IPSJ Transactions on System LSI Design Methodology Vol. 2 30–42 (Feb. 2009)

Regular Paper

Exploration of Schedule Space by Random Walk

Liangwei Ge,[†1] Song Chen[†1]
and Takeshi Yoshimura[†1]

Scheduling, an important step in high-level synthesis, is essentially a searching process in the solution space. Due to the vastness of the solution space and the complexity of the imposed constraints, it is usually difficult to explore the solution space efficiently. In this paper, we present a random walk based perturbation method to explore the schedule space. The method works by limiting the search within a specifically defined sub-solution space (SSS), where schedules in the SSS can be found in polynomial time. 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. Experiments on well-known benchmarks show that by controlling the number of perturbations, our method conveniently makes tradeoff between schedule quality and runtime. In reasonable runtime, the proposed method finds schedules of better quality than existing methods.

1. Introduction

In high-level synthesis, the scheduling of multiple operations to appropriate control steps in the presence of a set of constraints and objectives is an essential but intractable task [1),2)]. The schedule quality greatly influences the synthesized circuit on speed, delay, power consumption, etc. Therefore, there have been constant efforts to improve the schedule quality.

Recently, scheduling algorithms based on bipartite graph matching have been proposed [3)–6)], which have the advantage of optimizing complex objectives with more flexibility.[3),4)] were among the first to formulate scheduling as a matching problem. Though some reduction techniques were proposed,[3),4)] used an exact method that had unacceptable runtime on large cases.[5)] proposed a heuristic method that optimally schedules one path of operations each time. With a wider view on the design objectives,[5)] tends to achieve better results than list scheduling [7)–9)] and force-directed scheduling [10),11)]. The max-flow scheduling [6)] is different from the iterative, path-based scheduling [5)], which is capable of evaluating and adjusting previous scheduling decisions at later stage and therefore has greater flexibility. Nevertheless, the max-flow scheduling [6)] heuristically restricts the solution space to a sub-solution space (SSS), which may not cover the global optimal schedule. Moreover, it only assumes unit operation delay [6)].

In this paper, we improved the works of[6)] by proposing the random walk based enhanced max-flow scheduling. The enhanced max-flow scheduling supports operation delay of multiple clock cycles and is capable of searching better schedules under the same latency. We incorporated random optimization technique into scheduling and formulated the SSS perturbation as an N-dimension random walk on graph. If the obtained schedule is unsatisfactory, current SSS is perturbed and better schedules (either of lower latency or higher quality) are searched in the new SSS. The SSS perturbation expands the exploration in the solution space and consequently improves the schedule quality. Moreover, by controlling the perturbation number, tradeoff between schedule quality and runtime can be conveniently made. To improve the efficiency of searching the solution space, we also present a guided perturbation strategy that leads the random walk toward promising directions by utilizing the returned information of the max-flow algorithm. Experimental results show that in reasonable runtime, our method finds better schedules than existing methods.

The rest of the paper is organized as follows. Section 2 gives the overview of the proposed scheduling method. Section 3 explains the concept of SSS and the enhanced max-flow scheduling algorithm. Section 4 describes in detail the perturbation of SSS. Section 5 shows the experimental results. And Section 6 draws the conclusion.

2. Overview of Random Walk Based Max-Flow Scheduling Algorithm

Figure 1 shows the flow of the proposed scheduling algorithm. The original max-flow scheduling [6)] only consists of the first three steps: 1) set the schedule

†1 Graduate School of Information, Production and System, Waseda University
⋆1 This work was supported by a grant of Knowledge Cluster Initiative 2nd stage implemented by Ministry of Education, Culture, Sports, Science and Technology (MEXT).

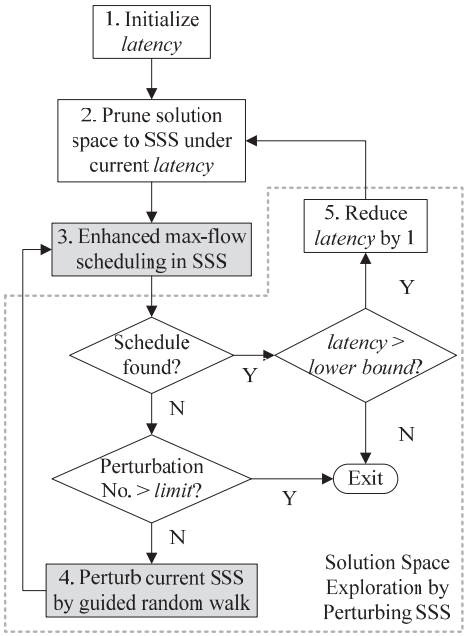


Fig. 1 Flow of random walk based max-flow scheduling.

latency; 2) prune solution space to SSS under given latency; 2) perform max-flow scheduling in SSS without enhancement. In this study, we incorporate random optimization techniques by adding two loops in the flow, as indicated within the dashed line. In the outer loop, the schedule latency decreases from the upper bound, which can be estimated by any heuristic (e.g. list scheduling), to the lower bound. Under each schedule latency, the original solution space is firstly pruned into a SSS. Then, the inner loop is triggered that iteratively explores the entire solution space in search of schedules: 1) the max-flow scheduling algorithm is executed to find schedules in current SSS. Different from the original max-flow scheduling [6] that assume unit operation delay, the enhanced algorithm supports multi-cycle operation delay; 2) if no schedule exists, current SSS is perturbed by randomly walking to a neighboring SSS. In order to reduce the detoured walks before reaching a SSS that contains satisfying schedules, the feedback of the max-flow algorithm is utilized, which guides the walk toward promising directions.

