A Framework for Loop Distribution on Limited On-Chip Memory Processors*

Lei Wang, Waibhav Tembe, and Santosh Pande**

Compiler Research Laboratory, Department of ECECS, ML 0030,
University of Cincinnati, PO Box 210030, Cincinnati, OH 45221-0030
leiwang,wtembe,santosh@ececs.uc.edu

Abstract. This work proposes a framework for analyzing the flow of values and their re-use in loop nests to minimize data traffic under the constraints of limited on-chip memory capacity and dependences. Our analysis first undertakes fusion of possible loop nests intra-procedurally and then performs loop distribution. The analysis discovers the *closeness factor* of two statements which is a quantitative measure of data traffic saved per unit memory occupied if the statements were under the same loop nest over the case where they are under different loop nests. We then develop a greedy algorithm which traverses the program dependence graph (PDG) to group statements together under the same loop nest legally. The main idea of this greedy algorithm is to transitively generate a *group* of statements that can legally execute under a given loop nest that can lead to a minimum data traffic. We implemented our framework in Petit [2], a tool for dependence analysis and loop transformations. We show that the benefit due to our approach results in eliminating as much as 30 % traffic in some cases improving overall completion time by a 23.33 % for processors such as TI's TMS320C5x.

1 Introduction

1.1 On-Chip Memory and Data Traffic

Due to significant advances in VLSI technology, 'mega-processors' made with a large number of transistors has become a reality. These processors typically provide multiple functional units which allow exploitation of parallelism. In order to cater to the data demands associated with parallelism, the processors provide a limited amount of on-chip memory. The amount of memory provided is quite limited due to higher area and power requirements associated with it. Even though limited, such on-chip memory is a very valuable resource in memory hierarchy. Due to tight integration and careful layouts, the latency delays amongst the on-chip resources are significantly smaller; thus, on-chip memory can serve as an ultra-fast cache as shown by Panda, Nicolau and Dutt [15]. One of the significant uses of on-chip memory is to store spill values as shown by

* Supported in part by NSF through grant no. #EIA 9871345
** Contact author for future communications about this paper

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 141–156, 2000.

Cooper et. al. [17]. Another important use of on-chip memory is to hold the instructions from short loops along with the associated data for very fast computation. Such schemes are very attractive on embedded processors where, due to the presence of dedicated hard-ware on-chip (such as very fast multipliers-shifters etc.) and extremely fast accesses to on-chip data, the computation time of such loops is extremely small meeting almost all real-time demands. Biggest bottleneck to performance in these cases are off-chip accesses and thus, compilers must carefully analyze references to identify good candidates for *promotion* to on-chip memory. In our earlier work [6,5], we formulated this problem in terms of 0/1 knapsack and proposed a heuristic solution that gives us good promotion candidates. Our analysis was limited to single loop nest. When we attempted extending this framework to multiple loop nests (intra-procedurally), we realized that not only it is important to identify good candidates for promotion but a careful restructuring of loops must be undertaken *before performing promotion* since data traffic of loading and storing values to on-chip memory poses a significant bottleneck.

Reorganization of loop nests is quite useful for signal processing applications which typically consist of a sequence of loop nests that tend to operate on the same data elements performing operations such as DCT transform, averaging, convolution, edge detection etc. in succession. Valuable data traffic can be saved in such cases by compiler analysis of flow of values to maximize re-use. This is the focus of this work.

2 Motivating Example: On-Chip Memory and Parallelism

Consider the following loop nests to be executed on a processor that can support 300 parallel adders. Assume that the processor has on-chip memory which can hold 300 data elements (this is a typical size for most on-chip memories which ranges from 128 words to about 512 words).

```
Example:
For i=1 to 100                  //L1
   m[i] = a[i] + e[i];            //S1
   n[i] = b[i] + c[i];            //S2
For i=1 to 100                  //L2
   p[i] = f[i] + d[i];            //S3
   q[i] = e[i-1] + m[i] + a[i+1];//S4
```

