IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 7, July 2017

Loop Fusion Methods: A Critical Review

Mahsa Ziraksima[1], Shahriar Lotfi[2], Habib Izadkhah[3]

Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran[1,2,3]

Abstract: Contemporary applications are much more complex than before and they need more time and resources while being executed. Investigators provide various strategies to improve the process of compiling, trying to improve execution speed. Hence, distinct transformations are proposed that are frequently used in modern compilers. One of them is loop fusion in which multiple loops merge together and form an extended one. Many studies have been done on this transformation and its impact on systems performance, reflecting the importance of this issue, each of them having its own perspective in enhancing the efficiency of loop fusion. This paper demonstrates a review of prominent researches in loop fusion and assesses the advantages and disadvantages of each.

Keywords: Optimizing Compilers; Loop transformations; Loop Fusion; Data Dependence.

I. INTRODUCTION

Loop fusion is one of the various forms of loop transformation that has been proposed to compensate the limitations of memory performance in contrast to processors considerable progress. Apposed to expectations, its great capabilities were observed, some of which applies significant improvements in program execution. Its impacts include reducing number of memory access and transferred data as a result of it the memory latency, declining loop examination overhead and required synchronizations, decreasing live time of data and thus minimizing demanded memory size [1, 2, 3, 4, 5, 6].
We assume that readers are familiar with different kind of data dependences one of which occurring among program statements, including true, anti, output and input [7, 8]; and the other one between different loop iterations, namely loop-independent and loop-carried [9]. In the following of this section some important terms and vocabularies are defined briefly for a better comprehension.
Loops are the most important part of a program that iterates a set of instructions for specific number. They can be grouped based on two factors, one of which is loop headers causing more than two types of loops; and the other one is their execution mode as being sequential or parallel. Moreover, some loop nests are perfect, containing all statements in the innermost loop, otherwise it is called imperfect loop nest.
Fusion means merging some loops and constructing single one with greater loop body. Distribution on the contrary, divides a loop to number of smaller ones. All the data dependences among loops form a directed acyclic graph which is used to model this problem. In this representation, each node and edge identifies a loop or statement and the dependence among them, respectively. If a loop independent dependence changes to loop carried due to fusion, it is called fusion preventing edge. Merging these loops, not considering the fusion preventing dependence between, changes the programs output when executing in parallel. Note that in all the following figures, distinct loop types are represented by specific shapes in the graph and fusion preventing edges are identified by a dash on the dependence edge.
In the fusion problem a set of loops are merged to minimize the number of nodes. Two basic conditions which must be met throughout fusion are preserving the relative order of nodes and avoiding cycle creation in the resulted graph. Regarding to the outlined terms, choosing a pair of loops for fusion is an important decision, determining the result of it. For instant, in figure 1 merging nodes 1 and 3 prevents fusing 2 and 4; and conversely. In the below graph parallel loops are shown by two circle and sequential ones by a circle.

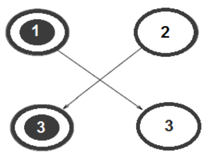


Fig. 1. Importance of loop order in fusion [3]

After discussing the basic concepts of loop fusion problem, in section 2 we will investigate complexity of it in different cases. The third section is going to present a survey of renowned fusion method and compares their properties. Finally, section 4 concludes and recommends some research areas that require further attention in order to achieve more preferable fusion results.

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 7, July 2017

II. BACKGROUND

The fusion can be interpreted as other well-known problems. For example, if we designate the dependence graph as a chain set in which each chain is a sequence of nodes, determining relative order of nodes, fusion becomes shortest common supersequence problem (SCS). Also, fusion can be modelled as the scheduling problem that has a solution with acyclic graph and components of it are minimally connected clusters whose nodes are connected maximally. When there is no fusion preventing edge, it turns into the scheduling problem on single machine that tries to run various kinds of tasks with minimum mode switching. Furthermore, the fusion changes to the traveling salesman problem which has to pass different cities in a specific order by least possible amount of moving among cities [3].
On the other hand, loop fusion problem has different complexities in distinct circumstances. For instance, fusing same typed loops, fusing two types of loops in absence of fusion preventing edges or ordered typed fusion are some polynomially solvable cases. But to find the greatest amount of fusion in the presence of fusion preventing edges with two or more types of loops, or when there is no fusion preventing edge among three or more types of loops, a heuristic algorithm is required owing to the NP-complete complexity [3]. Even maximum loop fusion problem for optimising data reuse has been proved to be NP-hard [10].

