Scheduling using Behavioral Templates

Tai Ly, David Knapp, Ron Miller, Don MacMillen
Synopsys Inc.
700B E. Middlefield Road
Mountain View, CA USA 94043

Abstract: This paper presents the idea of "behavioral templates" in scheduling. A behavioral template locks several operations into a relative schedule with respect to one another. This simple construct proves powerful in addressing: (1) timing constraints, (2) sequential operation modeling, (3) pre-chaining of certain operations, and (4) hierarchical scheduling. We present design examples from industry to demonstrate the importance of these issues in scheduling.

1.0 Introduction

The task of scheduling [4] is to sequence nodes in a control and data flow graph (CDFG) by assigning each node to a control step (*cstep*). We present the idea of behavioral templates, and describe how we use behavioral templates to address several issues that arise when applying scheduling to commercial designs. For the purpose of this paper, we assume timing constrained scheduling [5].

A behavioral template specifies a relative scheduling among its member CDFG nodes. It is a template in the sense that its member nodes can be treated as a single scheduling unit by assigning the starting cstep for the template. It is behavioral in the sense that it specifies a scheduling pattern as opposed to, for example, a structural pattern [8]. We extend scheduling algorithms to handle behavioral templates by recasting the task of scheduling as that of assigning templates to csteps.

Although a simple idea, behavioral templates provide a powerful way to address four issues in scheduling:

1. **Timing constraints**. We use behavioral templates to impose fixed and maximum timing constraints. This is more efficient than using precedence edges alone because an entire sequence of nodes is considered at once when scheduling one template.
2. **Multi-cycle operations**. To enable scheduling of complex multi-cycle operations, we use multiple CDFG nodes locked in a behavioral template to model the cycle-by-cycle I/O and resource requirements of such operations.
3. **Logic and bit-manipulation operations**. We use behavioral templates to force certain chaining of logic and bit-manipulation operations to save register costs. This reduces the scheduling design space, and therefore run times.
4. **CDFG hierarchy**. We implement hierarchical scheduling by inlining each scheduled subgraph, using a behavioral template to lock the inlined nodes according to the subgraph's schedule.

This paper is organized as follows. Section 2 compares this work to previous research. Section 3 defines behavioral templates. Section 4 describes extending scheduling for behavioral templates. Section 5 discusses applications. Section 6 presents results. Section 7 concludes this paper.

32nd ACM/IEEE Design Automation Conference ®

2.0 Related Work

The term "template" was used in [8] to describe structural patterns to exploit regularity. In [9] and [10], such templates are used to guide the clustering of CDFG nodes into super nodes which map to "regular" subcircuits. Both of these works focus on extracting regular patterns by pattern matching, whereas our work focuses on how to schedule a set of behavioral patterns. Our behavioral templates do not represent *repeating* patterns, but specify local scheduling constraints among CDFG nodes.

Most scheduling systems model multi-cycle operations using single CDFG nodes whose delays are greater than 1. In [7], multi-cycle operations are treated as multiple single-cycle operations. This turns out to be similar to our template-based model for sequential operations, except that we make deliberate use of cycle-by-cycle input/output and resource requirements to model complex operations.

Hierarchical scheduling based on super nodes are used in [9], [6], and [7]. We know of no other system which hierarchically schedules a design while taking advantage of possible resource sharing between nodes and edges in different subgraphs.

3.0 Behavioral Templates

We define a behavioral template, T, as a CDFG object which specifies a set of tuples, (n_i , o_i), where n_i is a CDFG node and o_i is an integer cycle *offset*. The semantics is that T imposes the constraint:

$$\text{schedule}(n_i) = \text{schedule}(T) + o_i \qquad \text{for all } (n_i , o_i) \text{ in } T$$

where $\text{schedule}(n_i)$ and schedule(T) denotes the schedules for n_i and T, respectively.

That is, if T is scheduled to cstep j, then every member node, n_i, of T must be scheduled to the cstep, $j + o_i$. This locks all nodes in T into a pattern of relative schedules, and we may schedule the entire group of nodes by scheduling the template T itself. Fig. 1(a) shows a template, T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }, containing 5 nodes. All CDFG edges have been omitted for clarity. (In the figures, we show behavioral template as a box containing one or more nodes in

slots. The top slot in the box is offset 0, the second slot from top is offset 1, and so on. For example, the node “e” in Fig. 1(a) has offset 5 in T1 because it is in the 6th slot from the top of the box.)

