Soft Scheduling in High Level Synthesis

Jianwen Zhu, Daniel D. Gajski
CECS, Information and Computer Science
University of California, Irvine, CA 92717-3425, USA
{jzhu,gajski}@ics.uci.edu

Abstract

In this paper, we establish a theoretical framework for a new concept of scheduling called soft scheduling. In contrasts to the traditional schedulers referred as hard schedulers, soft schedulers make soft decisions at a time, or decisions that can be adjusted later. Soft scheduling has a potential to alleviate the phase coupling problem that has plagued traditional high level synthesis (HLS), HLS for deep submicron design and VLIW code generation. We then develop a specific soft scheduling formulation, called threaded schedule, under which a linear, optimal (in the sense of online optimality) algorithm is guaranteed.

1 Introduction

High level synthesis (HLS) accepts a behavioral description, typically a sequential algorithm, and computes an optimal microarchitecture, typically composed of a datapath and a controller, which implements the behavior [1]. HLS task is intractable in nature, hence it is usually decomposed into subtasks of scheduling, register allocation, functional unit binding and interconnect binding. Such strategy of divide-and-conquer, helps to find good solution with reasonable computational cost. One the other hand, it suffers from an intrinsic problem of *phase coupling*, that is, the subtasks, or *phases*, adversely contribute to the optimality of each other, or even worse, invalidate each other.

Unfortunately, the problem of phase coupling becomes more severe when many realistic considerations, which are often omitted by traditional HLS, are factored in. Consider the following scenarios, where we assume traditional scheduling, which assign each operation in the behavior to a fixed time step, is performed before any other tasks (e.g., Figure 1 (b) is an ALAP schedule of the dataflow graph in Figure 1 (a)).

To appear in 36th Design Automation Conference, New Orlean, June, 1999

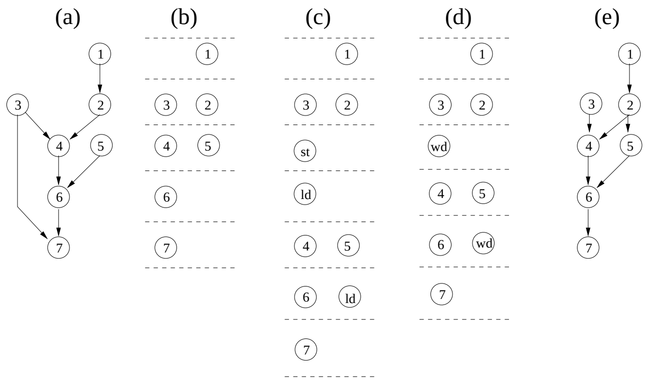

Figure 1: Examples of (a) dataflow graph (b) hard schedule (c) insertion of spill code (d) insertion of wire delay (e) soft schedule.

- **coupling with register allocation:** Traditional HLS assumes all values can be fit into registers or register files in the datapath. In reality, there are only limited number of such resources. Hence, *spilling*, which selectively stores values into background memory, has to be performed when the number of simultaneously alive values exceeds the number of registers available. Spilling effectively changes the original behavior. For example, assume the register allocator chooses to spill the value computed by vertex 3 in Figure 1 (a), then additional node for storing the value to memory and loading the value from the memory has to be inserted accordingly. This inevitably will affect the final schedule. In practice, either inferior result such as the one shown in Figure 1 (c) has to be accepted, or the entire design process has to be iterated. As another example, the ϕ nodes, as artifacts of static single assignment (SSA) analysis [11], can be resolved to either register moves or void operation only after register allocation.

- **coupling with physical design:** Traditional HLS ignores the interconnect delay, an abstraction not valid any more in *deep submicron design*. Unfortunately, the interconnect delay can be determined only after place and route, which in turn can be performed until HLS is performed. For example, if the register which stores the value computed by vertex 3 in Figure 1 (a) is placed far enough from the functional unit which uses its value, additional node representing the wire delay has to be introduced in the dataflow graph. In practice, either a pessimistic estimate of the interconnect delay has to be assumed in order to keep the original schedule valid; or trivial fix of the original schedule such as the one shown in Figure 1 (d) has to be performed, which leads to inferior result; or the entire design process has to be iterated.

