

Optimizing Nested Loops with Loop Distribution and Loop Fusion

Meilin Liu*

*Department of Computer Science and Engineering
Wright State University*

Chun Xue, Qingfeng Zhuge, Meikang Qiu

*Department of Computer Science
University of Texas at Dallas*

Zili Shao

*Department of Computing
Hong Kong Polytechnic University*

Edwin H.-M. Sha

*Department of Computer Science
University of Texas at Dallas, USA*

Contents

I. Introduction	1
II. Basic Concepts	3
A. Data Flow Graph	4
B. Loop Dependence Graph	4
III. Loop Distribution	5
A. Theorems of Loop Distribution	5
B. Maximum Loop Distribution	7
IV. Loop Distribution with Loop Fusion	8
A. Direct Loop Fusion	8
B. Loop Distribution with Direct Loop Fusion	8
V. Experiments	9
VI. Conclusion	10
References	10

I. INTRODUCTION

Loop fusion groups multiple distinct loops into a single loop. Loop fusion can be used to reduce the cost of loop bound testing. Loop fusion can also be used to exploit the instruction-level parallelism on the modern high-performance architecture such as VLIW [1, 8]. Loop fusion can enhance the data locality by reusing the variables from local storage, thus it reduces the number of memory references and the power consumption [1, 5, 7, 8].

Direct loop fusion is to find the legal fusion partition of the loop nodes so that the loop nodes inside one partition can be fused directly without transformation. Maximum direct loop fusion is to minimize the number of the fusion partitions, thus the resultant number of the fused loops is minimized.

Loop distribution separates independent statements inside a single loop (or loop nest) into multiple loops (or loop nests) [1, 3, 5, 8]. Loop distribution can be used to break up a large loop that doesn't fit into the cache [1, 3, 8]. It

*Electronic address: meilin.liu@wright.edu

can also improve memory locality by fissioning a loop that refers to many different arrays into several loops, each of which refers to only a few arrays. Loop distribution can also enable other loop optimization techniques such as loop fusion, loop interchanging and loop permutation [1, 3, 5, 8].

A lot of research for loop transformation has been done in loop fusion and loop distribution to improve the instruction-level parallelism and enhance data locality [1, 3, 5, 8]. In [3], Kennedy and McKinley use loop fusion and distribution to enhance data locality and maximize parallelism separately, which may not give the optimal results. McKinley et al. [5] tried to use a compound loop transformation algorithm that consists of loop permutation, fusion, distribution, and reversal to achieve the best loop structure for a loop nest in terms of the cache line references. The above loop fusion techniques, however, do not consider resultant code size of a transformed loop which is another critical concern for embedded system design [9].

Timing and code size are the two most important performance metrics for embedded systems with very limited on-chip memory resources [9]. Combining loop distribution and loop fusion can lead to an optimization solution with both reduced execution time and restricted code size. After loop distribution is applied to create the finest possible loop nests, direct loop fusion must be applied to exploit data locality or improve the parallelism. In this paper, we propose a technique of *loop distribution with maximum direct loop fusion* (LD_MDF), which performs *maximum loop distribution*, followed by *maximum direct loop fusion*. The technique significantly improves the timing performance compared to the original loops without jeopardizing the code size.

<pre> for i=0, N for j=0, M A[i,j]=A[i,j]*2+1; endfor for j=0, M B[i,j]=B[i,j-1]*3; endfor for j=0, M C[i,j]=(A[i,j+1]+B[i,j+1])/2; D[i,j]=B[i,j]*3; endfor for j=0, M E[i,j]=C[i,j-2]+5; F[i,j]=(C[i,j-1]+D[i,j+1])/2; endfor endfor </pre> <p style="text-align: center;">(a)</p>	<pre> for i=0, N A[i,0]=A[i,0]*2+1; B[i,0]=B[i,-1]*3; D[i,0]=B[i,0]*3; A[i,1]=A[i,1]*2+1; B[i,1]=B[i,0]*3; D[i,1]=B[i,1]*3; for j=0, M-2 A[i,j+2]=A[i,j+2]*2+1; B[i,j+2]=B[i,j+1]*3; D[i,j+2]=B[i,j+2]*3; C[i,j]=(A[i,j+2]+B[i,j+1])/2; E[i,j]=C[i,j-2]+5; F[i,j]=(C[i,j-1]+D[i,j+1])/2; endfor C[i,M-1]=(A[i,M+1]+B[i,M])/2; E[i,M-1]=C[i,M-3]+5; F[i,M-1]=(C[i,M-2]+D[i,M])/2; C[i,M]=(A[i,M+2]+B[i,M+1])/2; E[i,M]=C[i,M-2]+5; F[i,M]=(C[i,M-1]+D[i,M+1])/2; endfor </pre> <p style="text-align: center;">(b)</p>
---	--

FIG. 1: (a) The original 2-level loop with six inner loops. (b) The fused loop by the ULF_IP technique.

