Static Scheduling of Dynamic Execution for High-Level Synthesis

Yuki Toda Nagisa Ishiura Kousuke Sone

School of Science and Technology
Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, 669-1337, Japan

Abstract— This paper presents a concept of *variable* scheduling for high-level synthesis. While the existing schedulers assume fixed delays for all the operations, the latency of memory accesses and serial multiplication/division, for example, may vary depending on operand values. The new scheduling scheme attempts to exploit the dynamic delay variations; the control step to start each operation's execution is dynamically adjusted depending on the completion signals from the preceding operations. Note that the scheduling is adaptive but static, or precomputed during synthesis, thus the hardware would be simpler than that of superscalar microarchitecture. Experimental results show that the number of the execution cycles are reduced by about 4 to 18%, although the sizes of the finite state machines becomes much larger than that of the conventional fixed scheduling.

I. Introduction

High-level synthesis [1], a technology to auto-generate register transfer level circuit design from behavioral specification in programming language such as C, is now becoming one of the essential tools to expedite large scale VLSI system design.

Scheduling is one of the core processes in high-level synthesis in which the operations in a given program (or a data-flow graph) are assigned to the control steps (execution cycles) so that maximum performance per cost is achieved.

Due to the *static* nature of the scheduling in high-level synthesis, conventional schedulers decide the execution timing of the operations assuming that their delay, or the number of cycles they need to complete their tasks, are fixed. However, there are some operations whose delays may vary dynamically. Memory access operations and serial multiplication/division are such examples. Especially, changes on the number of cycles for memory accesses due to cache hit/misshit (or line/row hit/misshit, in the case of embedded systems) are significantly large. Dynamic scheduling by hardware, as implemented in superscalar microarchitecture, may be a solution for high-end and middle-end computing systems, but not for embedded systems where cost and power consumption for hardware schedulers would be prohibitive.

This paper presents a novel idea of *static scheduling of dynamic execution* of such indefinite latency operations. We assume that completion signals are available from functional units executing indefinite delay operations and we try to synthesize circuits which adaptively adjust the execution timing of operations depending on the completion signals from the other operations.

Our approach is different from high-level synthesis of asynchronous circuits [3, 4] in that circuits synthesized by our method are fully synchronous and the adjustment of timing is carried out by state based controllers.

We formulate the representation of such scheduling in terms of a state transition graph and give a list-based scheduling and binding algorithms for synthesizing register transfer level circuits. Experiments on some DFGs show that the numbers of execution cycles are reduced by about 4 to 18%, though the sizes of the finite state machines to control the datapath have grown by more than 10 times.

In the rest of this paper, we first describes examples of indefinite cycles operations and how such operations are handled in the conventional scheduling algorithms in Section 2. We then present the concept of the variable scheduling and an algorithm for it in Section 3, and show a binding algorithm in Section 4. After describing the experimental result obtained so far in Section 5, we give concluding remarks and future work in Section 6.

II. Indefinite cycle operations and their scheduling

A. Indefinite cycle operations

Memory accesses (load and store) are typical examples of indefinite cycle operations. The number of cycles to complete a load might be one on cache hit but ten on cache misshit, for example. RAMs with a burst mode allows faster access to the data within the same row, where the initial access takes four cycles but the successive read/write to the same row takes one cycle per each word, for example. Other examples are serial multipliers and shift-and-subtract dividors whose latency may vary depending on the values of the operands.

B. Handling of indefinite cycle operations in conventional scheduling

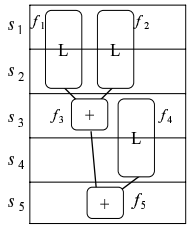
Figure 1 (a) is a simple example of resource constrained scheduling. We assume that "L," a load operation, takes either one or two cycles while addition always takes one cycle. We assume that we have two memory access units and one adder.