More such examples can be enumerated, all of which tend to result in an iterated design process if reasonable good solution is expected. Unfortunately, such iteration is expensive, since it often spans the entire design process, including physical design. More importantly, it might have no guarantee of *convergence* at all. Alternatively, global optimization approaches, which usually reduce the high level synthesis task to a linear integer programming problem, can be used to carry out the subtasks simultaneously. While the exact solution can be found, the problem size which these methods can tackle is limited.

In this paper, we focus on the scheduling techniques which help to alleviate the phase coupling problem without resorting to the exact approaches. The paper makes the following contributions: *First*, a new concept of scheduling, called soft scheduling, is proposed in contrasts to the traditional scheduling algorithms which insist the scheduled operations to be totally ordered. Instead, Soft scheduling assumes a weaker requirement that the scheduled operations only need to be partially ordered. Analogous to soft decoding in digital communication, soft scheduling hence makes soft decision at a time, since the decision can be refined later. The hard decision, or the exact mapping of operations to time steps, can thus be delayed to the desired stage, for example, after place and route is performed. *Second*, we propose a specific soft scheduling formulation, called threaded scheduling, to impose a structure in the partial order of the scheduled operations. Elegant theoretical result can be derived from such a structure, which leads to an algorithm both linear and optimal.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 gives a formal definition of soft scheduling. Section 4 describes threaded scheduling algorithm and proves its correctness, optimality and linearity. Section 5 gives the experimental results.

2 Related Works

Traditional HLS tools or VLIW compilers [1] typically use list scheduling [2] and force-directed scheduling [3], or their variants for resource constrained and timing constrained scheduling. Relative scheduling [4] has the additional capability of scheduling operations with undeterministic delay. Path-based scheduling [5], percolation scheduling , and trace scheduling exploits parallelism beyond the basic blocks.

The phase coupling problem between the HLS subtasks has been noted in several systems and their solutions are to solve all the subtasks simultaneously with an ILP formulation. Among them are the work by Gebotys [6], and the *OSCAR* system [7]. The phase coupling problem between the HLS tasks and the physical design is also addressed in several works. *3D Scheduling* [8] performs binding and floorplaning at the same time. Erwing [9] addresses the problem on a particular VLSI architecture called partitioned bus. *ChipEst* [10] performs HLS tasks with estimation of the physical information.

Our work focuses on removing the coupling of scheduling with other tasks. The goal is achieved neither by performing all the tasks simultaneously as the ILP approach, since it is expensive and unscalable; nor by incorporating an estimate of the possible effect of other tasks, since such effect is difficult to characterize. Instead, we perform soft scheduling to make decisions just necessary for other tasks to proceed. The exact mapping from operations to time steps can then be delayed until all the information, including interconnect delay, is available.

3 Soft Scheduling

Schedulers usually operate within the boundary of the basic block, or in order to increase the parallelism available, the super block. In both cases, the block behavior can be abstracted as a precedence graph, defined as follows:

Definition 1 *A* **precedence graph** *is a directed acyclic graph* $G = \langle V_G, E_G, D_G \rangle$ *where* V_G *is the set of vertices,* $E_G \subseteq V_G \times V_G$ *is the set of edges, and* $D_G : V_G \mapsto \mathcal{I}$ *is the delay function. The* **partial order** $\prec_G \subseteq V_G \times V_G$ *induced by* G *is the transitive closure of* E_G. $\forall v \in V_G$, *its* **source distance** $\| \rightsquigarrow v\|_G$ *is the sum of the delay of all the vertices along the longest path from the primary inputs, or the set of vertices without predecessors, to* v. *Its* **sink distance** $\|v \rightsquigarrow \|_G$ *is the sum of delay of all the vertices along the longest path from* v *to the primary outputs, or the set of vertices without successors. Its* **distance** $\| \rightsquigarrow v \rightsquigarrow \|_G$ *is the sum of delay of all the vertices along the longest path from primary inputs to primary outputs which passes* v. *The* **diameter** $\|G\|$ *of precedence graph* G *is the longest distance of all its vertices.*

The vertices of the graph G correspond directly to the operations in the behavior description, and the edges represent the dependency between the operations. The ultimate goal of scheduling, is then to assign a time step to each operation, such that the total order induced by such a mapping is consistent with the partial order $\prec_G$ derived from the dependency. In this paper, we are interested in the category of scheduling algorithms called procedural schedules, which schedule one operation at a time:

