Sect. 3.1 to perform interprocedural constant propagation:

1. Actual-in and formal-out vertices are treated as assignment vertices and parameter-in and parameter-out edges are treated as flow dependence edges.
2. Parallel to while-loop unrolling, application of the call-site rewrite rule of Sect. 2.2 is limited by the constants *call_size_increase* and *call_max_times*.
3. If all the incoming summary edges of an actual-out vertex v labeled "$a = x_{out}$" have constants of high confidence on them, then a new assignment vertex labeled "$a = c$" is added to the SDG. Constant c is the value produced for x by the called procedure (see below). The new vertex has an incoming control edge from the control predecessor of the call-site vertex and all outgoing data dependence edges of the actual-out vertex are moved to the new vertex. Finally, the actual-out vertex is marked *useless* (see Rule 6).
4. If the incoming parameter-out edge of an actual-out vertex labeled "$a = x_{out}$" has the constant c of high confidence on it then the actual-out vertex is replaced, as in Rule 3, by a vertex labeled "$a = c$."
5. If all incoming parameter-in edges to formal-in vertex labeled "$x = x_{in}$" have the same constant c on them and these constants have high confidence, then a new vertex labeled "$x = c$" is added. This vertex is control dependent on the called procedure's entry vertex and the flow edges of the formal-in vertex are moved to the new vertex. At all call-sites the corresponding actual-in vertices are marked as useless (see Rule 6).
6. If, at all call-sites on P, the actual-in and actual-out vertices for a parameter are marked as useless, then these vertices are removed from the SDG along with the corresponding formal-in and formal-out vertices. This rule uniformly removes one of P's parameters.
7. A call-site vertex with only useless actual-out vertices is removed. Recall that there are actual-out vertices for input and output streams. Thus, a call with no actual-out vertices returns no values and produces no side-effects.

Computing the value of an actual parameter in Rule (3) requires executing some part of the called procedure. A program that contains the necessary statements to compute this value can be obtained from the SDG using program slicing [15]. In particular, a "*b2*" slice contains the necessary program components. Once this slice has been computed there are two ways of determining the final value of x. First, the constant propagation algorithm could be run on a copy of the the slice. The second, more efficient, option is to produce execute object code from the slice and then execute this object code [7]. (Because of the possibility of non-terminating computations, some limit must be placed on the resources devoted to computing x's value.)

Example. (In the following two examples, Rule 2 is ignored; thus, no in-lining is performed.) For the system shown in Fig. 7, Fig. 8 shows the PDG for procedure P after using Rule 3 to rewrite both actual-out vertices of the first call-site on Q and Rule 7 to subsequently remove the call-site. The actual-out vertex for y, for example, was the target of two summary edges, which had the constants 2 and 8 on them. Rule 3 replaced this actual-out vertex with an assignment vertex labeled "$y = 12$" (12 is the value computed by Q using the input values 2 and 8). The final SDG and its corresponding program (see Sect. 3.3) are shown in Fig. 9.

```
main()            P(x, y)            Q(a, b)
{                 {                  {
   i = 2;            Q(x, y);           a = a * 2;
   j = 8;            x = x – 2;         b = b + a;
   P(i, j);          Q(x, y);        }
   print(j);         print (x);
}                 }
```

Figure 7. An interprocedural constant propagation example.

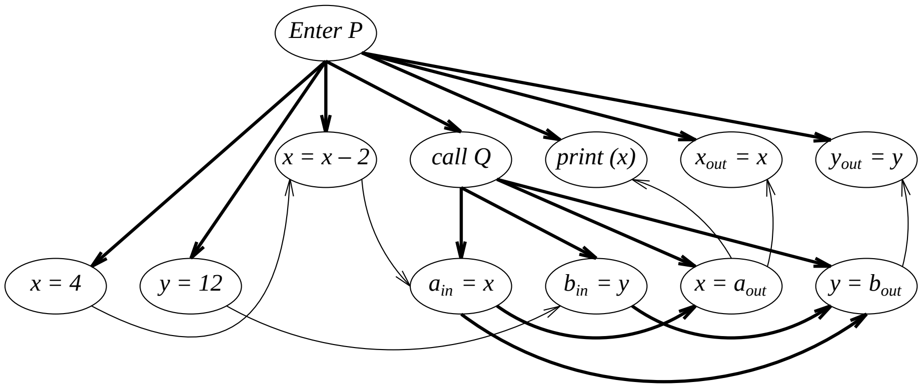

Figure 8. The PDG for procedure *P* of Fig. 7 after applying Rules 3 and 7 to the first call on *Q*.