III.LOOP FUSION METHODS

In recent decades, many studies have been done in the area of loop fusion, exposing the significance of it in order to provide more efficient compile performance. We will discuss various fusion strategies in this section and compare their distinct attributes. Note that in all the methods V and E identify number of vertex and edges in the related directed acyclic dependence graph, respectively; in addition, weights specify the amount of data reuse.
Kennedy and McKinley have mentioned two objectives separately in [10]. The first metric was maximizing parallelism. They proposed a greedy partitioning algorithm for fusing loops, typed parallel and sequential, which starts with fusing a subgraph containing parallel loops and related dependences and fusion preventing edges; and then similarly the sequential component. Note that after each partitioning phase, it combines the resulted graph with the original one. This method decreases the amount of required synchronization, with taking into account parallelism preservation by not fusing loops with various types or equality typed with a path containing different types. An illustration of this method is provided in the figure 2 in which parallel and sequential loops are identified by S and P, correspondingly. The second measure was optimizing data locality as a result of minimized data dependence between merged loops. Based on NP-hardness of loop fusion for reuse which is proved in this paper, they have provided two heuristic algorithms for it. One of them is a greedy method with complexity of $O(V \times E)$ in terms of time and space with obscured precision in comparison to an optimal solution. It moves loops between partitions considering all required legalities as absence of fusion preventing edge or having same types. Another one is based on the maximum-flow/minimum-cut algorithm which has a tight worse-case bound on its results' precision. Actually, it makes K (number of fusion preventing edges) minimum cut that maximize flow in the resulted graphs in $O(K \times V \times E \times \log(V^2/E))$. All mentioned algorithms have modelled loop fusion problem in the form of a directed acyclic graph in which each node identifies a loop and each edge between represents their dependence. Later, in [11] they proposed a solution for ordered typed fusion problem in presence of fusion preventing edge. There are more than two types of loops in this method and an absolute order of types is used when there is a conflict between them; in this case it is not important how this order affects the result. They intend to merge and minimize the number of loops, representing fusion problem by a directed acyclic graph as their previous work. Proposed algorithm carries out a greedy loop fusion strategy which is applied for each selected type separately. If considering T types, its complexity is $O((V+E)T)$. The initial number of nodes is observed by breath-first sort; then, they are renumbered and fused based on two variables, determining the first node of selected type from which there is no fusion preventing edge in the path between (maxBadPrev) and the highest number of a direct predecessor of mentioned node which it can be merged (maxPred). A selected node cannot fuse with any node that is numbered lower than the predecessor. Briefly, each node fuses with its successive nodes if they have the same type, there is no fusion preventing edge and no other typed node in the paths between them. Requirement of an order for loops is this approach's weak point. This algorithm is applied on figure 3 and the right side result is observed.
Singhai and McKinley in [4] solved a restricted case of loop fusion in which data dependences form a tree. They have proposed an optimal solution based on dynamic programing, regarding to data locality, parallelism and register pressure. This method runs linearly in terms of number of loops and quadratic time considering number of available registers. First maximal distribution is performed on the loops, resulting in the maximum possible number of them. Then the algorithm finds the maximal weight spanning tree, related to dependence graph, and after that merges them, improving data reuse according to the size of register. They reasoned that if data size of formed loops is larger than register, amount of data transformation increase and in this situation the caused overhead will overweight the benefit of fusion. Furthermore, if total weights of two determined loops are greater than register, the fusion preventing edge between them is eliminated; because undoubtedly it is replaced in the memory. They partition the resulted tree to minimize total weight of edges, the source and sink nodes of which are in two separate node after fusion, by using a

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 7, July 2017

method based on Lukes' algorithm in [12]. They combine optimal subtrees from bottom to up, starting from the leaf nodes, with the aim of reaching to larger partitions. It finds all possible subtrees and chooses the optimal one among them; when facing with a fusion preventing edge, it backtracks and selects a suboptimal partition that does not consist any fusion preventing edge to avoid parallelism loss. Considering register pressure is a big step forward, but since some edges are omitted while creating maximal spanning tree which could be a part of the optimal solution, we can conclude the fact that its output may not be the optimal; which is their main drawback.

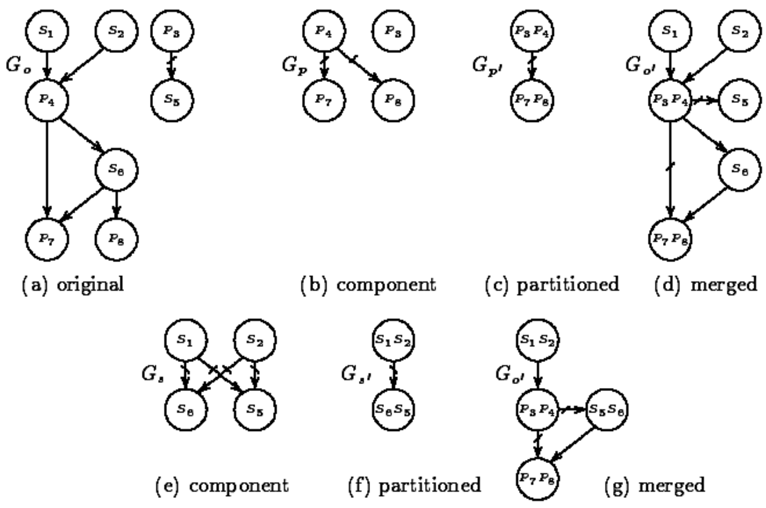


Fig. 2. Maximizing parallelism [10]

