May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models

Weiwei Chen, Xu Han, Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, USA
weiweic@uci.edu, hanx@uci.edu, doemer@uci.edu

***Abstract*—A well-defined system-level model contains explicit parallelism and should be free from parallel access conflicts to shared variables. However, safe parallelism is difficult to achieve since risky shared variables are often hidden deep in the design and are not exposed through simulation. In this paper, we propose a new static analysis approach based on segment graphs that identifies a tight set of potential access conflicts in segments that may-happen-in-parallel (MHP). Our experimental results show that the analysis is complete, accurate and fast to reveal dangerous shared variables in several embedded application models. Compared to earlier work, our approach significantly reduces the number of false conflict reports and thus saves the designer time.**

I. Introduction

Quickly creating clean and parallel system-level models is the *sine qua non* for designing cost-effective MPSoCs in a short time. Model parallelism is particularly important for Electronic System-Level (ESL) design so as to reflect the target architecture that can utilize parallel processing for high performance with low power.

To parallelize an application, system designers first identify suitable functions that can be efficiently parallelized. Then, they recode the model into System Level Description Languages (SLDLs), such as SystemC or SpecC, to expose the potential parallelism. The functionality of the model is typically validated through simulation. However, simulation cannot prove the absence of mistakes in ESL models, such as shared variable race conditions. The reference discrete event simulator for both SystemC and SpecC uses cooperative multi-threading which only executes one thread at one time. In this case, race conditions due to parallel accesses to shared variables hardly "show up" during simulation.

Parallel discrete event simulation in [1], [2] raises the likelihood of exposing parallel access conflicts since it allows multiple threads to run concurrently on multi-core CPUs. However, it is still difficult to reproduce simulation results since the execution order of concurrent threads is nondeterministic and can differ in multiple simulation runs. Most importantly, however, simulation cannot provide a *complete* list of potential access conflicts.

In this paper, we focus on the *safety* of the parallelism in ESL models. A *safe* parallel system model in the context of this paper is a model that is free from parallel access conflicts to shared variables. In particular, we propose a method to analyze the source code of a model at the *segment* level. Our algorithm identifies segments that *may happen in parallel* (MHP) and therefore detects all potential access conflicts with less false positives than previous approaches.

978-3-9815370-2-4/DATE14/©2014 EDAA

A. Related Work

Extensive research has been done for decades to analyze parallel programs.

[3] proposes a general on-the-fly algorithm for access anomaly detection in shared-memory parallel programs. The approach monitors variables and threads dynamically during execution with small storage requirements by using data compression and discarding obsolete information.

[4] presents an algorithm to statically detect race conditions in parallel programs synchronized by using event variables. Based on a graph representing both of the control and synchronization flow, the method determines whether two basic blocks can ever run in parallel. It uses safe distances to determine whether a data array can be accessed by multiple parallel loops. However, when dealing with loops, the analysis requires a special assumption that each loop iteration should use a different event variable.

[5] detects *may-happen-in-parallel* (MHP) statement sets by using a Trace Flow Graph (TFG). Based on rendezvous information, the method iteratively evaluates each TFG node to find the largest MHP set. However, it has limitations on analyzing loops and thus may report over-conservative results.

While these approaches focus on untimed concurrent programs in Ada, C or Java, we focus on timed concurrent models in SLDLs with the discrete event semantics without any limitation on the control flow.

[6] analyses non-concurrency information for OpenMP programs. The approach constructs OpenMP control flow graph and partitions the program into phases. The analysis is based on high level language semantics.

RacerX [7] detects races and deadlocks in operating systems. The algorithm is based on lockset analysis and uses heuristics to reduce false positives and negatives. Since its target system is general multithreading, user annotation is required to specify lock and unlock functions.

In contrast, our analysis is based on low level synchronization primitives and does not require any user annotation.

Formal model checking based approaches are used in [8] and [9] to detect races in SystemC models. These methods use tracing results which combine static analysis and simulation to provide comprehensive coverages. A dynamic partial order reduction technique is proposed to address the state explosion problem. [10] also uses simulation to analyze non-deterministic anomalies among parallel logical processes.

In contrast, our approach is fully static and does not require simulation.

II. Race Condition Detection for ESL Models

Race conditions among shared variables may cause non-deterministic behavior in parallel models and errors in the final implementation.

