Journal of Zhejiang University SCIENCE A
ISSN 1009-3095 (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Fast combination of scheduling chains under resource and time constraints*

WANG Ji-min†, PAN Xue-zeng, WANG Jie-bing, SUN Kang
(*School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*)
†E-mail: bigjim@zju.edu.cn

Received May 16, 2006; revision accepted Aug. 9, 2006

Abstract: Scheduling chain combination is the core of chain-based scheduling algorithms, the speed of which determines the overall performance of corresponding scheduling algorithm. However, backtracking is used in general combination algorithms to traverse the whole search space which may introduce redundant operations, so performance of the combination algorithm is generally poor. A fast scheduling chain combination algorithm which avoids redundant operations by skipping "incompatible" steps of scheduling chains and using a stack to remember the scheduling state is presented in this paper to overcome the problem. Experimental results showed that it can improve the performance of scheduling algorithms by up to 15 times. By further omitting unnecessary operations, a fast algorithm of minimum combination length prediction is developed, which can improve the speed by up to 10 times.

Key words: Fast combination algorithm, Chain-based scheduling algorithm, High-level synthesis (HLS), Minimum length prediction
doi:10.1631/jzus.2007.A0119 **Document code:** A **CLC number:** TP391.7

INTRODUCTION

High-level synthesis (HLS) maps a behavioral description of a digital system into a register-transfer-level (RTL) design. Operation scheduling is one of the major steps in HLS and "perhaps the most important step during structural synthesis" (Gajski *et al*., 1986). It assigns operations in the behavioral description into control steps. The scheduling problem is known to be an NP-complete problem (Ullman, 1975). Performance of the scheduling algorithm is of much importance to the overall performance of synthesis results. Generally, a good scheduling algorithm should be able to find scheduling sequence(s) with low cost (control steps used, number of registers needed, power consumption) in a relatively short time. As for chain-based scheduling algorithms (Yuan and Shen, 1998; Memik *et al*., 2005), which schedule a number of nodes (a chain) from the dataflow graph at a time, and combine it with existing chains under resource and time constraints while retaining the dependency of nodes, the key to find optimal or suboptimal scheduling sequence(s) is to combine scheduling chains quickly and find enough valid combination results. This is because optimal result may be achieved from any intermediate chain, so we have to maintain a reasonable number of intermediate chains, and to process these chains, the combination speed must be fast enough.

In this paper, a fast combination algorithm is presented, which can be used to solve resource-constrained or time-constrained scheduling problems. The algorithm takes both time constraint and resource constraint as input. The extra constraint (time constraint for resource-constrained scheduling problems or resource constraint for time-constrained scheduling problems) is used to pre-prune the search space so as not to get too much combination results. To further speed up the combination process in coping with resource-constrained problems, a minimum combi-

* Project (No. Y105355) supported by the Natural Science Foundation of Zhejiang Province, China

nation length prediction algorithm is also presented for fast prediction of the minimum length of combined chains, thus to avoid unnecessary combination.

RELATED WORK

There are two classes of scheduling problems: time-constrained scheduling (TCS) and resource-constrained scheduling (RCS) (Lin, 1997). There are many scheduling algorithms developed by researchers to address TCS and/or RCS problems. According to optimization of scheduling results, scheduling algorithms can be classified into two categories: exact solutions and heuristics. Integer Linear Programming (ILP) (Hwang *et al.*, 1991), SAT-based scheduling algorithm (Memik and Fallah, 2002), and Branch and Bounding algorithm (Narasimhan and Ramanujam, 2001) can find exact (optimal) solutions to the scheduling problem, but with a high computation complexity; Force Directed Scheduling (FDS) (Paulin and Knight, 1989) and its variants, list-based scheduling (Parker *et al.*, 1986) and its improved version (Sllame and Drabek, 2002) compute suboptimal solutions with low costs. Chain-based scheduling algorithms can be tuned to find optimal or suboptimal solutions, whose results are found to depend on how partial-scheduled chains are combined. Yuan and Shen (1998) adopted the simplest method, which was to match two chains from the very beginning and did not consider the cost of combined scheduling sequences. Memik *et al.*(2005) also developed a chain-based scheduling algorithm, but they converted it into a "max-weighted *k*-chain" problem, and used bipartite matching to solve it. Both of these two algorithms compute only one combination chain from two input chains, and thus result in suboptimal solutions. To compute optimal solutions, more combination results must be found and kept for further combination until the optimal results are found.

