OPERATION SCHEDULING FOR PARALLEL FUNCTIONAL UNITS USING GENETIC ALGORITHMS

Th. Zeitlhofer and B. Wess

INTHFT, Vienna University of Technology
Gusshausstrasse 25/389, A-1040 Vienna, Austria
E-Mail: thomas.zeitlhofer@tuwien.ac.at, bwess@email.tuwien.ac.at

ABSTRACT

In this paper, we describe a new and efficient approach to solve the scheduling problem for VLIW architectures. The scheduling times of the operations are used as the problems parameters. This in conjunction with a pruning technique based on critical path analysis leads to a significant reduction of search space complexity. A genetic algorithm is used to search for valid schedules of a given length. The genetic algorithm uses a fitness vector that guides the genetic operators crossover and mutation resulting in a fast convergence towards near perfect solutions. The proposed method is also applicable to the problem of register allocation by using a different fitness function. Another advantage of the genetic algorithm approach is that usually a great number of equally performing schedules is obtained allowing for further optimization subject to arbitrary constraints.

1. INTRODUCTION

The need for DSP compute cycles increases rapidly. This is primarily related to the growing field of media and communications processing. In modern DSPs a potential increase in performance is achieved by the VLIW architecture design [1] which enables the use of instruction level parallelism (ILP).

Involved with the existence of parallel functional units is an increased complexity of the scheduling problem. So the need for new and efficient compiler techniques becomes obvious, especially when the poor performance of C-compilers for DSPs is considered.

Scheduling techniques for VLIW architectures have been studied for several years. They can be subdivided into *local* and *global* compaction. Local compaction considers only *basic blocks* without control flow [2]. This work will focus on local compaction as these techniques can be reused in global compaction [3], too.

Heuristic approaches (e.g. *list scheduling*) are typically used for basic block scheduling. Heuristics have to be adapted for specific combinations of problem and architecture targeted.

Genetic algorithms (GAs) allow a more general approach to the scheduling problem. In [5] GAs were applied to the scheduling problem and also used to drive the heuristics of a list scheduler. GAs have been successfully applied to find optimized sequential schedules in [4].

In this paper, we use GAs to directly search for highly compact schedules without operating on ordered sequences. Our new approach parameterizes the scheduling problem in terms of scheduling times of the operations. The search space is reduced by a pruning technique which only removes invalid schedules. The global search behavior and the flexibility of GAs make them suitable to solve this problem.

We first describe the parameterization of the scheduling problem in the context of the GA. The pruning technique that significantly reduces the search space complexity is explained in detail. We discuss the main parts of our GA. Adaptation to different target architectures is achieved by modifying the *fitness function*. We devised *guided* genetic operators to increase the rate of convergence by reducing randomness in the search process. Operation scheduling and register allocation are strongly interdependent. We show that our approach is also applicable to the problem of register allocation. Finally, typical experimental results are presented to illustrate the performance of our GA.

2. GENETIC ALGORITHM

Genetic algorithms (GAs) were introduced by [6] and are an effective method to solve complex optimization problems. They are based on evolution in the biological sense and operate on a set of strings (chromosomes) the so-called population. A string represents a possible solution[1] for the parameters of the scheduling problem. New solutions (children) are generated using crossover and mutation operators. The chromosomes are rated using a *fitness function*. The members of the next generation are chosen based on the fitness.

GAs are well suited to the scheduling problem [5, 4] because of the global search behavior and the flexibility of the fitness function.

2.1. Operation scheduling

Data dependencies between operations in a program can be represented by a directed acyclic graph, the data dependence graph (DDG) $G(V, E)$. Each node $n_i \in V, i = 1 \ldots N$ corresponds to an *elementary* operation.[2] Each edge $e \in E$ corresponds to a data dependency constraint. We distinguish two kinds of constraints (compare [4]): *before-* and *after*-constraints. Suppose two nodes n_i and n_j. If n_j has to be scheduled at least d_a cycles after n_i, then there is an after-constraint between the two nodes

$$t(n_j) \geq t(n_i) + d_a. \tag{1}$$

If the latest possible time to schedule n_j is d_b cycles after n_i then we say there is a before-constraint between the two nodes

$$t(n_j) \leq t(n_i) + d_b. \tag{2}$$

This work has been supported by ÖNB grant 6867.

