An Unfolding-Based Loop Optimization Technique

Litong Song[1], Krishna Kavi[1], and Ron Cytron[2]

[1] Department of Computer Science,
University of North Texas, Denton, Texas, 76203, USA
{slt, kavi}@cs.unt.edu
[2] Department of Computer Science and Engineering
Washington University, St. Louis, MO 63130, USA
{cytron}@cs.wustl.edu

Abstract. Loops in programs are the source of many optimizations for improving program performance, particularly on modern high-performance architectures as well as vector and multithreaded systems. Techniques such as loop invariant code motion, loop unrolling and loop peeling have demonstrated their utility in compiler optimizations. However, many of these techniques can only be used in very limited cases when the loops are "well-structured" and easy to analyze. For instance, loop invariant code motion works only when invariant code is inside loops; loop unrolling and loop peeling work effectively when the array references are either constants or affine functions of index variable. It is our contention that there are many opportunities overlooked by limiting the optimizations to well structured loops. In many cases, even "badly-structured" loops may be transformed into well structured loops. As a case in point, we show how some loop-dependent code can be transformed into loop-invariant code by transforming the loops. Our technique described in this paper relies on unfolding the loop for several initial iterations such that more opportunities may be exposed for many other existing compiler optimization techniques such as loop invariant code motion, loop peeling, loop unrolling and so on.

1 Introduction

Loops in programs are the source of many optimizations for improving program performance, particularly on modern high-performance architectures as well as vector and multithreaded systems. Techniques such as loop invariant code motion, loop peeling and loop unrolling have demonstrated their utility among in compiler optimizations. However, many of these techniques can only be used in very limited cases when the loops are "well-structured" and easy to analyze. For instance, loop invariant code motion works only when invariant code is inside loops; loop unrolling and loop peeling work effectively when the loop indices and array references are either constant or affine functions. Let us first give a brief review on a few common loop optimization techniques such as loop invariant code motion, loop unrolling and loop peeling, and discuss the limitations of these techniques.

1.1 Reviews of A Few Loop Optimization Techniques

Loop invariant code motion is a well-known loop transformation technique. 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 performance. For instance, the evaluation of expression $a \times 100$ is loop invariant in Fig. 1(a); Fig. 1(b) shows a more efficient version of the loop where the loop invariant code has been removed from the loop.

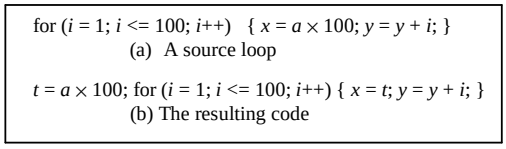

Fig. 1. An example for loop invariant code motion

Modern computer systems exploit both instruction level parallelism (ILP) and thread (or task) level parallelism (TLP). Superscalar and VLIW systems rely on ILP while multi-threaded and multiprocessor systems rely on TLP. In order to fully benefit from ILP or TLP, compilers must perform complex analyses to identify and schedule code for the architecture. Typically compilers focus on loops for finding parallelism in programs [26], [27]. Sometimes it is necessary to rewrite (or reformat) loops such that loop iterations become independent of each other, permitting parallelism. Loop peeling is one such technique [3], [15], [21]. When a loop is peeled, a small number of early iterations are removed from the loop body and executed separately. The main purpose of this technique is for removing dependencies created by the early iterations on the remaining iterations, thereby enabling parallelization.

The loop in Fig. 2(a) is not parallelizable because of a flow dependence between iteration $i = 1$ and iterations $i = 2 .. n$. Peeling the first iteration makes the remaining iterations fully parallel, as shown in Fig. 2(b). Using vector notation, the loop in Fig. 2(b) can be rewritten as: $a(2: n) = a(1) + b(2: n)$. That is to say, $n - 1$ assignments in $n - 1$ iterations of the loop can be executed in parallel.

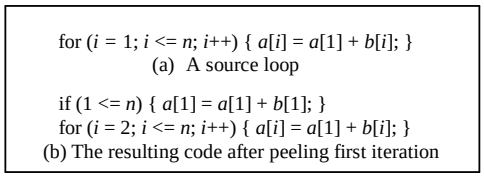

Fig. 2. The first example for loop peeling

The loop in Fig. 3(a) is not parallelizable because variable *wrap* is neither a constant nor a linear function of inductive and index variable *i*. Peeling off the first iteration allows the rest of loop to be vectorizable, as shown in Fig. 3(b). The loop in Fig. 3(b) can be rewritten as: $a(2: n) = b(2: n) + b(1: n\text{-}1)$.

Loop unrolling is a technique, which replicates the body of a loop a number of times called the unrolling factor *u* and iterates by step *u* instead of step 1. It is a fundamental technique for generating efficient instructions required to exploit ILP and

TLP. Loop unrolling can improve the performance by (i) reducing loop overhead; (ii) increasing instruction level parallelism; (iii) improving register, data cache, or TLB locality. Fig. 4 shows an example of loop unrolling, Loop overhead is cut in a second because one additional iteration is performed before the test and branch at the end of the loop. Instruction parallelism is increased because the first and second assignments can be executed on pipeline. If array elements are assigned to registers, register locality will improve because $a[i]$ is used twice in the loop body, reducing the number of loads per iteration.

```
for (i = 1; i <= n; i++) { a[i] = b[i] + b[wrap]; wrap = i; }
```
(a) A source loop

```
if (1 <= n) { a[1] = b[1] + b[wrap]; wrap = i; }
for (i = 2; i <= n; i++) { a[i] = b[i] + b[i-1]; }
```
(b) The resulting code after peeling first iteration