There are also scheduling algorithms aiming at multiple objective optimizations. Mohanty and Ranganathan (2005), Mohanty *et al.*(2006) and Kumar *et al.*(2004) developed scheduling algorithms for RCS and TCS problems that optimized for low power. Chantana *et al.*(2004), however, presented an algorithm that made resource and register usage optimizations in architectural synthesis.

PROBLEM FORMULATION

The behavior of the digital system is described in a data flow graph (DFG), which is in nature a directed acyclic graph (DAG). We denote DFG G by a 2-tuple (V, E), in which $V=\{v_1,v_2,\ldots,v_n\}$, each v_i $(1\le i\le n)$ is a vertex of G, $n=|V|$ is the number of vertices, and $E=\{e_1,e_2,\ldots,e_m\}$, each $e_j=(v_\tau,v_\varphi)$ $(1\le j\le m;\ v_\tau,\ v_\varphi\in V)$ is an edge of G, $m=|E|$ is the number of edges. If $(v_\tau,v_\varphi)\in E$, v_τ is called a predecessor of v_φ; v_φ is called a successor of v_τ. Direct and indirect predecessors of v_φ are called ancestors of v_φ; direct and indirect successors of v_φ are called descendants of v_φ. Component library is a set of hardware components (such as multiplier and ALU) which completes the operation of vertices in the DFG: $C=\{c_k|c_k=(t,d),\ 1\le k\le \Pi,\ t$ is the type of the component, d is delay in steps of this component, Π is the number of components$\}$. Resource constraint $\check{R}_t$ $(1\le t\le \Pi)$ is the number of available resource of type t (for ease of discussion, we assume that each operation in V can only be performed on one type of resource). Time constraint $\check{T}$ regulates the maximum number of control steps that a combination chain could distribute its operations into. A scheduling chain S is a sequence of control steps S_1, $S_2,\ldots$, each is a set that contains one or more operations $S_{11},S_{12},\ldots,S_{21},\ldots$ scheduled from whole or part of the DFG that satisfy the dependency and resource constraints. The chain combination problem under resource constraint $\check{R}$ and time constraint $\check{T}$ is formulated as follows.

To combine two independent chains $S_1,S_2,\ldots,S_\alpha$ and $T_1,T_2,\ldots,T_\beta$, is to find all valid chains $U_1,U_2,\ldots,U_\gamma$ that satisfy:

(1) $\gamma\le\check{T}$. (Time constraint)

(2) $\forall j,\ \forall t,\ 1\le j\le\gamma,\ 1\le t\le\Pi$
$|\{U_{jk}|U_{jk}\in U_j\wedge \text{type}(U_{jk})=t\}|\le\check{R}_t$.
(Resource constraint)

(3) $\sum_{i=1}^{\alpha}|S_i|+\sum_{i=1}^{\beta}|T_i|=\sum_{i=1}^{\gamma}|U_i|$.
(All operations must be scheduled)

(4) $\forall j,\ 1\le j\le\alpha,\ \forall k,\ 1\le k\le|S_j|,\ \sum_{i=1}^{\gamma}\psi(S_{jk},U_i)=1$,

and

$\forall j,\ 1\le j\le\beta,\ \forall k,\ 1\le k\le|T_j|,\ \sum_{i=1}^{\gamma}\psi(T_{jk},U_i)=1$, in which

$$\psi(op, S)=\begin{cases}1, & \text{if } op \in S,\\ 0, & \text{if } op \notin S.\end{cases}$$

(Each operation is scheduled only once)

(5) $\forall j$, $1\le j\le\gamma$, $\forall k$, $1\le k\le|U_j|$

$\neg\exists i$ that $i\ge j\wedge U_i$ contains an ancestor of U_{jk}.

