Experimental results show 3.6% improvement on the average over a compiler optimization step to generate a better access sequence. Assignment problem finding proper placement of variables in storage is called maximally utilized by careful placement of variables in storage, address arithmetic calculations. Such an addressing mode can improve both the size and performance of the code.

Assignment problem. Liao et al. [3] formulated it as a single address register. They first built an access graph from the expression tree to modify the order of operands of an instruction. We can further optimize the access sequence by transforming not only the expression tree of an instruction but also the schedule of instructions. We formulate the problem as statement scheduling and propose an algorithm that solves the problem.

The above approaches do not attempt to optimize the variable access sequence itself, which can significantly affect the result of storage assignment problem. Rao et al. [5] suggested modifying the variable access sequence using expression tree transformations and formulated it as the least cost access sequence (LCAS) problem and developed heuristic algorithms to solve it. They used algebraic transformations (such as commutativity) on the expression tree to modify the order of operands of an instruction.

2. SIMPLE OFFSET ASSIGNMENT

Address generation unit (AGU) of a processor that supports indirect addressing mode can usually compute the address used by the next instruction in parallel with the currently executing instruction. AGU is comprised of a file of $k$ address registers (ARs), as well as a file of $m$ modify registers (MRs). AR and MR indices are provided by AR and MR pointers respectively, which are the values of either special registers or part of instruction words. According to the decoded instruction, AGU generates an address based upon an AR, which is incremented or decremented by a constant or by the value of an MR. The range of the constant is represented by $r$. Thus $k$, $m$, and $r$ determine the configuration of the AGU.

When a program accesses a series of variables in memory, if the stride of the addresses is greater than the range, AR or MR should be reloaded with an immediate value. This additional instruction causes code size overhead of indirect addressing mode. Offset assignment (OA) is the problem of finding proper memory layout of variables to reduce the occurrence of strides larger than the
range supported by the AGU. The offset assignment problem is
classified according to the AGU parameters \(k\), \(m\), and \(r\) and
represented by \((k, m, r)\)-OA [7].

\((1, 0, 1)\)-OA is offset assignment on a processor with only one
address register, no modify register, and auto-increment and
auto-decrement range of 1. Figure 1 (a) shows an example of \((1, 0,
1)\)-OA, which results in nine instructions. By re-arranging the
variable in memory as shown in Figure 1 (b), we can reduce the
number of instructions down to seven.

Bartley [1] modeled the \((1, 0, 1)\)-OA problem as an undirected
edge-weighted access graph \(G(V,E,W)\), where \(V\) models the set of
variables and \(E\) models the set of transitions between variables.
For each edge \(e=(v_1, v_2)\) in \(E\), the weight \(w(e)\) is the number of
transitions from \(v_1\) to \(v_2\) or vice versa in the access sequence.
Large weight of edge \((v_1, v_2)\) means \(v_1\) and \(v_2\) are frequently
accessed in sequence so the two variables should be placed into
neighboring memory locations because accessing \(v_1\) after or before
\(v_2\) is supported by AGU's auto-increment or auto-decrement. Each offset assignment corresponds to a
Hamiltonian path in \(G\), i.e. a path that traverses all nodes just
once.

It is obvious that an optimum offset assignment corresponds to a
maximum weighted Hamiltonian path in \(G\). This problem is called
maximum weighted path covering (MWPC). Liao et al. [3] showed
that the offset assignment problem is NP-complete even for the
simple case of \((1, 0, 1)\)-OA and presented a heuristic based on the
access graph model to solve it. They generalized it to \((k, 0, 1)\)-OA
by partitioning \(G\) into \(k\) subgraphs each of which is covered by an
AR.

3. STATEMENT SCHEDULING

First we assume an accumulator machine with one accumulator

register and load/store instructions and consider simple offset
assignment problem. Figure 2 (a) shows an application code, the
 corresponding access sequence, the access graph, and the MWPC
solution. Edges that are not covered by the MWPC solution
represent transitions requiring additional instructions. The weight
of such an edge corresponds to the number of additional
instructions. Hence the cost of the MWPC solution is the sum of
the uncovered edges' weights, and the cost of the solution in
Figure 2 (a) is 3.