Consider executing the above loop nests (without any restructuring), on this processor. The first loop nest L1 needs 200 parallel adders and the second one L2 also needs 200 parallel adders. Since the processor can support 300 parallel adders, both of these needs are satisfied as far as the parallelism is concerned. However, due to limited on-chip memory, both loops must be blocked. The loop L1 is blocked with size 50 since each iteration demands storage for 6 data elements. The loop L2 is blocked with size 42 since each iteration demands storage for 7 elements. Thus, one can see that although the processor can support 300

adders, loop L1 can utilize only 100 parallel adders and loop L2 can utilize only 84 parallel adders due to the limitations on data storage. The total data traffic for the above blocked loop nests is 1300 (900 input values loaded to on-chip memory and 400 output values stored from on-chip memory into main cache). This traffic is essentially sequential and poses the largest overhead on loop execution. We now show that if we restructure the loops we can significantly reduce the costly traffic and also utilize parallelism more effectively. We see that arrays a[] and e[] are used in both S1 and S4, and m[i] (output of S1) is used in S4. Thus, statements S1 and S4 exhibit a higher degree of 'closeness' to each other since their execution under one loop nest can significantly increase data re-use and reduce effective traffic. We can thus group them together. Statements S2 and S3 however share no such data re-use and thus, we can put them under separate loop nests. Using the memory capacity constraint, the blocked loops are as follows:

```
Output Loop:
For i = 1 to 100 by 75           //L1
 For j = i to min(i+74, 100)
   m[j] = a[j] + e[j];           //S1
   q[j] = e[j-1] + m[j] + a[j+1];//S4
For i=1 to 100                   //L2
   n[i] = b[i] + c[i];           //S2
For i=1 to 100                   //L3
   p[i] = f[i] + d[i];           //S3
```

The block size for the first loop is found 75 since each iteration of the inner loop needs storage for 4 elements per iteration and since we have a memory capacity of 300 elements, we can 'block' 75 iterations together. Thus, during the execution we get fetch the entire data needed for complete execution of the blocked loop, so that no intermediate traffic is needed. Once the loop is executed, we output all the necessary data elements generated/modified in the program. No blocking is needed for other two loop nests. In the above restructured loops, the first loop nest utilizes a parallelism of 150 parallel adders whereas the second and third loop nests utilize parallelism of 100 parallel adders. One can see that these loop nests not only exhibit better parallelism but also have much lower data traffic needs. The first loop nest has total traffic of 400 (200 loads and 200 stores). The other two loop nests have traffic of 300 each (200 loads and 100 stores). Thus the total data traffic for the restructured loop nests is 1000. The total data traffic is thus reduced from 1300 to 1000 or a saving of about 23% is achieved. As mentioned earlier, the data traffic is sequential and has a higher latency and thus forms a dominant part of overall loop completion time. The reduction in data traffic has a more significant effect on reduction of overall completion time of the loop. This motivates our approach of restructuring the loop nests for minimizing data traffic. Our approach is based on determining a group of statements that should execute under a given loop nest to minimize data traffic. In the next section, we present an outline of our approach introducing terms and definitions.

3 Terms and Definitions

3.1 Outline of Our Approach

Our analysis begins by undertaking loop fusion assuming unlimited memory. We first calculate the amount of data re-use between the statements after carrying out loop fusion. We then calculate the *closeness factor* of every pair of statements. Closeness factor between two statements quantifies the amount of data reuse per unit memory required. We decorate the program dependence graph (PDG) with this information by inserting undirected edges between the nodes of the PDG that represent statements. The undirected edges have a weight equal to the closeness factor of the two statements. We then group the statements greedily under a given loop nest i.e. statements which have higher closeness factor are grouped together over those which have lesser. At every step, when we group statements, we examine if we must include some other statement(s) so as to preserve the dependences. (this step is explained in the next subsection). Sometimes we may not be able to include statements due to dependence and memory capacity constraints. In such cases, we adjust the dependence groups to eliminate *useless* re-use edges. Finally, we iterate to expand the group of statements and when we exceed the capacity of the on-chip memory we halt. We carry out the above steps of grouping and adjusting the dependence groups until there are no more re-use edges. We finally block the groups so as to fit the available memory size.