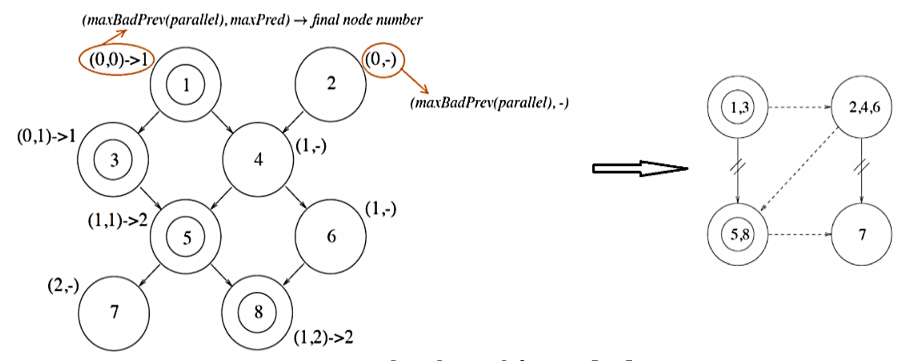
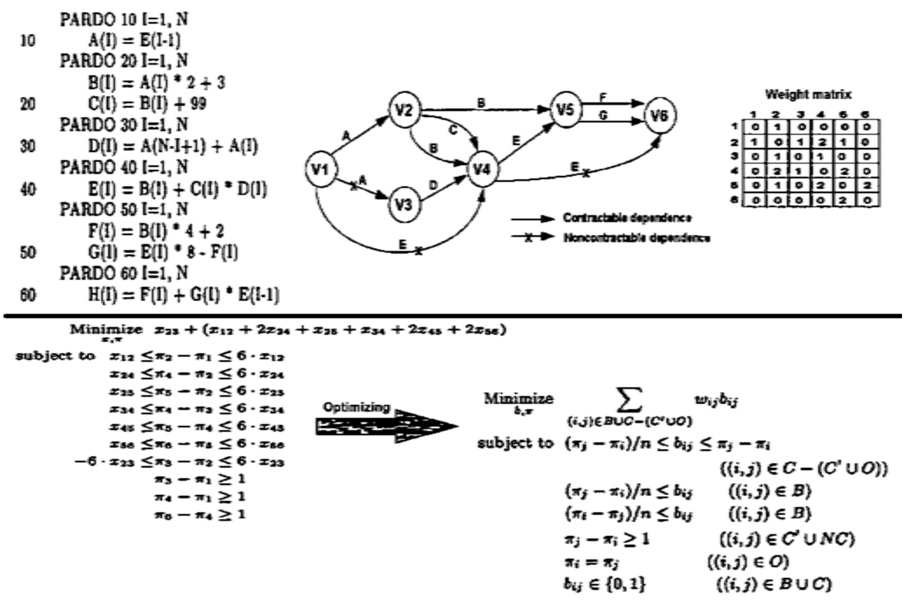

(maxBadPrev(parallel), maxPred) → final node number
(0,0)->1
(0,-)
(maxBadPrev(parallel), -)
(0,1)->1
(1,-)
(1,1)->2
(1,-)
(2,-)
(1,2)->2
1,3
2,4,6
5,8
7

Fig. 3. Ordered typed fusion [11]



```
      PARDO 10 I=1, N
10        A(I) = E(I-1)
      PARDO 20 I=1, N
          B(I) = A(I) * 2 + 3
20        C(I) = B(I) + 99
      PARDO 30 I=1, N
30        D(I) = A(N-I+1) + A(I)
      PARDO 40 I=1, N
40        E(I) = B(I) + C(I) * D(I)
      PARDO 50 I=1, N
          F(I) = B(I) * 4 + 2
50        G(I) = E(I) * 8 - F(I)
      PARDO 60 I=1, N
60        H(I) = F(I) + G(I) * E(I-1)
```

Weight matrix

	1	2	3	4	6	6
1	0	1	0	0	0	0
2	1	0	1	2	1	0
3	0	1	0	1	0	0
4	0	2	1	0	2	0
5	0	1	0	2	0	2
6	0	0	0	0	2	0

$$\underset{x,\pi}{\text{Minimize}} \quad x_{23} + (x_{12} + 2x_{24} + x_{25} + x_{34} + 2x_{45} + 2x_{56})$$

$$\begin{aligned}\text{subject to} \quad x_{12} &\le \pi_2 - \pi_1 \le 6 \cdot x_{12}\\ x_{24} &\le \pi_4 - \pi_2 \le 6 \cdot x_{24}\\ x_{25} &\le \pi_5 - \pi_2 \le 6 \cdot x_{25}\\ x_{34} &\le \pi_4 - \pi_3 \le 6 \cdot x_{34}\\ x_{45} &\le \pi_5 - \pi_4 \le 6 \cdot x_{45}\\ x_{56} &\le \pi_6 - \pi_5 \le 6 \cdot x_{56}\\ -6 \cdot x_{23} &\le \pi_3 - \pi_2 \le 6 \cdot x_{23}\\ \pi_3 - \pi_1 &\ge 1\\ \pi_4 - \pi_1 &\ge 1\\ \pi_6 - \pi_4 &\ge 1\end{aligned}$$

$$\underset{b,\pi}{\text{Minimize}} \sum_{(i,j)\in B\cup C-(C'\cup O)} w_{ij} b_{ij}$$

$$\begin{aligned}\text{subject to} \quad & (\pi_j - \pi_i)/n \le b_{ij} \le \pi_j - \pi_i \\ & \qquad ((i,j) \in C - (C' \cup O))\\ & (\pi_j - \pi_i)/n \le b_{ij} \qquad ((i,j) \in B)\\ & (\pi_i - \pi_j)/n \le b_{ij} \qquad ((i,j) \in B)\\ & \pi_j - \pi_i \ge 1 \qquad ((i,j) \in C' \cup NC)\\ & \pi_i = \pi_j \qquad ((i,j) \in O)\\ & b_{ij} \in \{0,1\} \qquad ((i,j) \in B \cup C)\end{aligned}$$

