Springplay : A New Class of Compile-Time Scheduling Algorithm for Heterogeneous Target Architectures

G. PALLER and C. WOLINSKI

IRISA/INRIA, EP-ATR group, Campus de Beaulieu F-35042 RENNES, FRANCE

Abstract. This paper presents a compile-time scheduling algorithm called *Springplay* which approaches the well-known problem of scheduling signal-flow graphs onto heterogeneous architectures from a new way. This technique uses *distances* for describing how well the node of the signal-flow graph matches to a processing unit and *forces* to affect these distances. The algorithm is related to certain class of neural networks. The method produced remarkable results during the experiments without excessive computation time.

Keywords. Compile-time scheduling, parallel processing, heterogeneous architectures, co-design

1. INTRODUCTION

The research efforts around the static scheduling algorithms has been intensified with the increasing number of applications requiring high processing throughput in real-time. These applications include various fields of Digital Signal Processing (radar-signal processing, image processing) and high speed real-time control systems. The restrictions imposed by the static data flow model presented in [1] are often acceptable in such applications and workarounds exist in the form of hybrid static/dynamic schedulers [2]. The static scheduling problem is NP-complete in a strong sense [3] so it is attacked by different form of heuristics. These heuristics can be divided into four groups.

1. Modified forms of Integer Linear Programming (ILP) methods [4]. Using ILP for the scheduling problem quickly becomes intractable [5] but heuristic can be used to limit the scope of ILP to certain sections of the graph [6].
2. Branch&Bound type decision tree search algorithms [7]. These algorithms suffer from an exponential growth of computation time as the size of the problem increases.
3. Generalized List Scheduling algorithms [8, 9, 10, 11, 12]. These algorithms employ different heuristic criterias to choose a candidate node among the ready nodes and a candidate processor for it. These methods suffer from the so called "horizont effect" because they can foresee the result of a local decision only in a limited distance.
4. Different kinds of graph partitioning algorithms. These methods cuts the graph first into sections by using heuristic methods which consider parallelism, communication costs, etc. then map the sections into the hardware [13]. Generally these techniques are unusable in heterogeneous processor environment.

Our design environment for DSP applications also features automatic code partitioning/scheduling. As this environment was targeted especially for heterogeneous multiprocessor architectures, an extensive bibliography research and performance evaluation has been made to choose a partitioning/ scheduling algorithm among the existing ones. During these experiments we were convinced that the methods we investigated were either unusable or provided very poor results in a highly heterogeneous environment. This way we were forced to develop an algorithm of our own.

The heterogeneous environment adds a new dimension of liberty to the scheduling problem so the complexity of the problem increases significantly. The - even partial - search of the decision tree or a formal ILP solution are proved to produce inacceptable execution times. Inspired by the succes of Hopfield class neural networks in solving optimization tasks [14, 15, 16, 17] and by a very graphic representation of affectations in the Force-Directed Scheduling algorithm [10] we de-

vised a new class of heuristic scheduler that we call *global heuristic optimizer*. In this method no decision is made but the heuristic rule is used to affect the state of the resolver system. This system is constructed in such a way that it converges toward the optimal solution. Springplay algorithm is the first application of this global optimizer idea.

This paper is organized as follows. In section 2 we describe the model of the system for which the scheduling must be made. In section 3 we will present the principle of Springplay. In sections 4, 5, 6 and 7 we describe the heuristic rule (called *force system*) used in this algorithm. In section 8 Springplay is compared with a Branch&Bound class and with a Generalized List Scheduler class algorithm and in section 9 conclusions are drawn. At the end we present the schedule made by all the four algorithms on an example signal-flow graph to demonstrate the advantage offered by Springplay.