In our approach, we analyze following three types of data reuse: flow dependence reuse, input dependence reuse, and iteration reuse. We calculate the total amount of data re-use between two statements based on the total re-use associated with each of the references within those statements adding them up. Closeness Factor (CF) of two statements is defined as the ratio of the total re-use present between these statements and the total number of data elements that occupy the memory to facilitate the re-use. Thus, the CF is a measure of how good is the re-use relative to a unit of memory occupied by references corresponding to the two statements. We say that two statements are closest to each other if they have the highest closeness factor. Thus, we rank the statements in the order of closeness factor. Our approach is to legally group statements in the order of CF under the constraints of available memory size and dependencies. The motivation behind grouping two 'closest' statements together under the same loop nest is that largest amount of data traffic is eliminated by doing so. This is due to the fact that by grouping statements, we have utilized maximum possible reuse per unit memory. We propose a greedy strategy to accomplish this goal. When we start grouping the statements together, certain re-use edges from the decorated PDG become useless. In other words, it becomes impossible to group statements together under one loop nest due to the groups already formed and due to dependence constraints. The next section explains the details along with an algorithm.

4 PDG Analysis

4.1 Legality of Grouping

Our framework starts grouping statements together greedily. When we find out a statement S is the closest statement to a group A, we want to check the following to make sure we can group S and A together: (1) We first want to check if we can group S and A legally according to PDG (2) whether there is sufficient memory available (3) when the above two situations are satisfied, if grouping one more statement will really increase CF. In this section, we discuss (1), the most important condition. There are some interesting issues faced in determining legality. In Figure 1, we have found a statement group Gi having three statements S2, S3 and S6 in it. We assume that at this point, no more statement can be grouped with Gi because of on-chip memory capacity. Amongst the remaining statements S1, S4, S5 and S7, statements S5 and S7 are the closest statements and we group them into Group A. At this stage, we find out that S4 is the next closest statement to A, but we can not group it with A. The reason is S4 must execute before Gi which must execute before A and thus, S4 can not be a part of A.

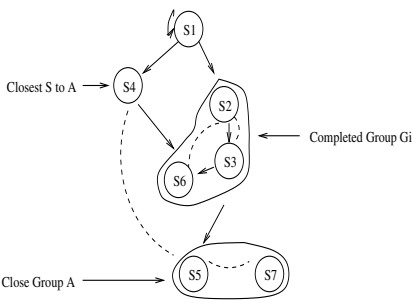

Fig. 1. Legality of grouping S4 with A

4.2 PDG Adjustment

After we find out that the grouping is legal, we need to adjust PDG for further PDG analysis use. In general, after we group S with X, one of the following scenarios may arise:

Case 1 If S and X are each other's direct and only successor/predecessor after they are grouped, S and X's other direct successors will be direct successors of this group; and their other direct predecessors will be direct predecessors of the group. In figure 2, after S2 and S3 are grouped, S1 is their predecessor and S5, S6 are their successors.

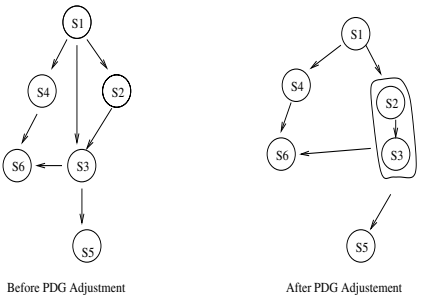

Fig. 2. Direct Predecessor/Successor Relationship

Case 2 If S and X belong to the same dependency group and they are not each other's successor and predecessor, but they have same (direct/indirect) and only successor(s)/predecessor(s), the two statements will be grouped together and their direct successors and predecessors will be the direct successors and predecessors of the group. In figure 3, after S3 and S4 are grouped together, S1 and S2 are their predecessors and S5, S6 are their successors.

