Addressing Optimization for Loop Execution Targeting DSP with Auto-Increment/Decrement Architecture*

Wei-Kai Cheng Youn-Long Lin
Computer Science Department, Tsing Hua University
Hsin-Chu, Taiwan 30043, R.O.C.

Abstract

Since most DSP applications access large amount of data stored in the memory, a DSP code generator must minimize the addressing overhead. In this paper, we propose a method for addressing optimization in loop execution targeted toward DSP processors with auto-increment/decrement feature in their address generation unit. Our optimization methods include a multi-phase data ordering *and a graph-based* address register allocation. *The proposed approaches have been evaluated using a set of core algorithms targeted towards the TI TMS320C40 DSP processor. Experimental results show that our system is indeed more effective compared to a commercial optimizing DSP compiler.*

1 Introduction

Most DSP processors' address generation units (AGU) have more than one address registers with auto-increment/decrement capability. Addressing optimization can be achieved via data ordering and address register allocation. Data ordering determines the order of data stored in the memory. Address register allocation assigns an address register to each data access for address generation. The goal of these optimizations is to maximize the usage of auto-increment/decrement and hence, reduce the number of address loading instructions.

Figure 1 illustrates the data ordering optimization when only one address register is available [4]. In this example, we have a sequence of accesses to four variables, $x[0]$, $x[1]$, $x[2]$, and $x[3]$. Figures 1(a) and (b) depict two different memory allocations and the corresponding sequences of memory-related instructions, respectively. Suppose the auto-increment/decrement range of the address register (AR) is 1. To access a data location too far away from the address currently pointed to by the address register (i.e., offset > 1), an address-modifying instruction is needed to load the new address into the address register. This increases the total execution time. Figures 1(b) shows that with proper data ordering in the memory, the number of inserted address-modifying instructions is reduced from 10 to 5.

In this paper, we deal with the addressing optimization problem for DSP processors with different auto-increment/decrement addressing ranges when data is accessed by different instructions. In addition to all constraints on data ordering and address register allocation for loop constructs, we also take into account the constraint on memory allocation for array constructs.

*Supported in part by a grant from the National Science Council of R.O.C. under contract no. NSC87-2215-E-007-017

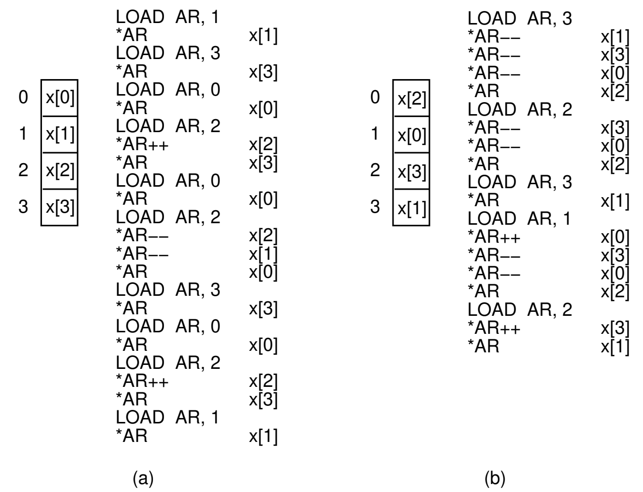

Figure 1: An example of addressing optimization: (a) code with unoptimized data ordering (10 load instructions), (b) code with data reordering (5 load instructions).

The rest of this paper is organized as follows. In section 2, we survey some previous research. Section 3 introduces constraints on data ordering and address register allocation for loop execution, and describes the architectural modeling. A graph model and detailed algorithms are described in section 4. In section 5, some experimental results are presented. Finally, in section 6 we draw conclusions and point to possible directions for future research.

