# Maximum Loop Distribution and Fusion for Two-level Loops Considering Code Size *

Meilin Liu[1] Qingfeng Zhuge[1] Zili Shao[2] Chun Xue[1] Meikang Qiu[1] Edwin H.-M. Sha[1]
[1]Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083, USA
[2]Department of Computing
Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong

## Abstract

*In this paper, we propose a technique combining loop distribution with loop fusion to improve the timing performance without increasing the code size of the transformed loops. We first develop the loop distribution theorems that state the conditions distributing any two-level nested loop in the maximum way. Based on the loop distribution theorems, we design an algorithm to conduct maximum loop distribution. Then we propose a technique of maximum loop distribution with direct loop fusion, which performs maximum loop distribution followed by direct loop fusion. The experimental results show that the execution time of the transformed loops by our technique is reduced 41.9% on average compared to the original loops without the increase of the code size.*

## 1 Introduction

Loop distribution separates independent statements inside a single loop (or loop nest) into multiple loops (or loop nests) [6, 1, 2, 4]. Loop distribution can enable other loop optimization techniques such as loop fusion, loop interchanging and loop permutation [6, 1, 2, 4].

Loop fusion, on the other hand, groups multiple loops to increase the instruction-level parallelism, and correspondingly reduces execution time. There are a lot of previous works using loop fusion to optimize the execution of loops [2, 6, 1, 5, 3, 4]. But sometimes loop shifting or retiming is needed to enable loop fusion, which will cause the code size expansion because of the generation of the prologue and epilogue. The maximum loop fusion technique (Max_LF) proposed in [3] can maximize the opportunities of loop fusion, but it cannot be applied in the cases where strict memory constraint applies.

In this paper, we propose a technique of *maximum loop distribution with direct fusion* (MLD_DF), which performs maximum loop distribution, followed by *direct loop fusion*. The technique significantly improves the timing performance compared to the original loops without jeopardizing the code size.

*This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001 and NSF CCR-0309461, USA.

The code shown in Fig. 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 $E[i, j]$ in the second loop requires the value of $B[i, j + 2]$ in the first loop, while $B[i, j + 2]$ has not been produced in iteration $(i, j)$ if we directly merge the loops. This kind of data dependence is called fusion-preventing dependence. The fusion-preventing dependences also exist between the computation of $H[i, j]$ and $B[i, j + 1]$, the computation of $I[i, j]$ and $D[i, j + 1]$, and the computation of $J[i, j]$ and $F[i, j + 1]$ as shown in Fig. 1(a), so the four inner loops cannot be fused without transformation. The problem can be solved by the Max_LF (Maximum Loop Fusion) technique presented in [3]. The execution time is reduced from $16 * N * M$ 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. But the code size is increased from 20 to 38 instructions, because the prologue and epilogue are generated when we transform the loops to eliminate the fusion-preventing dependences.

```
for  i=0, N
     for  j=0, M
          A[i,j]=J[i−1,j]+5;
          B[i,j]=A[i,j]*3;
     endfor
     for  j=0, M
          C[i,j]=A[i−1,j]+7;
          D[i,j]=C[i,j−1]*2;
          E[i,j]=D[i,j]+B[i,j+2];
     endfor
     for  j=0, M
          F[i,j]=A[i,j]*4;
          G[i,j]=F[i,j]*2;
          H[i,j]=B[i,j+1]+G[i,j−1];
     endfor
     for  j=0, M
          I[i,j]=J[i,j−1]+D[i,j+1];
          J[i,j]=I[i,j]+F[i,j+1];
     endfor
endfor
```

(a)

```
for  i=0, N
     for  j=0, M
          A[i,j]=J[i−1,j]+5;
          B[i,j]=A[i,j]*3;
          C[i,j]=A[i−1,j]+7;
          D[i,j]=C[i,j−1]*2;
          F[i,j]=A[i,j]*4;
     endfor
     for  j=0, M
          E[i,j]=D[i,j]+B[i,j+2];
          G[i,j]=F[i,j]*2;
          H[i,j]=B[i,j+1]+G[i,j−1];
          I[i,j]=J[i,j−1]+D[i,j+1];
          J[i,j]=I[i,j]+F[i,j+1];
     endfor
endfor
```

(b)

