A Technique for Variable Dependence Driven Loop Peeling

Litong Song Krishna M Kavi
Dept. Computer Science
University of North Texas

Abstract

Loops in programs are the source of many optimizations leading to performance improvements, particularly on modern high-performance architectures as well as vector and multithreaded systems. Among the optimization techniques, loop peeling is an important technique that can be used to parallelize computations. The technique relies on moving computations in early iterations out of the loop body such that the remaining iterations can be executed in parallel. A key issue in applying loop peeling is the number of iterations that must be peeled off from the loop body. Current techniques use heuristics or ad hoc techniques to peel a fixed number of iterations or a speculated number of iterations. To our knowledge, no formal or systematic technique that can be used by compilers to determine the number of iterations that must be peeled off based on the program characteristics. In this paper we introduce one technique that uses variable dependence analysis for identifying the number of iterations to be peeled off. Our goal is to find general techniques that can accurately determine the ideal number of iterations for loop peeling, while working within the context of other loop optimizations including code motion.

1: Introduction

Modern computer systems exploit both instruction level parallelism (ILP) and thread (or task) level parallelism. Superscalar and VLIW systems rely on ILP while multi-threaded and multiprocessor systems rely on thread level parallelism. In order to fully benefit from ILP or thread level parallelism (or both), compilers must perform complex analyses to identify and schedule code for the architecture. Typically compilers focus on loops for finding parallelism in programs [24]. 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, 23]. When a loop is peeled, a small number of early iterations are removed from the loop body and executed separately (before the start of the loop). If only one iteration is peeled, a common case, the code for that iteration can be enclosed within a conditional statement. If more than one iteration is peeled, it may be possible to use a separate loop for these iterations. The main purposes of this technique is for removing dependencies created by the early iterations on the remaining iterations, thereby enabling parallelization; and for matching the iteration control of adjacent loops to enable fusion. The loop in Figure 1(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 loop iteration fully parallel, as shown in Figure 1(b). Using vector notation (for processors), the loop in Figure 1(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.

```
DO i = 1 TO n
    a[i] := a[1] + b[i];
ENDDO;
```
(a) original loop

```
IF (1 <= n) THEN
    a[1] := a[1] + b[1];
ENDIF;
DO i = 2 TO n
    a[i] := a[1] + b[i];
ENDDO;
```
(b) after peeling one iteration from the original loop

Figure 1. First example of loop peeling.

The loop in Figure 2(a) is not parallelizable because of possible flow dependences between iterations. Peeling off the first iteration, once again, makes the loop parallel as shown in Figure 2(b). The loop in Figure 2(b) can be written as a vector assignment: a(2:n):=a(1)+b(2:n).

The loop in Figure 3(a) is not parallelizable because variable wrap is neither a constant nor a linear function of induction variable i. Peeling off the first iteration allows the rest of loop to be vectorizable, as shown in Figure 3(b). The loop in Figure 3(a) can be rewritten as a vector assignment: b(2:n):=a(2:n)+a(1:n-1). The code shown in Figure 3(a) is an example using an array to simulate a cylindrical coordinate system where the left edge of the

array must be adjacent to its right edge. Here, wrap is used as a wraparound variable.

```
DO i = 1 TO n
    a[i] := a[j] + b[i];
    j := 1;
ENDDO;
```
(a) original loop

```
IF (1 <= n) THEN
    a[1] := a[j] + b[1];
    j := 1;
ENDIF;
DO i = 2 TO n
    a[i] := a[1] + b[i];
ENDDO;
```
(b) after peeling one iteration from the original loop

Figure 2. Second example of loop peeling.

```
DO i = 1 TO n
    b[i] := a[i] + a[wrap];
    wrap := i;
ENDDO
```
(a) original loop

```
IF (1 <= n) THEN
    b[i] := a[i] + a[wrap];
    wrap := i;
ENDIF;
DO i = 2 TO n
    b[i] := a[i] + a[i-1];
ENDDO
```
(b) after peeling one iteration from the original loop

Figure 3. Second example of loop peeling.