One way of handling indefinite latency operation in the conventional scheduler is to assume the worst case delays: Two cycles for every load operation, as in Figure 1 (a). In this case, execution of this DFG always takes five cycles. Even if f_4 finishes its task in a single cycle, f_5 must wait for one cycle to start its execution.

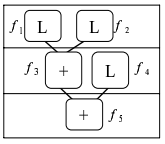
The other extreme is to expect the best case delays, one cycle for every load operation, as shown in Figure 1 (b). In case loads do not finish in one cycle, the controller stalls the entire datapath to avoid data hazards, as in Figure 1 (c).

The latter strategy, however, also fails to give the best scheduling. If we knew the delays for f_1, f_2, and f_4 are 1, 2, and 2, respectively, as in Figure 1 (c), we could start f_4 one step earlier, as in Figure 1 (d), to reduce the total cycles to four.

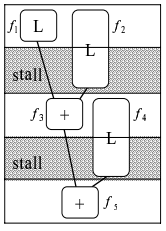
Even though we may assume typical delays or any combination of delays instead of the best/worst case, we can always find an example where scheduling based on the fixed delay model is not powerful enough to give the best scheduling under the existence of indefinite cycle operations.



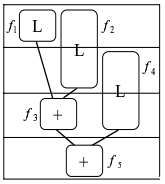
(a) Assuming the worst case delay



(b) Assuming the best case delay



(c) Adjustment by stall



(d) Ideal scheduling for (c)

Figure 1: Conventional scheduling with indefinite cycle operations.

We generalize the conventional high-level synthesis settings so that the delay (in terms of cycles, in this paper) of each operation is given in the form of a list of non-negative integers. For example, delay $< 2, 4, 6 >$ indicates the execution of the operation takes either 2, 4, or 6 cycles. A single element list such as $< 1 >$ implies the delay is fixed. We also assume that each functional unit that executes indefinite cycle operations has a dedicated output port to inform the completion of the operations, which we call a *completion signal*.

In our scheme, the controller adaptively changes the execution timing of the operations depending on the completion signals of the other operations. However, our scheduling is *static* in the sense that the execution is adaptive but the scheduling is precomputed.

The key idea is to represent the adaptive control in the form of a state transition graph (STG). The result of the conventional scheduling, Figure 1 (a) for example, may be regarded as a sequence of states as shown in Figure 2 where each state corresponds to a control step. Sink state s_F is a virtual state, which is introduced for notational convenience. It stands for "the end of this DFG" and corresponds to the initial states of the next DFGs.

An instance of adaptive scheduling is expressed by augumenting the transition edges in the STG with the completion signals: Figure 3 shows an example. It is a variable scheduling for the same setting as in Figure 1, where c_i is the completion signal for f_i.

The initial state s_1 corresponds to the first cycle in the conventional scheduling, which starts execution of f_1 and f_2. If both f_1 and f_2 complete in a single cycle, the next state s_2 is reached by following the edge labeled by $c_1\, c_2$, where execution of f_3 and f_4 starts. If f_4 also finishes in a cycle, s_9 and then s_F are reached. This path corresponds to the best case scheduling depicted in Figure 1 (b). If $\overline{c_1}\,\overline{c_2}$ edge and $\overline{c_4}$ edge are followed from s_1, then it is the worst case (Figure 1 (a)). The more tricky sequence in Figure 1 (d) is represented by path $s_1 \overset{c_1\overline{c_2}}{\rightarrow} s_3 \overset{\overline{c_4}}{\rightarrow} s_8 \rightarrow s_9 \rightarrow s_F$.

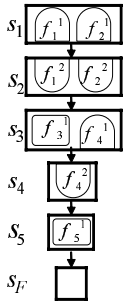


Figure 2: State transition graph of Fig. 1 (a)'s scheduling.

III. Variable scheduling

A. Variable scheduling and its expression

As a new approach to get more efficient scheduling under the existence of indefinite cycle operations, this paper presents a concept of variable scheduling.

