Optimizing Timing and Code Size Using Maximum Direct Loop Fusion *

Meilin Liu[1] Chun Xue[2] Meikang Qiu[2] Edwin H.-M. Sha[2]
[1]Department of Computer Science [2]Department of Computer Science
Wright State University University of Texas at Dallas
Dayton, Ohio 45435, USA Richardson, Texas 75083, USA

Abstract

In this paper, we develop the technique that combines loop distribution with maximum direct loop fusion (LD_MDF), which performs *maximum loop distribution*, followed by *maximum direct loop fusion* to optimize timing and code size simultaneously. We illustrate using examples that dependence cycle is not always a restriction for loop distribution for multi-level loops. We map the maximum direct loop fusion problem to the graph partitioning problem. A polynomial graph partitioning algorithm is developed to compute the fusion partitions. We prove that our maximum direct loop fusion algorithm produces the fewest number of resultant loop nests without violating dependence constraints. We also show that the resultant code size of the fused loops by the technique of loop distribution with maximum direct loop fusion is always smaller than the code size of the original loops.
Keywords: Loop Fusion, Loop Distribution, Code Size, Embedded DSP

1 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]. 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 [7, 1].

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. Maximal 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) [7, 1, 2, 5]. Loop distribution exposes parallelism by separating the statements that could be parallelized, and the statements that must be executed sequentially. Loop distribution can also be used to break up a large loop that doesn't fit into the cache [2, 7, 1].

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

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 [7, 1, 5, 2]. In [2], 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. 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 [8].

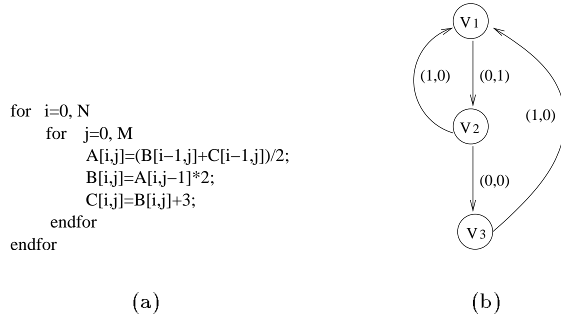
Timing and code size are the two most important performance metrics for embedded systems with very limited on-chip memory resources [8]. Combining loop distribution and loop fusion can lead to an optimization solution with both reduced execution time and restricted code size. After loop distribution 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 the technique of loop distribution with maximum direct loop fusion (LD_MDF). The authors of [2] 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 illustrate using examples that dependence cycle is not always a restriction for loop distribution for multi-level loops. We map the direct loop fusion problem to the graph partitioning problem and we develop a polynomial graph partitioning algorithm to get the fusion partitions. We prove that our maximum direct loop fusion technique produces the fewest number of resultant loop nests without violating dependence constraints. We also show 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 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 illustrate that dependence cycle is not always a restriction for loop distribution for multi-level loops using examples. We propose the maximum direct loop fusion technique in Section 4. The basic idea of the technique of loop distribution with maximum direct loop fusion (LD_MDF) is illustrated in Section 5. Section 6 presents the experimental results. Section 7 concludes the paper.

2 Basic Concepts

In this section, we provide an overview of the basic concepts and principles related to our technique, including multi-dimensional data flow graph ($MDFG$) and loop dependence graph (LDG).

2.1 Data Flow Graph



(a) (b)

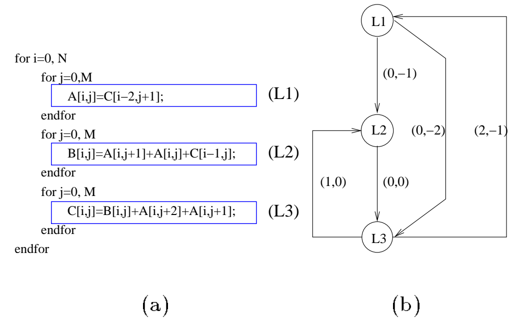
Figure 1: A two-level loop and its corresponding DFG.

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. A two-level loop is shown in Figure 1(a) and its corresponding data flow graph is shown in Figure 1(b).