In the examples shown here, the number of iterations that must be peeled off to achieve parallel (or vector) execution of the loops is 1. In general, however, more than one iteration may have to be peeled off, and the issue addressed in this paper is how to determine the "optimum" (i.e., least) number of iterations that must be peeled off. To our knowledge, conventional techniques fall into one of two kinds: (i) peel off a fixed number of iterations, (ii) peel off a predicted or speculated number of iterations. There exists no formal or systematic techniques that rely on static properties of programs to guide loop peeing.

Two case of loop peering can be distinguished.

- If the subscript expressions of array references are constant or linear expressions of the induction variable, then loop peeling can be carried out by analyzing the dependence between array references in different iterations. Figure1 is an example such a case. It should be noted, that some dependences might lead to cases where the loop cannot be made parallel by peeling loop iterations. Note that a loop might not need peeling at all in this case.
- If a subscript expression of array reference is neither constant nor linear expression of induction variable, when possible, loop iterations may be peeled off until this subscript expression either becomes a constant or a linear expression of induction variable. When this occurs, the remaining iterations of the loop satisfy the conditions of case 1 above. Figures 2 and 3 are examples of this case.

In case 1, loop peeling is based on the dependencies between array references in different iterations, which can be analyzed at compile time since the subscript expression of all array reference are constant or a linear expressions of the induction variable. Thus we will not address this case and concentrate only on the second case in the remainder of the paper. More specifically, we present a novel technique called *variable dependence driven loop peeling,* aimed at finding a general and systematic way to determine the number of loop iterations that must be peeled off to convert case 2 loops into case 1 loops.

2: Single Static Assignment

This section summarizes the single static assignment form. Variables inside a loop may be assigned a value more than once and those assignments may have different properties. In order to perform dependency analyses, it is necessary to distinguish the assignment, and we use the well-known single static assignment (SSA) [9] 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. 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 [8]. Note that this intermediate form is only used for program analysis and it will be converted back to original syntax after optimization. In the remainder of this paper, we will assume that all source programs are represented in SSA form.

3: Quasi-Invariant Variables and Quasi-Induction Variables

The invariant variables of a loop are those variables whose values are invariant in all the iterations of the loop. The induction variable of a loop is a variable whose values in successive iterations form an arithmetic

progression. The most obvious example of an induction variable is the index variable of a loop. Induction variables are often used in array subscript expressions. In this paper we present two notions:

- *Quasi-invariant variables*: those variables that are assigned inside a loop and will become invariant after a small number of iterations of the loop.
- *Quasi-induction variables*: those variables that are assigned inside a loop and will become linear expressions of induction variable after a small number of iterations of the loop.

Now we can introduce two related notions:

- *The peeling length of a quasi-invariant variable*: if a quasi-invariant variable of a loop becomes invariant after at least n iterations of the loop, then n is the peeling length of the variable.
- *The peeling length of a quasi-induction variable*: if a quasi-induction variable of a loop becomes a linear expression of the induction variable after *at least* n iterations of the loop, then n is the peeling length of the variable.

For example, variable j in Figure 2(a) is a quasi-invariant variable, and wrap in Figure 3(a) is a quasi-induction variable. Furthermore, the peeling length of both j and wrap is 1. We will present more formal definitions of these notions in section 6. We will use these notions in determining the number of iterations to peel off so that all quasi-invariant become invariant and all quasi-induction variables become linear expressions of induction variables. Now the challenge is:

- How to effectively identify quasi-invariant and quasi-induction variables?
- How to efficiently compute the peeling lengths of these variables?

To deal with the two issues, we introduce a few new terms in the next section.

4: Variable Dependence

Compiler relies heavily on both control and data dependence analyses for performing optimizations [4, 23]. Among the types of dependencies, we recognize three forms: true dependence, anti-dependence and output dependence. Typically these terms relate to dependencies among statements. For our purpose it is necessary to detect dependencies among variables (instead of statements), hence we introduce two new definitions for data dependencies and two definitions for control dependencies.

Definition 1 (*true-reference dependence*) If the assignment to a variable var_1 reads a variable var_2 which has been written before in the sense of static semantics, then var_1 has a true-reference dependence to var_2 (denoted by $var_2 \rightarrow var_1$).

For example, if we have statements: x:=a+b; y:=x+z; then y has a true-reference dependence to x ($x \rightarrow y$).

Definition 2 (*anti-reference dependence*) If the assignment to a variable var_1 reads a variable var_2 which will be written afterwards in the sense of static semantics, then var_1 has a anti-reference dependence to var_2 (denoted by $var_2 \dashrightarrow var_1$).