Fig. 4. Fusing with the help of integer programming [13]

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 7, July 2017

In [13] Megiddo and Sarkar formulate weighed loop fusion problem based on linear sized integer programing which is proportional to the number of variables and constraints and proved that if they find an optimal solution for the mentioned integer programing problem, they would be able to achieve an appropriate result for weighted fusion. In this problem each pair of loops has a weight, identifying the amount of data reuse and excluded memory operations after fusion. These are stored in a two dimensional array named weight matrix, presented in figure 4. In the observed graph each edge is marked with the name of associated array and noncontractable dependence are same as fusion preventing edges. All the inequalities ensure that the resulting graph is acyclic. Mentioned formulation is optimized using branch-and-bound algorithm to achieve more efficient model. In contrast to several prior arts, they considered weighted loop fusion which is a more general case, but their representation model is loop dependence graph of just two types of loops, including parallel and sequential, and each node is a perfectly nested loop.
Kennedy [14] solved the weighted loop fusion by a heuristic which repeatedly choses the heaviest edge (weights describe amount of reuse) in each step and fuses related pair of nodes in the observed dependence graph. All the relative order of the nodes is preserved throughout the fusion process. In this representation, there are two types of fusible and not fusible nodes that each output partition does not have a not fusible node or it is a not fusible node. Proposed algorithm takes O(V(E+V)) which is preferable, but it may not produce good qualified solutions at the end due to its priority scheme while deciding on fusing specified pair of nodes based on related reuse weight. For example, the algorithm's solution for the graph of figure 5 is fusing {a,b,c,d,f} with 16 reuse; but the optimum result is {c,d,e,f} having 22 reuse. Note that the black node has different type and cannot merge with other vertexes. In addition it prevents fusing a pair of node that has it in the path between. This situation avoids merging e in the algorithm's result.

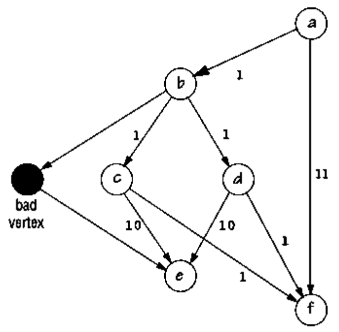


Fig. 5. An example of not finding the optimal fusion [14]

Ding and Kennedy tried to find optimal fusion by improving data caching in [5]. The strategy reorganizes a program in two steps, including computation fusion and data regrouping. The first stage fuses loops that have the same datum, decreasing amount of transferred data, shown in figure 6. But temporal locality alone is not sufficient. So the second step improves data layout of a loop to enhance spatial locality of a program by clustering a loop's data based on cache space, merging some arrays or splitting some others, as figure 7. In fact, it draws two successive references closer in terms of loops or individual arrays. But again, it only intended to improve data reuse and even in some cases can't make optimal outputs. Although this algorithm increases computational overhead by complicating loops execution, it worth when programs performances has slowed down due to the memory latency. Complexity of this method is O(N×N′×A) in which N and N′ is number of loops before and after fusion , respectively, and A is number of arrays. In this paper loop fusion is combined with data regrouping which is its effective corner stone. Moreover, it improves efficiency of the results by including other loop transformations. But it is not reliable in two cases, one of which is when similar computations are provided with the same code but various parameters; other one is when accessing to a memory location has not a specific pattern.

```
for i=2, N
  A[i]=f(A[i-1])
end for

A[1]=A[N]
A[2]=0.0

for i=3, N
  B[i]=g(A[i-2])
end for
```

```
for i=2, N
  A[i]=f(A[i-1])
  if (i==3)
    A[2]=0.0
  else if (i==N)
    A[1]=A[N]
  end if
  if (i>2 and i<N)
    B[i+1]=g(A[i-1])
  end if
end for
B[3]=g(A[1])
```

Fig. 6. Computation fusion [5]

```
for i                          for i
  for j                          for j
    g(A[j,i],B[j,i])              g(D[1,j,1,i],D[2,j,1,i])
  end for                        end for
  for j                  ⇨       for j
    t(C[j,i])                      t(D[j,2,i])
  end for                        end for
end for                        end for
```

Fig. 7. Data regrouping [5]

Verdoolaege et al. [6] provide a method that uses distance vector, including essential information for loop transformations, instead of dependence. The set of vectors that indicates dependence of statements in different loop iterations, construct a dependence polytope like figure 8 in which nodes represent statements of a program and each edge shows a dependence existence in specific iteration. The proposed algorithm selects a couple of nodes with minimum distance vector in which greater number of data is preferred, then fuse them if possible by performing some affine transformations; otherwise, just required affine transformation for improving the performance are applied. After each fusion, all the corresponding vectors are reset. It is not limited to perfectly nested loops and runs in O(VV′+E) that V′ is size of the greatest resulted loop. The complexity of the task is reduced by their mathematical view which divides searching for affine transformations to determining linear parts and identifying the offsets. Note that, this method tries to find a good solution is sufficient time, not the optimal output. The key feature of provided algorithm is its capability of fusing multi-dimensional loops in general programs; but because of only considering temporal locality, there is a slight and in some cases negligible improvement in the results.

```
for (i = 0; i <= n; i++)
  a[i]=f(i); // A
for (i = 1; i <= n; i++)
  b[i]=g(b[i-1], a[i-1]); // B
for (i = 0; i <= n; i++)
  c[i]=h(b[n], a[i]); // C
```