2.2 Loop Dependence Graph

Loop dependence graph is a higher-level graph model compared to the data flow graph. 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.

The loop dependence graph of the loop in Figure 2(a) is shown in Figure 2(b). In a loop dependence graph, a fusion-preventing dependence is represented by an edge e with edge weight $\delta(e) < (0,0)$. All the comparisons between two loop dependence vectors are based on the lexicographic order in this paper. A two-dimensional vector $\vec{v}$ is smaller than a two-dimensional vector $\vec{u}$ according to the lexicographic order if either $v_1 < u_1$ or $v_1 = u_1$ and $v_2 < u_2$. The fusion-preventing dependence edges are $e_1 : L1 \rightarrow L2$ and $e_2 : L1 \rightarrow L3$ in the LDG shown in Figure 2(b).

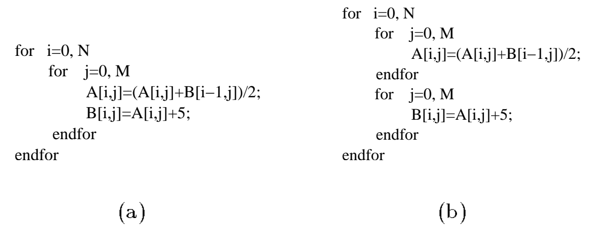


(a) (b)

Figure 2: (a) A two-level loop with three inner loops. (b) The corresponding loop dependence graph.

3 Loop Distribution

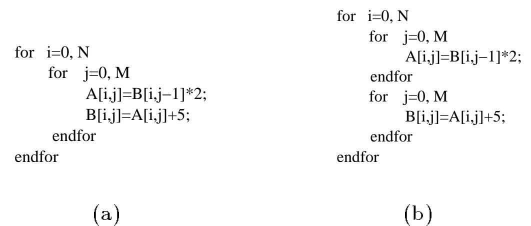
Loop distribution is an important part of our technique of loop distribution with maximum direct loop fusion (LD_MDF). But loop distribution is not always applicable. In the process of loop distribution, we must maintain all the data dependences to ensure that we won't change the semantics of the original program. We show that dependence cycle is not always a restriction for loop distribution for multi-level loops by illustration examples.



(a) (b)

Figure 3: An example loop that is legal to be distributed.

The program shown in Figure 3(a) has a dependence cycle in its corresponding data flow graph, but the statements involved in the dependence cycle can be distributed. The distributed loop shown in Figure 3(b) has the same semantics as the original program shown in Figure 3(a) since the true data dependence in the loop in Figure 3(a) is maintained to be a true data dependence in the distributed loop in Figure 3(b).



(a) (b)

Figure 4: An example loop that is illegal to be distributed.

The program shown in Figure 4(a) also has a dependence cycle in its corresponding data flow graph. If we distribute the statements involved in the dependence cycle, then the semantics of the program is changed. We can see that the code shown in Figure 4(b) computes differently from the code shown in Figure 4(a). $A[i,j]$ is dependent on

$B[i, j-1]$, which is a true data dependence in the original program as shown in Figure 4(a). If we directly distribute the loop shown in Figure 4(a), then this true data dependence becomes an anti-data dependence in the distributed loop as shown in Figure 4(b). In this case, the statements involved in the dependence cycle in the original loop cannot be distributed.

In summary, 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. The loop distribution theorems and the maximum loop distribution algorithm have been presented in [4].

4 Direct Loop Fusion Technique

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. We develop a polynomial graph partitioning algorithm 4.1 and we prove that the total number of the partitions got by our algorithm is minimal, i.e., the number of the fused loops is minimal.

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. A fusion partition is legal if and only if the following two conditions are satisfied:

1. Precedence dependence constraint: the relative order of two nodes connected by a precedence dependence edge cannot change.
2. Fusion-preventing dependence constraint: two nodes connected by a fusion-preventing dependence edge must be put into different partitions.

