ABSTRACT

Scheduling is a crucial task in behavioural synthesis and a NP-hard optimisation problem. Neural network computation paradigms bring potential for efficient solutions to such problems. This paper presents a new scheduling algorithm based on Kohonen's rule for self-organisation. The algorithm has an inherent hill climbing mechanism, copes with a comprehensive set of constraints and can be implemented on massively parallel structures. Its performance on well-known benchmark examples, presented in the paper, is on par with the best reported.

Keywords: Behavioural Synthesis, Neural Nets, Optimisation Techniques, Scheduling, Self-Organisation.

1.0 Introduction

A recent surge in the neural net research has resulted in some promising new optimisation techniques. The best known and most widely used of these is the Hopfield net. Kohonen's Self organisation algorithm, though presented more often in the context of learning and pattern recognition, has also been used to solve the travelling salesman problem TSP88, cell placement CELL89, SUNG89 and operation scheduling in high level synthesis NISCHE89. This paper addresses this last application and is a generalisation of the techniques discussed in NISCHE89.

High level synthesis of digital systems consists of translating an abstract behavioural specification of the system to a register transfer level structure. This is usually achieved in four steps:

1. Translating an abstract behaviour expressed in algorithmic style to a graph based representation. This graph structurally represents the control and data dependencies of the operations in the behaviour. The graph is usually called a control and data flow graph (CDFG).
2. This step is concerned with allocation of hardware resources. These are functional units like adders, multipliers etc. to realise operations, registers to store intermediate results between control steps and lastly busses and muxes for transferring data. Often only functional units are explicitly allocated and an attempt is made during scheduling (Step 3) and later during binding (Step 4) to minimise the use of registers, busses and muxes.
3. In this step operations are assigned to control steps or clock cycles. This is called operation scheduling and is the main concern of this paper. The primary objective of scheduling is to minimise the number of control steps required to fulfill a behaviour, given a certain amount of hardware resource. Alternatively, given a global time constraint the scheduler could be asked to minimise the amount of resource required. Steps 2 and 3 are highly interdependent and are not necessarily done in the order presented here. It is common to iterate between them to improve the overall performance of the system.

Scheduling is crucial as it directly influences the following cost variables of the synthesised structure:

- The number and type of functional units. A smarter scheduler can use fewer functional units to schedule a behaviour in the same number of control steps.
- The storage requirements (registers, memory). As the schedule decides how many intermediate results must be held at any instance, it influences the storage requirement.
- Scheduling together with binding also influences the data transfer requirements.

The above cost variables should be minimised by scheduler in proportion to their hardware cost.

4. The last step in data path synthesis is binding: a) allocated functional units to specific operations, b) registers and other storage elements to specific intermediate results and c) creating muxes and/or busses to realise data transfers.

2.0 Related work

In this section we briefly review the scheduling techniques currently in use and their associated strengths and weaknesses.

The simplest technique is to let the user do the scheduling. This is used in the Silc system, one of the earliest so called silicon compilation systems. Combinatorial explosion reduces the usefulness of this technique.

The simplest automatic technique is known as As Soon As Possible (ASAP) scheduling. This approach was used in CATREE and the Emerald/Facet systems among others. If the amount of hardware is restricted, the operations are conditionally postponed when there is a resource conflict. This approach was used in the MIMOLA, CMUDA and Flame1 systems. The problem with this technique is that often less critical operations block more critical ones, resulting in longer than necessary schedules.

This problem is alleviated in list scheduling algorithms. In list scheduling, operations are sorted according to their data and control dependencies. These are then scheduled one at a time in the order in which they are...
sorted. If there is a resource conflict a priority function is used to decide which operations to defer. The priority function usually only considers the local effect of assigning operations to control steps, resulting in locally minimum solutions. The priority function varies across the systems using this technique, examples of which are SLICER, EMUCS and CATHEDRAL-II.

Force directed scheduling as used in the HAL system tries to overcome the problem of locally minimum solutions by taking into account more global effects of assigning an operation to a control step. At each iteration an operation is assigned to a control step which causes the least increase in overall concurrency of operation, storage and interconnect requirement, weighed in proportion to their hardware cost. Although this algorithm is more global in each iteration, it is essentially a steepest decent algorithm. Once an operation has been assigned to a control step, its assignment is not reconsidered: It lacks a hill climbing mechanism.