Race condition detection can be dynamic or static.

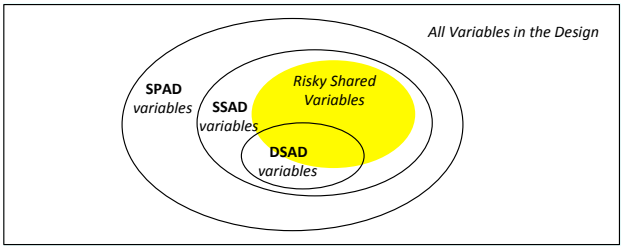
- **Static Analysis** extracts parallelism at the source code level to detect the potential shared variable parallel access conflicts. [11] proposes a **Static Parallelism Aware Detection (SPAD)** which derives the parallelism according to the concurrent syntax in SpecC, namely keywords *pipe* (pipelining execution) and *par* (parallel execution). For instance, Fig. 1 shows a simple design with a two stage pipeline (i.e. *a* and *b*), and each stage consists of two parallel submodules (i.e. *c* and *d* in *a*; *e* and *f* in *b*). **SPAD** first identifies the MHP module pairs hierarchically, such as (*a*, *b*), (*c*, *e*), (*c*, *f*), (*d*, *e*), (*d*, *f*). Then, the algorithm compares the variable access lists for each instance for potential shared variable conflicts.
- **Dynamic Analysis** simulates the model and detects race conditions at run-time. Parallel anomalies can be found precisely along the execution path. In addition to [8] and [9], [12] presents a **Dynamic Segment Aware Detection (DSAD)** approach which analyzes the model statically and monitors the simulation at runtime at the **segment** (the piece of code between two scheduling points) level. In particular, **DSAD** records the segment pairs that are running in parallel during simulation and reports the shared variable conflicts by comparing the segment variable access lists computed statically.

```
behavior A(...)
{
  ...
  void main(){
    par{
      c.main();
      d.main();
    }
  }
};

behavior B(...)
{
  ...
  void main(){
    par{
      e.main();
      f.main();
    }
  }
};

behavior Main()
{
  A a(...);
  B b(...);
  int main(){
    pipe{
      a.main();
      b.main();
    }
  }
};
```

Fig. 1. Example of a pipelined parallel SpecC design

Formal analysis for parallel programs synchronized by the rendezvous mechanism to determine the pairs of statements which *may happen in parallel* (MHP) is known to be NP-complete [13]. For this reason, much research aims to find the sets of MHP statements in larger granularity to reduce the algorithm complexity. However, conservative static approaches usually report a large amount of false positive results while dynamic checking may slow down the simulation and complete coverage is not guaranteed.

In this paper, we also follow the philosophy of detecting race conditions by identifying MHP statements. In Section III-B, we propose a **Static Segment Aware Detection (SSAD)** algorithm to analyze the model at the segment level which is more efficient and comprehensive than **DSAD**, and more precise than **SPAD**. Fig. 2(b) compares the coverage among **DSAD**, **SPAD**, and **SSAD**.



(a) Static and dynamic approaches for detecting parallel access conflicts

	DSAD	SPAD	SSAD
Speed	Slow	Fast	**Fast**
Coverage	Sparse	Comprehensive	**Comprehensive**
Accuracy	Low	Low	**Medium**

(b) Qualitative comparison of approaches

Fig. 2. Comparison of race condition detection approaches

III. Race Condition Analysis using Segment Graphs

We now propose the **Static Segment Aware Detection (SSAD)** algorithm for parallely accessed variables. The basic idea is to identify MHP segment pairs statically with respect to discrete event semantics.

A. Segment Graph Data Structures

To formally describe our SSAD algorithm, we need the following notations:

