Journal of Universal Computer Science, vol. 9, no. 8 (2003), 829-850
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © *J.UCS*

Constant Propagation on Predicated Code[1]

Jens Knoop[2]
(Vienna University of Technology, Austria
knoop@complang.tuwien.ac.at)

Oliver Rüthing
(University of Dortmund, Germany
ruething@ls5.cs.uni-dortmund.de)

Abstract: We present a new *constant propagation* (*CP*) algorithm for *predicated code*, for which classical CP-techniques are inadequate. The new algorithm works for arbitrary control flow, detects constancy of terms, whose operands are not constant themselves, and is *optimal* for acyclic code such as *hyperblocks*, the central "compilation units" for instruction scheduling of predicated code. The new algorithm operates on the *predicated value graph*, an extension of the well-known *value graph* of Alpern et al. [Alpern et al., 1988], which is tailored for predicated code and constructed on top of the *predicate-sensitive* SSA-form, which has been introduced by Carter et al. [Carter et al., 1999]. As an additional benefit, the new algorithm identifies off-predicated instructions in predicated code. They can simply be eliminated thereby further increasing the performance and simplifying later compilation phases such as instruction scheduling.
Key Words: Constant propagation, predicated code, IA-64, predicated SSA-form, predicated value graph, data-flow analysis, optimization.
Category: D.3.4, C.1.0

1 Motivation

Constant propagation (*CP*) aims at replacing term occurrences which always yield the same constant value at run-time by this value (cf. [Aho et al., 1977; Aho et al., 1985; Kennedy, 1981; Muchnick, 1997]). It belongs to the most important and widespread optimizations in contemporary compilers. However, current CP-techniques are inadequate to handle *predicated code*, i.e., code where instructions are guarded by a 1-bit register, which dynamically controls whether the effect of an instruction is committed or nullified. In order not to corrupt the program semantics, they have to be overly conservative. For illustration consider Figure 1, where statements are written using the syntax of the IA-64 for predication [Intel Corp., 1999; Dulong, 1998].

[1]A preliminary version of this article has been presented at the 7th Brazilian Symposium on Programming Languages (SBLP 2003) (Ouro Preto, MG, Brazil, May 28 - 30, 2003) [Knoop and Rüthing, 2003].
[2]Part of the work was done, while the author was at The FernUniversität / University of Hagen.

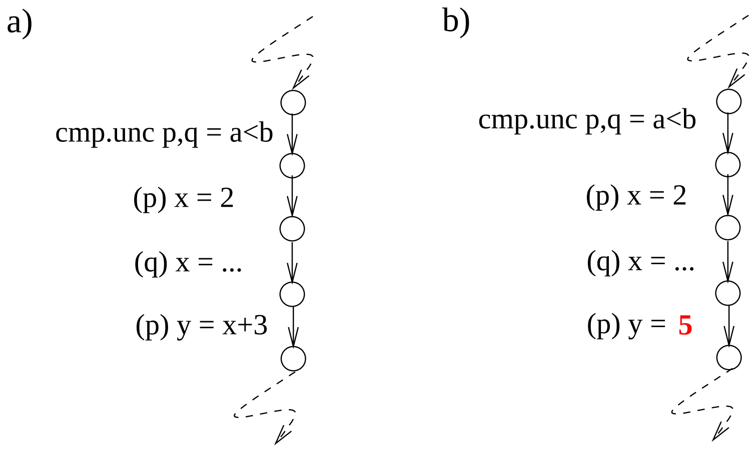

Figure 1: *Illustrating the essence of CP on predicated code.*

In this example, the predicates p and q guarding the execution of the statements will always have different truth values according to the semantics of the IA-64 machine model (cf. Section 2). The classical CP-techniques, however, are not prepared to incorporate such information. They do not realize that the assignment to y is not committed in case the second assignment to x is committed. Hence, they have to conservatively assume that variable x can be modified between the first assignment to x and its use site in the assignment to y. Consequently, they fail to achieve the desired optimization of Figure 1(b). Predicated code, however, is getting more and more common due to the emerging dissemination of architectures such as the IA-64 (cf. [Intel Corp., 1999; Dulong, 1998]). This demands for new predicate-sensitive techniques. Though two generic frameworks for bit-vector problems have recently been proposed [Hu, 2000; August, 2000], we are not aware of any predicate-sensitive technique applicable to constant propagation.

1.1 Predicated Code and Hyperblocks

An important advantage of predicated code is that it allows a compiler to eliminate branches by melting both paths of a branch to a single path. Even more, due to predication several smaller basic blocks can be combined to one larger *hyperblock* (cf. [Mahlke et al., 1992]). Following [Mahlke et al., 1992], a hyperblock is a predicated portion of code consisting of a group of basic blocks with one entry point and possibly multiple exit points. In predicated form, a hyperblock reduces to a straightline sequence of code. Fundamental for the construction of a hyperblock is a transformation known as *if-conversion* (cf. [Allen et al., 1983]). In the context of a hyperblock, if-conversion eliminates branches where both targets are in the hyperblock, by converting them to predicated instructions. As a consequence, there are no cyclic control-flow or data-flow dependences within the hyperblock, since remaining branches have targets outside the hyperblock. For architectures such as the IA-64, the importance of switching by means of predicated execution from (smaller) basic blocks to (larger) hyperblocks is that it increases *instruction level parallelism* (*ILP*) and removes *hard-to-predict* branches. As a consequence, basic blocks to be included in a hyperblock are carefully selected by compilers for predicated code. Typically, this is done on the basis of sophisticated program profiling exploiting information such as execution frequency, basic block size, operation latencies, and other characteristics (cf. [August et al., 1997; Allen et al., 1983; Carter et al., 1999; Park and Schlansker, 1991; Warter et al., 1993]). In fact, hyperblocks are most important "compilation units" for compilers handling predicated code, particularly for scheduling.

In this scenario, CP has an important impact on the quality of instruction schedules. Firstly, replacing arbitrary loads by more efficient loads of constants reduces the risk of pipeline stalls due to cache misses. Secondly, CP easily identifies *dead instructions* whose guarding predicates are equivalent to false (this is also called "off-predicated)." They can simply be removed from the code enhancing the performance further. Moreover, this clearance makes subsequent compilation phases more effective, in particular, the scheduling process of the instructions of the corresponding hyperblock (cf. [Carter et al., 1999]).

In comparison to ordinary code, hyperblocks show two important characteristics, which are central for the design of our CP-algorithm. First, they originate from acyclic control flow. This enables optimal solutions detecting all constants (cf. [Steffen and Knoop, 1989; Steffen and Knoop, 1991]). Second, they are of moderate size. In addition, this size is under the control of the compiler. Hence, the size of hyperblocks can be considered a (usually small) constant in that of the overall program. Together, these characteristics recommend hyperblocks as subjects also of in the worst case expensive, but powerful and aggressive optimizations. Optimal CP is one such an example.

