A Loop Optimization Technique Based on Quasi-Invariance

Litong Song[1], Yoshihiko Futamura[1], Robert Glück[1], Zhenjiang Hu[2]
[1] Dept. of Information and Computer Science, Graduate School of Science and Engineering, Waseda University, Okubo 3-4-1, Shinjuku-ku, Tokyo 169-0072, Japan,
E-mail: {slt, futamura}@futamura.info.waseda.ac.jp, glueck@acm.org
[2] Dept. of Information Engineering, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan,
E-mail: hu@ipl.t.u-tokyo.ac.jp

Abstract
Loop optimization plays an important role in compiler optimization and program transformation. Many sophisticated techniques such as loop-invariance code motion, loop restructuring and loop fusion have been developed. This paper introduces a novel technique called loop quasi-invariance code motion. It is a generalization of standard loop-invariance code motion, but based on loop quasi-invariance analysis. Loop quasi-invariance is similar to standard loop-invariance but allows for a finite number of iterations before computations in a loop become invariant. In this paper we define the notion of loop quasi-invariance, present an algorithm for statically computing the optimal unfolding length in While-programs and give a transformation method. Our method can increase the accuracy of program analyses and improve the efficiency of programs by making loops smaller and faster. Our technique is well-suited as supporting transformation in compilers, partial evaluators, and other program transformers.

1. Introduction

Loop-invariance code motion is a well-known loop transformation technique that plays an important role in compiler optimization. When a computation in a loop does not change during the dynamic execution of the loop, we can hoist this computation out of the loop to improve execution time of the loop. For example, the evaluation of expression $I\times10$ is loop-invariant in the following loop: *while* $i<I\times10$ *do* $s:=s+i; i:=i+1;$ *endwhile*. A more efficient program is: $t:=I\times10;$ *while* $i<t$ *do* $s:=s+i; i:=i+1;$ *endwhile*.

Traditional transformations move out of loops the following: (i) computations that are invariant during all loop iterations (loop invariant code motion [1]), (ii) computations that are invariant after the first iteration, (loop peeling, e.g., [23]), (iii) computations that are conditionally invariant (e.g., speculation motion [11]).

Techniques for code motion are basically limited to loop-invariant computations, this can be a problem in automatically produced programs. Consider the digital circuit in Fig. 1, which is simulated by the program in Fig. 2, where the simulation is carried out for T steps. We use f, g and g_i to denote functions simulating blocks.

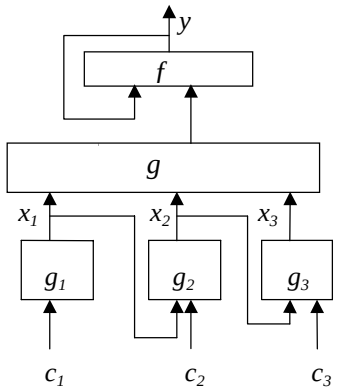


Fig. 1. A sequential digital circuit.

```
while t<T do
   x3:=g3(x2,c3);
   x2:=g2(x1,c2);
   x1:=g1(c1);
   y:=f(g(x1,x2,x3),y);
   t:=t+1;
endwhile.
```
Fig. 2 A simulation program for Fig. 1.

With traditional code motion technique, we can only hoist $g_1(c_1)$ out of the loop, and save it using a fresh variable. In fact, we can obtain a more efficient code (Fig. 3) by unfolding the original loop three times:

```
if t<T then      /* loop 1 */
   x3:=g3(x2,c3);
   x2:=g2(x1,c2);
   x1:=g1(c1);
   y:=f(g(x1,x2,x3),y);
   t:=t+1;
```

```
if t<T then      /* loop 2 */
    x3:=g3(x2,c3);
    x2:=g2(x1,c2);
    y:=f(g(x1,x2,x3),y);
    t:=t+1;
    if t<T then      /* loop 3 */
        x3:=g3(x2,c3);
        y:=f(g(x1,x2,x3),y);
        t:=t+1;
        _g:= g(x1,x2,x3);
        while t<T do       /* residual loop */
            y:=f(_g,y);
            t:=t+1;
        endwhile;
    endif endif endif
```

Fig. 3 The program after loop quasi-invariance code motion of Fig. 2.

In loop 1, the invariance of $g_1(c_1)$ results in the invariance of x_1, and the same assignment to x_1 in the following iterations can be removed safely. In loop 2, the invariance of $g_2(x_1,c_2)$ results in the invariance of x_2, and the assignments to x_2 afterwards can be safely removed too. For the same reason, the assignment to x_3 in loop 3 can be removed.

For the residual loop, after x_1, x_2 and x_3 have turned into loop-invariant variables, $g(x_1,x_2,x_3)$ becomes a loop-invariant computation. Applying traditional code motion technique to the remaining loop will yield the residual loop above (Fig. 2).

Table 1 shows the speedup where functions g_1, g_2, g_3, g, f, t and T are defined in Appendix 1.

Table 1. The comparison of the runtime of the loop in Fig. 2 and the program in Fig. 3.

System	Gateway E5250 Pentium II xeon×2 Speed: 450MHZ Memory: 1GB, Visual C++ 6.0		Sun ULTRA 5 SunOS 5.6 Speed: 270MHZ Memory:192MB Gcc 2.7	
	Runtime	Speed up	Runtime	Speed up
Source	22sec	1	25sec	1
Result	5sec	4.4	5sec	5

To conclude, we have seen that a full optimization of the program in Fig. 2 is not possible by traditional code motion, because some of the computations in the loop are stabilize only after a finite number of iterations. It is clear that a hand-optimization of program may be error-prone and tedious, and the initial specification of the circuit may change. This is why we are looking for an algorithmic solution.

The main contributions of this paper are:

- a formalization of loop quasi-invariance and optimal unfolding length for a loop,
- a static quasi-invariance analysis for computing the optimal unfolding length for any given loop in a program which allows to remove all quasi-invariant variables from a loop while avoiding over-unfolding,
- a loop transformation technique using the results of the analysis,
- an illustration of the effect of loop quasi-invariance code motion on program transformation, in particular partial evaluation.