(Dependency of DFG)

An example of scheduling chain combination is depicted in Fig.1. In this example, two chains "Chain 1" and "Chain 2" are to be combined under the resource constraint of one multiplier and one ALU, and the time constraint of 5 control steps. After combination, two valid chains R1 and R2 are found. Note that the two chains to be combined are carefully selected from the DFG, and preprocessed to be independent.

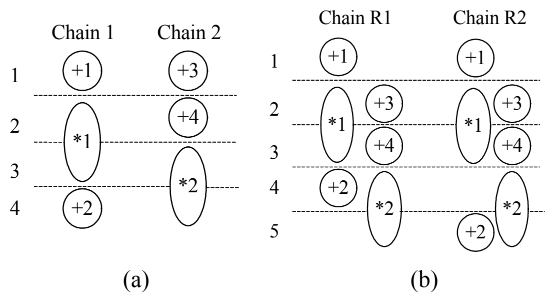

Fig.1 Example of chain combination. (a) Two chains to be combined; (b) Valid chains after combination

FAST COMBINATION ALGORITHM

A naïve way to combine two scheduling chains is to use backtracking to enumerate all possible solutions. Implementation of this algorithm is illustrated below. For ease of discussion, we assume that all resources in the component library take only one cycle to complete its operation.

Implementation of a naïve combination algorithm

```
// res: resource constraint
// time: time constraint
// $S_1\ldots S_n$: scheduling sequence one
// $T_1\ldots T_m$: scheduling sequence two
// pfx: prefix of combination result
// i and j: current indexes of S and T
procedure Combine(res, time, pfx, S, T, i, j)
  if ($i=n+1$ and $m-j+1\le time$)
    Output($pfx+T_j\ldots T_m$);     // concat and output
  else if ($j=m+1$ and $n-i+1\le time$)
    Output($pfx+S_i\ldots S_n$);     // concat and output
  else
    $bak\leftarrow pfx$;          // back up pfx
    $pfx\leftarrow pfx+S_i$;      // add $S_i$ to the end of pfx
    Combine(res, time−1, pfx, S, T, i+1, j);
    $pfx\leftarrow bak$;          // restore pfx
    $pfx\leftarrow pfx+T_j$;      // add $T_j$ to the end of pfx
    Combine(res, time−1, pfx, S, T, i, j+1);
    $pfx\leftarrow bak$;          // restore pfx
    (cur, len)←comb_cycle(res, $S_i$, $T_j$)
                                  // combine current cycle
    $pfx\leftarrow pfx+cur$;      // concat with pfx
    Combine(res, time−len, pfx, S, T, i+1, j+1);
  endif
end procedure

// top module
procedure top_combine(res, time, S, T)
  Combine(res, time, NULL, S, T, 1, 1);
end procedure
```

Because this algorithm does not perform any pruning on the search space, both worst case and average time complexity of the algorithm are exponential. Basically, the chain combination process is to make a traversal on the search tree, performance of the algorithm can be improved by pruning unreasonable branch of the search tree. As can be seen from the pseudocode, the kernel of the algorithm is *comb_cycle* procedure, which combines two control steps from two chains into one. The procedure is executed without any constraints; that is to say, it is executed without considering whether it is necessary. In fact, if some *comb_cycle*(res, S_i, T_j) procedure call can produce no more results than $S_i\ldots T_j$ and $T_j\ldots S_i$, this procedure call and the following recursive call to *Combine* are both unnecessary, because these two cases will be covered by other *Combine* calls. An example in Fig.2 illustrates one of the two cases.

There are several cases in which *comb_cycle*(res, S_i, T_j) procedure call produces only two results $S_i\ldots T_j$ and $T_j\ldots S_i$, with each case being called a combination failure. In the following cases, a combination failure will occur: (1) S_i (or T_j) consumes all available resources, so T_j (or S_i) must be scheduled in the following control step. (2) Suppose S_i (or T_j) consumes more or at least the same number of types of resources than T_j (or S_i), for each type of resource that T_j (or S_i) consumes at least one unit, S_i (or T_j) needs all available resources of that type. Combination failures are illustrated in Fig.3. In this example, two chains