1.2 The New CP-Algorithm

In this article, we propose a new algorithm for CP, which is tailored for predicated code. Conceptually, this algorithm resembles the CP-algorithm of [Knoop and Rüthing, 2000]. The algorithm of [Knoop and Rüthing, 2000] is unique in that it performs CP on the *value graph* (cf. [Alpern et al., 1988]) of a program, which is constructed on top of the *static single assignment* (*SSA*) form (cf. [Aycock and Horspool, 2000; Cytron et al., 1989; Cytron et al., 1991]) of a program; an intermediate program representation, which has been used with much success in optimizing compilation (cf. [Chow et al., 1997; Kennedy et al., 1999; Kennedy et al., 1998; Rosen et al., 1988]). As shown in [Knoop and Rüthing, 2000], the value graph is not only most adequate for CP, the resulting algorithm is even

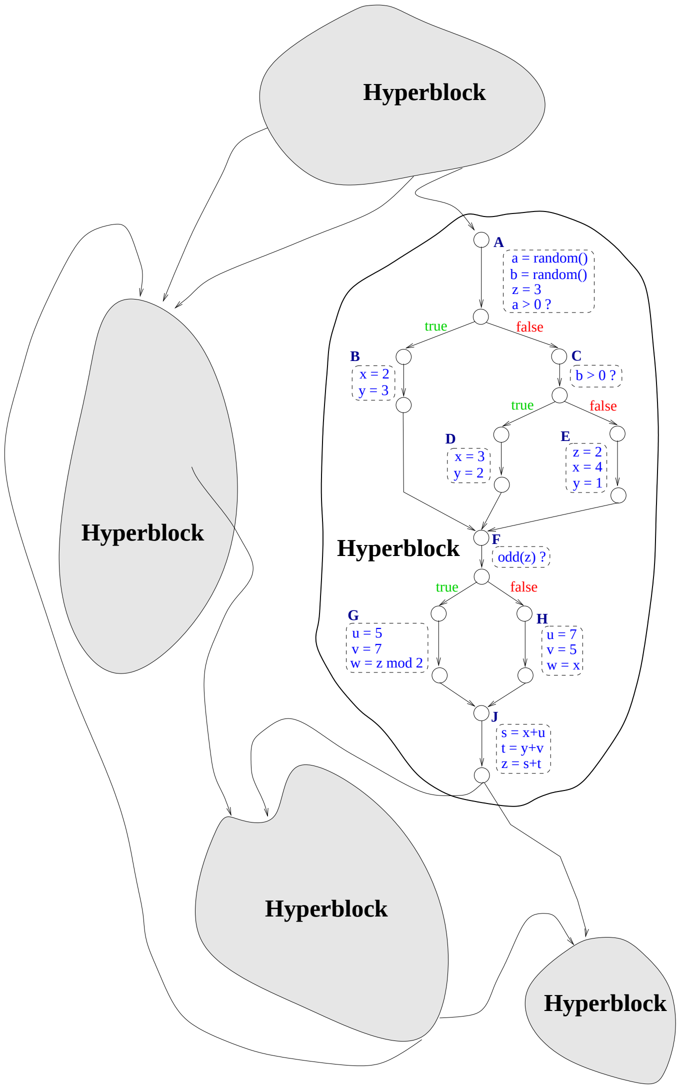

Figure 2: *The running example.*

more powerful than the state-of-the-art algorithms used in contemporary compilers for CP computing *simple constants* (cf. [Kildall, 1973; Kam and Ullman, 1977]). The algorithm, which we are going to present here, consists of a local component for *predicated hyperblocks*, and a global component propagating the locally gained information on constant terms throughout the program. The local component of this algorithm is based on an extension of the value graph being tailored for predicated code: the *predicated value graph* (*PVG*). The PVG of a program is constructed on top of the *predicated SSA* (*PSSA*) form for predicated hyperblocks, which has been proposed by Carter et al. [Carter et al., 1999]. Moreover, the local component is parameterized in the treatment of predicates. Already the basic version of the local component detects all constants in a hyperblock, which do not depend on the predication. Intuitively, this corresponds to a nondeterministic interpretation of the branching structure of the underlying hyperblock. The corresponding full version takes branching into account, too. In a hyperblock, it detects all constants with respect to the deterministic interpretation of branches, whose outcome depends only on the context the hyperblock is analyzed in. It is *trace-precise* as we call this property.

1.3 Illustrating the Power of the New CP-Algorithm

The power of the new algorithm is illustrated by means of the example of Figure 2 and Figure 3. In the article, we focus on the effect of our algorithm on a specific hyperblock of the program of Figure 2, where the shadowed boxes are assumed to represent hyperblocks, too. The basic version of our algorithm detects already the constancy of z at the end of this program fragment of Figure 3(a) in block J as it does not depend on the particular branching structure. The full version of our algorithm detects additionally the constancy of w at the end of the blocks G and H. Note that the outcome of the branching condition $odd(z)$ in block F cannot statically be determined – neither the true-branch nor the false-branch originating there are dead code, which, e.g., prevents the algorithm of [Wegman and Zadeck, 1985; Wegman and Zadeck, 1991] to detect the constancy of w in the blocks G and H. In fact, detecting this requires a "trace-precise" analysis. In this example, a successful analysis has to prove that only the occurrence of variable z assigned to in block A reaches the use site in block G (but not the one assigned to in block E), while only the occurrence of x assigned to in block E reaches the use site in block H (but not the occurrences assigned to in the blocks B and D). In fact, none of the occurrences of w is a conditional constant in the sense of [Wegman and Zadeck, 1985; Wegman and Zadeck, 1991]. Hence, their constancy will not be detected by the algorithm of [Wegman and Zadeck, 1985; Wegman and Zadeck, 1991].[3] Likewise, the occurrence of z in block J is neither a simple constant, nor a constant in the sense of Kam and Ullman's heuristic extension of simple constants of [Kam and Ullman, 1977]. Hence, the constancy of z will not be detected by the standard CP-algorithms used in contemporary optimizing compilers (though it would be detected using the algorithm of [Steffen and Knoop, 1989; Steffen and Knoop, 1991]).

[3] It should be noted that the algorithm of Wegman and Zadeck works for arbitrary control flow. However, it is not clear, how to adopt this algorithm to predicated code.

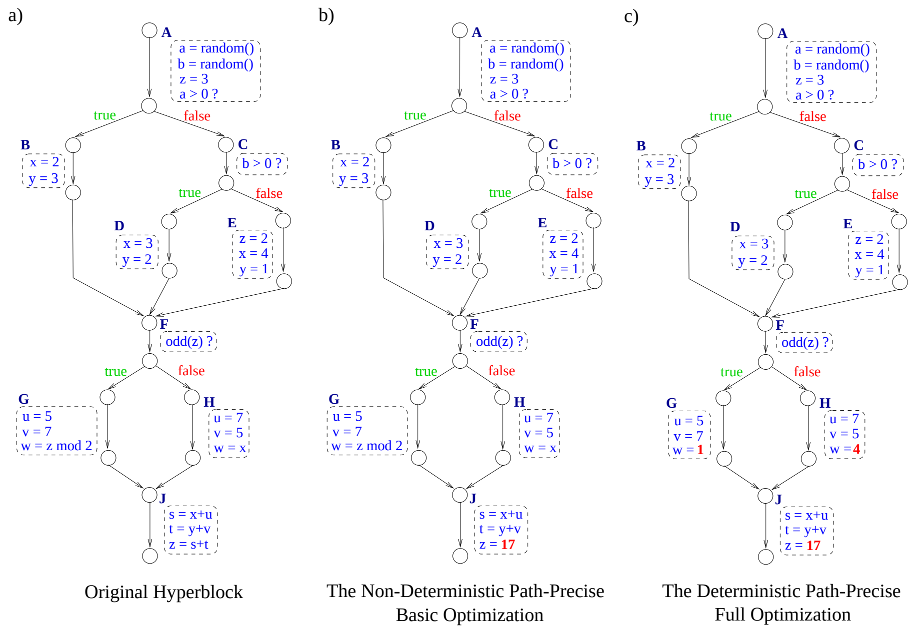

Figure 3: *The running example: Illustrating the unique power of path-precise constant propagation.*

1.4 Structure of the Article

In Section 2 we present our preliminaries, before recalling the PSSA-form of [Carter et al., 1999] in Section 3. Fundamental is Section 4, in which we illustrate our new algorithm for CP for predicated code. In particular, we oppose its local component to the CP-algorithm of [Knoop and Rüthing, 2000]. Subsequently, we present the main results applying to the new approach in Section 5, and discuss benefits-for-free evolving from it. In Section 6 , finally, we present our conclusions.