2. Preliminaries

We define a source language and summarize the single static assignment form.

2.1 While-language

We introduce an imperative *While-language*. The syntax is given in Fig. 4, the semantics is as in Pascal. A While-program is a sequence of statements. Each statement is either an *assignment*, a *conditional*, a *while* loop, a *function call*, or a *skip* statement. For simplicity of the technical presentation, we assume a call-by-value semantics for functions, and we treat all functions as primitive operations and assume they are free of side effects.

```
SS     ::=  S | S; SS
S      ::=  Ass | Cond | Loop | Call | skip
Ass    ::=  V:=E
Cond   ::=  if E then SS else SS endif
           | if E then SS endif
Loop   ::=  while E do SS endwhile
Call   ::=  F(E*)
E      ::=  Variable | Constant | Op(E*) | Call
Op     ::=  + | – |× | /
```

Fig. 4 The syntax of the While-language.

2.2 Single Static Assignment

This section summarizes the single static assignment form (SSA) [17]; readers familiar with SSA form may only want to look at the format of a *while* loop (Fig. 8). SSA form is a program representation in which there is only one assignment to each variable in the program and every use of a variable is defined by such an assignment. SSA form is important for program optimization.

Consider the following example:

```
while i<100 do
    x:=1; …x…;
    x:=i+2; …x…;
    i:=i+1;
endwhile
```

There are two assignments to variable *x* inside the loop. After one iteration variable *x* in assignment "*x:=1*" becomes invariant, but we can not move the assignment out of the loop because the value of *x* in the first use after the assignment would become equal to the value of *x* defined in assignment "*x:=i+2*" which is not correct.

The purpose of the SSA form is to represent data flow properties of a program in a normalized form. Many compilers use SSA or intermediate representation where a program is transformed into SSA form, optimized, and then transformed back to the original syntax. We follow the same approach.

Let us summarize the SSA form. First, the variable on the left-hand side of each assignment is given a unique name, and all of its uses are renamed correspondingly (Fig.5). Second, if the use of a variable *x* can be reached by two or more definitions, which maybe the case after a conditional, a special form of assignment called a *ϕ-function*, is added at the join point. The operands of the *ϕ-function* indicate which assignments to *x* reach the join point. Subsequent uses of *x* become uses of *ϕ-function* value. We call the values assigned by *ϕ-function* as *virtual variables*. This is illustrated in Fig. 6.

$x:= 1; \ldots := x+2;$ ⇒ $x_1:= 1; \ldots := x_1+2;$
$x:= 3; \ldots := x+4;$ $x_2:= 3; \ldots := x_2+4;$

Fig.5. Straight-line code and its SSA form.

if e then x:= 1; ⇒ *if e then* $x_1:= 1;$
else x:= 2; endif *else* $x_2:=2;$ *endif;*
$x_3:=\phi(x_1, x_2);$

Fig. 6 An if-statement and its SSA form.

An efficient algorithm that converts a program into SSA form and works essential linear in the size of the original program has been proposed in [6]. The following example (Fig. 7) briefly explains the transformation. We are not interested in what the program does, only in illustrating the SSA form.

while i≤0 do
  $w:=w+10^3\times x^3+10^2\times x^2+10\times x+1;$
  x:=u+1;
  if even(i) then y:=z+1; else y:=v; endif;
  y:=x;
  if z>0 then y:=1; if z>v then y:=z+1; endif
  endif;
  if z>1000 then y:=y+1; else y:=i; endif
  u:=z–1;
  z:=v+1;
  i:=i+1;
endwhile.

Fig. 7 An original loop.

Using SSA technique, the loop is transformed into SSA form (Fig. 8). Note that we assume any loop in SSA form is in the form of *while* $[ss_1]$ *e do* ss_2 *endwhile*, where ss_1 is a series of inserted assignments which should be executed before testing the entry condition and ss_2 is the body of the loop after SSA transformation.

while [$i_1:=\phi(i_0,i_2);$
  $x_1:=\phi(x_0,x_2);$
  $y_1:=\phi(y_0,y_{12});$
  $u_1:=\phi(u_0,u_2);$
  $z_1:=\phi(z_0,z_2);$
  $w_1:=\phi(w_0,w_2);$
 ] $i_1\le 0$ *do*
  $w_2:=w_1+10^3\times x_1^3+10^2\times x_1^2+10\times x_1+1;$
  $x_2:=u_1+1;$
  if even(i_1) *then* $y_2:=z_1+1;$ *else* $y_3:=v;$ *endif;*
  $y_4:=\phi(y_2,y_3);$
  $y_5:=x_2;$
  if $z_1>0$ *then* $y_6:=1;$
    if $z_1>v$ *then* $y_7:=z_1+1;$ *endif;*
    $y_8:=\phi(y_6,y_7);$
  endif;
  $y_9:=\phi(y_5,y_8);$
  if $z_1>1000$ *then* $y_{10}:=y_9+1;$ *else* $y_{11}:=i_1;$ *endif;*
  $y_{12}:=\phi(y_{10},y_{11});$
  $u_2:=z_1-1;$
  $z_2:=v+1;$
  $i_2:=i_1+1;$
endwhile.

Fig. 8 The SSA form of the loop in Fig. 7.

3. Variable Dependency Graph

The loop transformation we want to perform depends on the loop quasi-invariant variables and the unfolding length of loops. Before we define these notions, we introduce two variable dependency relations and variable dependency graph to perform our quasi-invariance analysis. First, we assume a loop contains only assignment, then we extend our discussion to conditionals and nested loops. Here and in the remainder of this paper, we assume that all source programs are represented in SSA form.

3.1 Variable Dependency Graph

For any assignment *v:=e*, we assume that the value of *v* directly depends on all variables used in *e*. We can define a variable dependency relation as follows.

Definition 1 (∠ *relation*). Let *o* be a loop, *x*, *y* be two

variables assigned to inside o. If the value of x depends on that of y, then the dependency relation between x and y, is denoted by $x \angle y$ (called $\angle$ *relation*).

Within relation $\angle$, we can distinguish another variable dependency relation.