Whenever a node is a member of two or more different templates, we can always merge these templates into one. Consider the templates T1 and T2 in Fig. 1(a) and 1(b). If node g is to be added to template T1 at offset 1, then we merge T1 and T2 into T3 of Fig. 1(c).

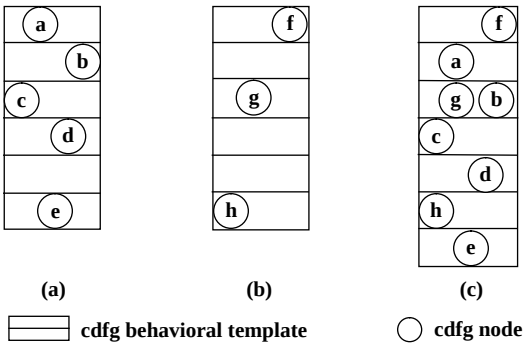

FIGURE 1. Template examples (a) T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }; (b) T2 = { (f,0) (g,2) (h,5) }; (c) T3 = { (f,0) (a,1) (g,2) (b,2) (c,3) (d,4) (h,5) (e,6) }

4.0 Scheduling with Behavioral Templates

Instead of scheduling individual CDFG nodes, we restate the scheduling problem in terms of behavioral templates. Initially, we create one template for every CDFG node, and then merge templates whenever nodes are added to other templates. This ensures that every CDFG node is a member of one and only one template. The timing constrained scheduling task is then to schedule all templates to minimize resource costs subject to timing constraints between templates. This section describes how we extend existing scheduling algorithms for behavioral templates.

4.1 Timing Constraints between Templates

From the CDFG, we construct a weighted, directed graph G=(V , E) where V is the set of all behavioral templates in the CDFG, and E is the set of directed edges between templates. The weight $d(T_x, T_y)$ of an edge $e(T_x , T_y)$ in E specifies the minimum delay between the schedules of T_x and T_y, i.e.,

$$\text{schedule}(T_x) + d(T_x , T_y) <= \text{schedule}(T_y) \qquad \text{Eq. 1}$$

The edges in E are constructed from the data/control dependencies between member nodes in the templates. For every pair of templates, T_x and T_y, $d(T_x , T_y)$ is the maximum value of

$$w(n_i , n_j) + o_i - o_j \qquad \text{Eq. 2}$$

over all (n_i , o_i) in T_x and all (n_j , o_j) in T_y, where $w(n_i , n_j)$ is the minimum cycle delay from node n_i to node n_j.

Note that Eq. 2 can be negative. This means $d(T_x , T_y)$ can be negative, and the graph G is not acyclic. If G contains any cycle of positive lengths, then the timing constraints are unsatisfiable. To check for positive cycles, we solve for the all-pairs-longest-path problem for G using a simple $O(N^3)$ algorithm, where N is the number of templates in G. The longest path lengths are stored in a matrix, LP, for subsequent incremental update of the as soon as possible (ASAP) and as late as possible (ALAP) schedules.

4.2 ASAP and ALAP Schedules

At the start of scheduling, we calculate the ASAP and ALAP schedules for all templates in G, to establish the scheduling time frame for each template. Since G may contain negative weighted edges, we use a relaxation algorithm similar to that in [3] to compute the initial ASAP/ALAP schedules:

(1) Propagate along positive edges in E only;

- for ASAP, propagate forward from the source of CDFG;
- for ALAP, propagate backward from the sink of CDFG;

(2) Relax schedules to satisfy constraints implied by negative edges in E;

(3) Repeat step 1 until no more changes in relaxation step.

When there are no positive cycles in G, the above algorithm is guaranteed to converge in e+1 iterations where e is the number of negative edges in E. The overall computational complexity is $O(N^2e)$ where N is number of templates in G.

