An Efficient Algorithm for Microword Length Minimization

Ruchir Puri and Jun Gu
Dept. of Electrical Engineering, Univ. of Calgary
Calgary, Canada T2N 1N4
gu@enl.ucalgary.ca

Abstract

The problem of microword length minimization is crucial to the synthesis of microprogrammed controllers in digital systems. The problem is NP-hard. Although various enumerative and heuristics methods have been developed in the past, they usually do not provide fast and efficient solutions to a large size problem. In this paper, we formulate the problem into a graph partitioning problem. We employ a local search approach to further reduce the microword length. This has resulted in an efficient algorithm which outperforms any of the existing techniques available to solve the problem. We have tested our algorithm with practical microcodes and with large size examples generated from random graphs. We will compare our experimental results with other existing methods.

1 Introduction

The optimization of microword length is vital to microprogrammed controller design in a digital system. A microprogram can be implemented into control memory by assigning the microoperations to memory word bits. The control memory size depends on the length of microword it stores. Thus, any reduction in the microword length reduces the control memory's silicon area and increases its real-time performance in a microprocessor.

The problem of microword length minimization is NP-hard [8]. An optimal solution obtained by various enumerative procedures [1, 2, 5, 9] cannot be justified at the expense of exponential time complexity. One way to solve the problem is to employ efficient heuristics which decrease the computing time and provide a near-optimal solution [3, 4, 7, 10] Nagle et al. [7] used an autonomous weight algorithm which takes an inadvertently long time as the parallelism in the microprogram decreases. Researchers have used simulated annealing approach [3]. So far there is little progress. Recently Hong et al. [4] proposed a probabilistic algorithm with an O(n^2\log n) heuristics that uses Kernighan-Lin's graph partitioning method [6]. Their approach, although improving over Nagle et al.'s algorithm, requires many trials before it can find the best solution.

In this paper, we present an efficient heuristic algorithm for microword length minimization problem.

2 Problem Formulation

The microcode of a microprogrammed controller consists of a sequence of microinstructions \( \mu_1, \mu_2, \ldots, \mu_M \), where each microinstruction is a set of microoperations \( O_1, O_2, \ldots, O_t \). The microoperations in each microinstruction are conflict free in resource accessing. Thus, two microoperations \( O_i \) and \( O_j \), belonging to a single instruction, are executed concurrently. They should not be encoded into a single group or field. Such pairs of microoperations are incompatible. On the other hand, two microoperations are grouped into a single field, if they do not appear together in any microinstruction. Such a pair of microoperations are compatible. A compatibility class is a set of microoperations, that are pairwise compatible. That is, they are never executed concurrently and thus can be encoded together. Let \( \{C_1, C_2, \ldots, C_k\} \) be a set of compatibility classes for a given set of microinstructions such that each microoperation appears in only one of the compatibility classes. Let \( |C_i| \) be the number of microoperations in a compatibility class \( C_i \). We can encode the compatibility class \( C_i \) into a single field with \( \lceil \log_2(|C_i| + 1) \rceil \) bits (where one is added to account for the NO-OP condition). The microword length \( L \) equals the total number of bits required to encode the compatibility classes \( \{C_1, C_2, \ldots, C_k\} \). That is: \( L = \sum_{i=1}^{k} \lceil \log_2(|C_i| + 1) \rceil \). Microword length optimization problem is to determine a set of compatibility classes...
classes \{C_1, C_2, \ldots, C_k\} such that \( L \) is minimum.

We can formulate the problem in a compatibility graph. A microoperation \( O_i \) is represented by node \( i \) in the graph. Two nodes \( i \) and \( j \) are connected by an undirected arc, if their corresponding microoperations \( O_i \) and \( O_j \) are compatible. A compatibility class is a complete subgraph such that all its nodes are connected to each other, i.e., they are pairwise compatible. In such a graph model, the microword length optimization problem is equivalent to finding a partitioning of the compatibility graph into \( k \) disjoint complete subgraphs \( \{C_1, C_2, \ldots, C_k\} \) such that the cost function \( L = \sum_{i=1}^{n} \log_2(|C_i| + 1) \) is minimum.

Figure 1 gives a simple microcode example. The microoperations \( \{O_0, O_1, \ldots, O_9\} \) are mapped as nodes \( \{0, 1, \ldots, 9\} \) in the compatibility graph. Microoperations \( O_0, O_3, O_5 \) and \( O_8 \) do not appear together in any of the microinstructions \( \mu_0, \ldots, \mu_3 \), so they are pairwise compatible. Hence, \( \{0, 2, 3, 6\} \) forms a compatibility class in the compatibility graph (as highlighted in Figure 1).