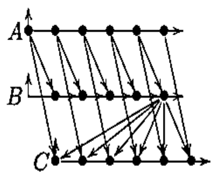


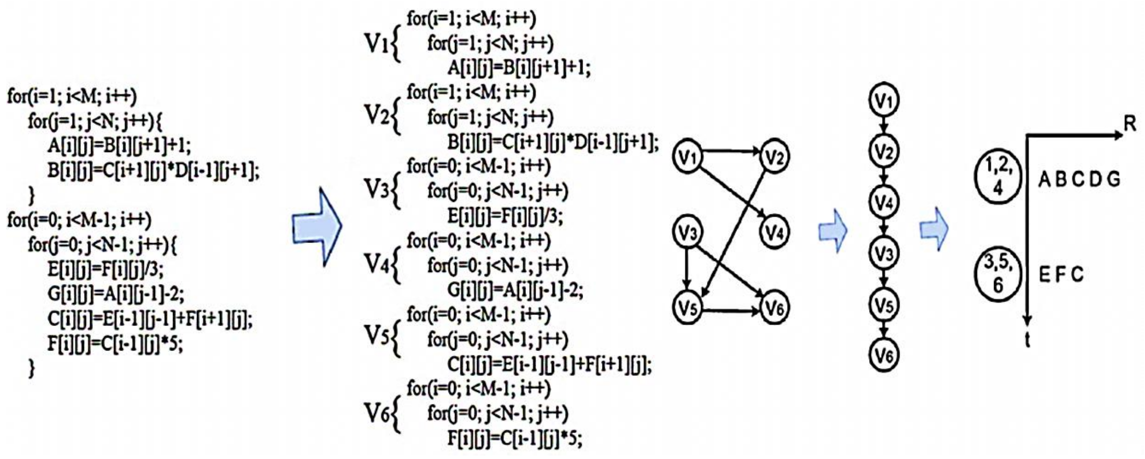
Fig. 8. An example of dependence polytope [6]

Qiu et al. discussed the impact of fusion on energy consumption in [15]. They maximize fusion by removing fusion preventing edges. Among all dependency vectors the smallest one is selected, compared based on lexicographic order, to improve the performance of the algorithm. A fusion preventing edge is represented with a negative weight on the dependence graph. These edges are eliminated by a rescheduling function ($\vec{r}(u)$) that moves a node from $\vec{i}$ th iteration to $\vec{i}-\vec{r}(u)$ and some iterations are moved to out of the loop, placed before or after it.
Tian et al., in [16], found a heuristic solution for loop fusion problem that tries to improve memory performance on stream processors when loading a program from off-chip memory into stream register file. The main objectives of this paper are minimizing number of loops and transferred data, regarding to the storage size by not allowing greater loop bodies. They have considered storage consumption (S) and data transfer time (T) in their fitness function as $f=k\times S+T$ and $T=k_s\times C_s+k_t\times C_t$. In proposed formula, k_s is a program (loop) transfer time, k_t is data (array) transfer time, C_s and C_t are number of transferred program and data, respectively. They have assessed the fitness function in three distinct cases, including when program transfer time overweight the data transfer time ($k_s\times C_s \gg k_t\times C_t$), when data transfer time is greater than program transfer time ($k_t\times C_t \gg k_s\times C_s$) and when these two are valued almost equally. All mentioned cases are proved to be NP-complete in the article and they have operated uniquely in each position as minimizing C_s, C_t and first C_t then C_s, correspondingly. The third case is illustrated in the figure 9, showing the original program in the left side. In the beginning of the algorithm, maximum loop distribution [17] is applied which decreases S and increases T and alters the code to six separate loop, followed by loop reordering and loop fusion to reduce C_s or C_t and consequently minimizing f in order to achieve optimized results. This strategy measures amount of transferred data, not taking into account fusion. Consequently, it arranges loops efficiently, not only in each partition, but also among them. Sometimes fusion did not preserve parallelism in this method and while merging, some parallel loops become sequential owing to not considering fusion preventing edges which is their main disadvantage.
Mehta et al., in [2], solved fusion problem in a polyhedral compiler framework, named PLuTo [18], providing a powerful mathematical framework based on parametric linear algebra and linear programing. In polyhedral framework, loop iterations are represented as a set of linear inequalities which forms a polyhedron; each integer point determines specific iteration. In provided formulation, a set of complicated loop transformations can be modelled as an algebraic operation. Moreover, the statement based vision of the algorithm enables it to find optimal solutions by enlarging the search space which is not possible in prior approaches. This perspective is associated with representing each statement by a single node in the dependence graph. The first fusion step in this algorithm is obtaining strongly connected