The ASAP and ALAP schedules define the initial time frames. Subsequently, as each template is scheduled, we update the time frames of other templates using the longest path lengths matrix, LP:

$$\text{schedule}(T_x) + LP(T_x , T_y) <= \text{schedule}(T_y) \quad \text{for all } T_x , T_y \text{ in V}$$

There is no need for relaxation in this incremental update because LP already takes into account all negative edges in E.

4.3 Cost Functions

We use a number of iterative/constructive scheduling algorithms each of which successively picks an unscheduled template and schedules it to a cstep in its time frame. The algorithms differ in how they pick the next template to schedule, and in how they pick which cstep to schedule the template to. We define the template priority/cost functions in terms of priority/cost functions on the CDFG nodes.

For example, in our implementation of list scheduling, the template priority function is defined as the maximum of its member nodes’ priority values. This gives priority to the template containing the highest priority nodes. In our implementation of greedy scheduling, the incremental cost function for scheduling a template $T=\{(n_i , o_i)\}$ to a cstep j, is defined as the sum total of the incremental costs for scheduling nodes n_i to csteps $j + o_i$.

Scheduling/de-scheduling moves on templates are implemented as moves on their member nodes. All data structures are updated as CDFG nodes are scheduled/de-scheduled. In particular, resource costs for functional units, registers and interconnects are still computed according to the lifetimes and mutual exclusivity of CDFG nodes and edges. This approach is easy to implement and leverages previous work on scheduling CDFG nodes.

4.4 Pre-assigned Operations

Allowing negative edges in G requires that we extend scheduling algorithms to handle maximum timing constraints. This is complicated by “pre-assigned” operations, i.e., operations that are

assigned to specific resources before scheduling. Examples of pre-assigned operations are memory read/write operations for the same RAM. We use a list scheduling algorithm to find an initial legal schedule based on source code ordering. However, list scheduling can fail to find a legal schedule when there are maximum timing constraints. So we augment list scheduling with a recovery step. When list scheduling fails, the recovery step relaxes the template schedules that caused scheduling failures, and iterates:

1. List scheduling step:

Successively consider operations in the ready list in increasing source code ordering. For each ready operation, n_i, check its template, T_x= {...(n_i , o_i)...}, for scheduling in the cstep s - o_i, where s is the current cstep. Postpone scheduling of T_x if any of the following is true:

- T_x has a "relaxed cstep" (see step 2) which is greater than s - o_i
- T_x has no "relaxed cstep", but ASAP is greater than s - o_i
- there is a resource contention if T_x is scheduled to s - o_i

When all nodes have been scheduled, exit with success.

If T_x is postponed due to resource contention, and if s - o_i is greater than or equal to the ALAP cstep for T_x, then list scheduling has failed. When this happens, go to step 2 and try to recover.

2. Recovery step:

When list scheduling fails to find a legal schedule for T_x, we try to recover by increasing its ALAP cstep and rerun list scheduling in step 1. In order to increase the ALAP cstep for T_x, we find all scheduled templates, T_y, for which

$$alap(T_x) == schedule(T_y) - LP(T_x , T_y) \qquad \text{Eq. 3}$$

where alap(T_x) is the ALAP cstep for T_x. For every such template, T_y, we set its "relaxed cstep" to schedule(T_y) + 1. This forces the next run of list scheduling to schedule T_y one cstep later.

This step exits with failure if any of the following is true:

- ALAP(T_x) is at maximum global cstep
- there is no template T_y which satisfies Eq. 3
- algorithm has iterated for N times (N is the # of templates)

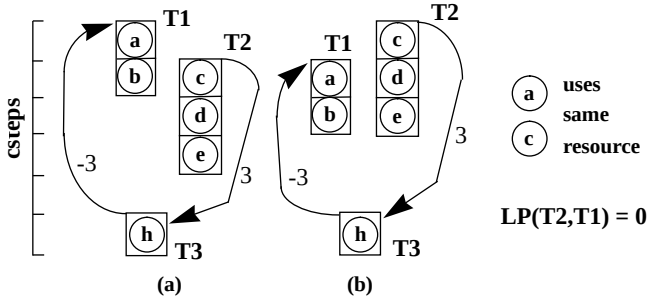

FIGURE 2. Example of list scheduling failure and recovery (a) first iteration fails at T2 (b) second iteration succeeds with T1 relaxed to cstep 1

Fig. 2 shows an example of this algorithm at work. In this example, CDFG nodes "a" and "c" are pre-assigned to the same resource. The source code ordering has "a" before "c" before "f". Initially, list scheduling in step 1 will schedule T1 to cstep 0, and then fails to schedule T2 because of resource contention at cstep 0, and because its ALAP cstep is 0 once T1 is scheduled to 0. In step 2, T1 will be assigned a relaxed cstep of 1. In the next iteration, list scheduling first schedules T2 to cstep 0, then schedules T1 to cstep 1 to avoid resource contention, and finally schedules T3 to cstep 3.