2. SYSTEM MODEL

The model used in this article of a problem to be solved and of the target hardware is the following :

- We suppose a problem which can be described by *static data flow* restricting the number of tokens consumed and produced by the operators to one. The real restriction that we impose is that we know the precise dependency graph. An extensive research has been made concerning the solution of *balance equations* [1] for the cases where multiple tokens are consumed/produced, using the methods presented in [18, 19] the firing order of the actors can be established, thus the precise dependency graph can be found and our system model can be extended to these cases as well.
- The precedence relations among the tasks can be represented as a directed acyclic graph.
- The target hardware system may contain heterogeneous processors. Processors can be programmed devices or synthesized circuits. The execution times of the tasks in the graph can be different for each processing units but they are known in advance and are constant.
- The communication links in the system may be heterogeneus as well but a fair estimate must be known in advance about the communication time of one data unit between each processor. It is supposed that there is no contention on the communication channels. If it is not the case, worst-case estimation is required.
- Communication takes time both at the sending and at the receiving side. This time can be an arbitrary function of the amount of data units communicated. The sending and receiving time required on the processors which take part in a given communication activity can be different. We suppose buffered asynchronous communication (nonblocking send, blocking receive).
- The amount of data units communicated among the tasks is constant and known in advance.
- The value to be minimized is the time at which all tasks are completed. Pipeline realizations are not considered in this article, but we envisage the extension of Springplay to support pipeline.

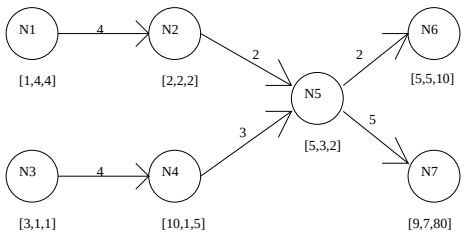


Fig. 1. A decorated acyclic precedence graph

For the graphic representation of the problem we use decorated acyclic precedence graph (DAPEG) in which edges (representing precedence relations) are decorated by the amount of data transferred through them and the execution time vector is attached to the nodes of the graph (representing tasks). The execution time vector is defined like the following :

$$\overline{t}_n^e \equiv [t_{n,1}^e, t_{n,2}^e, \ldots t_{n,P}^e] \ , \ 1 \leq n \leq N$$

where N is the number of the nodes in the graph, P is the number of the processors in the target hardware and $t_{n,p}^e$ is the execution time t^e of node n on processor p. Figure 1. shows an example DAPEG.

3. PRINCIPLES OF THE SPRINGPLAY ALGORITHM

In Springplay there is no decision making according to heuristic rules. Each node has a *state* which is an analog quantity. For clarity, we have chosen a geometric description form so node states are expressed as coordinates in a $P-1$ dimension coordinate system. The state of node i will be denoted by a $P-1$ dimensional vector $\vec{V}_n, 1 \leq n \leq N$. The processors are represented by similar points as well, we call points representing processors *fixpoints* denoted by $\vec{FP}_p$ where p is the processor number. The fixpoints are arranged in such a way that the distance between each point is 1, it is always possible to find P such points in a $P-1$ dimension coordinate system. The strongness of a node's membership to a certain processor is measured by the distance between the node state and the fixpoint of that processor. So the distance between

node n and processor p is :

$$d_{n,p} = \mid \vec{V}_n - \vec{FP}_p \mid$$

We use the following distance definition :

$$\mid \vec{a} - \vec{b} \mid = \sqrt{\sum_{i=1}^{P-1} (a_i - b_i)^2}$$

where a_i, b_i are the ith coordinates of vectors a, b, respectively. We call the processor whose fixpoint is the closest to the state of the nth node ($d_{n,p} = min(d_{n,p}), 1 \leq p \leq P$) *principal processor of node n* and it is denoted as p_n. Node states are influenced by *forces* which are created in such a way that they push and pull the node states to places which represent minimal-length schedule. This force system will be detailed later. The resulting force affecting node n is denoted $\vec{F}_n$ its component pointing to processor p will be referenced as $\vec{F}_{n,p}$. Figure 2. shows the arrangement for 3 processors and one node.

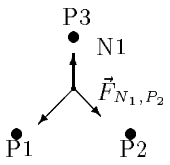


Fig. 2. Springplay arrangement of fixpoints, node state and forces

At the beginning all node states are initialized to the "center point" so that $\forall d_{n,i} = d_{n,j}, 1 \leq i, j \leq P, i \neq j$ then "naive schedule" is generated using a simple list scheduler. This schedule puts all the nodes to the first processor.

The algorithm maintains an actual schedule which is generated so that the nodes be put on their principal processor. Actual schedule is made by a simple list scheduler detailed later, at the beginning it is initialized to the naive schedule.