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 7, July 2017

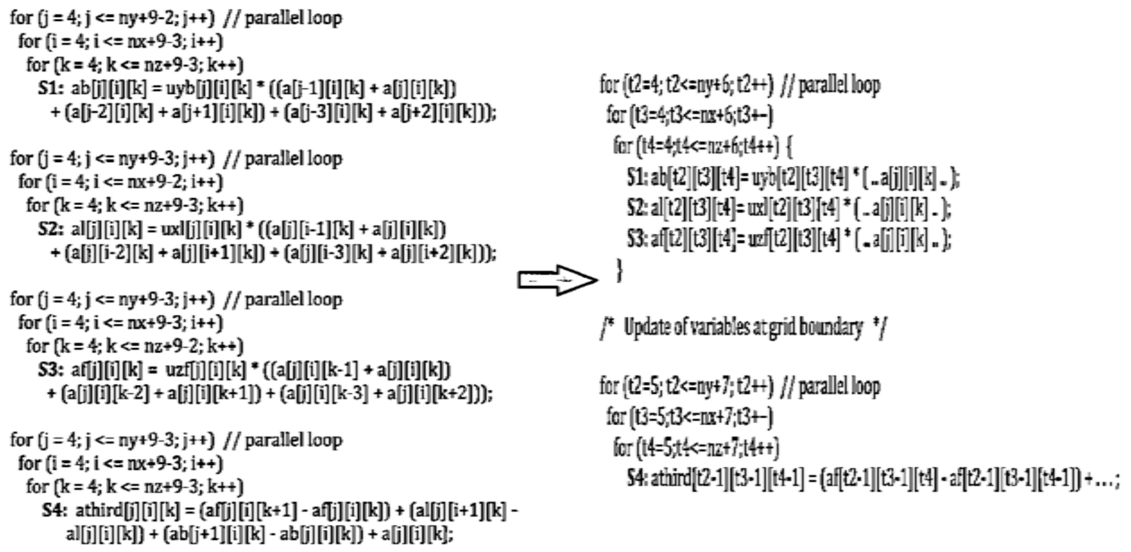
components of the data dependence graph which increases the probability of fusing consecutive nodes. Then, an appropriate order and partitioning of loops is determined, improving reuse with taking into account true and input dependences. After fusion, some parallel loops are altered to sequential because of a specific dependence. To stretch across this matter and preserving these partitions' parallelism, a cut operator is performed to eliminate the dependence with least amount of reuse loss as exposed in figure 10. In normal way all four loops are merged, but due to violated dependence of S4 the resulted loop would be sequential. In contrast, by a cut operation and insulating the S4, other loops can remain parallel. The important disadvantage of this method is lack of considering architectural parameter such as register pressure.



```
for(i=1; i<M; i++)
  for(j=1; j<N; j++){
    A[i][j]=B[i][j+1]+1;
    B[i][j]=C[i+1][j]*D[i-1][j+1];
  }
for(i=0; i<M-1; i++)
  for(j=0; j<N-1; j++){
    E[i][j]=F[i][j]/3;
    G[i][j]=A[i][j-1]-2;
    C[i][j]=E[i-1][j-1]+F[i+1][j];
    F[i][j]=C[i-1][j]*5;
  }
```

```
    for(i=1; i<M; i++)
V1{   for(j=1; j<N; j++)
        A[i][j]=B[i][j+1]+1;
    for(i=1; i<M; i++)
V2{   for(j=1; j<N; j++)
        B[i][j]=C[i+1][j]*D[i-1][j+1];
    for(i=0; i<M-1; i++)
V3{   for(j=0; j<N-1; j++)
        E[i][j]=F[i][j]/3;
    for(i=0; i<M-1; i++)
V4{   for(j=0; j<N-1; j++)
        G[i][j]=A[i][j-1]-2;
    for(i=0; i<M-1; i++)
V5{   for(j=0; j<N-1; j++)
        C[i][j]=E[i-1][j-1]+F[i+1][j];
    for(i=0; i<M-1; i++)
V6{   for(j=0; j<N-1; j++)
        F[i][j]=C[i-1][j]*5;
```

