A Fast Heuristic Scheduling Algorithm for Periodic ConcurrenC Models

Weiwei Chen and Rainer Doemer
Center for Embedded Computer Systems
University of California, Irvine, USA
Email: weiwei.chen@uci.edu, doemer@uci.edu

***Abstract*—Embedded system design usually starts from an executable specification model described in a C-based System Level Description Language (SLDL), such as SystemC or SpecC. In this paper, we identify a subset of well-defined C-based design models, called *periodic ConcurrenC models*, that can be statically scheduled, resulting in significant higher simulation and execution speed. We propose a novel heuristic scheduling algorithm that not only is faster than classic matrix-based synchronous dataflow (SDF) scheduling approaches, but also reduces the model execution time by an order of magnitude over the default discrete event simulation.**

I. Introduction

With advanced semiconductor technology, embedded systems gain tremendous functionality by integrating multiple processing elements onto one chip (MPSoC) to perform various algorithms. Although MPSoCs provide promising possibilities for embedded systems to meet the tight real-time constraints, challenges still exist in HW/SW partition, task scheduling, simulation, verification, and synthesis, etc.

C-based system-level description languages (SLDLs), like SystemC [7] and SpecC [6], are available for modeling and describing embedded systems at different levels of abstraction. However, in contrast to the popularity of the C-based SLDLs for system-level modeling and validation, and the presence of early design flows supported by existing tools, the use of a *well-defined* and *consistent system model* is often neglected.

In order to deemphasize the role of the design language being used and emphasize the importance of the *model* instead, we propose a new Model of Computation (MoC) named ConcurrenC [4] that only contains "the best of both" SystemC and SpecC modeling features. As shown in Fig. 1, ConcurrenC essentially contains the intersection of the MoCs that the SystemC and SpecC SLDLs can describe, omitting unnecessary and undesired language constructs (i.e. non-synthesizable and non-verifiable constructs).

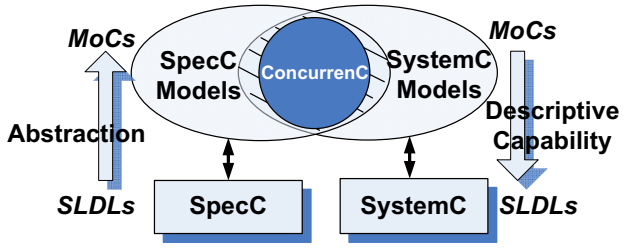


Fig. 1. Relationship between C-based SLDLs: SystemC and SpecC, and MoC: ConcurrenC

A design model described in SystemC or SpecC qualifies as a ConcurrenC model only if it is well-defined and follows ConcurrenC semantics. A ConcurrenC model has a clear separation between computation and communication. In the realm of computation abstraction, a ConcurrenC model consists of blocks, channels and interfaces, and fully supports structural and behavioral hierarchy. Blocks can be flexibly composed in space and time to execute sequentially, in parallel/pipelined fashion, or by use of state transitions. In the realm of communication abstraction, ConcurrenC is intentionally restricted to a set of predefined channels (see Table I) that follow a strict typed message passing paradigm rather than allowing user-defined freely programmable channels.

ConcurrenC Category	Channel Type	Buffer Size	Receiver	Sender
(C)SDF-like	Q_n	n (fixed)	Blocking	Blocking
KPN-like	Q_∞	∞ (infinite)	Blocking	–
ConcurrenC	Q_0	0 (double handshake)	Blocking	Blocking
	Signal	n/a	Blocking	–
	Shared Variable	1 (storage)	–	–

TABLE I
Parameterized Communication Channels

In the broader scope, ConcurrenC is a new MoC specifically designed for the modern C-based SLDLs. As such, ConcurrenC enforces the use of well-defined modeling guidelines and imposes restrictions on the available subset of the used SLDL. As a practical MoC, ConcurrenC aims to be a realistic MoC between fully formal MoCs, such as Kahn Process Network (KPN) [9] and Synchronous DataFlow (SDF) [5], and the fully syntactical SpecC and SystemC SLDLs.

Due to limited page space, we cannot describe ConcurrenC entirely here. We will, for the rest of this paper, restrict our discussion of ConcurrenC features to the execution/simulation and scheduling semantics.

Since ConcurrenC models can be expressed by modern SLDLs, it is obvious that the model can be simulated by the discrete event (DE) simulation mechanism implemented by the native SLDL simulators. This is the default case for executing our ConcurrenC models. However, we will identify in this paper a specific subset of ConcurrenC models, which we call *periodic ConcurrenC models*, that can be statically scheduled. Such static scheduling eliminates the need for many dynamic context switches and thus results in significantly increased simulation and execution speed.

