Loop Distribution and Fusion with Timing and Code Size Optimization for Embedded DSPs ⋆

Meilin Liu[1], Qingfeng Zhuge[1], Zili Shao[2], Chun Xue[1], Meikang Qiu[1], and Edwin H.-M. Sha[1]

[1] University of Texas at Dallas
Richardson, Texas 75083, USA
{mxl024100, qfzhuge,cxx016000, mxq012100, edsha}@utdallas.edu
[2] Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
cszlshao@comp.polyu.edu.hk

Abstract. Loop distribution and loop fusion are two effective loop transformation techniques to optimize the execution of the programs in DSP applications. In this paper, we propose a new technique combining loop distribution with direct loop fusion, which will improve the timing performance without jeopardizing the code size. We first develop the loop distribution theorems that state the legality conditions of loop distribution for multi-level nested loops. We show that 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; otherwise, they should be put in the same loop after loop distribution. Then, we propose the technique of maximum loop distribution with direct loop fusion. The experimental results show that the execution time of the transformed loops by our technique is reduced 21.0% on average compared to the original loops and the code size of the transformed loops is reduced 7.0% on average compared to the original loops.
Keywords: Loop Fusion, Loop Distribution, Embedded DSP, Code Size, Scheduling

1 Introduction

Timing performance and code size are two major concerns for embedded systems with very limited on-chip memory resources [9]. Since loops are prevalent in multimedia processing, digital signal processing, etc., loop transformations such as loop fusion, loop distribution are needed to optimize the execution of the loops in embedded DSP applications [8, 1–3, 5].

Loop distribution separates independent statements inside a single loop (or loop nest) into multiple loops (or loop nests) [8, 1, 3, 2, 5]. Loop distribution can be used to break up a large loop that doesn't fit into the cache [2, 3, 8, 1].

⋆ This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCF-0309461, NSF IIS-0513669, Microsoft, USA.

It 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 [8, 1, 3, 5].

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, 3, 8, 1, 4, 5, 7]. 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 [4] can maximize the opportunities of loop fusion, but it cannot be applied in the cases where strict memory constraint applies.

A combination of loop distribution and loop fusion can find an optimization solution with both reduced execution time and restricted code size if the loop properties are fully understood. Therefore, it is very important to revisit and formalize the loop distribution and loop fusion theorems on graph models, and find an effective way to combine these two oppositely directed optimization techniques. 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.

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, 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 N-level loops based on the loop distribution theorems. We then propose the technique of maximum loop distribution with direct fusion (MLD_DF). The experimental results showed that the execution time of the transformed loops by our MLD_DF technique can be improved 21.0% 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 2. In Section 3, we propose the loop distribution theorems to guide loop distribution. We propose the technique of maximum loop distribution with direct fusion (MLD_DF) 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 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 $G = (V, E, \boldsymbol{d}, t)$ 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, $\boldsymbol{d}$ 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.

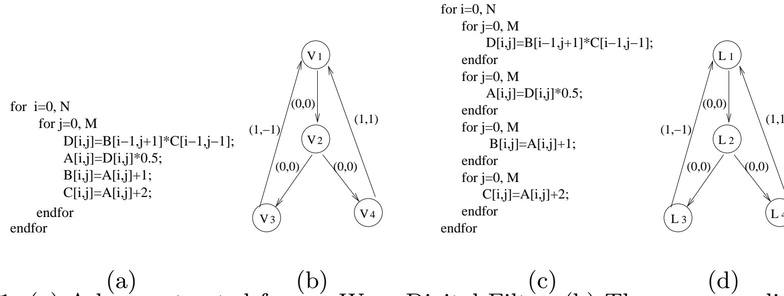


Fig. 1. (a) A loop extracted from a Wave Digital Filter. (b) The corresponding Data Flow Graph. (c) The distributed loop. (d) The loop dependence graph of the distributed loop.

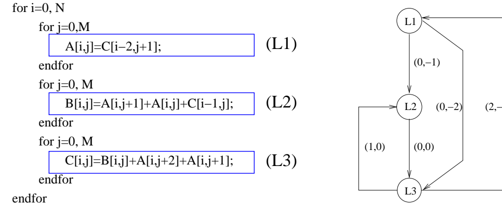
The execution of each node in V exactly once represents an iteration, i.e., the execution of one instance of the loop body. Iterations are identified by a vector $\boldsymbol{i}$, equivalent to a multi-dimensional index. Inter-iteration dependences are represented by vector-weighted edges. For any iteration $\boldsymbol{j}$, an edge e from u to v with delay vector $\boldsymbol{d}(e)$ means that the computation of node v at iteration $\boldsymbol{j}$ requires the data produced by node u at iteration $\boldsymbol{j} - \boldsymbol{d}(e)$. An edge with delay $(0, \cdots, 0)$ 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 $(0, \cdots, 0)$.