B. Formulation of variable scheduling

Let F be a set of operations in a given program or a DFG. Let M be a set of the types of the available

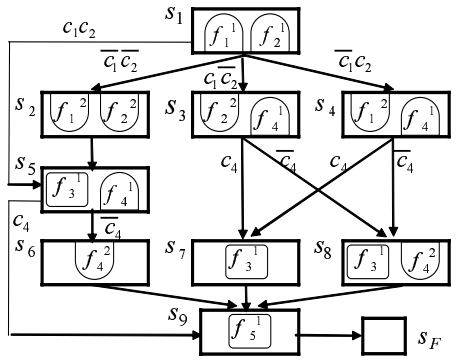


Figure 3: Variable scheduling.

functional units and $n(m)$ be the number of $m \in M$. For operation $f \in F$, $P(f) \subseteq F$ is the set of the operations on which f depends, $D(f) \subseteq Z^+$ is the set of the delays (the number of the possible cycles) for f, and $M(f) \in M$ is the set of the types of the functional units that can execute f.

Given the above settings, variable scheduling computes $< S, \delta, I, m >$ as defined below:

- S is a set of the states, which must contain the initial state s_0 and the final state s_F.
- $\delta(s, Z) \in S$, where $s \in S$ and $Z \subseteq F$, is the state transitioned from s when the set of the operations that complete their execution by the end of s is exactly euqals to Z.
- $I(f) \subseteq S$, where $f \in F$, is the set of states at which the execution of f starts.
- $m(s, f) \in M(f)$, where $s \in S$ and $f \in F$, is the type of functional unit that executes f at s.

We define x and A in order to describe constraints posed on valid state transition graphs.

- $x(s, f) \in Z^+ \cup \{\perp\}$, where $s \in S$ and $f \in F$, indicates that $x(s, f)$-th cycle of f is executed at s, where $x(s, f) = \perp$ means f is not executed at s.
  - $x(s, f) = 1$ if $s \in I(f)$.
  - $x(\delta(s, Z), f) = x(s, f) + 1$ if $f \notin Z$.
  - $x(s, f) = \perp$ otherwise.
- $A(s) \in F$, where $s \in S$, is the set of operations which have completed their execution before s:
  - $A(s_0) = \phi$.
  - $A(\delta(s, Z)) = A(s) \cup Z$.

A valid state transition graph must satisfy the following conditions:

1. $\delta(s_F, Z)$ is undefined for any Z, but $\delta(s, Z)$ must be defined for all the other $s \in S$ for some Z. $x(s_F, f) = \perp$ for all $f \in F$. $\delta(s, Z)$ must not be s_0 for any combinations of s and Z.
2. x and A must be consistently defined for s, s_1, and s_2 where $s = \delta(s_1, Z_1) = \delta(s_2, Z_2)$. That is:
   - If $x(s, f)$ is defined, then $x(s_1, f) = x(s_2, f)$.
   - $A(s_1) \cup Z_1 = A(s_2) \cup Z_2$
3. If $\delta(s, Z)$ is defined, then $x(s, f) \in D(f)$ for all $f \in Z$.

A valid variable scheduling must satisfy the following constraints:

1. Dependency constraint

   $$\forall f \in F, \forall s \in I(f) : P(f) \subseteq A(s).$$

   All the operations on which f depends must have been computed before f starts.

2. Resource constraint

   Let $F(s, m) = \{f \in F \mid x(s, f) \neq \perp, m(s, f) = m\}$. Then,

   $$\forall s \in S, \forall m \in M : |F(s, m)| \leq n(m).$$

   The number of the operations which use the unit of type m at state s is limited.

3. Completion constraint

   $$A(s_F) = F.$$

   All the operations must be computed by this state transition graph.