The rest of this paper is organized as follows: research work for static scheduling is briefly reviewed in Section II. In Section III, periodic ConcurrenC models are defined and the ConcurrenC simulation scheme is proposed. Our heuristic static scheduling algorithm is proposed in Section IV first for a SDF-like model, and then extended for CSDF-like ones. Finally, optimization is discussed for performance efficiency. In Section V, experimental results will show that the algorithm works and simulation speed improves if static scheduling is possible for the model. Conclusions and future work are listed in Section VI.

II. Related Work

Popular modeling formalism like Kahn Process Network (KPN) [9] and its extensions are proved to be suitable paradigms to deal

978-1-4244-5766-3/10/$26.00 ©2010 IEEE

with distributed systems and signal processing applications. KPN is a distributed MoC which consists of deterministic processes connected with unbounded FIFO communication channels. Due to the independency on the execution order of the processes, the flexibility of scheduling simplifies the multi-processor architecture mapping of the system.

Synchronous Dataflow (SDF) [5] can be viewed as a special case of KPN which improves the ideal unbounded channel size by fixing the number of the tokens consumed and produced by an actor (process). Cyclo-static dataflow (CSDF) [3] is a more expressive extension of SDF MoC which allows varying rates of token consumptions and productions for different actor invocations. Static scheduling can be performed during compile time for both SDF and CSDF [5], [2]. This also helps in improving the model simulation in SLDLs, e.g. by modifying the simulation kernel [11]. The improvements of the scheduling proposed in this paper are driven by the work of SystemC kernel extension [12] for different MoC modeling and efficient scheduling. However, our work is quite different in the way to achieve this modeling and performance improvement goal. ConcurrenC is a superset of certain MoCs, like KPN, SDF and CSDF, and being an abstract model, ConcurrenC allows direct implementation in SLDL for efficient simulation without modifying the SLDL simulation kernels.

Specifically for this paper, ConcurrenC is a superset MoC of KPN and SDF. According to [10], [2] and [11], SDF and CSDF models are proved to be able to be scheduled statically and the simulation speed increases by applying static scheduling as well when the models are expressed in SLDL. [14] and [15] discuss a technique, implemented in a tool SDF^3, to analyze the throughput and storage capacity requirements of these two models. The buffer minimization problem for the (C)SDF is defined as finding the smallest storage capacity for which the model can run without deadlocks and an infinite execution is possible within this bound. Although this problem is proved to be NP-complete [1], an optimal result can be achieved by pruned state-space exploration for moderately sized models [15].

The basic SDF and CSDF static scheduling algorithm used in [10] and [2] first builds an incidence matrix of the connected model graph, solves the balance equation to get repetition vectors for the actors, and then gets the *periodic admissible sequential schedule (PASS)* if it exists. In this paper, we will propose a new algorithm to find one PASS of (C)SDF without the need to solve the balance equation. Our heuristic algorithm achieves similar low buffer sizes at the same time. Experiments show that our heuristic approach is much faster almost as resource-efficient as the existing ones. In particular, it is even able to find storage bounds for those cases that existing tools cannot handle in reasonable amount of time.

III. ConcurrenC Model Simulation

A flexible and efficient simulation strategy, see Fig. 2, for ConcurrenC models can be achieved considering static scheduling of the periodic ConcurrenC models. For such models, we generate efficient implementation based on the static scheduling results; otherwise, the model will be run by the traditional discrete event simulator. The details about the scheduling algorithm and implementation in SLDL will be discussed in Section IV.

A. Definition of Periodic ConcurrenC Model Subset

A certain kind of ConcurrenC models have a periodic process schedule order. The features of this subset are listed below:

- **not input dependent**. The control flow of all the processes only depends on the internal states of themselves rather than the input data from other eternal processes.

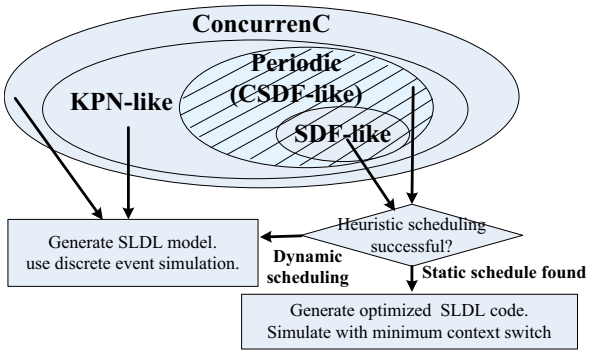


Fig. 2. Simulation Strategy for ConcurrenC Models

- **KPN-like**. The function blocks only communicate via FIFO channels. There are no shared variables and signals between blocks.
- **periodically schedulable**. Formally, $\forall$ process $p_i \in$ the model, $\exists$ a sequence of process state $S_i = s_0, s_1, ..., s_j, ...$ ($j \in \mathbb{N}$), so that $\exists$ T, n $\in \mathbb{N}$, $\forall$ k $\geqslant$ n, $s_k = s_{k+T}$.
  Here s_j is the model state, including the value of internal control variables, the channels that each process is waiting for, and the number of the produced and consumed tokens after the jth process execution iteration.