Definition 2 ($\lhd$ *relation*). Let o be a loop, x, y be two variables defined inside o, and $x \angle y$. If y is assigned to inside o after being used by x inside o, then the dependency relation between x and y, is denoted by $x \lhd y$ (called $\lhd$ *relation*).

To formalize loop quasi-invariance, we introduce a directed graph called *variable dependency graph*.

Definition 3 (*Variable dependency graph*). Let o be a loop. The *variable dependency graph* (VDG) of o is a directed graph where $Node(o)=\{x \mid$ variable x is defined in $o\}$ and $Edge(o)=\{x \rightarrow y \mid (y \angle x) \wedge \neg(y \lhd x)\} \cup \{x \twoheadrightarrow y \mid y \lhd x\}$.

For example, for the loop in Fig. 2, we have the $\angle$ relations: $\{x_2 \angle x_1, x_3 \angle x_2, y \angle x_1, y \angle x_2, y \angle x_3, y \angle y, t \angle t\}$, the $\lhd$ relations: $\{x_2 \lhd x_1, x_3 \lhd x_2, y \lhd y, t \lhd t\}$, and the VDG in Fig. 9, where relations $\angle$ and $\lhd$ are indicated by thin and thick edges, respectively. Note that there is only one assignment to one variable in Fig. 2, for concision we ignore the SSA form of Fig. 2 but only use the original form of Fig. 2.

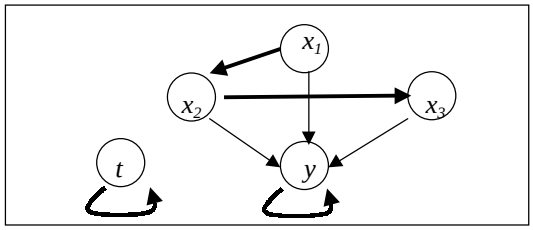


Fig. 9 The variable dependency graph of Fig. 2.

3.2 Loop Quasi-Invariant Variable

Based on VDG, we now give a formal definition of loop quasi-invariant (LQIV) variable.

Definition 4 (*LQIV*). Let o be a loop. For any node x on the VDG of o, variable x is loop quasi-invariant (LQIV) variable of loop o, if among all the paths ending in x, there is no path which contains a node that is a node on a circular path.

Furthermore, we can give a formal definition of invariant length of LQIV variable.

Definition 5 (*Invariant length*). Let o be a loop. For any LQIV node x (x is a LQIV variable of o) on the VDG of o, the *invariant length* of x $IL(x,o) =_{def} 1+\max\{n \mid n$=the number of $\twoheadrightarrow$ edges on a path ending in node $x\}$.

For any LQIV variable x of a loop o, $IL(x,o)$ means that x will be turned into loop-invariant variable after $IL(x,o)$ iterations of o.

Proposition 1. Let o be a loop and x be LQIV of o. If $x^{(n)}$ is used to represent the value of x after n iterations of o, then $\forall n \geq IL(x,o).(x^{(n+1)}=x^{(n)})$.

The proposition is proven by induction over n; proof omitted. Based on the invariant lengths of the LQIV variables inside a loop, we can directly infer how many times the loop should be unfolded.

Definition 6 (*Unfolding length*). Let o be a loop. If there exists LQIV variable inside o, then the *unfolding length* of o $UL(o) =_{def} \max\{IL(x,o) \mid x$ is LQIV and not a virtual variable of $o\}$.

For Example, in the program of Fig. 2, we have LQIV variables x_1, x_2, x_3 and $IL(x_1,o)=1$, $IL(x_2,o)=2$, $IL(x_3,o)=3$, $UL(o)=3$.

Virtual variables are newly inserted variables and they will be removed at unfolding time. Therefore, the invariant lengths of virtual variables are ignored in the definition of the unfolding length.

According to Definition 4, there exists no circular path ending in a LQIV node x. Therefore, the number of edges in all paths ending in x is finite, and the invariant length of x is finite. This ensures the termination of the analysis and loop unfolding. After a loop o is unfolded $UL(o)$ times, all LQIV variables in the residual loop of o become invariant and can be removed from the residual loop.

4. Conditionals in Loops

The variable dependency relation defined above is induced by assignment statements. As in other static analysis techniques, conditionals induce new variable dependencies. New dependencies are induced between the variables used in the test of a conditional and the variables defined in the branches of the conditional. We distinguish two cases:

CASE 1: *if* $op(y_1,\ldots,y_n)$ *then* … $x_i:=\ldots$; …
else … $x_j:=\ldots$; … *endif*;
$x_k:=\phi(x_i, x_j)$;

Whether the assignment to x_i and x_j can be removed from the conditional depends on the variables used in expression $op(y_1,\ldots,y_n)$. If one of the variables $y_1,\ldots,y_n$ is not LQIV, which means that the value of $op(y_1,\ldots,y_n)$ may change at run time, none of the assignments to x_i and x_j can be removed even if they would otherwise be LQIV variables. This kind of dependency relation between x_i, x_j and $y_1,\ldots,y_n$ can be expressed by adding dependency relations between x_i, x_j and $y_1,\ldots,y_n$: $x_j \angle y_1,\ldots,x_j \angle y_n$; $x_i \angle y_1,\ldots,x_i \angle y_n$. We say, these relations are induced by the conditional.

CASE 2: $x_j:=\ldots$;

if $op(y_1,\ldots,y_n)$ *then* … x_i:=…; …
else … *endif;*
$x_k:=\phi(x_i, x_j)$;

This is similar to CASE 1, except that the assignment to x_j is laid out of the conditional but not in one branch. For the same reason as in CASE 1, we need to establish the dependency relations between x_i, x_j and $y_1,\ldots,y_n$: $x_i \angle y_1$, …, $x_i \angle y_n$; $x_j \angle y_1$, …, $x_j \angle y_n$;

Before x_j becomes loop-invariant, assignment x_i:=… can not be removed. (otherwise x_k will always be equal to x_j, which is not correct.) Therefore we add dependency relation $x_i \angle x_j$. (we also say, this relation is induced by the conditional.)

