Exploration of Schedule Space by Random Walk

Liangwei Ge, Song Chen, Takeshi Yoshimura

Graduate School of Information, Production and System, Waseda University
Yoshimura Lab, 2-7 Hibikino Wakamatsu-ku, Kitakyushu 808-0135, Japan
Tel & Fax: +81-93-692-5364
Email: liangwei_ge@ruri.waseda.jp

Abstract
Scheduling, an important step in high-level synthesis, is a searching process in the solution space. Due to the vastness of solution space, it is usually difficult to search schedules efficiently. In this paper, we present a random walk based perturbation method to explore the schedule space. The method first limits the search within a specific sub-solution space (SSS). Then, the SSS is repeatedly perturbed by using an N-dimension random walk so that better schedules can be searched in the new SSS. To improve the search efficiency, a guided perturbation strategy is presented that leads the random walk toward promising directions. Experiment shows that in runtime comparable to existing methods, ours finds schedules of better quality.

Keywords
Scheduling, high-level synthesis, random walk.

1. Introduction
In high-level synthesis, the scheduling of multiple operations to appropriate control steps in the presence of a set of constraints is an essential but intractable task [1][2], which influences the synthesized circuit on speed, delay, power consumption, etc. Recently, scheduling algorithms based on bipartite graph matching have been proposed [3]-[5], which have the advantage of optimizing complex objectives with more flexibility. [3][4] used an exact method that had unacceptable runtime on large cases. [5] proposed an efficient heuristic method, which however restricts the solution space to a sub-solution space. In this paper, we improve the works of [5] by proposing the random walk based max-flow scheduling. Like [5], the proposed method searches schedules in the sub-solution space (SSS). However, we incorporate random optimization technique into scheduling. If the optimum schedule of SSS is unsatisfactory, current SSS is perturbed and better schedules are searched in the new SSS. To improve the efficiency of exploring the solution space, we also present a guided perturbation strategy that leads the random walk toward promising directions. Experimental results show that in runtime comparable to existing methods, ours finds schedules of better quality.

2. Sub-Solution Space and Max-Flow Scheduling
2.1 Problem Formulation
Given an operation set of t types $V = \{V_j \mid j = 1, 2, \ldots, t\}$ where each type has $n(j)$ operations $V_j = \{v_{ji} \mid i = 1, 2, \ldots, n(j)\}$, the scheduling task is to find integer labels of all the operations CS: $V \rightarrow Z^+$, where $CS(v_{ji}) \in Z^+$ is the control step that operation v_{ji} is scheduled to. The scheduling task can be formulated as a matching problem between the operations and the control steps on bipartite graphs. We denote the bipartite graph of type j by $BG_j(V_j, S_j, A_j)$, where V_j is the operation set of type j, S_j the control step interval of type j, and $A_j \in V_j \times S_j$ the edge set. There is an edge $e(v, s) \in A_j$ if v can be potentially scheduled in s.

Two kinds of constraints are considered: the resource constraint and the dependency constraint. Typically, the dependency constraint is represented by a polar, directed, acyclic graph $G(V, E)$, called *data flow graph* [5]. The edge $(v_{ix}, v_{jy}) \in E$ defines the dependency constraint of $CS(v_{jy}) \geq CS(v_{ix}) + D(i)$, where $D(i)$ is the delay of the operation type of v_{ix} in term of clock cycles. The resource constraint, $\{FU(j) \mid j = 1, 2, \ldots, t\}$, specifies the number of functional units available for each operation type j.

Definition 2.1 (operation freedom). *The freedom of an operation v, FR(v), is an integer set* $\{i \mid CS_e(v) \leq i \leq CS_l(v)\}$, *where* $CS_e(v)$ *and* $CS_l(v)$ *are the earliest and the latest starting control steps of v without violating any dependency constraint. FR(v) is also equivalently represented by* $[CS_e(v), CS_l(v)]$.

The operation freedom can be calculated by the *as soon as possible* (ASAP) and the *as late as possible* (ALAP) scheduling. Fig. 1 shows a data flow graph (DFG) example with the operation freedom calculated. In this example, we assume multiplication as type 1, addition as type 2, $FU(1) = FU(2) = 2$, $D(1) = D(2) = 1$, and the schedule latency $CS(sink)$ denoted by L.

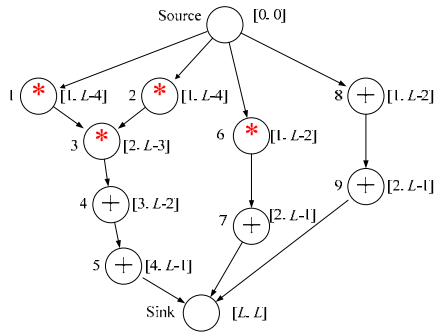

Fig. 1. A DFG example with operation freedom calculated.