Fig. 9. Execution process in the third case [16]



```
for (j = 4; j <= ny+9-2; j++)  // parallel loop
 for (i = 4; i <= nx+9-3; i++)
  for (k = 4; k <= nz+9-3; k++)
   S1: ab[j][i][k] = uyb[j][i][k] * ((a[j-1][i][k] + a[j][i][k])
     + (a[j-2][i][k] + a[j+1][i][k]) + (a[j-3][i][k] + a[j+2][i][k]));

for (j = 4; j <= ny+9-3; j++)  // parallel loop
 for (i = 4; i <= nx+9-2; i++)
  for (k = 4; k <= nz+9-3; k++)
   S2: al[j][i][k] = uxl[j][i][k] * ((a[j][i-1][k] + a[j][i][k])
     + (a[j][i-2][k] + a[j][i+1][k]) + (a[j][i-3][k] + a[j][i+2][k]));

for (j = 4; j <= ny+9-3; j++)  // parallel loop
 for (i = 4; i <= nx+9-3; i++)
  for (k = 4; k <= nz+9-2; k++)
   S3: af[j][i][k] =  uzf[j][i][k] * ((a[j][i][k-1] + a[j][i][k])
     + (a[j][i][k-2] + a[j][i][k+1]) + (a[j][i][k-3] + a[j][i][k+2]));

for (j = 4; j <= ny+9-3; j++)  // parallel loop
 for (i = 4; i <= nx+9-3; i++)
  for (k = 4; k <= nz+9-3; k++)
   S4: athird[j][i][k] = (af[j][i][k+1] - af[j][i][k]) + (al[j][i+1][k] -
     al[j][i][k]) + (ab[j+1][i][k] - ab[j][i][k]) + a[j][i][k];
```

```
for (t2=4; t2<=ny+6; t2++) // parallel loop
 for (t3=4;t3<=nx+6;t3+-)
  for (t4=4;t4<=nz+6;t4++) {
   S1: ab[t2][t3][t4]= uyb[t2][t3][t4] * ( ..a[j][i][k] .. );
   S2: al[t2][t3][t4]= uxl[t2][t3][t4] * ( ..a[j][i][k] .. );
   S3: af[t2][t3][t4]= uzf[t2][t3][t4] * ( ..a[j][i][k] .. );
  }

/* Update of variables at grid boundary */

for (t2=5; t2<=ny+7; t2++) // parallel loop
 for (t3=5;t3<=nx+7;t3+-)
  for (t4=5;t4<=nz+7;t4++)
   S4: athird[t2-1][t3-1][t4-1] = (af[t2-1][t3-1][t4] - af[t2-1][t3-1][t4-1]) + ...;
```

Fig. 10. An example of the cut operation [2]

IV. CONCLUSION AND FUTURE WORKS

This paper has discussed different methods of loop fusion; each of them had its own merits and demerits. A summarized comparison of them is presented in table 1. To sum up, there have been numerous studies concerning fusion's impact on amount of required memory, energy consumption, data reuse and execution time. According to the obtained overview, none of them are appropriate enough, owing to their priority scheme while choosing nodes for fusion. This proceeding could eradicate random behavior of this decision and divert the algorithm to local optimums. Furthermore, they did not meet all the criteria and requirements of a perfect fusion result. According to fusion's essential role in accelerating the computational speed, there are still vacancies for a comprehensive loop fusion strategy. For instance, following cases can be considered:

•Finding a loop reordering method that improves the performance in more complex situations. Sorting Loops for fusion process, affects the output; because merging a couple of loops may preserve fusing others and selecting them by trial and error is time consuming. So, a beneficial heuristic is required.

•Forming loop fusion problem as mentioned subjects in section 2 to simplify the issue and thus design a more appropriate strategy for solving it.
•Finding best ordering of loop transformations. There are several loops optimizing transformation that if we apply them along with fusion, it will enhance the efficiency of the results. In this situation, order of these distinct transformations becomes a notable debate which is still not answered.
•Adding other computer architectural parameters for the sake of designing better fusion algorithms which are more practical in perplex occasions.

TABLE I. Comparing different loop fusion applications

[Reference]	Objective	Ordering method	Representation	Complexity	Advantages	Disadvantages
[10]	Improving parallelism – Improving data reuse	Breath-first sort with priority of parallel loops	Directed acyclic graph	$O(k \times V \times E \times \log(V^2/E))$ and $O(V \times E)$	✓Provides simple algorithms for maximizing parallelism and data locality.	×Improves parallelism and data locality in separate algorithms. ×Considers two distinct types of perfect loops as parallel and sequential. ×Other loop transformation is not included.
[11]	Minimizing number of resulted loops	Breath-first sort with priority of selected loop type	Directed acyclic graph	$O((V+E)T)$	✓More than one type of loops is considered.	×Mentioned samples are simple for evaluating the performance of the algorithm. ×Pre-set of loop types is required. ×Other loop transformation is not included.
[4]	Improving parallelism and cache locality	Bottom up	Weighed acyclic tree	linear	✓Optimizes fusion in terms of parallelism and data locality. ✓Provided fusion is profitable for different architectures based on storage size.	×Applies fusion on maximal weight spanning tree. ×Computer architectural impact is not considered comprehensively.
[13]	Preserving parallelism and improving data reuse	Topological sort	Weighted directed acyclic graph	linear	✓Provides a solution based on integer programing for typed fusion which is more general case than previous arts. ✓Preserving parallelism and improving data reuse are considered while fusion.	×Pre-set of loop order is required. ×Two types of loops are considered. ×Other loop transformation is not included.
[14]	Improving data reuse	Topological sort with priority of maximum weight	Weighted acyclic graph	$O(EV+V^2)$	✓An algorithm for weighted loop fusion is proposed which runs in suitable speed.	×Fusion results of this method are not well qualified. ×Other loop transformation is not included. ×Just data reuse improvement is considered.
[5]	Improving data reuse	Topological order with priority of data reuse	Applied on code	$O(N \times N' \times A)$	✓Loop fusion and data regrouping have combined which elevated efficiency of the algorithm. ✓Other loop transformations are considered.	×Just data reuse improvement is considered. ×The algorithm cannot provide appropriate output when memory access pattern is complicated.
[6]	Improving data reuse	True dependence with minimum distance vector and maximum data	Directed graph	$O(VV'+E)$	✓By using this approach, loop fusion could easily combine with other transformations. ✓Key features are its applicability to general programs, simplicity and reasonable speed.	×Intended to improve temporal locality and not spatial locality, thereby resulting advancements are negligible in some cases.
[15]	Energy consumpti	Lexicographically the	Loop dependenc	-	✓By combining fusion and scheduling, it elevates the	×Other loop transformations, especially those which are related