After we get the variable dependency relations induced by the conditionals in a loop, we must transfer the same relations from virtual variables to their operands via the corresponding *ϕ-function*. This is necessary because virtual variables will be removed afterwards, (as discussed in Section 6.) and the dependency relations (from virtual variables to other variables) induced by conditionals are actually the ones between their operands and other variables. Moreover, if the operands of virtual variables are also virtual variables, the dependency relations have to be transferred further. We now define how the dependency relations induced by conditionals in a loop *o* are transferred via virtual variables. For any a dependency relation $x \angle y$ directly induced by the conditionals like CASE 1 and CASE 2, we use a function called *CS* to evaluate the above-mentioned transitive closure of relation $x \angle y$.

$$CS_o(x, y) =_{def} \begin{cases} \{\ \} & \text{if } x \text{ and } y \text{ are identical} \\ CS_o(x_1, y) \cup CS_o(x_2, y) & \text{if } x \text{ is a virtual variable of } o \\ & \quad \text{defined by } x:=\phi(x_1,x_2) \\ \{x \angle y\ \} & \text{otherwise} \end{cases}$$

Let *Cond_Rel(o)* be the set of variable dependencies directly induced by the conditionals in loop *o*, (as defined in CASE 1 and CASE 2.) and *Vars(o)* be the set of all variables defined in *o*. Then we define *VS(o)* (the set of all the relations derived from *Cond_Rel(o)*, including *Cond_Rel(o)*) as follow:

$$VS(o) =_{def} \bigcup_{x \in Vars(o)} \left(\bigcup_{y \in Vars(o) \wedge x \angle y \in Cond\_Rel(o)} CS_o(x, y) \right)$$

For example, let a loop contain the conditional:

x_1:=1;
if i>j *then if* k>5 *then* x_2:=2; *else* x_3:=3; *endif;*
$x_4:=\phi(x_2,x_3)$; *endif;*
$x_5:=\phi(x_1,x_4)$;

We can derive all the dependency relations induced by the conditional as follows:

$x_1 \angle i$, $x_1 \angle j$;
$x_2 \angle i$, $x_2 \angle j$, $x_2 \angle k$, $x_2 \angle x_1$
$x_3 \angle i$, $x_3 \angle j$, $x_3 \angle k$, $x_3 \angle x_1$;
$x_4 \angle i$, $x_4 \angle j$, $x_4 \angle x_1$

We now present the VDG (Fig. 10) of the program in Fig. 8, where white nodes indicate variant variables, grey nodes indicate LQIV variables, and names of virtual variables are written italic. Relations $\angle$ and $\triangleleft$ are shown as thin and thick edges, respectively. According to Fig. 10, LQIV variables and their invariant lengths can be easily derived. LQIV variables: {z_1, z_2, x_1, x_2, y_5, y_6, y_7, y_8, y_9, y_{10}, u_1, u_2}; $IL(z_1)=2$, $IL(z_2)=1$, $IL(x_1)=4$, $IL(x_2)=3$, $IL(y_5)=3$, $IL(y_6)=3$, $IL(y_7)=3$, $IL(y_8)=3$, $IL(y_9)=3$, $IL(y_{10})=3$, $IL(u_1)=3$, $IL(u_2)=2$. Because x_1 is a virtual variable, unfolding length $UL(o)=3$.

For any loop in the form of *while* $[ss_1]$ *e do* ss_2 *endwhile*, only the variables defined in ss_1 depend on the variables that will be statically assigned afterwards. For example, in Fig. 8, y_1 depends on y_0 and y_{12}, where y_{12} will be assigned afterwards.

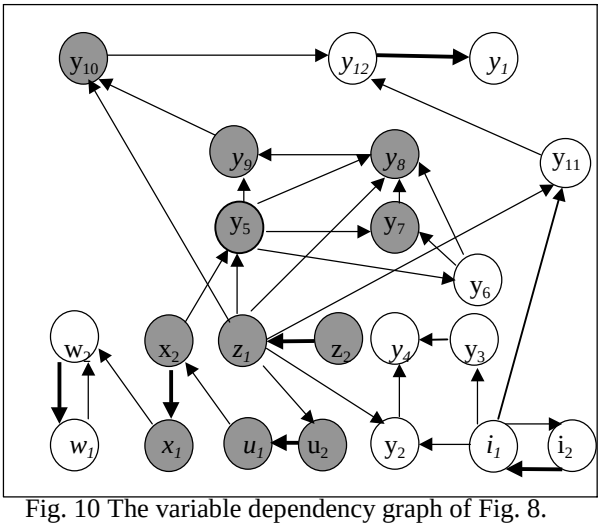


Fig. 10 The variable dependency graph of Fig. 8.

5. An Algorithm for LQIV Analysis

LQIV analysis consists of two phases. The first phase detects the dependency relations between the variables defined in a loop, the second phase finds all LQIV variables, computes their invariant lengths and the loop's unfolding length.

The second phase is the heart of the analysis. The algorithm for the second phase is given below. It is based on the classical algorithms by Warshall [22] and Floyd [8]. The time complexities of Warshall algorithm and Floyd algorithm are $O(n^3)$ in the worst case where *n* is the number of variables in the given loop. The first phase is not shown here; it can be done while parsing a program.

We assume that there are n variables (denoted with $v_1,\ldots,v_n$) in a loop o, the relations $\angle$ and $\lhd$ between $v_1,\ldots,v_n$ have been stored in a Boolean $n\times n$ matrix $R_{\angle}$ and an integer $n\times n$ matrix $R_{\lhd}$, respectively, where for any two variables v_i and v_j, if $v_i \angle v_j$ then $R_{\angle}[i,j]=1$ else $R_{\angle}[i,j]=0$ and if $v_i \lhd v_j$ then $R_{\lhd}[i,j]=1$ else $R_{\lhd}[i,j]=0$.

Algorithm (for LQIV analysis to singular loop):

Input: $R_{\angle}$, $R_{\lhd}$.

Output: the set (Lqivs) of all the LQIV variables, the invariant lengths of these LQIV variables, and the unfolding length (UL) of o.