Algorithm 1: Calculation of control step interval S_j

1. for ($i = 1$; $i \leq n(j)$; i++)
   $g_{ji} = CS_e(v_{ji})$; $h_{ji} = CS_l(v_{ji})$;
2. Sort $\{g_{ji}\}$ and $\{h_{ji}\}$ in increasing order;
3. for($i = FU(j)+1$; $i \leq n(j)$; i++) // limit earliest request
   $g_{ji} = \max(g_{ji}, g_{j(i - FU(j))} + D(j))$;
4. for($i = n(j) - FU(j)$; $i \geq 1$; i--) // limit latest request
   $h_{ji} = \min(h_{ji}, h_{j(i + FU(j))} - D(j))$;

Algorithm 1 shows how to calculate the control step interval S_j. The control step interval of type j, $S_j = \{[g_{ji}, h_{ji}] \mid i = 1, 2, \ldots, n(j)\}$, firstly records the $n(j)$ ASAP and ALAP scheduling requests respectively. Then, some earliest/latest scheduling requests are

delayed/advanced, based on the resources available. $\{g_{ji}\}$ and $\{h_{ji}\}$ are two arrays of integers, with $\{h_{ji}\}$ containing the schedule latency L. Obviously, for any interval $[g_{ji}, h_{ji}]$, $g_{ji} \le h_{ji}$. This gives an estimation of the schedule latency L.

Once the schedule latency L, the freedom of V_j, and the control step interval S_j are known, the bipartite graph of type j, $BG_j(V_j, S_j, A_j)$, can be built. There is an edge $e(v, s) \in A_j$ if and only if the freedom of v has overlap with interval s. Thus, the scheduling problem can be formulated as finding perfect matching between the operations V_j and the control step intervals S_j in $BG_j(V_j, S_j, A_j)$. Fig. 2 shows the bipartite graphs of the example DFG under the schedule latency of ($L = 5$).

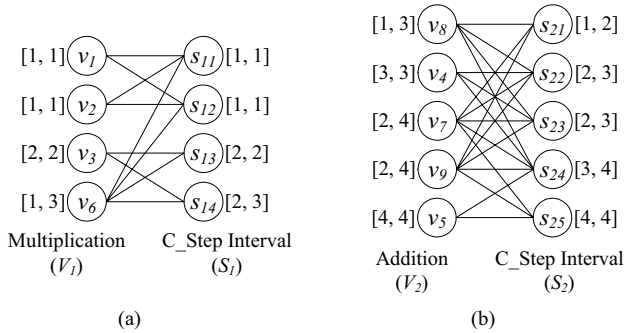

Fig. 2. Bipartite graphs. (a) $BG_1(V_1, S_1, A_1)$. (b) $BG_2(V_2, S_2, A_2)$.

2.2 Resource and Dependency Constraints

The matching problem can be solved in polynomial time by the max-flow algorithm [6]. Schedules based on a perfect matching ensure each operation a control step interval, which implicitly guarantees the resource for the matched operation. However, the dependency constraint cannot be satisfied easily, because in the bipartite graph the operations of V_j are assumed independent of each other.

Fig. 3(a) shows an example of finding a perfect matching in $BG_2(V_2, S_2, A_2)$ by the max-flow algorithm, as indicated by the thick edges. Fig. 3(b) shows the corresponding schedule. Obviously, the resource constraint is satisfied but the dependency of (v_8, v_9) is violated.

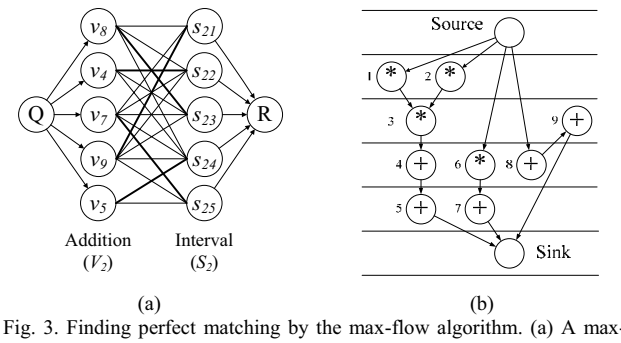

Fig. 3. Finding perfect matching by the max-flow algorithm. (a) A max-flow network. (b) A schedule that violates the dependency constraint.