The scheduling we search for is the one that minimizes the average, or the total number of the cycles. Let $Q_{<S,\delta,I,m>}$ be the set of all the paths (sequences of the states) from the initial state to the final state for scheduling $< S, \delta, I, m >$. Then, our scheduling problem is formulated as to find $< S, \delta, I, m >$ minimizing

$$\sum_{q \in Q_{<S,\delta,I,m>}} |q|$$

under the above constraints.

C. Algorithm for variable scheduling

We have developed an algorithm for variable scheduling which is based on the list-scheduling method [1]. Figure 4 shows its outline.

Function *main* (lines 01–08) calls recursive function *schedule* after initialization. X, Q, C are sets of operations which are being executed, ready, and completed, respectively. $x[f]$ and $m[f]$ indicate which cycle and on which type of functional unit f is being executed (for $f \in X$).

Function *schedule* builds a state transition graph for given X, Q, C, x, m and returns its initial state. Scheduling is computed until both X and Q become empty (line 11). A functional unit to execute each $f \in Q$ is searched, and if it is available, it is scheduled to start execution at this state (lines 12–19). A new state is created for updated status, but no two equivalent states will be instanciated so as to avoid state explosion (lines 20–22). Function *schedule* is recursively called for every possible Z with updated status to find the next state $\delta(s, Z)$.

```
01: void main () {
02:    X = φ;  /* set of executing ops  */
04:    C = φ;  /* set of completed ops  */
03:    Q = {f | P(f) = φ};  /* set of ready ops  */
05:    for (f ∈ F) x[f] = ⊥;  /* f is executing x[f]-th cycle */
06:    for (f ∈ F) m[f] = ⊥;  /* unit type to execute f  */
07:    s0 = schedule(X, C, Q, x, m);
08: }
09:
10: State schedule (X, C, Q, x, m) {
11:    if (X==φ && Q== φ) return sF;
12:    for (f ∈ Q) {
13:       if (some u ∈ M(f) is available) {
14:            x[f] = 0;
15:            m[f] = u;
17:            X = X ∪ {f};
16:            Q = Q − {f};
18:       }
19:    }
20:    for (f ∈ X) x[f]++;
21:    if (there exists t with t =< X, Q, C, x, m >) return t;
22:    s = new State with s =< X, Q, C, x, m >;
23:    for (Z ∈ all possible combinations of ops
24:          which can complete execution) {
25:       X' = X − Z;
26:       C' = C ∪ Z;
27:       Q' = Q ∪ {f | P(f) ⊆ C'} − X' − C';
28:       x' = x;
29:       m' = m;
30:       s' = schedule(X', C', Q', x', m');
31:       let δ(s, Z) = s'
32:    }
33:    return s;
34: }
```

Figure 4: Algorithm of variable scheduling.

IV. Binding for variable scheduling

A. Difference from conventional binding

In the conventional scheduling, a single functional unit is assigned to each operation in the binding process. However, in the case of variable scheduling, an operation may be executed on different functional units depending on states.

Figure 5 illustrates such an example. Suppose a fraction of a state transition graph in Figure 5 (a) is generated by variable scheduling, and we have two units M_1 and M_2 that can execute f_1, f_2, and f_3. Figure 5 (b) is an example of binding for Figure 5 (a). Let f_1 and f_2 be executed by M_1 and M_2, respectively, at state s_1. At state s_2, operation f_3 must be executed on M_1 since M_2 is still executing the second cycle of f_1. On the other hand, at s_3, the same operation f_3 must be executed on M_2 which is the only available functional unit. Thus, f_2 should be executed on different units depending on the states [1].

similar situation occurs on register binding. Figure 5 (c) is an example of register binding for Figure 5(b). Suppose only four registers R_1 through R_4 are available. At state s_1, operation f_1 reads R_1 and R_2, and operation f_2 reads R_3 and R_4. At state s_2, registers R_1 and R_2 must be assigned to the operands of f_3, since R_3 and R_4 are still read by f_2. In contrast, in state s_3, the operands of f_3 must be held in registers R_3 and R_4, instead. Thus, register binding also depends on the states [2].