Example. In the preceding example constant propagation removed procedure *Q* entirely. This example illustrates the removing of one of *Q*'s parameters. Consider the optimization of the program in Fig. 7 if the initial value of *j* is non-constant (*i.e.*, replace "$j = 8$" with "*read* (*j*)"). In this case, the call-sites on procedure *Q* cannot be removed from *P*; however, propagating constants *through* call-sites, the incoming parameter-in edges of the formal-in vertex for parameter *x* in procedure *Q* both have the constant 2 with high confidence on them. Rule 5 rewrites the graph by replacing this formal-in vertex with the assignment vertex "$x = 2$." This produces a version of *P* without actual parameter *x* and a version of *Q* without formal parameter *a*. The resulting optimized version of the system is shown below:

```
main()            P(y)               Q(b)
{                 {                  {
   read(j);          Q(y);              b = b + 4;
   P(j);             Q(y);           }
   print(j);         print(4);
}                 }
```

3.3. Producing Source Code After Optimization

This section discusses how optimized source code is produced from the optimized SDG. Three uses of this source code include illustrating the effects of optimization to students and others unfamiliar with optimization, debugging optimized code, and producing optimized object code (although using the SDG as an intermediate form and directly producing optimized object code would be more efficient). For example, in debugging, making a programmer aware of optimization's effect allows the programmer to better understand the output from a debugger when debugging optimized code. This avoids the need to only debug unoptimized code or use a debugger that attempts the difficult and often impossible task of reversing optimization in an attempt to recover the state of the unoptimized program [10].

Producing source code from the optimized SDG is a two step process. Step one "backs up" optimization in order to allow step two to reconstitute a system from the SDG. The reasons this backup is necessary is that the data-flow interpretation of the SDG allows partial evaluation of statements. For some statement types (*e.g.*, call statements) partially evaluated statements cannot be part of a source program.

The backing up of optimization is necessary in two cases. First, when a constant has been propagated to an actual-in vertex but the actual-out vertex has not been removed from the graph. This is illustrated in Fig. 10(a) where the actual-in vertex labeled "$x_{in} = 1^a$" requires the call statement "*call* $P(1)$", while the actual-out vertex labeled "$a = x_{out}$" requires the call statement "*call* $P(a)$." The second case occurs

```
main()
{
   P();
   print(16);
}

P()
{
   print (4);
}
```

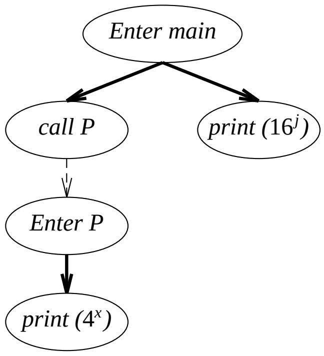

Figure 9. The result of optimizing the program from Fig. 7.