- **Simulation Time**: We define time as tuple (t, δ) where t=time, δ=delta-cycle, and order time stamps as follows:
  - **equal**: $(t_1, \delta_1) = (t_2, \delta_2)$, iff $t_1 = t_2$, $\delta_1 = \delta_2$
  - **before**: $(t_1, \delta_1) < (t_2, \delta_2)$, iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$
  - **after**: $(t_1, \delta_1) > (t_2, \delta_2)$, iff $t_1 > t_2$, or $t_1 = t_2$, $\delta_1 > \delta_2$
- **Segment** seg_i: source code statements executed by a thread between two scheduling steps. Note that segment has larger granularity than single statements.
- **Segment Boundary** v_i: SLDL statements which call the scheduler, i.e. *wait*, *waitfor*, *par*, *pipe*, etc.
- **Segment Graph (SG)**: SG=(V, E), where V = $\{v \mid v$ is a segment boundary$\}$, E=$\{e_{ij} \mid e_{ij}$ is the set of statements

```
1: int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2: int x = 0, y = 0, z = 0, w = 0; //global variable
3: behavior A(event e)
4: {
5:   void main(){
6:     int i = 0;                    // stack variable
7:     for(i=0;i<9;i++ ){  // i(RW)
8:       y = x + 27;                 // x(R), y(W)
9:       waitfor 1;                  // segment boundary
10:      w ++;                       // w(RW)
11:      wait e;                     // segment boundary
12:    x = array[i]*42;}} // i(R), array(R), x(W)
13: };
14: behavior B(event e)
15: {
16:   void main(){
17:     int i = 0;                    // stack variable
18:     for(i=0;i<9;i++ ){  // i(RW)
19:       y = y*42 + z;               // y(RW), z(R)
20:       waitfor 2;                  // seg boundary
21:       array[i] = array[i]*4 + x++; // i(R), array(RW), x(RW)
22:       notify e;                   // notify event
23:       wait e;                     // segment boundary
24:       z ++;}                      // z(RW)
25:   }
26: };
27: behavior Main()
28: {
29:   event e;    // event instance
30:   A a(e);     // behavior instantiation
31:   B b(e);     // behavior instantiation
32:   int main(){
33:     while(1){  // segment boundary
34:       par {
35:         a;
36:         b; }    // segment boundary
37:     }
38:   }
39: };
```

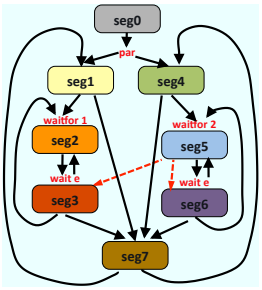
Fig. 3. SLDL source code for a simple design example

between v_i and v_j, where v_j could be reached from v_i, and $seg_i = \cup e_{ij}\}$. The Segment Graph can be derived from the Control Flow Graph (CFG) of the SLDL models [14].

- **Current time advance table** $CTime[N]$ lists the time increment that a thread will experience when it enters the given segment. N is the total number of segments. The time increase for different segments is listed in Table I.

TABLE I
TIME ADVANCES AT SEGMENT BOUNDARIES.

Segment boundary	Time Increment		Add to (t', δ')
wait event	inc by one delta cycle	(0:1)	$(t', \delta' + 1)$
waitfor t	increment by time t	(t:0)	$(t' + t, 0)$
par/end of par	no time increment	(0:0)	(t', δ')



segment	0	1	2	3	4	5	6	7
CTime	N/A	(0:0)	(1:0)	(0:1)	(0:0)	(2:0)	(0:1)	∞

segment	0	1	2	3	4	5	6	7
VAList		x(R) y(W)	w(RW)	array(R) x(RW) y(W)	y(RW) z(R)	array(RW) x(RW)	y(RW) z(RW)	∞

Fig. 4. Segment Graph, current time advance table and segment variable access lists for the simple example

seg	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5				T			T	
6								
7								

(a) Event notification table

seg	0	1	2	3	4	5	6	7
0	∞	(0:0)	(1:0)	(1:1)	(0:0)	(2:0)	(2:1)	(0:0)
1	∞	∞	(1:0)	(1:1)	∞	∞	∞	(0:0)
2	∞	∞	(1:0)	(0:1)	∞	∞	∞	(0:1)
3	∞	∞	(1:0)	(1:1)	∞	∞	∞	(0:0)
4	∞	∞	∞	∞	∞	(2:0)	(2:1)	(0:0)
5	∞	∞	∞	∞	∞	(2:0)	(0:1)	(0:1)
6	∞	∞	∞	∞	∞	(2:0)	(2:1)	(0:0)
7	∞	(0:0)	∞	∞	(0:0)	∞	∞	∞

(b) Segment shortest path table

Fig. 5. Event notification table and segment shortest path table for the simple example

Table I also shows the result of adding the time increment to a given timestamp (t', δ').