The program shown in Fig. 1(a) is extracted from a wave digital filter and its corresponding data flow graph is shown in Fig. 1(b).

2.2 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.



(a) (b)

Fig. 2. A loop and its corresponding LDG

The loop dependence graph of the loop in Fig. 2(a) is shown in Fig. 2(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 $\delta(e) < (0,0,\cdots,0)$. The fusion-preventing dependence edges for the LDG shown in Fig. 2(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. 2(b), node $L1$ represents the first inner loop of the loop shown in Fig. 2(a), which is labeled with 1 according to the execution sequence. Node $L3$ represents the third inner loop of the loop shown in Fig. 2(a), which is labeled with 3. According to the definition, in the loop dependence graph shown in Fig. 2(b), the backward edges include the edge from node $L3$ to $L1$, and the edge from $L3$ to $L2$.

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. In this paper, we only consider the loops with uniform data

dependences. A lot of DSP applications fit in this category. In the following, the general theorems for the N-level nested loops is proposed.

Theorem 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 $(\delta_1(e), \cdots, \delta_J(e)) \geq (0, \cdots, 1)$.*

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 $(\delta_1(e), \cdots, \delta_J(e))$ represent the dependence distance of the shared outer loop. If $(\delta_1(e), \cdots, \delta_J(e))$ 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. Obviously, the code shown in Fig.3(b) computes differently from the code shown in Fig.3(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.3(a). If we directly distribute the loop shown in Fig.3(a), then this true data dependence becomes an anti-data dependence in the distributed loop as shown in Fig. 3(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.

```
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)

Fig. 3. An illustration example

Fig. 1(c) shows the distributed loop of the original program shown in Fig. 1(a). Because the backward edges $e_1 : L3 \rightarrow L1$ and $e_2 : L4 \rightarrow L1$ in the LDG shown in Fig. 1(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 2 is satisfied. A loop is completely distributed when each loop unit after distribution only has one array assignment statement.

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

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

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

Due to the space limit, we do not give a formal proof here. It directly follows from Theorem 1, since all the data dependences in the LDG of the distributed loop come from the data dependences of the original loop. Theorem 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. 4(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 2, we can distribute the loop at the second loop level. The distributed loop is shown in Fig. 4(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. 4(c).

```
for  i=0, N
     for  j=0, M
          for  k=0, S
               A[i,j,k]=C[i−1,j,k]+7;
               B[i,j,k]=A[i,j,k]*3;
               C[i,j,k]=B[i,j,k]+5;
          endfor
     endfor
endfor
```

(a)

```
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
```

(b)

```
for  i=0, N
     for  j=0, M
          for  k=0, S
               A[i,j,k]=C[i−1,j,k]+7;
          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
endfor
```

(c)

Fig. 4. (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 3 identifies the dependence cycle in the data flow graph that prevents the statements involved in the dependence cycle from distribution.

Theorem 3. *Given a N-level nested loop and its corresponding data flow graph $G = (V, E, \boldsymbol{d}, t)$, 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 $\boldsymbol{d}(c)$ satisfies that $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_J(c)) = (0, \cdots, 0)$, 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 $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_J(c)) = (0, \cdots, 0)$, then there will be a dependence cycle c with $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_J(c)) = (0, \cdots, 0)$ 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 $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_J(c)) \geq (0, \cdots, 0)$. So the backward edge e' in the cycle c must have $(\boldsymbol{d}_1(e'), \cdots, \boldsymbol{d}_J(e')) = (0, \cdots, 0)$, which is contradictory to Theorem 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 $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_J(c)) = (0, \cdots, 0)$ must be put into one loop after the loop is distributed on the $(J+1)$-th loop level.

4 Loop Distribution and Loop Fusion

In this section, we first use an example to show how to conduct maximum loop distribution for N-level nested loops based on the loop distribution theorems proposed in Section 3. Then, we propose the technique of maximum loop distribution with direct fusion (MLD_DF).

4.1 Maximum Loop Distribution

(a)

```
for  i=0, N
    for  j=0, M
        for  k=0, S
            A[i,j,k]=D[i,j,k−2];
            B[i,j,k]=A[i−1,j,k];
            C[i,j,k]=B[i,j,k]+D[i,j,k−1];
            D[i,j,k]=C[i,j,k]+7;
        endfor
    endfor
endfor
```

(b)