The first item restricts the possible network state. The last one ensures the existence of a PASS. It is obvious that the consistent SDF and CSDF fit into the periodic ConcurrenC subsets. We call these kind of ConcurrenC models that have PASS **periodic ConcurrenC model**.

IV. Periodic ConcurrenC Scheduling

Any ConcurrenC model can be represented in both SpecC and SystemC SLDLs. Blocks are expressed as $behavior$ in SpecC, and $module$ in SystemC. In SpecC, the execution order is organized by using the keywords seq, par, fsm, $pipe$ for sequential, concurrent, finite-state machine, and pipelined execution, respectively. In SystemC, functional behavior is described as SC_METHOD and SC_THREAD containing sequential statements and having their own execution threads which operate concurrently. Both of the simulation kernels of these two SLDLs are based on C++ discrete-event (DE) simulation according to [16], [13]. DE simulation sometimes suffers from many delta cycles and a large number of context switches between concurrent threads that hinder the simulation speed.

In this section, we are working out that for periodic ConcurrenC model, optimization work could be applied for efficient simulation by eliminating context switches caused by multiple threads created in DE simulation and minimizing the number of delta cycles.

A. Scheduling of SDF-like Models

1) Periodic Admissible Sequential Schedule of SDF: As discussed in Section II, ConcurrenC is a superset of SDF which means some of the ConcurrenC models are SDF-like. Fig. 3(a) shows a simple SDF-like model containing three actors communicating via three channels. It is quite straightforward to describe this model in SpecC SLDL, as shown in Fig. 4.

Algorithm 1 below determines the PASS without creating the incidence matrix of the network and solving the balance equation as in [10] and [11]. Our algorithm is based on the optimistic assumption that a PASS does exist so that for each execution step, as long as the firing rule meets (the number of the input tokens on the input channels are enough), the actor (process) can be a candidate to run next. Priorities are set when multiple actors (processes) qualify to be

(a) SDF-like ConcurrenC Example (b) KPN-like example that qualifies as CSDF-like

Fig. 3. SDF-like and KPN-like ConcurrenC Examples

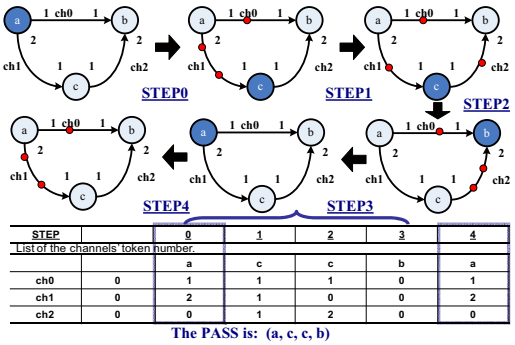
the next candidates. The less number of tokens stored in the channels after the actor fires, the higher priority the actor (process) has. Thus, those processes which just consume but not produce, gathered in set P_e in Algorithm 1, have the highest priority. This heuristically tries to minimize the size of the communication buffers. In order to find the period of the execution order, we store the status of the model for each firing. After each process firing, the function `match()` is called to compare the status with the previous ones and return whether an identical one is found or not.

More specifically, suppose there are **N** processes and **P** channels in model. The data structure we need for the proposed algorithm is (1) a list for the number of tokens stored in the communication channels, **Token[P]**, (2) a static input channel list for each actor (process) with the number of the tokens to be consumed, **chWait[N][P]**, and (3) a hashing table to store the history of the scheduling status. The status of the model is maintained in the **Token[P]** list, which specifies the number of tokens stored in each communication channel. For an SDF-like model, **chWait[N][P]** is static since the numbers of the consumed and produced tokens are fixed.

```
behavior a(i_int_sender ch0,
           i_int_sender ch1)
{
  int i = 0;
  void main()
  {
    while(1){
      ch0.send(i);
      ch1.send(i);
      ch1.send(i);
    }
  }
};

behavior b(i_int_receiver ch0,
           i_int_receiver ch2)
{
  int i0, i1, i2;
  void main()
  {
    while(1)
    {
      ch0.receive(&i0);
      ch2.receive(&i1);
      ch2.receive(&i2);
    }
  }
};

behavior c(i_int_receiver ch1,
           i_int_sender ch2)
{
  int i;
  void main()
  {
    while(1){
      ch1.receive(&i);
      ch2.send(i);
    }
  }
};

behavior Main(void)
{
  c_int_queue ch0(1ul), ch1(2ul), ch2(2ul);
  a b_a(ch0, ch1);
  b b_b(ch0, ch2);
  c b_c(ch1, ch2);
  int main(int argc, char** argv)
  {
    par{
      b_a.main();
      b_b.main();
      b_c.main();}
      return 0;
        }
};
```