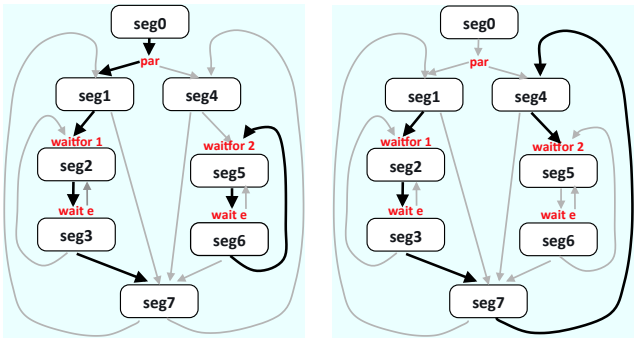
- **Event notification table** $NTab[N, N]$:

$$NTab[i,j] = \begin{cases} T & \text{if } seg_i \text{ notifies an event that} \\ & seg_j \text{ is waiting for;} \\ F & \text{otherwise.} \end{cases}$$

- **Shortest path table**, $SPTab[N, N]$ stores the minimum time advance between two segments:
  $SPTab[i, j]$ = the minimum time advance from seg_i to seg_j.
  Note that this value can be the accumulated current simulation time advances along a *valid* path from seg_i to seg_j (For *valid* path, see Lemma II below).
- **Variable Access List:** $VAList_i$ is the list of the variables that are accessed in seg_i. Each entry for a variable in this list is a tuple of (*Var, AccessType*).
- **May happen in parallel table** $MHP[N, N]$:

$$MHP[i,j] = \begin{cases} T & \text{if } seg_i \text{ may happen in parallel with } seg_j; \\ F & \text{otherwise.} \end{cases}$$

Fig. 3 shows the source code of a simple SpecC design with its segment graph and analysis tables shown in Fig. 4 and Fig. 5.



(a) Two valid execution paths (b) An invalid execution path

Fig. 6. Valid and invalid execution paths in the segment graph when calculating the minimum time advance

B. Determining MHP Segments

We propose an elimination-based algorithm for computing the MHP segments. Initially, all segments are MHP with all others. We use the following lemmas to eliminate MHP segment pairs so as to reduce the number of false conflicts.

Lemma I: $\forall seg_i$, seg_i is only executed by one thread.

Lemma I states that a segment cannot execute in parallel with itself because a segment can only be executed by one thread. When a module (i.e. behavior or channel in SpecC) is instantiated multiple times, multiple copies of the segments

are added to the segment graph [14]. Therefore, one segment only belongs to one specific instance and different instances never share same segments.

Note that a channel instance may be used by multiple threads in the same segment. However, channel accesses are always mutual exclusive in SpecC. This guarantees that a segment in a channel instance is never executed by multiple threads at the same time. ■

Lemma II: If $\exists$ a *valid execution path* from seg_i to seg_j in the **Segment Graph**, seg_i and seg_j will not be executed in parallel. A *valid execution path* is a path that does not exit and re-enter a parallel statement. Fig. 6(a) shows two valid execution paths ($seg_1 \rightarrow seg_7$ and $seg_5 \rightarrow seg_5$) and Fig. 6(b) shows an invalid one ($seg_1 \rightarrow seg_5$).

Lemma II states that sequentially executed segments cannot run in parallel. A *valid execution path* in the segment graph indicates sequentiality. The only exception would be an *invalid* path which exits a par statement from a branch (a child instance) and re-enters another branch (a different child instance) in the same par statement (i.e. $seg_1 \rightarrow seg_5$ in Fig. 6(b)). ■

Lemma III: $\forall seg_i$ where seg_i notifies seg_j, if $\exists seg_k$ which happens either before all seg_i or more than (0, 1) time cycles after all seg_i on the path between them, then seg_k does not happen in parallel with seg_j.

If seg_i notifies an event seg_j is waiting for, seg_j will be executed (0, 1) time cycles after seg_i finishes. Therefore, any segment seg_k (k may equal to i) that is before seg_i or more than (0, 1) time cycles away from seg_i, will not not happen in the same simulation cycle with seg_j.