For example, if we have statements: y:=x+z; x:=a+b; then y has an anti-reference dependence to x ($x \dashrightarrow y$).

Definition 3 (*true-control dependence*) If there is an assignment to a variable var_1 within arbitrary branch of a conditional, and the test expression of this conditional reads a variable var_2 which has been written before in the sense of static semantics, then var_1 has a true-control dependence to var_2 (denoted by $var_2 \hookrightarrow var_1$).

For example, if we have statements: x:=a+b; IF x= c THEN y:=d; ENDIF; then y has a true-control dependence to x ($x \hookrightarrow y$).

Definition 4 (*anti-control dependence*) If there is an assignment to a variable var_1 within arbitrary branch of a conditional, and the test expression of this conditional reads a variable var_2 which will be written afterwards in the sense of static semantics, then var_1 has a anti-control dependence to var_2 (denoted by $var_2 \not\hookrightarrow var_1$).

For example, if we have statements: IF x=c THEN y:=d; ENDIF; x:=a+b; then y has an anti-control dependence to x ($x \not\hookrightarrow y$).

5: Variable Dependence Graph

After deriving the four dependences among variables assigned inside a loop, we can construct a directed graph called variable dependence graph.

Definition 5 (*variable dependence graph*) The variable dependence graph (DRG) of a loop (denoted by loop) is a directed graph where Node(loop)={var | var is a variable assigned inside loop}; Edge(loop)=

{directed line $\rightarrow$ from node var_1 to node var_2 | $(var_1 \in IV) \wedge ((var_1 \rightarrow var_2) \vee (var_1 \dashrightarrow var_2))$ }

$\cup$ {directed line $\Rightarrow$ from node var_1 to node var_2 | $(var_1 \in IV) \wedge ((var_1 \hookrightarrow var_2) \vee (var_1 \not\hookrightarrow var_2))$ }

$\cup$ {directed line $\rightarrow$ from node var_1 to node var_2 | $(var_1 \notin IV) \wedge (var_1 \rightarrow var_2)$ }

$\cup$ {directed line $\Rightarrow$ from node var_1 to node var_2 | $(var_1 \notin IV) \wedge (var_1 \hookrightarrow var_2)$ }

$\cup$ {directed line $\dashrightarrow$ from node var_1 to node var_2 | $(var_1 \notin IV) \wedge (var_1 \dashrightarrow var_2)$ }

$\cup$ {directed line $\not\Rightarrow$ from var_1 to var_2 | $(var_1 \notin IV) \wedge (var_1 \not\hookrightarrow var_2)$ }

Where IV indicates the set of induction variables.

Using the variable dependence graph, we can not only identify quasi-invariant variables and quasi-induction variables, but also compute their peeling lengths.

6: Quasi-Invariant/Quasi-Induction Variables and Their Peeling Lengths

In section 3 we gave an informal definition for quasi-invariant and quasi-induction variables. With the introduction of the variable dependence graph, we can present a more formal definition for these terms; the formal definition themselves describe algorithms for identifying them.

6.1: Quasi-Invariant Variables and Their Peeling Lengths

Definition 6 (*quasi-invariant variable*) For any variable node on the variable dependence graph of a loop, if among all the paths ending in this node, there is no path which contains a node that is a node on a strongly connected path, then we say this variable is a quasi-invariant variable.

Definition 7 (*peeling length of quasi-invariant variable*) For any quasi-invariant variable var on a variable dependence graph, the peeling length of var is defined as: max{ length | length = the number of edge ⇢ (anti-reference dependence) and edge ➔ (anti-control dependence) on a path ending in var }.

6.2: Quasi-Induction Variables and Their Peeling Lengths

In this subsection, let us investigate quasi-induction variables and their peeling lengths. For any variable assigned inside a loop, it must be either a quasi-invariant variable or a variant variable. We can further distinguish three kinds of variant variables: (i). induction variables; (ii). quasi-induction variables; (iii). others. Identification of induction variable has been studied by many others and thus omitted in this paper and we will assume that induction variables have been identified. Our goal is the identification of quasi-induction variables. Before formally defining quasi-induction variables, we must clarify a case arising in conditionals. Within a loop, if the test expression of a conditional uses a variant variable, then all variables assigned inside all branch paths of this conditional are not quasi-induction variables, since in our definition, a reference to a quasi-induction variable can be replaced by a linear function of the induction variable after a small number of loop iterations.