4.1 Graph Partitioning Algorithm

We formulate maximal fusion problem as a scheduling problem, i.e., we are looking for a mapping P from the loop nodes V to the positive integers N so that the following two conditions are satisfied:

1. Precedence dependence constraint: $e = u \rightarrow v$, $u, v \in V$, e is a precedence dependence edge, $P(v) \geq P(u)$ (1)
2. Fusion-preventing dependence constraint: $e = u \rightarrow v$, $u, v \in V$, e is a fusion-preventing dependence edge, $P(v) \geq P(u) + 1$ (2)

We compute the partition number for each loop node in the loop dependence graph based on the two conditions as stated above. More specifically, we compute the partition number of each node by using the shortest path algorithm after constructing the constraint graph based on the two conditions of the legal loop partition. The basic idea of the partition algorithm by using the shortest path algorithm is as follows:

Given the input loop dependence graph, we distinguish two kinds of edges, the fusion-preventing dependence edges and the precedence dependence edges. We construct a constraint graph based on the given loop dependence graph and the two conditions coming from the precedence dependence edge and the fusion-preventing dependence edge. Then we apply the shortest path algorithm on the constraint graph to compute the partition number for each loop node.

Algorithm 4.1 Graph Partitioning Algorithm

Require: A LDG $G = (V, E, \delta, o)$
Ensure: Partitions of loop nodes of the LDG
  Remove all the edges e with $\delta_1(e) \geq 1$ from E, and get a graph $G' = (V, E', \delta)$, where $E' \leftarrow E - \{e | \delta_1(e) \geq 1\}$
  Add one corresponding node P_i in the constraint graph G_c for each node v_i in graph G'
  for all edges $e' \in E'$ in graph G' **do**
    if e' is a fusion-preventing edge in the original loop dependence graph **then**
      We get a constraint $P_i - P_j \leq -1$. So we add one edge from P_j to P_i in the constraint graph G_c, which is weighted by -1.
    else
      The edge is a precedence edge, and we get a constraint $P_i - P_j \leq 0$. So we add one edge from P_j to P_i in the constraint graph G_c, which is weighted by 0.
    end if
  end for
  Add the source node P_0 in the constraint graph G_c, and also the edges from P_0 to all the other nodes P_i in the constraint graph G_c. All these added edges from the source node P_0 to all the other nodes P_i are weighted with 0
  /* Compute the shortest distance from the source node to all the other nodes in the constraint graph by using the shortest path algorithm */
  Call the Bellman-Ford algorithm to compute $D(P_0, P_i)$, the shortest distance from P_0 to each node $P_i \in V_c$
  /* Compute the minimum shortest distance */
  for all nodes $v \in V_c$ **do**
    $D_{min} \leftarrow min_{P_i}(D(P_0, P_i))$
  end for
  for all nodes $v \in V_c$ **do**
    $P(P_i) \leftarrow D(P_0, P_i) - D_{min} + 1$
  end for
  for all nodes $v \in V$ in the original LDG **do**
    Put node v_i in partition $P(P_i)$
  end for
  return Partitions

In the algorithm, we first remove all the edges e with $\delta_1(e) \geq 1$ from the LDG and we obtain a directed acyclic graph (DAG) G'. Then we construct the corresponding constraint graph G_c based on graph G'. For each node

v_i in graph G', we have a corresponding node P_i in the constructed constraint graph G_c. The edges in the constraint graph are constructed based on the precedence dependence constraint and the fusion-preventing dependence constraint. For each precedence edge $e : v_i \to v_j$ in the original loop dependence, the constraint is $P_i - P_j \leq 0$, so an edge from P_j to P_i is added in the constraint graph, and the edge is weighted by 0. For each fusion-preventing dependence edge $e : v_i \to v_j$ in the original loop dependence, the constraint is $P_i - P_j \leq -1$, since v_j must be put into a partition different from the partition in which v_i is put into, and the partition number of v_j must be larger than the partition number of v_i. So an edge from P_j to P_i is added in the constraint graph, and the edge is weighted by -1. Then, we add a source node P_0 and also the edges from P_0 to all the other nodes P_i. The added edges from the source node P_0 to all the other nodes P_i are all weighted by 0. The construction of the constraint graph G_c does not create cycles. The Bellman-Ford algorithm can be applied to compute the shortest distance $D(P_0, P_i)$ from the source node P_0 to each other node P_i in the constructed constraint graph.