Fig. 4. SpecC description of the SDF-like ConcurrenC model

Fig. 5 simulates this algorithm on the example SDF in Fig. 3(a). The PASS of it is: **(a, c, c, b)**.



STEP		0	1	2	3	4
List of the channels' token number.						
		a	c	c	b	a
ch0	0	1	1	1	0	1
ch1	0	2	1	0	0	2
ch2	0	0	1	2	0	0

Fig. 5. Scheduling example for SDF-like model

Note that for the SDF-like models with inconsistent sample rate, the number of the tokens in some communication channel may keep increasing so that the algorithm cannot find an identical status in the history status table. We set a number of maximum schedule step for the termination of the algorithm to handle this case.

Initialization: Scan each processes. If the process has initialization, run it (put it into the scheduling history stack).
$count$ = 1;
while $count < MaxStep$ **do**
  Get the candidate processes' set to be scheduled, P_c={p | there are enough tokens in the channels that p is waiting for.};
  $count++$;
  Separate P_c into two sets, P_e and P_{ne}, where
  Pe = {p | p is $\in P_c \wedge p$ does not produce any token},
  Pne = {p | p is $\in P_c \wedge p \notin P_e$.};
  if P_e is empty **then**
    Get the set P_m, P_m = {p | $p \in P_{ne} \wedge$ the maximum number of the tokens stored in the channels after firing p is the least among those for the other candidates.};
    Select the first $p \in P_m$;
    Schedule p;
    Push p into the scheduling history stack
    update the Token list;
    Store the Token list into the history status hash table;
    if match() **then**
      return the periodic scheduling order (from the stack);
    end if
  else
    while P_e is not empty **do**
      Get one p from P_e;
      Schedule p;
      Push p into the scheduling history stack;
      Update the Token list;
      Remove p from P_e;
      Store the Token list into the history status hash table;
      if match() **then**
        return the periodic scheduling order (from the stack);
      end if
    end while
  end if
end while
return ("failed");

Algorithm 1: Scheduling algorithm for SDF-like models

```
behavior a_seq(i_int_sender ch0,
               i_int_sender ch1)
{
  int i = 0;
  void main()
  {
    //while(1){
      ch0.send(i);
      ch1.send(i);
      ch1.send(i);
    //}
  }
};

behavior b_seq(i_int_receiver ch0,
               i_int_receiver ch2)
{
  int i0, i1, i2;
  void main()
  {
    //while(1)
    //{
      ch0.receive(&i0);
      ch2.receive(&i1);
      ch2.receive(&i2);
    //}
  }
};

behavior c_seq(i_int_receiver ch1,
               i_int_sender ch2)
{
  int i;
  void main()
  {
    //while(1){
      ch1.receive(&i);
      ch2.send(i);
    //}
  }
}

behavior Main(void)
{
  c_int_queue ch0(1ul), ch1(2ul), ch2(2ul);
  a_seq b_a(ch0, ch1);
  b_seq b_b(ch0, ch2);
  c_seq b_c(ch1, ch2);
  int main(int argc, char** argv)
  {
    while(1){
      b_a.main();
      b_c.main();
      b_c.main();
      b_b.main();}
    return 0;
  }
};
```

Fig. 6. Optimized SpecC code for fast simulation

2) SLDL Generation for Fast Simulation: In Fig. 4, each actor (process) in the SDF-like model executes as concurrent behavior in the top $Main$ behavior within the par statement. When simulating with the DE kernel, each actor is wrapped as an individual thread, and communication events between the actors are evaluated and updated for each delta cycle to decide the available thread to run next. Context switches occur when scheduling different threads and often a great number of delta cycles leads to avoidable simulation overhead.