Fig. 4(a) shows the freedom distribution of Fig. 1. Each rectangle stands for an operation with the vertical span representing its freedom. The grey area indicates the *overlapped freedom* [5]. The overlapped freedom between two operations represents the possibility that the dependency between them be violated. This possibility can be eliminated by *partitioning* the overlap [5]. Fig. 4(b) shows one possible partition of the overlaps.

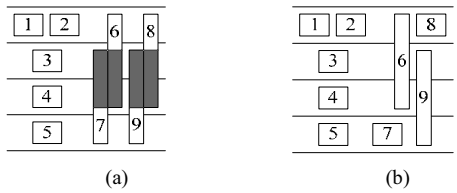

Fig. 4. Freedom distribution in the DFG example. (a) With overlap. (b) Overlap removed through partition.

Definition 2.3 (overlap partition). *The partition of an overlapped operation pair (v_{ix}, v_{jy}) is decreasing the freedom of v_{ix} and v_{jy} to $[CS_e(v_{ix}), k]$ and $[k + D(i), CS_l(v_{jy})]$ respectively, where k is an integer between ($CS_e(v_{jy}) - D(i)$) and $CS_l(v_{ix})$.*

Overlap partition decreases the freedom of the operations involved, which reduces the edge set A_j of $BG_j(V_j, S_j, A_j)$. Thus, the original solution space is pruned into a sub-solution space (SSS). Within SSS, every perfect matching corresponds to schedules that satisfy the resource and the dependency constraints.

2.3 Enhanced Max-Flow Scheduling Algorithm

Resource-constrained scheduling is an NP-complete problem [7]. Therefore, it is difficult to optimally partition all overlapped pairs. In this study, a force-directed heuristic partition method [5] is used. The max-flow scheduling algorithm under given schedule latency is described as follows:

Algorithm 2: Max-flow scheduling under given latency:

1. Set $CS(sink)$ at the given schedule latency L;
2. Calculate the freedom of the operations of V_j under L;
3. Calculate the control step interval S_j under the resource constraint and the schedule latency L;
4. Partition all operation overlaps by the force-directed heuristic method [5];
5. Build the bipartite graphs $BG_j(V_j, S_j, A_j)$, $1 \le j \le t$;
6. Perform the max-flow algorithm on the bipartite graphs to find perfect matching;
7. If perfect matching exists, calculate the corresponding schedule;

3. Perturbation of SSS by Random Walk

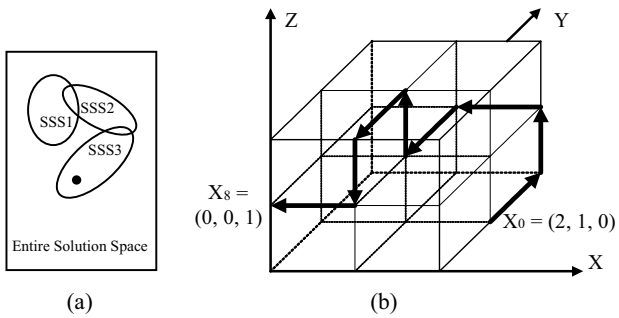

Fig. 5. Exploring schedule space. (a) SSS perturbation. (b) SSS perturbation as random walk on graph G_{SSS}.

Since the original solution space is heuristically pruned to sub-solution space (SSS) [5], the global optimum schedule may not be in the SSS. It is thus desired to perturb current SSS and search schedules in new SSS. Fig. 5(a) illustrates the perturbation of

SSS: the dot stands for an acceptable schedule. SSS1 and SSS2 contain no schedules. So, perturbation goes on. Finally, SSS3 covers the schedule and the max-flow scheduling method assures finding it out.

3.1 Formulating SSS Perturbation as Random Walk

Assume there are N overlapped operation pairs to be partitioned: $p_1, p_2, \ldots, p_N$. The i-th pair p_i has overlap of $(k_i - 1)$ control steps. Then, there are k_i different partitions of p_i. Define the SSS state space X, in which each state $x \in X$ represents a SSS, or one partition scheme of the N overlapped pairs. x is encoded as: $(u_1, u_2, \ldots, u_N)$. The i-th dimension u_i, $1 \le i \le N$, takes the value among $\{0, 1, 2, \ldots, k_i - 1\}$ that stands for the k_i different partitions of p_i. Apparently, there are $\prod_{i=1}^{N} k_i$ states (SSSs) in X.