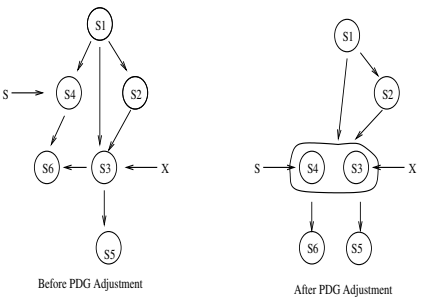

Fig. 3. Common Successor/Predecessor Relationship

Case 3 In some cases we cannot directly include a statement in a group without including some other statements. If two closest statements are in the same group and are each other's non-direct successor and predecessor (suppose S is X's predecessor). In this case, we have to resort to the following:

1. Find out all the statements between them in the PDG, that is, find out all of intermediate statements between S and X. We call this set as the intermediate statement set of S and X : ISGroup(S,X).
2. Count total data storage requirements of statements S, X and ISGroup(S,X), and compare it with available memory. If it can fit, group S, X and ISGroup(S,X). If the capacity is not big enough, S and X can not be grouped

together and the re-use edge must be discarded. From figure 4, one can see that if capacity is big enough, statements S1, S2, S3, S4 and S6 will be grouped into group G. Otherwise, input dependence reuse between S1 and S6 will be removed and no group is formed.

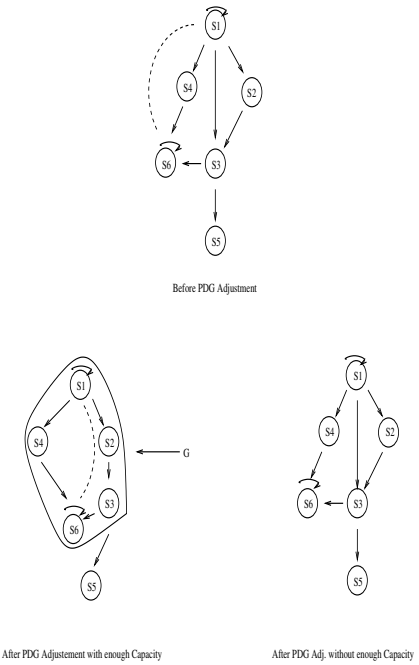

Fig. 4. Indirect Predecessor/Successor Relationship

In the above three cases, the two closest statements are in the same dependency group.

Case 4 If two closest statements belong to different dependency groups, there is no clear execution order for the two statements, that is, the two statements can be executed in any order. In figure 5, S1 must be executed before S3 and S6; however, S5 can be executed in any order with respect to S1, S3 or S6. However, when S3 and S2 are grouped together under one loop nest, it automatically fixes the order of execution of S5 with respect to S1 and can't be potentially combined with it.It can still be combined with any one of S4, S6 or S8 though. Thus, as long as the two statements S and X or one statement and a group belong to different dependency groups, they can be group together. The two dependency groups should be combined into one, predecessors and successors of S and/or X will be the predecessors and successors of the group. We have devised an

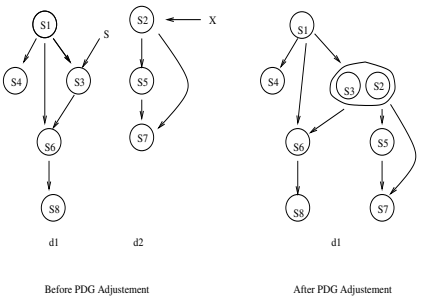

Fig. 5. S and X are in different Dependence Groups

algorithm based on these ideas to check possible groupings of the statements and the CF of the group is an indicator of the profitability. It is shown below.

```
Input : PDG for the given loop
Output: Rearranged statements.
1.For every pair of ungrouped statements
  calculate the closeness factor(CF).
2.while(some ungrouped statements can be grouped)
 2.A.select the pair of ungrouped statements with
     highest CF
 2.B.Check the possibility of grouping based on
     available memory size
 2.C.if grouping possible
   2.C.a. Form the new group
   2.C.b. Adjust the parents and children of
          all statements in the new group
   2.C.c. Find all possible ungrouped statememts
          that share data with this group and
          if such statement found
            merge that statements into the group
            (if memory size allows)
            goto 2.C.b
          else goto 2.A
     else
     set the CF of this pair to zero.
     goto 2.
```

5 Complexity

The algorithm attempts to first form a group using two statements. It then examines statements which can be included in this group to facilitate more data re-use based upon a re-use edge. This step could potentially involve examining

O(|V|) nodes where |V| is the number of nodes in the graph. In worst case the number of re-use edges present in the graph could be $|V^2|$. Thus, the overall complexity of the algorithm in worst case can be O($|V^3|$).