2 Previous Work

Most researchers [3, 4, 6, 8] model the addressing optimization problem as finding paths in an access graph. Suppose the auto-increment/decrement range is 1. They represent an access sequence as an access graph, partition the graph, find a maximal weight path for each partitioned subgraph, and allocate an address register for each set of data corresponding to each path found. Figure 2(a) shows the access graph for the access sequence of Figure 1. Each node in the graph denotes a data location, and an edge weight denotes the number of successive accesses to the two data corresponding to the end-nodes of the edge. Figure 2(b) shows the maximal weighed path of Figure 2(a). An optimal data ordering of $x[2]$, $x[0]$, $x[3]$, $x[1]$ (or $x[1]$, $x[3]$, $x[0]$, $x[2]$) is implied.

Beginning with an empty path, Liao et al. [6] proposed iteratively adding a maximal weighted edge to the path under construction if it causes no cycle nor tree.

Figure 2: An example to illustrate the data ordering optimization: (a) the access graph of Figure 1(a), (b) the maximal weight path.

Arbitrary selection is used to break a tie. Leupers and Marwedel [4] proposed an improved cost function for edge selection. In contrast to starting with an empty path, Iimuro et al. [3] proposed removing one edge at a time from the graph until no cycle nor tree exists.

Sudarsanam et al. [8] extended the problem to that of greater auto-increment/decrement range (i.e. > 1). If the range is l, the path weight includes all the weight of edges whose end-nodes have distance no more than l along the path. For the example of Figure 2(a), the maximal weight path is also as that shown in Figure 2(b) when the range of auto-increment/decrement is 2. In addition to the weight of edges in this path, the path weight includes the weight of edges connecting $x[2]$ and $x[3]$, and $x[0]$ and $x[1]$. Therefore, the path weight is 3+4+2+2+1=12.

None of the above described previous research considers the characteristics of loop execution. Cheng and Lin [2] combined the retiming technique and the pattern matching technique to reduce the number of address-modifying instructions across loop iterations. Liem et al. [7] combined address variables with the same address offset among loop iterations. Araujo et al. [1] allocated address registers for array accesses within loop constructs by means of graph coloring. Leupers et al. [5] utilized both matching and path-based methods in a branch-and-bound procedure to optimize array index computation.

Although these researches [1, 2, 5, 7] take into consideration the characteristics of loop execution, they do not optimize data ordering. In this paper, we propose approaches for data ordering and address register allocation for loop execution targeted towards DSP processors.

3 Addressing Constraints and Architectural Modeling

In this section, we describe two types of constraints on data ordering and address register allocation, respectively, during addressing optimization for loop execution. We also describe architectural modeling for these constraints.

3.1 Data Ordering Constraint

Figure 3 illustrates the data ordering constraint. Figure 3(a) shows a program segment of data accesses. Figure 3(b) shows the access sequence to array data for each iteration. In Figure 3(c), we show the best data ordering in the memory for each iteration when the auto-increment/decrement range is 1. The best ordering for the first iteration is $x[1]$, $x[3]$, and $x[2]$; while for the second iteration it is $x[2]$, $x[4]$, and $x[3]$. A conflict exists between $x[2]$ and $x[3]$. Also there exists a conflict between $x[3]$ and $x[4]$ for iterations 2 and 3, and so on. Unless there exists no conflict among all iterations, data ordering cannot be applied.

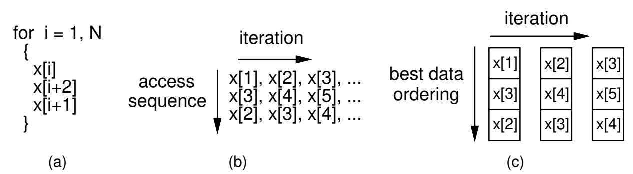

Figure 3: An example of constraint on data ordering optimization: (a) a program segment for data accesses, (b) the access sequence in each iteration, (c) the best data ordering in the memory for each iteration.

3.2 Address Register Allocation Constraint

Figure 4 illustrates the constraint on address register allocation. Figure 4(a) shows a program segment for data accesses. Figure 4(b) shows the access sequence to array data stored in the memory for each iteration. The first two data accesses have a stride of 1 across loop iterations, and the last two have a stride of 2. Because of different strides, if we have only one address register, additional address-modifying instructions are needed as shown in Figure 4(c). Therefore, it is preferred that array data accesses with different strides are addressed using different address registers, as shown in Figure 4(d).