Particularly, if seg_i belongs to a loop and the time advance for seg_i which goes back to itself is greater than (0, 1) time cycles, seg_i cannot happen in parallel with seg_j. When multiple seg_is exist (an event can be notified by statement(s) belonging to multiple segments), seg_k are compared with all seg_is to determine whether seg_k and seg_i cannot happen in parallel.

In summary, Lemma III is general enough to cover all models with loops and event notifications. ■

Lemma IV: If seg_i and seg_j share the same parent,

1) the parent segment starts a par statement (i.e. seg_1 and seg_4 in Fig. 4), then seg_i will *always happen in parallel (AHP)* with seg_j;
2) the parent segment starts a $pipe$ statement, then seg_i will never happen in different simulation cycle with seg_j within the same pipeline iteration.

Lemma IV defines the segment pairs that *alway-happen-in-parallel* (AHP) according to the execution semantics. Note that *pipe* is a special case of *par*. Although the first segment in each pipeline stage does not happen all the time while the pipeline is filling or flushing, they either AHP or never happen for a specific iteration. Therefore, we categorize the first segments following a *pipe* statement the same as those of the *par* statement. ■

Corollary IV.1: If seg_i and seg_j AHP, $\forall seg_k$ on a valid execution path in the segment graph with seg_i, seg_k will not happen in parallel with seg_j.

Corollary IV.1 states that all the segments that execute sequentially with seg_i in different simulation cycles will never happen in parallel with any AHP segments of seg_i. ■

C. MHP Algorithm for Race Condition Analysis

Our race condition analysis consists of three main steps:

1) Building the segment graph: we first go through all statements in the design, find the segment boundaries, and construct the segment graph similar to [14].

2) MHP segment elimination: In contrast to **SPAD**, we aim at minimizing MHP segment pairs.

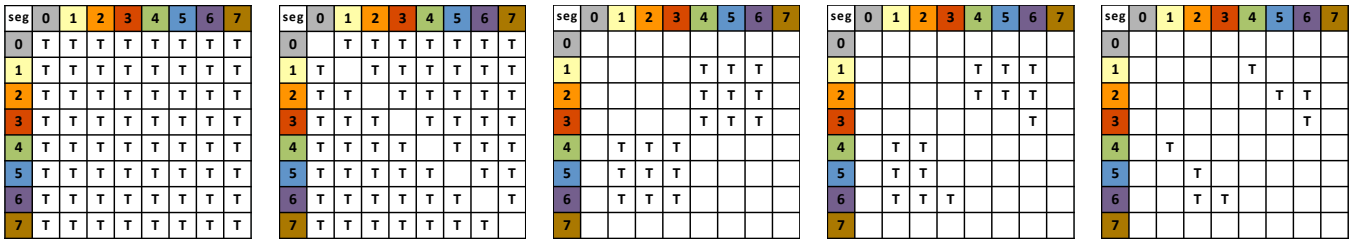
Our *MHP* table is initially filled with *true (T)* value, which means all segments are conservatively assumed to be potentially in parallel (Fig. 7(a)).

Next, we apply the Lemmas presented in Section III-B to eliminate false positives as follows:

- Lemma I eliminates the *T*s on the diagonal of the *MHP* table, as shown in Fig. 7(b).
- Lemma II eliminates the segment pairs that are on the same valid sequential execution path.
  The $SPTab$ reflects the execution order among the segment pairs:
  - $SPTab[i,j] \neq \infty$ and $SPTab[j,i] \neq \infty \Rightarrow$ seg_i and seg_j are a loop (i.e. seg_2 and seg_3 in Fig. 5(b));
  - $SPTab[i,j] \neq \infty$ and $SPTab[j,i] = \infty \Rightarrow$ seg_i will happen before seg_j (i.e. seg_0 and seg_7 in Fig. 5(b));
  - $SPTab[i,j] = \infty$ and $SPTab[j,i] \neq \infty \Rightarrow$ seg_i will happen after seg_j (i.e. seg_5 and seg_4 in Fig. 5(b));
  - $SPTab[i,j] = \infty$ and $SPTab[j,i] = \infty \Rightarrow$ seg_i and seg_j may happen in parallel (i.e. seg_1 and seg_4 in Fig. 5(b)).