Begin

{ Based on Warshall algorithm, computing all the LQIV variables assigned to inside loop o. }

$R1_{\angle} := R_{\angle}$;
for i=1 to n do
for j=1 to n do
if $R1_{\angle}[j,i]=1$
then for k=1 to n do
$R1_{\angle}[j,k] := R1_{\angle}[j,k] \vee R1_{\angle}[i,k]$;
endfor
endif
endfor endfor
$Lqivs := \{ i \mid \forall i_{(1\le i\le n)} \forall j_{(1\le j\le n)} . (R1_{\angle}[i,j]=1 \rightarrow R1_{\angle}[j,j]\neq 1)\}$;

{ Based on Floyd algorithm, computing the invariant lengths of all LQIV variables and the unfolding length of o. Remark: all the variables in Lqivs and all dependency relations among these variables can be viewed as a directed graph, where any node represents a LQIV variable and any edge between two nodes represents $\angle$ *or* $\lhd$ *relation between the two variables. Therefore, the invariant length of any LQIV variable is actually equivalent to the longest path ending in the variable, if the length of any* $\angle$ *edge is defined as 0 and that of any* $\lhd$ *edge is defined as 1. }*

$R1_{\lhd} := R_{\lhd}$;
for any $i \in Lqivs$ *do*
for any $j \in Lqivs$ *do*
for any $k \in Lqivs$ *do*
if $R1_{\angle}[j,i]=1 \wedge R1_{\angle}[i,k]=1 \wedge R1_{\angle}[j,k]=1$
then if $R1_{\lhd}[j,i]+R1_{\lhd}[i,k] > R1_{\lhd}[j,k]$
then $R1_{\lhd}[j,k] := R1_{\lhd}[j,i]+R1_{\lhd}[i,k]$; *endif;*
endif;
endfor endfor endfor
for any $i \in Lqivs$ *do* $IL[i]:=0$; *endfor;*
for any $i \in Lqivs$ *do*
$IL[i] := 1+\max\{ R1_{\lhd}[j,i] \mid j \in Lqivs \}$;
endfor;
$UL := \max\{ IL[i] \mid i \in Lqivs \wedge i \notin$ *virtual variables* $\}$;

End.

6. Loop Transformation

The purpose of loop transformation is to remove all LQIV variables from a given loop. The unfolding length computed by the algorithm in the previous section tells us how many iterations are needed before all LQIV variables become invariant. The loop transformation is based on decomposing the loop into two loops where the first loop iterates $UL(o)$ times to compute the values of the LQIV variables and the second loop is the remained loop after code motion. In the process of code motion, variable renaming is necessary.

6.1 A Note on Variable Renaming

When generating code for a loop represented in SSA form, we have to remove all assignments to virtual variables and to rename their uses correspondingly. Below we define how to rename variables.

For any virtual variable x, (assume it is defined as $x:=\phi(y,z)$.) all the uses of x are actually the uses of y or z. If $x:=\phi(y,z)$ is removed then all the uses of x will become undefined. Therefore, we must use same name for x, y and z so that all uses of x will naturally become the uses of y and z. We are going to discuss the problem in the following 2 cases.

CASE 1: y or z is also virtual variable.

When y or z is a virtual variable, we have the same situation as with x and its operands should also be renamed using the same name. The process continues recursively until no new virtual variables are met.

CASE 2: x is an operand of another virtual variable.

When x is an operand of another virtual variable (e.g., $w:=\phi(v,x)$), by the same reason, w, v and x should also be renamed using the same name. The process continues recursively until no new virtual variables are met.

We now define which variables should be renamed using the same name in a loop o. For this purpose, we define function RE which determines the set of variables that should be renamed using the same name.

$$RE_o(x) =_{def} \begin{cases} \{x\} & \text{if } x \text{ is not a virtual variable} \\ \{x\} \cup RE_o(x_1) \cup RE_o(x_2) \cup RE'_o(x) & \text{if } x \text{ is a virtual variable defined by } x:=\phi(x_1,x_2) \end{cases}$$

$$RE'_o(x) =_{def} \begin{cases} \{\ \} & \text{if } x \text{ is not an operand of another virtual variable} \\ \{y\} \cup RE_o(z) \cup RE'_o(y) & \text{if } x \text{ is an operand of another virtual variable } y \text{ defined by } y:=\phi(x,z) \text{ or } y:=\phi(z,x) \end{cases}$$

For example, in Fig. 8 there are five ϕ-*function* assignments to y: $y_1:=\phi(y_0, y_{12})$, $y_4:=\phi(y_2, y_3)$, $y_8:=\phi(y_6,$

y_7), y_9:=ϕ(y_5, y_8), y_{12}:=ϕ(y_{10}, y_{11}). The variables in set $RE_o(y_1)=RE_o(y_{12})=\{y_1, y_0, y_{12}, y_{10}, y_{11}\}$, $RE_o(y_4)=\{y_4, y_2, y_3\}$, $RE_o(y_8)=RE_o(y_9)=\{y_8, y_6, y_7, y_9, y_5\}$ should be renamed using the same name respectively (e.g., y, Y_2, Y_3). Similarly, the variables in each of the sets $\{x_1, x_0, x_2\}$, $\{i_1, i_0, i_2\}$, $\{u_1, u_0, u_2\}$, $\{z_1, z_0, z_2\}$, and $\{w_1, w_0, w_2\}$ should be renamed using the same names (e.g., x, i, u, z, w).

6.2 Loop Transformation

In transforming a loop, there are two methods. On the one hand, we can unfold a loop by adding unfolding length copies (strictly, not full copies) of the loop to residual loop. On the other hand, these copies can also be organized into a loop, and thus the residual code never contains more than two loops. We discuss here only loop decomposition in the following.

Code Motion by Loop Decomposition

Concretely, if there is a loop o in Fig. 11(a), then it can be transformed into two loops in Fig. 11(b). Remark: as a special case, if $UL(o)=1$ then a conditional can be used instead of the first loop.

while e *do* ss; *endwhile*;

(a). An original loop.

counter=0;
while $e \wedge (counter<UL(o))$ *do*
  ss_1;
  counter:=counter+1;
endwhile;
while e *do* ss_2; *endwhile*;

(b). The transformed residual code.