Definition 8 (*quasi-induction variable*) Let induction variable be a quasi-induction variable. For any variant variable node on the variable dependence graph of a loop, if, any path ending in this node, contains only quasi-induction or quasi-invariant variables, and contains neither a ➡ edge (true-control dependence) nor an ➔ edge (anti-control dependence) that starts from a variant variable node, then we say this variable is a quasi-induction variable.

Definition 9 (*peeling length of quasi-induction variable*) For any quasi-induction variable var on a variable dependence graph, the peeling length of var is defined as: max{ length | length = the number of edge ⇢ (anti-reference dependence) and edge ➔ (anti-control dependence) on a path that ends in var and contains at most one induction variable node. }.

7: Algorithms

The main work of this paper is divided into a couple of phases as follows:

- Quasi-invariance and quasi-induction analysis;
  - Detecting the dependences among variables;
  - Identifying quasi-invariant variables and quasi-induction variables, and computing their peeling lengths;
- Loop peeling;

We have discussed how to detect the dependences among variables. Based on the dependences among variables, we present two efficient algorithms to identify quasi-invariant variables and quasi-induction variables, and to compute their peeling lengths, respectively. The first algorithm exploits the well-known algorithm presented by Warshall [22]. The time complexities of Warshall algorithm is $O(n^3)$ in the worst case, where n is the number of the variables assigned inside a given loop. When computing the peeling lengths of quasi-invariant variables and quasi-induction variables, we can exploit the well-known algorithm of Floyd [10] for computing the shortest distance between a pair of nodes on a directed graph. Since the second algorithm is a variation of Floyd's algorithm, its worst-case time complexity is $O(n^3)$. Because the main focus of computing peeling lengths should be the anti-dependences between variables, we set the length of each anti-dependence edge to 1 and that of each true-dependence edge to 0. Floyd's algorithm was originally used to compute the shortest path between a pair of nodes on a directed graph, but we need to compute the longest path. The difference between longest and shortest path computations depends on whether the graph contains strongly connected subgraphs or not. When a directed graph contains no strongly connected subgraphs, then there is no difference between computing shortest and longest paths between a pair of nodes using Floyd's algorithm. For a quasi-induction variable, if we delete the edges involving induction variables, then all paths ending in a quasi-induction variable do not contain strongly connected graphs.

8: Loop Peeling

When the two previous algorithms are applied we can identify the set of quasi-invariant variables and quasi-induction variables and compute their peeling lengths. All that remains now is to select the maximum of the peeling lengths as the number of iterations that must be peeled off to make the loop vectorizable. However, we converted the source program into SSA form for the purpose of quasi-invariant and quasi-induction variable analysis, it is necessary to convert the SSA form back into the source code before the loop peeling. The main issue to deal with is removal of the ϕ-functions by using variable renaming. For any ϕ-variable var (say defined as var:=ϕ(x, y)), each reference to var is actually a reference to x or y. To preserve the correctness of semantics, we must use a same name for variables var, x and y such that each reference to var will actually be a reference to x or y. The following two cases must be considered.

- Either x or y is a ϕ-variable. If either x or y is a ϕ-variable, we have a recursive substitution until no new ϕ-variables are encountered.
- var is an operand of another ϕ-variable. If var is an operand of another ϕ-variable (e.g., u:=ϕ(v, var)), u, v and var should also be renamed using the same name. The process continues recursively until no new ϕ-variables are encountered.

After peeling, the assignment to each quasi-invariant variable can be removed and its value can be directly used within the remaining loop, since the variable has turned into invariant (this can also be called loop quasi-invariant code motion [18, 19]: an extension of loop invariant code motion). It should be noted that some quasi-invariant variables could have been removed earlier as long as their peeling lengths are less than the peeling length of the loop.