Next, we define the G_{SSS} graph as a set of vertices X, equipped with a symmetric neighborhood relation (a subset of $X \times X$). Vertices $x, y \in X$ are neighbors if and only if the Euclidean distance between x and y is one. The degree of vertex x, deg(x), is its number of neighbors. Since the G_{SSS} is an N-dimension lattice, each vertex of X (except the ones on G_{SSS} boundary) has $2N$ neighbors. The SSS perturbation can be formulated as an N-dimension random walk [8][9] on G_{SSS}, which is a sequence of X-valued random variables $\{X_t: t = 0, 1, 2, \ldots\}$ such that X_i neighbors X_{i-1}. X_t represents the random position in X at time t. X_0 is the starting state obtained by the force-directed overlap partition heuristic. The walk ends at $t = n$, when X_n is the first visited state that contains schedules of acceptable quality. Random walk can be described by a discrete-time irreducible Markov chain (X, P), where X is the defined state space and $P = (p(x, y))_{x, y \in X}$ is the stochastic transition matrix [8].

Fig. 5(b) shows an example of random walk on a $2 \times 2 \times 2$ G_{SSS}. (2, 1, 0) is the initial overlap partition scheme. After 8 perturbations, (0, 0, 1), the desired partition scheme, is reached.

3.2 Improve Perturbation by Guided Random Walk

The transition matrix P decides how the solution space is explored. Here, we present guided random walk to improve the perturbation efficiency by using the schedule search result in current SSS.

Overlap partition essentially allocates the overlapped freedom to the involved operations. Improper partition gives some operations too much freedom, while overpruning others. In the max-flow network, like Fig. 3(a), an overpruned operation v doesn't have enough outgoing edges to the control step intervals S, which cannot find a path to node R. Thus, by finding operations that cannot send flow to R in the max-flow algorithm, we obtain operations whose freedom is quite possibly overpruned. With this information, the SSS perturbation can be improved by expanding the freedom of the overpruned operation by one control step rather than perturbing a randomly selected pair. This heuristic technique reduces the chance of perturbing a correctly partitioned pair and is more likely the direction toward SSSs that contain schedules.

Consider the DFG in Fig. 6(a). Operation o1 and o2 (in grey) are additions. Others are multiplications. Two adders and two multipliers are available. Obviously, no schedule exists under such overlap partition, as o15 ~ o18 are congested in control step 4. Assume o15 is the operation that causes the failure of finding a perfect matching. SSS can then be perturbed by expanding o15 (rather than a random operation). Assume FR(o15) expands from [4, 4] to [3, 4]. Meanwhile, the related ancestors {o1, o13, o14} found by upward topological search are pushed upward as well. The rise of o1 further enables o16 be expanded to [3, 4]. The new SSS (partition), shown in Fig. 6(b), apparently contains schedules. By utilizing the max-flow algorithm feedback, a feasible SSS is reached within one perturbation.

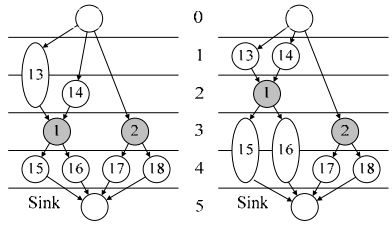

Fig. 6. Perturbation using max-flow feedback. (a) SSS before perturbation. (b) SSS after perturbation.

Finally, we give the random walk based max-flow scheduling algorithm:

Algorithm 3: Random walk based max-flow scheduling:

```
Initialize schedule latency;
Perform ASAP and ALAP scheduling;
go_on = true
while(go_on)
{
    Calculate control step interval S under latency;
    Calculate freedom of V under latency;
    Partition overlapped freedom [5];
    Build bipartite graphs BG_j(V_j, S_j, A_j), 1 ≤ j ≤ t;
    step = 0; acceptable = false;
    while(!acceptable && (step < limit))
    {
        step = step + 1;
        Perform max-flow scheduling by Algorithm 2;
        if(schedule exists)
            if(quality is good)
                acceptable = true;
            else
                Perturb SSS randomly;
        else
            Perturb SSS by the guided random walk;
    }
    if(step ≥ limit)
        go_on = false;
}
```

4. Experiment

Table 1 DFGs Used in Experiments

DFG	Description	Node #	Edge #	Resources
mpeg	Motion compensation of MPEG decoding	53	112	1 mult, 1 ALU, 1 comp, 1 shift
adpcm_en	ADPCM audio encoding	53	141	2 mult, 2 add
adpcm_de	ADPCM audio decoding	46	113	2 mult, 1 add
fft	Fast Fourier Transform	129	240	1 mult, 2 add, 2 sub
ar	AR lattice filter	28	40	2 mult, 1 add
elliptic	Fifth-order elliptical filter	34	65	3 mult, 2 add
different	Differential equation	11	16	1 mult, 1 add

All the experiments are carried out by a Pentium 4 CPU of 2.4GHz. The DFGs used are mainly extracted from the C programs in the MediaBench benchmark suite [10]. Table 1 lists the details of the DFGs and the resource constraint. 'mult' has a delay of 2 clock cycles. Other functional units have unit operation delay.