2 Preliminaries

Predication. We assume the architectural support of general predicated execution as modelled in the IA-64 architecture [Intel Corp., 1999; Dulong, 1998], in which the execution of an instruction can be guarded by a qualifying predicate. In particular, there is a special predicate `p0`, which cannot be written and whose value is always equal to true.[4] Predicates can be set by means of compare instructions of the form: `(qp) cmp.ctype p1, p2 = r2, r3 (crel)`, where `p1` and `p2` are two target predicates, and `qp` is the qualifying predicate. The two

[4]Such a predicate is featured by several predicated architectures including the IA-64.

source operands, `r2` and `r3`, are compared based on the relation specified by `crel`. There is a number of comparison types, in the article, however, we will only use the type `unc` which is a shorthand for `unconditional` (see e.g. [Knoop et al., 2000] for other types). If the qualifying predicate holds then p_1 and p_2 are set exclusively true, otherwise p_1 and p_2 are set to false.

To explore predication, a compiler usually applies a technique called *if-conversion* (cf. [Allen et al., 1983]). This technique eliminates branch instructions and converts affected instructions to appropriate predicated form. Table 1 summarizes the result of the if-conversion transformation for our running example of Figure 3(a). The left column of this table shows the source code of this example. The right column shows the code after if-conversion. In the following we focus on programs representing hyperblocks, which are given in if-converted form.

```
Original Hyperblock | After if-Conversion
=================== | ===================
                    |
begin               | begin
 a = random();      |  (p0) a = random();
 b = random();      |  (p0) b = random();
 z = 3;             |  (p0) z = 3;
 if a>0 then        |  (p0) cmp.unc B,C (a>0);
   x = 2;           |  (B)  x = 2;
   y = 3            |  (B)  y = 3;
 elsif b>0 then     |  (C)  cmp.unc D,E (b>0);
   x = 3;           |  (D)  x = 3;
   y = 2            |  (D)  y = 2;
 else               |
   z = 2;           |  (E)  z = 2;
   x = 4;           |  (E)  x = 4;
   y = 1 fi;        |  (E)  y = 1;
 if odd(z) then     |  (p0) cmp.unc G,H (odd(z));
   u = 5;           |  (G)  u = 5;
   v = 7;           |  (G)  v = 7;
   w = z mod 2      |  (G)  w = z mod 2;
 else               |
   u = 7;           |  (H)  u = 7;
   v = 5;           |  (H)  v = 5;
   w = x fi;        |  (H)  w = x;
 s = x+u;           |  (p0) s = x+u;
 t = y+v;           |  (p0) t = y+v;
 z = s+t            |  (p0) z = s+t
end.                | end.
```

Table 1: *Source code and if-converted code of the running example of Figure 3(a).*

Semantics of Terms. We consider terms $t \in \mathbf{T}$ which we assume to be inductively built from variables $v \in \mathbf{V}$, constants $c \in \mathbf{C}$, and operators $op \in \mathbf{Op}$ of arity $r \geq 1$. The *semantics* of terms is induced by an *interpretation* $I = (\,\mathbf{D}' \cup \{\perp, \top\}, I_0\,)$, where $\mathbf{D}'$ denotes a non-empty data domain, $\perp$ and $\top$ two new data not in $\mathbf{D}'$, and I_0 a function mapping every constant $c \in \mathbf{C}$ to a datum $I_0(c) \in \mathbf{D}'$, and every r–ary operator $op \in \mathbf{Op}$ to a total function $I_0(op) : \mathbf{D}^r \rightarrow \mathbf{D}$, $\mathbf{D} =_{df} \mathbf{D}' \cup \{\perp, \top\}$ being strict in $\perp$ and $\top$ with $\perp$ prioritized over $\top$ (i.e., $I_0(op)(d_1, \ldots, d_r) = \bot$, whenever there is a j, $1 \le j \le r$, with $d_j = \bot$, and $I_0(op)(d_1, \ldots, d_r) = \top$, whenever there is no j, $1 \le j \le r$, with $d_j = \bot$, but a j, $1 \le j \le r$, with $d_j = \top$). $\Sigma =_{df} \{ \sigma \mid \sigma : \mathbf{V} \to \mathbf{D} \}$ denotes the set of *states*, and $\sigma_\bot$ the distinct *start state* assigning $\bot$ to all variables $v \in \mathbf{V}$. This choice reflects that we do not assume anything about the context of a program being optimized. The *semantics* of a term $t \in \mathbf{T}$ is then given by the inductively defined *evaluation* function $\mathcal{E} : \mathbf{T} \to (\Sigma \to \mathbf{D})$:

$$\forall\, t \in \mathbf{T}\; \forall\, \sigma \in \Sigma.\; \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t = x \in \mathbf{V} \\ I_0(c) & \text{if } t = c \in \mathbf{C} \\ I_0(op)(\mathcal{E}(t_1)(\sigma), \ldots, \mathcal{E}(t_r)(\sigma)) & \text{if } t = op(t_1, \ldots, t_r) \end{cases}$$

Note that the above definitions can naturally be extended to predicate registers and to terms composed of relators like $<$, $=$, $>$, etc. For both, the semantic domain is given by the set of Boolean truth values $\mathcal{B} =_{df} \{ true, false \}$. This extension allows us to take the outcome of branching instructions into account, too. For convenience we assume $\mathcal{B} \subseteq \mathbf{D}' \subseteq \mathbf{T}$, i.e., we identify the set of data $\mathbf{D}'$ with the set of constants $\mathbf{C}$.

3 Predicated Static Single Assignment Form

Intuitively, the *predicated static single assignment* (*PSSA*) form of a program proposed by Carter et al. [Carter et al., 1999] is a predicate-sensitive implementation of the well-known *static single assignment* (*SSA*) form (cf. [Cytron et al., 1989; Cytron et al., 1991; Aycock and Horspool, 2000]), which is tailored for hyperblocks.

In non-predicated code, the essence of SSA is to reveal use-definition chains of variables by replacing the variables of the original program by new renamed versions such that every variable has a unique definition point. At join points of the control flow pseudo-assignments $x_k := \phi(x_{i1}, \ldots, x_{ik})$ are introduced meaning that x_k gets the value of x_{ij}, if the join node is entered via the jth ingoing edge. This is illustrated in Figure 4(b). It shows the SSA-representation of the program of Figure 4(a), where ϕ-operators are indexed by their corresponding join node.