9: Related Work

As two code motion techniques, loop peeling and loop invariant code motion have widely been studied and used in compilers. For a survey of these and other source level optimization can be found in [3]. Loop peeling was originally mentioned in [12], and automatic loop peeling techniques were discussed in [13]. August [2] showed how loop peeling could 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 used only heuristic loop peeling techniques. We feel that when applied to new and innovative architectures such as the SDF [11] (Scheduled Dataflow architecture, a decoupled memory/execution, multithreaded architecture using non-blocking threads), our loop peeling approaches may prove to be of significant importance.

Loop invariant code motion was originally mentioned in [1, 7]. When an invariant computation appears inside a loop, a compiler can move that computation to outside the loop. The notion of quasi-invariant grew out of our work on partial evaluation [17]. Loop quasi-invariant code motion is an extension of loop invariant code motion, which hoists invariant code to outside of loops by peeling/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 [16, 21]. More recent PRE algorithms based on control flow restructuring [5, 20] 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 technique statically determines a finite fixed point of computations induced by assignments, loops and conditionals and tries to compute the optimal peeling length to get maximal code motion and parallelization. Moreover, our algorithm has a polynomial time complexity. Many optimization techniques can be formalized conveniently using single static assignments, including the elimination of partial redundancies [16], constant propagation [6, 14], and code motion [9]. We followed the same approach to express our loop optimization technique.

10: Conclusion

In this paper, we presented a technique for variable driven loop peeling, which is based on a static analysis for loop quasi-invariant variables, quasi-induction variables and their peeling lengths. Our analysis tries to determine the optimal peeling length needed to parallelize loops. To the best our knowledge, this is the first attempt of systematically making use of static properties of a loop. When we peel off a loop for parallelization, not only each quasi-induction variable becomes a linear function of an induction variable, but also each quasi-invariant variable becomes an invariant variable of the loop. So, besides loop peeling, this technique can be used for loop quasi-invariant code motion, which is well-suited as supporting transformation in compilers, partial evaluators, and other program transformers. Our technique has the potential to increase the accuracy of program analyses and to expose newer program optimizations (e.g., branch predication: it is an extremely valuable tool in extracting instruction-level parallelism from programs.), which are

of central importance to many compilers and program transformations. The algorithms presented in this paper use the infrastructure already present in many compilers, such as dependence graphs and single static 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.

References

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

[2] August D. I., "Hyperblock performance optimizations for ILP processors", M.S. thesis, Department of Electrical and ComputerEngineering, University of Illinois, Urbana, IL, 1996.

[3] 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.

[4] Banerjee, U., "An introduction to a formal theory of dependence analysis", Journal of Supercomput. Vol. 2, No.2, 1988, pp.133-149.

[5] 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.

[6] Bulyonkov M. A., Kochetov D. V., "Practical aspects of specialization of Algol-like programs", eds. Dancy O., Clck R., Thiemann P., "Partial Evaluation", Proceedings. LNCS, Vol. 1110, pp.17-32, Springer-Verlag, 1996.

[7] Cocke J., Schwartz J. T., "Programming languages and their compilers (preliminary notes)", 2nd ed. Courant Institute of Mathematical Science, New York University, New York.

[8] 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.

[9] 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

[10] Floyd R. W., "Algorithm 97: shortest path", Communications of the ACM, 1962, Vol. 5, No. 6, pp.345.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Song L., "studies on termination methods of partial evaluation", Ph.D. thesis, Department of Computer Science, Waseda University, Tokyo, Japan, 2001.

[18] Song L., Futamura Y., Glück R., and Hu Z., "Loop Quasi-Invariant Code Motion", IEICE Transactions on Information & System, October 2000, Vol. E83-D, No. 10: pp.1841-1850.

[19] Song L., Futamura Y., Glück R., and Hu Z., "A Loop Optimization Technique Based on Quasi-Invariance", Proceedings of IFIP Conference on Software: Theory and Practice (16th World Computer Congress 2000), Beijing, August 2000, pp.80-90.

[20] Steffen B., "Property oriented expansion", Symposium on Static Analysis, LNCS 1145, pp.22-41, Springer-Verlag, 1996.

[21] 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.

[22] Warshall S., "A theorem on Boolean matrices", Journal of the ACM, January 1962, Vol. 9, No. 1, pp.11-12.

[23] Wolfe, M. J., "Optimizing supercompilers for supercomputers", Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, Mass.

[24] Zima H., and Chapman B., "Supercompiler for parallel and vector computers", Frontier, Series, ACM Press, 1990.