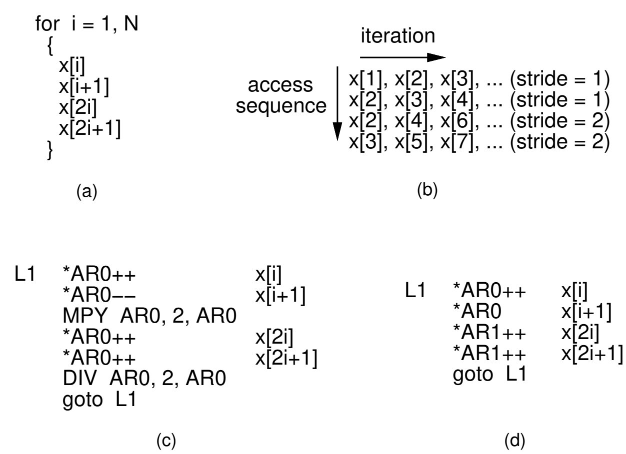

Figure 4: An example of constraint on address register allocation: (a) a program segment for data accesses, (b) the access sequence in each iteration, (c) addressing of data accesses using one address register, (d) addressing of data accesses using two address registers.

3.3 Architectural Modeling

We assume all data of an array are stored in consecutive memory locations and each instruction has a specific auto-increment/decrement range when addressing its operands.

Under the first assumption, data ordering can only be applied to data within an array. It is preferred that data accesses to different arrays are addressed using different address registers. Furthermore, because of the address register allocation constraint, data accesses to an array are also preferred to be addressed using multiple address registers in case of different address strides across loop iterations. Therefore, we further divide an access sequence into multiple data lists as an initial address register allocation.

(a)

```
for  i = 1, N ; step = 2
 {
  C[i]=B[i+1]*(A[i]+xi)
  C[2i]=A[2i]*(B[i+2]-tm)
  C[i+1]=B[i]*(A[i+1]+xi)
  C[2i+1]=A[2i+1]*(B[i+3]-tm)
  A[i]=le*C[i]+wn*C[2i]
  A[i+1]=le*C[i+1]+wn*C[2i+1]
  D[i]=C[i]-wn
  D[i+1]=C[i+1]-wn
 }
```

(b)

```
        L20 :
 1      ADDF      R9,*A[i],R0
 2      MPYF      R0,*B[i+1],R0
 3      STF       R0,*C[i]
 4      SUBF      R11,*B[i+2],R1
 5      MPYF      R1,*A[2i],R1
 6      STF       R1,*C[2i]
 7      ADDF      R9,*A[i+1],R1
 8      MPYF      R1,*B[i],R1
 9      STF       R1,*C[i+1]
10      SUBF      R11,*B[i+3],R1
11      MPYF      R1,*A[2i+1],R1
12      STF       R1,*C[2i+1]
13      MPYF      R4,*C[i],R1
14      MPYF      R5,*C[2i],R0
        ADDF      R0,R1,R1
15      STF       R1,*A[i]
16      MPYF      R5,*C[2i+1],R0
17      MPYF      R4,*C[i+1],R1
        ADDF      R1,R0,R0
18      STF       R0,*A[i+1]
19      SUBF      R5,*C[i],R0
20      STF       R0,*D[i]
21      SUBF      R5,*C[i+1],R1
22      STF       R1,*D[i+1]
        ADDI      RS,BK
        CMPI      R2,BK
        BLT       L20
```

Figure 5: An example for data list construction: (a) source code, (b) the assembly code segment.