In a sequential execution of the behaviors, context switch can be eliminated since no extra threads needs to be forked for simulation. We generate an equivalent sequential simulation code for the example model Fig. 6. This code can be easily generated by rewriting the implementation code from Fig. 4. The rewritten parts are (1) remove the top iteration loop (e.g. $while(1)$ statement) for each process (if there are initializations before the loop, we split the function into two parts, one for initialization, and one for the loop body), (2) create an infinite loop in the top $Main$ behavior in which the behaviors are run according to the pre-calculated PASS order. The modified code has the same function as the original one but leads to great simulation speed improvement, which will be shown in Section V.

B. Scheduling of Periodic ConcurrenC Models

As discussed in Section III-A, despite not in the SDF-like subset, certain ConcurrenC models still possess the feature of periodical execution, or in other words, have a finite PASS. The first ever KPN example mentioned in [9](Fig.1 and Fig.2) falls into this category. The diagram of this example is shown in Fig. 3(b) and the SLDL implementation is shown in Fig. 7. One PASS of this KPN-like model is (initialization is done by running h(0) and h(1) first): **(f, g, h(0), f, g, h(1))**.

The same algorithm presents in Section IV-A can be applied to find the PASS of this periodic ConcurrenC model by adding two additional data structures: (1) the internal state of each process (e.g. the local variables for control state), and (2) the lists of waiting channels and number of input tokens for each process which will be updated for each firing step. Both of them become part of the status for each execution step and are stored for periodic schedule search.

However, difficulties exist in updating the waiting channel lists of the processes since the input channel for the next execution iteration cannot be predicted in previous execution iterations. In order to update not only the list **Token[P]** but also **chWait[N][P]** dynamically, we apply **Code Slicing** on the input model description for the scheduling algorithm.

The source code is sliced into two pieces, one for updating the numbers of tokens(`_t()`) and one for predicting the token consuming / producing rate of the channels(`_ch()`).

The basic operations for code slicing are **REPLACE**, **REMOVE**, and **KEEP**, as listed below:

- **REPLACE**: replace channel send / receive with **Token[P]** list updating (in `_t()`) and **chWait[N][P]** predicting (in `_ch()`).
- **REMOVE**: (a) the loop statement, e.g. while(1); (b) statements dealing with the data variables.
- **KEEP**: statements dealing with the state variables (in `_t()`) and make these variables *static* to the functional block.

Here, **data variables** refer to those variables which are input dependent or those local temporary variables for computation rather than controlling, e.g. *int i* in *behavior h()* (in Fig. 7); **state variables** refer to those local static variables which are used for determining the control flow inside the functional blocks, e.g. *int b* in *behavior g()* (in Fig. 7). Besides these slicing rules, we create new classes for each actor instance to wrap these sliced codes for scheduling, e.g. *class f_slice* (in Fig. 8).

The details of our code slicing for the KPN example are shown in Fig. 8. The left code pieces (`_t()`) in each slicing group contain the computation of the condition variables and the code for **Token[P]** updating. The right code pieces (`_ch()`) in the slicing group omit the computation of the condition variables but contain the code for **chWait[N][P]** updating. When a process is selected as the next one to run, we first call the code for **Token[P]** updating (`_t()`). After this, the control flow may reach the branch for the next iteration. Then we call the code for **chWait[N][P]** updating (`_ch()`) so that the channel prediction can be achieved.

```
behavior f(i_int_receiver Y,
           i_int_receiver Z,
           i_int_sender W)
{
    int b = 1;
    void main()
    {
        while(1) {
            if(b)
                Y.receive(&i);
            else
                Z.receive(&i);
            X.send(i);
            b = (b + 1) % 2;
        }
    }
};

behavior g(i_int_receiver X,
           i_int_sender T1,
           i_int_sender T2)
{
    int b = 1, i;
    void main()
    {
        while(1)
        {
            X.receive(&i);
            if(b)
                T1.send(i);
            else
                T2.send(i);
            b = (b + 1) % 2;
        }
    }
};

behavior h(i_int_receiver U,
           i_int_sender V, in int init)
{
  int i;
  void main()
  {
    V.send(init);
    while(1) {
      U.receive(&i);
      V.send(i);
    }
  }
};

behavior Main(void)
{
  c_int_queue X(1ul), Y(1ul);
  c_int_queue Z(1ul), T1(1ul), T2(1ul);
  f b_f(Y, Z, X);
  g b_g(X, T1, T2);
  h b_h0(T1, Y, 0);
  h b_h1(T2, Z, 1);

  int main(int argc, char** argv)
  {
    par{
      b_f.main();
      b_g.main();
      b_h0.main();
      b_h1.main();
    }
    return 0;
  }
};
```

Fig. 7. The KPN-like example expressed in SpecC

Note that for the periodic ConcurrenC scheduling, when comparing two status columns of the model (`match()`), the internal condition variables, the process being scheduled, and the number of the tokens in the communication channels are taken into consideration. Identical status occurs as long as the model can be scheduled with bounded buffer size. Otherwise, we may never have the chance to find the repetitive schedule of the periodic ConcurrenC model. Fig. 9 shows the schedule steps for the KPN-like example in Fig. 3(b). Static scheduling is indeed possible and found by our algorithm.