The code shown in Figure 1(a) contains four sequential loops enclosed in one shared outermost loop. To reduce the execution time of this program, one of the solutions is to fuse all the loops. But these loops cannot be fused directly. The computation of $C[i, j]$ in the third loop requires the value of $A[i, j + 2]$ and $B[i, j + 1]$, while $A[i, j + 2]$ and $B[i, j + 1]$ has not been produced in $(i, j)^{th}$ iteration if we directly merge the loops. We call this kind of data dependence fusion-preventing dependence. The fusion-preventing dependences also exist between the computation of $F[i, j]$ and $D[i, j + 1]$ as shown in Figure 1(a), so the four inner loops cannot be fused without transformation.

The problem can be solved by the General Legalizing Loop Fusion Technique (ULF_IP) presented in [4]. The resultant code after fusing all the loops is show in Fig. 1(b). The execution time is reduced from $15 * N * M$ to $5 * N * M$ after fusing all the loops, assuming that any computation can be finished within one time unit, and there are 8 functional units. The execution time is defined to be the schedule length times the total iterations. The schedule length is the number of time units to finish one iteration of the loop body. For the sequentially executed loops, the execution time is the sum of the execution time of each individual loop. Here, N is the total number of the iterations for the outermost loop, and M is the total number of the iterations for the innermost loop. But the code size is increased from 16 to 22 instructions, because the prologue and epilogue are generated when we transform the loops to eliminate the fusion-preventing dependences.

<pre> for i=0, N for j=0, M A[i,j]=A[i,j]*2+1; endfor for j=0, M B[i,j]=B[i,j-1]*3; endfor for j=0, M C[i,j]=(A[i,j+2]+B[i,j+1])/2; endfor for j=0, M D[i,j]=B[i,j]*3; endfor for j=0, M E[i,j]=C[i,j-2]+5; endfor for j=0, M F[i,j]=(C[i,j-1]+D[i,j+1])/2 endfor endfor </pre>	<pre> for i=0, N for j=0, M A[i,j]=A[i,j]*2+1; B[i,j]=B[i,j-1]*3; D[i,j]=B[i,j]*3; endfor for j=0, M C[i,j]=(A[i,j+2]+B[i,j+1])/2; E[i,j]=C[i,j-2]+5; F[i,j]=(C[i,j-1]+D[i,j+1])/2; endfor endfor </pre>
(a)	(b)

FIG. 2: (a) The distributed loop. (b)The fused loop by the LD_MDF technique.

To improve the timing performance and maintain a reasonable code size, we first maximumly distribute the loop. The maximumly distributed loop for the original program is shown in Figure 2(a). After loop distribution, there are totally six loops. Then, we find that the six loop nodes in the loop dependence graph can be partitioned into two fusion partitions, and all the loop nodes connected by fusion-preventing dependences are partitioned into different loop partitions as illustrated in Section IV. So the loop nodes inside one loop partition can be fused into one loop directly. Thus, the six loops can be fused into two loops without transformation. The resultant code by our LD_MDF technique is shown in Figure 2(b). The execution time of the final loop in Figure 2(b) is $8 * N * M$ when there are 8 functional units, which is still much better than the original loop, but a little bit larger than the fused loop by the ULF_IP technique. The code size, however, is only 12 instructions, which is much smaller than the code size of the fused loop by the ULF_IP technique in Figure 1(b), and even smaller than the code size of the original loop shown in Figure 1(a), because loop control instructions are reduced.

Loop distribution is an important part of our technique of maximum loop distribution with direct fusion (LD_MDF). But loop distribution is not simple. All the data dependences have to be preserved when breaking one single loop into multiple small loops. The authors of [2, 3, 5] stated that loop distribution preserves dependences if all statements involved in a data dependence cycle in the original loop are placed in the same loop. We show that dependence cycle is a restriction for loop distribution for one-level loops only. For multi-level nested loops, dependence cycle is not always a restriction for loop distribution. If the summation of the edge weights of the dependence cycle satisfies a certain condition, then the statements involved in the dependence cycle can be distributed.

In this paper, we propose general loop distribution theorems for multi-level loops to state the legality conditions of loop distribution based on the understanding of loop properties on graph models. Then, we show how to conduct maximum loop distribution for multi-level loops based on the loop distribution theorems. We then propose the technique of maximum loop distribution with direct fusion (LD_MDF). The experimental results showed that the execution time of the transformed loops by our LD_MDF technique can be improved 25.3% on average compared to the original loops when there are eight functional units.