Fig. 11 The transformation based on LQIV code motion. Note that ss_1 indicates the full copy (except variable renaming) of ss, and ss_2 indicates the ss after LQIV code motion.

Obviously, the space taken up by the residual code will be less than twice the space taken up by the original loop, and thus has a modest impact on the overall code size. This is useful when the optimal unfolding length is large.

For example, we can transform the loop in Fig. 8 into the residual code in Fig. 12 where expression $10^3\times x^3+10^2\times x^2+10\times x+1$ can be moved outside the residual loop. A formalization for transformation is given in Appendix 2.

counter:=0;
while $i\le 0 \wedge counter<3$ *do*
  $w:=w+10^3\times x^3+10^2\times x^2+10\times x+1;$
  $x:=u+1;$
  if even(i) then $Y_2:=z+1;$ *else* $Y_2:=v;$ *endif;*
  $Y_3:=x;$
  if $z>0$ *then* $Y_3:=1;$
    if $z>v$ *then* $Y_3:=z+1;$ *endif;*
  endif;
  if $z>1000$ *then* $y:=Y_3+1;$ *else* $y:=i;$ *endif;*
  $u:=z-1;$
  $z:=v+1;$
  $i:=i+1;$
  counter:=counter+1;
endwhile
while $i\le 0$ *do*
  $w:=w+10^3\times x^3+10^2\times x^2+10\times x+1;$
  if even(i) then $Y_2:=z+1;$ *else* $Y_2:=v;$ *endif;*
  if $z>1000$ *then skip; else* $y:=i;$ *endif;*
  $i:=i+1;$
endwhile

Fig. 12 The residual code of the loop in Fig. 8.

Compared with the original loop, in the residual code, some new variables are introduced and thus the other parts of the program containing the original loop must be modified for consistence. For any variable (e.g., x) defined in an original loop, there must be an assignment like $x_1:=\phi(x_0,x_n)$ in the corresponding SSA form and only x_1 is visible to the outside of the loop, where, x_0 refers to the x defined out of the loop and x_n refers to the x defined finally in the loop. According to variable renaming, x_1, x_0, x_n will be renamed with same name. If we use x for the name, then the variable consistence problem above can be easily solved. However, the other new variables (e.g., Y_2 and Y_3) have to be declared.

To give an indication of the speedup, Table 2 shows the runtime of the original loop and the residual code, where $i=-10^8$ and $x=y=z=u=v=w=10^3$.

Table 2. The comparison of the runtime of the original loop in Fig. 8 and the residual code in Fig. 12.

System	Gateway E5250 Pentium II xeon×2 Speed: 450MHZ Memory: 1GB Visual C++ 6.0		Sun ULTRA 5 SunOS 5.6 Speed: 270MHZ Memory: 192MB Gcc 2.7	
	Runtime	Speed up	runtime	speed up
Source	9sec	1	52sec	1
Result	3sec	3	30sec	1.73

7. About Nested Loops

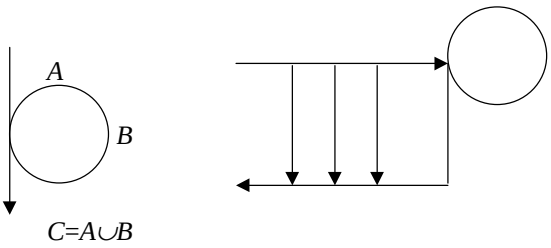
In the previous section we considered the treatment

of conditionals inside loops. Nested loops can be handled in a similar fashion. Here we give just a short description because the methods introduced so far can also handle the analysis of nested loops.

From the viewpoint of an outer loop *o*, only linear dependencies between the variables of an inner loop *q* are interesting. A variable which is variant in *q* may well be quasi-invariant in *o* (e.g., an inductive variable of *q* is variant in *q*, but can be quasi-invariant with respect to *o*). This means we are not interested in cyclic dependencies local to inner loops when analysing *o*. Conceptually, this can be done by building a separate VDG for each loop *o* from a program where all inner loops of *o* are viewed as conditionals with empty *else*-branch (CASE 2 in Section 4.), thereby avoiding the representation of cycles local to the inner loops. In practice, more efficient construction methods may exist.

8. Discussion of Applications

A program analysis has to compute a safe approximation of information about the dynamic behavior of a program. To compute such information an analysis must consider all possible computation paths and determine a safe approximation of the collected information at each point where control flow meets (usually done by a least upper bound/greatest lower bound operation). This situation is illustrated in Fig. 13(a) for the program in Fig. 2 where information *A* and *B* is merged to form a safe approximation *C* at the join point. As long as information is propagated along code sequences without join points, no such approximation occurs.



(a). Program in Fig. 2. (b). Program in Fig. 3

Fig. 13 The control flow before and after LQIV code motion.

In comparison, the control flow of program Fig. 3 is shown in Fig. 13(b). It should be clear that a program analysis of the loop Fig. 3 will produce results that are at least as good as those produced for the loop in Fig. 2, but potentially better. All loop quasi-invariant computations have been moved outside the residual loop; they are now invariant inside the loop. This means any static analysis can in principle benefit from the loop optimization.

To illustrate the effect on a concrete program analysis, consider offline partial evaluation, a well-known program transformation method based on constant propagation which is controlled by a separate binding-time analysis (BTA). (We assume the reader is familiar with the principles of partial evaluation; for more details see the book [13].)

Let us consider a pointwise BTA where the information computed by the analysis ("static", "dynamic") is merged at each program point. In addition, let the BTA dynamize static values under dynamic control as done in the TEMPO specializer for C [12]. In fact, the same technique is used when inserting explicators in online partial evaluation (cf, [15]).

To compare the effect of loop quasi-invariant code motion on the analysis, we specialize the programs in Fig. 2 and Fig. 3. For simplicity we shall treat functions f, g, g_1, g_2, g_3 as primitive operators at specialization time.