Definition 2 *A* **procedural schedule** *of precedence graph G is a tuple $P = \langle M_P, F_P \rangle$, where the* **meta schedule** *M_P is a sequence over V_G; and F_P is an* **online schedule** *of G.*

According to Definition 2, a procedural schedule consists of two parts: the meta schedule determines the order of operations to feed into the online schedule, while the online schedule schedules one operation at a time. Traditionally, the set of scheduled operations maintained by the online schedule has to be total ordered. Our definition relaxes this assumption:

Definition 3 *An online schedule of precedence graph G is a function $F : V_G \times S_F \mapsto S_F$, where S_F is a set of precedence graphs, called the* **scheduling states***, which satisfy the following:*

1. **initial condition***: $\langle \emptyset, \emptyset \rangle \in S_F$.*
2. **correctness condition***: $\forall S \in S_F, \forall p, q \in V_S$, then $p \prec_G q \rightarrow p \prec_S q$.*
3. **incremental condition***: $\forall S \in S_F$, then $p \prec_S q \rightarrow p \prec_{F(v,S)} q$; and $v \in V_S \rightarrow F(v, S) = S$; and $v \notin V_S \rightarrow V_{F(v,S)} = V_S \cup \{v\}$.*

An online schedule F is said to be **hard** *if $\forall S \in S_F, V_S$ is total ordered. It is* **soft** *otherwise.*

The online scheduler can be considered as an automaton which takes an empty graph as its initial scheduling state. It updates its state every time an operation not already in its state is given. According to Definition 3, the scheduling state maintained by the online schedule needs to maintain only a partial order among the set of scheduled operations, as long as the partial order is *correct*, that is, it is consistent with the partial order of the original precedence graph; and *incremental*, that is, the partial order of the updated state is consistent with that of the original state.

Residing in one extreme of this definition is the traditional scheduler, such as list scheduling and force-directed scheduling, where the scheduling state is totally orderd. The total ordering invariant turns out to be overly restrictive for the later passes of the design process, which motivates the class of schedulers called soft schedulers. The relaxed assumption on the order between the scheduled operations in the soft scheduler contributes to its flexibility, since the partial order can be refined later. For example, the soft schedule shown in Figure 1 (e) represents a partial order subject to refinement, such as the introduction of spill code, register moves or wiring delay. On the other hand, the partial order maintained by a soft scheduler is usually “tighter” than that of the original data flow graph, where the tighter part reflects the design decisions made. For example, the edge between vertex 2 and vertex 5 Figure 1 (e) is an artificial edge, introduced in order to serialize the accesses of the common functional unit shared by vertex 2 and vertex 5. The design of an online schedule is then an art of imposing a structure, or the set of additional invariants that the scheduling state has to hold, to make the desired tradeoff between the flexibility, decision completeness, and complexity of the algorithm.

4 Threaded Scheduling

4.1 Problem Formulation

One specific soft schedule formulation is defined as follows.

Definition 4 *A* **threaded schedule** *of G is an online schedule F whose scheduling state is a* **threaded graph***. A threaded graph is a precedence graph TG whose vertices are covered by a partition $T \subseteq 2^{V_{TG}}$, such that $\forall t \in T, \forall p, q \in t$, either $p \prec_{TG} q$ or $q \prec_{TG} p$. $\forall t \in T$ is called a* **thread***. A threaded graph TG is said to be* **K-threaded** *if $|T| = K$.*

According to Definition 4, there exist a fixed number of threads in the scheduling state of a threaded schedule. Every scheduled operation belongs to exactly one thread. While operations across threads are partial ordered, within a thread the operations are total ordered. In practice, each thread corresponds to one functional unit in the datapath. The task of scheduling one operation then consists of finding the best thread, or the best functional unit, and finding the best position within the thread, or the best way to serialize the access of the functional unit, in order to optimize some figure of merit. An alternatively view of the schedule problem is to *embed* the original precedence graph onto the threaded structure by introducing artificial precedence relationship between operations such that some figure of merit is optimized. For example, Figure 1 (e) is produced by a threaded scheduler for the precedence graph in Figure 1 (a). Here, vertex 3, 4, 6, and 7 belongs to one thread, and vertex 1, 2, and 5 belongs to another thread. From this threaded graph, a hard schedule of 5 states can be constructed. However, if additional spill code as shown Figure 1 (c) is introduced, the resultant threaded schedule leads to a hard schedule of only 6 states. Similarly, if wire delay as shown in Figure 1 (d) is introduced, the resultant threaded schedule leads to a hard schedule of only 5 states.