  Algorithm 1 shows our modified **Floyd−Warshall** algorithm to compute the $SPTab$. As shown in Fig. 5(b), we get the $SPTab$ for the simple example in Fig. III-A. Then, we use the $SPTab$ to eliminate segment pairs from the *MHP* table and get Fig. 7(c).
- Lemma III eliminates the MHP pairs based on the semantics for event notifications (Algorithm 2).
  We get Fig. 7(d) as the *MHP* table for the simple example. For instance, (seg_3, seg_4) are removed from being MHP since seg_4 will at least be (2:0) time cycles away from seg_5 which is longer than the time advance (0:1) between seg_3 and seg_5.
- Lemma IV eliminates MHP pairs based on the AHP segments (Algorithm 3). In the simple example, since seg_1 and seg_4 AHP (both starting from the par statement in line 34) and seg_5 and seg_4 are not MHP (Lemma II), seg_1 and seg_5 thus will never be MHP as shown in Fig. 7(e).

3) Race condition variable analysis: With the reference of the *MHP* table, we compare the variable access lists of the MHP segment pairs for potential variables that may cause race conditions due to parallel accesses.



(a) Initial MHP table (b) Lemma I applied (c) Lemma I and II applied (d) Lemma I, II and III applied (e) Lemma I, II, III, and IV applied

Fig. 7. MHP table generation by applying Lemma I through IV

Algorithm 1 Modified **Floyd−Warshall** algorithm for computing the $SPTab$

```
1:  let SPTab be a NxN array of minimum time advances initialized to ∞.
2:  let Next be a NxN array of integers initialized to -1 (record the shortest path).
3:  void BuildSPTab(){
4:    ∀ edge from seg_i to seg_j,
5:      SPTab[i, j] = CTime[j]; /* time advance when entering seg_j. */
6:    for k from 0 to N-1 do
7:      for i from 0 to N-1 do
8:        for j from 0 to N-1 do
9:          if (SPTab[i, k] + SPTab[k, j] < SPTab[i, j] then
10:           oldVal = SPTab[i, j];
11:           SPTab[i, j] = SPTab[i, k] + SPTab[k, j]; Next[i, j] = k;
12:           if (IsValidPath(i, k, j)==false) then
13:             SPTab[i, j] = oldVal; Next[i, j] = -1; end if
14:         end if
15:       end for
16:     end for
17:   end for
18: }
19:
20: bool IsValidPath(i, k, j){
21:   /* check whether the shortest path from seg_i to seg_j through seg_k is valid */
22:     path = FindPath(i, k) + FindPath(k, j);
23:     if (path exits a par or pipe statement first and then re-enters the same one)
24:       then return false; end if
25:     return true;
26: }
27:
28: string FindPath (i, j){
29:   if (SPTab[i,j] == CTime[j]) then return "ij"; end if
30:   tmp = next[i, j];
31:   if (tmp == -1) then return ""; end if
32:   return FindPath(i, tmp) - string(tmp) + FindPath(tmp, j);
33: }
```

Algorithm 2 Lemma III for MHP segment elimination

```
1: ∀ seg_j and seg_i, if NTab[i][j] = true /*seg_i notifies seg_j*/
2:   then ∀ seg_k
3:     if ((SPTab[i, k] > (0, 1) and SPTab[i, k] ≠ ∞)
4:       or (SPTab[i, k] == ∞ and SPTab[k, i] ≠ ∞)) then
5:           MHP[k, j] = MHP[j, k] = false end if
6:   end if
```

Algorithm 3 Lemma IV for MHP segment elimination

```
1: ∀ seg_i and seg_j, if (seg_i and seg_j are AHP) /*always happen in parallel*/
2:   then
3:     ∀ seg_k, if (MHP[k, i] = false) then
4:       MHP[k, j] = MHP[j, k] = false; endif
5:     ∀ seg_k, if ( MHP[k, j] = false) then
6:       MHP[k, i] = MHP[i, k] = false; endif
7:   end if
```

In particular, if both lists contain entries of the same variable, and at least one of the access types is write or read-write, then this is reported as a potential race condition.

Table II shows the analysis results after applying the MHP algorithms. It clearly shows that the lemmas help to reduce the MHP segments and thus narrow down the potential conflicts to just y (line 2) for the example shown in Fig. 3.