[1] A string in the population not necessarily corresponds to a valid schedule. So we distinguish between solutions and *valid* solutions.

[2] An elementary operation is an operation that can be evaluated by at least one of the functional units.

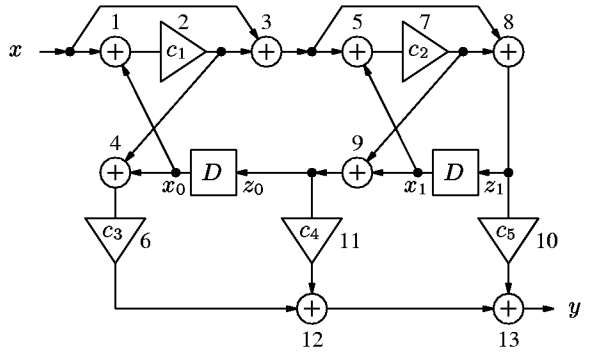

Figure 1: Second order lattice filter.

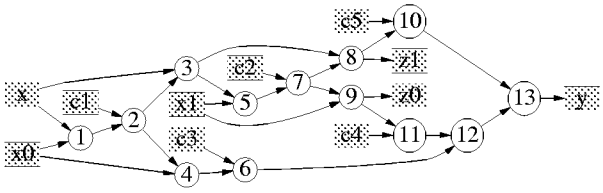

Figure 2: Data dependency graph of the lattice filter.

So each edge of the DDG is associated with two weights d_a and d_b. The concept of weighted constraints enables us to take pipeline effects into account where there has to be a minimum delay between certain operations (e.g. write operations and consecutive read operations accessing the same memory location).

For the following discussion we consider the second order lattice filter in Figure 1. The operations are labeled with node numbers which are used in the corresponding DDG in Figure 2. All edges have equal weights: $d_a = 1$ and $d_b = \infty$.

The scheduling problem is to assign a time $t(n)$ to each node $n \in V$. Thereby it has to be taken into account that no data dependence constraint is violated and that in every cycle t_i there are sufficient functional units so that all nodes $n_i \in V | t(n_i) = t_i$ can be evaluated.

We do *position pruning* similar to [4] to reduce the search space. For each node n_i, the *as-soon-as-possible* $t_s(n_i)$ and *as-late-as-possible* $t_l(n_i)$ scheduling times are determined [7]. This information can be represented by a *pruning matrix*

$$\mathbf{P} = (P_{ij}) = \begin{cases} 1 & \text{for} \quad t_s(n_j) \leq i \leq t_l(n_j) \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The number of rows of $\mathbf{P}$ defines the length of the schedule. Note that in our approach we do not start with an initial sequential schedule which we then try to compact. The total length of the schedule is selected first and then we attempt to find a valid schedule of that length. So we start to look at the most compact (parallelized) schedules first. The great advantage of this approach is the minimization of the search space complexity. The lower bound of the length of the schedule is defined by the length of the critical path (CP), $lb_g = \text{length(CP)}$. The problem of finding a valid schedule of length lb_g has minimum search space complexity. For the lattice filter in Figure 1, the nodes $n_i, i = \{1, 2, 3, 5, 7, 9, 11, 12, 13\}$ belong to the critical path of length 9. If no assumptions concerning the possible scheduling interval for each node are made, a valid solution must be found out

t	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}	n_{13}
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	1	1	1	0	0	0	0	0	0	0
5	0	0	0	1	0	1	1	0	0	0	0	0	0
6	0	0	0	1	0	1	0	1	1	0	0	0	0
7	0	0	0	0	0	1	0	1	0	1	1	0	0
8	0	0	0	0	0	0	0	0	0	1	0	1	0
9	0	0	0	0	0	0	0	0	0	0	0	0	1

Table 1: Pruning matrix for the lattice filter.

of $9^{13} \approx 3 \cdot 10^{12}$ possible schedules. Computing the *as-soon-as-possible* and *as-late-as-possible* scheduling times allows to construct the pruning matrix in Table 1. If position pruning is applied, just 64 possible schedules remain.

The existence of a valid schedule of given length depends on the functional units provided by a specific architecture. If N operations have to be scheduled on m functional units, then a lower bound for the length of a possible schedule is given by $lb_a = \lceil \frac{N}{m} \rceil$.