Devadas and Newton incorporate a hill climbing mechanism in their system by using an algorithm based on simulated annealing. They use a complex cost function and model the problem as a placement problem, an area where simulated annealing has been very successful. The best that can be said about the difficulty in parallelising simulated annealing is that it not easy: the tradeoff between accuracy and speed is tricky.

The algorithm described in this paper is inherently parallel in nature, has a hill climbing mechanism and has a built in cost weighting mechanism which allows it to do tradeoffs in functional unit, register and interconnect requirements based on their hardware cost. It is similar to HAL in the sense that it reduces the resource usage by trying to uniformly distribute them over control steps as much as possible. It differs from HAL and others in: a) treating schedule space as a continuous space. b) all operations influence assignment of an operation to a position in the schedule space; HAL considers the effect of the assignment on concurrency of only successors and predecessors. c) it evaluates several solutions many of them worse than previous, before settling for a near minmal solution.

3.0 The Self Organising Scheduling Algorithm

3.1 Input and Output

The primary input to the algorithm is a control and data flow graph (CDFG). It is a directed graph \((V, A)\). \(V\) is a set of nodes corresponding to the operations to be scheduled. \(A \subseteq V \times V\), is set of arcs connecting nodes of \(V\). These are called data arcs.

Besides CDFG, the other input to the algorithm is a set of constraints to be satisfied. These constraints take the following form:

- Type and number of functional units:
  \[N = N_1 + N_2 + \ldots + N_m\]
  \[T = \{T_1, T_2, \ldots, T_n\}\]
  Where \(T_i\) is the \(i\)th functional unit type and \(N_i\) is the number of functional units of type \(T_i\).

- A function \(g\), mapping \(V\) to \(T\): \(g: V \rightarrow T\).

- Global time constraint \(MAX_K\), local time constraints, cycle time and propagation delay of each functional unit type.

The output from the algorithm is a schedule table \(S = MAX_K \times P\) showing which operation has been assigned to which control step. The number of units of type \(T\) actually used is decided by the maximum used in any control step for type \(T\) and is generally less than \(N_i\).

3.2 Network Organisation and related concepts

![Network organisation and neigbourhood concept](image)

N.B. Although every output node is connected to the input node only a few are shown in the figure for clarity.

The components and organisation of the network used in the algorithm are as follows: The network is made up of an input node pair \((k, f)\), and a set of output nodes \(V\) corresponding to the operations to be scheduled. An input vector is defined as \(I = (I_1, I_2, \ldots, I_N)\), where \(I_1\) and \(I_2\) are continuous random variables having uniform probability distribution over the range \([0, \ldots, NMAX_K]\) and \([1, \ldots, N]\) respectively. Every output node \(v \in V\) is connected to the input nodes \((k, f)\) by a weight pair \(W_v = (W_a, W_r)\) which specifies the node's position in the schedule space. \(W_a\) and \(W_r\) are continuous variables over the range \([0, \ldots, NMAX_K]\) and \([1, \ldots, N]\) respectively.

![Schedule space, time frames and type frames](image)

Kohonen's algorithm for self organisation is a learning algorithm. It trains the net by iteratively presenting input vectors at the input nodes and adapting the weights of
output nodes. As a result the output nodes are uniformly distributed over the schedule space because of the uniform distribution of input vectors.

We use the following concepts to describe the algorithm:

The data dependency distance $x$, between two data dependent operations is defined to be the longest path between them in the CDFG $(V, A)$.

With every operation is associated a time frame and a type frame. The time frame of an operation is defined as the range of control steps between ASAP and ALAP values. The ASAP and ALAP values are determined by performing as soon as possible and as late as possible scheduling. Time frames of operations are shown in fig. 2. Each functional unit type $T_i$ occupies a contiguous slot of length $N_i$ in the $N$ dimension of the schedule space $S = \text{MAX}_K \times N$. This slot is called the type frame of operations which are mapped to this type by the function $g(v) \rightarrow T_i$. At any instant $t$, the neighbourhood $\text{NE}_u(t)$ of operation $v$ is defined as the set of predecessors and successors to $v$ whose dependency distance with $v$ is less than the current neighbourhood radius $\sigma(t)$. Initially large and slowly decreases over time. These concepts are illustrated in fig. 1.