After we get the shortest distance from the source node P_0 to each other node P_i, we can compute the minimum shortest distance min_D of all the nodes in the constraint graph, i.e., $min_D = \min_{P_i}(D(P_0, P_i))$. Then for each node P_i, we compute the partition number $P(P_i)$ for P_i by $P(P_i) = D(P_0, P_i) - min_D + 1$. In the algorithm, we get the partition number for every node, and the nodes with the same partition number are put into the same fusion partition.

The distributed loop of the example loop in Figure 6(a) is shown in Figure 6(b), and its corresponding LDG is shown in Figure 5(a). The constructed constraint graph by our graph partitioning algorithm 4.1 is shown in Figure 5(b). After we run algorithm 4.1, 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 5(c). 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 5(d).

Theorem 4.1 *Given a loop and its corresponding LDG, the partition computed by algorithm 4.1 is a legal partition. And the total number of the partitions computed by algorithm 4.1 is minimal.*

Proof 4.1 *By contradiction:*

According to the properties of a legal loop, every legal loop dependence graph $G = (V, E, \delta, o)$ must satisfy that $\delta_1(e) \geq 0, \forall e \in E$, and $\delta_1(c) \geq 1$, for each cycle c in G. Therefore, in algorithm 4.1, after we remove all the edges e with $\delta_1(e) \geq 1$, the obtained graph G' is a DAG. Then, we construct the constraint graph G_c based on the graph G'. For each node L_i in graph G', there is a corresponding node P_i in the corresponding constraint graph. If the original

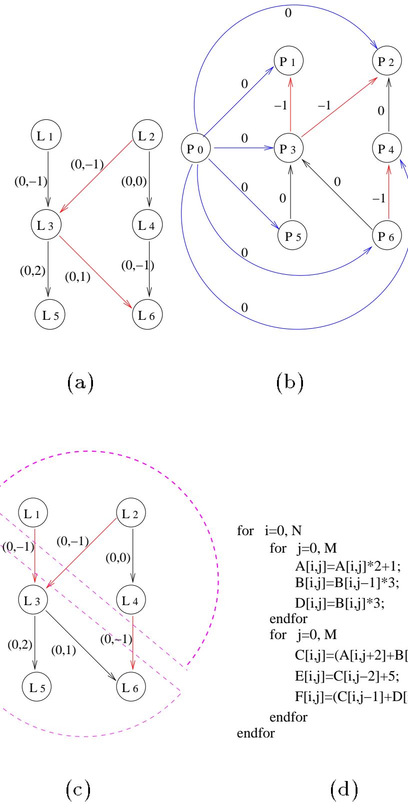


Figure 5: (a) The LDG of the distributed loop in Figure 6(b). (b) The constructed constraint graph for partitioning by algorithm 4.1. (c) The graph partitioning result. (d)The fused loop by the LD_MDF technique.

edge $e : L_i \to L_j$ in G' is a precedence edge, then in the constraint graph, there is one edge $e_c : P_j \to P_i$, which is weighted by 0. If the original edge $e : L_i \to L_j$ in G' is a fusion-preventing dependence edge, then in the constraint graph, there is one edge $e_c : P_j \to P_i$, which is weighted by -1. And we also add a source node V_0 and also the edges from node V_0 to all the other nodes in the constraint graph, and all the added edges from the source node V_0 to other nodes are weighted by 0 too.

The construction of the constraint graph G_c does not create cycles. Thus, G_c is a DAG. Bellman-Ford algorithm is applied to compute the single-source shortest distance $D(P_0, P_i)$ from P_0 to all the other nodes P_i. Note that there must at least exist a node P so that $D(P_0, P) = 0$ since the constraint graph is a DAG. Also, since in the constraint graph, all the edges are weighted either by 0 or by -1, then computed $D(P_0, P_i)$ values must be continuous integers.