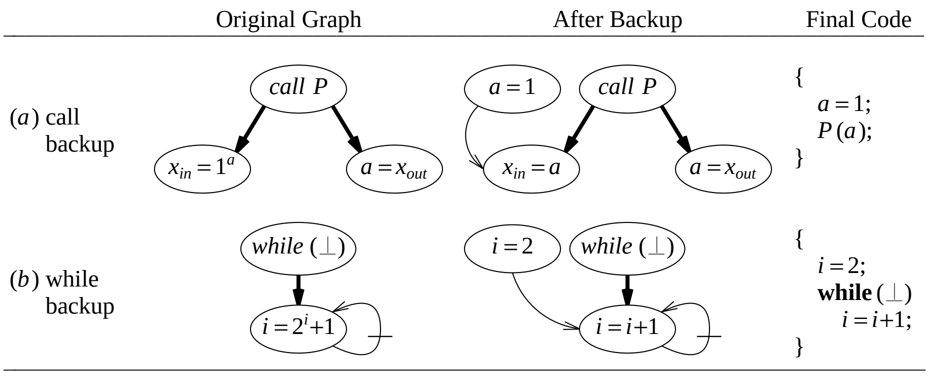

Figure 10. Examples of the two cases in which optimization backup is necessary.

when c^v has replaced v but there remains a reaching definition of v. This is illustrated in Fig. 10(b) where the use of i in the while loop shown in Fig. 10(b) has an incoming loop-carried flow dependence edge.

In both cases, backup of the optimization that replaced v with c^v at vertex x involves three steps: (1) potentially adding a new assignment vertex, labeled "$v = c$," (2) adding a flow dependence edge from this new vertex to x, and (3) replacing c^v with v in the label of x. Step (1) adds an assignment vertex only if a previous backup has not already added an assignment to the same variable at the same *location*. The *location* of this vertex is that of an assignment statement not nested within any other control structure[2] and immediately before the statement representing x and all statements representing vertices that have flow dependence edges to x.

Example. Figure 10(a) shows a partial SDG before and after applying the call-site backup step and (part of) the final reconstituted program. Figure 10(b) shows a partial SDG before and after applying the second backup step along with (part of) the final reconstituted program.

After all necessary backups, the final step in the constant propagation algorithm reconstitutes optimized source code from the optimized SDG. The difficult problem in reconstitution is finding a total order for the vertices in each *region* of each PDG (roughly put, a region is the body of an if statement or while loop—see [14]). In general the reconstitution problem is NP-complete [14]; however, for an SDG optimized using the rules in this paper, reconstitution requires only $O(n \log n)$ time, where n is the number of vertices in the optimized SDG (see Sect. 4). The NP-complete aspect of the problem is avoided by using order information from the unoptimized program.

4. Related Work

This section compares the SDG based constant propagation algorithm to previous constant propagation algorithms discussed by Wegman and Zadeck [19] and Callahan, Cooper, Kennedy, Torczon, and Grove [8, 13].

Wegman and Zadeck [19] discuss an intraprocedural constant propagation algorithm that finds all simple constants—constant that can be found assuming no information about the direction of branches and storing only one value per variable—and one that finds all *conditional constants*—those constants that can be determined by including tests for constants in the conditional branches. (Wegman and Zadeck actually discuss two algorithms of each type, which differ only in their complexity.) These algorithms are based on the static single assignment (SSA) graph [11], which augments a standard control-flow graph with special ϕ vertices—assignments of the form "$x = \phi(x, x)$." These vertices are placed at control join-points whenever two definitions of x reach the join-point. The result is that every use of x (other than at a ϕ vertex) is reached by a single definition. A similar effect can be obtained using the SDG and valve nodes [9]. Reaching definitions are captured in the SSA graph by SSA-edges, which are similar to data dependence edges in an SDG. For some programs, including ϕ vertices and SSA-edges can reduce the number of edges representing reaching definitions when compared to the SDG for the same program. This is done at the expense of introducing additional vertices.

---

[2] For c^v to replace v a rewrite rule from Sect. 2.2 must have been used. Therefore, no incoming control edge should be present for the vertex (*i.e.*, it should not be nested within any other control structure).

The algorithm discussed in Sect. 3.1, which finds all conditional constants, is essentially equivalent to the intraprocedural constant propagation algorithms given by Wegman and Zadeck. For interprocedural constant propagation, which Wegman and Zadeck also discuss, the algorithm developed in Sect. 3.2 has some advantages. These include, the use of summary edges in Interprocedural Rule (3) to determine that a procedure returns a constant value at a particular call-site even through other call-sites on the same procedure may pass non-constant values to the procedure. Rewriting an actual-out vertex using Rule (3) can “decouple” the body of the called procedure from the computation of the return value for the actual parameter. For example, consider a procedure with two parameters, a counter and some other variable. If the counter’s return value can be determined to be constant using Rule (3) then the computation of the called procedure that involves the other variable can proceed in parallel with subsequent computations of the calling procedure that involve the counter.