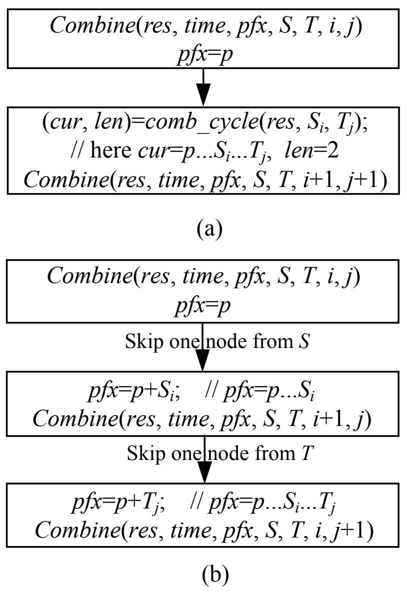
Combine(res, time, pfx, S, T, i, j)
pfx=p
(cur, len)=comb_cycle(res, S_i, T_j);
// here cur=p...S_i...T_j, len=2
Combine(res, time, pfx, S, T, i+1, j+1)
(a)
Combine(res, time, pfx, S, T, i, j)
pfx=p
Skip one node from S
pfx=p+S_i; // pfx=p...S_i
Combine(res, time, pfx, S, T, i+1, j)
Skip one node from T
pfx=p+T_j; // pfx=p...S_i...T_j
Combine(res, time, pfx, S, T, i, j+1)
(b)

Fig.2 Example that shows the case of redundant *Combine* and *comb_cycle* calls. (a) Situation reached through *comb_cycle* call; (b) The same situation reached without *comb_cycle* call

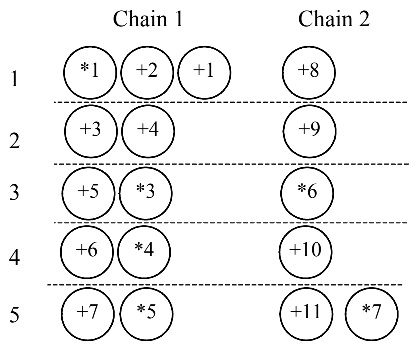

Fig.3 Various combination cases

"Chain 1" and "Chain 2" are to be combined under the resource constraint of one multiplier and two ALUs. In the first control step, "Chain 1" consumes all available resources, so whatever the corresponding control step in "Chain 2" is, a combination failure will occur. In control steps 2 and 3 of the example, although none of the two control steps consumes all available resources, no significant "new" results can be generated from combination of the two control steps [Note that although (+3,+4)…+9 and (+3,+9)…+4 are two different sequences, there is no significant difference between them in the view of operation pattern]. Two cases of successful combination are also shown in the figure. In control step 4, total resource requirements of Chain 1 and Chain 2 are just the amount of available resources, so it is a perfect match. In control step 5, total resource requirements of Chain 1 and Chain 2 exceed the amount of available resources, but significantly different results can also be generated [for example, (+7,*5)…(+11,*7) and (+7,*5,+11)…(*7) are two different sequences]. Two control steps are said to be compatible if they can be successfully combined.

Knowing in which cases a *Combine* call is necessary and in which cases it is redundant, we can revise the program to remove redundant function calls and improve its performance. The revised algorithm is listed as follows.

```
Fast chain combination algorithm
// res: resource constraint
// time: time constraint
// $S_1…S_n$: scheduling sequence one
// $T_1…T_m$: scheduling sequence two
procedure Combine(res, time, S, T)
  init stack s;
  link up available control steps of S, T;
  $i$←first_avail(S);   // get index of first available step in S
  $j$←first_avail(T);   // get index of first available step in T
  $k$←match($S_i$,$T_j$,1,res); // try to find a match of $T_j$ in S from $S_i$
  push(s, k, j, NULL, $S_1...S_{k-1}$, $T_1...T_{j-1}$, time–(k+j–2));
  ...  // omit code for searching along T
  while (not empty(s))
    (i, j, pfx, pa, pb, time)←pop(s);
    while ($i \neq -1$ and $j \neq -1$)
      back up pfx, pa;
      $k$←match($S_{next\_avail(S,i)}$, $T_j$, 1, res);
      if (k=–1 and time>=(n–i+m–j))
        Output(pa, pb, pfx, $S_{i+1}$, $T_{j+1}$);
      else
        if (pfx=NULL)
          pa←pa+($S_i...S_{k-1}$);
        else
          pfx←pfx+($S_i...S_{k-1}$);
        endif
          push(s, k, j, pfx, pa, pb, time–(k–i–1));
      endif
      restore pfx, pa;
      ... // omit code for searching along T
      (cur, len)←comb_cycle($S_i$, $T_j$); // combine current step
      pfx←pfx+cur;        // concat with pfx
      time←time–len;      // adjust time limit
      i←next_avail(S, i); // seek to next available step of $S_i$
      j←next_avail(T, j); // seek to next available step of $T_j$
      ... // omit code for searching along S and T. If one match
       // is found, related variables are updated; if two
       // matches are found, the second one is pushed onto
       // the stack. If any attempt fails, try to output the results.
    endwhile
  endwhile
end procedure
```