Figures 5 and 6 show an example of the data list construction. Figures 5(a) and (b) show the source code and the assembly code, respectively. There are 22 data accesses in the loop body. Suppose that for instructions **MPYF**, **ADDF**, and **SUBF**, the auto-increment/decrement ranges are all 1 when addressing operands stored in the memory, and for instruction **STF**, the auto-increment/decrement range is 2. Figure 6(a) shows the access sequence and the auto-increment/decrement range of each data access. There are 4 arrays accessed in the sequence. For accesses to arrays **A** and **C**, there are two strides (2 and 4). While for accesses to arrays **B** and **D**, all strides are 2. Therefore, the access sequence is divided into 6 data lists (two each for **A** and **C**, and one each for **B** and **D**), as shown in Figure 6(b).

4 Addressing Optimization

Because a data may be accessed in more than one data list, the addressing optimization problem cannot be modeled as finding maximal weight paths in the access graphs for all data lists simultaneously. Therefore, we optimize the data ordering and the address register allocation separately. Initially, an address register is allocated for each data list. Data ordering is optimized for each data list under the data ordering constraint.

Then in the second phase, we further reduce the number of inserted address-modifying instructions by optimizing the address register allocation. If the number of data lists is greater than the number of address registers, we perform data list merging until the number of lists is equal to the number of address registers. On the other hand, if the number of data lists is smaller than the number of address registers, data lists are split. Detailed algorithms and illustrations of these optimizations are described in the following subsections.

(a)

```
for  i = 1, N ; step = 2
 {
 1   A[i]        (1)
 2   B[i+1]      (1)
 3   C[i]        (2)
 4   B[i+2]      (1)
 5   A[2i]       (1)
 6   C[2i]       (2)
 7   A[i+1]      (1)
 8   B[i]        (1)
 9   C[i+1]      (2)
10   B[i+3]      (1)
11   A[2i+1]     (1)
12   C[2i+1]     (2)
13   C[i]        (1)
14   C[2i]       (1)
15   A[i]        (2)
16   C[2i+1]     (1)
17   C[i+1]      (1)
18   A[i+1]      (2)
19   C[i]        (1)
20   D[i]        (2)
21   C[i+1]      (1)
22   D[i+1]      (2)
 }
              range
```

(b)

```
L1 : stride = 2           L4 : stride = 2
 1   A[i]      (1)         3   C[i]      (2)
 7   A[i+1]    (1)         9   C[i+1]    (2)
15   A[i]      (2)        13   C[i]      (1)
18   A[i+1]    (2)        17   C[i+1]    (1)
                          19   C[i]      (1)
                          21   C[i+1]    (1)

L2 : stride = 4           L5 : stride = 4
 5   A[2i]     (1)         6   C[2i]     (2)
11   A[2i+1]   (1)        12   C[2i+1]   (2)
                          14   C[2i]     (1)
                          16   C[2i+1]   (1)

L3 : stride = 2           L6 : stride = 2
 2   B[i+1]    (1)        20   D[i]      (2)
 4   B[i+2]    (1)        22   D[i+1]    (2)
 8   B[i]      (1)
10   B[i+3]    (1)
```

Figure 6: The constructed data lists of the example in Figure 5: (a) the access sequence, (b) the constructed data lists.

4.1 Data Ordering

We optimize the data ordering of a data list (L_i) in n passes where n is the number of auto-increment/decrement ranges. The access graph G is split into n graphs ($G = G_1 + G_2 + \ldots + G_n$). Each split graph G_k denotes an access graph for all the data accesses with auto-increment/decrement range l_k in the data list ($l_1 < l_2 < \ldots < l_n$). We apply the algorithm proposed by Sudarsanam et al. [8] to each split access graph for optimizing the data ordering with an auto-increment/decrement range parameter l_k (l_k-SOA). Because the optimization has a more constrained solution space for a smaller auto-increment/decrement range, we start with the one having the smallest range. The result obtained from a split access graph G_{k-1} is then used as the partial solution for the next access graph G_k.