Note that in our formulation we only determine an input register for each operation, because this automatically determines the output registers. Thus, the output register for each operation depends on the next state, which is determined by combination of the present state and the completion signals.

The circuits synthesized by variable scheduling and binding are basically the same as those generated by the conventional methods. The controller of a circuit delivers control signals to select appropriate operations and input/output registers for each functional unit and to indicate writes to each registers. The only difference is that the completion signals are fed into the controller.

B. State splitting

There are bad news about binding for variable scheduling: The number of the states of STG may increase during the binding due to the need for state splitting. This is caused both by operation binding and by register binding.

(1) State splitting by operation binding

Let us consider the STG in Figure 6 (a) which are obtained by adding s_4 to that in Figure 5 (a). Operation f_3 is executed on different units M_1 and M_2 at states s_2 and s_3, respectively, which incurs a contradiction on reconvergence state s_4. To reconcile this situation, the state s_4 must be split as shown in Figure 6 (b).

(2) State splitting by register binding

The same thing happens during register binding. In Figure 7 (a), f_3 is reading R_1 and R_2 in state s_2 and R_3 and R_4 in state s_3. In order to avoid conflict on the reconvergence, state s_4 must be split as shown in Figure 7 (b).

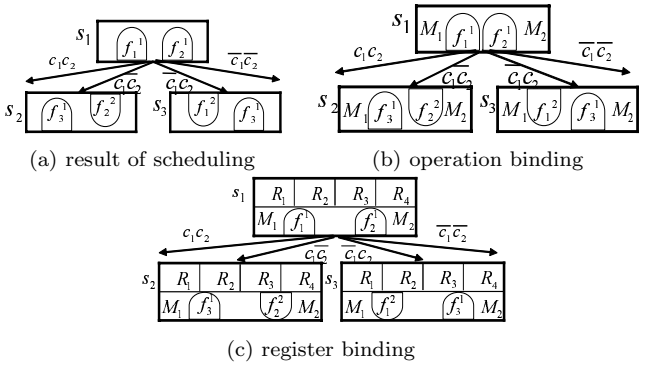
Note that some of such state splitting are inevitable, but others may be avoidable by appropriate operation and register binding. For example, splitting in Figure 7 would be avoided if R_3 and R_4 are used instead of R_1 and R_2 in state s_2. Thus, it is an important goal to find a binding that curve the state increase due to state splitting.

C. Formulation of binding

Let V be the set of all the input and output values (operands) of the operations in F, R be a set of available registers, and PI (PO) be the set of the input (output) ports of the functional units. For operation type $m \in M$, $U(m)$ is the set of the functional units of type m. Let $pi(u,k)$ ($po(u,k)$) be the k-th input (ouput) port. Let $vi(f,k)$ ($vo(f,k)$) be the k-th input (output) operand value to operation f, and $nvi(f)$ ($nvo(f)$) be the number of input (output) operands. Let $u(s,f)$ be the functional

[1] If another unit M_3 were available, the operation binding could be made state-independent by assigning M_3 to f_3.

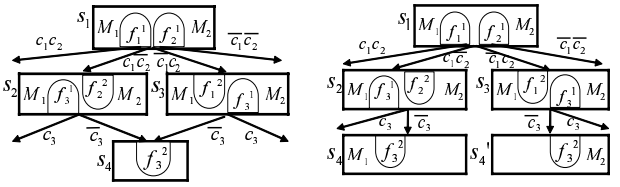
[2] If there were sufficient number of registers, the register binding could be made state-independent by using R_5 and R_6 for input operands of f_3.



(a) result of scheduling (b) operation binding

(c) register binding

Figure 5: variable binding.