We also use a gain term which is a function of dependency distance $x$, the current neighbourhood radius and the initial neighbourhood radius $\text{INIT}_\sigma$. It also decreases with time and is defined as follows:

$$\eta(x, t) = \frac{1}{\sqrt{2\pi} \sigma(t)} \exp\left(\frac{-x^2}{2\sigma(t)^2}\right)$$

3.3 The Algorithm:

Step 1. Initialise the positions, i.e., the weight pairs $W = (W_i, W_f)$ of all operations to small random values. Set the initial value of $\sigma(t)$ to $\text{INIT}_\sigma$. We found that setting $\text{INIT}_\sigma$ to half the critical path in CDFG $(V, A)$ was adequate.

Step 2. Present a new input by generating a random vector $l(t) = (l_i(t), l_f(t))$, such that probability distribution of $l_i$ and $l_f$ are uniform over the ranges $1..\text{MAX}_K$ and $1..N$.

Step 3. Compute the distance $d_v$ between $l(t)$ and the weight pair $W_i(t)$ of all operations whose time frame and type frame include the vector $l(t)$. This way all eligible operations compete for the position specified by the input vector $l(t)$.

$$d_v = \sqrt{(l_i(t) - W_i)^2 + \alpha_i \cdot (l_f(t) - W_f)^2}$$

$\alpha_i$ is a control parameter, where $i$ corresponds to the $T_i$ functional unit type. It allows the algorithm to do tradeoffs in resource usage proportional to their hardware cost.

Step 4. Select the operation $v^*$ with the minimum distance $d_v$. $v^*$ is said to have won the position specified by the input vector $l(t)$ in the schedule space.

Step 5. Adapt the weights of operation $v^*$ and its neighbours as follows:

$$W_{<}(v^*) = W_i(t) + \eta(x, t) (l(t) \pm \text{min_dist} (v, v^*) - W_i(t))$$

For $v \in \text{NE}_v(t)$.

Here $x$ is the dependency distance between $v$ and $v^*$. $\text{min_dist} (v, v^*)$ is similar to data dependency distance, except that it takes into account the actual propagation delays and the effect of aligning operations to control step boundaries. The plus sign is used if $v$ is successor of $v^*$, the minus sign if $v$ is a predecessor to $v^*$.

The above equation shows that both $W_i$ and $W_f$ weights of $v^*$ and its neighbours are adapted. But in reality $W_i$ weights are adapted only for $v^*$ and those neighbours which are mapped to the same operation type.

By adapting the weights, $v^*$ moves towards the position specified by vector $l(t)$ and pulls its neighbours towards $l(t)$, respecting data dependencies. The movement of $v^*$ is said to be primary and that of its neighbours secondary.

The primary moves are made as a result of winning the competition in step 3 and as such are sensitive to the presence of other operations, both data dependent and data independent, whereas the secondary moves are not. Sensitivity of secondary moves to other operations allows the algorithm to make hill climbing steps. As the neighbourhood decreases with time the number of primary moves increases while the number of secondary moves decreases. When the neighbourhood radius is less than 1.0 only primary moves are possible.

As a result of steps 3, 4 and 5, the operations which are initially clustered near the centre of the schedule space tend to disperse and move to the positions where they maximise their chances of winning as many input vectors as possible. The net effect is uniform distribution of operations over the schedule space.

As every output node is connected to the input nodes by an independent weight pair, we could compute the distance function of step 3 simultaneously for all output nodes, if massively parallel structures were available. The same holds for step 5.

Step 6. Decrease $\sigma(t)$. Repeat by going to STEP 2 until $\sigma(t) = 0.0$.

Essential to the convergence of the algorithm are shrinking neighbourhood and decreasing gain, the latter being a function of the former. Initially, when the gain and the neighbourhood are large (analogous to high temperature in simulated annealing) operations can move over large distances, influence far off neighbours and often ignore the presence of other operations. As the neighbourhood and gain decrease with number of iterations, operations move smaller distances, influence closer neighbours and can rarely ignore the presence of other operations. This behaviour is responsible for the hill climbing mechanism in the algorithm.

Step 7. Round the $W_i$ weights of operations to the nearest control step or time boundary if the operations are chained in a control step.