6 Results

Table 1. Benchmark I Data Traffic Comparison. Ideal Traffic is 280,006

OnChip MemSize	Original Traffic	Our Traffic	Traffic Improved
24	440,000	380,000	15.79%
36	410,000	341,500	16.71%
48	400,000	362,667	9.33%
60	397,142	332,500	16.28%
74	388,300	318,182	18.06%
112	388,000	304,706	21.47%
140	385,128	291,235	24.38%
172	384,953	286,482	25.19%
224	382,477	281,134	26.49%

Typically in DSP applications, a sequence of steps (such as convolution, filtering, edge detection etc.) operate successively on data. The procedure calls corresponding to these steps take place one after another. We first turned on inlining and formed a sequence of intra procedural loop nests using the DSP kernels. We then performed fusion and performed our analysis. Please refer to Appendix A for the codes of the benchmarks (the codes shown in each benchmark are sequences of loop kernels inlined intra procedurally). We used TMS320C5x for analysis. The framework has been implemented in Petit [2], a tool for data dependency analysis developed by University of Maryland. Petit provides a good support for finding the dependence distances as well as types of dependences between them.

6.1 Benchmark I

Table 1 shows the comparison of original traffic and traffic after applying our work when on chip memory size varies. The ideal (minimum possible) data traffic for the above example is about 280,006.

6.2 Benchmark II

Table 2 shows the comparison of original traffic and traffic after applying our work when on chip memory size varies for Benchmark II. It also shows the improvement. We assume the loop size N = 10000. The ideal amount of data traffic for Benchmark II is 320,005. The ideal amount of data traffic for Benchmark III

Table 2. Benchmark II data traffic comparison.

On-Chip MemSize	Original Traffic	Our Traffic	Traffic Improved
24	380,625	371,647	2.36%
36	363,750	352,466	3.10%
48	355,312	343,677	3.27%
60	350,250	338,634	3.32%
74	346,530	335,031	3.32%
112	340,945	329,664	3.31%
140	338,709	327,728	3.24%
172	337,105	326,273	3.21%
224	335,435	324,775	3.18%
268	334,614	323,986	3.17%

is 430,004. Table 3 shows the comparison when on chip memory size varies on Benchmark III. It also shows the improvement. Benchmark III is composed of

Table 3. Benchmark III data traffic comparison. Ideal Traffic was 430,004.

On-Chip MemSize	Original Traffic	Our Traffic	Traffic Improved
24	690,971	620,000	10.27%
36	597,756	572,857	4.16%
48	544,000	533,414	1.94%
60	498,987	478,888	4.03%
74	464,256	461,111	0.68%
112	455,735	450,000	1.26%
140	452,419	445,730	1.48%
172	450,073	442,363	1.71%
224	447,661	439.379	1.85%

two large groups of statements and its own data reuse is very strong. That can explain why our improvement on data traffic is around 1-2 percent most of the time. Thus, the additional gain in re-use due to our work is small.

The code for Benchmark V as shows a loop used frequently in DSP applications. The first one is a matrix multiplication code. The second one is a convolution. The results have been summarized in Table 4.

7 Related Work

We now contrast our work with existing work related to solving data locality and data re-use problems on memory hierarchy. Two important directions of work are: Tiling or iteration space blocking [12] and data-centric approaches

Table 4. Traffic comparison for benchmark IV, V and VI

Code	Original Traffic	Optimized Traffic	Saved Traffic	% Traffic Reduction
Benchmark IV	30000	21000	9000	30
Benchmark V	$18*10^6$	$11.25*10^6$	$6.75*10^6$	37.5
Benchmark VI	27692	22532	5160	18.63

such as data shackling [14]. In tiling or data blocking (which is a control centric transformation), a loop nest is tiled to maximize temporal locality [13,7,8,16]. Previous research on optimizing compilers [18] [19] [11] has proposed algorithms to detect and perform loop interchange to increase temporal locality. In data centric transformations, all the iterations that touch a data shackle are executed together giving better control to the compiler to directly focus on data than resorting to side effect of control centric transformation [14].