Definition 5 *A threaded schedule F is said to be* **optimal** *if $\forall S \in S_F, \forall v \in V_S, \forall F' \neq F, \|F(v, S)\| \leq \|F'(v, S)\|$.*

The criterion established by Definition 5 optimizes performance, where performance is measured in terms of the diameter of the scheduling state, or the critical path length. For simplicity of presentation, we assume each function unit can implement all the operations, in other words, an operation can be partitioned to an arbitrary thread. Our results apply equally well when this assumption is relaxed.

4.2 Algorithm Implementation

We present our implementation of the threaded schedule in Algorithm 1 using an object oriented notation for the data structure and a methemetical notation for the algorithm itself. Here, a `ThreadedGraph` object implements a precedence graph denoted by **this**. Each vertex of the graph contains the field in, which points to its immediate predecessors; the field out, which points to its immediate successors; the fields $sdist$ and $tdist$, which record its source distance and sink distance respectively; the field $thread$, which records a number ranging from 0 to $K-1$, to indicate the thread to which the vertex belong; and the field $delay$, which indicates the delay of the vertex. Initially, the graph contains an array (of size K) of vertices, called s, connected to another array of vertices, called t. The graph is updated every time the method $schedule$ is called, with a new vertex v added to the graph, and the edges modified. The $schedule$ method proceeds by first calling the $select$ method, which finds the best position to insert the new vertex; and then the method $commit$, which performs the actual update of the graph.

Based on Definition 5, a naive implementation of the $select$ method would evaluate every position to insert the node by first speculatively updating the graph, and then compute the diameter of the resultant graph. Finally, the position which leads to the smallest diameter is returned. While updating the graph takes $O(|V_G|)$ time, the diameter computation takes $O(|V_G| * |E_G|)$ time, assuming Bellman-Ford algorithm is used. Hence the total time spent on evaluating all the position is $O(|V_G|^2 * |E_G|)$.

We can actually find the best position without the expensive speculation by taking advantage of the special structure inherent in the threaded graph. In Algorithm 1, the $select$ method starts by labeling every vertex with its source distance and sink distance. It then computes the *intrinsic source delay* of the vertex to be added, which is the maximum source distance of its predecessors which are already scheduled. It computes its *intrinsic sink delay* as well. Note that both the intrinsic source delay and sink delay are quantities not dependent on the position to be selected. It then evaluate every position by compute a cost which combines the intrinsic delay information and the delay associated with the position selected. Note that the cost computation can be computed in constant time. The one with the minimum cost can then be selected in linear time. The optimality theorem in Section 4.3 shows that the best position selected according to this algorithm indeed leads to the optimal solution defined by Definition 5.

The $commit$ method first links v, the vertex to be added, into the given thread k at the given position. For every predecessor p of v in the original precedence graph, it then further update the scheduling state if p is already in the state, according to if there exists an edge e from p to a vertex q in thread k: If e exists and it happens that q is before v in thread k, as shown in Figure 2 (a), then the current state remains untouched. On the other hand, if e does not exist, as shown in Figure 2 (b), then an edge from p to v is added. Otherwise, e is replaced by an edge from p to v. Similarly, the current state is updated for every successor of v, according to rules shown in Figure 2 (d)(e)(f). The correctness theorem of Section 4.3 shows that the scheduling state updated in the fashion defined by this algorithm is indeed consistent with Definition 4.