**Figure 1.** (a) The original 2-level loop with four inner loops. (b) The fused loop.

We propose a technique of maximum loop distribution with direct fusion (MLD_DF) to improve the instruction-level parallelism

and maintain a reasonable code size. After loop distribution, there are totally nine loops. Then, we find that the nine loops can be fused into two loops without dealing with fusion-preventing dependences, therefore no transformation is involved. The final loop by our MLD_DF technique is shown in Fig. 1(b). The execution time of the final loop in Fig. 1(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 Max_LF technique. The code size, however, is only 16 instructions, which is much smaller than the code size of the fused loop by the Max_LF technique, and even smaller than the original code size in Fig. 1(a), because loop control instructions are reduced.

Loop distribution is an important part of our technique of maximum loop distribution with direct fusion (MLD_DF). 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, 4] 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 two-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 two-level loops based on the understanding of loop properties on graph models. Then, we design an algorithm of maximum loop distribution for two-level loops based on the loop distribution theorems. We then present the technique of maximum loop distribution with direct fusion (MLD_DF). Finally, based on the maximum loop fusion (Max_LF) technique proposed in [3] and the MLD_LF technique, we propose a technique of loop distribution and fusion considering memory constraint (LD_LF_MEM), which is to select one of the two techniques, the Max_LF technique and the MLD_DF technique as the final solution considering the memory constraint of the target processors. The experimental results show that the timing performance of the transformed loops by our LD_LF_MEM technique is improved 41.9% compared to the original loops on average, and the code size still satisfies the memory constraint of the target processors.

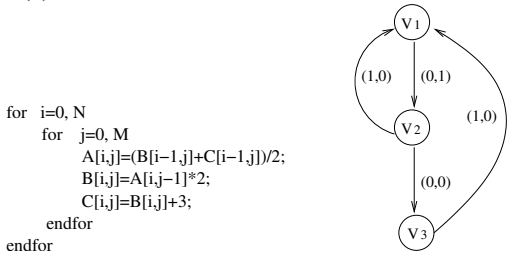
The rest of the paper is organized as follows: We introduce the basic concepts and principles related to our technique in Section 2. Section 2.3 introduces our loop fusion technique. In Section 3 we present the loop distribution theorems to guide the loop distribution. We propose the algorithm of maximum loop distribution, the MLD_LF technique and the LD_LF_MEM technique in Section 4. Section 5 presents the experimental results. Section 6 concludes the paper.

# 2 Basic Concepts

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

## 2.1 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. We use $d(e) = (d.x, d.y)$ as a general formulation of any delay shown in a two-dimensional DFG (2DFG). A two-level loop is shown in Fig. 2(a) and its corresponding data flow graph is shown in Fig. 2(b).

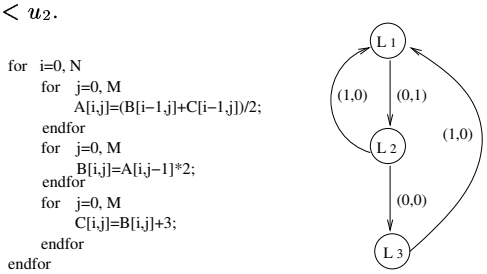



(a) (b)

**Figure 2.** (a)A two-level loop and its corresponding DFG.

## 2.2 Loop Dependence Graph

Loop dependence graph is a higher-level graph model compared to the data flow graph [3]. 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. $\delta$ 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. For example, in the two-dimensional case, a vector $\vec{v} = (v_1, v_2)$ is smaller than a vector $\vec{u} = (u_1, u_2)$ according to the lexicographic order if either $v_1 < u_1$ or $v_1 = u_1$ and $v_2 < u_2$.




(a) (b)

**Figure 3.** A loop and its corresponding LDG.

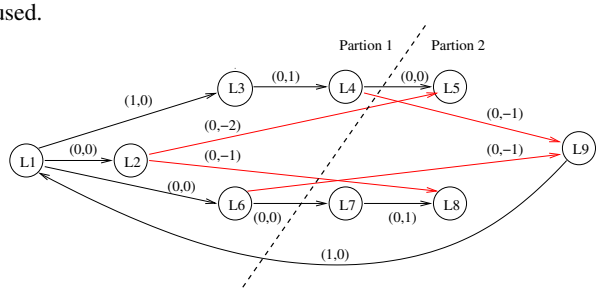
The loop dependence graph of the loop in Fig. 3(a) is shown in Fig. 3(b). In a loop dependence graph, a fusion-preventing dependence is represented by an edge $e$ with edge weight $\delta(e) < (0, \cdots, 0)$.