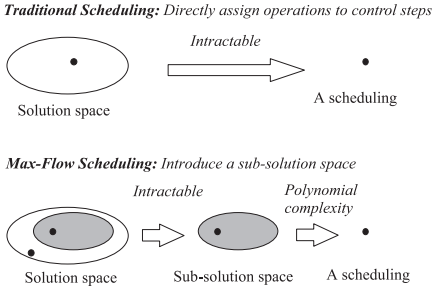


Fig. 2 Comparison of space-constrained scheduling with traditional scheduling.

3. Sub-Solution Space and Enhanced Max-Flow Scheduling Algorithm

Figure 2 compares the scheduling in confined solution space with the traditional methods. Traditional methods usually search schedules directly in the entire solution space, which is an intractable problem. In contrast, the max-flow scheduling method restricts the search within the sub-solution space, as indicated by the grey area. The SSS, which is obtained by a force-directed heuristic [6], has unique properties, such as the optimum schedule of SSS can be found in polynomial time by using the enhanced max-flow scheduling algorithm.

3.1 Problem Formulation

Given an operation set of t types $V = \{V_j \mid j = 1, 2, ..., t\}$ where each type has $n(j)$ operations $V_j = \{v_{ji} \mid i = 1, 2, ..., n(j)\}$, 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. Therefore, the scheduling task can be formulated as a matching problem between the operations and the control steps on bipartite graphs in the presence of various constraints and objective. We denote the bipartite graph of type j by $\mathrm{BG}_j(V_j, S_j, A_j)$, where V_j is the operation set of type j, S_j the control step interval set 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 and only if the *freedom* of v has overlap with interval s (in other word, v can be potentially scheduled in s). The following will explain how to calculate the freedom of V_j and the control step interval set S_j

respectively.

Two kinds of constraints are considered in this paper: 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 [6]. 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, $\{FU(j) \mid j = 1, 2, ..., t\}$, specifies the number of functional units available for each operation type j. In this paper, we assume non-pipelined functional units.

Definition 3.1 (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) scheduling and the *as late as possible* (ALAP) scheduling. **Figure 3** shows a data flow graph (DFG) example with the freedom of each operation 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.

The freedom of v, $[CS_e(v), CS_l(v)]$, defines an interval in which v gives the earliest and the latest scheduling requests. However, the requests may not be satisfied, because the ASAP/ALAP scheduling assumes unlimited resources by

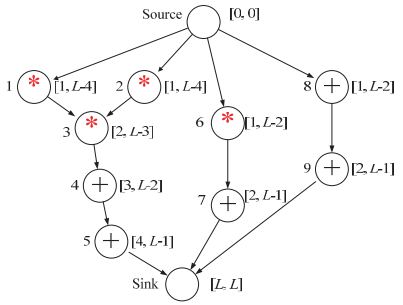


Fig. 3 A DFG example with operation freedom calculated.

only considering the dependency constraint.[3),4)] proposed a method of linear complexity to limit the timing of the scheduling requests due to the resource limit. A similar method is used in this study, as Algorithm 1 shows. The control step interval set of type j, $S_j = \{[g_{ji}, h_{ji}] \mid i = 1, 2, ..., n(j)\}$, firstly records the $n(j)$ ASAP and ALAP requests respectively. Then, some earliest/latest scheduling requests are delayed/advanced, based on the resources available.

Algorithm 1: *Calculation of control step interval set* 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
   $h_{ji} = \min(h_{ji}, h_{j(i+FU(j))} - D(j))$;

$\{g_{ji}\}$ and $\{h_{ji}\}$ are two arrays of integers, with $\{h_{ji}\}$ using the schedule latency L. Obviously, for any interval $[g_{ji}, h_{ji}]$, $g_{ji} \leq h_{ji}$. This results in an estimation of the schedule latency L. The estimated latency serves as the *lower bound* in Fig. 1, because constraints are relaxed during the two-phase S_j calculation: 1) the resource constraint is ignored during the ASAP/ALAP scheduling; 2) the dependency constraint is ignored during the limiting of the scheduling requests (step 3 and step 4 of Algorithm 1).

Table 1 and **Table 2** show the control step interval set S_1 and S_2 of the example DFG. The original interval set is based on the sorted ASAP/ALAP scheduling requests. The narrowed interval set is calculated by Algorithm 1, which takes the resource constraint into account. From the intervals colored in

Table 1 Control step interval set $S_1 = \{[g_{1i}, h_{1i}]\}$ of example DFG.

i	1	2	3	4
Original	[1, L-4]	[1, L-4]	[1, L-3]	[2, L-2]
Narrowed	[1, L-4]	[1, L-4]	[2, L-3]	[2, L-2]

Table 2 Control step interval set $S_2 = \{[g_{2i}, h_{2i}]\}$ of example DFG.

i	1	2	3	4	5
Original	[1, L-2]	[2, L-2]	[2, L-1]	[3, L-1]	[4, L-1]
Narrowed	[1, L-3]	[2, L-2]	[2, L-2]	[3, L-1]	[4, L-1]

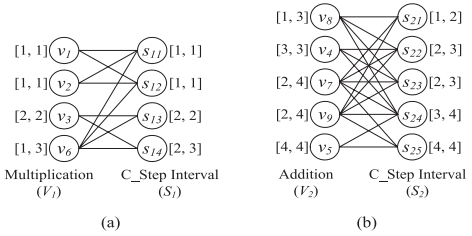


Fig. 4 Bipartite graphs. (a) $BG_1(V_1, S_1, A_1)$. (b) $BG_2(V_2, S_2, A_2)$.