Figure 7 shows an example. Figure 7(a) shows a sequence of accesses and auto-increment/decrement ranges. Figure 7(b) shows the corresponding access graph G. Because there are two different ranges, the graph is split into two graphs, G_1 and G_2, as depicted in Figures 7(c) and 7(d), respectively. There are three successive accesses to $x[0]$ and $x[2]$. For the 6th and the 11th accesses, the range is 1, while for the 3rd access, the range is 2. Therefore, the edge weights between nodes $x[0]$ and $x[2]$ of Figure 7(c) and that of Figure 7(d) are 2 and 1, respectively. The weights of other edges are calculated in the same way. Figure 7(e) shows the data ordering result of Figure 7(c). There exists a non-zero-weight, off-path edge between $x[0]$ and $x[3]$. This ordering is used as the partial solution for the ordering of access graph G_2, as depicted in Figure 7(f). Because of the partial ordering, $x[1]$ can only be ordered in adjacent to either $x[0]$ or $x[3]$, but not $x[2]$. The final result is shown in Figure 7(g). There are two off-path edges, each weight 1, implying the need of inserting two address-modifying instructions.

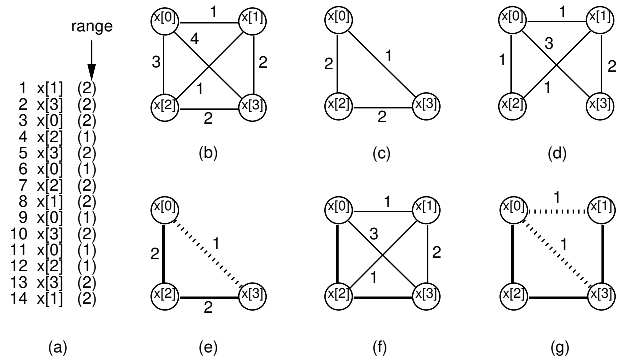

Figure 7: An example of data ordering optimization: (a) the access sequence, (b) the access graph G, (c) the split access graph G_1, (d) the split access graph G_2, (e) the data ordering for G_1, (f) the access graph G_2 with partial data ordering obtained from G_1, (g) the final data ordering.

4.2 Data List Merging

Figure 8 illustrates the insertion of address-modifying instructions resulting from data list merging. Figure 8(a) shows the data ordering in the memory. Figure 8(b) shows the access sequence. Figure 8(c) shows the constructed data lists. Figures 8(d) and (e) show the merging of data lists $L1$ and $L2$, and $L1$ and $L3$, respectively. In Figure 8(d), there is one successive data access between $L1$ and $L2$. In Figure 8(e), there are four successive data accesses between $L1$ and $L3$. Therefore, the data list merging of Figure 8(d) is more preferable.

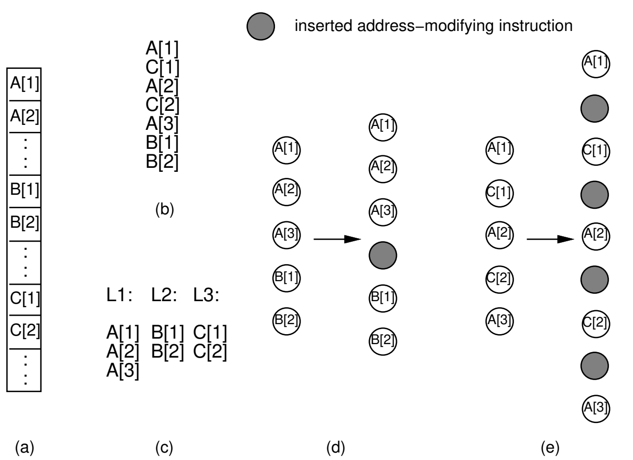

Figure 8: An example of data list merging: (a) the data ordering in the memory, (b) the access sequence, (c) the data lists, (d) merging of data lists $L1$ and $L2$, (e) merging of data lists $L1$ and $L3$.

Given the number of available address registers, we iteratively select and merge two data lists that require the fewest number of inserted instructions until the number of data lists remained is equal to the number of address registers.

4.3 Data List Splitting

In data list splitting, each data list is associated with the maximal number of removable address-modifying instructions when it is split into two lists. We split the data lists one at a time starting with the one with the largest number of removable instructions until the number of data lists is equal to the number of address registers available.