3 An Efficient Heuristic Algorithm

Most of the techniques \([1, 2, 4, 5, 9]\) proposed to solve the microword length minimization problem build a search space by deriving all the maximal complete subgraphs (i.e., maximal compatibles) in the compatibility graph. Hong et al.'s approach partitions the compatibility graph by recursively extracting a maximal clique and removes it from the graph, until no nodes are left in the graph. We observed that this approach, which is quite time-consuming for graphs with higher densities, does not yield a good initial disjoint partitioning. We now describe our algorithm which consists of two parts: an algorithm for initial graph partitioning and a local search algorithm for cost minimization.\(^1\)

3.1 A Fast Algorithm for Initial Graph Partitioning

A maximal clique algorithm that partitions a graph into the minimum number of groups is not the best way to solve this problem, since our goal is to minimize cost \( L \). Thus, the absolute maximality of the clique sizes derived from the graph will not contribute to a good engineering solution. Besides it is very costly to generate the maximal cliques (NP-complete). We describe a fast algorithm for an initial partitioning of the compatibility graph, where the graph partitions obtained may not be maximal. The pseudocode of the initial graph partitioning algorithm is shown in Figure 2.

The algorithm initially selects a seed node and deletes all the nodes from vertex set \( V \) that are not connected to the seed node. Then another seed node is selected from the remaining nodes in the vertex set and the same deletion procedure is performed. This step is repeated until all the nodes remaining in the vertex set are connected to each other. This is called growing a clique from a "seed." A clique is grown from each node of the graph \( G(V, E) \) and the one having

\(^1\)A nontrivial case study of local search is given in the Appendix.
maximum number of nodes forms the first compatibility class \( C_1 \). The nodes belonging to \( C_1 \) and their corresponding arcs are then removed from the compatibility graph. This guarantees that the compatibility classes are disjoint. The process of growing the cliques, deriving the compatibility classes and removing them from the compatibility graph is repeated until the vertex set of the graph is empty. This guarantees that the set of compatibility classes \( \{C_1, C_2, \ldots, C_n\} \) covers the compatibility graph.

Applying this procedure to the microcode example shown in Figure 1, we get four compatibility classes: \( C_1 = \{0,2,3,6\} \), \( C_2 = \{1,4,8\} \), \( C_3 = \{5,7\} \), and \( C_4 = \{9\} \). Since the number of bits required to encode a compatibility class \( C_i \) having \( |C_i| \) nodes is \( \lceil \log_2(|C_i|+1) \rceil \), the number of bits required to encode \( C_1, C_2, C_3 \) and \( C_4 \) are 3, 2, 2, and 1, respectively. Thus, the initial cost \( L \) is: \( L = 3 + 2 + 2 + 1 = 8 \). This yields an initial value of cost function \( L \), which can be further reduced using a local search algorithm described in the following section.

### 3.2 An Efficient Local Search Algorithm for Cost Minimization

The initial partitioning generated four compatibility classes for the compatibility graph shown in Figure 1. Compatibility class \( C_1 \) has 4 nodes. Compatibility classes \( C_2 \), \( C_3 \) and \( C_4 \) have 3 nodes, 2 nodes, and 1 node, respectively. If we can move a node from \( C_1 \) to \( C_2 \), then the cost of encoding \( C_1 \) will be reduced by one, whereas that of encoding \( C_2 \) will remain the same. The movement of a node will reduce the cost \( L \) from 8 to 7. When moving a node from \( C_1 \) to \( C_2 \), we must ensure that the node under consideration is compatible with all the nodes in \( C_2 \). Since node 6 in \( C_1 \) is compatible with all the nodes in \( C_2 \), it is qualified and can be moved to \( C_2 \).

In the following discussion, let \( \text{overflow}(C_i) \) be the minimum number of nodes which must be removed from \( C_i \) in order to reduce its encoding length by one. That is, \( \text{overflow}(C_i) = \left( |C_i| + 1 \right) - 2\lceil \log_2(|C_i|+1) \rceil \). Let \( \text{underflow}(C_i) \) be the maximum number of nodes which may be added to \( C_i \) without increasing its encoding length. That is, \( \text{underflow}(C_i) = 2\lceil \log_2(|C_i|+1) \rceil - \left( |C_i| + 1 \right) \).