The rest of the paper is organized as follows: We introduce the basic concepts and principles related to our technique in Section II. In Section III, we first propose the loop distribution theorems to guide loop distribution. Then, based on the loop distribution theorems, we show how to conduct maximum loop distribution using an example. We propose the technique of maximum loop distribution with direct fusion (LD_MDF) in Section IV. Section V presents the experimental results. Section VI concludes the paper.

II. BASIC CONCEPTS

In this section, we provide an overview of the basic concepts and principles related to our technique.

A. Data Flow Graph

We use a multi-dimensional data flow graph (*MDFG*) to model the body of one nested loop. A *MDFG* $G = (V, E, \vec{d}, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \subseteq V \times V$ is the set of edges representing dependences, \vec{d} is a function from E to Z^n , representing the multi-dimensional delays between two nodes, where n is the number of dimensions, and t is a function from V to positive integers, representing the computation time of each node.

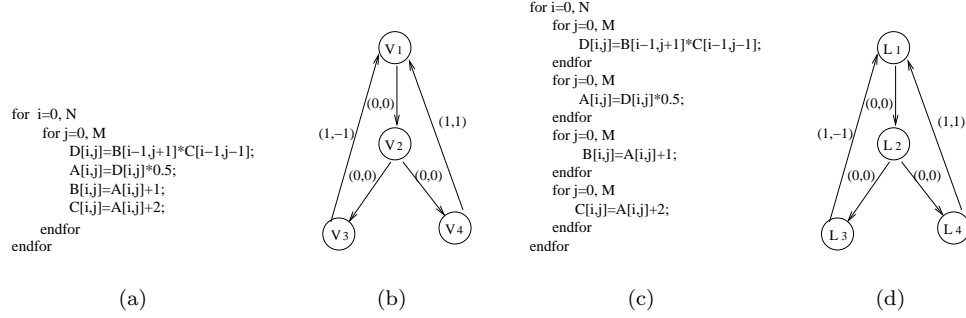


FIG. 3: (a) A loop extracted from a Wave Digital Filter. (b) The corresponding Data Flow Graph. (c) The distributed loop. (d) The loop dependence graph of the distributed loop.

The execution of each node in V exactly once represents an iteration, i.e., the execution of one instance of the loop body. Iterations are identified by a vector \vec{i} , equivalent to a multi-dimensional index. Inter-iteration dependences are represented by vector-weighted edges. For any iteration \vec{j} , an edge e from u to v with delay vector $\vec{d}(e)$ means that the computation of node v at iteration \vec{j} requires the data produced by node u at iteration $\vec{j} - \vec{d}(e)$. An edge with delay $(0, \dots, 0)$ represents a data dependence within the same iteration. A legal *MDFG* must have no zero-delay cycle, i.e., the summation of the edge weights along any cycle can not be $(0, \dots, 0)$.

The program shown in Fig. 3(a) is extracted from a wave digital filter and its corresponding data flow graph is shown in Fig. 3(b).

B. Loop Dependence Graph

Loop dependence graph (LDG) is a higher-level graph model compared to the data flow graph [4]. It is used to model the data dependences between multiple loops. A multi-dimensional loop dependence graph (MLDG) $G = (V, E, \delta, o)$ is a node-labeled and edge-weighted directed graph, where V is a set of nodes representing the loops. $E \subseteq V \times V$ is a set of edges representing data dependences between the loops. δ is a function from E to Z^n , representing the minimum data dependence vector between the computations of two loops. o is a function from V to positive integers, representing the order of the execution sequence. All the comparisons between two data dependence vectors are based on the lexicographic order in this paper.

The loop dependence graph of the loop in Fig. 4(a) is shown in Fig. 4(b). There are three nodes $V = \{L1, L2, L3\}$ in the loop dependence graph that represent the three innermost loops in the program. The loop dependence edges are $E = \{e_1 : L1 \rightarrow L2, e_2 : L1 \rightarrow L3, e_3 : L2 \rightarrow L3, e_4 : L3 \rightarrow L2, e_5 : L3 \rightarrow L1\}$. The data dependence vectors are $\{(0, -1), (0, 0)\}$ between nodes $L1$ and $L2$, $\{(0, -2), (0, -1)\}$ between nodes $L1$ and $L3$, $\{(0, 0)\}$ between nodes $L2$ and $L3$, $\{(1, 0)\}$ between nodes $L3$ and $L2$, and $\{(2, -1)\}$ between nodes $L3$ and $L1$. According to our MLDG definition, $\delta(e_1) = (0, -1)$, $\delta(e_2) = (0, -2)$, $\delta(e_3) = (0, 0)$, $\delta(e_4) = (1, 0)$, $\delta(e_5) = (2, -1)$.

In a loop dependence graph, a fusion-preventing dependence is represented by an edge e with edge weight $\delta(e) < (0, 0, \dots, 0)$. The fusion-preventing dependence edges for the LDG shown in Fig. 4(b) are e_1 and e_2 .