Observing that the two statements \(c = d + 2\) and \(a = b\) have no
data/control dependency to each other, we can change their order
without affecting the functionality. Figure 2 (b) shows a modified
code and the corresponding access sequence, access graph, and
MWPC solution with cost of 1. This way, proper statement
scheduling can result in lower MWPC cost and this is the
motivation of our work.

We formulate the problem as follows. Given a basic block\(^1\) of an
application code, schedule the statements in such a way that the
cost of the MWPC solution for the corresponding access graph is
minimized. We use the term statement to denote a schedulable
element of compiler's intermediate representation—medium level
intermediate representation (MIR) [6]—for a given application
code. One statement corresponds to one MIR instruction.

\(^1\) Currently, we do not allow branches and perform the optimization within
a basic block.

\(^2\) We assume all local variables reside not in registers but in memory.
Hence instruction scheduling considering register spill is not of our
concern.

---

\(^1\) Currently, we do not allow branches and perform the optimization within
a basic block.

\(^2\) We assume all local variables reside not in registers but in memory.
Hence instruction scheduling considering register spill is not of our
concern.
hereafter. Each node means a statement (or MIR instruction) and each directed edge means dependency between two statements. Figure 3 illustrates the application code in Figure 2 using the dependence DAG.

4. ALGORITHMS

We need to find a statement schedule that gives the minimum cost. However, to compute the cost for a given schedule, we need to solve the MWPC problem which itself is NP-complete. We devise a heuristics based on the observation that the access graph of Figure 2 (b) has fewer edges than that of Figure 2 (a). When an access graph becomes sparser, maximum weighted path covering tends to result in less cost because the number of edges to be covered are reduced and the edge weights tend to be concentrated on some small set of edges. So our algorithm aims to generate the sparsest access graph.

4.1 List Scheduling

The proposed list scheduling algorithm constructs an access graph by selecting a statement to be scheduled from the dependence DAG and updating the access graph with the new transitions. The update may add new edges, add new vertices and/or increase edge weights. It implements a greedy heuristic that selects a statement that adds least new edges at that schedule step.

Figure 4 (a) shows a statement schedule of the example in Figure 3. Let’s assume that statement (1), (2), and (3) are already scheduled and (4) and (5) are not scheduled yet. If we schedule statement (4) first, two new edges will be added to the access graph as shown in Figure 4 (b). Then statement (5) is scheduled and the cost is 3 as shown in Figure 2 (a). If we schedule statement (5) first, one new edge will be added to the access graph as shown in Figure 4 (c). Then statement (4) is scheduled and the cost is 1 as shown in Figure 2 (b). This is the case where list scheduling leads to an optimum solution.

4.2 Exhaustive Search

Exhaustive search method examines all possible statement schedules and finds an optimal schedule, which generates sparsest access graph. Branch pruning can be used to accelerate the exhaustive search. At each schedule step, it selects a statement for the next schedule, updates the edge count, and compares it to the optimal cost found up to that time.

4.3 Hybrid Algorithm

List scheduling executes fast but as is usual for a greedy algorithm, can lead to a local optimum. And it is not easy to set good criteria for tie break. On the contrary, exhaustive search guarantees an optimal solution but runs in time exponential to the size of the problem. Even the pruning method does not guarantee to improve the execution time. More aggressive pruning is needed to improve the execution time. The hybrid algorithm confines the exhaustive search to the successors of the statement that is scheduled most recently. Other statements are excluded because they tend to generate new edges in the access graph. For example, consider the dependence DAGs shown in Figure 5. Figure 5 (a) illustrates that the list scheduling method selects one statement from the candidates for the next schedule. Figure 5 (b) illustrates that the exhaustive search considers all candidate statements for the next schedule. However, the hybrid algorithm confines the search to the reduced number of candidates as shown in Figure 5 (c).

5. EXPERIMENTAL RESULTS

We implemented the three algorithms on SPAM compiler middle- and back-end targeting Texas Instruments’ TMS320C25 DSP. The
Table 1 shows the code size reduction in number of words. SOA means the size of code obtained by MWPC but without statement scheduling. The gain of SS gives the code size reduction obtained by the hybrid method with respect to the original code size. It shows the average gain of 3.6%. We could not obtain the result of the exhaustive search on biquad_N_sections due to the enormous amount of running time but the hybrid method found a solution. Execution times of the proposed three algorithms are shown in table 2.