Fig. 3. The second example for loop peeling

```
for (i = 2; i <= n; i++) { a[i] = a[i-2] + b[i]; }
```
(a) A source loop

```
for (i = 2; i <= n-1; i = i+2) { a[i] = a[i-2] + b[i]; a[i+1] = a[i-1] + b[i+1]; }
if (mod(n-2, 2) == 1) { a[n] = a[n-2] + b[n]; }
```
(b) The resulting code after loop unrolling

Fig. 4. An example of loop unrolling

1.2 Issues

As we mentioned previously, loop invariant code motion, loop peeling and loop unrolling are all very practical and important compiler optimization techniques for today's architectures. Nevertheless, these techniques are only suitable for well-structured loops, which are relatively easy to analyze. For loop invariant code motion, it works only when there are clearly and easily identifiable invariant code inside loops; for loop unrolling and loop peeling, they usually work when subscripts of array references are constants or affine functions. In many practical programs, loops are not well-structured; but in some cases, these loops may be quasi well-structured ones. That is to say, they may be converted into well-structured. For instance, in the loop of Fig. 5(a), there is only one invariant expression $b \times c$. If we unfold the loop twice, however, we can get the resulting code in Fig. 5(b), which is much more efficient than the source loop. This is because: (i) variables x and y become invariant variables in the resulting loop, so that assignments $x = y + a$ and $y = b \times c$ can be removed from the remaining loop; (ii) expression $x \times y$ and $x > d$ are invariant expressions in the remaining loop so they can be hoisted outside the remaining loop, which can actually be done by the conventional loop invariant code motion; (iii) because expression $x > d$ is invariant during the dynamic execution of the remaining loop, it will improve the branch predication and significantly decrease branch misses of the conditional contained in the remaining loop. This example shows that an effective transformation of badly structured loops is possible and desirable.

```
while (i <= n) { x = y + a; y = b ×  c; if (x > d) i = i + x ×  y; else i = i + 1; }
```
(a) A source loop

```
if (i <= n) { x = y + a; y = b ×  c; if (x > d) i = i + x ×  y; else i = i + 1; }
if (i <= n) { x = y + a; if (x > d) i = i + x ×  y; else i = i + 1; }
while (i <= n) { if (x > d) i = i + x ×  y; else i = i + 1; }
```
(b) The resulting code after unfolding two iterations

Fig. 5. Loop quasi-invariant code motion

For the loop in Fig. 6(a), in the two assignments $a[i] = b[i] + b[j]$ and $c[i] = c[j] \times b[i]$, j and *wrap* are not constants or affine functions of index variable i, so we have no way to directly parallelize any of them, and we can not even unroll the loop since we do not know what is going on for loop-carried dependences. If peeling or unfolding the loop for two iterations, however, the remaining loop in Fig. 6(b) is very suitable for parallelization and loop unrolling. Statement $a[i] = b[i] + b[i\text{-}2]$ can be parallelized to be $a[3: n] = b[3: n] + b[1: n\text{-}2]$, and statement $c[i] = c[i\text{-}2] \times b[i]$ can be unrolled to be $c[i] = c[i\text{-}2] \times b[i]$; $c[i+1] = c[i\text{-}1] \times b[i+1]$; such that the two statements can be executed in parallel since there is no loop-carried dependence among them. Thus, some pre-optimizations or transformations based on loop unfolding may be very useful and lead to the application of conventional compiler optimization techniques.

```
for (i = 1; i <= n; i++) { a[i] = b[i] + b[j]; c[i] = c[j] ×  b[i]; j = i – wrap; wrap = 1; }
```
(a) A source loop

```
if (1 <= n) { a[1] = b[1] + b[j]; c[1] = c[j] ×  b[1]; j = 1 – wrap; wrap = 1; }
if (2 <= n) { a[2] = b[2] + b[j]; c[2] = c[j] ×  b[2]; j = 1; }
for (i = 3; i <= n; i++) { a[i] = b[i] + b[i-2]; c[i] = c[i-2] ×  b[i]; }
```
(b) The resulting code after peeling two iterations

Fig. 6. An example for loop peeling and loop unrolling

In this paper we present a technique that is based on loop dependence analysis, so that traditional optimization techniques can benefit from it. In particular, our goal is to find a general and systematic way for pre-optimizations of using loop unfolding to remove anti-dependences as much as possible.

2 Preliminaries

This section provides the background necessary for the rest of the paper, including a simple language we will use to describe our loop optimization technique and the well-known static single assignment (SSA) form.

2.1 DO-language

For the purpose of describing our technique, we first introduce a simple imperative language, shown in Fig. 7; the semantics is similar to C. For the sake of simplifying

the presentation, we assume a call-by-value semantics for function parameters, assume freedom of side effects, and we treat all functions as primitive operations.

Sts	::= *St* | *St*; *Sts*
St	::= *Ass* | *Cond* | *Loop* | *Call*
Ass	::= *Var* = *Exp*
Cond	::= if (*Exp*) { *Sts* } else { *Sts* }
Loop	::= for (*Var* = *Exp*; *Exp*; *Var* = *Var* + *Exp*) { *Sts* } | while (*Exp*) do { *Sts* }
Call	::= *f*(*Exp**)
Exp	::= *Var* | *Const* | *Op*(*Exp**) | *Call*
Op	::= + | – | × | / | > | < | <= | >= | = | ! |

Fig. 7. The syntax of the DO-language

2.2 Static Single Assignment

Variables inside a loop may be modified for multiple times. In order to perform dependency analyses, it is necessary to distinguish the modifications. Here, we make use of the well-known static single assignment (SSA) [10] for this purpose. SSA form is a program representation in which every variable is assigned only once, and every use of the variable is defined by that assignment. Most compilers use SSA representations for performing optimizations. Here we use the term to refer to variables as ϕ-*variables* assigned by ϕ-function. An efficient algorithm that converts a program into SSA form with linear time complexity (in term of the size of the original program) was presented in [9].