Due to space limitation, some codes of the algorithm are omitted, however, the readability of the algorithm is not affected since the algorithm is symmetrical in general and omitted codes can be easily deduced. The algorithm uses stack instead of recursive function call to decrease overhead. To avoid unnecessary processing of control steps that have already consumed all available resources, "available" steps (those that consume fewer resources than available) are linked up in a linked list, and can be traversed by *first_avail* and *next_avail* function calls. The *match*(S_i, T_j, 1, *res*) function call is used to find the next compatible step of T_j in chain S from step S_i by calling function *next_avail* repeatedly. Firstly the initial match along each chain is detected and pushed onto the stack. Then for each record in the stack, the algorithm tries along each chain to find a match and push any match onto the stack. Partial combined chains are stored as prefix in "*pa*", "*pb*" or "*pfx*" member and when any chain reaches its end, the prefix is used to construct final results to output. This version of *Output* function is also different from the one listed in "the implementation of naïve combination algorithm" where it has more arguments. Among these arguments, *pa* and *pb* are uncombined parts of chains S and T respectively, *pfx* is the combined prefix, $S_{i+1}\ldots S_n$ and $T_{j+1}\ldots T_m$ are the remainder of chains S and T respectively. An example run of *Output* is depicted in Fig.4. In this example, none of $S_{i+1},S_{i+2},\ldots,S_n$ is compatible with T_{j+1}.

```
Output("a...b","c...d","e...f","g...h","i...j...k")
The output is:
    a...b...c...d...e...f...g...h...i...j...k
    a...b...c...d...e...f...g...i...j...k...h
    a...b...c...d...e...f...i...j...k...g...h
    c...d...a...b...e...f...g...h...i...j...k
    c...d...a...b...e...f...g...i...j...k...h
    c...d...a...b...e...f...i...j...k...g...h
```

Fig.4 Example run of *Output* function
pa and *pb* are independent and either one can be put on the first position. *pfx* follows *pa* and *pb*. The last two arguments, "g…h" and "i…j…k", are treated differently: "i…j…k" is treated as a whole, while "g…h" can be separated

MINIMUM COMBINATION LENGTH PREDICTION

In scheduling algorithms based on chain combination, the combination operation will be iterated several times, and in each iteration there may be a large number of intermediate chains. Combining such intermediate chains is time-consuming. However, its performance can be improved by rapidly predicting the minimum combination length and avoiding unnecessary combinations (If the minimum length of combination results is still greater than the time constraint, the combination is unnecessary).

The fast minimum combination length prediction algorithm is listed below. It is similar to the fast chain combination algorithm, but it runs much faster. Firstly, it does not need to maintain the prefix chain any more; only its length is needed. Secondly, no backtracking is needed: once a match is found, both chains move forward to the next available step. A new function *compatible* is introduced in the algorithm, which is used to judge whether current steps of the two chains can be combined into one. The algorithm also uses double-linked list to speed up available node search and uses a stack to remember intermediate matches. Firstly initial matches are detected and pushed onto stack. Then each item in the stack is tested for feasibility, and feasible matches are pushed onto stack again. Each time an infeasible chain is found, length of the final chain is computed and minimum length is updated if necessary.