A *backward edge in the loop dependence graph* is defined as an edge from a node labeled with a larger number to a node labeled with a smaller number. For example, in the loop dependence graph shown in Fig. 4(b), node $L1$ represents the first inner loop of the loop shown in Fig. 4(a), which is labeled with 1 according to the execution sequence. Node $L3$ represents the third inner loop of the loop shown in Fig. 4(a), which is labeled with 3. According to the definition, in the loop dependence graph shown in Fig. 4(b), the backward edges include the edge from node $L3$ to $L1$, and the edge from $L3$ to $L2$.

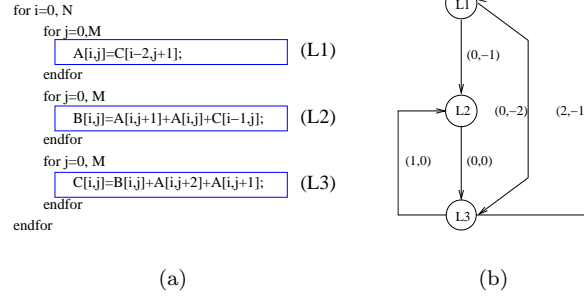


FIG. 4: A loop and its corresponding LDG.

III. LOOP DISTRIBUTION

In the process of loop distribution, we must maintain all the data dependences to ensure that we won't change the semantics of the original program. To guarantee the correctness of loop distribution, we propose general loop distribution theorems to state the legality condition of loop distribution for multi-level nested loops. Based on the legality condition of loop distribution, we show how to conduct maximum loop distribution on multi-level nested loops.

A. Theorems of Loop Distribution

Theorem III.1 *Given a N -level perfect nested loop and its corresponding data flow graph, after loop distribution, if the shared outer loop level is J , $J \leq N - 1$, then the backward edge e in the LDG $G = (V, E, \delta, o)$ of the distributed loop must satisfy that $(\delta_1(e), \dots, \delta_J(e)) \geq (0, \dots, 1)$.*

If we distribute the loop on the $(J + 1)$ -th loop level, then the distributed loop has J -level shared outer loop. A backward edge e in a LDG represents loop-carried data dependence. The first J elements in an edge weight vector $(\delta_1(e), \dots, \delta_J(e))$ represent the dependence distance of the shared outer loop. If $(\delta_1(e), \dots, \delta_J(e))$ of a backward edge e in the LDG of the distributed loop is a non-positive value, then true data dependences are changed to anti-data dependences by loop distribution. Thus, loop distribution becomes illegal.

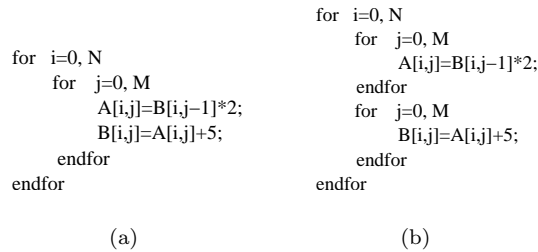


FIG. 5: An illustration example.

The code shown in Fig.5(b) computes differently from the code shown in Fig.5(a). $A[i, j]$ is dependent on $B[i, j - 1]$, which is a true data dependence in the original program as shown in Fig.5(a). If we directly distribute the loop shown in Fig.5(a), then this true data dependence becomes an anti-data dependence in the distributed loop as shown in Fig. 5(b). Therefore, any legal LDG of an N -level nested loop (distributed loop) must have a positive value on the first J elements of the weight vectors of the backward edges. Fig. 3(c) shows the distributed loop of the original program shown in Fig. 3(a). Because the backward edges $e_1 : L3 \rightarrow L1$ and $e_2 : L4 \rightarrow L1$ in the LDG shown in Fig. 3(d) have the edge weight $(1,-1)$ and weight $(1,1)$ respectively, i.e., the backward edges e_1 and e_2 both have positive value on the first element, the correct execution of the original loop is able to be preserved in the distributed loop.

When there are dependence cycles existing in the data flow graph of a loop, it's important to know whether the computations involved in a dependence cycle can be distributed or not. In the following, we show that the nodes in the dependence cycles of the LDG of a N -level nested loop can be *completely distributed* when the necessary condition in Theorem III.2 is satisfied. A loop is completely distributed when each loop unit after distribution only has one array assignment statement.

Theorem III.2 *Given a N -level perfect nested loop and its corresponding data flow graph $G = (V, E, \vec{d}, t)$,*

1. *if there is no dependence cycle in the data flow graph G ,*
2. *or if for any dependence cycle $c = \{v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n \rightarrow v_1\}$ in G , the summation $\vec{d}(c)$ of the edge weights of cycle c satisfies that $(\vec{d}_1(c), \dots, \vec{d}_J(c)) \geq (0, \dots, 1), J \leq N - 1$,*

then the loop can be completely distributed on the $(J + 1)$ -th level loop until the N -th level.