The PSSA form presented in [Carter et al., 1999] is a *predicate-sensitive* extension of the SSA-form for predicated hyperblocks, whose construction rests on the following substantial design decisions. First, ϕ-statements are resolved in PSSA-form. Reflected in terms of SSA, this would essentially mean to transform the program of part (a) of Figure 4 into the one of part (c) rather than the one of part (b). This means, instead of ϕ-functions path-specific *duplicates* of statements are introduced. Second, for each basic block of a hyperblock, a so-called *full-path predicate* (*FPP*) is introduced in PSSA. Intuitively, an FPP

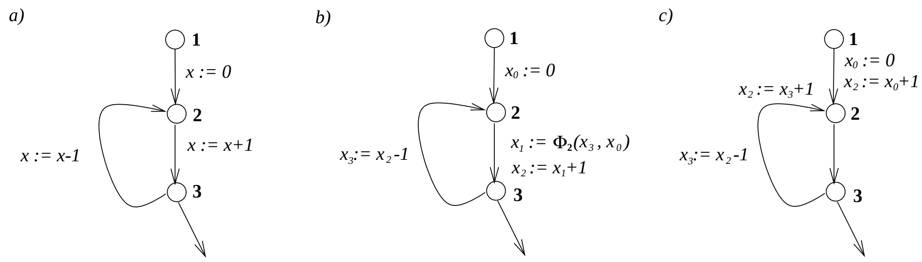

Figure 4: *SSA-form and value graph: (a) the original program, (b) the program after SSA-conversion, and (c) the SSA-form with resolved ϕ-functions.*

specifies the unique path along which an operation is valid for execution. Each of the FPP-definitions has the appropriate operand versions for its path, and is guarded by the FPP that defined the path prior to reaching the new block. This enables PSSA to provide correct guarding predicates for the duplicate statements previously described. Third, *block predicates* are introduced which are defined as the union of the FPPs associated with the paths that reach the block. In the PSSA-form the block predicates are defined by means of `OR`-statements.

The motivation underlying these design decisions is to enable the PSSA-form of a program to encode information about each path existing in the hyperblock. Hence, it is exponential in the size of a hyperblock. It is thus important to remember that the size of a hyperblock is compiler-controlled, which in practice can be considered a (reasonably sized) constant in the overall program size.

The construction of the PSSA form of a program proceeds by successively processing the instructions of the if-converted hyperblock. This process relies on two subprocedures, called *Control PSSA* and *Normal PSSA* (cf. [Carter et al., 1999]). They are invoked depending on the instruction encountered in the if-converted code. If this is a normal instruction, i.e., different from a predicate register defining `cmp`-instruction, Normal PSSA is invoked. This procedure proceeds essentially as its counterpart for ordinary SSA-conversion except for the case that the instruction is part of a join block. In this case multiple versions of the operands may be alive. In ordinary SSA, this is handled by means of ϕ-functions. In PSSA, however, these instructions are duplicated for each path leading to the join, and the correct operand versions for each path will be used. The intuition behind this duplication is illustrated in Figure 4(c). The duplicates are guarded by the FPPs associated with the path along which the operands are defined. It is worth noting that all these duplicates are predicated on disjoint predicates. Hence, at run-time only one of them can possibly be true, and be committed. For illustration consider the multiple guarded assignments to $s1$ and $t1$ at the bottom of the right column of Table 2 which are all guarded by distinct FPPs.

When a `cmp`-instruction is encountered, Control PSSA is invoked. The `cmp`-

instruction is then replaced by one or more new `cmp`-instructions, where each of them is associated with a particular path leading to that block.

In [Carter et al., 1999], the process of PSSA-conversion is described in full detail. In particular, it is exposed how to avoid the introduction of unnecessary duplicates, i.e., of "duplicated" duplicates, and thus an unnecessary growth of the PSSA-form. In this article, we do not make use of this heuristics in order to keep the technical development of Section 4 as simple as possible. Table 2 shows the PSSA-form of the if-converted code of Table 1.[5]

4 Constant Propagation on Predicated Code

In this section, we present the new CP-algorithm. It operates on a new data structure called the *predicated value graph* (PVG), which is constructed on top of the PSSA-form of a program. For the convenience of the reader, we recall the essence of the underlying CP-algorithm for non-predicated code of [Knoop and Rüthing, 2000] first. This algorithm works on the value graph, which is constructed on top of te SSA-form of a program.

4.1 Background

Basically, the value graph of a program represents the value transfer of SSA variables along the control flow of the program. Following the definition in [Muchnick, 1997], the value graph is a labelled directed graph, where

- the nodes correspond to occurrences of nontrivial assignments, i.e., assignments whose right-hand sides contain at least one operator, and to occurrences of constants in the program. Every node is labelled with the corresponding constant or operator symbol, and additionally with the set of variables whose value is generated by the corresponding constant or assignment. The generating assignment of the left-hand side variable of a trivial assignment $x := y$ is defined as the generating assignment of y. An operator node is always annotated with the left-hand side variable of its corresponding assignment, and the left-hand side variable of a trivial assignment $x := c$ is attached to the annotation of the corresponding node associated with c. By convention, constant and operator labels are written inside the circle visualizing the node, and variable annotations outside.
- Directed edges point to the operands of the right-hand side expression associated with the node. Here, we make the implicit assumption that edges are

[5] Instructions marked by "[$*$]" can be eliminated as a side-effect of our approach. They are off-predicated as detected by our CP-algorithm. Similarly, instructions marked by "[$-$]" can be simplified as certain predicates occurring are detected to be false. The resulting program is shown in Table 3. For (algorithmic) details see Section 4.

```
        begin
          (p0)      A = OR(TRUE);
          (A)       a1 = random();
          (A)       b1 = random();
          (A)       z1 = 3;
          (A)       cmp.unc BA,CA (a1>0);
          (p0)      B = OR(BA);
          (p0)      C = OR(CA);
          (B)       x1 = 2;
          (B)       y1 = 3;
          (C)       cmp.unc DCA,ECA (b1>0);
          (p0)      D = OR(DCA);
          (p0)      E = OR(ECA);
          (D)       x2 = 3;
          (D)       y2 = 2;
          (E)       z2 = 2;
          (E)       x3 = 4;
          (E)       y3 = 1;
          (BA)      FBA = OR(TRUE);
          (DCA)     FDCA = OR(TRUE);
          (ECA)     FECA = OR(TRUE);
          (p0)      F = OR(FBA,FDCA,FECA);
          (FBA)     cmp.unc GFBA,HFBA (odd(z1));
          (FDCA)    cmp.unc GFDCA,HFDCA (odd(z1));
          (FECA)    cmp.unc GFECA,HFECA (odd(z2));
    [-]   (p0)      G = OR(GFBA,GFDCA,GFECA);
    [-]   (p0)      H = OR(HFBA,HFDCA,HFECA);
          (GFBA)    w1 = z1 mod 2;
          (GFDCA)   w1 = z1 mod 2;
    [*]   (GFECA)   w1 = z2 mod 2;
          (G)       u1 = 5;
          (G)       v1 = 7;
    [*]   (HFBA)    w2 = x1;
    [*]   (HFDCA)   w2 = x2;
          (HFECA)   w2 = x3;
          (H)       u2 = 7;
          (H)       v2 = 5;
          (GFBA)    JGFBA = OR(TRUE);
          (GFDCA)   JGFDCA = OR(TRUE);
    [*]   (GFECA)   JGFECA = OR(TRUE);
    [*]   (HFBA)    JHFBA = OR(TRUE);
    [*]   (HFDCA)   JHFDCA = OR(TRUE);
          (HFECA)   JHFECA = OR(TRUE);
    [-]   (p0)      J = OR(JGFBA,JGFDCA, JGFECA,JHFBA, JHFDCA,JHFECA);
          (JGFBA)   s1 = x1+u1;
          (JGFBA)   t1 = y1+v1;
    [*]   (JGFDCA)  s1 = x2+u1;
    [*]   (JGFDCA)  t1 = y2+v1;
          (JGFECA)  s1 = x3+u1;
          (JGFECA)  t1 = y3+v1;
    [*]   (JHFBA)   s1 = x1+u2;
    [*]   (JHFBA)   t1 = y1+v2;
    [*]   (JHFDCA)  s1 = x2+u2;
    [*]   (JHFDCA)  t1 = y2+v2;
    [-]   (JHFECA)  s1 = x3+u2;
          (JHFECA)  t1 = y3+v2;
          (J)       z3 = s1+t1;
        end.
```