Implementation of minimum combination length prediction algorithm

```
// res: resource constraint
// $S_1\ldots S_n$: scheduling sequence one
// $T_1\ldots T_m$: scheduling sequence two
procedure MinLen(res, S, T)
  min_len←∞;
  init stack s;
  link up available control steps of S, T;
  i←first_avail(S);          // first available step in S
  j←first_avail(T);          // first available step in T
  k←match($S_i$, $T_j$, 1, res); // try to find a match of $T_j$ in S from $S_i$
  push(s, k, j, k–1);
  k←match($S_i$, $T_j$, 2, res); // try to find a match of $S_i$ in T from $T_j$
  push(s, i, k, i–1);
  while (not empty(s))
    (i, j, pfx_len)←pop(s);
    while ($i \neq -1$ and $j \neq -1$)
      if (compatible(res, $S_i$, $T_j$)==1)
        k←next_avail(S, i);
        j←next_avail(T, j);
        if ($k \neq -1$ and $j \neq -1$)
          pfx_len←pfx_len+(k–i–1);
          i←k;
        else
          $min\_len$←min($min\_len$, $m$+$pfx\_len$+$n$–$i$);
        endif
      else
        $k$←$match$($S_i$, $T_j$, 1, $res$);
          // try to find a match of $T_j$ in $S$ from $S_i$
        if ($k$>0)
          push($s$, $k$, $j$, $pfx\_len$+$k$–$i$–1);
        else
          $min\_len$←min($min\_len$, $m$+$pfx\_len$+$n$–$i$+1);
        endif
        ... // omit code for searching along $T$
      endif
    endwhile
  endwhile
  return $min\_len$;
end procedure
```

ADAPTING THE ALGORITHMS TO REAL-LIFE APPLICATIONS

When presenting the algorithms, an assumption is made that each type of resource takes only one cycle to run. However, in real-life applications, there are many types of hardware components that are of multiple-cycle. For example, a multiplier generally takes two cycles to perform a multiplication. The fast combination algorithm and fast minimum combination length prediction algorithm must be modified to accommodate this situation. To represent multiple-cycle operation, multiple nodes can be used, with each representing one stage of the operation. For example, a 2-stage multiplier can be represented by an original multiplier node followed by a node of new type "dummy". Here a "dummy" node performs the second stage of multiplication. According to the definition of dummy operation, each dummy node must follow a multiplier node and no other node can be inserted in-between; also each multiplier node must be followed by a dummy node. When scheduling nodes in chains, extra care must be taken so as not to split a multiplier-dummy node pair.

An example of combining two chains that contain 2-stage multipliers is shown in Fig.5. Fig.5b is a snapshot of running combination algorithm with input "Chain 1" and "Chain 2" shown in Fig.5a. At this time, nodes "dummy" and "+4" are a match, and the algorithm searches along Chain 2, trying to find a new match and putting it onto stack. Of course "dummy" and "+5" are also a match. However, it is not a feasible solution, since node "+4" will be inserted between "(*1,+3)" and "dummy", and "*1-dummy" node pair will be broken.

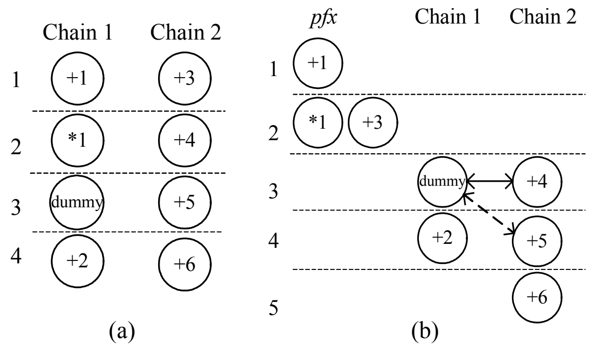

Fig.5 Example of combination algorithm. (a) Two chains to be combined; (b) Snapshot of the combination algorithm

EXPERIMENTAL RESULTS

To test the performance of the two algorithms, we implement them with Visual C++ 6.0, and apply them to standard benchmarks EWF and FDCT. For comparison, running times of fast combination algorithm with and without fast minimum combination length prediction are collected respectively. The naïve version of combination algorithm is also implemented and its running times on EWF and FDCT with and without fast minimum combination length prediction are collected. Experimental results are shown in Tables 1~2. All data are collected on a machine with 1.6 GHz Intel Pentium IV CPU, 512 M memory machine running Windows XP SP2.