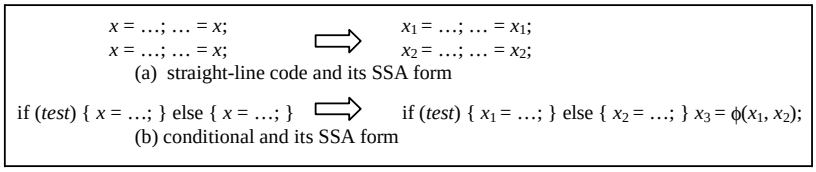

Fig. 8. SSA form transformation

3 Quasi Invariant and Quasi Index Variables

The invariant variables of a loop are those variables whose values are invariant in all the iterations of the loop. The index variable of a loop is a variable whose values in successive iterations form an arithmetic progression. Index variables are often used in array subscripts. Here, we present four notions:

- **Quasi invariant variable**. *A variable that is not invariant inside a loop but will become invariant after a small number of iterations of the loop*.
- **Quasi index variable**. *A variable that is not an index variable but will become equal to an affine function of the index variable after a small number of iterations of the loop*.

- **Unfolding factor of quasi invariant variable**. *If a quasi invariant variable becomes invariant after at least n iterations of a loop, n is referred to as the unfolding factor of the variable.*
- **Unfolding factor of quasi index variable**. *If a quasi index variable becomes an affine function of the index variable after at least n iterations of a loop, n is referred to as the unfolding factor of the variable*.

For instance, in Fig. 5, *x* and *y* are quasi invariant variables, and their unfolding factors are 2 and 1, respectively; in Fig. 6, *wrap* is a quasi invariant variable but *j* is a quasi index variable, and their unfolding factors are 1 and 2, respectively. Now, we face two issues: (i) identifying quasi invariant and quasi index variables; (ii) calculating the unfolding factors of these variables.

4 Variable Dependences

Compiler usually relies on both control and data dependence analyses for performing optimizations [5], [27]. These dependencies relate to those among statements. In our case, we only rely on dependencies among variables. We recognize two forms of data dependences: true data dependence, anti-data dependence, and two forms of control dependences: true control dependence, anti-control dependence.

- **True data dependence**. *The first statement stores into variable x that is later read by the second statement*:
  S_1: $x = \ldots$; S_2: $y = \ldots x$;
  We say y has a true data dependence to x, and denote the dependence as $y\ \delta_d\ x$.
- **Anti-data dependence**. *The first statement reads x into which the second statement later stores*:
  S_1: $y = \ldots x$; S_2: $x = \ldots$;
  We say y has an anti-data dependence to x, and denote the dependence as $y\ \delta_d^-\ x$.
- **True control dependence**. *The first statement stores into variable x that is later read by the test of second statement* (*conditional*):
  S_1: $x = \ldots$; S_2: if $(\ldots x)$ $y = \ldots$; else $y = \ldots$;
- **Anti-control dependence**. *The test of first statement* (*conditional*) *reads x into which the second statement later stores*:
  S_1: if $(\ldots x)$ $y = \ldots$ else $y = \ldots$; S_2: $x = \ldots$;
  We say y has an anti-control dependence to x and denote it as $y\ \delta_c^-\ x$.

According to the definitions above, the variable dependences in Fig. 5(a) and Fig. 6(a) should be: $x\ \delta_d^-\ y$, $x\ \delta_c\ i$, $i\ \delta_d\ i$, $j\ \delta_d^-\ wrap$, $j\ \delta_d\ i$, $i\ \delta_d\ i$
Note that we only discuss the dependences between scalar variables here.

5 An Extension of Control Dependences

In Sect. 4 we presented two general notions for control dependences. In this section, we present special cases of conditionals to elaborate on control dependences. Variable assignments inside conditionals can be distinguished into two cases:

- *A variable is assigned inside both then-part and else-part of a conditional*:

for (i = 1; i <= n ; i++) { if (*test*) { $x_1 = e_1$; } else { $x_2 = e_2$; } $x_3 = \phi(x_1, x_2)$; }

The assignment to a quasi invariant variable can be removed after the variable becomes invariant, and the symbolic value (an affine function) of a quasi index variable might be substituted for references to the variable after it is equal to an affine function. Whether $x_1 = e_1$ or $x_2 = e_2$ can be removed or not, is dependent on not only e_1 or e_2 but also *test*. If *test* is variant then neither $x_1 = e_1$ nor $x_2 = e_2$ can be removed even if e_1 or e_2 may be invariant. Otherwise, x_3 might be assigned to an incorrect value. By contrast, if *test* is invariant then either $x_1 = e_1$ or $x_2 = e_2$ can be removed as long as e_1 or e_2 is invariant. This is because the selection of the value of x_3 is invariant inside the remaining loop.

- *A variable is assigned both inside one branch of a conditional and outside the conditional*:

  for (i = 1; i <= n ; i++) { $x_1 = e_1$; if (*test*) { $x_2 = e_2$; } $x_3 = \phi(x_1, x_2)$; }

  Similar to case 1, both x_1 and x_2 are control dependent on *test*. In addition, we distinguish between two cases as below:

  - *There exist references to* x_1.