Table 2: *The PSSA-form of the if-converted code of Table 1.*

ordered from left to right. For convenience, we assume that all operators are binary.[6]

Figure 5(a) illustrates the construction above by means of the value graph of the program of Figure 4(b). Note that the toned oval boxes are not part of the value graph, but represent the auxiliary information used by the CP-algorithm working on it. Intuitively, the CP-algorithm of [Knoop and Rüthing, 2000] iteratively visits the nodes of the value graph of a program, until the greatest fixed point is reached. Figure 5 illustrates this fixed point iteration for the example of Figure 4(b). Key is the evaluation of nodes labelled by ordinary operators and ϕ-operators, respectively. It is done according to the following two rules, which reflect the evaluation of terms, and the merge of data-flow information at join nodes in the program. The variable $\mathrm{dfi}[n]$ (short for *data-flow information*) stores the information at the value graph node n: (1) *Evaluating terms*: $\mathrm{dfi}[n] = I_0(\omega)(\mathrm{dfi}[\mathrm{l}(n)], \mathrm{dfi}[\mathrm{r}(n)])$ with *opt* the operator at node n. (2) *Merging DFA-info's at join nodes*: $\mathrm{dfi}[n] = \mathrm{dfi}[\mathrm{l}(n)] \sqcap \mathrm{dfi}[\mathrm{r}(n)]$.

4.2 The Global CP-Algorithm

The complete algorithm is composed of a global and a local component. In essence, the global component controls the propagation of the constant propagation information collected by the local component in hyperblocks throughout the program. Hence, the global algorithm works on a partitioning of the program in hyperblocks, and matches the usual pattern of a fixed point algorithm. Note that a hyperblock may have multiple exits. This must be taken care of when propagating data-flow information through the program during the course of updating the successor-environment of a hyperblock. In the algorithm below, we denote the data-flow information valid at an (exit) instruction m in a hyperblock hb by $\mathit{Info}[hb@m]$. This information can easily be extracted from the predicated value graph, the data structure used by the local component of our algorithm (cf. Section 4.3).

Algorithm 1 The Global CP-Algorithm.

Input: (1) A flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, and (2) a partitioning of G into a set of hyperblocks $HB = \{HB_1, \ldots, HB_k\}$ such that HB_1 contains the start node $\mathbf{s}$ of G, and a start information $c_{\mathbf{s}} \in [\mathit{Var} \to D' \cup \{\bot, \top\}]$ mapping every variable occurring in the program to a value ensured to be valid on entering G.

Output: An annotation of G with constant propagation information.

Remark: The variable *workset* controls the iterative process. Its elements are hyperblocks of G, whose annotating informations have recently been updated.

[6] Extensions to operators of arbitrary arity are straightforward.

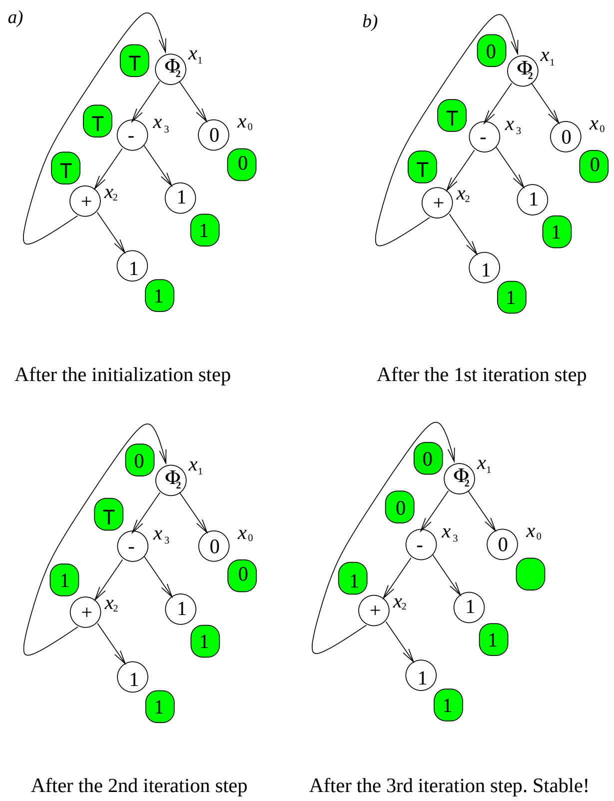

Figure 5: *The data-flow analysis on the value graph of the program of Figure 4(b): (a) the start annotation and (b) the greatest fixed point annotation.*

$\top_F$ is assumed to denote the greatest element of $[Var \to D' \cup \{\perp, \top\}]$, pred and succ denote the predecessors and successors of a node.

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)
FORALL $hb \in HB \backslash \{HB_1\}$ DO $inf[hb] := \top_F$ OD;
$inf[HB_1] := c_{\mathbf{s}}$;
$workset := \{HB_1\}$;

(Main process: Iterative fixed point computation)
WHILE $workset \neq \emptyset$ DO
    CHOOSE $hb \in workset$;
        $workset := workset \backslash \{hb\}$;
        (Calling the Local Component for CP on Hyperblocks (cf. Section 4.3))
        "choose basic resp. full version of the local algorithm to analyze hb";
        (Update the successor-environment of block hb)
        FORALL $hb' \in succ(hb)$ DO

```
        meet := inf[hb'] ⊓ ⊓{Info[hb@m] | m ∈ Nodes(hb) ∩ pred(start(hb'))};
        IF meet ⊏ inf[hb']
          THEN
            inf[hb'] := meet;
            workset := workset ∪ {hb'} FI
    OD
  ESOOHC
OD.
```

4.3 The Local CP-Algorithm

The local component of our new algorithm works on predicated hyperblocks. Conceptually, it follows the lines of the algorithm of [Knoop and Rüthing, 2000] working on the value graph of a program. Most important is thus the introduction of a predicated version of the value graph. We do this in two steps leading us to two value graph variants of different expressivity. Each of them induces a local CP-algorithm, called the basic and the full version, respectively. The unique power of these algorithms as illustrated in part (b) and (c) of the running example of Figure 3.

4.3.1 Basic CP-Algorithm

Figure 6 shows the PSSA-based variant of the value graph resulting from the first step mentioned above. We call it the basic variant as it does not model guarding predicates. In essence, this means that program branches are treated non-deterministically. This value graph variant allows already trace-precise CP with respect to non-deterministically interpreted program branches. In comparison to the value graph for non-predicated code, the *super nodes* represented by rectangular boxes are the only novelty. They allow us to handle the various duplicates introduced in PSSA for computations occurring in join blocks. While the indexed versions of e.g., $s1$ and $t1$ represent unique SSA-names for the left-hand sides of the duplicated instructions, the unindexed variables $s1$ and $t1$ attached to the super nodes allow us to maintain information common to all their indexed versions representing pathwise specializations. Given this value graph, the new CP-algorithm proceeds essentially as its counterpart of [Knoop and Rüthing, 2000]. This is illustrated in Figure 7. The only novelty is the treatment of nodes whose outgoing edges point to supernodes, like e.g., the node labelled by $z3$. Here, the evaluation has to take all combinations of corresponding, also called *matching* operands into account, i.e., to evaluate $s1_{JGFBA} + t1_{JGFBA}, s1_{JGFDCA} + t1_{JGFDCA}, \ldots, s1_{JHFECA} + t1_{JHFECA}$, and to merge the results. This way, the new algorithm detects the constancy of z at the end of the hyperblock, though none of its subterms is constant. Finally, it is worth noting that the annotation of the PSSA-based value graph requires only a single pass starting from the leafs and proceeding to the root nodes. This is because of the absence of cyclic dependencies as the local component is dealing with hyperblocks. The following paragraph makes the notion of the basic version of the *predicated value graph* (PVG), and the evaluation procedure more precise.