Our work differs from these in that we focus on data traffic as against issues of locality. This is important since in our problem, we are faced with a small amount of memory that results in excessive load/stores of short arrays between on-chip and off-chip memories. Thus, in order to minimize data traffic, we must not only concentrate on re-use of fetched values (as is typically the goal in the most memory hierarchy oriented optimizations described above) but also carefully analyze the flow and use of generated values and transitive closure of their uses and values which they generate in turn. This is because on-chip memory can be viewed somewhat intermediate between caches and register files. It is much smaller than traditional caches leading to the above differences. However, it is larger than register files. This also leads to different issues than traditional load/stores. In fact the on-chip memory is *large enough* to act as an ideal temporary store for intermediate array subsections with short live ranges. To maximally utilize the property of this temporary store, we must analyze the tradeoffs between values that get re-used across iterations (iteration re-use) input values that get re-used across two statements (due to input dependencies) and most importantly values that are produced and consumed (due to flow dependencies) in short span of iterations under the constraint of limited memory capacity and legality. Gupta et. al. [23] [22] have addressed the problem of register allocation for subscripted array variables by analyzing their liveness based on the number of iterations that elapse between definition and use. They propose a register allocation algorithm based on this liveness information. Our work addresses orthogonal problem to theirs - in our approach we are interested in determining the best grouping of statements inside a loop nest such that best re-use per memory occupied results to minimize. Our work also differs from McKinley and Kennedy [20] and Gao and Sarkar [21] in that we define a new measure of data traffic based on closeness factor than simply attempting to maximize data re-use as in their loop fusion framework. It can be shown that simply attempting to maximize data re-use can incur higher data traffic than found by using closeness factor.

8 Conclusion

We have proposed a framework on how to get a better performance by analyzing the flow of values and their re-use to effectively reduce data traffic for limited on-chip memory processors. A new concept of Closeness Factor has been developed which is the measure of data reuse between statements per unit memory requirement. The loop restructuring algorithm proposed by us helps to effectively utilize the on-chip memory while preserving the data dependences between the statements in the loop. Good performance enhancements for DSP codes are obtained using our framework. These loop restructuring transformations should be very useful for limited on-chip memory processors.

References

1. J. Eyre and J.Bier,"DSP Processors Hit the Mainstream", 'COMPUTER', 31(8):51-59, August 1998.
2. Petit, Uniform Library, Omega Library, Omega Calculater. 'http://www.cs.umd.edu/projects/omega/index.html' 141, 149
3. Texas Instruments. 'TMS 320C5x User's Guide.
4. Embedded Java. http://java.sun.com/products/embeddedjava/.
5. A. Sundaram and S. Pande, "Compiler Optimizations for Real Time Execution of Loops on Limited Memory Embedded Systems", *Proceedings of IEEE International Real Time Systems Symposium*, Madrid, Spain, pp.154–164. 142
6. A. Sundaram and S. Pande, "An Efficient Data Partitioning Method for Limited Memory Embedded Systems", *1998 ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (in conjunction with PLDI '98)*, Montreal, Canada, Springer–Verlag, pp. 205–218. 142
7. F.Irigoin and R.Triolet, "Supernode Partitioning". in *15th Symposium on Principles of Programming Languages (POPL XV)*, pages 319–329, 1988. 151
8. J.Ramanujam and P.Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers." *Journal of Parallel and Distributed Computing*, 16:108–120, 1992. 151
9. U. Banerjee, "Loop transformations for restructuring compilers", Boston: Kluwer Academic, 1994.
10. W. Li, "Compiling for NUMA parallel machines", Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.
11. M. Wolfe, *High Performance Compilers for Parallel Computing, Addison Wesley, 1996.* 151
12. M. Wolfe, "Iteration space tiling for memory hierarchies" in *Third SIAM Conference on Parallel Processing for Scientific Computing, December 1987.* 150
13. R. Schreiber and J. Dongarra, "Automatic Blocking of Nested Loops". *Technical report, RIACS, NASA Ames Research Center, and Oak Ridge National Laboratory, May 1990.* 151
14. I. Kodukula, N. Ahmed and K. Pingali, "Data Centric Multi-level Blocking" in *ACM Programming Language Design and Implementation 1997 (PLDI '97)*, pp. 346–357. 151