The maximum of lb_g and lb_a defines the minimal number of rows for the pruning matrix $\mathbf{P}$.

As we consider the general case of an architecture which offers a set of heterogeneous functional units, it cannot be guaranteed that a valid schedule of this length exists. In the case that the genetic algorithm does not find a valid solution the following two strategies may be applied:

1. Increase the length of the schedule and start the GA again (*iterative* approach).
2. Resolve violations of constraints by inserting additional cycles where needed (*direct* approach).

The *direct* approach may be very inefficient if many conflicts have to be resolved. So this approach is applicable if only a few constraints are violated. However, in this case a valid schedule can be found easily without solving the entire problem again. If many constraints are violated, the iterative approach is likely to find a better solution but the problem gets harder as the complexity of the search space increases.

2.1.1. Fitness Function

A solution of the scheduling problem is represented as a vector $\mathbf{s} = (s_1, s_2, \ldots, s_N)^T$ where $s_i = t(n_i)$ represents the scheduling time of node n_i.

The fitness function has to take into account both data dependence constraints and resource constraints. For solution $\mathbf{s}$, a cost value is associated with element s_i if any constraint of the corresponding node is violated . So we maintain a cost vector $\mathbf{c} = (c_1, c_2, \ldots, c_N)^T$ where the element c_i represents the cost value associated with node n_i.

In relation to the data dependence constraints, each edge in the DDG has to be examined. Is there an edge between nodes n_i and n_j, then the resulting costs are given by

$$\begin{aligned} c_i = c_j \quad &= \quad (s_i - s_j + d_a)\, \sigma\,(s_i - s_j + d_a) \\ &+ \quad (s_j - s_i - d_b)\, \sigma\,(s_j - s_i - d_b)\,, \end{aligned} \tag{4}$$

where $\sigma(x) = \frac{1}{2}(1 + \text{sign}(x))$ is the Heaviside function. The cost function in (4) judges the violation of a data dependence constraint

not only qualitatively but also quantifies the degree of violation.[3]

Additional costs accrue if there are resource conflicts. For all scheduling times, the cost value of those nodes is increased that cannot be scheduled at that time due to an insufficient number of functional units. As we consider in our approach a set of heterogeneous functional units, each operation has to be classified and possible parallel combinations of classes have to be specified. A class of an operation is not necessarily associated with a functional unit. Consider we have two unrelated functional units: one ALU and one MAC unit. Assume that the MAC unit can perform ADD and SUB instructions too. So we need three classes to define the multifunction-instruction set. The first class contains all ALU instructions except ADD and SUB. The second class contains the MAC instructions (without ADD and SUB) and the third class consists only of addition and subtraction. In table 2, the possible parallel combinations of these three classes are shown. At a

	class 1	class 2	class 3
combination 1:	1	1	0
combination 2:	1	0	1
combination 3:	0	1	1
combination 4:	0	0	2

Table 2: Parallel combinations of instruction classes.

given time, the number of required functional units for each class has to be compared with all combinations the architecture offers. The combination which contributes the least overall costs is used to compute the additional costs. Assume at a specific time two ADD/SUB operations are needed. In this case, combination 4 in Table 2 is used and the resulting costs are zero. If two ALU operations are needed combination 1 or combination 2 can be used. Both result in a cost value of 1 for all nodes that belong to class ALU. If combination 3 or combination 4 is used, then the cost value would be 2 because the cost value for all nodes of a certain class is computed as the difference between the number of needed resources and available resources. That is, also in the case of resource constraints, the degree of violation is quantified.

The overall costs of a solution $\mathbf{s}$ are the sum of the cost values associated with the nodes: $\sum_{i=1}^{N} c_i$. The fitness of a solution is given by the negative cost value (plus a certain offset to yield a positive fitness).

2.1.2. Genetic Operators

The basic operators in GAs are crossover and mutation. The proper choice of these operators determines the performance for a certain problem. In [5], crossover operators that emphasize order were found to perform best for scheduling problems.

We implemented *order-based* [5], *position-based* [5] and *exchange* [4] crossover. The best results we obtained with the *exchange* operator which is shown in Figure 3.