The basic predicated value graph consists of elementary nodes EN and super nodes SN. Elementary nodes correspond to the nodes of an ordinary value graph. Those encapsulated by a super node are also called *component nodes*. The indexed variables attached to a component node are unique SSA-names for the left-hand sides of the duplicated instructions. The index of the corresponding full path predicate identifies them uniquely. Applied to a supernode, the function *ElemNodes* yields the set of its component nodes, applied to an elementary node, it returns the argument. Two component nodes n_{s_1} and n_{s_2} of different super nodes s_1 and s_2 *match* each other if their indexing full path predicates are the same.[7] Additionally, a component node of a supernode *matches* each elementary node which is not a component node. We denote the corresponding relation by *match*. It is central to formulate the rule governing the evaluation of terms having super nodes as operands. Next, we are presenting these evaluation rules which are the analogue to the rules of the unpredicated setting recalled earlier in Section 4. The extension of the function *ElemNodes* and the relation *match* to elementary nodes allows us to avoid lengthy case distinctions. Note that the main step below does not require iterated visits of a node, if nodes are visited levelwise starting at the level of leaves, and if on each level the elementary nodes are processed before their enclosing super nodes.

Initial Step: For each node n in the predicated value graph initialize:

$$\text{dfi}[n] = \begin{cases} I_0(c) & \text{if } n \text{ is an elementary leaf node with } \text{lab}[n] = c \\ \top & \text{otherwise} \end{cases}$$

Main Step:

1. For an elementary inner node n labelled with operator ω:
$\text{dfi}[n] = \sqcap\{I_0(\omega)(\text{dfi}[cnl], \text{dfi}[cnr]) \mid cnl \in \mathit{ElemNodes}(\mathrm{l}(n)) \ \wedge$
$cnr \in \mathit{ElemNodes}(\mathrm{r}(n)) \ \wedge \mathit{match}(cnl, cnr)\}$ **(Evaluating terms)**

2. For a supernode n: $\text{dfi}[n] = \sqcap\{\text{dfi}[cn] \mid cn \in \mathit{ElemNodes}(n)\}$
(Merging DFA-info's of duplicated variables)

[7] Suppressing the introduction of (some) unnecessary duplicates in the style of Carter et al., the indexing predicates are generally disjunctions of full path predicates. In this case, the nodes n_{s_1} and n_{s_2} match each other, if the indexing predicate of n_{s_1} is a disjunctive operator in the one of n_{s_2} or vice versa.

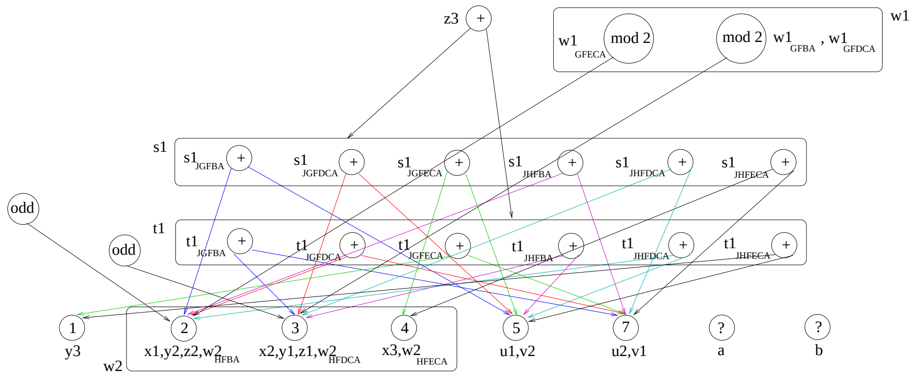

Figure 6: *The basic PSSA-based value graph.*

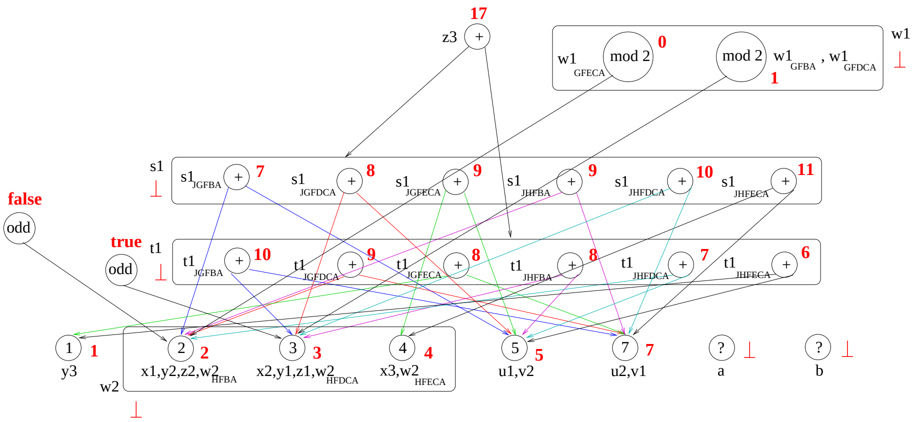

Figure 7: *The basic PSSA-based value graph after constant propagation.*

4.3.2 Full CP-Algorithm

The full version of the PSSA-based PVG takes additionally guarding predicates into account. In essence, predicates are treated as any other variable occurring in the program. In addition, however, the PVG connects each variable with the PVG node representing the predicate guarding the instruction in the PSSA of the program assigning to it. This makes the PVG predicate-sensitive as illustrated in Figure 8, and allows us to interpret branching deterministically, where possible. For clarity only those edges are shown in Figure 8, which are needed for the example.

Given the PVG, the evaluation process proceeds essentially as for the basic algorithm. It proceeds from the leaves towards the roots and does not require any iterations. In fact, as in the basic algorithm, each node is visited exactly once. The result of this evaluation process is displayed in Figure 9. In distinction to the basic algorithm, the evaluation of nodes whose outgoing edges point to supernodes is now controlled by the edges indicating the relevant guarding predicates. Technically, this is achieved by replacing the evaluation step for terms of the evaluation procedure of the basic algorithm by

1′. For an elementary inner node n labelled with operator ω:

$$\begin{aligned}\mathrm{dfi}[n] = \sqcap\{I_0(\omega)(\mathrm{dfi}[cnl], \mathrm{dfi}[cnr]) \mid\ & cnl \in ElemNodes(\mathrm{l}(n)) \ \wedge \\ & cnr \in ElemNodes(\mathrm{r}(n)) \ \wedge match(cnl, cnr) \ \wedge \\ & guardingPred(\{cnl, cnr\}) \subseteq \{true, \top\}\,\}\end{aligned}$$

(Evaluating terms)

New is the predicate $guardingPred$. It acts as a predicate-sensitive filter causing the evaluation of terms only for those operands, whose guarding predicate is true ("*true*"), or has not yet been evaluated ("$\top$"). To avoid case distinctions in the definition, $guardingPred$ is defined to be true for elementary nodes which are no component nodes. For a component node it holds, if the data-flow information attached to the node of its guarding predicate is *true* or $\top$. The complete step then leads to predicate-sensitive trace-precise CP. In addition to $z3$, the full algorithm detects the constancy of $w1$ and $w2$ located in the blocks G and H in the original program, too.