We propose a model, called the block access graph for splitting a data list. The data list splitting problem is transformed into the two-way graph partition problem. The method of constructing the block access graph and the graph partition algorithm are described in the following subsections.

4.3.1 Block Access Graph Construction

Figure 9 illustrates the construction of the block access graph. Figure 9(a) shows the data list. Suppose that the auto-increment/decrement range is 1 for all the data accesses. For each data access, we find the nearest data access before and after the data access, respectively, such that the address offsets between the data access and the data accesses found do not exceed the auto-increment/decrement range, and connect the data access with the data accesses found with edges, called access edges. Each edge in Figure 9(a) is an access edge. For successive data accesses that the address offset between them exceeds the auto-increment/decrement range, access edges are constructed as shown in Figure 9(b) with dashed lines. When a data list is split, no address-modifying instruction is required if two data accesses connected by an access edge become successive data accesses in the same split data list.

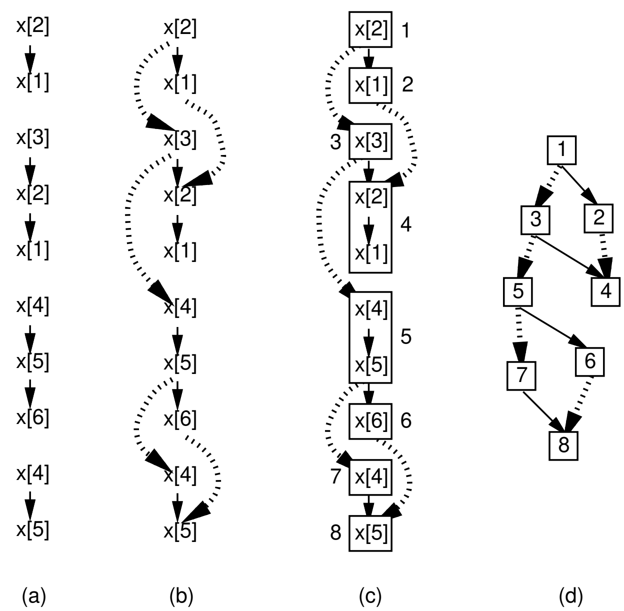

Figure 9: An example to illustrate the construction of block access graph: (a) the data list, (b) the construction of access edges, (c) the construction of access blocks, (d) the constructed block access graph.

In the second step, we construct access blocks from the data list. Figure 9(c) shows the access blocks constructed from Figure 9(b). An access block means that all the data accesses in it should be partitioned to the same split data list according to the access edges constructed. For the example in access block 4, $x[2]$ is connected to $x[1]$ only. While $x[2]$ in access block 1 is connected to both access blocks 2 and 3. Therefore, access blocks 1 and 2 cannot be grouped into an access block.

Finally, we construct the block access graph from the access blocks and the access edges. Each node in the graph denotes an access block constructed before. An edge in the graph denotes an access edge that connects two access blocks. Figure 9(d) shows the block access graph constructed from Figure 9(c).

4.3.2 Block Access Graph Partition

We can map the data list splitting problem into a two-way partition problem on the block access graph. For the example in Figure 9(d), the optimum solution of data list splitting is to partition all the access blocks into the sets {3, 5, 6, 7, 8}, and {1, 2, 4}. Only access block 6 and access block 7 are not connected by an access edge, and address-modifying instruction is required.

We optimize the two-way graph partition problem in two steps: (1) minimal number of disjoint path construction and (2) merging the disjoint paths into two sets with smallest number of address-modifying instruction inserted. For the example in Figure 9(d), the minimum number of disjoint paths is 3: path 3-5-7-8, path 1-2-4, and path 6. Then by merging path 3-5-7-8 and path 6, we obtain the optimum graph partition.