C. Performance Optimization of the Scheduling

The idea we use for optimization is to reduce the time of status comparisons for the scheduling algorithm. We originally compare the new status with the historical ones for each scheduling step. Since the PASS is periodic, we can reduce this to compare the status only every N (N is a number greater than 1) scheduling steps to avoid redundant comparisons.

This optimization technique still ensures to provide a correct PASS finally. Assume that the length of the PASS is l and we find that the current status is identical with the previous ones at the ith scheduling step. This means that the previous status is for the $(i-l)$ scheduling step. Since the scheduling order is periodic, for any status for the jth scheduling step when j is greater than i, it will be identical with the $(j-l)$th one. Therefore, for the tth comparison with N intervals where t is the minimum number to make tN greater than i, we find the status for the tNth step is identical with the $(tN-l)$th one, and thus obtain the PASS. Although the PASS may be different from the original one, the length of it will still be the same.

In turn we reduce the number of comparisons from $i(i-1)/2$ to $t(t-1)N/2-1$, where $(t-1)N < i \leq tN$. For simplicity, we just set N to be the number of the blocks in the model.

Finally, in order to reduce the redundant initial steps before the PASS caused by the optimization technique, we back-trace the status whenever we find the identical one with N intervals to find the first repetitive one.

```
int Token[N], chWait[P][N]; // P = the number of the processes
                            // N = the number of the channels
enum ch                     enum proc
{                           {
  X, Y, Z, T1, T2             h0_proc, h1_proc, f_proc, g_proc
}e_channel;                 }e_process;

class f_slice               class g_slice               class h0_slice              class h1_slice
{                           {                           {                           {
public:                     public:                     public:                     public:
 static int b;               static int b;               static int b;               static int b;
 static void f_t();          static void g_t();          static void h0_t();
static void h1_t();
 static void f_ch(); static void g_ch(); static void h0_ch(); static void h1_ch();
};                          };                          };                          };
int f_slice::b = 1; int g_slice::b = 1;
int h0_slice::flag = 0; int h1_slice::flag = 0;

/********** (a) code slices for process f **********************/
void  f_slice::f_t()              void  f_slice::f_ch()
// update Token[]                 // update chWait[][]
{                                 {
  //while(1){                     //while(1){
      if(b == 0)                      if(b == 0) {
        Token[Y] --;                    chWait[f_proc][Y] = 1;
      else if(b == 1)                   chWait[f_proc][Z] = 0;}
        Token[Z] --;                  else(b == 1) {
      Token[X] ++;                      chWait[f_proc][Y] = 0;
      b = (b + 1) % 2;//}               chWait[f_proc][Z] = 1;} //}
}                                 }
/********** (b) code slices for process g **********************/
void g_slice::g_t()               void g:slice::g_ch()
// update Token[]                 // update chWait[][]
{                                 {
  //while(1){                     //while(1)
      Token[X] --;                //{
      if(b == 0)                      chWait[g_proc][X] = 1;
        Token[T1] ++;               //}
      else if(b == 1)             }
        Token[T2] ++;
      b = (b + 1) % 2; //}
}
/********** (c) code slices for process h0 **********************/
void  h0_slice::h0_t()            void  h0_slice::h0_ch()
//update Token[]                  // update chWait[][]
{                                 {
  if(!flag){                      if(!flag){}
    Token[Y] ++; flag = 1;}       else{
  else{                           //while(1){
  //while(1){                         chWait[h0_proc][T1] = 1;
      Token[T1] --;               //}
      Token[Y] ++; //}            }
  }                               }
}
/********** (d) code slices for process h1 **********************/
void  h1_slice::h1_t()            void h1_slice::h1_ch()
//update Token[]                  // update chWait[][]
{                                 {
  if(!flag){                      if(!flag){}
    Token[Z] ++;flag = 1;}        else{
  else{                           //while(1){
  //while(1){                         chWait[h1_proc][T2] = 1;
      Token[T2] --;               //}
      Token[Z] ++; //}            }
  }                             }
}
```

Fig. 8. Source code slicing for static scheduling

V. Experiments

For our experiments, we use SDF^3 [14], a random synchronous dataflow generation tool, to automatically generate SDF-like and CSDF-like models with different sizes, and a real-world application, JPEG encoder [8] (Fig. 10).