15. P. Panda, A. Nicolau and N. Dutt, "Memory Organization for Improved Data Cache Performance in Embedded Processors", *Proceedings of 1996 International Symposium on System Synthesis.* 141
16. N. Mitchell, K. Hogstedt, L. Carter and J. Ferrante, "Quantifying the Multi-level Nature of Tiling Interactions", International Journal of Parallel Programming, Vol 26, No 6, 1998, pp. 641–670. 151
17. K. Cooper and T. Harvey,"Compiler Controlled Memory", *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 3-7, 1998, San Jose, CA. 142
18. K. McKinley, S. Carr, and C.-W.Tseng, Improving data locality with loop transformation. ACM Transactions on Programming Languages and Systems (PLDI) 18(4):424-453, July 1996. 151
19. M. Wolf and M. Lam, "A data locality optimizing algorithm", in proceedings of *ACM Special Interest Group on Programming Languages (SIGPLAN) 91 Conf. Programming Language Design and Implementation(PLDI'91)*, pp. 30-44, Toronto, Canada, June 1991. 151
20. K. Kennedy and K. McKinley, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution" in *Languages and Compilers for Parallel Computing (LCPC) 1993.* 151
21. G. Gao, R. Olsen, V. Sarkar and R. Thekkath "Collective Loop Fusion for Array Contraction" in *Languages and Compilers for Parallel Computing (LCPC) 1992.* 151
22. E. Dusterwald, R. Gupta and M.Soffa, "A Practical Data-flow Framework for Array Reference Analysis and its Application in Optimization" in *ACM Programming Language Design and Implementation (PLDI)* 1993 pp. 68-77. 151
23. R.Gupta and R. Bodik, "Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures," in *Eighth Annual Workshop on Languages and Compilers for Parallel Computing (LCPC) 1995.* Also published in International Journal of Parallel Computing, Vol. 24, No. 6, pages 481-512,1996. 151

9 Appendix

Benchmark I

```
for(i=0; i<N; i++)
  tmpPtr[i] = (a1[i] * b[i]);
for(i=0; i<N; i++)
  c[i] = ((sum[i] * a1[i]) >> 15) + b[i];
for(i=0; i<N; i++)
  r[i] = rd[i];
  g[i] = gd[i];
  b[i] = bd[i];
  p[i] = (r[i]&mask[i])+((g[i]&mask[i])>>5)
            + ((b[i] & mask[i]) >> 10);
for(i=0; i<N; i++)
  k[i] = c[i];
  h[i] = b[i] + (k[i] * m[i]);
  f[i] = b[i] * k[i] + m[i];
for(i=0; i<N; i++)
  sum[i] = a[i] + a[i];
for(i=0; i<N; i++)
  tmpPtr2[i] = tmpPtr[i];
for(i=0; i<N; i++)
  R1[i] = tmpPtr[i] * tmpPtr2[i];
  G1[i] = tmpPtr[i] * tmpPtr[i];
for(i=0; i<N; i++)
  sum1[i] = mask[4+i-1]-a[4+i-1];
  sum2[i] = mask[4+i-2]-a[4+i-2];
  sum3[i] = mask[4+i-3]-a[4+i-3];
  sum4[i] = mask[4+i-4]-a[4+i-4];
  oPtr[4+i] = sum[i];
  sum5[i] = oPtr[4+i];
```