*A backward edge of 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. 3(b), node $L1$ represents the first inner loop of the

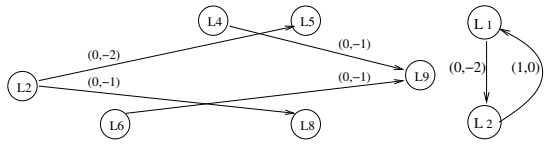
loop shown in Fig. 3(a), which is labeled with 1 according to the execution sequence. Node $L3$ represents the third inner loop of the loop shown in Fig. 3(a), which is labeled with 3. According to the definition, in the loop dependence graph shown in Fig. 3(b), the backward edges include the edge from node $L3$ to $L1$, and the edge from $L2$ to $L1$.

### 2.3 Loop Fusion Techniques

Direct loop fusion is to fuse two or more loops into one loop when there are no fusion-preventing dependences between these loops. Our direct loop fusion technique is based on the loop dependence graph, and it is mapped to the graph partitioning technique [2, 4]. 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. Thus, we guarantee that there is no fusion-preventing dependence existing between the nodes inside one partition and all the loop nodes inside one partition can be directly fused.




(a)




(b) (c)

**Figure 4.** (a) The LDG of the distributed loop of the motivation example shown in Fig. 1(a). (b) The graph with only the fusion-preventing dependence edges in the LDG shown in Fig. 4(a). (c) The LDG of the final loop shown in Fig. 1(b).

For the example loop shown in Fig. 1(a), the LDG of the distributed loop is shown in Fig. 4(a). To guarantee that the loop nodes in the LDG $G$ connected by the fusion-preventing dependences will be put into different partitions, we construct a graph $G^p$ with only the fusion-preventing dependence edges in $G$ as shown in Fig. 4(b), which has Nodes $L2, L4, L5, L6, L8, L9$. We partition the nodes in graph $G^p$ such that the nodes connected by the fusion-preventing dependence edges are put into different partitions. The resultant partitions for the nodes in Fig. 4(b) are $P_1 = \{L2, L4, L6\}$, $P_2 = \{L5, L8, L9\}$. All the other nodes can be put into partition $P_1$ and $P_2$ based on the data dependences. The final partitions are $P_1 = \{L1, L2, L3, L4, L6\}$, $P_2 = \{L5, L7, L8, L9\}$. Therefore, there are two loop nodes in the corresponding LDG of the fused loop as shown in Fig. 4(c) and Fig. 1(b) shows the final loop by the technique of maximum loop distribution with direct fusion.

## 3 Theorems of 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 several theorems to guide loop distribution for two-level loops.

**Theorem 3.1** *Given a two-level nested loop and its corresponding data flow graph, the backward edge $e$ in the LDG $G = (V, E, \delta, o)$ of the distributed loop must satisfy that $\delta_1(e) \geq 1$, where $\delta_1(e)$ denotes the first element of the weight vector $\delta(e)$.*

A backward edge $e$ in a LDG represents loop-carried data dependence. The first element in an edge weight vector $\delta_1(e)$ represents the dependence distance of the outermost loop. If $\delta_1(e)$ 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. Then, loop distribution becomes illegal. Obviously, 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 a two-level nested loop (distributed loop) must have a positive value on the first element of the weight vectors of the backward edges.

```
for  i=0, N
     for  j=0, M
          A[i,j]=B[i,j–1]*2;
          B[i,j]=A[i,j]+5;
     endfor
endfor
```

(a)

```
for  i=0, N
     for  j=0, M
          A[i,j]=B[i,j–1]*2;
     endfor
     for  j=0, M
          B[i,j]=A[i,j]+5;
     endfor
endfor
```

(b)

**Figure 5. An illustration example**

Fig. 3(a) shows the distributed loop of the original program shown in Fig. 2(a). Because the backward edges $e_1 : L3 \rightarrow L1$ and $e_2 : L2 \rightarrow L1$ in the LDG shown in Fig. 3(b) both have the edge weight (1,0), 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 the original 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 dependence cycles of a two-level nested loop can be *completely distributed* when the necessary condition in Theorem 3.2 is satisfied. A loop is completely distributed when each loop unit after distribution only has one array assignment statement.