Experimental results show that no matter whether fast minimum combination length prediction algorithm is applied, the fast version of combination algorithm gets a better performance than the naïve recursive version: it runs 2~15 times faster. Also, both versions of combination algorithms get better results when fast minimum combination length prediction is used than running alone, but due to the limitation we impose on the number of intermediate results, the potential of prediction algorithm is not fully brought into play.

Table 1 Experimental results for EWF benchmark

Res	Pred	Fast	Inter	Time (s)	Results	Res	Pred	Fast	Inter	Time (s)	Results
1*,1+Non-pipelining	N	N	1000	5.17	158	1*,1+Pipelining	N	N	2000	11.40	302
	N	Y	1000	0.58	249		N	Y	2000	0.79	666
	Y	Y	1000	0.57	249		Y	Y	2000	0.75	666
	Y	N	1000	5.12	158		Y	N	2000	6.84	282
1*,2+Non-pipelining	N	N	2000	40.03	2000	1*,2+Pipelining	N	N	2000	0.60	351
	N	Y	2000	15.27	2000		N	Y	2000	11.33	2000
	Y	Y	2000	15.99	2000		Y	Y	2000	3.51	2000
	Y	N	2000	19.30	2001		Y	N	2000	5.79	351
2*,2+Non-pipelining	N	N	4000	0.89	12	2*,2+Pipelining	N	N	4000	0.92	18
	N	Y	4000	0.99	12		N	Y	4000	0.99	18
	Y	Y	4000	0.44	12		Y	Y	4000	0.45	18
	Y	N	4000	1.10	12		Y	N	4000	1.07	18
2*,3+Non-pipelining	N	N	4000	1.90	4000	2*,3+Pipelining	N	N	4000	0.06	78
	N	Y	4000	1.16	4000		N	Y	4000	0.10	81
	Y	Y	4000	1.01	4000		Y	Y	4000	0.05	81
	Y	N	4000	1.16	4002		Y	N	4000	0.23	78
3*,3+Non-pipelining	N	N	4000	0.23	78	3*,3+Pipelining	N	N	4000	0.27	117
	N	Y	4000	0.11	81		N	Y	4000	0.11	120
	Y	Y	4000	0.06	81		Y	Y	4000	0.06	120
	Y	N	4000	0.20	78		Y	N	4000	0.24	117

"Res" indicates the number of resources used, in which "*" is multiplier and "+" is ALU. "Pred" indicates whether prediction algorithm is used; "Fast" indicates whether the fast version combination algorithm is used; "Inter" indicates the number of intermediate results limit; "Results" indicates the number of results found. "Time" is the time in seconds used to find these results

Table 2 Experimental results for FDCT benchmark

Res	Pred	Fast	Inter	Time (s)	Results	Res	Pred	Fast	Inter	Time (s)	Results
3*,3+Non-pipelining	N	N	4000	8.18	101	3*,3+Pipelining	N	N	4000	2.44	90
	N	Y	4000	4.31	102		N	Y	4000	1.78	83
	Y	Y	4000	3.21	102		Y	Y	4000	1.77	83
	Y	N	4000	7.06	101		Y	N	4000	2.43	90
4*,3+Non-pipelining	N	N	10000	3.53	101	4*,3+Pipelining	N	N	10000	1.47	102
	N	Y	10000	1.51	101		N	Y	10000	1.31	100
	Y	Y	10000	1.46	101		Y	Y	10000	1.27	100
	Y	N	10000	3.13	100		Y	N	10000	1.43	102
4*,4+Non-pipelining	N	N	10000	2.83	100	4*,4+Pipelining	N	N	10000	1.34	100
	N	Y	10000	1.66	100		N	Y	10000	0.66	100
	Y	Y	10000	1.50	100		Y	Y	10000	0.64	100
	Y	N	10000	2.15	100		Y	N	10000	1.32	100
4*,5+Non-pipelining	N	N	10000	5.88	101	4*,5+Pipelining	N	N	10000	1.42	101
	N	Y	10000	1.35	101		N	Y	10000	1.25	101
	Y	Y	10000	0.92	101		Y	Y	10000	1.23	101
	Y	N	10000	1.40	101		Y	N	10000	1.37	101
8*,4+Non-pipelining	N	N	4000	0.32	110	8*,4+Pipelining	N	N	4000	0.05	43
	N	Y	4000	0.24	100		N	Y	4000	0.04	40
	Y	Y	4000	0.21	100		Y	Y	4000	0.04	40
	Y	N	4000	0.32	110		Y	N	4000	0.04	43