After this the node state modification phase starts. The algorithm takes the nodes one by one, calculates the actual value of the force exerted then the node state is updated according to

$$\Delta \vec{F}_n = \vec{F}_n \Delta t \tag{1}$$

$$\vec{V}_n = \vec{V}_n + \Delta \vec{F}_n \tag{2}$$

If the principal processor has changed because of the node state modification, the list scheduler is invoked. The modification of the principal processor changes the execution time of the node, the communication time and the parallelism.

The list scheduler in Springplay uses the actual principal processors to determine where the nodes should be scheduled and higher priority is given to nodes which can be started earlier.

$$SL = max(\text{finish time of all the predecessor nodes}) \tag{3}$$

The list scheduler simply gets the node with the lowest SL value (one is chosen arbitrarily if there is more of them) and schedules it on its principal processor. No communication activities are scheduled this time.

When one pass of node state modification is finishes - all the node states has been modified - Δt is normalized according to Eq. 4.

$$\Delta t = \frac{0.2}{max(\Delta \vec{F}_n)} \tag{4}$$

where $max(\Delta \vec{F}_n)$ is the maximum force modification calculated so far. The node state modification continues until any of the stopping condition is satisfied. As no stability property has been proved for the algorithm, we use strict stopping criteria at the beginning which is changed to more indulgent ones later and after a certain number of iterations the modification is stopped. At this point a final schedule is made which is the same list schedule as above but it schedules the communication activities as well. Recently send communication activities are inserted just after the node which generates the output to be sent and receive operations are put just before the nodes which need the value. This time we do not consider the possible hardware support of parallel computation and communication. This simple scheme will be refined in the future. The definition of SL (Eq. 3) is modified so that it considers now the communication time.

$$FSL = \text{earliest time when all the inputs are available} \tag{5}$$

This schedule is the output of the algorithm. Figure 15 shows the pseudocode of the entire algorithm.

The force system consists of 4 different components, each represents a certain property of the schedule.

$$\vec{F}_n = \vec{F}_n^T + \vec{F}_n^C + \vec{F}_n^P + \vec{F}_n^A \tag{6}$$

In the following sections each of these force components will be detailed.

4. EXECUTION TIME MINIMIZING COMPONENT

This component introduces a preference that the principal processor of a node be the one on which the node achieves the shortest execution time. For this purpose forces are created toward each fixpoint which mean "how strongly" the node would like to be on that processor. We define now a shorthand notation for the unit vector pointing from the node state to a processor fixpoint.

$$\vec{e}_{n,p} = \begin{cases} 0 & \text{if } \mid \vec{FP}_p - \vec{V}_n \mid = 0 \\ \dfrac{(\vec{FP}_p - \vec{V}_n)}{\mid \vec{FP}_p - \vec{V}_n \mid} & \text{else} \end{cases}$$

The force affecting node state n from processor p is :

$$\vec{F}^T_{n,p} = D^T_{n,p}\vec{e}_{n,p} \tag{7}$$

where $D^T_{n,p}$ is the coefficient and it is multiplied by the unit vector toward the processor.

$$D^T_{n,p} = \sum_{i=1, i\neq p}^{P} t^e_{n,i} \tag{8}$$

The $D^T_{n,p}$ coefficient is defined in a bit complicated way so that it have time dimension. It is the sum of all the components in the execution time vector but the execution time on the processor for which it is calculated which gives the result that the coefficient of the processor on which the node achieves the smallest execution time will be the biggest. This will try to pull the node state to that processor. The final force from this component is the sum of all forces with the same n index.

$$\vec{F}^T_n = \sum_{p=1}^{P} \vec{F}^T_{n,p} \tag{9}$$

5. COMMUNICATION COST COMPONENT

This term adds forces which try to influence the actual schedule so that the communication cost be minimal. Let C_n denote the number of nodes connected - both inputs and outputs - to node n and $N_{n,i}$ will stand for the node with connection number i connected to node n. We will use $C_{n,i}$ to refer to the ith connection of node n. We will use the following shorthand notation : $T_{n,p,i}$ is the time required for communication between processor p if node n is scheduled on it and the principal processor of node $N_{n,i}$. If $N_{n,i}$ consumes the result of node n $T_{n,i}$ is the time of the send activity on p_n, else it is equal to the time necessary for receiving the output of $N_{n,i}$ from its principal processor. An example for explaining the notations can be seen in figure 3.