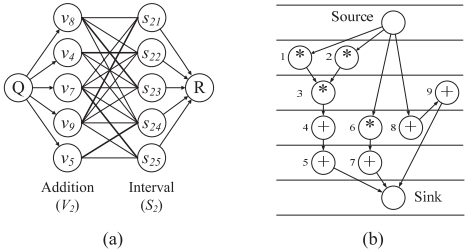


Fig. 5 Finding perfect matching by the max-flow algorithm. (a) A max-flow network. (b) A schedule that violates the dependency constraint.

grey (e.g. [1, L-4]), the schedule latency can be estimated as ($1 \leq L$-4, or $L \geq 5$), which serves as the lower bound.

It should be emphasized that unlike freedom $FR(v_{ji})$, which is bound to v_{ji}, interval $[g_{ji}, h_{ji}]$ is not bound to any operation. $[g_{ji}, h_{ji}]$ only indicates an interval in which the resources allow a type j operation be scheduled. Also, $\{g_{ji}\}$ and $\{h_{ji}\}$ respectively contain the modified $n(j)$ earliest and $n(j)$ latest scheduling requests, which form exactly $n(j)$ intervals. Thus, the number of control step intervals $|S_j|$ equals that of operations $|V_j|$, with one operation scheduled in one interval [3),4)].

Once the schedule latency L, the freedom of V_j, and the control step interval set 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. **Figure 4** shows the bipartite graphs of the example DFG under the schedule latency of ($L = 5$).

By introducing the control step interval $[g_{ji}, h_{ji}]$, which contains arbitrary clock cycles, we enhanced the scheduling method of[6)] by supporting multi-cycle operation delay. However, for explanation convenience, we assume unit operation delay in the example DFG.

3.2 Resource and Dependency Constraints

Once the bipartite graphs $BG_j(V_j, S_j, A_j)$, $1 \leq j \leq t$, are built, the scheduling task is transformed into finding perfect matching between the operations V_j and the control step interval set S_j in the presence of various constraints. The matching problem can be solved in polynomial time by many algorithms. This paper uses the max-flow algorithm [12)]. If a perfect matching is found, the freedom of each operation shrinks to the common part of its original freedom and the matched control step interval. It is possible that some operations still have freedom of multiple control steps. So, a perfect matching is not a strict schedule, but very close to it. A schedule can be easily obtained from a perfect matching.

The schedule based on a perfect matching always satisfies the resource constraint. The reason is that a perfect matching ensures 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.

Figure 5 (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.5 (b) shows the corresponding schedule. Obviously, the resource constraint is satisfied but the dependency of (v_8, v_9) is violated.

Definition 3.2 (overlapped freedom). *For dependency constraint (v_{ix}, v_{jy}) $\in E$, if $CS_e(v_{jy}) < CS_l(v_{ix}) + D(i)$, v_{ix} and v_{jy} are called an overlapped pair. $[CS_e(v_{jy})$, $CS_l(v_{ix}) + D(i) - 1]$ is the freedom of v_{jy} that overlaps with v_{ix}. $[CS_e(v_{jy}) - D(i) + 1$, $CS_l(v_{ix})]$ is the freedom of v_{ix} that overlaps with v_{jy}.*

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.

Definition 3.3 (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*

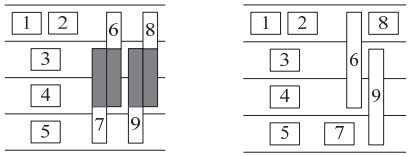


Fig. 6 Freedom distribution in the DFG example. (a) With overlap. (b) Overlap removed through partition.

$CS_l(v_{ix})$.

Figure 6 (a) shows the freedom distribution of Fig. 3 under schedule latency (L = 5). Each rectangle stands for an operation with the vertical span representing its freedom. The grey area indicates the overlapped freedom. Fig. 6 (b) shows one possible partition of the overlaps.

After partitioning all the overlaps, the freedom of the involved operations is decreased, which reduces the edge set A_j of $\mathrm{BG}_j(V_j, S_j, A_j)$. Consequently, the original solution space is pruned into a sub-solution space (SSS). Within SSS, every perfect matching corresponds to schedules that satisfy both the resource and the dependency constraints.

One of the advantages of the max-flow scheduling is that it can conveniently optimize some complex objectives. Edges of A_j, which represent potential assignment between operations and control steps, can be evaluated by assigning weight to them. The edge weight helps find better schedules by using the min-cost max-flow algorithm [6),12)].

3.3 Enhanced Max-Flow Scheduling Algorithm

It is well known that the resource-constrained scheduling is an NP-complete problem [13)]. Therefore, it is difficult to optimally partition all overlapped pairs. In this study, a force-directed heuristic method [6)] is used to partition the overlaps. The enhanced max-flow scheduling under given schedule latency is described in Algorithm 2:

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 set S_j under the resource constraint and the schedule latency L;
4. Partition all operation overlaps by the force-directed heuristic method [6];
5. Build the bipartite graphs $\mathrm{BG}_j(V_j, S_j, A_j)$, $1 \le j \le t$;
6. Perform the max-flow algorithm on the bipartite graphs to find perfect matching;
7. If perfect matching exists, calculate the corresponding schedule. Otherwise, increase the schedule latency by one cycle or perturb current SSS as Section 4 will describe;

4. Perturbation of SSS by Random Walk

As mentioned in Section 3.3, a heuristic method is used to prune the original solution space into the sub-solution space (SSS). Therefore, it is possible that the global optimum schedule is not contained in the SSS. If the optimum schedule of SSS, which can be found in polynomial time by the max-flow scheduling algorithm of Section 3, does not exist or is not satisfactory, it is desirable to perturb current SSS and search better schedules in the new SSS.