In the algorithm 4.1, we compute $D_{min} \leftarrow min_{P_i}(D(P_0, P_i))$, which is the minimum value of the longest shortest path. D_{min} is a negative value according to the construction of the constraint graph. And the computed values for the nodes in the constraint graph include $0, \cdots, D_{min}$, i.e., so there are totally $-D_{min} + 1$ distinct computed values for all the loop nodes. Define $K_{max} = -D_{min}$, then there are totally $K_{max} + 1$ parti-

tions after running the algorithm.

We here prove that the number of the partitions computed by algorithm 4.1 is minimal. Suppose that the number of the partition is smaller than K_{max}. Consider node P_m whose $D(P_0, P_m) = -D_{min}$. According to the construction of the constraint graph, and the definition of the shortest path algorithm, there must exist a path with K_{max} number of edges with weight -1 from the source node P_0 to P_m. The edge with weight -1 is corresponding to the fusion-preventing dependence edges in the original loop dependence graph. Along this path, it is obvious that all the nodes in the original data flow graph associated with these K_{max} fusion-preventing dependence edges must be put into $(K_{max}+1)$ different partitions. This path from P_0 to P_m by the shortest path algorithm has given us the lower bound of the number of partitions between v_0 and v_m. So it is impossible to have the total number of the partitions less than $(K_{max}+1)$ by simply considering this path from v_0 to v_m.

5 Loop Distribution with Maximum 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 using the maximum loop distribution algorithm presented [4].
2. Construct the corresponding loop dependence graph G_L of the distributed loop.
3. Apply the maximum direct loop fusion algorithm 4.1 to compute the fusion partitions.
4. Get the fused loop L_f 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 using the maximum loop distribution algorithm proposed in [4]. 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. In other words, 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, all the loop nodes inside one fusion partition can be directly fused [2]. After we get the fusion partitions, direct loop fusion is applied on each fusion partition. We prove that the total number of the fusion partitions got by our graph partitioning algorithm is minimum, i.e., our maximum direct loop fusion algorithm produces the fewest number of fused loops. According to the implementation procedure of the LD_MDF technique, it is obviously that 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 5.1.

Corollary 5.1 *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  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
```

(a)

```
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
```

(b)

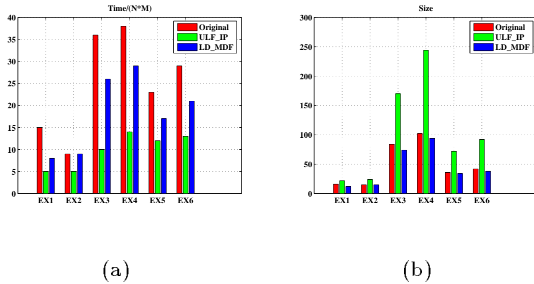
Figure 6: (a) The original 2-level loop with six inner loops. (b)The distributed loop.

For example, the code shown in Figure 6(a) contains four sequential loops enclosed in one shared outermost loop. To apply our LD_MDF technique, we first maximumly distribute the loop using the maximum loop distribution algorithm proposed in [4]. The maximumly distributed loop for the original program is shown in Figure 6(b). After loop distribution, there are totally six loops. Then, we partition the six loop nodes in the loop dependence graph as shown in Figure 5(a) into two fusion partitions as shown in Figure 5(c). 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 5(d), 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.

6 Experiments

In our experiments, we performed the general legalizing loop fusion technique (ULF_IP) presented in [3] 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.



(a) (b)

Figure 7: (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 6(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 [3].

In Figure 7(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 7(a). Figure 7(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 [3] 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.

7 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 developed a polynomial graph partitioning algorithm to compute the fusion partitions. We prove that our maximum direct loop fusion technique produces the fewest number of resultant loop nests without violating dependence constraints. We also show 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.

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] M. Liu, Q. Zhuge, Z. Shao, C. Xue, M. Qiu, and E. H.-M. Sha. Loop distribution and fusion with timing and code size optimization for embedded DSP. In *Proc. The 2005 International Conference on Embedded And Ubiquitous Computing (EUC)*, pages 121–130, Lecture Note in Computer Science, Springer, Nagasaki, Japan, Dec. 2005.

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

[8] 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.