We have two recourses when the above algorithm fails: First, we can continue to try other scheduling algorithms which may still find legal schedules. Second, we can insert precedence constraints to sequentialize pre-assigned operations by their source code ordering. Any unsatisfiable maximum timing constraints would then be detected as positive cycles in the graph G.

5.0 Applications for Behavioral Templates

This section highlights how we use behavioral templates to advantage. Fig. 3 shows the overall flow of our scheduling process.

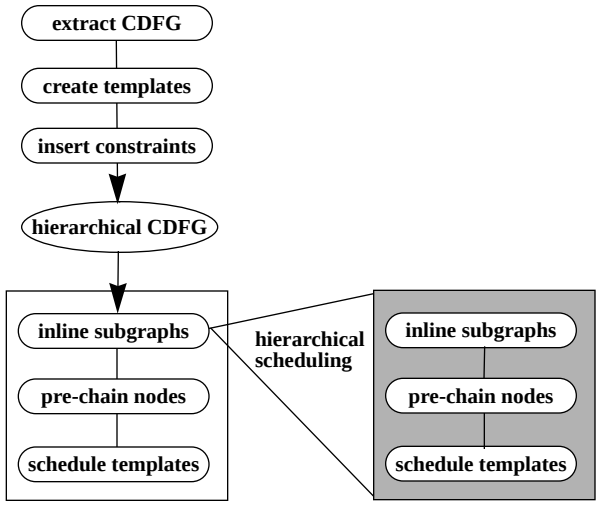

FIGURE 3. Overall flow for hierarchical scheduling

5.1 Inserting Timing Constraints

After extracting the CDFG and creating the initial templates, user-specified timing constraints are added to the CDFG. Minimum timing constraints are represented by precedence edges between nodes, but fixed timing constraints and maximum timing constraints are represented with the help of behavioral templates. Fixed timing constraints are when two or more operations must be scheduled in a fixed number of cycles apart. This is represented by adding one operation to the template of the other operation with the proper offset. For example, if n_j must start k csteps after n_i starts, and if n_i is in template T= {...(n_i , o_i)...}, then we add n_j to T at offset o_i + k (Fig. 4(a)); if n_j must start k csteps after n_i *ends*, and n_i has a delay of d cycles, then we add n_j to T at offset o_i + d - 1 + k (Fig. 4(b)).

However, if n_j must start k csteps after n_i ends, and n_i does not have a static delay (e.g., n_i is a subgraph), then we decompose the fixed constraint into a k-cycle minimum timing constraint from the end of n_i to the start of n_j, plus a k-cycle maximum timing constraint from the start of n_j to the end of n_i. This is shown in Fig. 4(c).

Note that in Fig. 4(c), we create a dummy *place holder* node, ph, and lock it in a template with n_j (k cycles apart). This template combines with the precedence edge of weight 0 from ph to the end of n_i

to represent the maximum timing constraint from the start of n_j to the end of n_i. The precedence edge of weight k from the end of n_i to start of n_j, represents the minimum timing constraint from the end of n_i to the start of n_j.

In general, a maximum timing constraint of k cycles from a set of nodes, A, to the set of nodes, B, is represented by creating two place holders, ph1 and ph2, fixed k cycles apart in a template, and inserting a 0-weight precedence edge from ph1 to all nodes in A, and inserting a 0-weight precedence edge from all nodes in B to ph2. Fig. 9(a) contains an example of this where the dummy place holder nodes, t2 and t3, are used to lock a write operation to 0 cycle after the end of a loop.