Consider specializing the programs with respect to the static values $c_1=c_2=c_3=true$. The residual programs are shown in Fig. 14(a) and (b) respectively. The difference should be obvious. In particular, observe that static value of $g=true$ is inlined in the residual loop Fig. 14(b) (possibly triggering further static computations in a loop). This is possible because all computations of LQIV variables have been moved out of the loop. Since all LQIV variables are invariant in the loop, their static values be treated as constants inside the loop even though the BTA strategy dynamizes static values under dynamic control.

```
while t<TMAX do
  x3:=g3(x2,true);
  x2:=g2(x1,true);
  x1:=true;
  y:=f(g(x1,x2,x3),y);
  t:=t+1;
endwhile
```

(a). The result specializing the source program in Fig. 2.

```
if t<TMAX
then x3:=g3(x2,true);
     x2:=g2(x1,true);
     y:=f(g(true,x2,x3),y);
       t:=t+1;
       if t<TMAX
       then x3:=g3(x2,true);
            y:=f(g(true,true,x3),y);
            t:=t+1;
            if t<TMAX
```

```
then y:=f(true,y);
        t:=t+1;
        while t<TMAX do
            y:=f(true,y);
            t:=t+1;
        endwhile
endif endif endif
```

(b). The result specializing the source program in Fig.3

Fig. 14 as described in this section.

To conclude, our technique may not result in dramatic speedups, but has the potential to increase the accuracy of program analyses and to trigger stronger program optimizations which is of central importance in almost any kind of program transformation.

9. Related Work

Code motion is a well-known technique in compiler optimization [1]. Code motion and loop-invariance have many uses in the optimization of source programs written in high-level languages as well as target programs written in assembly language. (e.g., code generated for indexing multi-dimensional arrays.) A comprehensive survey of different loop transformations and other data-flow based compiler optimizations can be found in [3]. However, none of them considers the combination of code motion, unfolding and loop quasi-invariance.

A recently developed transformation is partial redundancy elimination (PRE) which is a global optimization technique generalizing the removal of common subexpressions and loop-invariant computations. Originally, implementation of PRE failed to completely remove the redundancies. The recently developed PRE algorithms based on control flow restructuring [4, 11, 19, 20] can achieve a complete PRE and are capable of eliminating loop-quasi invariant code. However, these techniques are usually exponential in the worst case and the resulting code duplication may cause code size explosion.

Many optimization techniques can be formalized conveniently using single static assignments, including the elimination of partial redundancies [17], constant propagation [14, 21], and code motion [7]. We followed the same approach to express our loop optimization technique.

The notion of quasi-invariance grew out of work on automatic program transformation, in particular partial evaluation where the optimization of loops is of central importance (e.g., [2, 10, 12, 15]). Our technique statically determines a finite fixed point of computations induced by assignments, loops and conditionals and computes the optimal unfolding length (e.g., to make maximal use of known information while ensuring termination as shown in Section 8). LQIV code motion may support other static termination analyses, e.g., techniques for detecting static bounded variation [10], and other generalization techniques.

10. Conclusion and Future Work

In this paper we introduced LQIV code motion, which is based on a static analysis for loop quasi-invariance. The notion of loop quasi-invariance is similar to the traditional notion of loop-invariance but allows for a finite number of iterations before computations in a loop become invariant.

Our analysis determines the optimal unfolding length needed to remove all quasi-invariant computations from a loop while avoiding over-unfolding. The residual loop becomes smaller and faster. Our technique is well-suited as supporting transformation in compilers, partial evaluators, and other program transformers. It may not result in dramatic speedups in large practical applications, but has the potential to increase the accuracy of program analyses and to trigger stronger program optimizations which is of central importance in almost any kind of program transformation. The algorithms presented in this paper use the infrastructure already present in many compilers, such as control flow graphs and single static assignments. Thus they do not require fundamental changes to existing systems. To the best of our knowledge more efficient quasi-invariant code motion has not been considered earlier for program optimization.

It was shown [14] that standard constant folding can provide useful speed up for large hand-written programs, so loop quasi-invariance optimization may also be beneficial for practical applications written by hand. However, more should be known about the effect of quasi-invariance code motion on optimizing compilers. This and the application of our technique to larger practical programs will be a topic for future work.

Reference

[1] Aho A. V., Sethi R., Ullman J. D., Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[2] Andersen L. O., Program analysis and specialization for the C programming language. Ph.D.Thesis, DIKU Report 94/19. Dept. of Computer Science, University of Copenhagen,

1994.
[3] Bacon D. F., Graham S. L., Compiler transformations for high-performance computing, ACM Computing Surveys, Vol.26, No.4, 345-420, December 1994.
[4] Bodik R., Gupta R., Soffa M. L., Complete removal of redundant expressions, Proceeding of the ACM Conference on Programming Language Design and Implementation, 1-14, ACM Press 1998.
[5] Bulyonkov M. A., Kochetov D. V., Practical aspects of specialization of Algol-like programs, Danvy O., Glück R., Thiemann P. (eds.), Partial Evaluation. Proceedings. LNCS, Vol. 1110, 17-32, Springer-Verlag 1996.
[6] Cytron R., Ferrante J., Efficiently computing static single assignment form and the control dependence graph, ACM TOPLAS, Vol. 13, No. 4, 451-490, October, 1991.
[7] Cytron R., Lowry A., Zadeck F. K., Code motion of control structures in high-level languages, Conference Record of the 13th ACM Symposium on Principle of Programming Languages, 70-85, ACM Press 1986.
[8] Floyd R. W., Algorithm 97: shortest path, Comm. ACM 5:6, 345, 1962.
[9] Glenstrup A. J., Jones N. D., BTA algorithms to ensure termination of off-line partial evaluation, Bjørner D., Broy M., Pottosin I. V. (eds.), Perspectives of System Informatics. Proceedings. LNCS, Vol. 1181, 273-284, Springer-Verlag 1996.
[10] Glenstrup A., Makholm H., Secher J. P., C-Mix: specialization of C programs, Hatcliff J., Mogensen T., Thiemann P. (eds.), Partial Evaluation: Practice and Theory, LNCS, Vol. 1706, 108-153, Springer-Verlag 1999.
[11] Gupta R., Berson D. A., Fang J. Z., Path profile guided partial redundancy elimination using speculaton, IEEE International Conference on Computer Languages, 230-239, IEEE Society Press 1998.
[12] Hornof L., Noyé J., Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity, Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 63-73, ACM Press 1997.
[13] Jones N. D., Gomard C. K., Sestoft P., Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.
[14] Metzger R., Stroud S., Interprocedural constant propagation: an empirical study, ACM Letters on Programming Languages and Systems, 2(1-4): 213-232, 1993.
[15] Meyer U., Techniques for partial evaluation of imperative language, Symposium on Partial Evaluation and Semantics-Based Program Manipulation, (Sigplan Notices, vol. 26, no.9, September 1991), 94-105, ACM Press,1991.
[16] Nielson F., Nielson H. R., Hankin C., Principles of Program Analysis. Springer-Verlag 1999.
[17] Rosen B. K., Wegman M. N., Zadeck F. K., Global value numbers and redundant computations. Conference Record of the 15th ACM Symposium on Principles of Programming Languages, 12-27, ACM Press, 1988.
[18] Song L. T., Futamura Y., Quasi-invariant variable and loop unfolding, The 15th National Conference of Software Society of Japan, B6-2, 221-224, Sept. 1998.
[19] Steffen B., Property oriented expansion, Symposium on Static Analysis, LNCS 1145, 22-41, Springer-Verlag 1996.
[20] Steffen B., Knoop J., Rüthing O., The value flow graph: a program representation for optimal program transformations, Jones N. D. (ed.) ESOP'90, LNCS 432, 389-405, Springer-Verlag 1990.
[21] Warshall S., A theorem on Boolean matrices, Journal of the ACM, 9(1): 11-12, January 1962.
[22] Wegman M. N., Zadeck F. K., Constant propagation with conditional branches, ACM TOPLAS, Vol. 13, No. 2, 181-210, April, 1991.
[23] Zima H., Chapman B., Supercompiler for Parallel and Vector Computers, Frontier, Series, ACM Press, 1990.