    Because the value of *test* is unknown, $x_1 = e_1$ can not be removed even if the *test* is invariant. Note that x_1, x_2 and x_3 will be renamed to be a same name in resulting program, which will be described in Sect. 8. Accordingly, $x_2 = e_2$ can not be removed either. If x_1 and x_2 are ϕ-variables, their operands can not be removed either, and thus a recursive processing is needed to determine which assignments can not be removed from the resulting loop. Assuming that we use γ to denote the closure of this kind of variables, and σ to denote the variables already handled, γ will be defined as follows:

$$\gamma(x)_\sigma = \begin{cases} \sigma & \text{if } x \in \sigma \\ \sigma \cup \{x\} & \text{if } x \notin \sigma \wedge x \notin \phi\text{-variables} \\ \gamma(x_1)_{\sigma \cup \{x\}} \cup \gamma(x_2)_{\sigma \cup \{x\}} & \text{if } x \notin \sigma \wedge x = \phi(x_1, x_2) \end{cases}$$

  - *There exists no reference to* x_1.

    Because $x_1 = e_1$ is outside the conditional, $x_2 = e_2$ can be removed only when assignment $x_1 = e_1$ is removed (otherwise x_3 will be always equal to x_1). The special dependence between x_1 and x_2 is actually an *ad hoc* true control dependence, which is still denoted by $x_2 \, \delta_c \, x_1$.

After the analysis of control dependences, we need to collect all the related dependences introduced by ϕ-functions. A ϕ-function is temporarily introduced only for static analysis and it will be removed in resulting programs, so any control dependence introduced by a ϕ-variable is actually a dependence introduced by the operand variables of the ϕ-variable. This is a recursive process and a closure should be computed. Assuming that there exists a control dependence denoted as $x_1 \, \delta_c \, x_2$, function φ is used to denote the closure, and σ is used to denote the dependences already handled, function φ can be defined as follows:

$$\varphi(x)_\sigma = \begin{cases} \sigma & \text{if } x \, \delta_c \, y \wedge (x \, \delta_c \, y) \in \sigma \\ \sigma \cup \{(x \, \delta_c \, y)\} & \text{if } x \, \delta_c \, y \wedge (x \, \delta_c \, y) \notin \sigma \wedge y \notin \phi\text{-variables} \\ \varphi(x, y_1)_{\sigma \cup \{(x \, \delta c \, y)\}} \cup \varphi(x, y_2)_{\sigma \cup \{(x \, \delta c \, y)\}} & \text{if } x \, \delta_c \, y \wedge (x \, \delta_c \, y) \notin \sigma \wedge y = \phi(y_1, y_2) \end{cases}$$

For instance, suppose we have the following program segment inside a loop:

$x_1 = 1$; if ($i > j$) { if ($k > 5$) { $x_2 = 2$; } else { $x_3 = 3$; } $x_4 = \phi(x_2, x_3)$; } $x_5 = \phi(x_1, x_4)$;

We can compute the following dependences: $x_1\ \delta_c\ i$, $x_1\ \delta_c\ j$, $x_2\ \delta_c\ i$, $x_2\ \delta_c\ j$, $x_2\ \delta_c\ k$, $x_2\ \delta_c\ x_1$, $x_3\ \delta_c\ i$, $x_3\ \delta_c\ j$, $x_3\ \delta_c\ k$, $x_3\ \delta_c\ x_1$, $x_4\ \delta_c\ i$, $x_4\ \delta_c\ j$, $x_4\ \delta_c\ x_1$

6 Dependence Relation Graph

Based on the two types of data dependences and two types of control dependences, we can construct a directed graph called dependence relation graph.

Definition 1 (**Dependence relation graph**). *The* dependence relation graph (DRG) *of a loop is a directed graph* (V, E), *where*

V = { *x* | *x is a variable modified inside the loop*};

E = { *a directed real thin line from x to y* | $y\ \delta_d\ x$ }∪{ *a directed real bold line from x to y* | $y\ \delta_c\ x$ }∪{ *a directed dotted thin line from x to y* | $y\ \delta_d^-\ x$ }∪{ *a directed dotted bold line from x to y* | $y\ \delta_c^-\ x$ }

```
for (i = 1; i <= n; i++) {
   a[i] = p[x] + q[y+k];
   if (odd(t)) { w = i – 1; b[i] = b[w] + c[z]; } else { w = i;  k = d;  b[i] = b[w] + c[z]; }
   t = j + z;  z = 2;  x = y;  y = i + 1;
}
```

(a) A source loop

```
for [ x1 = φ(x0, x2);  t1 = φ(t0, t2);  z1 = φ(z0, z2); y1 = φ(y0, y2);  k1 = φ(k0, k3);  w1 = φ(w0, w4); ]
     (i = 1; i <= n; i++) {
   a[i] = p[x1] + q[y1+k1];
   if (odd(t1)) { w2 = i – 1; b[i] = b[w2] + c[z1]; } else { w3 = i; k2 = d; b[i] = b[w3] + c[z1]; }
   w4 = φ(w2, w3); k3 = φ(k1, k2); t2 = j + z1; z2 = 2; x2 = y1; y2 = i + 1;
}
```

(b) The corresponding SSA form

Fig. 9. An example for SSA form conversion

For instance, assuming that we have a program segment shown in Fig. 9, the DRG for this program is shown in Fig. 10. Here, the semantics of loop for [*Sts*] (*Var* = *Exp*; *Exp*; *Var* = *Var* + *Exp*) { *Sts* } means that statements in [*Sts*] will be executed before the evaluation of loop test. Note that this intermediate form is only used for static analysis and it will be converted back to original form after optimization.