In this example the number of nodes is $N = 7$. The starting point of the crossover is node number 4. To generate child 1 first the scheduling time for node number 4 of parent 1 is changed from 5 to 1. Then parent 2 is searched for a node with scheduling time 5.[4] This is node number 6. Now the scheduling time for this node of parent 1 is changed from 6 to 5. Then parent 2 is searched for

[3]The degree of violation is the distance from the allowed region as defined in (1) and (2).

[4]If more than one node with scheduling time 5 exists the first one found is taken.

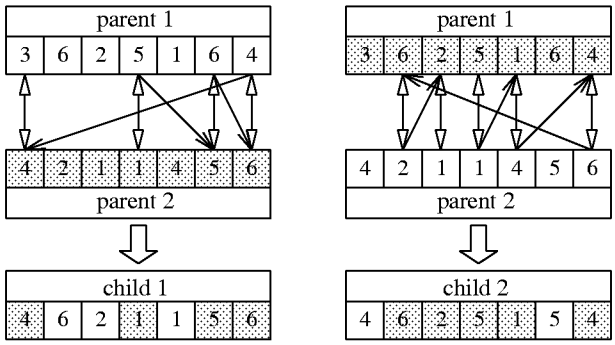

Figure 3: Exchange crossover.

an node with scheduling time 6 and so on. Child 2 is based on parent 2 and so at the beginning of the crossover the scheduling time for node number 4 is changed from 1 to 5. Then parent 1 is searched for a node with scheduling time 1 and so on. The exchange crossover stops if a node is found which already has been modified or a specific scheduling time can't be found in the other parent. We also use the information from the fitness function to perform the crossover. As the starting point of the crossover, we choose the node with the highest associated costs.

For mutation we use a kind of *guided* mutation. Again we take the node with the highest costs and assign randomly another scheduling time (within the corresponding pruned interval). In addition, a second randomly chosen node is treated accordingly. We also use a variable mutation rate. When the average cost of all solutions in the population comes closer to the minimum cost value in the population, the mutation rate is increased accordingly.

2.2. Register Allocation

The problem is to allocate a register for the output of each node. So the structure is similar to the scheduling problem – instead of scheduling times, valid registers have to be found. Also the concept of the pruning matrix is applicable. If for instance certain functional units require a special subset of registers for input/output, then the number of possible register allocations is reduced.

A major advantage of GAs is the flexibility of the fitness function. Constraints concerning heterogeneous register sets can be easily taken into account.

3. EXPERIMENTAL RESULTS

We used the proposed genetic approach to find a valid schedule for the lattice filter in Figure 1. To increase the complexity of the example, we unfolded the filter loop four times. This results in a scheduling problem for 52 nodes and a critical path of length 27. Position pruning reduces the search space from $3 \cdot 10^{74}$ to $6 \cdot 10^{23}$ possible solutions. We assumed an architecture that provides two homogeneous functional units (two arbitrary operations can be performed at the same time). In that case, it can be shown that no valid schedule of length 27 exists – a minimum of 28 cycles is required. We start to find a schedule of length 27 and so a minimum cost value of 1 must be expected.

The population consists of 50 solutions and is randomly initialized. Out of this population, 100 children are generated using