During disjoint path construction, we select one path at a time as a constructed disjoint path from all the possible paths in the block access graph. When a path is selected, all the other paths that have common access blocks with the selected path are modified by removing the common access blocks. A path after modification is deleted if it becomes a partial path of the other paths. This process is repeated until all the access blocks are covered in the selected paths.

We propose two criteria for path selection: (1) the number of access blocks covered in more than one paths and (2) the total number of access edges connected to and from the access blocks described in the first criterion. The path with the smallest number of access blocks described in the first criterion is selected first. When the first criterion cannot determine a unique path, the path with the smallest number of connected access edges described in the second criterion is selected. Otherwise, a path is selected randomly.

Figure 10 illustrates the disjoint path construction algorithm. Figure 10(a) shows all the possible paths in Figure 9(d). Access blocks 1, 3, 4, 5, and 8 are covered in more than one paths. The path 1-2-4 has the smallest number of access blocks covered in more than one paths, which is selected first. Other paths are modified by removing access blocks 1 and 4. Because access block 3 is the only remainder for the path 1-3-4 after the modification and it also becomes a partial path of the other two paths remaining, it is deleted. Figure 10(b) shows all the paths after path 1-2-4 is selected. Because we cannot determine a unique one according to the two criteria proposed above for the two remaining paths, a path is selected randomly. Figure 10(c) shows the final result of the disjoint path construction.

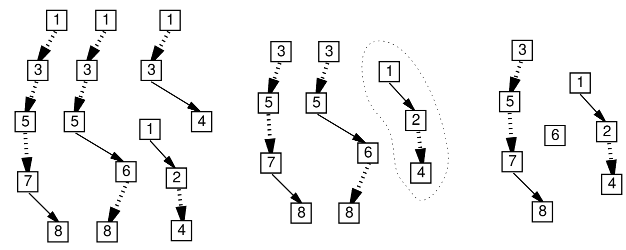

Figure 10: An example of disjoint path construction: (a) the possible paths in Figure 9(d), (b) the paths after path 1-2-4 is selected, (c) the constructed disjoint paths.

During path merging, each time we merge two paths that require the minimum number of address-modifying instructions. This process is repeated until the number of paths remaining is equal to 2. For the example in Figure 10(c), merging of the two paths 3-5-7-8 and 1-2-4 inserts two address-modifying instructions, one between access blocks 2 and 3 and another between blocks 4 and 5. While the merging for the other two cases requires only one inserted address-modifying instruction between access blocks 4 and 6, and 6 and 7, respectively. Therefore, the minimal cost of merging is 1. Since three inserted address-modifying instructions were needed originally, as shown in Figure 9(a), the gain of splitting this data list is 2.

5 Experimental Results

We have implemented the proposed approaches in a software system using the C programming language. Our system has been tested with a set of benchmark programs targeted towards the TMS320C40 DSP processor. There are 8 address registers and 2 index registers in its address generation units, and 2 auto-increment/decrement ranges in its instruction set. We compared our results with TI's TMS320C40 C compiler (all optimization terms were turned on) in terms of the number of inserted address-modifying instructions and total execution cycles.

Table 1 shows the improvement of our results over the codes generated by the TI compiler in terms of the number of inserted address-modifying instructions. Table 2 shows the improvement in terms of the number of execution cycles. The second and the third rows show the improvement when only data ordering, and address register allocation was applied, respectively. Our algorithm inserted fewer address-modifying instructions than TI's compiler did in all the benchmarks. In the last row, we show the results when data ordering and address register allocation were all optimized by our algorithm. Our algorithm was able to remove all the inserted address-modifying instructions in three benchmarks, and remove most of the inserted instructions for the other four benchmarks.