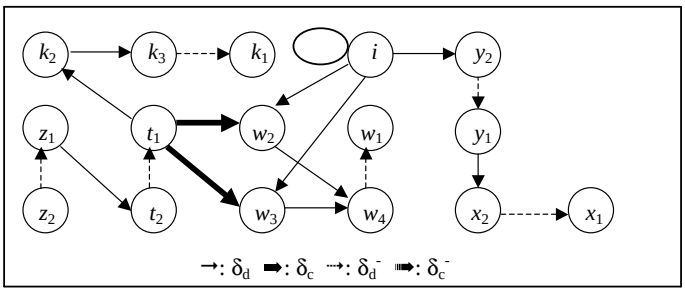

Fig. 10. The DRG of the loop in Fig. 9

7 Identifying Quasi Invariant/Index Variables and Computing Their Unfolding Factors

In Sect. 3 we defined quasi invariant variables, quasi index variables and their unfolding factors. Using dependence relation graphs, we can identify quasi invariant variables and quasi index variables, and efficiently compute their unfolding factors.

7.1 Quasi Invariant Variables and Unfolding Factors

- **Quasi invariant variable**. *For any vertex on the DRG of a loop, if among all the paths ending in this vertex, there is no path that contains a vertex that is a vertex on a strongly connected path, then the variable corresponding to the vertex is a quasi invariant variable.*
- **Unfolding factor of quasi invariant variable**. *For any quasi invariant variable x on a* DRG, *the unfolding factor of x is equal to* max{ *n | n = the number of dependence* δ_d^- *edges* (*represented by directed thin dotted line*) *and dependence* δ_c^- *edges* (*represented by directed bold dotted line*) *on a path ending in x* }.

For instance, in Fig. 10, t_1, t_2, z_1, z_2, k_1, k_2 and k_3 are all quasi invariant variables, but the other variables are not because each of them is on a path which contains a strongly connected graph. Because there is a path ending in quasi invariant variable t_1 and this path contains two (maximum) directed thin dotted lines, the unfolding factor of t_1 is 2. In the same way, the unfolding factors of quasi invariant variables t_2, z_1, z_2, k_1, k_2 and k_3 are 1, 1, 0, 3, 2 and 2, respectively.

7.2 Quasi Index Variables and Unfolding Factors

For any variable assigned inside a loop, it must be either a quasi invariant variable or a variant variable. We can further distinguish three types of variant variables: (i) index variables; (ii) quasi index variables; (iii) others. Identification of index variables has been studied by many others, thus we assume here that index variables have been identified. Our goal is to identify quasi index variables. Within a loop, if the test of a conditional is variant, then all variables assigned inside the branches of the conditional are not quasi index variables, since any reference to a quasi index variable can be replaced by an affine function of index variable after a small number of loop iterations.

- **Quasi Index variable**. *For any variant variable* (*non-invariant variable and non-quasi-invariant variable*) *x on the DRG of a loop, if any path ending in the vertex of x contains, only vertexes of index, quasi index or quasi invariant variables, and contains neither* δ_c *dependence edges nor* δ_c^- *dependence edges that starts from a vertex of variant variable, then x is a quasi index variable*.
- **Unfolding factor of quasi-index variable**. *For any quasi index variable x on a DRG, the unfolding factor of x is equal to max{ n | n = the number of* δ_d^- *edges* (*represented by directed thin dotted line*) *and* δ_c^- *edges* (*represented by directed bold dotted line*) *on a path that ends in x and contains no strongly connected graph.* }.

For instance, in Fig. 10, y_1, y_2, x_1, x_2, w_1, w_2, w_3 and w_4 are quasi index variables, and their unfolding factors are 1, 0, 2, 1, 3, 2, 2 and 2, respectively.

8 Algorithms of Evaluating Quasi Invariant/Index Variables and Unfolding Factors

In this section, we present efficient algorithms for identifying quasi invariant/index variables and computing their unfolding factors. The main work of this paper is divided into two phases: 1. Quasi invariance/index analysis that includes (i) detecting dependences among variables and (ii) identifying quasi invariant/index variables and computing their unfolding factors; 2. Loop unfolding. We already discussed how to detect dependences among variables. Based on the dependences, we present two efficient algorithms to identify quasi invariant/index variables and to compute their unfolding factors. **Alg. 1** is based on the well-known algorithm presented by Warshall [24]. The time complexities of Warshall algorithm is $O(n^3)$ in the worst case, where n is the number of the variables modified inside a given loop. Assume that there are n variables $x_1 \ldots x_n$ modified inside a given loop, and five Boolean $n \times n$ matrices Φ_{δ_d}, $\Phi_{\delta_d^-}$, Φ_{δ_c}, $\Phi_{\delta_c^-}$ indicating δ_d, δ_d^-, δ_c, δ_c^- dependence relations among these variables, respectively. $\Phi=\Phi_{\delta_d}\vee\Phi_{\delta_d^-}\vee\Phi_{\delta_c}\vee\Phi_{\delta_c^-}$. Here, for any two variables x_i and x_j, we have:

$$\Phi_{\delta_d}(i,j) = \begin{cases} 1, & \text{if } x_i\ \delta_d\ x_j \\ 0, & \text{otherwise} \end{cases} \qquad \Phi_{\delta_c}(i,j) = \begin{cases} 1, & \text{if } x_i\ \delta_c\ x_j \\ 0, & \text{otherwise} \end{cases}$$

$$\Phi_{\delta_d^-}(i,j) = \begin{cases} 1, & \text{if } x_i\ \delta_d^-\ x_j \\ 0, & \text{otherwise} \end{cases} \qquad \Phi_{\delta_c^-}(i,j) = \begin{cases} 1, & \text{if } x_i\ \delta_c^-\ x_j \\ 0, & \text{otherwise} \end{cases}$$

Moreover, suppose *Ix* denotes the set of index variables, *Qiv* denotes the set of quasi invariant variables and *Qix* denotes the set of quasi index variables.