Descriptions of "Res", "*", "+", "Pred", etc. are the same as those in Table 1. Number of the final results is limited to around 100

CONCLUSION

In this paper, a fast scheduling chain combination algorithm and a fast algorithm used to predict minimum length of chain combination are presented. They can be used in chain combination based scheduling algorithms with experimental results showing that they can greatly improve the performance of scheduling algorithms. According to the nature of the scheduling algorithm, multiple global optimized results can be obtained in much shorter time, thus other objective functions such as power and register usage can be incorporated into the algorithms to achieve multi-objective optimization.

References

Chantana, C., Wanlop, S., Edwin, S., 2004. Efficient Scheduling for Design Exploration with Imprecise Latency and Register Constraints. Proc. EUC. Aizu-Wakamatsu City, Japan, p.259-270.

Gajski, D., Dutt, N., Pangrle, B., 1986. Silicon Compilation (Tutorial). Proceedings of the IEEE Custom Integrated Circuits Conference. IEEE Computer Society Press, Los Alamitos, California, p.102-110.

Hwang, C.T., Lee, J.H., Hsu, Y.C., 1991. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **10**(4):464-475. [doi:10.1109/43.75629]

Kumar, A., Bayoumi, M., Elgamel, M., 2004. A methodology for low power scheduling with resource operating at multiple voltages. *Integration, the VLSI Journal*, **37**(1):29-62. [doi:10.1016/j.vlsi.2003.09.005]

Lin, Y.L., 1997. Recent development in high level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, **2**(1):2-21. [doi:10.1145/250243.250245]

Memik, S.O., Fallah, F., 2002. Accelerated SAT-based Scheduling of Control/Data Flow Graphs. Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02). Washington DC, USA, p.395-400. [doi:10.1109/ICCD.2002.1106801]

Memik, S.O., Kastner, R., Bozorgzadeh, E., Sarrafzadeh, M., 2005. A scheduling algorithm for optimization and early planning in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, **10**(1):33-57. [doi:10.1145/1044111.1044115]

Mohanty, S.P., Ranganathan, N., 2005. Energy-efficient datapath scheduling using multiple boltages and dynamic clocking. *ACM Transactions on Design Automation of Electronic Systems*, **10**(2):330-353. [doi:10.1145/1059876.1059883]

Mohanty, S.P., Ranganathan, N., Chappidi, S.K., 2006. ILP models for simultaneous energy and transient power minimization during behavioral synthesis. *ACM Transactions on Design Automation of Electronic Systems*, **11**(1):186-212. [doi:10.1145/1124713.1124725]

Narasimhan, M., Ramanujam, J., 2001. A fast approach to computing exact solutions to the resource-constrained scheduling problem. *ACM Transactions on Design Automation of Electronic Systems*, **6**(4):490-500. [doi:10.1145/502175.502178]

Parker, A.C., Pizarro, J., Mlinar, M., 1986. Maha: A Program for Datapath Synthesis. Proc. DAC. Las Vegas, USA, p.461-466.

Paulin, P.G., Knight, J.P., 1989. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **8**(6):661-679. [doi:10.1109/43.31522]

Sllame, A.M., Drabek, V., 2002. An Efficient List-based Scheduling Algorithm for High-level Synthesis. Proc. DSD'02. Dortmund, Germany, p.316-323.

Ullman, J., 1975. NP-complete scheduling problems. *Journal of Computer System Science*, **10**(3):384-393.

Yuan, X.L., Shen, X.B., 1998. A path-based scheduling algorithm. *Computer Research and Development*, **35**(3):279-282 (in Chinese).