	on minimization	smallest dependency vector	e graph		performance and power consumption in computer systems.	to systems' architecture, are not considered.
[16]	Improving memory performance on stream processors	Topological sort using a reordering heuristic method	Directed acyclic graph	-	✓This method, unlike most previous methods, improves data locality by considering storage as a parameter for leading to better results.	×Fusion preventing edges are not considered. These edges are mentioned when preserving parallelism is important in an algorithm.
[2]	Preserving parallelism and improving data reuse	Order of loops in the program	Directed acyclic graph	-	✓Improves data locality and preserving parallelism. ✓Other loop transformations are considered.	×Other architectural parameters could have been considered.

REFERENCES

[1] Kennedy, K., & Allen, J. R. (2001). Optimizing compilers for modern architectures: a dependence-based approach.
[2] Mehta, S., Lin, P. H., & Yew, P. C. (2014, February). Revisiting loop fusion in the polyhedral framework. In ACM SIGPLAN Notices (Vol. 49, No. 8, pp. 233-246). ACM.
[3] Darte, A. (2000). On the complexity of loop fusion. Parallel Computing, (Vol. 26, No. 9, pp. 1175-1193).
[4] Singhai, S. K., & McKinley, K. S. (1997). A parametrized loop fusion algorithm for improving parallelism and cache locality. The Computer Journal, (Vol. 40, No. 6, pp. 340-355).
[5] Ding, C., & Kennedy, K. (2001, April). Improving effective bandwidth through compiler enhancement of global cache reuse. In Parallel and Distributed Processing Symposium., Proceedings 15th International (pp. 10-pp). IEEE.
[6] Verdoolaege, S., Bruynooghe, M., Janssens, G., & Catthoor, F. (2003, June). Multi-dimensional incremental loop fusion for data locality. In Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on (pp. 17-27). IEEE.
[7] Bernstein, A. J. (1966). Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers, (5), 757-763.
[8] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., & Wolfe, M. (1981, January). Dependence graphs and compiler optimizations. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 207-218). ACM.
[9] Allen, R., & Kennedy, K. (1987). Automatic translation of Fortran programs to vector form. ACM Transactions on Programming Languages and Systems (TOPLAS), 9(4), 491-542.
[10] Kennedy, K., & McKinley, K. S. (1993). Maximizing loop parallelism and improving data locality via loop fusion and distribution (pp. 301-320). Springer Berlin Heidelberg.
[11] Kennedy, K., & McKinley, K. S. (1994). Typed fusion with applications to parallel and sequential code generation. Rice University, Department of Computer Science.
[12] Lukes, J. A. (1974). Efficient algorithm for the partitioning of trees. IBM Journal of Research and Development, 18(3), 217-224.
[13] Megiddo, N., & Sarkar, V. (1997, June). Optimal weighted loop fusion for parallel programs. In Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures (pp. 282-291). ACM.
[14] Kennedy, K. (2001). Fast greedy weighted fusion. International Journal of Parallel Programming, (Vol. 29, No. 5, pp. 463-491).
[15] Qiu, M., Sha, E. H. M., Liu, M., Lin, M., Hua, S., & Yang, L. T. (2008). Energy minimization with loop fusion and multi-functional-unit scheduling for multidimensional DSP. Journal of Parallel and Distributed Computing, (Vol. 68, No. 4, pp. 443-455).
[16] Tian, W., Xue, C. J., Li, M., & Chen, E. (2012). Loop fusion and reordering for register file optimization on stream processors. Journal of Systems and Software, (Vol. 85, No. 7, pp. 1673-1681).
[17] Liu, M., Zhuge, Q., Shao, Z., Xue, C., Qiu, M., & Sha, E. M. (2005, December). Maximum loop distribution and fusion for two-level loops considering code size. In Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on (pp. 6-pp). IEEE.
[18] Pluto: An automatic parallelizer and locality optimizer for multicores. Available online at http://pluto-compiler.sourceforge.net

BIOGRAPHIES

Mahsa Ziraksima is majored from the Department of Computer Science, Faculty of Mathematical Sciences at University of Tabriz, Iran with B.Sc. and M.Sc. in Computer Science. Her research interests are evolutionary computation and optimizing compilers.

Shahriar Lotfi received the B.Sc. in Software Engineering from the University of Isfahan, Iran, the M.Sc. degree in Software Engineering from the University of Isfahan, Iran, and the Ph.D. degree in Software Engineering from Iran University of Science and Technology in Iran. He is currently an assistant professor of computer Science at the University of Tabriz. His research interests include compilers, super-compilers, parallel algorithms, evolutionary computing and algorithms.

Habib Izadkhah is an Assistant Professor in the Department of Computer Science, Faculty of Mathematical Sciences at University of Tabriz, Iran. He holds a Ph.D. degree in Computer Science from University of Tabriz. His current researches focus on view based software engineering and reverse engineering.