Theorem III.2 directly follows from Theorem III.1, since all the data dependences in the LDG of the distributed loop come from the data dependences of the original loop. Theorem III.2 also shows that if we can distribute the loop at the J -th ($J < N$) loop level for a N -level loop, we can distribute this N -level loop at the innermost loop level.

For example, there is one cycle in the data flow graph of the 3-level loop shown in Fig. 6(a). The summation of the edge weights of the dependence cycle c in the corresponding data flow graph has the property that $d_1(c) = 1$. According to theorem III.2, we can distribute the loop at the second loop level. The distributed loop is shown in Fig. 6(b). There is one-level shared outer loop in the distributed loop. Also we can distribute the innermost loop of this 3-level loop, and the distributed loop is shown in Fig. 6(c).

<pre> for i=0, N for j=0, M for k=0, S A[i,j,k]=C[i-1,j,k]+7; endfor endfor for j=0, M for k=0, S B[i,j,k]=A[i,j,k]*3; C[i,j,k]=B[i,j,k]+5; endfor endfor endfor </pre> <p style="text-align: center;">(a)</p>	<pre> for i=0, N for j=0, M for k=0, S A[i,j,k]=C[i-1,j,k]+7; endfor endfor for j=0, M for k=0, S B[i,j,k]=A[i,j,k]*3; endfor endfor for j=0, M for k=0, S C[i,j,k]=B[i,j,k]+5; endfor endfor endfor </pre> <p style="text-align: center;">(b)</p>	<pre> for i=0, N for j=0, M for k=0, S A[i,j,k]=C[i-1,j,k]+7; endfor endfor for k=0, S B[i,j,k]=A[i,j,k]*3; endfor for k=0, S C[i,j,k]=B[i,j,k]+5; endfor endfor </pre> <p style="text-align: center;">(c)</p>
--	--	--

FIG. 6: (a) A 3-level nested loop. (b) The distributed loop with 1-level outer shared loop. (c) The distributed loop with 2-level outer shared loop.

Theorem III.3 identifies the dependence cycle in the data flow graph that prevents the statements involved in the dependence cycle from distribution.

Theorem III.3 *Given a N -level nested loop and its corresponding data flow graph $G = (V, E, \vec{d}, t)$, for any dependence cycle $c = \{v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n \rightarrow v_1\}$ in G , such that the summation of the edge weights $\vec{d}(c)$ satisfies that $(\vec{d}_1(c), \dots, \vec{d}_J(c)) = (0, \dots, 0)$, the statements involved in dependence cycle c must be placed in the same loop after the loop is distributed on the $(J + 1)$ -th loop level.*

If we distribute the statements in a dependence cycle c with $(\vec{d}_1(c), \dots, \vec{d}_J(c)) = (0, \dots, 0)$, then there will be a dependence cycle c with $(\vec{d}_1(c), \dots, \vec{d}_J(c)) = (0, \dots, 0)$ in the correspondent LDG of the distributed loop. For a legal program, all the edges e involved in a dependence cycle c in the LDG must have the property $(\vec{d}_1(c), \dots, \vec{d}_J(c)) \geq (0, \dots, 0)$. So the backward edge e' in the cycle c must have $(\vec{d}_1(e'), \dots, \vec{d}_J(e')) = (0, \dots, 0)$, which is contradictory to Theorem III.1. When the first J elements of the summation of the edge weights of a cycle are all zeros, it indicates that all the statements in the cycle have to be computed within one loop. In other words, all the statements involved in a dependence cycle c with $(\vec{d}_1(c), \dots, \vec{d}_J(c)) = (0, \dots, 0)$ must be put into one loop after the loop is distributed on the $(J + 1)$ -th loop level.

B. Maximum Loop Distribution

In this subsection, we use an example to show how to conduct maximum loop distribution for multi-level nested loops based on the loop distribution theorems proposed in Section III A.

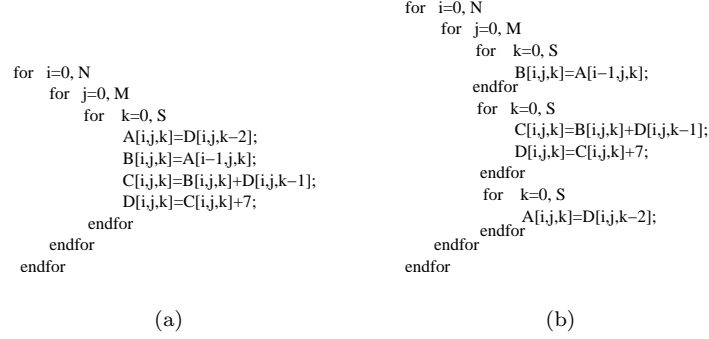


FIG. 7: (a) An example loop. (b) The distributed loop.