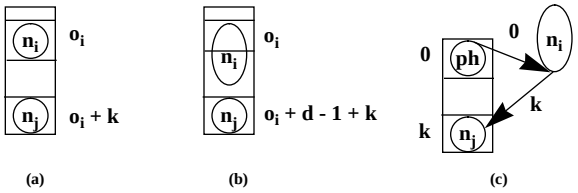

FIGURE 4. Timing constraints (a) n_j starts k cycles after n_i starts, (b) n_j starts k cycles after n_i ends and n_i has static delay d, (c) n_j starts k cycles after n_i ends and n_i's delay is not static.

5.2 Modeling Multi-cycle Operations

Behavioral templates also help model complex multi-cycle operations. When a single CDFG node is used to model a multi-cycle operation, it imposes some limitations due to CDFG semantics:

- Execution cannot start until ALL inputs are available.
- ALL inputs must be held stable throughout operation execution.
- ALL outputs are produced in the last cycle of execution.

This makes it difficult to model, for example, a 3-cycle RAM write operation where the address must be stable for the first two cycles, the data must be stable in the second cycle, and the write sequence finishes in the third cycle. To model such complex operations, we differentiate between **combinational** and **sequential** multi-cycle operations. A combinational operation has cycle synchronous inputs and outputs, so it is modeled by a single CDFG node. A sequential operation can have different cycle-by-cycle input/output connections and even resource requirements, and is modeled by several CDFG nodes that are locked into consecutive csteps by a behavioral template. Fig. 5 shows the single-node and multiple-nodes-in-a-template models for the above 3-cycle RAM write operation. Note that in our model (Fig. 5(b)), the address and data inputs are de-coupled in terms of when and for how long each input must be stable. This also de-couples multiple outputs (if any).

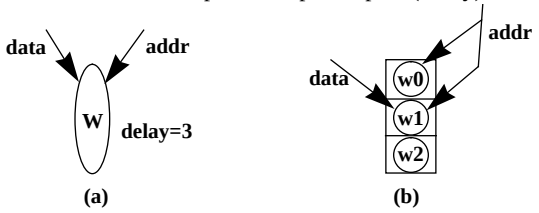

FIGURE 5. Models for a 3-cycle RAM write operation: (a) single node with delay = 3; (b) 3 nodes locked in a template.

This multiple-nodes-in-a-template model is even more powerful when resource requirements are added. For example, a pipelined operation uses different pipe-stage in different cycles, allowing overlapping pipelined operations to share the same hardware module as long as they do not have resource contention on any pipe-stage. If we view pipe-stages as internal resources and assign each pipe-stage a named "token", then we may label each node in the template model with the resource tokens it requires. As a node is scheduled to a cstep, we reserve its resource tokens for that cstep. The number of conflicting tokens (i.e., number of non-mutually-exclusive nodes that require the same token) in any cstep gives the number of pipelined modules needed in that cstep. Overlapping of pipelined operations can be scheduled on the same module because successive nodes in the template model require different tokens.

This removes assumptions about pipelined operations from the scheduling algorithms. We may now model operations on complicated pipelines, and template-based scheduling will properly schedule these operations on the pipelined modules. Fig. 6 shows examples of operations on pipelines with internal feedback, sequential inputs, and multiple outputs. We use "a[s1]" to denote an operation named "a" which requires the token "s1".

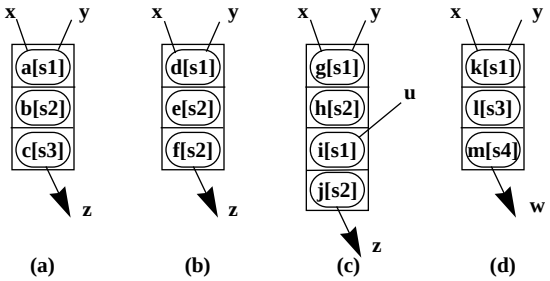

FIGURE 6. Template models for: (a) basic 3-stage pipelined operation, (b) 3-cycle pipelined operation with 2 stages and internal feedback, (c) 4-cycle pipelined operation with 2 stages and sequential inputs, (d) pipelined operation using a different internal path and output port.