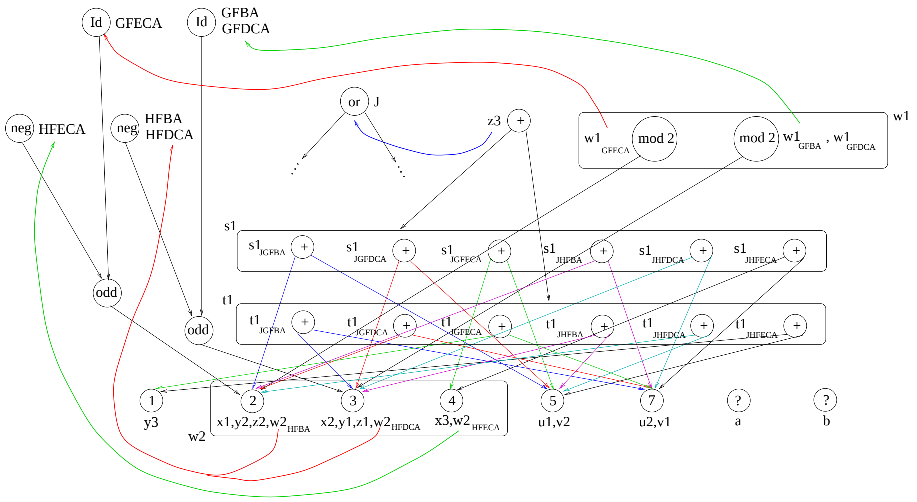

Figure 8: *The predicated value graph (as far as being relevant for the example).*

4.3.3 The Running Example.

Table 3 shows the effect of our CP-algorithm to the running example. Off-predicated instructions marked by [∗] in Table 2 have been removed. Instructions marked by [−] have been simplified in comparison to their counterparts of Table 2 as some predicates occurring as operands in these instructions have been detected to be false by our algorithm.

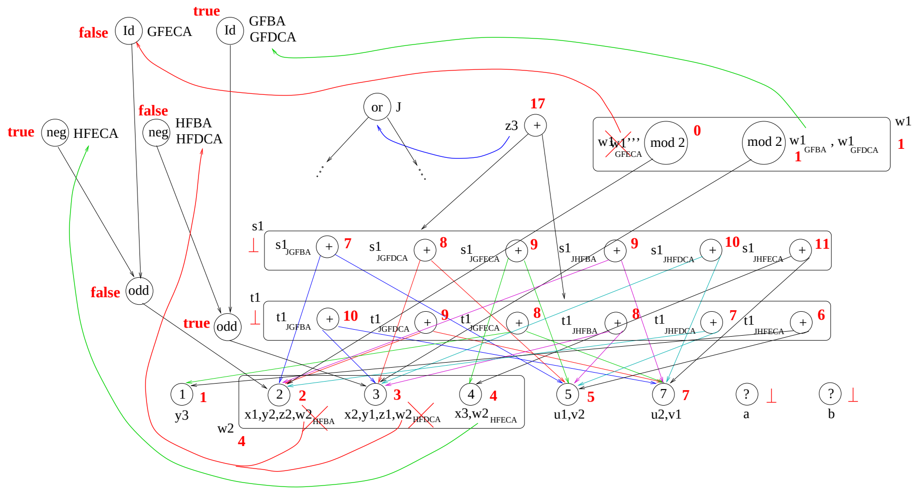

Figure 9: *The predicated value graph after constant propagation.*

5 Main Results

In this section we summarize the main results applying to the new algorithm and its local components. First of all, the overall approach is sound, i.e., whenever a term is reported to be a specific constant, then it is.

Theorem 2 Soundness.
The global CP-algorithm (Algorithm 1) as well as the basic and full version of its local component are sound.

Additionally, both the basic and the full version of its local component enjoy specific completeness (optimality) results. The basic version is optimal on acyclic code with respect to a non-deterministic interpretation of control branches; the full version is optimal with respect to a trace-precise deterministic interpretation. In essence, both the soundness and optimality results are consequences of the power of the underlying PSSA-form, which maintains information about each control path existing in the underlying hyperblock.

Theorem 3 Completeness (Optimality).

1. *The basic version of the local component of the new CP-algorithm is* trace-precise *with respect to the non-deterministic interpretation of branching, i.e., it detects all constants, which do not depend on the particular branching structure of the underlying hyperblock.*

2. *The full version of the local component of the new CP-algorithm is* predicate-sensitive trace-precise, *i.e., it detects all constants in the hyperblock, which do not depend on specific initial information being valid on entering the underlying hyperblock.*

```
         begin
           (p0)      A = OR(TRUE);
           (A)       a1 = random();
           (A)       b1 = random();
           (A)       z1 = 3;
           (A)       cmp.unc BA,CA (a1>0);
           (p0)      B = OR(BA);
           (p0)      C = OR(CA);
           (B)       x1 = 2;
           (B)       y1 = 3;
           (C)       cmp.unc DCA,ECA (b1>0);
           (p0)      D = OR(DCA);
           (p0)      E = OR(ECA);
           (D)       x2 = 3;
           (D)       y2 = 2;
           (E)       z2 = 2;
           (E)       x3 = 4;
           (E)       y3 = 1;
           (BA)      FBA = OR(TRUE);
           (DCA)     FDCA = OR(TRUE);
           (ECA)     FECA = OR(TRUE);
           (p0)      F = OR(FBA,FDCA,FECA);
           (FBA)     cmp.unc GFBA,HFBA (TRUE));
           (FDCA)    cmp.unc GFDCA,HFDCA (TRUE);
           (FECA)    cmp.unc GFECA,HFECA (FALSE);
   [-]     (p0)      G = OR(GFBA,GFDCA);
   [-]     (p0)      H = OR(HFECA);
           (G)       w1 = 1;
           (G)       u1 = 5;
           (G)       v1 = 7;
           (HFECA)   w2 = 4;
           (H)       u2 = 7;
           (H)       v2 = 5;
           (GFBA)    JGFBA = OR(TRUE);
           (GFDCA)   JGFDCA = OR(TRUE);
           (HFECA)   JHFECA = OR(TRUE);
   [-]     (p0)      J = OR(JGFBA,JGFECA,JHFECA);
           (JGFBA)   s1 = 7;
           (JGFBA)   t1 = 10;
           (JGFECA)  s1 = 9;
           (JGFECA)  t1 = 8;
           (JHFECA)  s1 = 11;
           (JHFECA)  t1 = 6;
           (J)       z3 = 17;
         end.
```

Table 3. *The PSSA-form of the if-converted code of the running example after constant propagation and simplification.*

As mentioned earlier, there are two important benefits of our approach coming for free. First, the local component of our CP-algorithm provides a new approach for CP of acyclic non-predicated code. Remapping the analysis results obtained by its basic and full version to the original program results in the promised results of part (b) and (c) of Figure 3. Second, the full version of the local component allows us to identify off-predicated insertions as it detects the guarding predicate of being equivalent to false. They can simply be eliminated as demonstrated in

Table 3. Besides enhancing the performance further, it makes subsequent compilation phases such as instruction scheduling more effective [Haga and Barua, 2001]. In fact, the positive impact on such interdepending transformations is a major achievement of our approach.