(a) before splitting (b) after splitting

Figure 6: State splitting by operation binding.

unit to execute f at s, and $r(s, v)$ be the register to store value v at s. Then biding is to find $u(s, f)$ and $r(s, v)$ for $s \in s$, $f \in F$, and $v \in V$ which satisfies the following constraints:

- $\forall s \in S, \forall f_1, f_2 \in V \ : \ u(s, f_1) \neq u(s, f_2)$.
- $\forall s \in S, \forall v_1, v_2 \in V \ : \ r(s, v_1) \neq r(s, v_2)$.
- Let $s' = \delta(s, Z)$ and $x(s, f) \neq \perp \ \wedge \ x(s', f) \neq \perp$ holds. Then,
  - $u(s, f) = u(s', f)$, and
  - $r(s, vi(f, k)) = r(s', vi(f, k))$ for $1 \leq k \leq nvi(f)$.

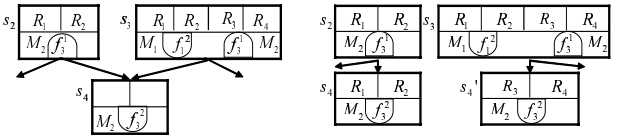
The third constraint is for multicycle operations where the functional units nor the input registers must not change during their execution.

There are various cost functions to minimize, among which we try to curve the number of the interconnections between the registers and the input/output ports of the functional units.

minimize $|C|$, where
$C = \{(r(s, vi(f, k)), pi(u(s, f), k)),$
$(po(u(s, f), l), r(s, vo(f, l))) \ \mid$
$1 \leq k \leq nvi(f), 1 \leq l \leq nvo(f)\}$

D. Algorithm of binding

Figure 8 shows the outline of the binding algorithm we have developed so far. As is described in *main* function (lines 01–04), operation binding and register binding are



(a) before splitting (b) after splitting

Figure 7: State splitting by register binding.

done seperately. This is because we found in our earlier experiment that simultaneous register binding with operation biding would cause tremendous state increase due to state splitting to avoid register conflicts. In function *fu_bind*, the recursion continues until the final state s_F is reached (line 07). An avaialbe functional unit is assigned to each operation f whose execution starts at this state s (lines 08–09). Then, every possible next state s' is tested if it has been already visited. If not, recursion continues on s' with constraints regarding multicycle operation passed to s' (lines 12–15). Otherwise, since the recursion has reached a reconvergence, it checkes for the consistency. If contradictions are found, then the states are split (lines 16–23). On the other hand, in register binding, we use a simpler strategy. We first build for each value v the list of states where v is alive, and then try to find a conflict-free mapping from the set of the values to a set of registers by solving a convering problem by a greedy method.

```
01: void main () {
02:    fu_bind(s0); /* operation binding */
03:    reg_bind(s0); /* register binding */
04: }
05:
06: void fu_bind (State s) {
07:    if (s == sF) return;
08:    for s (such that s ∈ I(f)) {
09:       let u(s, f) be some available unit;
10:    }
11:    for s' (such that s' = δ(s, Z) for some Z) {
12:       if (binding of s' is not yet done) {
13:          copy multicycle info from s to s';
14:          fu_bind(s');
15:       }
16:       else if (binding of s' is contradictory) {
17:          /* state splitting */
18:          s'' = new State;
19:          s'' = s';
20:          overwrite s'' with multicycle info from s;
21:          fu_bind(s'');
22:          let δ(s, Z) = s''
23:       }
24:       else { ; }
25:    }
26: }
27:
28: void reg_bind (State s) {
29:     build ALIVE(v); /* list of the states where v is alive */
30:     solve the covering probem by greedy algorithm;
31: }
```

Figure 8: Algorithm of binding.

Table I: Result of variable scheduling and binding.