**Theorem 3.2** *Given a two-level nested loop and its corresponding data flow graph $G = (V, E, \vec{d}, t)$, if there is no dependence*

*cycle in the data flow graph G, or for any dependence cycle* $c = \{v_1 \rightarrow \cdots \rightarrow v_n \rightarrow v_1\}$ *in G, the summation of the edge weights* $\vec{d}(c)$ *satisfies that* $\vec{d}_1(c) \geq 1$*, then the loop can be completely distributed on the innermost loop level.*

Due to the space limit, we do not give a formal proof here. It directly follows from Theorem 3.1, since all the data dependences in the LDG of the distributed loop come from the data dependences of the original loop. If the data flow graph of the original program has no dependence cycle, then the LDG of the distributed loop has no dependence cycle too. There is no backward edge in the loop dependence graph of the distributed loop, so the legality condition for loop distribution stated in theorem 3.1 can be satisfied automatically in this case.

If the data flow graph of the original program has a dependence cycle $c$ that has a positive value on the first element of the summation of the edge weights of this cycle, i.e., $d_1(c) \geq 1$, then there will be a dependence cycle in the correspondent LDG of the distributed loop that has the same summation of the edge weights. We can always reorder the nodes inside a cycle in LDG to make the first element of the edge weight of a backward edge $e$ satisfy $\delta_1(e) \geq 1$, such that the legality condition for the loop distribution stated in theorem 3.1 can be satisfied.

For example, in the 2-level loop shown in Fig. 2(a), there are two cycles $c_1 = \{V_1 \rightarrow V_2 \rightarrow V_1\}$ and $c_2 = \{V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_1\}$ in the data flow graph as shown in Fig. 2(b). The summation of the edge weights of the dependence cycle $c_1$ and $c_2$ in the data dependence graph has the property that $\vec{d}_1(c) = 1$, and $\vec{d}_2(c) = 1$ respectively. According to Theorem 3.2, the statements involved in the cycle $c_1$ and $c_2$ can be completely distributed, so the innermost loop can be distributed even there are two dependence cycles $c_1$ and $c_2$ in the data flow graph and the distributed loop is shown in Fig. 3(a).

Theorem 3.3 identifies the dependence cycle in the data flow graph that prevents the statements involved in the dependence cycle from distribution.

**Theorem 3.3** *Given a two-level nested loop and its corresponding data flow graph* $G = (V, E, \vec{d}, t)$*, for any dependence cycle* $c = \{v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1\}$ *in G, such that the summation of the edge weights* $\vec{d}(c)$ *satisfies that* $\vec{d}_1(c) = 0$*, the statements involved in dependence cycle c must be placed in the same loop after the loop is distributed on the innermost loop level.*

If we distribute the statements involved in a dependence cycle $c$ with $\vec{d}_1(c) = 0$, then there will be a dependence cycle $c$ with $\delta_1(c) = 0$ 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 $\delta_1(e) \geq 0$. So the backward edge $e'$ in the cycle $c$ must have $\delta_1(e') = 0$, which is contradictory to Theorem 3.1. When the first element of the summation of the edge weights of a cycle is zero, 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 $\vec{d}_1(c) = 0$ must be put into one loop after the loop is distributed.

For example, in the 2-level loop shown in Fig. 6(a), there is a cycle $c$ with $d_1(c) = 0$ in the data flow graph of this loop. According to Theorem 3.3, the statements involved in the cycle must be placed in the same loop after loop distribution and the distributed loop is shown in Fig. 6(b).

```
for  i=0, N
     for   j=0, M
           A[i,j]=B[i,j−1]+C[i−1,j];
           B[i,j]=A[i,j]+5;
           C[i,j]=B[i,j]*3;
     endfor
endfor
```

(a)

```
for i=0, N
     for   j=0, M
           A[i,j]=B[i,j−1]+C[i−1,j];
           B[i,j]=A[i,j]+5;
     endfor
     for   j=0, M
           C[i,j]=B[i,j]*2;
     endfor
endfor
```