To conduct maximum loop distribution for a multi-level loop on the innermost loop level, we first remove the edges e with weight $(\vec{d}_1(e), \dots, \vec{d}_{N-1}(e)) \geq (0, \dots, 1)$ from the LDG of the given loop. Thus, if the summation $\vec{d}(c)$ of the edge weights of a cycle c in the LDG satisfies that $(\vec{d}_1(c), \dots, \vec{d}_{N-1}(c)) \geq (0, \dots, 1)$, then cycle c is broken. Then, we merge each cycle c with $(\vec{d}_1(c), \dots, \vec{d}_{N-1}(c)) = (0, \dots, 0)$ into one node. This is used to guarantee that the statements involved in the dependency cycle whose summation of the edge weights satisfies that $(\vec{d}_1(c), \dots, \vec{d}_{N-1}(c)) = (0, \dots, 0)$ will be put into the same loop after loop distribution. Then, we can reorder the nodes by the topological order to ensure that the edge weight $\delta(e)$ of a backward edge e in the LDG of the distributed loop has positive value on its first $N - 1$ elements, i.e., $(\delta_1(e), \dots, \delta_{N-1}(e)) \geq (0, \dots, 1)$. Every node in the transformed graph corresponds to a loop unit in the distributed loop.

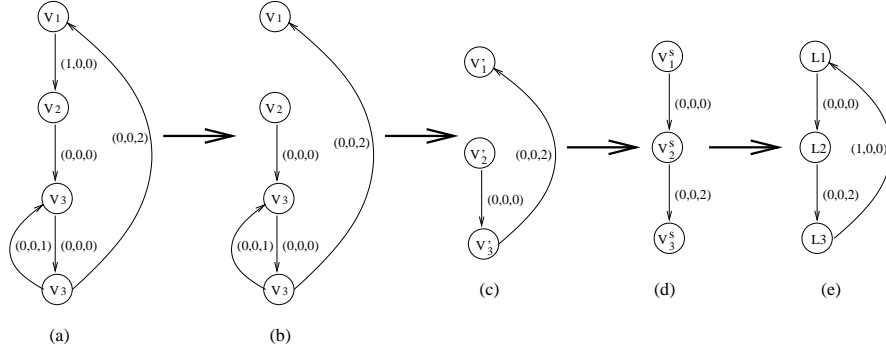


FIG. 8: The graph transformation process of the maximum loop distribution algorithm.

We use a 3-level nested loop as shown in Fig. 7(a) to show the graph transformation process of maximum loop distribution for N -level nested loops. There are two cycles c_1, c_2 in its corresponding data flow graph as shown in Fig. 8(a). The summation of the edge weights of the first cycle $c_1 = \{V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_1\}$ satisfies that $\vec{d}(c_1) = (1, 0, 2)$, so cycle c_1 will be broken after the edges e with $(\vec{d}_1(e), \vec{d}_2(e)) \geq (0, 1)$ are removed. The summation of the edge weights of the second cycle $c_2 = \{V_3 \rightarrow V_4 \rightarrow V_3\}$ has the property that $\vec{d}(c_2) = (0, 0, 1)$, so cycle c_2 will be merged into one node according to the basic idea of maximum loop distribution since $(\vec{d}_1(c), \vec{d}_2(c)) = (0, 0)$. Thus, we get a DAG G' . Then, we perform topological sort on graph G' and obtain the node-reordered graph G^s shown in Fig. 8(d). Each node in the graph G^s corresponds to one loop unit in the distributed loop. According to the sorted nodes, we can generate the code of the distributed loop as shown in Fig. 7(b). Then we can get the LDG of the distributed loop as shown in Fig. 8(e). Fig. 8 shows the graph transformation process.

IV. LOOP DISTRIBUTION WITH LOOP FUSION

In this section, we first illustrate direct loop fusion using an example. Then, we propose the technique of maximum loop distribution with maximum direct fusion (LD_MDF), which performs maximum loop distribution followed by maximum direct loop fusion.

A. Direct Loop Fusion

Direct loop fusion is to find the legal fusion partition of the loop nodes so that the loop nodes inside one partition can be fused directly. To apply direct loop fusion, we partition the loop nodes in the LDG into several partitions so that loop nodes connected by a fusion-preventing dependence edge are partitioned into different partitions. Maximal loop fusion is to minimize the number of the fusion partitions, thus the resultant number of the fused loops is minimized. A fusion partition is a partition of loop nodes V in the loop dependence graph into different partitions: each partition represents a set of loops to be fused.