In the microcode example \( C_1 \) has 4 nodes and should give away one node to reduce its encoding length by one. Thus \( C_1 \) has an \( \text{overflow} \) of one. Compatibility class \( C_2 \) has two nodes and it can accommodate one more node without increasing its encoding length, thus it has an \( \text{underflow} \) of one. A node \( k \) can be moved from compatibility class \( C_i \) to \( C_j \), if the following four constraints are satisfied:

- \( \text{Overflow}(C_i) \geq 1 \) (i.e., \( C_i \) needs to give away some nodes in order to reduce its cost).
- \( \text{Underflow}(C_j) \geq 1 \) (i.e., \( C_j \) can accept some nodes without increasing its cost).
- Node \( k \) in \( C_i \) is compatible with all nodes in \( C_j \).
- \( \sum_{i=1,i\neq j}^{n} \text{Underflow}(C_i) \geq \text{Overflow}(C_i) \). That is, the movement of nodes from \( C_i \) can possibly result in a cost reduction.

The pseudocode of the local search algorithm is given in Figure 3. It swaps the nodes of compatibility classes under the constraints stated above and minimizes cost \( L \). This algorithm, when applied to microcode shown in Figure 1, reduces the cost \( L \) by one. Value 7 of the cost function obtained from our algorithm turns out to be the optimal cost. The final partitioning generated by the procedure is \( \{0,2,3\}, \{1,4,8\}, \{5,6,7\} \) and \( \{9\} \). Thus, the microword for this particular example is programmed as four fields. These fields are: \( \{O_0, O_2, O_3\}, \{O_1, O_4, O_8\}, \{O_5, O_6, O_7\}, \{O_9\} \).
4 Experimental Results

1. Results using Practical Microcodes

The algorithm was implemented in C language. To compare our results with other approaches, we ran our algorithm on the examples used by Hong et al. in [4]. In order to maintain a uniform platform for comparison purposes, the examples were tested on a SUN 3/60 system. The results are given in Table 1.

By analysing these results, one can draw the following conclusions:

- Baer et al.'s branch and bound algorithm [1] takes exponential time to find a solution. Although it guarantees an optimal solution, it is not of much practical use in optimizing even moderately sized microcodes.

- Nagle et al.'s autonomous weight algorithm [7] may yield a solution in lesser time than Baer's approach but suffers from the similar drawback of exponential time complexity. Also, the solutions obtained are far from the optimal and the non-optimality of the solution increases with the increasing number of nodes and edges.

- Hong et al.'s method [4], depends on random initial starting point. To select the best solution, the approach must perform many trials. Thus, it effectively requires much larger time (more trials) to yield a good solution. Also, the time required for a single trial is quite large.

As evident from results in Table 1, our approach is much faster and gives a near-optimal solution. Let us consider the large microcode example ex6 having 165 microoperations and a graph density of 96.64%. Our method yielded a cost of 33 in 5.58 seconds. This is incomparably faster and gives a better cost solution than that obtained from either Nagle's or Baer's approach. Our algorithm does not depend on any random starting point and thus, is not a test and trial method like Hong's approach. For microcode ex6, out of 40 trials, only 4 yielded the best solution (32 bits) with Hong's algorithm and the run time for a single trial was 884.11 CPU seconds on a SUN3/60. Whereas, our method required only 5.58 CPU seconds on the same platform to yield a near-optimal cost of 33 bits.

2. Performance Measures on Random Graphs

<table>
<thead>
<tr>
<th>Problem Instances</th>
<th>Results (Non-Opt.)</th>
<th>Results (Optimized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>Edges</td>
<td>Density</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
<td>50%</td>
</tr>
<tr>
<td>200</td>
<td>900</td>
<td>50%</td>
</tr>
<tr>
<td>1,000</td>
<td>4,000</td>
<td>50%</td>
</tr>
<tr>
<td>2,000</td>
<td>8,000</td>
<td>50%</td>
</tr>
<tr>
<td>3,000</td>
<td>12,000</td>
<td>50%</td>
</tr>
<tr>
<td>4,000</td>
<td>16,000</td>
<td>50%</td>
</tr>
<tr>
<td>5,000</td>
<td>20,000</td>
<td>50%</td>
</tr>
<tr>
<td>6,000</td>
<td>24,000</td>
<td>50%</td>
</tr>
<tr>
<td>7,000</td>
<td>28,000</td>
<td>50%</td>
</tr>
<tr>
<td>8,000</td>
<td>32,000</td>
<td>50%</td>
</tr>
<tr>
<td>9,000</td>
<td>36,000</td>
<td>50%</td>
</tr>
</tbody>
</table>