(b)

**Figure 6.** (a)The original loop. (b) The distributed loop.

One-level loops are the special cases of the two-level nested loops when the two-level nested loop is executed in the row-wise execution order. Loop distribution on one-level loops is a special case of loop distribution on two-level nested loops when a singular edge weight in DFG of the one-level loop is replaced by a 2-D weight vector whose first element is always zero. In other words, the distance of the outer loop-carried dependence of a one-level loop is virtually zero when we treat it as a two-level loop. As a result, computations in a dependence cycle of a one-level loop cannot be distributed in any circumstance. This result, which directly follows from Theorem 3.3, actually is very different from the case of nested loops. Corollary 3.1 states the legality condition of loop distribution for one-level loops.

**Corollary 3.1** *Given a one-level loop and its corresponding data flow graph* $G = (V, E, d, t)$*, if the data flow graph G contains cycles, then the statements involved in a dependence cycle must be placed in one loop after loop distribution.*

Theorem 3.1, Theorem 3.2 and Theorem 3.3 all can be easily generalized to the $N$-level nested loops based on the understanding of the basic properties of the $N$-level nested loops. Due to the space limitation, we do not present the loop distribution theorems for $N$-level nested loops in detail.

# 4 Loop Distribution and Loop Fusion

In this section, we first propose the Maximum Loop Distribution algorithm for two-level nested loops based on the loop distribution theorems proposed in Section 3. Then, we present the technique of maximum loop distribution with direct fusion (MLD_DF). Finally, based on the Max_LF technique proposed in [3] and the MLD_DF technique, we propose the technique of loop distribution and fusion considering memory constraint (LD_LF_MEM).

**Maximum Loop Distribution**

Based on the legality condition of loop distribution, we develop an algorithm of maximum loop distribution for two-level nested loops as shown in algorithm 4.1. In algorithm 4.1, we first remove the edges $e$ with edge weight $\vec{d}_1(e) \geq 1$ from the DFG. Thus, if the summation $\vec{d}(c)$ of the edge weights of a cycle $c$ satisfies that $\vec{d}_1(c) \geq 1$, then cycle $c$ is broken. Then, we merge each cycle $c$ with $\vec{d}_1(c) = 0$ into one node. This is used to guarantee that the statements involved in the dependence cycle $c$ with $\vec{d}_1(c) = 0$ 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 element, i.e., $\delta_1(e) \geq 1$. Every node in the transformed graph corresponds to a loop unit in the distributed loop.

**Algorithm 4.1** Maximum Loop Distribution Algorithm for Two Level Nested Loops

**Require:** A loop and its corresponding data flow graph $G = (V, E, \vec{d}, t)$
**Ensure:** LDG of the distributed loop
/* Remove all the edges $e$ with $\vec{d}_1(e) \geq 1$ from $E$ */
$E' \leftarrow E - \{ e \mid \vec{d}_1(e) \geq 1\}$
Merging each cycle $c$ with $\vec{d}_1(c) = 0$ in the Graph $G$ into one node and obtaining graph $G'$
$G^s \leftarrow TOPOLOGICAL - SORT(G')$
Generate the distributed loop based on the graph $G^s$
$G_L \leftarrow$ LDG of the distributed loop
return $G_L$

We use a 2-level nested loop as shown in Fig. 6(a) to show the graph transformation process of the loop distribution as shown in Fig. 7. There are two cycles $c_1, c_2$ in its corresponding data flow graph as shown in Fig. 7(a). The summation of the edge weights of the cycle $c_1 = \{V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_1\}$ satisfies that $\vec{d}(c_1) = (1, 0)$, so cycle $c_1$ is broken after the edges $e$ with $\vec{d}_1(e) \geq 1$ are removed as shown in Fig. 7(b). The summation of the edge weights of the cycle $c_2 = \{V_1 \rightarrow V_2 \rightarrow V_1\}$ has the property that $\vec{d}(c_2) = (0, 1)$, so cycle $c_2$ is merged into one node since $\vec{d}_1(c_2) = 0$. Thus, we get a DAG $G'$ as shown in Fig. 7(c). Then, we perform the topological sort on graph $G'$ and obtain the node-reordered graph $G^s$ shown in Fig. 7(d). Each node in the graph $G^s$ corresponds to one loop unit in the distributed loop. According to the sorted nodes, we generate the code of the distributed loop as shown in Fig.6(b).