Algorithm 1

```
public class ThreadedGraph {
  static class Vertex {
    Vertex[]  $in$ = new Vertex[$K$];
    Vertex[]  $out$ = new Vertex[$K$];
    int       $sdist = 0$;
    int       $tdist = 0$;
    int       $thread$;
    int       $delay = 0$;
    }
  Vertex[]  $s$ = new Vertex[K];
  Vertex[]  $t$ = new Vertex[K];
  Graph     $G$;

  public ThreadedGraph( Graph $g$ ) {
    $\forall k \in [0, K-1]$ {
      $s[k]$ = new Vertex(); $s[k].thread = k$;
      $s[k].in[k] = \mathbf{null}$; $s[k].out[k] = t[k]$;
      $t[k]$ = new Vertex(); $t[k].thread = k$;
      $t[k].in[k] = s[k]$; $t[k].out[k] = \mathbf{null}$;
      $G = g$;
      }
    }
  void  $commit$( Vertex $pos$, Vertex $v$ ) {
    int   k = pos.thread;
    $v.thread = k$;
    $pos.out[k].in[k] = v$; $v.out[k] = pos.out[k]$;
    $pos.out[k] = v$; $v.in[k] = pos$;
    $\forall p, p \prec_G v$ {
      if( $p.out[k] == \mathbf{null} || v \prec_{\mathbf{this}} p.out[k]$ ) {
        if( $p.out[k] != \mathbf{null}$ )
          $p.out[k].in[p.thread] = \mathbf{null}$;
        $p.out[k] = v$; $v.in[p.thread] = p$;
        }
      }
    $\forall q, v \prec_G q$ {
      if( $q.in[k] == \mathbf{null} || q.in[k] \prec_{\mathbf{this}} v$ ) {
        if( $q.in[k] != \mathbf{null}$ )
          $q.in[k].out[q.thread] = \mathbf{null}$;
        $q.in[k] = v$; $v.out[q.thread] = q$;
        }
      }
    }
  void  $label$() {
    $forwardLabel()$; // $s.t. \forall v \in V_{\mathbf{this}}, v.sdist = \| \rightsquigarrow v \|$
    $backwardLabel()$; // $s.t. \forall v \in V_{\mathbf{this}}, v.tdist = \| v \rightsquigarrow \|$
    }
  Vertex    $select$( Vertex $v$ ) {
    int   $curDelay, bestDelay = INFINITY$;
    int   $sdist, tdist, intrinsicSrcDist, intrinsicSnkDist$;
    Vertex  $cur, best$;

    $label()$;
    $intrisicSrcDist = max_{p \in V_{\mathbf{this}}, p \prec_G v} p.sdist$;
    $intrisicSnkDist = max_{q \in V_{\mathbf{this}}, v \prec_G q} q.tdist$;
    $\forall k \in [0, K-1]$
      for( $cur = s.out[k]$; $cur != t[k]$; $cur = cur.out[k]$ ) {
        $sdist = max(cur.sdist, intrinsicSrcDist)$;
        $tdist = max(cur.out[k].tdist, intrinsicSnkDist)$;
        $curDelay = sdist + tdist + cur.delay$;
        if( $!(v \prec_G cur) \&\& !(cur.out[k] \prec_G v)$
            $\&\& curDelay < bestDelay$ ) {
          $bestDelay = curDelay$; $best = cur$;
          }
        }
    return $best$;
    }
  public void $schedule$( Vertex $v$ ) {
    Vertex  pos;
    if( $v \in V_{\mathbf{this}}$ ) return;
    $pos = select(v)$;
    $commit(pos, k, v)$;
    }
}
```

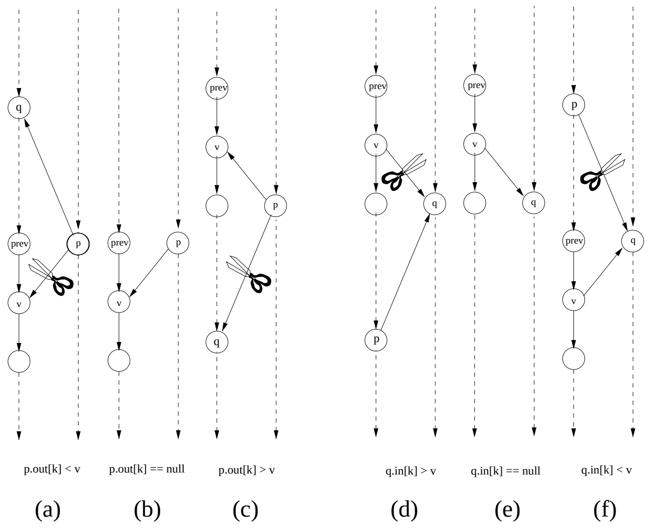

Figure 2: Update of scheduling state.

4.3 Algorithm Analysis

In this section, we study the correctness, optimality and complexity of Algorithm 1. For space reason, Interest readers are referred to [12] for proofs of claims. We first establish the relationship between the precedence graph **this** implemented by Algorithm 1 and our definition of a threaded schedule.