4.1 Experiment 1: Importance of Perturbation and Max-Flow Feedback

Experiment 1 is designed to illustrate the necessity of SSS perturbation and the effectiveness of the max-flow algorithm feedback to improve perturbation. The guided perturbation utilizes the feedback, while the equalized perturbation uses random perturbation. Each DFG is tested 3 times and the number of perturbations performed to find the optimum schedule (of the minimum latency) is listed respectively. If the optimum is not found within 1400 perturbations, it is marked with 'F'. Table 2 shows that 4 DFGs out of 7 needn't perturbation, which means the force-directed overlap partition heuristic [5] correctly prunes the solution space with the global optimum schedule contained in the initial SSS. For the other DFGs (43%), however, perturbation must be performed in order to obtain the optimum schedule. This ratio justifies our effort to integrate random walk into the max-flow scheduling. Moreover, for the 3 DFGs that require perturbation, 'Random perturbation' needs much more perturbations. This result confirms our claim that feedback from the max-flow algorithm improves perturbation efficiency.

Table 2 Hitting Time with/without Using Max-Flow Feedback

DFG	Guided perturbation	Random perturbation
mpeg	0/0/0	0/0/0
adpcm_en	0/0/0	0/0/0
adpcm_de	18/306/15	F/F/F
fft	0/0/0	0/0/0
ar	97/103/166	1312/F/F
elliptic	37/261/15	181/651/717
different	0/0/0	0/0/0

4.2 Experiment 2: Comparison of Runtime

Table 3 compares the runtime of our method with the list scheduling and the exact scheduling using ILP [11]. Each DFG is run 3 times and the averaged runtime is listed. As expected, runtime of the ILP solver increases sharply as the DFG grows larger. In the FFT case, even tens of hours are required to find the optimum due to the equation explosion. This explains why exact method is seldom used in practice. For our proposed method, all DFGs are finished within 0.5 second, which is slightly slower than list scheduling due to the exploration in the solution space. There are several ways to shorten the runtime. Improving the lower bound estimation of the schedule latency is one approach. Decreasing the max perturbation number is another, however, at the cost of possibly deteriorating the schedule quality. Therefore, by controlling the perturbation number, tradeoff between schedule quality and runtime can be conveniently made.

Table 3 Comparison of Runtimes (second)

DFG	List	Random walk based max-flow scheduling	ILP
mpeg	0.002	0.14	2.9
adpcm_en	0.002	0.003	0.28
adpcm_de	0.001	0.41	0.59
fft	0.005	0.013	43654
ar	0.001	0.23	1.1
elliptic	0.001	0.12	0.17
different	0.001	0.001	0.15

References

[1] M. C. Mcfarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Porceddings of the IEEE*, vol. 78, no. 2, pp. 301-318, 1990.

[2] Y. Lin, "Recent developments in high-level synthesis," *ACM Trans. Design Automation of Electronic Systems*, vol. 2, no. 1, pp. 2-21, 1997.

[3] A. H. Timmer and J. A. G. Jess, "Execution interval analysis under resource constraints," *ICCAD*, pp. 454-459, 1993.

[4] A. H. Timmer and J. A. G. Jess, "Exact scheduling strategies based on bipartite graph matching," *DATE*, pp. 42-47, 1995.

[5] L. Ge, S. Chen, K. Wakabayashi, T. Takenaka, and T. Yoshimura, "Max-flow scheduling in high-level synthesis," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Science*, vol. E90-A, no. 9, pp. 1940-1948, 2007.

[6] R. Sedgewick, "*Algorithms in C++ Part 5 Graph Algorithms*," 3rd edition, Addison Wesley, pp. 453-472, 2002.

[7] G. D. Micheli, "Synthesis and optimization of digital circuits," McGraw-Hill, pp. 207-208, 1994.

[8] D. Aldous and J. A. Fill, "*Reversible Markov chains and random walks on graphs*," 2002, in preparation, ch. 2.

[9] W. Woess, "*Random walks on infinite graphs and groups*," Cambridge Univ. Press, ch. 1, 2000.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *Int. Symposium on Microarchitecture*, pp. 330-335, 1997.

[11] MOSEK, "*Solution through mathematical optimization*," available at www.mosek.com/index.html