Alg. 1 (*identifying quasi invariant vs index variables*)
Input: Φ, *Ix*
Output: *Qiv*, *Qix*
Begin
  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if ($\Phi(j, i)$) *for* (k = 1; k <= n; k++) { $\Phi(j, k) = \Phi(j, k) \vee \Phi(i, k)$; }
  $Qiv = \{x_i \mid \forall i_{(1\le i\le n)} \forall j_{(1\le j\le n)} \bullet (\Phi(i, j) \rightarrow \neg\Phi(j, j))\}$;
  $Qix = \{x_i \mid \forall i_{(1\le i\le n)} \bullet (x_i \notin Qiv \wedge \forall j_{(1\le j\le n)} \bullet (\Phi(i, j) \rightarrow \neg\Phi(j, j) \vee (\Phi(j, j) \wedge x_j \in Ix)))\}$;
End

The worst case time complexity of **Alg. 1** is $O(n^3)$. Note that *Ix* is a subset of set *Qix*. While computing the unfolding factors of quasi invariant/index variables, we can exploit the well-known algorithm of Floyd[13] for computing the shortest distance between a pair of vertexes. Because the main focus of computing unfolding factors is anti-dependences, we suppose the length of each anti-dependence edge to be 1 and that of each true dependence edge to be 0. Floyd's algorithm was originally used to compute the shortest path between a pair of vertexes on a directed graph, but we need to compute the longest path here. If a directed graph does not contain any strongly

connected subgraphs, then essentially there will be no difference between computing shortest and longest paths between a pair of vertexes when using Floyd's algorithm. If we delete all the edges starting from or ending in index variable, then all the paths ending in a quasi index variable should not contain any strongly connected graph. In addition to the variables used in **Alg. 1**, we utilize two additional integer $n \times n$ matrices $\wp_{IV}$ and $\wp_{IX}$ defined as: $\wp_{Iv}=\wp_{Ix}=\Phi_{\delta d}^{-} \vee \Phi_{\delta c}^{-}$. $\omega(x)$ indicates the unfolding factor of variable x. Alg. 2 is a variation of Floyd's algorithm, its worst-case time complexity is $O(n^3)$.

Alg. 2 (*computing the unfolding factors of quasi invariant vs index variables*)
Input: Φ, *Ix*, *Qiv*, *Qix*, $\wp_{Iv}$, $\wp_{Ix}$
Output: ω
Begin
  //*Computing the unfolding factors of quasi invariant variables.*
  for any $x_i \in Qiv$
  for any $x_j \in Qiv$
  for any $x_k \in Qiv$
    if $(\Phi(j, i) \wedge \Phi(i, k) \wedge \Phi(j, k))$
      if $(\wp_{Iv}(j, k) < \wp_{Iv}(j, i) + \wp_{Iv}(i, k))$ $\wp_{Iv}(j, k) = \wp_{Iv}(j, i) + \wp_{Iv}(i, k)$;
  for any $x_i \in Qiv$ $\omega(x_i) = max\{\wp_{Iv}(j, i) \mid x_j \in Qiv\}$;
  //*Computing the unfolding factors of quasi index variables.*
  for any $x_i \in Ix$
  for any $x_j \in Ix$
    $\wp_{Ix}(i, j) = \Phi(i, j) = 0$;
  for any $x_i \in Qix$
  for any $x_j \in Qix$ *for any* $x_k \in Qix$
    if $(\Phi(j, i) \wedge \Phi(i, k) \wedge \Phi(j, k))$
      if $(\wp_{Iv}(i, k) < \wp_{Iv}(j, i) + \wp_{Iv}(i, k))$ $\wp_{Iv}(j, k) = \wp_{Iv}(j, i) + \wp_{Iv}(i, k)$;
  for any $x_i \in Qix$
    $\omega(x_i) = max\{\wp_{Iv}(j, i) \mid x_j \in Qix\}$;
End

9 Loop Unfolding

After identifying the set of quasi invariant/index variables and figuring out their unfolding factors by using Alg. 1 and Alg. 2, all that remains now is to select the maximum unfolding factors as the number of iterations that should be unfolded. Because source programs have been converted into SSA form for the purpose of static analysis, it is necessary to convert the SSA form back into original source forms. The main issue to deal with is the removal of all ϕ-functions. For any ϕ-variable x (say defined as $x_3 = \phi(x_1, x_2)$), each reference to x_3 is actually a reference to x_1 or x_2. To preserve the correctness of semantics, we must use a same name for x_1, x_2 and x_3 such that each reference to x_3 will actually be a reference to x_1 or x_2. The following two cases must be considered.

- *Either x_1 or x_2 is a ϕ-variable*. We recursively rename until no new ϕ-variable is encountered.

- x_3 *is an operand of another* ϕ*-variable*. Suppose x_3 is an operand of another ϕ-variable (e.g., $y = \phi(z, x_3)$), y, z and x_3 should also be renamed using the same name. The process continues recursively until no new ϕ-variable is encountered.