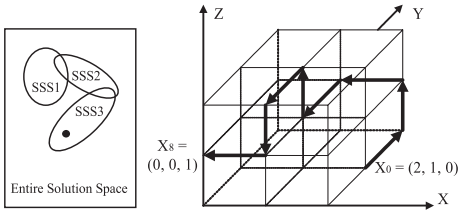


Fig. 7 Exploring schedule space. (a) SSS perturbation. (b) SSS perturbation as random walk on graph G_{SSS}.

Figure 7 (a) illustrates the perturbation of SSS: the outer rectangle represents the entire solution space and 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.

4.1 Formulation of SSS Perturbation as Random Walk on Graph

Assume there are N overlapped operation pairs to be partitioned: p_1, p_2, ..., p_N. The i-th pair p_i has overlap of (k_i - 1) control steps. Then, there are k_i

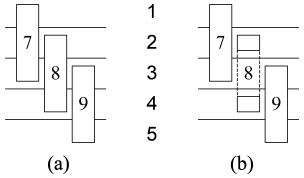


Fig. 8 Example of infeasible state. (a) related operation pairs (v_7, v_8) and (v_8, v_9). (b) Improper partition that causes freedom inconsistency.

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, ..., u_N). The i-th dimension u_i, $1 \leq i \leq N$, takes the value among {0, 1, 2, ..., 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. Note that partitioning p_i reduces the freedom of not only the overlapped pair but also the operations along the same path in the DFG. Thus, some operation pairs may relate to each other. In other words, there might be *infeasible* states in X, where freedom inconsistency is caused by improper partition of related pairs.

Figure 8 shows an example of infeasible state: assume a path of $v_7 \rightarrow v_8 \rightarrow v_9$ in the DFG, where (v_7, v_8) and (v_8, v_9) originally overlap in interval [2, 3] and [3, 4] respectively. The partition scheme of Fig. 8 (b) changes $CS_e(v_8)$ to 4 and $CS_l(v_8)$ to 2. Thus, $FR(v_8)$ becomes [4, 2], which is infeasible.

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 [14),15)] on G_{SSS}, which is a sequence of X-valued random variables {X_t: t = 0, 1, 2, ...} 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 [6)]. The walk ends at $t = n$, when X_n is the first visited state that contains schedules of acceptable quality.

Figure 7 (b) shows an example of random walk on a 2 × 2 × 2 G_{SSS}. (2, 1, 0) is the initial overlap partition scheme. After 8 perturbations, (0, 0, 1), the desired partition scheme, is reached.

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 [14)]. We firstly assume the transition between neighboring states with equal probability:

$$p(x,y) = \begin{cases} 1/\deg(\mathrm{x}), \text{ if y neighbors x} \\ 0, \text{ otherwise} \end{cases} \quad (1)$$

Markov chain has a good property of remembering the previous state. New state can be obtained by modifying the previous state. For SSS perturbation, this property means the force-directed overlap partition algorithm is performed only once to calculate the initial SSS (state X_0). New SSS is obtained by modifying the partition position of an operation pair by one clock cycle.

Assume state j, $j \in X$, is the SSS that contains acceptable schedules. Hitting time on state j is defined as:

$$T_j = min\{t > 0 : X_t = j\} \quad (2)$$

Once the state space X, the transition matrix P, and the starting state $i = X_0$ are known, the expectation of hitting time on state j can be calculated by Eq. (3)[14)]:

$$E_i T_j = (Z_{jj} - Z_{ij})/\pi_j \quad (3)$$

where π is the stationary distribution of the Markov chain, $Z_{ij} = \sum_{t=0}^{\infty} p_{ij}^{(t)} - \pi_j$, and $p_{ij}^{(t)}$ the t-step transition probability.

Equation (3) shows that the transition matrix P has great influence on the hitting time. Therefore, optimizing matrix P is an effective way to shorten the hitting time on an acceptable state. The following subsections will present some heuristic techniques to improve the perturbation efficiency by modifying the transition matrix P.

4.2 Techniques to Improve Perturbation Efficiency

The equalized transition of Eq. (1) generally suffers a long hitting time. This is because during the perturbation (i.e. randomly select a pair and modify its partition position by one clock cycle), the equalized transition may 1) frequently

select a correctly partitioned pair that should not be perturbed; 2) frequently walk into an infeasible state caused by freedom inconsistency among related pairs. Here, we present two techniques to improve the perturbation efficiency: 1) narrow the selection of operations for next perturbation by using the schedule search result in current SSS, which decreases the probability of perturbing correctly partitioned pairs; 2) reduce the chance of entering infeasible states by keeping the dependency consistent between related pairs.

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. 5 (a), an overpruned operation v does not have enough outgoing edges to the control step intervals of S, which therefore cannot find a path to node R. We use the push-relabel algorithm to find perfect matching (details can be found in Ref. 12)):

Algorithm: *Push-relabel based max-flow*

- Initially, each node $v \in V$ has one unit of excess flow;
- All nodes have an integer label called *height*. height(v) = 2, for $v \in V$; height(s) = 1, for $s \in S$; height(R) = 0;
- Repeat
  - Select a node that has excess flow;
  - Send its excess flow to other nodes of lower height through residual edges;
  - While excess flow still exists
    - Raise the height of the selected node by 1;
    - Send excess flow to other nodes of lower height;

If any node's height exceeds a threshold of 2U, where U is the number of nodes in the flow network, perfect matching does not exist. Thus, by monitoring the first operation $v \in V$ whose height exceeds the threshold, we easily obtain an operation 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 greatly reduces the chance of perturbing a correctly partitioned pair and is more likely the direction toward SSSs that contain schedules, avoiding the random walks in irrelevant directions.