Table II shows the scheduling results of our heuristic scheduling algorithm. The *period* (the number of input/output patterns), *#actors*, and the *sum of repetition vectors* are specified in the configuration file of the (C)SDF generator. Ignoring the timing information, we set the execution time of the actors to $zero$ and the bandwidth of the channels to be inf. We automatically prepare the input data structures for the static schedule algorithm presented in Section IV-A from the model generated and run the algorithm automatically to compare the PASSes. Finally, based on the static scheduling result, we automatically create the optimized implementation of these models in both SpecC and SystemC.

We have simulated all models on a host machine with Intel(R) Pentium(R) 4 CPU at 3.00GHz. As shown in Table II, the result of the *length of PASS* provided by our algorithm matches the original

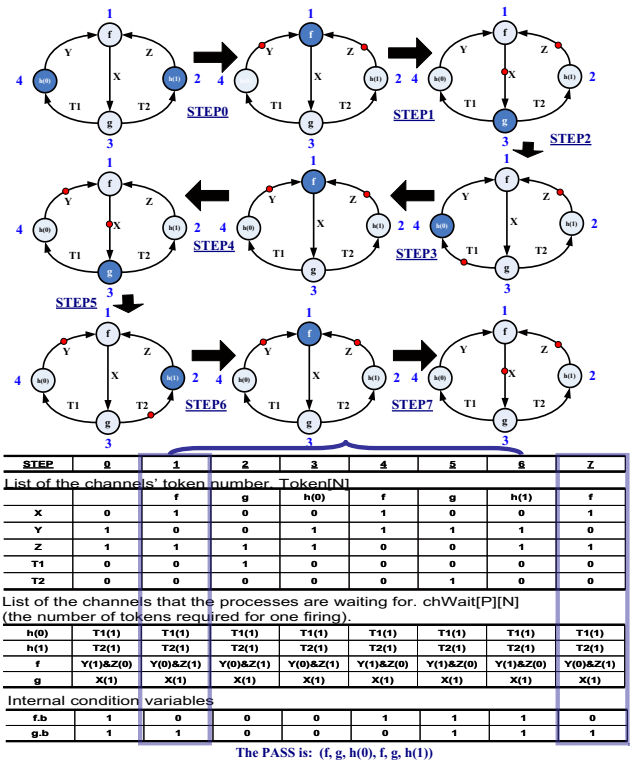


Fig. 9. Static scheduling steps for the KPN-like example

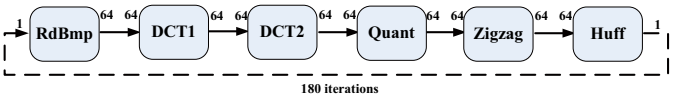


Fig. 10. Diagram of the JPEG encoder example, the PASS is (RdBmp, DCT1, DCT2, Quant, Zigzag, Huff)

specification; *the total buffer size* is not much greater than the optimal ones provided by SDF^3; and the execution time is much quicker. The last five cases did not finish analysis by running the reference SDF^3 tool (exec time $>$ 1 day), but resulted in seconds by applying our algorithm. # initial firings states the number of the initial processes' firings before the beginning of a founded repetition vector.

In Table III, we compare the models between generic SLDL implementations (listed as dyn model) and the optimized implementations (listed as $static$ model). The dyn models regard each process as separate concurrent ones and communicate asynchronously. The $static$ models order the processes according to the PASS and delta cycles and context switch overheads of the DE simulation kernel. Similar to the experimental results discussed in [11], we achieve great speed up by applying this static scheduling to the model simulation. The average speedup is **6.55** for SpecC and **4.10** for SystemC models.

VI. Conclusion and Future Work

In this paper, we have discussed the scheduling of periodic ConcurrenC models which are similar to SDF and CSDF models. We confirm that static scheduling helps to improve the simulation speed of periodic ConcurrenC models.