Benchmark II

```
for (i = 0; i < n; i++)
  b[i] = ia[i];
  f[i] = b[i] * k[i];
  a[i] = f[i] * k[i] + b[i];
  g[i] = a[i];
for(i=0; i < n; i++)
  c[i] = w[2*j];
  s[i] = w[2*j+1];
  m[i] = ia[i] + n2[i];
  rtemp[i] = x[i] - y[2*i+1];
  x1[i] = x[i] + y[2*i+2];
  itemp = m[i] - 2*m[i];
```

```
  x2[i] = x1[i] + 2*m[i];
  x3[i] = c[i]*rtemp[i] - s[i]*itemp[i];
  x4[i] = c[i]*itemp[i] + s[i]*rtemp[i];
 for (i=0; i < n; i++)   {
  sum0[i] = x[i+0]*p[i] + x2[i+1]*p[i]
          + x3[i+2]*p[i];
  sum1[i] = x[i]*p[i] + x2[i+2]*p[i]
          + x3[i+3]*p[i];
  dist0[i] = sum0[i] - point[i];
  dist1[i] = sum1[i] - point[i];
  dist3[i] = fabsf[i];
  dist4[i] = fabsf[i];
  retval[i] = (int)&z[i+3];
```

Benchmark III

```
for(j=0; j<n2; j++)
  ia2[j] = ia1[j] + ia1[j];
  ia3[j] = ia1[j] + ia2[j];
  co1[j] = w[j*2];
  si1[j] = w[j*2+1];
  co2[j] = w[j*2];
  si2[j] = w[j*2+1];
  co3[j] = w[j*2];
  si3[j] = w[j*2+1];
for(j=0; j<n2; j++)
  i2[j] = i1[j] + n2[j];
  i3[j] = i2[j] + n2[j];
  r1[j] = x[j*2] + x[j*2];
  r3[j] = x[j*2] - x[j*2];
  s1[j] = x[j*2+1] + x[j*2+1+2];
  s3[j] = x[j*2+1] - x[j*2+1+2];
  r2[j] = x[j*2] + x[j*2+3];
  r4[j] = x[j*2] - x[j*2+3];
  s2[j] = x[j*2+1] + x[j*2+1+4];
  s4[j] = x[j*2+1] - x[j*2+1+4];
  x[j*2] = r1[j] + r2[j];
  r12[j] = r1[j] - r2[j];
  r11[j] = r3[j] - s4[j];
  r13[j] = r3[j] + s4[j];
  z[j*2+1] = s1[j] + s2[j];
  s12[j] = s1[j] - s2[j];
  s11[j] = s3[j] + r4[j];
  s13[j] = s3[j] - r4[j];
  x1[j] = co1[j]*r3[j] + si1[j]*s3[j];
  x2[j] = co1[j]*s3[j] - si1[j]*r3[j];
  x3[j] = co2[j]*r2[j] + si2[j]*s2[j];
```

```
   x4[j] = co2[j]*s2[j] - si2[j]*r2[j];
   x5[j] = co3[j]*r1[j] + si3[j]*s1[j];
   x6[j] = co3[j]*s1[j] - si3[j]*r1[j];
   il[j] = i0[j] + n2[j];
```

Benchmark IV

```
 for (i = 0; i < N; i++)
   m[i] = sd[i] + sd[i + 1];
   m5[i] = sd[i] - sd[i + 1];
   m11[i] = -sd[i] + sd[i + 1];
   m14[i] = -sd[i] - sd[i + 1];
 for (i=0; i < N; i++)
   new_s = m[i] + m11[i];
   old[i] = m5[i] + m14[i];
 for (i=0; i<N; i++)
   a[i] = old[i] + new_s[i];
   b[i] = - mi[i] * 2
   term[i] = a[i] + sd[i];
   trans[i] = b[i] + mi[i];
   mj[i] = trans[i];
```

Benchmark V

```
do I = 1 .. N
 do J = 1 .. N
  do K = 1 .. N
    C[I,J] = C[I,J] + A[I,K]*B[K,J] // S1
do I = 1 .. N
 do J = 1 .. N
  do K = 1 .. N
    D[K,J] = D[I,J]*C[I,K] - B[K,J] // S2
```

Benchmark VI

```
for i=1 to N
  c[i]=a[i]-b[i];                    //S1
  r[i]=d[i-9]/e[i];                  //S2
  k[i]=a[i+1]*b[i-1]+p[i]-q[i];      //S3
for i=1 to N
  n[i]=d[i]*c[i]+e[i];               //S4
  f[i]=o[i]+d[i]/e[i];               //S5
  g[i]=r[i]/m[i+1];                  //S6
  h[i]=n[i]*m[i+1]+e[5+i]-c[i+10];   //S7
```