Actually, resource tokens need not correspond to physical hardware resources, but may be considered a more general mechanism for specifying how different types of operations can overlap in time on the same module. Consider a 2-cycle RAM which has one read-port and one write-port, whose read/write cycles must be synchronized. Fig. 7 shows how we use resource tokens to specify this constraint to scheduling. If two such operations are pre-assigned to the same RAM, then resource contention on any of "s1", "s2", "s3" and "s4" implies an illegal schedule. The token "s3" prevents read operations for the same RAM to overlap; the token "s4" prevents write operations for the same RAM to overlap; and the tokens "s1" and "s2" prevent read and write operations for the same RAM from being scheduled exactly one cycle apart.

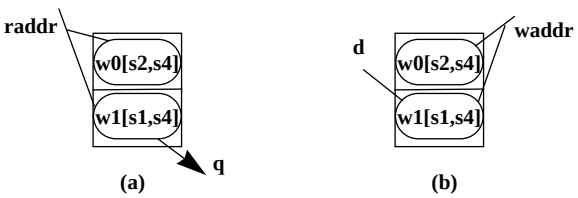

FIGURE 7. Template Models for RAM (a) 2-cycle read, and (b) 2-cycle write.

To handle multi-port RAM's, we allow a module to carry more than 1 copy of a given resource token. For example, to model a 4-port

RAM where each port can be used for both read or write, we would define the RAM module to have 4 “r/w” tokens, and model read and write operation on this RAM to require 1 “r/w” token each. This would allow scheduling to perform up to 4 simultaneous read or write operations on the same module.

5.3 Pre-Chaining

Just before scheduling, we selectively force operation chaining by locking operations in the same cycle using behavioral templates. This “pre-chaining” step reduces scheduling complexity at the expense of scheduling freedom. User-specified chaining directives are applied in this step. We also implement automatic pre-chaining for logic operations and bit-manipulation operations to save registers.

Logic operations include bit-wise AND, OR, NOT, EXOR operations, and bit-manipulation operations include bit-extract, bit-concatenate, constant bit/word generator operations. These operations are good candidates for pre-chaining because they have small propagation delays and they are not resource shared. Thus pre-chaining can be done on the basis of register costs alone. We implement a greedy algorithm for pre-chaining:

1.In a forward traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its predecessors if there are fewer output bits than input bits;

2. In a reverse traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its successors if there are fewer input bits than output bits.

3. Iterate until there are no more changes.

Fig. 8 shows examples of good pre-chaining configurations.

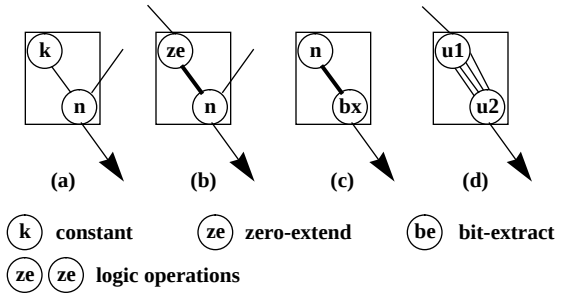

FIGURE 8. Pre-chaining examples: (a) constant with successor, (b) zero-extension with successor, (c) bit-extract with predecessor, (d) multi-input logic with predecessor or multi-output logic with successor

5.4 Hierarchical Scheduling

As shown in Fig. 3, our extracted CDFG is hierarchical, in which each level of the hierarchy corresponds to a loop or a subroutine. Hierarchical scheduling proceeds in a bottom-up traversal of the hierarchy. At each level, instead of representing subgraphs as super nodes, we inline each subgraph and use a behavioral template to interlock the inlined nodes according to the subgraph’s schedule.

Inlining subgraphs allows certain boundary optimizations. First, unused subgraph outputs (and the operations that produce these outputs) can be deleted. This deletion can recurs to unused subgraph inputs and then to operations that feed these inputs. Second, inlining subgraphs allows scheduling of neighboring nodes to take advantage of when individual subgraph inputs/outputs are actually required/produced, whereas representing subgraphs as super nodes would force scheduling to assume that all subgraph inputs/outputs are required/produced in the same cycles.

Another advantage of inlining subgraphs is that scheduling maintains accurate cycle-by-cycle resource costs. This allows, for example, to calculate resource costs of scheduling operations in the first and last cycles of a loop subgraph. (When an outside operation is scheduled in the first/last cycle of a loop it is performed when entering/exiting the loop).