We have shown a new heuristic static scheduling algorithm without balance equation solving for the static scheduling of the *periodic ConcurrenC models*. Our algorithm is an order of magnitude faster than classic matrix-based algorithms and can handle large models

Example					Proposed Algorithm statistics						Reference SDF^3 statistics		Comparison	
exp	period	#actors	sum rep vector	#chs	length of PASS	#initial firings	total buffer size	exec time (μsec)	opt exec time (μsec)	opt exec speedup factor	total buffer size	exec time (μsec)	speedup factor	buffer size (%)
sdf5	1	5	20	9	20	0	63	63	10	6.30	62	188	18.80	101.6
sdf10	1	10	50	15	50	0	140	106	82	1.29	135	248	3.02	103.7
sdf25	1	25	100	42	100	23	353	575	355	1.62	303	872	2.46	116.5
sdf50	1	50	120	92	120	110	329	1403	937	1.50	313	3446	3.68	105.1
sdf75	1	75	150	132	150	90	425	1638	1481	1.11	404	7587	5.12	105.2
sdf100	1	100	200	257	200	150	869	3703	2637	1.40	827	20413	7.74	105.1
csdf5	2	5	20	7	20	2	16	64	55	1.16	16	1835	33.36	100
csdf10	2	10	40	14	40	14	29	116	102	1.14	25	446783	4380.23	116
csdf25	2	25	100	45	100	154	136	1583	1295	1.22	n/a	>1 hour	>36000	n/a
csdf50	2	50	200	90	200	283	755	11253	9219	1.22	n/a	>1 hour	>36000	n/a
csdf75	2	75	300	138	300	625	685	62615	43267	1.45	n/a	>1 hour	>36000	n/a
csdf100	2	100	400	177	400	514	631	77696	71654	1.08	n/a	>1 hour	>36000	n/a
JPEG Encoder	1	6	6	6	6	0	321	54	53	1.02	321	160	3.02	100

TABLE II
SCHEDULING RESULTS, FOR VARIOUS PERIODIC CONCURRENC EXAMPLES

exp	SpecC (sec)			SystemC (sec)		
	dyn	static	speedup	dyn	static	speedup
sdf5	11.22	2.04	5.50	5.46	1.55	3.52
sdf10	12.34	2.25	5.48	4.88	1.53	3.19
sdf25	11.63	2.52	4.62	6.57	2.06	3.19
sdf50	9.74	0.91	10.70	4.44	0.76	5.84
sdf75	13.95	1.56	8.94	5.69	1.24	4.59
sdf100	17.60	2.44	7.21	7.94	1.89	4.20
csdf5	8.26	1.05	7.87	4.42	1.03	4.29
csdf10	9.66	1.12	8.63	4.68	1.00	4.68
csdf25	11.43	1.30	8.79	5.58	1.16	4.81
csdf50	10.92	3.75	2.91	10.44	3.00	3.48
csdf75	13.04	2.32	5.62	7.46	1.86	4.01
csdf100	31.20	5.71	5.46	13.16	3.96	3.32
JPEG_Encoder	4.64	3.28	1.41	3.15	2.84	1.11

TABLE III
SIMULATION RESULTS, FOR VARIOUS PERIODIC CONCURRENC EXAMPLES

that cannot be handled by the previous algorithm in reasonable time. Based on the result of the static scheduling, efficient model implementations in SLDLs have been generated, significantly improving the model simulation speed.

Future work includes (1) support for simulation timing information in the ConcurrenC blocks, and (2) extend this scheduling strategy into a distributed simulation environment.

ACKNOWLEDGMENT

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Static scheduling of multi-rate and cyclo-static DSP-applications. In *VLSI Signal Processing, VII, 1994., [Workshop on]*, pages 137–146, 1994.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995.

[4] W. Chen and R. Doemer. ConcurrenC: A new approach towards effective abstraction of C-based SLDLs. In *Proceedings of the International Embedded Systems Symposium, "Analysis, Architectures and Modelling of Embedded Systems" (ed. A. Rettberg, M. Zanella, M. Amann, M. Keckeisen, F. Rammig)*, pages 57–65, Langenargen, Germany, 2009. Springer.

[5] E.A.Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[6] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[8] International Telecommunication Union (ITU). *Digital Compression and Coding of Continous-Tone Still Images*, September 1992. ITU Recommendation T.81.

[9] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, pages 471–475, 1974.

[10] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), January 1987.

[11] H. Patel and S. Shukla. Towards a heterogeneous simulation kernel for system-level models: a systemc kernel for synchronous data flow models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(8):1261–1271, Aug. 2005.

[12] H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions For Heterogenous System Modeling: A Framework for Multi-MoC Modeling & Simulation*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[13] N. Savoiu, S. K. Shukla, and R. K. Gupta. Automated concurrency re-assignment in high level system models for efficient system level simulation. In *In Proceedings of Design, Automation and Test in Europe (DATE'2002*, pages 875–889. IEEE Press, 2002.

[14] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[15] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[16] J. Zhu and D. Gajski. Compiling specc for simulation. In *Proceedings of ASP-DAC 2001, Asia and South Pacific Design Automation Conference 2001, January 30-February 2, 2001, Yokohama, Japan*, pages 57–62.