Definition 6 *A function $F : V_G \times S_F \mapsto S_F$ is a schedule of precedence graph G implemented by Algorithm 1 if $\forall S \in S_F$ is formed by the subgraph of* **this** *spanned by $V_{\mathbf{this}} \setminus s \setminus t$.*

4.3.1 Correctness

In order to show Algorithm 1 is indeed a threaded schedule, we first prove a set of lemmas.

Lemma 1 *Let F be the schedule of precedence graph G implemented by Algorithm 1. Then $\langle \emptyset, \emptyset \rangle \in S_F$.* $\Box$

Lemma 2 *Let F be the schedule of precedence graph G implemented by Algorithm 1. And $\forall S \in S_F, v \in V_G$, let $S^{'} = F(v, S)$. Then $v \in V_S \rightarrow S^{'} = S$; and $v \notin V_S \rightarrow V_{S'} = V_S \cup \{v\}$; and $p \prec_S q \rightarrow p \prec_{S'} q$.* $\Box$

Lemma 3 *Let F be the schedule of precedence graph G implemented by Algorithm 1. Then $\forall S \in S_F, \forall p, q \in V_S, p \prec_G q \rightarrow p \prec_S q$.* $\Box$

We can then prove the correctness theorem.

Theorem 1 *Let F be the schedule of precedence graph G implemented by Algorithm 1, then F is a threaded schedule.* $\Box$

4.3.2 Optimality

By definition, a threaded schedule is incremental, we can hence assert that the diameter of its scheduling state is monotonic:

Lemma 4 *Let F be a threaded schedule of the precedence graph G. Then $\forall v \in V_G, \forall S \in S_F, \|S\| \leq \|F(v, S)\|$.* $\Box$

By definition, the distance of a vertex in a precedence graph can be computed as the sum of the maximum of the source distance of its predecessors, the maximum of the sink distance of its successors, and its own delay:

Lemma 5 *Let G be a precedence graph. Then $\forall v \in V_G, \| \rightsquigarrow v \rightsquigarrow \| = D_G(v) + max_{p \prec_G v} \| \rightsquigarrow p\| + max_{v \prec_G q} \|q \rightsquigarrow \|$.* $\Box$

We then make the following observations in Lemma 6, which states that if a new vertex v is scheduled by Algorithm 1, then the source distance of its predecessors will not change their values. Similarly, the sink distance of its successors will not change their values.

Lemma 6 *Let F be the schedule of precedence graph G implemented by Algorithm 1. If $p \in V_S$ and $p \prec_G v$, then $\| \rightsquigarrow p\|_{F(v,S)} = \| \rightsquigarrow p\|_S$. If $q \in V_S$ and $v \prec_G q$, then $\|q \rightsquigarrow \|_{F(v,S)} = \|q \rightsquigarrow \|_S$.* $\Box$

According to Lemma 5, the distance of the new vertex scheduled in the new state can be computed by just looking at the old state. We can hence believe that Algorithm 1 ensures that the distance of the new vertex scheduled is minimum.

Corollary 1 *Let F be the schedule of precedence graph G implemented by Algorithm 1. Then $\forall F^{'} \neq F, S \in S_F, v \in V_g, \| \rightsquigarrow v \rightsquigarrow \|_{F(v,S)} \leq \| \rightsquigarrow v \rightsquigarrow \|_{F'(v,S)}$.* $\Box$

We can then assert the optimality theorem:

Theorem 2 *Let F be the schedule of precedence graph G implemented by Algorithm 1, then F is optimal in the sense of Definition 5.* $\Box$

4.3.3 Complexity

The algorithm left unspecified in Algorithm 1 is $forwardLabel$ and $backwardLabel$, which computes the source and sink distance of each vertex in the scheduling state. We claim that they can be computed in linear time by exploiting the fact that the maximum degree of a threaded graph maintained by Algorithm 1 is K.

Lemma 7 *Let F be the schedule of precedence graph G implemented by Algorithm 1, then $\forall S \in S_F, \forall v \in V_S$, $|(p, v) \in E_S| \leq K$, and $|(v, q) \in E_S| \leq K$.* $\Box$

It is hence trivial to prove the complexity theorem.