The second technique tries to solve the freedom inconsistency encountered in the equalized transition of Eq. (1): when an operation's freedom is changed during

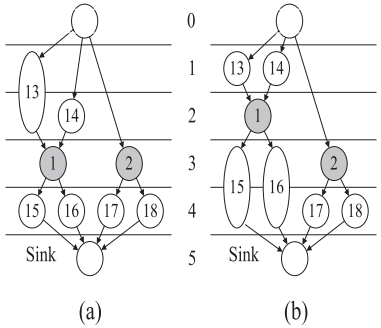


Fig. 9 Perturbation using max-flow feedback and operation dependency. (a) SSS before perturbation. (b) SSS after perturbation.

the perturbation, the freedom of any other related operations, which can be efficiently found by the upward and downward topological search on the DFG, is updated accordingly. In this way, the chance of entering infeasible states of X is avoided and the related pairs are virtually grouped into one pair.

Consider the DFG in **Fig. 9** (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 $\sim$ o18 are congested in control step 4. Assume o15 is the first operation whose height exceeds the threshold (failure of finding a perfect matching). SSS can then be perturbed by expanding the freedom of o15 (rather than a randomly selected operation). If FR(o15) is expanded downward from [4, 4] to [4, 5], the freedom of *sink* will be changed from [5, 5] to [6, 5], which is infeasible. Thus, o15 is expanded upward 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. 9 (b), apparently contains schedules. By utilizing the max-flow algorithm feedback and keeping the dependency constraint between related pairs, a feasible SSS is reached within one perturbation.

As mentioned previously, the heuristic overlap partition method [6] may erroneously partition some operation pairs. However, correcting these errors is not easy. Thus, expanding the operation returned by the max-flow algorithm is a sim-

ple, effective heuristic to correct partition errors. The corresponding probability of expanding $v \in V$ is shown in Eq. (4):

$$\text{Probability to expand } v \in V = \begin{cases} 1, & \text{if } v \text{ is the operation returned by the max-flow algorithm;} \\ 0, & \text{for other operations in } V; \end{cases} \quad (4)$$

Algorithm 3 shows how to expand the freedom of a specified operation without causing freedom inconsistency. Obviously, by using Eq. (4) and Algorithm 3, the perturbation directions for a SSS that does not contain perfect matching is reduced from $2N$ in the equalized transition of Eq. (1) to two: expand the returned operation either upward or downward. In essence, Eq. (4) and Algorithm 3 reduce the N-D random walk to 1-D walk.

Algorithm 3: *Expand the freedom of specified operation o*

```
1.  Assume o has freedom of FR(o) = [I, J];
2.  upward is a random binary;
3.  if (upward)
4.      Expand o upward to [I-1, J];
5.      Let H denotes the operation set of o and its ancestors in
        the DFG;
6.      Sort H in upward topological search order;
7.      For any h ∈ H, if (FR(h) changed)
8.          Push up the direct ancestors of h, if necessary;
9.          Expand the direct successors of h upward, if possible;
10. else
11.     Expand o downward to [I, J+1];
12.     Let H denotes the operation set of o and its successors in
        the DFG;
13.     Sort H in downward topological search order;
14.     For any h ∈ H, if (FR(h) changed)
15.         Push down the direct successors of h, if necessary;
16.         Expand the direct ancestors of h downward, if
            possible;
```

4.3 Guided SSS Perturbation Method

Equation (4) narrows the selection of operations for next perturbation by using the feedback of the max-flow algorithm. It virtually changes the transition matrix P, as transition between some neighboring states becomes prohibited. It should be emphasized that though Eq. (4) is an efficient heuristic, as proved by experiment, it may potentially make G_{SSS} disconnected.

(a) (b)

Fig. 10 Improper modification of P makes G_{SSS} disconnected. (a) Original G_{SSS}. (b) G_{SSS} after modifying P.

Figure 10 shows an example of disconnected G_{SSS}. The dotted edge in Fig. 10 (b) represents reducing the transition probability between two neighboring states to zero. Apparently, G_{SSS} is divided into two connected components.

Disconnected G_{SSS} greatly deteriorates the quality of SSS perturbation. No matter how many times SSS is perturbed, the SSS containing the optimum schedule cannot be reached if the random walk starts from a different component. To solve this problem, we propose the guided SSS perturbation strategy (as shown in Eq. (6)): if current SSS does not contain any perfect matching, Eq. (4) is selected with high priority α (typically $\alpha = 90\%$) and random perturbation, as shown in Eq. (5), is selected with low probability (1 - α). Apparently, the guided perturbation strategy greatly reduces the possibility of making G_{SSS} disconnected.

$$\text{Probability to expand a randomly selected } v = 1/|V|, \text{ for any } v \in V \quad (5)$$

$$\text{Guided SSS perturbation strategy} \begin{cases} \text{select Eq. (4) with probability of } \alpha \\ \text{select Eq. (5) with probability of } (1\text{-}\alpha) \end{cases} \quad (6)$$

Finally, we give the complete description of the random walk based max-flow scheduling in Algorithm 4.

During the exploration of the entire schedule space, a large number of SSSs may be visited. Therefore, Algorithm 4 adopts simple operation expansion strategies (e.g., Eqs. (5) and (6)) in order to reduce the perturbation complexity. Based on the specific optimization objective, it is possible to devise more sophisticated strategies and integrate them into the framework of Algorithm 4, however, at the cost of increasing runtime.