6 Conclusions

We presented an algorithm for CP on predicated code, which, to the best of our knowledge, is the first CP-algorithm for the predicated setting. The new algorithm works for arbitrary control flow, detects constancy of terms whose operands are not constant themselves, and is optimal for hyperblocks, i.e., its results are trace-precise taking the guarding predicates of the instructions fully into account. As an important side-effect, this allows us to identify off-predicated instructions. Eliminating them enhances the performance, and makes compilation phase like scheduling more effective. Moreover, the new algorithm provides a new means for optimal CP on acyclic programs. Because of taking branching into account, it is even more powerful than the algorithm for finite constants of [Steffen and Knoop, 1989; Steffen and Knoop, 1991], which interprets branching non-deterministically. The complexity of both algorithms is exponential, but unavoidably for optimal algorithms as recent research indicates (cf. [Müller-Olm and Rüthing, 2001]). Moreover, the compiler-controlled construction of hyperblocks provides an easy means for controlling the computational costs of the algorithm's local component. Conceptually, this component resembles the algorithm of [Knoop and Rüthing, 2000] for non-predicated code as both work on a (P)SSA-based variant of a value graph. On the one hand side, the elegant extensibility of this approach from the non-predicated setting with the SSA-form to the predicated one with the PSSA-form shows the adequacy of the overall approach simultaneously answering a demand posed in [Carter et al., 1999] for further applications of the PSSA-form. As mentioned, with the advent of the IA-64 architecture the need for such techniques will dramatically increase (cf. [Knoop et al., 2000]). On the other hand, the limitation of the PSSA-form of [Carter et al., 1999] to hyperblocks shows that predicated code still lacks a fully adequate analogue of the successful SSA-representation of non-predicated code. We are currently investigating such extensions and their applications.

References

Aho, A. V., , and Ullman, J. D.: "Principles of Compiler Design"; Addison-Wesley, Reading, MA (1977).

Aho, A. V., Sethi, R., and Ullman, J. D.: "Compilers: Principles, Techniques and Tools"; Addison-Wesley (1985).

Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. "Conversion of control dependence to data dependence"; Conf. Rec. 10th Annual Symp. on Principles of Prog. Lang. (POPL) (1983), pages 177 – 189. ACM, NY.

Alpern, B., Wegman, M. N., and Zadeck, F. K.: "Detecting equality of variables in programs"; Conf. Rec. 15th Symp. Principles of Prog. Lang. (POPL) (1988), pages 1 – 11. ACM, NY.

August, D. I.: "Systematic Compilation For Predicated Execution"; PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Electrical and Computer Engeneering (2000).

August, D. I., Hwu, W., and Mahlke, S. A.: "A framework for balancing control flow and predication"; Proc. 30th Annual IEEE/ACM Int. Symp. on Microarchitectures (MICRO-30), volume 28 of ACM SIGMICRO Newsletter (1997), pages 92 – 103.

Aycock, J. and Horspool, N.: "Simple generation of static single-assignment form"; Proc. 9th Int. Conf. on Compiler Construction (CC) (2000), LNCS 1781, pages 110 – 124. Springer-V.

Carter, L., Simon, B., Calder, B., Carter, L., and Ferrante, J.: "Predicated static single assignment"; Proc. 7th IEEE Int. Conf. on Parallel Architectures and Compilation Techniques (PACT) (1999), pages 245 – 255. IEEE Comp. Soc., Los Alamitos.

Chow, F., Chan, S., Kennedy, R., Liu, S., Lo, R., and Tu, P.: "A new algorithm for partial redundancy elimination based on SSA form"; Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI), volume *32*,5 of ACM SIGPLAN Not. (1997), pages 273 – 286.

Intel Corp.: "IA-64 Application Developer's Architecture Guide" (1999).

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K.: "An efficient method of computing static single assignment form"; Conf. Rec. 16th Symp. on Principles of Prog. Lang. (POPL) (1989), pages 25 – 35. ACM, NY.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K.: "Efficiently computing static single assignment form and the control dependence graph"; ACM Trans. Prog. Lang. Syst., 13(4) (1991):451 – 490.

Dulong, C.: "The IA-64 architecture at work"; IEEE Computer, 31(7) (1998):24 – 32.

Haga, S. and Barua, R.: "EPIC instruction scheduling based on optimal approaches"; 1st Int. Workshop on Explicitly Parallel Instruction Computing (EPIC-1) (2001).

Hu, P.: "Static analysis for guarded code"; Technical Report 3879, INRIA, Institut National de Recherche en Informatique et en Automatique (2000), available via web page: http://www.inria.fr/rrr/rr-3979.html.

Kam, J. B. and Ullman, J. D.: "Monotone data flow analysis frameworks"; Acta Informatica, 7 (1977):305 – 317.

Kennedy, K.: "A survey of data flow analysis techniques"; In Muchnick, S. S. and Jones, N. D., editors, "Program Flow Analysis: Theory and Applications" (1981), chapter 1, pages 5 – 54. Prentice Hall, Englewood Cliffs, NJ.

Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P., and Chow, F.: "Partial redundancy elimination in SSA form"; ACM Trans. Prog. Lang. Syst., 21(3) (1999): 627 – 676.

Kennedy, R., Chow, F., Dahl, P., Liu, S.-M., Lo, R., and Streich, M.: "Strength reduction via SSAPRE"; Proc. 7th Int. Conf. on Compiler Construction (CC) (1998), LNCS 1383, pages 144 – 158. Springer-V., Germany.

Kildall, G. A.: "A unified approach to global program optimization"; Conf. Rec. 1st Symp. Principles of Prog. Lang. (POPL) (1973), pages 194 – 206. ACM, NY.

Knoop, J., Collard, J.-F., and Ju, R. D.: "Partial redundancy elimination on predicated code"; Proc. 7th Static Analysis Symposium (SAS) (2000), LNCS 1824, pages 260 – 279. Springer-V.

Knoop, J. and Rüthing, O.: "Constant propagation on the value graph: Simple constants and beyond"; Proc. 9th Int. Conf. on Compiler Construction (CC) (2000), LNCS 1781, pages 94 – 109. Springer-V.

Knoop, J. and Rüthing, O.: "Constant propagation on predicated code"; In Ierusalimschy, R., Figueiredo, L., and Valente, M. T., editors, Proc. 7th Brazilian Symposium on Programming Languages (SLBP) (2003), pages 135 – 148. Sociedade Brasileira de Computação.

Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A.: "Effective compiler support for predicated execution using the hyperblock"; 25th Annual Int. Symp. on Microarchitecture (MICRO-25), volume *23*,1&2 (1992), pages 45 – 54.

Muchnick, S. S.: "Advanced Compiler Design and Implementation"; Morgan Kauf-

mann, San Francisco, CA (1997).
Müller-Olm, M. and Rüthing, O.: "The complexity of constant propagation"; Proc. 10th European Symposium on Programming (ESOP) (2001), LNCS 2028, pages 190–205. Springer-V.
Park, J. C. H. and Schlansker, M.: "On predicated execution"; Technical Report HPL-91-58 (1991), HP Labs.
Rosen, B. K., Wegman, M. N., and Zadeck, F. K.: "Global value numbers and redundant computations"; Conf. Rec. 15th Symp. Principles of Prog. Lang. (POPL) (1988), pages 2 – 27. ACM, NY.
Steffen, B. and Knoop, J.: "Finite constants: Characterizations of a new decidable set of constants"; Proc. 14th Int. Symp. on Math. Foundations of Computer Science (MFCS) (1989), LNCS 379, pages 481 – 491. Springer-V., Germany.
Steffen, B. and Knoop, J.: "Finite constants: Characterizations of a new decidable set of constants"; TCS, 80(2) (1991):303 – 318.
Warter, N. J., Mahlke, S. A., Hwu, W.-M., and Rau, B. R.: "Reverse if-conversion"; Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI) (1993), volume *28*,6 of *ACM SIGPLAN Not.*, pages 290 – 299.
Wegman, M. N. and Zadeck, F. K.: "Constant propagation with conditional branches"; Conf. Rec. 12th Annual Symp. on Principles of Prog. Lang. (POPL) (1985), pages 291 – 299. ACM, New York.
Wegman, M. N. and Zadeck, F. K.: "Constant propagation with conditional branches"; ACM Trans. Prog. Lang. Syst., 13(2) (1991):181 – 210.