In general, the optimization of address register allocation is more effective than the optimization of data ordering because data ordering cannot be applied to all the data lists due to the constraint described in section

Table 1: Ratio of the number of inserted address-modifying instructions of the codes generated by our system over those generated by TI's TMS320C40 compiler ($TI.o + TI.a$). $TI.o$: data ordering by TI's compiler, $TI.a$: address register allocation by TI's compiler, $Our.o$: data ordering by our system, $Our.a$: address register allocation by our system.

	fir	**iir**	**dft**	**fft**	**edge**	**dct**	**wiener**	**average**
$TI.o + TI.a$	1	1	1	1	1	1	1	1
$Our.o + TI.a$	0.5	0.714	0.85	0.812	0.644	0.855	0.762	0.591
$TI.o + Our.a$	0	0.286	0.422	0.563	0.378	0.925	0.254	0.524
$Our.o + Our.a$	0	0	0.222	0.233	0.126	0.496	0	0.225

Table 2: Ratio of the number of execution cycles of the codes generated by our system over those generated by TI's TMS320C40 compiler ($TI.o + TI.a$).

	fir	**iir**	**dft**	**fft**	**edge**	**dct**	**wiener**	**average**
$TI.o + TI.a$	1	1	1	1	1	1	1	1
$Our.o + TI.a$	0.909	0.902	0.963	0.932	0.892	0.945	0.968	0.899
$TI.o + Our.a$	0.819	0.754	0.924	0.895	0.826	0.972	0.900	0.884
$Our.o + Our.a$	0.819	0.656	0.905	0.819	0.756	0.808	0.865	0.814

Table 3: Number of total data lists and number of data lists that data ordering is applied.

	fir	**iir**	**dft**	**fft**	**edge**	**dct**	**wiener**
lists	3	3	6	6	4	10	5
ordering	1	1	1	1	2	2	2

3.1. Table 3 shows the number of total data lists and the number of data lists that data ordering is applied for each benchmark. A special case is the dct benchmark in which the optimization of data ordering is more effective than the optimization of address register allocation. This is because that the number of data lists in this benchmark is close to the number of available address registers. Therefore, very little improvement can be achieved for this optimization.

6 Conclusion and Future Work

We have presented approaches for addressing optimization targeted to the DSP processors with auto-increment/decrement features. In data ordering, we propose a multi-passes approach to solve the problem of different auto-increment/decrement ranges in the instruction set. In address register allocation, we propose a new graph model and solve the problem by disjoint path construction and path merging on the graph.

Experimental results show that our approaches are indeed very effective in comparison with TI's TMS320C40 compiler. Unlike previous research which emphasizes the importance of data ordering for straight-line codes, our results show that it is not as effective as the optimization of address register allocation when taking into account all the constraints on loop execution.

A new graph model to capture all the constraints such that data ordering and address register allocation can be optimized simultaneously is a future work of our research. Because of the hierarchical memory structure, extending our approaches by taking into account the memory allocation problem together with the addressing optimization problem to further improve the code efficiency is another area for possible future research.

References

[1] G. Araujo, A. Sudarsanam, and S. Malik, "Instruction Set Design and Optimizations for Address Computation in DSP Architectures," Proc. 9th International Symposium on System Synthesis, pp. 102-107, 1996.

[2] W. K. Cheng, and Y. L. Lin, "A Transformation-Based Approach for Storage Optimizations," Proc. 32nd Design Automation Conference, pp. 158-163, June 1995.

[3] S. Iimuro, N. Sugino, A. Nishihara, and N. Fujii, "Code Optimization Method Utilizing Memory Addressing Operation and its Application to DSP Compiler," Proc. APCCAS'94, pp. 151-156, 1994.

[4] R. Leupers, P. Marwedel, "Algorithms for Address Assignment in DSP Code Generation," Proc. International Conference on Computer Aided Design, pp. 109-112, November 1996.

[5] R. Leupers, A. Basu, and P. Marwedel, "Optimized Array Index Computation in DSP Programs," Proc. Asia and South Pacific Design Automation Conference, February 1998.

[6] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Storage Assignment to Decrease Code Size," ACM Transactions on Programming Languages and Systems, pp. 186-195, May 1996.

[7] C. Liem, P. Paulin, and A. Jerraya, "Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures," Proc. 33rd Design Automation Conference, pp. 597-600, June 1996.

[8] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," Proc. 34th Design Automation Conference, pp. 287-292, June 1997.