In fact, hierarchical scheduling is used to implement sequential multi-cycle operations. In the initial CDFG, each sequential operation is a subroutine call to some library function, which is a pre-scheduled CDFG whose nodes are labelled with the required resource tokens. During inlining, each sequential operation is replaced by an inlined copy of its function’s CDFG, and a template is created to lock these inlined nodes. This creates the multiple-nodes-in-a-template model for sequential operations.

The disadvantage of inlining subgraphs is that more nodes are scheduled instead of a small number of super nodes. This is balanced somewhat by the fact that inlined nodes are grouped by templates into a few scheduling units, so at least the scheduling solution space is not much bigger.

6.0 Results

Behavioral templates have been implemented in the Synopsys Behavioral Compiler™ product. Behavioral Compiler™ inputs a VHDL or Verilog behavioral description, performs scheduling, allocation, module selection, binding, and control optimization, and outputs a RTL design which is then optimized by RTL optimization [2], FSM optimization, and logic synthesis.

We will use “dft”, a discrete fourier transform design, to illustrate behavioral templates. On reset, dft sequentially reads in the real and imaginary parts of the coefficients into arrays cmem and dmem. These arrays are mapped to the memory “CRAM” (cmem in the lower bank, dmem in the upper bank). It then enters the main processing loop. In each iteration, dft signals it is ready for processing, do a busy wait for the “start” signal, and then sequentially reads in the real and imaginary parts of the data points into arrays amem and bmem, which are also mapped to two halves of a memory, “DRAM”. It then enters two nested FOR loops which compute the discrete fourier transform values and write them out. The memories are two-cycle RAM’s whose read/write models are shown in Fig. 7. The multiply operations are done on a 2-stage pipelined multiply.

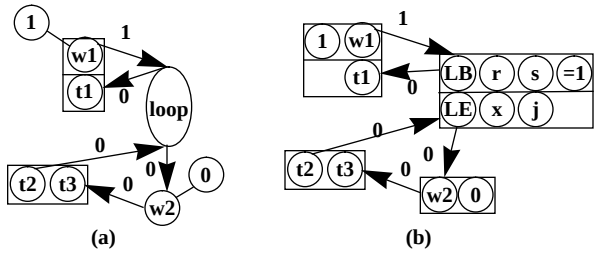

FIGURE 9. Handshaking for start signal: (a) original CDFG with timing constraints, (b) final CDFG scheduled

Fig. 9 shows the CDFG fragment for the busy-wait on the “start” signal. Fig. 9(a) shows the initial CDFG containing the fixed constraints that the “ready” signal be asserted one cycle before the busy wait, and deasserted 0 cycle after the busy wait. Fig. 9(b) shows the same CDFG fragment that is finally scheduled. By this time, hierarchical scheduling has already scheduled the busy wait loop, and the loop body is inlined and locked in a template. Also, pre-chaining has locked the constants with the write operations.

However, the main scheduling problem is in the inner-most loop, which reads from memories the complex data *(a , jb)*, and coefficient *(c , jd)*, and computes *psum* += (*a*c - b*d*) and *ipsum* += (*a*d + b*c*). Table 1 shows the scheduling and allocation results for the computation part of this loop. Note the pipelined RAM read’s are chained with the pipelined multiply operations.

amem	bmem	cmem	dmem	p-mult		+/-
r0[s1]			r6[s1]			
r1[s2]	r2[s1]	r4[s1]	r7[s2]	x0[s1]		
	r3[s2]	r5[s2]		x1[s2]	x2[s1]	
				x4[s1]	x3[s2]	add1
				x5[s2]	x6[s1]	add2
					x7[s2]	sub1
						add3

TABLE 1. Scheduling/allocation results for dft’s inner loop

FIGURE 10. Scheduling/allocation result for computation part of dft’s inner-most loop

In Table 2 and 3, we present some design statistics. #line is number of VHDL/verilog lines in the source. #loop is number of loops with nesting levels in brackets. #node is number of CDFG nodes. #template is total number of templates scheduled. The ratio of nodes to templates are shown in brackets. #RAM is the number of on-chip memories used. #gate is gate count after logic synthesis (excluding RAM’s). Note the difference between #node and #templates.