Algorithm 4: *Random walk based max-flow scheduling*

```
1.  Perform list scheduling;
2.  latency = the latency of list schedule;
3.  Perform ASAP and ALAP scheduling;
4.  Under given resources, modify the timing of ASAP and
    ALAP scheduling requests using Algorithm 1;
5.  Estimate lower_bound of schedule latency;
6.  go_on = true
7.  while(go_on)
8.  {
9.      Calculate control step interval set S under latency;
10.     Calculate freedom of V under latency;
11.     Partition overlapped freedom;
12.     Build bipartite graphs BG_j(V_j, S_j, A_j), 1 ≤ j ≤ t;
13.     step = 0; acceptable = false;
14.     while(!acceptable && (step < limit))
15.     {
16.         step = step + 1;
17.         Find perfect matching by max-flow algorithm;
18.         if(perfect matching exists)
19.             if(corresponding schedule quality is good)
20.                 acceptable = true;
21.             else
22.                 Perturb SSS by expanding a random
                    operation using Eq. (5);
23.         else
24.             Perturb SSS by the proposed guided
                perturbation strategy using Eq. (6);
25.     }
26.     if(acceptable) /*good schedule found*/
27.         Output current schedule;
28.         if(latency > lower_bound)
29.             latency = latency - 1;
30.             go_on = true;
31.     else /*no good schedule at current latency*/
32.         go_on = false;
33. }
```

5. Experiment

All the experiments are carried out by a Core Duo CPU of 2.00 GHz (T7300). The DFGs that are used in the experiments are extracted from the C programs in the SPEC CPU2006 benchmark suite [16], generated by the TGFF tool [17], or obtained from the Internet [18),19)]. **Table 3** lists the details of the DFGs.

Table 4 shows the resource constraint for each DFG. 'mult' has a delay of 2 clock cycles. Other functional units have unit operation delay.

Table 3 DFGs used in experiment.

DFG	Description	Node #	Edge #
decode	ADPCM decoding [18]	46	113
mpeg	Motion prediction in MPEG-1 decoding algorithm [19]	53	112
gromacs	Workload 435 of SPEC CPU2006 that simulates molecule motion [16]	869	1200
lbm	Workload 470 of SPEC CPU2006 that simulates incompressible fluids [16]	979	1820
rand1	Random directed acyclic graph generated by TGFF [17]	157	232
rand2	Random directed acyclic graph generated by TGFF [17]	631	928
decode20	Obtained from DFG (*decode*) after unrolling 20 loops	939	2140
mpeg16	Obtained from DFG (*mpeg*) after unrolling 16 loops	863	1760

Table 4 Resource constraint of benchmark DFGs.

DFG	Resource Constraint
decode	2 mult, 1 add
mpeg	1 mult, 2 add
gromacs	2 mult, 1 add
lbm	3 mult, 2 add
rand1	2 mult, 2 add, 1 sub
rand2	2 mult, 2 add, 1 sub
decode20	2 mult, 1 add
mpeg16	1 mult, 2 add

5.1 Experiment 1: Reduction of Schedule Latency

Experiment 1 compares the proposed method with the list scheduling and the exact scheduling (using an ILP solver MOSEK [20)]) on minimizing schedule latency. The fundamental objective of resource-constrained scheduling is minimizing schedule latency. Therefore, we set schedule latency as the optimization objective of the ILP solver. Meanwhile, we use the length of the longest path from operation to sink node in the DFG as the operation priority during list scheduling. Such a priority function tends to generate schedules of low latency. The lower bound estimation of schedule latency [3),4)] is given as well. As **Table 5**

Table 5 Reduction of schedule latency.

DFG	Lower bound	ILP	Random walk-based		List
			Guided perturbation	1-D perturbation	
decode	50	51	51	51	52
mpeg	38	39	39	39	40
gromacs	483	NA	572	586	630
lbm	505	NA	532	532	560
rand1	63	NA	63	64	64
rand2	243	NA	253	254	256
decode20	962	NA	1020	1020	1040
mpeg16	593	NA	624	624	640

shows, the proposed method based on 'guided perturbation' finds schedules of shorter latency than list scheduling in all DFGs. The reduction on latency ranges from $1 \sim 58$ control steps. On the other hand, the exact method using ILP cannot find schedules for DFGs that have more than 150 nodes within an hour. The experiment proves the effectiveness of the proposed scheduling method.

We also tested the two SSS perturbation strategies proposed: the '1-D perturbation' that only uses Eq. (4) and the 'guided perturbation' of Eq. (6) that accepts random perturbation of Eq. (5) with low probability (10%). Experiment on *gromacs*, *rand*1, and *rand*2 shows that the 'guided perturbation' is capable of finding schedules shorter than the '1-D perturbation'. This proves the importance of Eq. (5) to keep G_{SSS} connected. Therefore, in the following experiment, 'guided perturbation' is used as the default perturbation strategy with α set at 90%.

5.2 Experiment 2: Importance of Guided Perturbation

Experiment 2 is designed to show how the max-flow algorithm feedback reduces the hitting time. The 'guided perturbation' utilizes the feedback, while the 'equalized perturbation' uses the random perturbation of Eq. (5). For every DFG, the number of perturbations performed under each perturbing strategy to find a schedule is listed respectively. If the schedule of specified latency (as indicated in Table 5) cannot be found within 10000 perturbations, it is marked with 'F'. As **Table 6** shows, all DFGs require performing perturbations to get a schedule shorter than the list scheduling. This justifies our effort to integrate random walk into the max-flow scheduling. Moreover, the 'Equalized perturbation' needs much more perturbations than the 'Guided perturbation' to achieve the same schedule latency. This result confirms our claim that feedback from the max-flow algorithm is essential for guiding the walk toward promising directions.