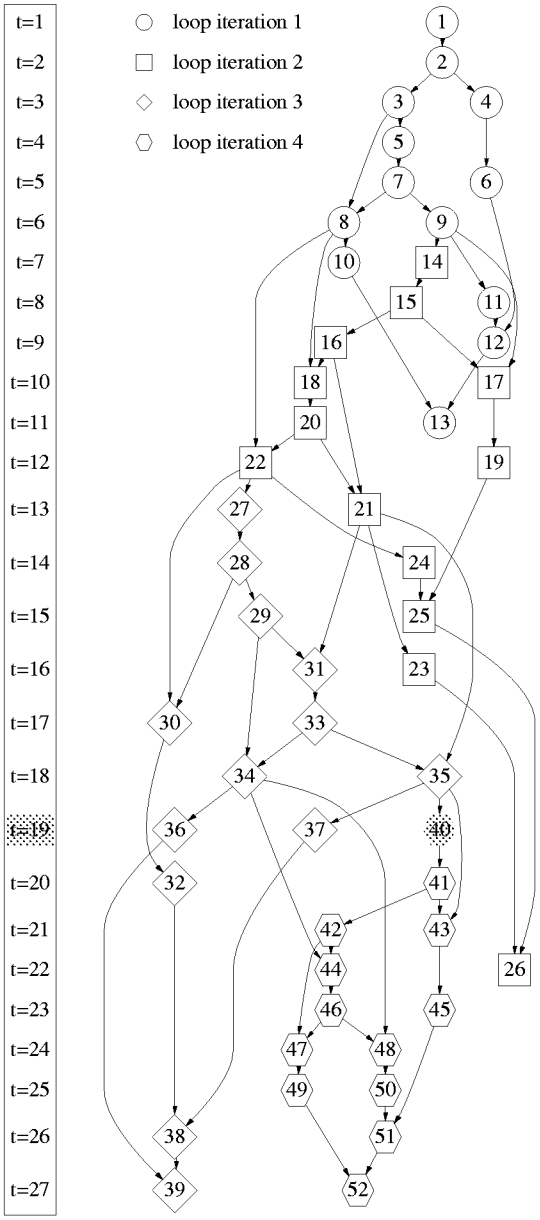

Figure 4: Solution with cost value 1.

roulette wheel parent selection.[5] Then from the resulting 150 solutions, the 50 best are chosen to form the next generation. The average performance of 10 independent invocations of the GA is shown in Figure 5. It can be seen that in the beginning the costs are reduced very quickly. This means, if less generations are computed, the result still will be relatively good. In nine of ten runs, the GA ends up with cost value 1 solutions. A particular result is shown in Figure 4. This cost value reflects the fact that two multiplications and one addition should be performed in parallel at $t = 19$. As this is not possible with the assumed architecture, one of the three nodes (n_{40}) has a cost value of 1. However, a

[5]Roulette wheel parent selection means that a solution is chosen as one of the parents according to its relative fitness value in the population.

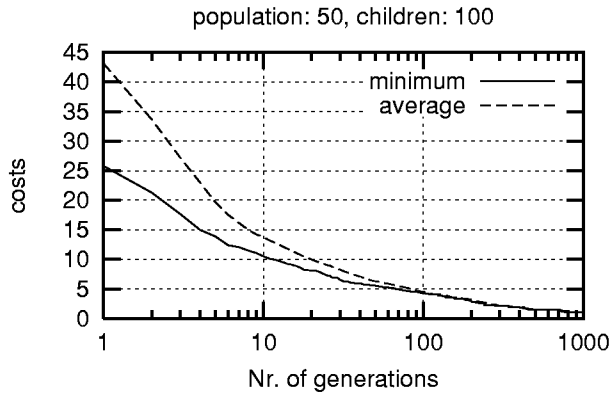

Figure 5: Average convergence performance of the GA.

valid schedule can be easily found using the *direct* approach (see section 2.1). An additional cycle is inserted immediately after the scheduling time $t = 19$ for node n_{40}. So a schedule of length 28 is found which is optimal for this example as we mentioned before. But in fact not only one valid schedule but 50 valid schedules are found since each member of the corresponding population has a cost value of 1 after 480 generations. To maintain variety we ensure all solutions in the population to be different. Therefore 50 valid schedules can be generated.

4. CONCLUSIONS

A new genetic algorithm approach for the scheduling problem with parallel functional units has been presented. Minimizing the number of search space points and using guided genetic operators allow to realize a fast converging algorithm that handles both scheduling and register allocation.

5. REFERENCES

[1] P. Faraboschi, G. Desoli, and J.A. Fisher, "The latest word in digital and media processing", *IEEE Signal Processing Magazine*, March 1998.

[2] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local microcode compaction techniques", *ACM Computing Surveys*, vol. 12, pp. 261–294, September 1980.

[3] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction", *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[4] T. Chung, *CHARTS: A compiler for hard real-time systems*, PhD thesis, Purdue University, August 1995.

[5] S.J. Beaty, *Instruction Scheduling Using Genetic Algorithms*, PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado 80523, October 1991.

[6] J. Holland, *Adaptation in Natural and Artificial Systems*, PhD thesis, University of Michigan, Ann Arbor, MI, 1975.

[7] R. E. Chrochiere and A. V. Oppenheim, "Analysis of linear digital networks", *Proceedings of the IEEE*, vol. 63, pp. 581–595, April 1975.