Callahan et. al. discuss several interprocedural constant propagation algorithms centered around the constructions of *jump* and *return* functions. These functions describe how constants flow from procedure to procedure. The summary edges in the SDG and program slices taken with respect to actual-out vertices can be used to compute jump and return functions analogous to those discussed in [8]. A comparison of this approach to those suggested in [8] is the topic of future research.

5. Conclusion

Using dependence graphs and a data-flow model provides an efficient algorithm for interprocedural constant propagation. In many cases graph transformations are easier to visualize, create, and understand than transformations involving sets of variables and program text. One reason for this is that dependence graphs allow the program components involved in the optimization to be more easily isolated from other program components regardless of their textual location in the source code.

A key feature of the SDG is its inclusion of summary edges. These edges allow constants to propagate through called procedures, which increases the number of constants found by the algorithm. Summary edges have also proved useful in other operations involving the SDG, for example the two step program slicing algorithm presented in [15].

Finally, the adequacy of the SDG for representing programs [5] and the use of semantics preserving graph transformations [18] provides safety. Because each rewriting step is guaranteed to preserve the semantics of the original program, the program produced as a result of constant propagation is guaranteed to preserve the semantics of the original unoptimized program. The algorithm inherits its correctness proof from the correctness of the semantics based transformations. Thus, it successfully combines the rather theoretical results involving the adequacy and semantics of the SDG with the production of a practical algorithm. The result is a provably correct algorithm for interprocedural constant propagation.

References

1. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).

2. Allen, R. and Kennedy, K., “Automatic translation of FORTRAN programs to vector form,” *ACM Transactions on Programming Languages and Systems* **9**(4) pp. 491-542 (October 1987).

3. Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages,* (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).
4. Barth, J.M., "A practical interprocedural data flow analysis algorithm," *Commun. of the ACM* **21**(9) pp. 724-736 (September 1978).
5. Binkley, D., Horwitz, S., and Reps, T., "The multi-procedure equivalence theorem," TR-890, Computer Sciences Department, University of Wisconsin, Madison, WI (November 1989).
6. Binkley, D., "Slicing in the presence of parameter aliasing," pp. 261-268 in *1993 Software Engineering Research Forum,* (Orlando, FL, Nov, 1993), (1993).
7. Binkley, D., "Precise executable interprocedural slices," *to appear in ACM Letters on Programming Languages and Systems,* (1994).
8. Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L., "Interprocedural constant propagation," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986), *ACM SIGPLAN Notices* **21**(7) pp. 152-161 (July 1986).
9. Cartwright, R. and Felleisen, M., "The semantics of program dependence," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices,* (1989).
10. Chambers, C., Hölzle, U., and Ungar, D., "Debugging optimized code with dynamic deoptimization," *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation,* (San Francisco, CA, June 17-19, 1992), *ACM SIGPLAN Notices* **27**(7) pp. 32-42 (June 1992).
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
12. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).
13. Grove, D. and Torczon, L., "Interprocedural constant propagation: A study of jump function implementations," *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation,* (Albuquerque, NM, June 23-25, 1993), pp. 90-99 ACM, (1993).
14. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).
15. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
16. Kildall, G.A., "A unified approach to global program optimization," pp. 194-206 in *Conference Record of the First ACM Symposium on Principles of Programming Languages,* (October 1973), ACM, New York, NY (1973).
17. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).
18. Selke, R, "A graph semantics for program dependence graphs," pp. 12-24 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, January 11-13, 1989), ACM, New York, NY (1989).
19. Wegman, M.N. and Zadeck, F.K., "Constant propagation and conditional branches," *ACM Transactions on Programming Languages and Systems* **13**(2) pp. 181-210 ACM, (April, 1991).