The communication forces are defined similarly to F^T. For each connection of node n forces are added

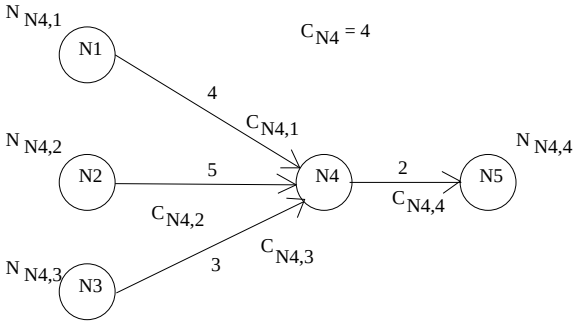


Fig. 3. Communication cost notations

toward each processor which symbolize how necessary it would be to put the node onto that processor because of the communication cost with its neighbours. $F^C_{n,C_{n,i},p}$ is the force caused by connection $C_{n,i}$ toward processor p.

$$\vec{F}^C_{n,C_{n,i},p} = D^C_{n,C_{n,i},p}\vec{e}_{n,p} \tag{10}$$

$$D^C_{n,C_{n,i},p} = \sum_{j=1, j\neq p}^{P} T_{n,j,i} \tag{11}$$

$$\vec{F}^C_n = \sum_{i=1}^{C_n}\sum_{p=1}^{P} \vec{F}^C_{n,C_{n,i},p} \tag{12}$$

The meaning of $D^C_{n,C_{n,i},p}$ in Eq. 11 is similar to the constant defined in Eq. 8, its value is bigger if the communication cost to processor p is smaller.

6. PARALLELISM OPTIMIZATION

The two previous terms assure that nodes tend to be placed on processors on which they achieve the lowest execution time while minimizing communication costs. To get Springplay to strive toward solutions where the total execution time is minimized by exploiting parallelism in the algorithm to be scheduled we introduce a new term. First we define our notion of parallelism of node n. In our definition parallelism is the sum of the occupied time of all the other processors in the time frame of node n. This value is used to express if there is a possible amelioration of the actual schedule by exploiting the possible parallelism better. Figure 4 illustrates the definition. In the following we will denote this parallelism quantity calculated for node n in a schedule S as $t^{par}_{n,S}$. Moreover, we define the amount of parallelism of node n with respect to processor p as the occupied time on processor p in the time frame of node n and we will denote this quantity as $t^{par}_{n,p,S}$. For example $t^{par}_{N1,P3,S} = 2$ in figure 4.

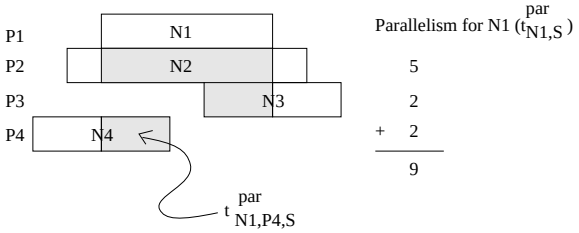


Fig. 4. Measuring parallelism in the graph

The force calculation is based on the "ideal parallelism" measured in ASAP schedule. The algorithm makes an ASAP schedule first using the $t_n^{asap} = min(\overline{t}_n^e)$ as node execution time then calculates the parallelism in this schedule. Ideal parallelism is defined as the following :

$$t_n^{ipar} = \begin{cases} t_{n,ASAP}^{par} & \text{if } t_{n,ASAP}^{par} \leq P \cdot t_n^{asap} \\ P \cdot t_n^{asap} & \text{otherwise} \end{cases} \quad (13)$$

Eq. 13 says that the ideal parallelism is measured in the ASAP schedule but limited to the amount of parallelism that can be exploited in the target hardware. This limitation is imposed by the number of processors (P) available. ASAP schedule supposes unlimited processors so it can use more processors than available, hence the limitation.