Appendix 1 (The definitions of g_1, g_2, g_3, g, f, t and T)

$g_1(c_1)=c_1$,
$g_2(x_1,c_2)=(\neg x_1 \wedge c_2)\vee(x_1 \wedge c_2)$,
$g_3(x_1,x_2,c_3)=(\neg x_1 \wedge x_2 \wedge c_3)\vee(x_1 \wedge \neg x_2 \wedge c_3)\vee(x_1 \wedge x_2 \wedge c_3)$,
$g(x_1,x_2,x_3)=(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$,
$f(x,y)=x \wedge y$,
$t=1$,
$T=10^8$.

Appendix 2 (The algorithm for transformation)

Let us assume: there is a loop of the form *<u>while</u> [ss_1] e <u>do</u> ss_2 <u>endwhile</u>*, R indicates the store memorizing the new names of the variables defined in the loop, L indicates the set of all the LQIV variables, U indicates the unfolding length of the loop. Then the algorithm of loop transformation is given by the rules as follow:

Loop:

$$\frac{(R)\vdash_E : e \Rightarrow e' \quad (R)\vdash_{SS1} : ss_2 \Rightarrow ss_2' \quad (L,R)\vdash_{SS2} : ss_2 \Rightarrow ss_2''}{(U)\vdash_{Loop} : \underline{while}\ [ss_1]\ e\ \underline{do}\ ss_2\ \underline{endwhile} \Rightarrow \underline{counter := 0;}\ \underline{while}\ e' \wedge counter < UL\ \underline{do}\ ss_2';\ count := counter + 1\ ;\ \underline{endwhile}\ \underline{while}\ e'\ \underline{do}\ ss_2''\ \underline{endwhile;}}$$

Statements:

$$\frac{(R)\vdash_{S1} : s \Rightarrow s' \quad (R)\vdash_{SS1} : ss \Rightarrow ss'}{(R)\vdash_{SS1} : s; ss \Rightarrow s'; ss'}$$

$$\frac{(L,R)\vdash_{S2} : s \Rightarrow s' \quad (L,R)\vdash_{SS2} : ss \Rightarrow ss'}{(L,R)\vdash_{SS2} : s; ss \Rightarrow s'; ss'}$$

$$\frac{}{(R)\vdash_{S1} : v := \ominus(\ldots) \Rightarrow skip}$$

$$\frac{(R)\vdash_E : e \Rightarrow e'}{(R)\vdash_{S1} : v := e \Rightarrow R(v) := e'}$$

$$\frac{}{(L,R)\vdash_{S2} : v := \ominus(\ldots) \Rightarrow skip}$$

$$\frac{v \in L}{(L,R)\vdash_{S2} : v := e \Rightarrow skip}$$

$$\frac{v \notin L \quad (R)\vdash_E : e \Rightarrow e'}{(L,R)\vdash_{S2} : v := e \Rightarrow R(v) := e'}$$

Conditional:

$$\frac{(R)\vdash_E : e \Rightarrow e' \quad (R)\vdash_{SS1} : ss_1 \Rightarrow ss_1' \quad (R)\vdash_{SS1} : ss_2 \Rightarrow ss_2'}{(R)\vdash_{S1} : \underline{if}\ e\ \underline{then}\ ss_1\ \underline{else}\ ss_2\ \underline{endif} \Rightarrow \underline{if}\ e'\ \underline{then}\ ss_1'\ \underline{else}\ ss_2'\ \underline{endif}}$$

$$\frac{(R)\vdash_E : e \Rightarrow e' \quad (L,R)\vdash_{SS1} : ss_1 \Rightarrow ss_1' \quad (L,R)\vdash_{SS2} : ss_2 \Rightarrow ss_2'}{(L,R)\vdash_{S1} : \underline{if}\ e\ \underline{then}\ ss_1\ \underline{else}\ ss_2\ \underline{endif} \Rightarrow \underline{if}\ e'\ \underline{then}\ ss_1'\ \underline{else}\ ss_2'\ \underline{endif}}$$

Expression:

$$\frac{}{(R)\vdash_E : constant \Rightarrow constant}$$

$$\frac{}{(R)\vdash_E : v \Rightarrow R(v)}$$

$$\frac{(R)\vdash_E : e_i \Rightarrow e_i'}{(R)\vdash_E : p(e_1,\ldots,e_n) \Rightarrow p(e_1',\ldots,e_n')}$$