Table 6 Hitting time with/without using Max-flow feedback.

DFG	Guided perturbation	Equalized perturbation
decode	3	691
mpeg	2	183
gromacs	8250	F
lbm	1920	F
rand1	2271	F
rand2	8336	F
decode20	1645	F
mpeg16	59	3056

Table 7 The number of states in state space.

DFGs	Node #	Edge #	Overlapped pair #	Average overlap length	Estimated state #
gromacs	869	1200	495	5.988	$\leq 5.988^{(495)}$ ($\approx$ 5.68e+384)
lbm	979	1820	532	2.528	$\leq 2.528^{(532)}$ ($\approx$ 1.89e+214)
rand1	157	232	151	15.728	$\leq 15.728^{(151)}$ ($\approx$ 4.99e+180)
rand2	631	928	594	14.697	$\leq 14.697^{(594)}$ ($\approx$ 2.16e+693)
decode20	939	2140	243	4.045	$\leq 4.045^{(243)}$ ($\approx$ 3.03e+147)
mpeg16	863	1760	241	3.261	$\leq 3.261^{(241)}$ ($\approx$ 5.22e+123)

Table 7 shows the number of overlapped pairs (*pair_no*) and the averaged overlap length (*o_length*) in each DFG. Thus, the number of states in the state space can be roughly estimated by (*o_length*) ^ (*pair_no*). As is shown, the state space (which includes infeasible states) is vast. It verified the importance of the 2nd technique proposed in Section 4.2. By "keeping the dependency consistent between related pairs", the chance of entering infeasible states is eliminated, which greatly improves the perturbation efficiency. Table 7 also justified the 1st technique of Section 4.2 - the 'guided perturbation'. Comparing the huge

state number of Table 7 to the perturbation number of Table 6 (no more than 10000), the 'guided perturbation' strategy is really efficient to lead the random walk through the vast state space.

5.3 Experiment 3: Comparison of Runtime

Table 8 compares the runtime of the enhanced max-flow scheduling with the list scheduling and the exact scheduling. As expected, runtime of the ILP solver increases sharply as the DFG grows. For DFGs of more than 150 nodes, none can be scheduled within 1 hour by the exact method due to the equation explosion. This explains why exact method is seldom used in practice. For the proposed method, the average runtime is 133 seconds, which is slower than the list scheduling but much faster than the exact method. Considering the reduction on schedule latency ($1 \sim 58$ cycles shorter) and the time used on solution space exploration, the runtime is reasonable. 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. In the extreme case when perturbation is not performed, the random walk based scheduling method, as Algorithm 4 shows, is reduced to list scheduling. Therefore, by controlling the perturbation number, tradeoff between schedule quality and runtime can be conveniently made, as the next experiment will show.

Table 8 Comparison of runtime (second).

DFG	List	Random walk based max-flow scheduling	ILP
decode	4.531	4.547	5.121
mpeg	3.218	3.234	3.498
gromacs	3.625	294.640	NA
lbm	4.125	78.421	NA
rand1	2.375	52.093	NA
rand2	2.797	603.797	NA
decode20	4.750	21.015	NA
mpeg16	4.796	6.609	NA
Average	**4.54**	**133.04**	**NA**

5.4 Experiment 4: Tradeoff between Schedule Latency and Runtime

Experiment 4 is designed to reveal the relationship between runtime/schedule quality (latency in this experiment) and the perturbation numbers. **Figure 11** show the latency of the optimal schedules found under different perturbation numbers. Meanwhile, **Fig. 12** gives the corresponding runtime. Experiment on

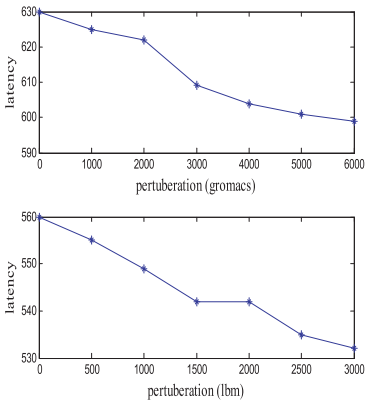


Fig. 11 Schedule latency drops as perturbation number increases. (upper) DFG *gromacs*. (lower) DFG *lbm*.

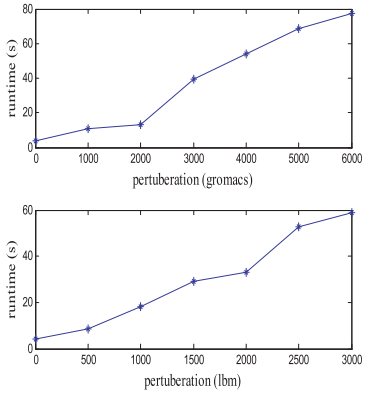


Fig. 12 Relation between runtime and perturbation number. (upper) DFG *gromacs*. (lower) DFG *lbm*.

DFG *lbm* shows that as the allowed perturbation number increases gradually from 0 to 3000, the schedule latency drops steadily and the runtime becomes longer. Similar pattern can be observed on DFG *gromacs*. This result confirms our claim that more perturbations find better schedules but require longer runtime. Therefore, users can conveniently control the schedule quality and runtime by adjusting the perturbation number.