Assuming that function α is used to compute the set of variables that should be renamed by a same name, and σ denotes the set of variables already handled, α is defined as below:

$$\alpha(x)_\sigma = \begin{cases} \sigma & \text{if } x \in \sigma \\ \sigma \cup \{x\} & \text{if } x \text{ is not a } \phi\text{-variable} \\ \alpha(y)_{\sigma\cup\{x\}} \cup \alpha(z)_{\sigma\cup\{x\}} \cup \beta(x)_{\sigma\cup\{x\}} & \text{if } x = \phi(y, z) \end{cases}$$

$$\beta(x)_\sigma = \begin{cases} \sigma & \text{if } x \in \sigma \text{ or } x \text{ is not an argument of a } \phi\text{-function} \\ \alpha(z)_{\sigma\cup\{y\}} \cup \beta(y)_{\sigma\cup\{y\}} & \text{if } y = \phi(x, z) \end{cases}$$

For instance, in Fig. 9 there are two ϕ-function assignments: $w_1 = \phi(w_0, w_4)$ and $w_4 = \phi(w_2, w_3)$. All the variables in the set $\alpha(w_1) = \alpha(w_4) = \{w_1, w_0, w_2, w_3\}$ should be renamed by the same name (e.g., w). Similarly, the variables in each of the sets $\{x_1, x_0, x_2\}$, $\{y_1, y_0, y_2\}$, $\{z_1, z_0, z_2\}$, $\{t_1, t_0, t_2\}$, $\{k_1, k_0, k_2, k_3\}$ should be renamed with same names, respectively. After renaming variables, we can unfold loops. The unfolded code of Fig. 9 is shown in Fig. 11. After unfolding a loop, the assignment to each quasi invariant variable can be eliminated since the variable becomes invariant inside the remaining loop. In the remaining loop, each quasi index variable is substituted for a linear expression of index variable. Thus any reference to a quasi index variable can be replaced by the corresponding linear expression of index variable. For instance, x and y are equal to $i - 1$ and i, and w in then-part and else-part are equal to $i - 1$ and i, respectively. In the remaining loop of Fig. 11, $a[i] = p[i–1] + q[i+k]$, $b[i] = b[i] + c[2]$ can be vectorized as $a[3{:}\ n] = p[2{:}\ n–1] + q[3+k{:}\ n+k]$, $b[3{:}\ n] = b[3{:}\ n] + c[2]$, respectively.

```
if (1 <= n) {
   a[1] = p[x] + q[y+k];
   if (odd(t)) { w = 0; b[1] = b[0] + c[z]; } else { w = 1; k = d; b[1] = b[1] + c[z]; }
   t = j + z; z = 2; x = y; y = 2;
}
if (2 <= n) {
   a[2] = p[x] + q[2+k];
   if (odd(t)) { w = 1; b[2] = b[1] + c[2]; } else { w = 2; k = d; b[2] = b[2] + c[2]; }
   t = j + 2; x = y; y = 3;
}
if (3 <= n) {
   a[3] = p[2] + q[3+k];
   if (odd(t)) { w = 2; b[3] = b[2] + c[2]; } else { w = 3; k = d; b[3] = b[3] + c[2]; }
   t = j + 2; x = y; y = 4;
}
for (i = 4; i <= n; i++) {
    a[i] = p[i–1] + q[i+k];
    if (odd(t)) { w = i – 1; b[i] = b[i–1] + c[2]; } else { w = i; b[i] = b[i] + c[2]; }
    x = y;  y =i + 1;
}
```

Fig. 11. The unfolded code of Fig. 9

10 Related Work

As three code optimization techniques, loop invariant code motion, loop unrolling and loop peeling have widely been studied and used by compilers. A comprehensive survey of these and other source level optimization can be found in [4]. A more recent survey of many state of the art optimization techniques for high performance architectures can be found in [2], [19].

Loop invariant code motion was originally mentioned in [1]. The notion of quasi invariant grew out of our work on partial evaluation [21]. Loop quasi invariant code motion is an extension of loop invariant code motion, which hoists invariant code to outside of loops by unfolding loops for a small number of iterations. A recently developed transformation is partial redundancy elimination (PRE), which is a global optimization technique, generalizing the removal of common sub-expressions and loop-invariant computations. Initial implementation of PRE failed to completely remove the redundancies [20], [23]. More recent PRE algorithms based on control flow restructuring [6], [24] can achieve a complete PRE and are capable of eliminating loop quasi invariant code. However, these techniques have exponential (worst-case) time complexity as well as code size explosion resulting from replication of the code. Our techniques statically determine a finite fixed point of computations induced by assignments, loops and conditionals and tries to compute the optimal unfolding factors to get maximal code motion and parallelization; and our algorithm has a polynomial time complexity.

Loop peeling was originally mentioned in [15], and automatic loop peeling techniques were discussed in [16]. August [3] showed how loop peeling can be applied in practice, and elucidated how this optimization alone may not increase program performance, but may expose opportunities for other optimization leading to performance improvements. August [3] used only heuristic loop peeling techniques. We feel that when applied to new and innovative architectures such as the SDF [14] (Scheduled Dataflow architecture, a decoupled memory/execution, multithreaded architecture using non-blocking threads), our pre-optimization approach may prove to be of significant importance. The benefits of loop unrolling have been studied for various architectures [11]. It is a fundamental technique for generating the long instruction sequences required by VLIW machines [12]. A key issue in applying loop peeling and loop unrolling is the number of iterations that must be peeled off or replicated from the loop body. Current techniques use heuristic or ad hoc techniques that are based on loop-carried dependence analysis.

Many optimization techniques can be formalized conveniently using static single assignments, including the elimination of partial redundancies [16], constant propagation [7], [17], and code motion [10]. We followed the same approach to express our loop optimization technique.