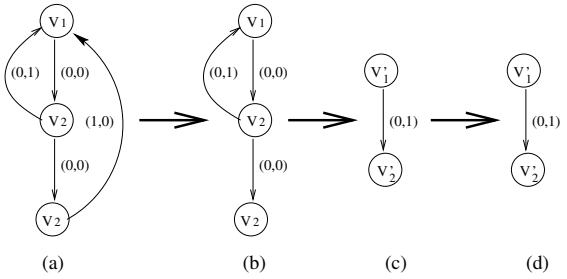



**Figure 7.** The graph transformation process of the maximum loop distribution algorithm.

**Maximum Loop Distribution with Direct Fusion**

The technique of maximum loop distribution with direct fusion (MLD_DF) tries to combine loop distribution with loop fusion to improve the timing performance of the loops without the increase of the code size. We first apply maximum loop distribution on a given loop. After we perform maximum loop distribution, we partition the loop nodes in the LDG of the distributed loop such that there is no fusion-preventing dependences existing between the nodes inside one partition and all the loop nodes inside one partition can be directly fused [2]. Then, direct loop fusion is applied. The technique of direct loop fusion has been proposed in Section 2.3.

**Loop Distribution and Fusion Considering Memory Constraint**

Timing performance and code size are two major concerns for embedded systems with very limited on-chip memory resources [7]. We develop the technique of loop distribution and fusion considering memory constraint (LD_LF_MEM) to satisfy the high performance requirement with memory constraint. The LD_LF_MEM technique is based on the Max_LF and the MLD_DF technique and it selects one of these two techniques, the Max_LF technique and the MLD_DF technique as the final solution according to the memory requirement of the target processors. If the code size of the fused loop by the Max_LF technique can fit in the on-chip memory, then we use this loop fusion technique to fuse the loops since it achieves a smaller execution time. Otherwise, we will select the MLD_DF technique to fuse the loops to satisfy the memory constraint of the target processors. We use examples to show how the LD_LF_MEM technique works. The execution time of the fused loop of motivation example as shown in Fig. 1(a) by the Max_LF technique is about $5 * N * M$ when there are 8 functional units, while the execution time of the fused loop of the motivation example by our MLD_DF technique is $8 * N * M$. But the code sizes of the fused loops by these two techniques are 38 and 16 respectively. The code size of the fused loop by the Max_LF technique is larger than the memory constraint of the target processor which is assumed to be 32, so the LD_LF_MEM technique chooses the MLD_DF technique as the final solution. Thus, the execution time of the final loop of the motivation example by our LD_LF_MEM technique is $8 * N * M$, and the code size of the final loop is 16.

# 5 Experiments

This section presents the experimental results of our techniques. We simulate a DSP processor with 8 functional units. We compare the code sizes and the execution time of the original loop with those of the fused loops produced by the three techniques: the maximum loop fusion technique (Max_LF) proposed in [3], the technique of maximum loop distribution with direct fusion (MLD_DF), and the technique of loop distribution and loop fusion considering memory constraint (LD_LF_MEM). 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 summation of the execution time of each individual loop. The standard list scheduling algorithm is used in our experiments.

LL18 is the eighteenth kernel from the Livermore benchmark. LDG1, LDG2, and LDG3 refer to the examples presented in Figure 2, 8, and 17 in [5]. LDG4 and LDG5 refer to the examples shown in Figure 2(a) and Figure 6(a) in [3]. Each node of an LDG is a DSP benchmark. The DSP benchmarks include WDF (Wave Digital filter), IIR (Infinite Impulse Response filter), DPCM (Differential Pulse-Code Modulation device), and 2D (Two Dimensional filter). We set the code size constraints as follows: we assume that the code-size constraint is 128 instructions for the cases LDG1-LDG5 and LL18. We estimate the code size in memory by