When calculating F_n^P force the parallelism in the actual schedule is calculated and compared with t_n^{ipar}. If the actual parallelism is bigger than a givent percent (called *exploitation factor*, η_e) of the ideal one, we strengthen the node's membership to its principal processor, else we introduce forces which pull the node state to other processors. The value of the exploitation factor determines, how good parallelism exploitation will be accepted. The closer it is to 1, the better schedules the algorithm produce but its stability deteriorate. We found that $\eta_e = 0.9$ is a good compromise between stability and schedule quality.

if $t_{n,ACT}^{par} > \eta_e t_n^{ipar}$

$$\vec{F}_n^p = (t_{n,ACT}^{par} - t_n^{ipar})\vec{e}_{n,p_n} \quad (14)$$

otherwise

$$\vec{F}_n^p = \sum_{p=1}^{P} (t_n^{ipar} - t_{n,ACT}^{par})\vec{e}_{n,p}(\frac{t_{n,p}^e - t_{n,p,ACT}^{lpar}}{t_{n,p}^e}) \quad (15)$$

$$t_{n,p,ACT}^{lpar} = \begin{cases} t_{n,p,ACT}^{par} & \text{if } t_{n,p,ACT}^{par} \leq t_{n,p}^e \\ t_{n,p}^e & \text{else} \end{cases} \quad (16)$$

Eq. 14,15 express that depending on the parallelism exploited the node state is pulled to p_n or pushed to other processors. This latter force depends on how much time is available in the given time window on other processors, it is 0 if the time window has already been filled by other tasks and bigger if there is unoccupied time on that processor. In Eq. 16 we limit the time we demand on other processors to the execution time of node n on that processor.

7. ANCHORING NODES

Race situations can exist in some cases when the schedule alternatives are equally good. Let us imagine for example one task and 2 processors and equal task execution times on both processors, in this case we are free to schedule the task on any processor. Race situations cause instability in Springplay. To prevent this, F^A component is added which introduces slight preference toward the principal processor. The amount of this preference is controlled by the *anchoring factor* (η_a).

$$\vec{F}_n^A = \vec{e}_{n,p}(\eta_a min(\overline{t}_n^e)) \quad (17)$$

During the experiments $\eta_a = 0.1$ gave good stabilizing effect without deteriorating the quality of the final schedule.

8. RESULTS

We made comparative tests between Springplay and two other algorithms : a Branch&Bound type algorithm described in [7] and the DLS algorithm [12] which is a Generalized List Scheduler class method. We chose these two ones because of their support of heterogeneous architectures which seems to be a less-investigated topic according to our bibliography research. Further problem was that the system model of these two algorithm was different to ours regarding the cost of communication : [7] considers no communication cost while [12] supposes dedicated communication hardware. We forced our - more realistic - system model to these algorithms which aggravated the runtime problems of the B&B method ameliorating the quality of the solutions it produced at the same time. In the case of the DLS the differences of the system models have less importance. In the following we use BBOPT and BBH2 to refer to the Branch&Bound algorithm with two types of heuristic functions in [7] and DLS to denote the method presented in [12].

The prototypes of the algorithms were realized in LISP. To counterbalance the slow execution speed of LISP we allowed a generous amount of runtime before terminating the computation. This caused problems only for the B&B type algorithms, the exponential growth of computation requirement resulted in inacceptable run times over 25 nodes. For this reason more graphs were generated with smaller node num-

alg/SP results(%)	worst	best	average
BBOPT	-44.12	38.46	3.39
BBH2	-37.75	69.44	9.7
DLS	-32	137.5	46.8

Table 1: Performance comparation of Springplay with the reference algorithms

bers and less processors.

The algorithms were tested with 80 randomly generated data-flow graphs where the number of the nodes was between 16 and 50 and the node execution times were in the [1,11] interval, the number of the processors was 4 and 5. Instead of randomly generating the communication costs we used four communication models: a totally interconnected model with small cost, the same with heavy cost and a chain-like model (each processor is connected only with two neighbours) with small and heavy communication costs.