DFG	#op	#unit	fixed (max)		fixed (min)		variable scheduling & binding		
		(+, *, M)	#st	#cy	#st	#cy	#st	#cy	CPU (sec)
GSM:Auto	740	(2, 2, 1)	847	847	421	518.16	943	496.11	48.375
GSM:RC	225	(2, 2, 1)	247	247	106	155.01	263	143.00	0.180
GSM:TLAR	70	(2, 2, 1)	54	54	33	48.00	54	45.00	0.997
GSM:QC	413	(2, 2, 1)	561	561	243	294.00	561	270.00	0.514
ellip.c	48	(1, 1, 1)	45	45	28	38.94	134	33.42	0.505
		(2, 2, 1)	35	35	20	29.76	611	25.58	2.919
matrix3.c	114	(3, 3, 1)	306	306	93	122.84	585	106.00	3.275

the number of cycles: +<1>, ∗<2, 3, 4>, M<1, 4>

V. Experiments

A program to compute variable scheduling and biding has been implemented based on the algorithm described so far on Unix (Mac OS X) by Perl (5.8.6). Table I summarizes the results of the experiments. DFGs "ellip.c" and "matrix3.c" are an eliptic filter and 3×3 matrix multiplication. "GSM:Auto," "GSM:RC," "GSM:TLAR," and "GSM:QC" are functions Autocorrelations, Reflection_coefficients, Transformation_to_Log_Area_Rations, and Quantization_and_coding, respectively, taken from the GSM benchmark of CHStone [5].

#op is the number of operations in the DFG, #unit is the list of the numbers of functional units (adders (+), multipliers (∗), and memory access units (M)). The numbers of the cycles are $< 1 >$ (fixed) for addition, $< 2, 3, 4 >$ for multiplication, and $< 1, 4 >$ for memory accesses. We assume the three possibilities of delays for multiplication have equal probability (1/3). On the other hand, we assume a memory access takes one cycle if the address is within the same 256 word row as in the previous access, but four cycles otherwise. The columns "fixed (max)" and "fixed (min)" are results (the number of states, or control steps, and average execution cycles) by the conventional fixed delay scheduling, where the maximum and minimum delays, respectively, are assumed for multiplication and memory accesses. The columns under "variable scheduling & binding" lists the the number of the states (after binding) and the average execution cycles, and the CPU time for the computation (scheduling and binding).

Across all the benchmarks we tried, fixed (min) gave better results than fixed (max). Variable scheduling reduced the execution cycles over fixed (min) by 13 to 18% on ellip.c and matrix3.c. However, the increase in the STG size is prohibitively large. On GSM benchmarks, where parallelism is not very high, the execution cycles are reduced by 4 to 8% while the STG sizes are almost the same as fixed (max).

The CPU time is reasonable for the benchmarks in the table, but there were cases where the CPU time grew prohibitively with the STG size especially during binding.

VI. Conclusion

We have presented a concept of variable scheduling, or precomputed dynamic scheduling, and described its formulation and scheduling/biding algorithms.

There are definitely many things that have to be done to see if this concept would be practical. The STG size in terms of the number of states must be reduced throughout scheduling and biding. The impacts and trade-offs between the performance and the cost of synthesized circuits should be evaluated, and many factors should be taken into account for biding algorithms.

Acknowledgements

Authors would like to express their appreciattion to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Nagoya University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable comments. We would also like to thank to Mr. Yoshitaka Iritani and the other members of Ishiura Lab. of Kwansei Gakuin University.

References

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[2] Yuki Toda: *Dynamic Scheduling for High-Level Synthesis Considering Indefinite Cycle Operations* (in Japanese), Bachelor Thesis, Department of Informatics, Kwansei Gakuin University (Mar. 2008).

[3] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers: "High level synthesis of timed asynchronous circuits," in *Proc. IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 178–189 (Mar. 2005).

[4] A. Prihozhy: "Asynchronous scheduling and allocation," in *Proc. DATE 1998*, pp. 963–964 (Feb. 1998).

[5] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii: "CHStone : A benchmark program suite for practical C-based high-level synthesis," in *Proc. ISCAS 2008* (May 2008).