```
for  i=0, N
    for  j=0, M
        for  k=0, S
            B[i,j,k]=A[i−1,j,k];
        endfor
        for  k=0, S
            C[i,j,k]=B[i,j,k]+D[i,j,k−1];
            D[i,j,k]=C[i,j,k]+7;
        endfor
        for  k=0, S
            A[i,j,k]=D[i,j,k−2];
        endfor
    endfor
endfor
```

Fig. 5. (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 $(\boldsymbol{d}_1(e), \cdots, \boldsymbol{d}_{N-1}(e)) \geq (0, \cdots, 1)$ from the LDG of the given loop. Thus, if the summation $\boldsymbol{d}(c)$ of the edge weights of a cycle c in the LDG satisfies that $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_{N-1}(e)) \geq (0, \cdots, 1)$, then cycle c is broken. Then, we merge each cycle c with $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_{N-1}(c)) = (0, \cdots, 0)$ into one node. This is used to guarantee that the statements involved in the dependency cycle whose summation of the edge weights satisfies that $(\boldsymbol{d}_1(c), \cdots, \boldsymbol{d}_{N-1}(c)) = (0, \cdots, 0)$ 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., $(\delta_1(e), \cdots, \delta_{N-1}(e)) \geq (0, \cdots, 1)$.

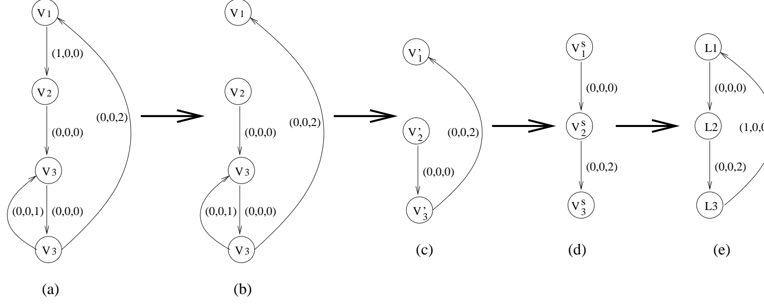


Fig. 6. The graph transformation process of the maximum loop distribution algorithm

Every node in the transformed graph corresponds to a loop unit in the distributed loop.

We use a 3-level nested loop as shown in Fig. 5(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. 6(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 $\boldsymbol{d}(c_1) = (1, 0, 2)$, so cycle c_1 will be broken after the edges e with $(\boldsymbol{d}_1(e), \boldsymbol{d}_2(e)) \geq (0, 1)$ 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 $\boldsymbol{d}(c_2) = (0, 0, 1)$, so cycle c_2 will be merged into one node according to the basic idea of maximum loop distribution since $(\boldsymbol{d}_1(c), \boldsymbol{d}_2(c)) = (0, 0)$. 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. 6(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. 5(b). Then we can get the LDG of the distributed loop as shown in Fig. 6(e). Fig. 6 shows the graph transformation process.

4.2 Maximum Loop Distribution with Direct Loop Fusion

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 [3, 5]. 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.

The technique of maximum loop distribution with direct fusion (MLD_DF) is to perform maximum loop distribution followed with direct loop fusion, such that the timing performance of the loops is increased 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 [3]. Then, direct loop fusion is applied.

5 Experiments

This section presents the experimental results of our technique. We simulate a DSP processor with eight functional units. We compare the code sizes and the execution time of the original loops with those of the fused loops produced by the technique of maximum loop distribution with direct fusion (MLD_DF). 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.

LDG1, LDG2, and LDG3 refer to the examples presented in Figure 2, 8, and 17 in [6]. LDG4 and LDG5 refer to the examples shown in Figure 2(a) and Figure 6(a) in [4]. 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 estimate the code size in memory by the number of instructions.

Table 1. The code size and the execution time of the original loops and the transformed loops by the MLD_DF technique

	Original			MLD_DF				
Program	**#Node**	**Size**	**Time**	**#Node**	**Size**	**Time**	**Time Impro.**	**Size Red.**
LDG1	4	15	9*N*M	4	15	9*N*M	0%	0%
LDG2	7	84	36*N*M	2	74	26*N*M	27.8%	11.9%
LDG3	7	102	38*N*M	3	94	29*N*M	23.7%	7.8%
LDG4	3	36	23*N*M	2	34	17*N*M	26.1%	5.6%
LDG5	4	42	29*N*M	2	38	21*N*M	27.6%	9.5%
Improvement	-			-			21.0%	7.0%

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

Although the maximum loop fusion technique (Max_LF) proposed in [4] 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. The experimental results showed that the timing performance of the fused loop by our MLD_DF technique can be improved 21.0% on average compared to the original loops, and the code size is reduced 7.0% on average compared to the original loops.

6 Conclusion

The maximum loop fusion technique (Max_LF) proposed in [4] can maximize the opportunities of loop fusion, but it also increases code size. In this paper, we developed the technique of combining loop distribution and loop fusion to achieve a shorter execution time with a reduction of the code size. The experimental results showed that the timing performance of the transformed loops by our MLD_DF technique can be improved significantly compared to the original loops, and the code size of the transformed loops is also reduced compared to the original loops.

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. 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.