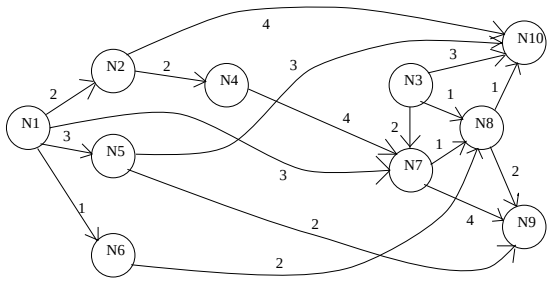
As all the methods tested use some kind of heuristic, their results show a significant variance, "good cases" (where the heuristic rule matches well to the problem) and "bad cases" can be found for any of the four algorithms. For this reason, we present our results in the form of histograms and express the performance of the algorithms by means of average. The histograms 12, 13, 14 compare an algorithm pair by calculating the ratio

$$\frac{t_{sched,alg1}}{t_{sched,alg2}} 100 - 100\%$$

of schedule lengths produced for the same input graph and by representing the distribution of this expression for the ensemble. There are less samples in the case of BBOPT and BBH2 ensembles because they could not terminate the computation for bigger node numbers. It must be noted that in spite of claims in [7] BBOPT heuristic function can significantly overestimate the finish time of a partial schedule thus it cannot always find the optimal solution. This is the reason why Springplay could produce better results than BBOPT in numerous cases.

In table 1 we compare the worst, the best and the average performance ratio for each pair. The advantage of Springplay against BBOPT is 3.4%, against BBH2 it is 9.7%. The advantage is more significant against DLS : the method presented here is 46.8% better and there is an important number of cases where Springplay is 80-100% better.

The significant performance advantage of Springplay is demonstrated on an example signal-flow graph (Figure 5.). The graph was generated randomly. Figures 6,7,8 and 9 shows the schedules generated by BBOPT, BBH2, DLS and Springplay, respectively.



Execution time vectors :

N1 : [6,9,3,9]
N2 : [7,1,5,1]
N3 : [9,3,8,9]
N4 : [11,7,2,5]
N5 : [9,3,4,8]
N6 : [8,7,8,3]
N7 : [8,4,11,8]
N8 : [9,1,7,4]
N9 : [10,9,1,5]
N10 : [10,6,6,7]

Fig. 5. The example DAPEG

- Springplay does not schedule the longest path on one processor. The "critical path" techniques loose much of their efficiency on heterogeneous environments.
- We supposed asynchronous buffered communication model in this article. It means that send activities are guaranteed to be scheduled before or paralelly with corresponding receive operations but no other restrictions are imposed.
- Springplay not only generated the shortest schedule but used less processors than BBOPT and BBH2.

Figures 10 and 11 shows the convergence process in two cases. For generating these plots the original algorithm was slightly modified, after each iteration the final schedule routine was called and the length of this schedule is shown on the diagrams. This modification does not affect the parallelism optimization forces, they are calculated from the actual schedule generated by the internal list scheduler as before. Figure 10 shows the convergence in the case of the example DAPEG. The algorithm finds the solution in one step, the following steps are generated to trigger the stopping condition. In figure 11 the schedule length changes are shown in the case of a 20 node DAPEG chosen from the test set where a limit cycle can be observed at the end of the convergence process. This shows that the heuristic rule does not guarantee that the schedule length will not increase. The simple stopping conditions used during the tests does not consider this fact, we intend to develop more exact stopping criterias.

9. CONCLUSIONS

We have presented a new scheduling method which try to overcome the problem imposed by the local decision making in heuristic algorithms. An applic-

ation of this idea was investigated and the results are promising. During the experiments Springplay produced slightly better solutions in $O(N^3)$ computational complexity as B&B in exponentional complexity. The tests revealed the advantage of global optimization over local decision making as well. There are still several important questions to answer : stability, convergence to solution and local minimas. Despite its good performance in average, there were few cases when Springplay produced bad quality results. We intend to examine thoroughly these cases, refine the heuristic rule and the structure of the resolver system. Our current research activity concerning Springplay covers the pipeline scheduling as well.

APPENDIX : FIXPOINT GENERATION

Generating fixpoints needs solving a simple geometrical problem. Given a $P-1$ dimension space, we want P points so that the distance between each pair is 1. The following simple algorithm can generate the solution from the $P-1$ point case.

Let us denote the coordinates as elements of a $P-1$ dimension vector : $[c_1, c_2 ..., c_{P-1}]$ and $c_{i,j}$ will mean the ith coordinate of the jth point.