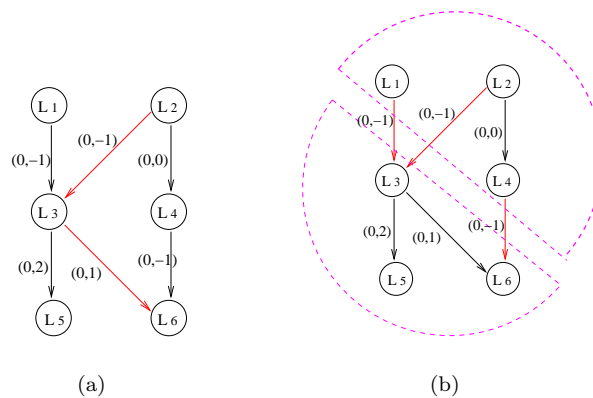


FIG. 9: (a) The LDG of the distributed loop in Figure 2(a). (b) The graph partitioning result.

The distributed loop of the example loop in Figure 1(a) is shown in Figure 2(a), and its corresponding LDG is shown in Figure 9(a). We can see there are three fusion-preventing dependences in the LDG. Using our graph partitioning method, the partition number we got for the loop nodes in the LDG are as follows: $P(L1) = 1; P(L2) = 1; P(L3) = 2; P(L4) = 1; P(L5) = 2; P(L6) = 2$. So the partition results are $Partition_1 = \{L1, L2, L4\}$, and $Partition_2 = \{L3, L5, L6\}$ as shown in Figure 9(b). There are totally two fusion partitions according to this partitioning result, which achieves the maximal fusion, since there are totally three fusion-preventing dependence edges in the original LDG. The fused loop is shown in Figure 2(b).

B. Loop Distribution with Direct Loop Fusion

The technique of loop distribution with maximum direct loop fusion (LD_MDF) combines loop distribution with maximum direct loop fusion to improve the timing performance of the loops without the increase of the code size. The basic idea and implementation of the LD_MDF technique are illustrated as follows:

1. Apply maximum loop distribution on the given loop.
2. Construct the corresponding loop dependence graph of the distributed loop.
3. Apply graph partitioning algorithm to compute the fusion partitions.
4. Get the fused loop by fusing all the loop nodes in the same fusion partition.

The technique of loop distribution with maximum direct loop fusion first applies maximum loop distribution on a given loop. After we perform maximum loop distribution on the given loop, we construct the loop dependence graph

of the distributed loop. Then, we partition the loop nodes in the LDG of the distributed loop so that there is no fusion-preventing dependences existing between the nodes inside one fusion partition. Thus, all the loop nodes inside one fusion partition can be directly fused [3]. After we get the fusion partitions, direct loop fusion is applied on each fusion partition. According to the implementation procedure of the LD_MDF technique, the number of the fused loops by the LD_MDF technique is always smaller than the number of the loops in the original program. Correspondingly, we can conclude that the code size of the fused loops by the LD_MDF technique is always smaller than the code size of the original loops as stated in Corollary IV.1.

Corollary IV.1 *Given a multi-level loop and its corresponding data flow graph, after we apply the technique of loop distribution with maximum direct loop fusion (LD_MDF) to fuse the loops, the code size of the transformed loop is always smaller than the code size of the original loop.*

For example, the code shown in Figure 1(a) contains four sequential loops enclosed in one shared outermost loop. To apply our LD_MDF technique, we first maximumly distribute the loop. The maximumly distributed loop for the original program is shown in Figure 2(a). After loop distribution, there are totally six loops. Then, we partition the six loop nodes in the loop dependence graph as shown in Figure 9(a) into two fusion partitions as shown in Figure 9(b). We can see that all the loop nodes connected by fusion-preventing dependences are partitioned into different loop partitions. So the loop nodes inside one loop partition can be fused into one loop directly. Thus, the six loops can be fused into two loops without transformation. The fused loop by our LD_MDF technique is shown in Figure 2(b), which has two inner loops. The number of the loops in the fused loop is less than the number of the loops in the original loop. The code size of the fused loop is 12 instructions, which is also smaller than the code size of the original loop, which is 16 instructions, since the loop control instructions are reduced.

V. EXPERIMENTS

In our experiments, we performed the general legalizing loop fusion technique (ULF_IP) presented in [4] and the technique of loop distribution with maximum direct loop fusion (LD_MDF). All the test cases are extracted from real DSP applications, most of them are extracted from the real filters including WDF (Wave Digital filter), IIR (Infinite Impulse Response filter), DPCM (Differential Pulse-Code Modulation device), and 2D (Two Dimensional filter). The VLIW architecture is used as the test platform. We simulated a VLIW architecture based DSP processor with eight functional units. We compare the timing performance and the code size of the original loops, the fused loops by the ULF_IP technique, and the fused loop by the LD_MDF technique. The experiments are performed on a Dell PC with a P4 2.1G processor and 512MB memory running Red Hast Linux 9.0. Every experiment is finished within one minute.