6. Conclusion

In high-level synthesis, searching schedules in the vast solution space is an important but intractable task. We proposed an efficient method to explore the solution space by using the random walk technique. Our main contributions are the formulation of SSS perturbation as random walk on graph and an efficient SSS perturbation heuristic that utilizes the search result of current SSS. Experimental results show that the proposed method effectively enhances the original max-flow algorithm. Meanwhile, users are given greater control over the schedule quality and runtime. Compared with existing methods, our method generally finds better schedules in reasonable runtime.

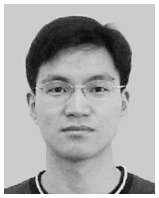
References

1) Mcfarland, M.C., Parker, A.C. and Camposano, R.: The High-Level Synthesis of Digital Systems, *Proc. IEEE*, Vol.78, No.2, pp.301–318 (1990).
2) Lin, Y.: Recent developments in high-level synthesis, *ACM Trans. Design Automation of Electronic Systems*, Vol.2, No.1, pp.2–21, (1997).
3) Timmer, A.H. and Jess, J.A.G.: Execution interval analysis under resource constraints, *Proc. International Conference on Computer-Aided Design*, pp.454–459 (1993).
4) Timmer, A.H. and Jess, J.A.G.: Exact scheduling strategies based on bipartite graph matching, *Proc. Design, Automation and Test in Europe*, pp.42–47 (1995).
5) Memik, S.O., Kastner, R., Bozorgzadeh, E. and Sarrafzadeh, M.: A scheduling algorithm for optimization and early planning in high-level synthesis, *ACM Trans. Design Automation of Electronic Systems*, Vol.10, No.1, pp.33–57 (2005).
6) Ge, L., Chen, S., Wakabayashi, K., Takenaka, T. and Yoshimura, T.: 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).
7) Allan, V.H., Su, B., Wijaya, P. and Wang, J.: Foresighted instruction scheduling under timing constraints, *IEEE Trans. Computers*, Vol.41, No.9, pp.1169–1172 (1992).
8) Pangrle, B.M. and Gajski, D.D.: Design tools for intelligent silicon compilation, *IEEE Trans. Computer-Aided Design*, Vol.6, no.6, pp.1098–1112 (1987).
9) Girczyc, E.F., Buhr, R.J.A. and Knight, J.P.: Applicability of a subset of Ada as an algorithmic hardware description language for graph-based hardware compilation, *IEEE Trans. Computer-Aided Design*, Vol.4, No.2, pp.134–142 (1985).
10) Paulin, P.G. and Knight, J.P.: Force-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Trans. Computer-Aided Design*, Vol.8, No.6, pp.661–679 (1989).
11) Paulin, P.G., Knight, J.P. and Girczyc, E.F.: HAL: A multi-paradigm approach to automatic data path synthesis, *Proc. Design Automation Conference*, pp.263–270 (1986).
12) Sedgewick, R.: Algorithms in C++ Part 5 Graph Algorithms, 3rd edition, Addison Wesley, pp.453–472 (2002).
13) Micheli, G.D.: Synthesis and optimization of digital circuits, McGraw-Hill, pp.207–208 (1994).
14) Aldous, D. and Fill, J.A.: Reversible Markov chains and random walks on graphs, in preparation, ch. 2 (2002).
15) Woess, W.: Random walks on infinite graphs and groups, Cambridge Univ. Press, ch. 1 (2000).
16) The Standard Performance Evaluation Corporation, SPEC CPU2006 benchmarks, available at: www.spec.org/cpu2006/press/V1.1release.html
17) Rhodes, D., Dick, R. and Vallerio, K.: TGFF: task graph for free, available at: http://ziyang.eecs.northwestern.edu/dickrp/tgff/
18) ADPCM (Adaptive Differential Pulse Code Modulation) benchmark, the University of California, 2000.
19) Motion prediction block in MPEG-1 decoding algorithm, the University of California, 2000.
20) MOSEK, Solution through mathematical optimization, available at: www.mosek.com/index.html

(Received May 23, 2008)
(Revised August 22, 2008)
(Accepted October 9, 2008)
(Released February 17, 2009)

(Recommended by Associate Editor: *Hiroshi Saito*)

Liangwei Ge received his B.E. from Xi'an Jiaotong University, China, in 2000 and M.E. from Tsinghua University, China, in 2003. He is currently a Ph.D. candidate in the Graduate School of Information, Production and System, Waseda University, Japan. His research interests include high-level synthesis, computer arithmetic, and VLSI design automation.



Song Chen received the B.S. degree in computer science from Xi'an Jiaotong University, China, in 2000 and Ph.D. in computer science from Tsinghua University, Peking, China, in 2005. From August 2005, he has been a post-doctor in the Graduate School of Information, Production and System, Waseda University, Japan. His research interests include high-level/physical synthesis of VLSI circuits, especially floorplanning/placement for 2D/3D ICs, integration of high-level synthesis and physical synthesis, etc.

Takeshi Yoshimura received B.E., M.E., and Dr. Eng. degrees from Osaka University, Osaka, Japan, in 1972, 1974, and 1997. He joined the NEC Corporation, Kawasaki, Japan, in 1974, where he has been engaged in research and development efforts devoted to computer application systems for communication network design, hydraulic network design, and VLSI CAD. From 1979 to 1980 he was on leave at the Electronics Research Laboratory, University of California, Berkeley, where he worked on very large scale integration computer-aided design layout. He received Best Paper Awards from the Institute of Electronics, Information and Communication Engineers of Japan (IEICE) and the IEEE CAS Society. Dr. Yoshimura is a Member of the IEICE, IPSJ (the Information Processing Society of Japan), and IEEE.