When we generate the new point we exploit the fact that the $P-1$ point problem could be solved in $P-2$ dimension so the $P-1$th coordinate of the points coming from the $P-1$ point solution is necessarily 0. Now we have to calculate a Pth point whose distance from all the old points is 1. This is done by generating a set of well-known distance equations for the new point.

$$\begin{aligned}&(c_{1,P}-c_{1,p})^2+(c_{2,P}-c_{2,p})^2+...+\\&(c_{P-1,P}-c_{P-1,p})^2=1, 1\le p\le P-1\end{aligned} \tag{18}$$

Note that $c_{P-1,p}\equiv 0$. Systematically subtracting the $2\le p\le P-1$ members of this equation set from the $p=1$ element we get

$$\frac{\sum_{i=1}^{P-2}(c_{i,p}^2-c_{i,1}^2)}{2}=\sum_{i=1}^{P-2}(c_{i,p}-c_{i,1})c_{i,P}, 2\le p\le P-1 \tag{19}$$

which gives us $P-2$ linear equations for $c_{i,P}, 1\le i\le P-2$. The missing $c_{P-1,P}$ coordinate can be yielded by substituting the known $c_{i,P}$ components into Eq. 18.

REFERENCES

1 E. A. Lee and D. Messerschmitt, "Static scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol C-36, pp. 24-35, No. 1, January 1987.

2 J. Buck, S. Ha, E. A. Lee and D. Messerschmitt, "Multirate Signal Processing in Ptolemy," *Proc. IEEE ICASSP-91,* pp. 1245-1248, Toronto, Canada, April 1991.

3 V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, MA: MIT Press, 1989.

4 J. H. Lee, Y. C. Hsu and Y. L. Lin, "A New Integer Linear Programming Formulation For The Scheduling Problem in Data Path Synthesis," *Proc. IEEE ICCAD-89,* pp. 20-23, Santa Clara, California, November 1989.

5 D. Gajski, N. Dutt, A. Wu and Steve Lin, *High-Level Synthesis.* Kluwer Academic Publishers, 1992.

6 C. T. Hwang and Y. C. Hsu, "Zone Scheduling," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, No. 7, pp. 926-934, July 1993.

7 B. Greenblatt and C. Linn, "Branch & Bound Style Algorithms for Scheduling Communicating Tasks in a Distributed System," *Proc. IEEE, Compcon-87,* pp. 12-17, San Francisco, February 1987.

8 T. C. Hu "Parallel sequencing and assembly line problems," *Oper. Res.*, vol. 9, No. 6, pp. 841-848, Nov. 1961.

9 G. Mirchandani and D. G. Ogden, "Experiments in Partitioning and Scheduling Signal Processing Algorithms for Parallel Processing," *Proc. IEEE ICASSP-88,* pp. 1690-1693, New York, N.Y. April 1988.

10 P. G. Paulin and J. P. Knight "Force-directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Trans. on Computer-Aided Design*, vol 8, No. 6, pp. 661-679, June 1989.

11 P. D. Hoang and J. M. Rabaey, "Scheduling of DSP programs onto Multiprocessors for Maximum Throughput," *IEEE Trans. on Signal Processing*, vol 41, No. 6, pp. 2225-2235, June 1993.

12 G. C. Sih and E. A. Lee "A Compile-Time Scheduling Heuristic for Interconnection- Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, No 2, pp. 175-187, February 1993.

13 G. C. Sih and E. A. Lee, "Declustering : A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, No. 6, pp. 625-637, June 1993.

14 J. J. Hopfield and D. W. Tank, "Neural computation of Decisions in Optimization Problems," *Biol. Cybern.* 52,141 (1985).

15 B. Kamgar-Parsi and B. Kamgar-Parsi, "An Efficient Model of Neural Networks for Optimization," *Proc. IEEE ICNN-87,* Vol. 3, pp. 785-789, San Diego, California, June 1987.

16 M. Mittler and P. Tran-Gia, "Performance of a Neural Net Scheduler used in Packet Switching Interconnection Networks," *IEEE ICNN-93,* pp. 695-700, San Francisco, California, March 1993.

17 L. Rabelo, Y. Yih, A. Jones and G. Witzgall, "Intelligent FMS Scheduling Using Modular Neural Networks," *IEEE ICNN-93,* San Francisco, California, pp. 1224-1229, March 1993.

18 G. R. Gao, R. Govindarajan, P. Panangaden, "Well-behaved Dataflow Programs for DSP Computation," *Proc. IEEE ICASSP-92,* Vol. 5, pp. 561-564, San Francisco, California, March 1992.