We tested our algorithm on large size examples generated from random graphs having up to 2000 nodes and more than 1.8 million edges. Experiments were performed with graphs, having graph density varying from 15% to 90% and having the number of nodes varying from 100 to 2000. The experiments were performed on a SPARC 1+ Workstation. The results are given in Table 4. The non-optimized results indicate that the experiment was performed without using local search cost optimization. The experimental results are plotted in Figure 4. The low slopes of curves (almost horizontal) in Figure 4(a) indicate the independence of our algorithm to variations with graph density. It is observed that the execution time decreases slightly as the graph density increases. This is
due to the extraction of larger cliques at higher densities. Thus, the procedure initial\_graph\_partitioning requires less number of iterations as the graph density increases. It was observed that the runtime for a graph having 2000 nodes and 90% density, i.e., having 1.8 million edges was around 2,416 CPU seconds. This is incomparably faster than those of any presently known approaches. Figure 4(b) shows that for almost all the cases, the optimized cost obtained using our algorithm is better than the cost obtained using Hong et al.'s approach.

5 Conclusion

The microword length minimization problem is important to the design of a microprogrammed controller. In this paper, we presented an efficient algorithm to solve the microword length minimization problem. Our algorithm is capable of finding fast and near-optimal solutions for very large size microcodes, efficiently. This performance is achieved because of the application of an efficient initial graph partitioning and the application of an efficient local search algorithm for cost minimization.

6 Acknowledgements

S.K. Hong provided the microcode examples and his results. S.S. Ravi provided their recent work.

References


APPENDIX: Local Search for the Satisfiability (SAT) Problem — J. Gu (gu@eine.ucalgary.ca)

Local search is one of the early techniques proposed during the mid-sixties to cope with the overwhelming computational intractability of NP-hard combinatorial optimization problems. A primitive local search algorithm for the satisfiability (SAT) problem was formulated. The SAT6.0 algorithm, is given in Figure 1 [1,2].

Figure 1: SAT6.0: A Local Search Algorithm for the Satisfiability (SAT) Problem.

The algorithm for the satisfiability (SAT) problem, the SAT6.0 algorithm, is given in Figure 1 [1,2]. Procedure obtain_a_SAT_instance() initializes a SAT instance with \( n \) clauses, \( m \) variables and average \( l \) literals per clause. An object function of various forms representing the SAT problem was formulated. The SAT problem thus becomes a minimization problem to the object function. To begin, an initial solution point is chosen and the corresponding value of the object function is computed. During a single iterative search step, the test to see if the object function can be minimized is performed by function test_min(). If this is true, the minimization operation is performed by procedure perform_min(), followed by operation evaluate_object_function() that updates the object function. The algorithm terminates when a solution of the SAT instance is found. In practice, during the search, the algorithm could be stuck at a locally optimum point. To improve the convergent performance of the algorithm, a local handler may be added.

Algorithms SAT6.1, SAT6.2, SAT6.3 and SAT6.4 handle 1-dimensional or multi-dimensional search problems for arbitrary \( f \) or \( f \) of the UniSAT problem models [1,2]. Some twenty families of such SAT algorithms were developed and experimented, which include, for examples, the discrete local search methods (SAT1 family), the steepest descent methods (SAT7 family), modified steepest descent methods (SAT8 family), newton methods (SAT10 family), quasi-newton methods (SAT11 family), descent methods (SAT14 family), conjugate direction methods (SAT16 family), Boolean difference method (SAT20 family), and other techniques [1,2,3]. Among hundreds of SAT algorithms developed/experimented, some such SAT algorithms run in \( O(n!l) \) time and take \( O(nl) \) space [1,2,3,4]. Table 1 gives the real execution performance of an early backtracking version of a SAT algorithm for some practical CSP problem instances [2,4]. The hardness of the SAT problem increases when the number of literals \( l \) in each clause decreases or \( n/m \) increases. Tested through years of real algorithm executions, our SAT algorithms are capable of computing hard, very large-scale and practical satisfiability problems with excellent convergence performance.

![Table 1: Computing Performance on a SUN SPARC-1 for practical CSP Problems. Time unit: seconds.](image)

<table>
<thead>
<tr>
<th>Problems</th>
<th>UniSAT</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>200</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>1000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>2000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>3000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>4000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>5000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>6000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>7000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>8000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>9000</td>
<td>944</td>
<td>1161</td>
</tr>
<tr>
<td>10000</td>
<td>944</td>
<td>1161</td>
</tr>
</tbody>
</table>