Extended Cycle Shrinking: A Restructuring Method For Parallel Compilation

S.Biswas   A.Sanyal   L.M.Tewari

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay 400076, INDIA

Abstract

An important part of a parallelizing compiler is the restructuring phase, which extracts parallelism from a sequential program. We consider an important restructuring transformation, called cycle shrinking [1], which partitions the iteration space of a loop so that the iterations within each partition can be executed in parallel. We propose a new cycle shrinking transformation, called extended cycle shrinking, which is an improvement over the existing methods. We present the conditions under which our method can be applied, and give an algorithm which performs this transformation. Further, we present results to show that our method always leads to a minimal number of partitions, whereas the earlier methods do not. Thus our algorithm is, in this sense, optimal.

1 Introduction And Basic Definitions

In this paper, we shall be concerned with indexed statements only. Let $S_i(I_1, \ldots, I_n)$ represent an indexed statement surrounded by n loops, where $I_j$ represents the index of the jth loop (with lower bound 1 and upper bound $N_j$). Such a statement has $\prod_{1 \leq j \leq n} N_j$ different instances, one for each value of each of $I_j$. The order of execution is given by the relation $< \text{ where } S_i < S_j \text{ means } S_i \text{ precedes } S_j \text{ in execution order.}$

If $S_i(i_1, \ldots, i_n)$ represents an instance of the statement $S_i$, then $\text{OUT}(S_i(i_1, \ldots, i_n))$ and $\text{IN}(S_i(i_1, \ldots, i_n))$ denote the set of variable instances defined and used respectively by this statement instance. Two statements $S_i(I_1, \ldots, I_n)$ and $S_j(I_1, \ldots, I_n)$ are involved in a dependence $S_i \delta S_j$ iff there exist index values $(i_1, \ldots, i_n)$ and $(j_1, \ldots, j_n)$, such that $S_i(i_1, \ldots, i_n) < S_j(j_1, \ldots, j_n)$, and $\text{OUT}(S_i(i_1, \ldots, i_n)) \cap \text{IN}(S_j(j_1, \ldots, j_n)) \neq \emptyset$ (flow dependence), or $\text{IN}(S_i(i_1, \ldots, i_n)) \cap \text{OUT}(S_j(j_1, \ldots, j_n)) \neq \emptyset$ (anti dependence), or $\text{OUT}(S_i(i_1, \ldots, i_n)) \cap \text{OUT}(S_j(j_1, \ldots, j_n)) \neq \emptyset$ (output dependence).

In all three cases $S_i(I_1, \ldots, I_n)$ is called the dependence source and $S_j(I_1, \ldots, I_n)$ is called the dependence sink. Note that the dependence $S_i \delta S_j$ could have several different instances.

A program data dependence graph is a directed graph $G(V, E)$, with a set of nodes $V = S_1, \ldots, S_n$ corresponding to statements in the program, and a set of arcs $E = \{(S_i, S_j) | S_i, S_j \in V \text{ and } S_i \delta S_j\}$.

The data dependence graph is used to determine whether loops can be parallelized. A loop whose iterations can execute in parallel is called a DOALL loop.
\begin{verbatim}
DO I = 1, N
ENDO
\end{verbatim}

Figure 1: (a) Loop with dependence cycle (b) Dependences shown in iteration space. The top arrows represent dependence instances from \( S_1 \) to \( S_2 \) and the bottom arrows from \( S_2 \) to \( S_1 \). (c) Parts which can be executed in parallel

The body of a loop which does not contain any dependence cycles, can be parallelized using well known transformations such as node splitting. However, if there are cycles of dependences in the body of the loop, then a straightforward parallelization cannot be done, especially if the dependences involved in the cycle are flow dependences [2]. In such cases cycle shrinking can be used.

2 Cycle Shrinking

Loops which contain cycles of data dependences can be parallelized partially, if the gaps between the dependences are known. The method was first discussed in [1]. We now provide a brief description of the method.

Consider the loop shown in Figure 1. Clearly, there is a cycle of dependence between the statements \( S_1 \) and \( S_2 \). However, since a value generated by \( S_1 \) in a particular iteration is used five iterations later by \( S_2 \), and similarly a value generated by \( S_2 \) is used four iterations later by \( S_1 \), we can partition the iteration space into groups of four iterations so that the following important condition is satisfied. There is an ordering of the groups of the partition such that the data items used in iterations of group \( j \) are defined in the iterations of groups 1 to \( j - 1 \). Under such a condition, if all the iterations of groups 1 to \( j - 1 \) are already done, then the iterations of the group \( j \) can be done in parallel. We call such a partition of the iteration space a legal partition and the notion is formalized below.

**Definition 2.1** Consider a partition \( P = \{p_1, \ldots, p_m\} \) of the iteration space, i.e. each \( p_i \) is a group of iterations such that \( p_i \cap p_j \) is empty if \( i \neq j \), and \( \bigcup_{1 \leq i \leq m} p_i = \mathcal{N} \) (we denote the entire iteration space \( N_1 \times N_2 \times \ldots, N_n \) as \( \mathcal{N} \)). \( P \) is a legal partition, if for any \( i \), the sources of the dependences corresponding to the sinks in \( p_i \) are confined to the groups \( \{p_j | j < i\} \).