19 J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token-flow model," *Proc. IEEE ICASSP-93,* Minneapolis, April 1993.

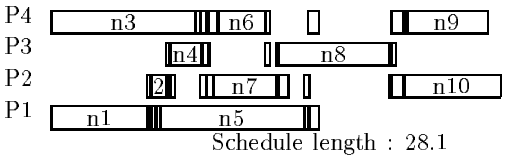


Fig. 6. Example schedule by Branch&Bound OPT

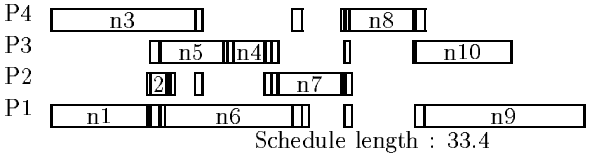


Fig. 7. Example schedule by Branch&Bound H2

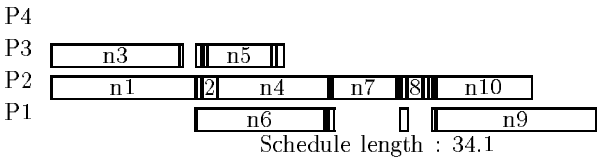


Fig. 8. Example schedule by DLS

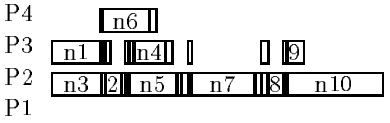


Fig. 9. Example schedule by Springplay

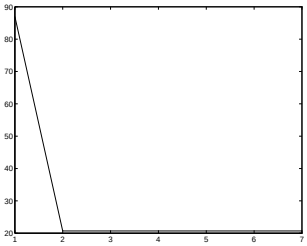


Fig. 10. Convergence in the example case

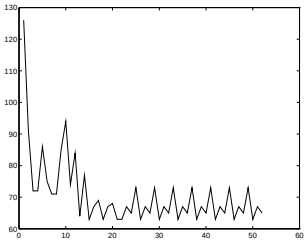


Fig. 11. Convergence in a 20 node case

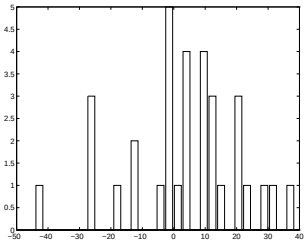


Fig. 12. Distribution of BBOPT/SP results, 33 samples

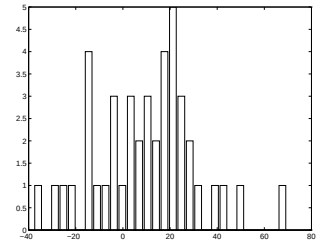


Fig. 13. Distribution of BBH2/SP results, 43 samples

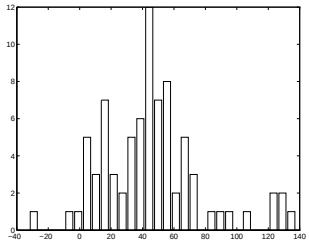


Fig 14. Distribution of DLS/SP results, 80 samples

```
procedure SpringPlay
begin
  Initialize all node principal processors
  to the 1st processor;
  Initialize all node states to the center point;
  List Schedule;
  Limit1 := 1.5 * number of nodes;
  Limit2 := 2.5 * number of nodes;
  dt := 0.001;
  maxdF := 0;
  ActLimit := Limit1;
  IterationWithoutChanges := 0;
  IterationCount := 0;
  do
    ChangeList := empty list
    for all nodes do
      F:= Calculate force;
      dF := F * dt;
      Modify node state;
      if principal processor changed then
        List Schedule;
        add changing to ChangeList;
      endif
      if dF > maxdF then
        maxdF := dF
      endif
    endfor
    Update dt according to maxdF;
    Increment IterationCount;
    if ChangeList == empty list then
      Increment IterationWithoutChanges;
    else
      IterationWithoutChanges := 0;
    endif;
    if IterationCount <= Limit1
      ActLimit := 5;
    elseif IterationCount <= Limit2
      ActLimit := 1;
    else
      ActLimit := 0;
    endif;
  until ( IterationWithoutChanges > ActLimit );
  Final Schedule
end;
```

Fig 15. Pseudocode listing of Springplay