3.5 Tradeoffs in resource usage

As mentioned in the step 3 of the algorithm we use the parameter $\alpha_i$ to do tradeoffs in the resource usage. Parameter $\alpha_i$ when greater than 1.0 has the effect of shortening the distance in the resource dimension. This causes fewer operations of a particular type to be assigned to the same control step. This is illustrated in fig. 3, where $\alpha_i$ for the adder type is smaller than that for multipliers. When $\alpha_i$ and $\alpha_f$ compete for position $x$, $\alpha_f$ wins as it is closer to $x$. This allows it to be concurrent with $\alpha_f$. In a similar
competition op2 loses to op4 because of the larger $\alpha$ for multipliers and cannot be concurrent with op4. In general this mechanism makes the scheduler favour solutions using fewer expensive resources at the expense of more cheap resources. The same concept can be used to do tradeoffs in register and interconnection usage. We use the following heuristic to calculate $\alpha$:

$$\alpha_i = \frac{N_i}{N_{i+1}}$$

The rationale behind this heuristic is that we want to make $\frac{N_i}{N_{i+1}}$ a distance in the resource dimension equal to one control step, thereby discouraging the algorithm from crowding a control step with more operations than the desired $N_{i+1}$.

For example, the fifth order elliptic wave filter from KUNG85 was used as a benchmark in the 1988 workshop on high level synthesis and has been used by HAL88 and others. This example has 43 operations subjected to 60 data precedence constraints. We use this example to demonstrate the algorithm's ability to handle multicycle operations and pipelined units. Like HAL88, we assume addition and multiplication take one and two control steps respectively. A multiplier is a two stage pipelined unit and coefficients are not necessarily multiples of two, so multiplication is not performed by a shifter.

We set the global time constraint to 19, 18 and 17 control steps and made available more than the necessary number of functional units. In the first case the algorithm found the cheapest schedule using 2 adders and 1 multiplier. In the last case it found the fastest schedule using 3 adders and 2 multipliers. In terms of number of functional units these results are equivalent to HAL88. The results are summarised in Table 1. Fig 4 shows copies of the actual schedules taken from the screen. The first three schedules are respectively for 19, 18 and 17 control steps. The schedule space is divided into two type frames and 22, 21 and 20 control steps, respectively. The type frames are separated by a thicker line between adder type on the left and multiplier type on the right. Start and stop times are shown at the top of the picture.

The fourth picture shows an experiment in using the same algorithm to aid binding. This is done by letting the algorithm create a schedule and then running it again to adapt the weights for the resource dimension only. This has the effect of aligning operations sharing data. This can be seen in the fourth picture where adders sharing data are more sharply aligned than in the second picture which shows the same schedule before the binding run.

Our next example is borrowed from MAHA86 and used by PAULIN88. This example illustrates the algorithm's ability to handle multiple operations per cycle. Briefly the constraints are: four control steps, more than necessary adders and subtractors, both having a propagation delay of 40 ns and a cycle time of 100 ns. The results are summarised in Table 2.

We ran the algorithm on a number of benchmark and other examples of moderate to large size. The results in all cases were on par with the best reported in the literature. Here, we present two of these examples:

4.0 Results

We present the results in all cases were on par with the best reported in the literature. Here, we present two of these examples:

5.0 Conclusion

We have presented a new algorithm for scheduling data path operations using Kohonen's Self Organising algorithm. We have also presented the algorithm's performance on a well known benchmark example and another example taken from current literature. The algorithm has some additional capabilities which we are exploring now. These are its ability to optimise the usage of registers, busses and muxes in the same way as functional units. Another feature apparent from the obtained schedules is the ability of the algorithm to perform binding by aligning data dependent operations of the same type. We are currently also extending the algorithm to handle arithmetic pipelining.

6.0 Acknowledgements

The work reported in this paper has been funded as part of the Swedish Information Technology program (F4). The authors would like to thank Ian Hill of Swedish Institute of Microelectronics for valuable suggestions and comments to improve the quality of this paper.
Fig 4. Schedules of the fifth order elliptic filter

REFERENCES


[SUNG89] Sung–So0 Kim and Chong–Min Kyung, Circuit Placement in Arbitrarily-Shaped Region using Self Organisations, ISCAS ’89.