Theorem 3 *Let F be the schedule of precedence graph G implemented by Algorithm 1. Then $\forall S \in S_F, \forall v \in V_G$, $F(v, S)$ can be computed in $O(|V_G|)$ time.* $\Box$

BM	Sched. Alg.	Results		
		2+/-,2*	4+/-, 4*	2+/, 1*
	meta sched1	8	6	14
	meta sched2	8	6	14
HAL	meta sched3	8	6	13
	meta sched4	8	6	13
	list sched	8	6	13
	meta sched1	19	11	34
	meta sched2	19	11	34
AR	meta sched3	19	11	34
	meta sched4	19	11	34
	list sched	19	11	34
	meta sched1	19	17	24
	meta sched2	19	17	24
EF	meta sched3	19	17	24
	meta sched4	19	17	24
	list sched	19	17	24
	meta sched1	11	7	19
	meta sched2	11	7	19
FIR	meta sched3	11	7	19
	meta sched4	11	7	19
	list sched	11	7	19

Figure 3: Scheduling results of benchmarks under resource constraints.

5 Experimental Result

The criterion established by Definition 5 is in the sense of online optimality. Schedulers constructed with this criterion only promise the best result for small changes of a schedule. Theoretically, the optimality of a schedule built from scratch, cannot be guaranteed with an arbitary meta schedule. In practice, many meta schedules can lead to results comparable to the traditional list scheduler.

Figure 3 lists the experimental result of several benchmarks by applying Algorithm 1 with different meta schedule. Meta schedule 1 traverses the precedence graph with the depth first order. Meta schedule 2 follows a topological order. Meta schedule 3 partitions the operations into paths, and then feeds the online scheduler with paths ordered by their length. Meta schedule 4 follows an order similar to those determined by the list scheduling heuristics. The experiments are repeated on the benchmarks for different resource constraints. The results are compared with the traditional list scheduler. With few exceptions, we observe that the threaded scheduler is able to achieve the same result as the list scheduler with a number of meta schedules.

6 Conclusion and Outlook

We have presented in this paper a new concept called soft scheduling and theoretical results for a linear, online optimal algorithm called threaded scheduling. While experimental results show that the performance of full schedulers engined with threaded scheduling matches those traditional hard schedulers, our algorithm enjoys unprecedented flexibilities which are valuable in a number of occasions. First, the result of the schedule can be refined and are hence immune to the phase coupling problem or engineering changes. Second, the meta schedule, or the order of operations to feed into our algorithm, is flexible. It can hence be embedded as a kernel into other algorithms which need to take scheduling effect into account, or need to incrementally change the schedule. For example, polynomial time algorithms can be constructed for both the problem of resource constrained technology mapping and resource constrained retiming.

References

[1] D. Gajski, N. Dutt, A. Wu, S. Lin. High Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, 1992.

[2] J. Nestor and D.E Thomas. *Behavioral Synthesis with Interfaces*. Proceedings of the IEEE Conference on Computer Aided Design, November 1986.

[3] P.G. Paulin, J.P. Knight. *Force-Directed Scheduling for the Behavioral Synthesis of ASIC's*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, June 1989.

[4] D. Ku, G. De Micheli. *Relative Scheduling under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits*. IEEE Transactions on CAD/ICAS, Vol. 11, No. 6, April 1992.

[5] R. Camposano. *Path-Based Scheduling for Synthesis*. IEEE Transaction on CAD/ICAS, Vol. 10, No.1, January, 1991.

[6] C.H. Gebotys, M.I. Elmasry. *Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis*. Proceedings of 28th DAC, 1991.

[7] B. Landwehr, P. Marwedel, R. Dömer. *Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming*. Proceedings of EuroDAC, 1994.

[8] J. Weng, A.C. Parker. *3D Scheduling: High-Level Synthesis with Floorplanning*. Proceedings of DAC, 1991.

[9] C. Ewering. *Automatic High-Level Synthesis of Partitioned Busses*. Proceedings of EuroDAC, 1990.

[10] M. Xu, F.J. Kurdahi. *Layout-driven RTL Binding Techniques for High-Level Synthesis*. Proceedings of 9th ISSS, 1996.

[11] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems, October, 1991.

[12] J. Zhu, D.D. Gajski. *Soft Scheduling in High Level Synthesis*. Technical Report ICS-98-37, Information and Computer Science, UC, Irvine, August, 1998.