| Program | Original | | | Max_LF | | | MLD_DF | | | LD_LF_MEM | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #Node | Size | Time | #Node | Size | Time | #Node | Size | Time | #Node | Size | Time | Time Impro. |
| LDG1 | 4 | 15 | 9*N*M | 1 | 24 | 5*N*M | 4 | 15 | 9*N*M | 1 | 24 | 5*N*M | 44.4% |
| LDG2 | 7 | 84 | 36*N*M | 1 | 170 | 10*N*M | 2 | 74 | 26*N*M | 2 | 74 | 26*N*M | 27.8% |
| LDG3 | 7 | 102 | 38*N*M | 1 | 244 | 14*N*M | 3 | 94 | 29*N*M | 3 | 94 | 29*N*M | 23.7% |
| LDG4 | 3 | 36 | 23*N*M | 1 | 72 | 12*N*M | 2 | 34 | 17*N*M | 1 | 72 | 12*N*M | 47.8% |
| LDG5 | 4 | 42 | 29*N*M | 1 | 92 | 13*N*M | 2 | 38 | 21*N*M | 1 | 92 | 13*N*M | 55.1% |
| LL18 | 3 | 18 | 9*N*M | 1 | 30 | 5*N*M | 3 | 18 | 9*N*M | 1 | 30 | 5*N*M | 44.4% |
| MOT1 | 4 | 20 | 16*N*M | 1 | 38 | 5*N*M | 2 | 16 | 8*N*M | 2 | 16 | 8*N*M | 50% |
| **Improvement** | - | | | - | | | - | | | - | | | 41.9% |

**Table 1.** The code sizes and the execution time of the original loop and the fused loop by various techniques

the number of instructions. MOT1 refers to the motivation example as shown in Fig. 1(a). For the motivation example, we assume the constraint is 32 instructions.

In Table 1, the columns "*Original*", "*Max_LF*", and "*MLD_DF*" contain three fields: the number of loop nodes (#Node), the code size (Size), and the execution time (Time). The Column "*LD_LF_MEM*" contains four fields: the number of loop nodes (#Node), the code size (Size), the execution time (Time), and the improvement of the execution time (Time Impro.). The Column "*Original*" shows the number of the loop units, the code size and the execution time of the original loop. The Column "*Max_LF*" shows the number of the loop units, the code size and the execution time of the fused loop by the Max_LF technique. The Column "*MLD_DF*" shows the number of the loop units, the code size and the execution time of the fused loop by the MLD_DF technique. The Column "*LD_LF_MEM*" shows the number of the loop units, the code size and the execution time and the improvement of the execution time of the fused loop by the LD_LF_MEM technique. 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.

The LD_LF_MEM technique selects the best execution time produced by the Max_LF technique and the MLD_DF technique within code-size constraint. For test cases such as LDG1, LDG4, LDG5, LL18, the LD_LF_MEM technique chooses MAX_LF technique as the final solution since the code sizes of the fused loop by the MAX_LF technique satisfy the memory constraint and a smaller execution time is achieved. That is when both the MAX_LF and the MLD_DF technique satisfy the memory constraint of the target processor, the LD_LF_MEM technique chooses the one which has a smaller execution time. But for some test cases such as LDG2, LDG3, MOT1, the code sizes of the fused loops produced by the Max_LF technique, though have smaller execution time, do not satisfy the memory constraint, so we can only select the results of MLD_DF as our final solution.

Although the maximum loop fusion technique (Max_LF) proposed in [3] can always achieve a shorter execution time than the technique of maximum loop distribution with direct fusion (MLD_DF), it increases the code size. In many cases, this technique cannot be applied because of the memory constraint. Compared to the Max_LF technique, the MLD_DF technique takes advantage of both loop distribution and loop fusion, so it reduces the original execution time and also avoids the code-size expansion. Our technique of loop distribution and fusion considering memory constraint (LD_LF_MEM) can always get the best result by selecting one of these two techniques as the final solution. The experimental results showed that the timing performance of the fused loop by our LD_LF_MEM technique can be improved 41.9% on average compared to the original loops, and the code size still satisfies the memory constraint of the target processors.

## 6 Conclusion

To improve the timing performance of the loops while satisfying the memory constraint commonly occurring in the embedded system, we developed the technique of combining loop distribution and loop fusion in this paper. The experimental results showed that the timing performance of the fused loops by our LD_LF_MEM technique can be improved significantly compared to the original loops, and the code size still satisfies the memory constraint of the target processors.

## References


[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. 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.

[3] 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.

[4] 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.

[5] 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.

[6] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Inc., 1996.

[7] 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.