Table 2 presents several HLSW benchmarks, modified to use more realistic bit-widths. EWF is the fifth order elliptic wave filter example (20 csteps for the main loop, using 1 16x16 pipelined multiply, 2 32-bit adders, 10 32-bit registers and 1 16-bit register). KF is the Kalman filter modified to use 5 RAM’s.

	#line	#loop	#node	#template	#RAM	#gate
EWF	128	2 (2)	103	78 (1.32)	0	9677
KF	207	10 (4)	261	144 (1.81)	5	7963
i8251	596	54 (3)	1238	625 (1.98)	0	5274
gcd	57	2 (2)	77	22 (3.50)	0	825

TABLE 2. Design statistics for several HLSW benchmarks

	#line	#loop	#node	#template	#RAM	#gate
dft	218	5(3)	151	65 (2.32)	2	3920
rgb filter	247	3 (2)	286	109 (2.62)	4	4316
idct	274	8 (3)	902	287 (3.14)	3	32677
viterbi	262	3 (2)	1797	296 (6.07)	0	3653
graphics ctrl	123	2 (2)	98	30 (3.27)	0	4071
high pass	389	5 (2)	310	153 (2.03)	11	8970

TABLE 3. Design statistics for several industrial examples

Table 3 lists several industrial examples. Compared to benchmark examples, these designs tend to:

- have more complicated reset sequences before the main processing loop,
- have more cycle-by-cycle timing constraints on IO operations,
- have more logic operations and bit-manipulation operations,
- use multiple RAM’s or multi-port RAM’s to improve RAM access bottlenecks,
- use pipelined operations to increase throughput.

7.0 Conclusion

In this paper, we have presented our work on scheduling using behavioral templates. The most important value of behavioral templates is that they enable simple solutions to the problems of (1) enforcing fixed and maximum timing constraints, (2) modeling complex sequential operations, (3) pre-chaining of logic and bit-manipulation operations, and (4) hierarchical scheduling. For this reason, behavioral templates have been instrumental in our productization of behavioral synthesis.

Future work will investigate adding structural templates to partition the design based on structural regularity.

8.0 Acknowledgment

We would like to acknowledge Russ Segal and Dennis Fogg, who designed and implemented library interface for our sequential operations. We would also like to acknowledge Pradeep Fernandes and Hazem Almusa for helping us collect the results reported in Tables 1, 2 and 3.

9.0 References

[1] R. Camposano, “Path-based scheduling for synthesis”, IEEE transactions on Computer-Aided Design, vol. CAD-10, pp 85-93, Jan 1991.

[2] B. Gregory, D. MacMillen & D. Fogg, “ISIS: A System for Performance Driven Resource Sharing”, in Proc. of 29th DAC, pp285-290, June 1992.

[3] D. Ku & G. De Micheli, “Relative Scheduling under Timing Constraints”, in Proc. of 27th DAC, pp59-64, June, 1990.

[4] M.C. McFarland, A. C. Parker & R. Camposano, “The High-Level Synthesis of Digital Systems”, in Proc. of IEEE, vol. 78, pp301-318, Feb. 1990.

[5] P. Michel, U. Lauther and P. Duzy, “The Synthesis Approach to Digital System Design”, Kluwer Academic Publishers, 1992.

[6] S. Note, W. Geurts, F. Catthoor & H. De Man, “Cathedral III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications”, in Proc. 28th DAC, pp597-602, June, 1991.

[7] M. Nourani & C. Papachristou, “Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems”, in Proc. 29th DAC, pp99-105, June, 1992.

[8] D. S. Rao & F. J. Kurdahi, “Partitioning by Regularity Extraction”, in Proc. 29th DAC, pp235-238, June, 1992.

[9] D. S. Rao & F. J. Kurdahi, “System Modeling for High-Level Synthesis Using Regularity Extraction”, D. S. Rao & F. J. Kurdahi, in Proc. Sixth International Workshop on High Level Synthesis, pp267-272, Nov 1992.

[10]K. Rose & C. T. Chang, “Cluster-Oriented Scheduling in Pipelined Data Path Synthesis”, in Proc. ICCD, Oct. 1993.