11 Conclusion and Future Work

In this paper, we presented a loop oriented optimization technique based on dependence analysis. In particular, our technique detects anti-dependencies among variables

involved in loop, and then tries to remove some anti-dependencies as much as possible by unfolding loops for a small number of iterations. After the removal of quasi invariant variables and the substitution of quasi index variables for linear functions of index variables, there will be only inductive variables inside loops, and thus loops will be relatively clean and easy to analyze and expose more opportunities for other optimization leading to performance improvements. Exploiting this technique, we can extend conventional loop invariant code motion to loop quasi invariant code motion, which is capable of moving not only invariant code but also quasi invariant code. Loop quasi invariant code motion is well-suited as a supporting transformation in compilers, partial evaluators, and other program transformers. Moreover, removing loop-independent dependences may make static analysis based on loop-carried dependence easier, which will be very beneficial to many other optimizations leading to performance improvements such as loop unrolling, loop peeling and so on. Our technique has the potential to increase the accuracy of program analyses and to expose newer program optimizations (e.g., branch predication, for extracting instruction-level parallelism from programs.), which are of central importance to many compilers and program transformations. The algorithms presented in this paper uses the infrastructure already present in many compilers, such as dependence graphs and static single assignments. Thus they do not require fundamental changes to existing systems. The application of this technique to our ongoing compiler for the multithreaded architecture SDF, and larger practical programs is hoped to reveal the significance of the work presented here. To the best of our knowledge, this is the first attempt of systematically making use of loop-independent dependences among variables to unfold loops for optimization.

References

1. Aho A. V., Sethi R., Ullman J. D., “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, Reading, Mass, 1986.
2. Allen R., Kennedy K., “Optimization Compilers for Modern Architectures”, Morgan Kaufmann Publishers, 2002.
3. August D. I., “Hyperblock performance optimizations for ILP processors”, M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.
4. Bacon D. F., and Graham S. L., “Compiler transformations for high-performance computing”, ACM Computing Surveys, December 1994, Vol. 26, No. 4, pp.345-420.
5. Banerjee, U., “An introduction to a formal theory of dependence analysis”, Journal of Supercomput. Vol. 2, No.2, 1988, pp.133-149.
6. Bodik R., Gupta R., Soffa M. L., “Complete removal of redundant expressions”, Prod. ACM Conf. On Programming Language Design and Implementation, pp.1-14, ACM Press, 1998.
7. Bulyonkov M. A., Kochetov D. V., “Practical aspects of specialization of Algol-like programs”, eds. Dancy O., Glueck R., Thiemann P., “Partial Evaluation”, Proceedings. LNCS, Vol. 1110, pp.17-32, Springer-Verlag, 1996.
8. Cocke J., Schwartz J. T., “Programming languages and their compilers (preliminary notes)”, 2nd ed. Courant Institute of Mathematical Science, New York University, New York.

9. Cytron R., Ferrante J., “Efficiently computing static single assignment form and the control dependence graph”, ACM TOPLAS, October, 1991, Vol. 13, No. 4, pp.451-490.
10. 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, pp.70-85, ACM Press, 1986
11. Dongarra J., Hind A. R., “Unrolling loops in Fortran”, Softw. Pract. Exper., Vol. 9, No. 3, pp.219-226, 1979.
12. Ellis J. R., “Building: A Compiler for VLIW Architecture”, ACM Doctoral Dissertation Award. MIT Press, Cambridge, Mass, 1986.
13. Floyd R. W., “Algorithm 97: shortest path”, Communications of the ACM, 1962, Vol. 5, No. 6, pp.345.
14. Kavi K. M., Giorgi R. and Arul J., “Scheduled Dataflow: Execution paradigm, architecture and performance evaluation”, IEEE Transactions on Computer, Vol. 50, No. 8, pp.834-846, Aug. 2001.
15. Lin D. C., “Compiler support for predicated execution in superscalar processors”, M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.
16. Mahlke S. A., “Exploiting instruction level parallelism in the presence of conditional branches”, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.
17. Metzger R., Stroud S., “Interprocedual constant propagation: An empirical study”, ACM Letters on Programming Languages and Systems, Vol. 2, No.1, pp.213-232, 1993.
18. Padua D. A., and Wolfe M. J., “Advanced compiler optimizations for supercomputers”, Communications of the ACM, December 1986, Vol. 29, No. 12, pp.1184-1201.
19. Pande S., Agrawal D. P., (Eds.) “Compiler Optimizations for Scalable Parallel Systems”, LNCS 1808, Springer, 1998.
20. Rosen B. K., Wegman M. N., and Zadeck F. K., “Global value numbers and redundant computations”, Conference Record of the 15th ACM Symposium on Principles of Programming Languages, ACM Press, 1988, pp.12-27.
21. Song L., “Studies on Termination Methods of Partial Evaluation”, Ph.D. thesis, Department of Computer Science, Waseda University, Tokyo, Japan, 2001.
22. Steffen B., “Property oriented expansion”, Symposium on Static Analysis, LNCS 1145, pp.22-41, Springer-Verlag, 1996.
23. Steffen B., Knoop J., Rüthing O., “The value flow graph: A program representation for optimal program transformations”, ed. Jones N. D., ESOP’90, LNCS 432, pp.389-405, Springer-Verlag, 1990.
24. Warshall S., “A theorem on Boolean matrices”, Journal of the ACM, January 1962, Vol. 9, No. 1, pp.11-12.
25. Wolfe, M. J., “Optimizing supercompilers for supercomputers”, Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, Mass.
26. Wolfe, M. J., “High performance compilers for parallel computing”, Addison-Wesley Publishing Company, Inc., 1996.
27. Zima H., and Chapman B., “Supercompiler for parallel and vector computers”, Frontier, Series, ACM Press, 1990.