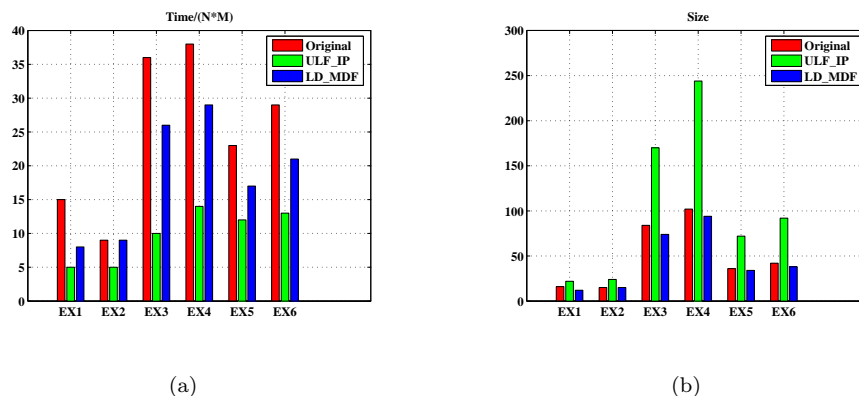


FIG. 10: (a) The Execution Time of the original loops and the fused loops by various techniques. (b) The Code Size of the original loops and the fused loops by various techniques.

EX1 refers to the example loop shown in Figure 1(a). EX2, EX3, and EX4 refer to the DSP applications presented in [6] that have several loops, including WDF (Wave Digital filter), IIR (Infinite Impulse Response filter), DPCM (Differential Pulse-Code Modulation device), and 2D (Two Dimensional filter). EX5 and EX6 refer to the example loops presented in [4].

In Figure 10(a), we compare the execution time of the original loops and the fused loops by the ULF_IP technique and the LD_MDF technique. The execution time is defined to be the schedule length times the total iterations. The schedule length is the number of time units to finish one iteration of the loop body. We assume that each computation can be finished in one time unit. For the sequentially executed loops, the execution time is the sum of the execution time of each individual loop. N denotes the total number of the iterations for the outermost loop, and M denotes the total number of the iterations for the innermost loop in Figure 10(a). Figure 10(b) compares the code size of the original loops and the fused loops by the ULF_IP technique and the LD_MDF technique. We calculate the code size by the number of instructions.

Although the ULF_IP technique proposed in [4] can always achieve a shorter execution time than the LD_MDF technique, it increases the code size. In many cases, this technique cannot be applied because of the memory constraint. Compared to the ULF_IP technique, the LD_MDF technique takes advantage of both loop distribution and loop fusion, so it reduces the original execution time and also avoids the code-size expansion. The experimental results showed that the timing performance of the fused loop by our LD_MDF technique can be improved 25.3% on average compared to the original loops, and the code size is reduced 10.0% on average compared to the original loops.

VI. CONCLUSION

In this paper, we developed the technique of combining loop distribution with maximum direct loop fusion (LD_MDF) to achieve a shorter execution time with a reduction of the code size. We first proposed loop distribution theorems for multi-level loops to guide loop distribution. Then we showed how to conduct maximum loop distribution on multi-level nested loops. We then proposed the technique of maximum loop distribution with maximum direct loop fusion (LD_MDF), which is to perform the maximum loop distribution followed with direct loop fusion. We also showed that the resultant code size of the fused loop by the LD_MDF technique will be always smaller than the code size of the original loop. The experimental results showed that our LD_MDF technique can reduce the execution time without increasing the code size of the fused loop.

-
- [1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.
 - [2] K. Kennedy and K. S. Mckinley. Loop distribution with arbitrary control flow. In *Proc. of the 1990 conference on Supercomputing*, pages 407 – 416, Nov. 1990.
 - [3] K. Kennedy and K. S. Mckinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, Number 768*, pages 301–320, 1993.
 - [4] M. Liu, Q. Zhuge, Z. Shao, and E. H.-M. Sha. General loop fusion technique for nested loops considering timing and code size. In *Proc. ACM/IEEE International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2004)*, pages 190–201, Sep. 2004.
 - [5] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424 – 453, July 1996.
 - [6] E. H.-M. Sha, T. W. O’Neil, and N. L. Passos. Efficient polynomial-time nested loop fusion with full parallelism. *International Journal of Computers and Their Applications*, 10(1):9–24, Mar. 2003.
 - [7] S. Verdoolaege, M. Bruynooghe, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Proc. of the Application-Specific Systems, Architectures, and Processors*, pages 14–24, 2003.
 - [8] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Inc., 1996.
 - [9] Q. Zhuge, B. Xiao, and E.-M. Sha. Code size reduction technique and implementation for software-pipelined DSP applications. *ACM Transactions on Embedded Computing Systems(TECS)*, 2(4):590–613, Nov. 2003.