To compare the effect of statement scheduling with that of expression tree transformation, we quoted the gain from [5]. Sometimes expression tree transformation shows better result. But the two approaches are not totally exclusive. We expect additional gain by extending our approach from MIR- to LIR-based scheduling to subsume the effect of expression tree transformation and we set aside it for future work.

Table 1: Code Size Reduction

<table>
<thead>
<tr>
<th></th>
<th>SOA list schedule</th>
<th>exhaustive</th>
<th>hybrid</th>
<th>% gain of SS</th>
<th>% gain of ETT</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex_multiply</td>
<td>34</td>
<td>32</td>
<td>32</td>
<td>5.882</td>
<td>2.3</td>
</tr>
<tr>
<td>convolution</td>
<td>92</td>
<td>90</td>
<td>86</td>
<td>5.435</td>
<td>5.81</td>
</tr>
<tr>
<td>dot_product</td>
<td>75</td>
<td>71</td>
<td>71</td>
<td>5.333</td>
<td>0</td>
</tr>
<tr>
<td>fir</td>
<td>138</td>
<td>134</td>
<td>129</td>
<td>4.348</td>
<td>7.03</td>
</tr>
<tr>
<td>biquad_N_sections</td>
<td>222</td>
<td>218</td>
<td>N/A</td>
<td>3.604</td>
<td>N/A</td>
</tr>
<tr>
<td>matrix2</td>
<td>287</td>
<td>278</td>
<td>275</td>
<td>3.484</td>
<td>0.74</td>
</tr>
<tr>
<td>matrix1x3</td>
<td>71</td>
<td>69</td>
<td>69</td>
<td>2.817</td>
<td>N/A</td>
</tr>
<tr>
<td>fir2dim</td>
<td>365</td>
<td>361</td>
<td>361</td>
<td>1.096</td>
<td>2.4</td>
</tr>
<tr>
<td>biquad_one_section</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>0</td>
<td>1.74</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td></td>
<td>3.6</td>
<td>2.9</td>
</tr>
</tbody>
</table>

SS : statement scheduling with hybrid method
ETT : expression tree transformation [5]

Table 2: Execution Time (seconds)

<table>
<thead>
<tr>
<th></th>
<th>list scheduled</th>
<th>exhaustive</th>
<th>hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex_multiply</td>
<td>0</td>
<td>0.74</td>
<td>0.67</td>
</tr>
<tr>
<td>convolution</td>
<td>0</td>
<td>0.61</td>
<td>0.61</td>
</tr>
<tr>
<td>dot_product</td>
<td>0</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>fir</td>
<td>0</td>
<td>28.91</td>
<td>8.71</td>
</tr>
<tr>
<td>biquad_N_sections</td>
<td>0.01</td>
<td>N/A</td>
<td>171.1</td>
</tr>
<tr>
<td>matrix2</td>
<td>0</td>
<td>6.66</td>
<td>0.83</td>
</tr>
<tr>
<td>matrix1x3</td>
<td>0</td>
<td>0.86</td>
<td>0.82</td>
</tr>
<tr>
<td>fir2dim</td>
<td>0.01</td>
<td>3.06</td>
<td>2.74</td>
</tr>
<tr>
<td>biquad_one_section</td>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Figure 5. Pruning the search space for statement scheduling.

Figure 6. Overall flow of the optimizing compiler.

overall flow of the optimizing compiler is shown in Figure 6.
We calculated the effect of code size reduction on the performance for the case of complex_multiply and dot_product, which resulted in 5.88% and 4.94% improvement in cycle count respectively.

6. CONCLUSION
We showed that statement scheduling can further improve the result of simple offset assignment. Among the three scheduling algorithms proposed in this paper, the hybrid method results in the cost generally lower than the list scheduling method and runs faster than the exhaustive search.

Generalization of statement scheduling to solve GOA does not seem to be difficult. Using variable partitioning method, we can also partition dependence DAG into many subgraphs and execute the proposed scheduling algorithm assuming one address register for each subgraph.

The proposed scheduling is done at medium-level intermediate representation (MIR) in SPAM compiler. But if we target low-level intermediate representation, we may be able to obtain further improvement.

REFERENCES