TABLE II
EXPERIMENTAL RESULTS FOR THE SIMPLE EXAMPLE

Analysis Steps	#MHP Segment Pairs	Potential Conflict Variables	
		numbers	variable list
Initial	36	5	x, y, z, w, array
Lemma I	28	4	x, y, z, array
Lemma I, II	9	3	x, y, array
Lemma I - III	7	2	x, y
Lemma I - IV	4	1	y

IV. EXPERIMENTS AND RESULTS

We implemented and performed the three approaches, DSAD, SPAD and SSAD, on our in-house ESL models for seven embedded applications. In Table III, we show the size of the applications, the number of parallel accessed variables, and the execution time of the analysis for each approach.

Overall, the MHP analysis approach, i.e. SSAD, reports more accurate and complete results than DSAD and SPAD with a very short execution time. Note that, while it is hard to measure, it can easily be seen that the reduced set reported race conditions (due to less false positives compared to SPAD) translates directly into significant savings in the system designer's analysis, testing, and debugging time.

The last four columns in Table III also show the effectiveness of the lemmas in Section III on narrowing down the set of potential conflict variables in the application models.

- **Mandelbrot Renderer**: a graphics application which visualizes the points of the *Mandelbrot set* with 16 parallel slice renderers. The 2 conflict variables reported by SSAD, $image$ and t, are array variables. $image$ can be resolved by splitting it into dedicated slices for each parallel rendering unit. t is a timestamp in the test bench which is safely set by the stimulus and read by the monitor (false positive).
- **JPEG image encoder**: encodes a color image with 3 parallel encoders and 1 sequential *Huffman* encoder at the end. The 3 reported variables $input$, ch, and fin are all pointer variables which are never accessed in parallel. SSAD could avoid reporting these variables with further pointer analysis (future work).
- **Fixed-point MP3 audio decoder**: MP3 audio decoder with stereo channel decoders using fixed-point calculations. 2 variables, $file\_handle$ and $decend$ are reported by SSAD. $file\_handle$ is a pointer reported due to the lack of pointer

TABLE III
EXPERIMENTAL RESULTS FOR EMBEDDED APPLICATIONS

Embedded Applications	Lines of Code	Number of Segments	#Potential Race Conditions			Analysis Time (sec)			#Potential Race Conditions after Applying Lemma				
			DSAD	SPAD	SSAD	DSAD	SPAD	SSAD	none	I	I-II	I-III	I-IV
Mandelbrot graphics	0.6k	38	1	2	**2**	22.31	0.01	0.01	3	3	2	2	2
JPEG image encoder	2.5k	47	1	5	**3**	1.79	0.02	0.02	21	10	5	5	3
MP3 decoder (fixed point)	7k	21	1	47	**2**	4.31	0.08	0.07	48	39	2	2	2
MP3 decoder (floating point)	14k	66	2	54	**9**	75.33	0.23	0.22	60	45	9	9	9
GSM vocoder	16k	50	2	141	**24**	0.78	0.08	0.08	145	108	25	24	24
H.264 video decoder	40k	54	78	183	**162**	79.82	0.93	1.46	209	207	170	170	162
H.264 video encoder	70k	251	128	503	**151**	1531.22	12.24	31.75	504	440	151	151	151

analysis. $decend$ is a debugging variable in the test bench which does not affect the safety of the design.

- **Floating-point MP3 audio decoder**: MP3 audio decoder based on floating-point operations. 9 variables are reported by SSAD, 7 of them can be avoided with further pointer analysis (future work); $hybrid\_blc$ can be resolved by splitting it into two pieces for the two stereo channels, and $decend$ is a debugging variable in the test bench which does not affect the safety of the design.
- **GSM Vocoder**: Global System Mobile (GSM) vocoder whose functionality is defined by the European Telecommunication Standards Institute (ETSI). 24 variables are reported by SSAD, 4 of them can be resolved by using proper channels, and 18 can be avoided with further pointer analysis. The $Overflow$ variable can be resolved by being localized to a stack variable; the Old_A variable can be resolved by being duplicated for parallel modules.
- **H.264 video decoder**: H.264 video decoder with 4 parallel slice decoders and sequential slice dispatcher and synchronizer. SSAD reports 162 conflicting variables including 21 global variables and 141 pointers. The global variables contain 1 counter for debugging purposes and 20 values constant to each frame, so they are shared safely among parallel threads. Parallel access to the pointers could be eliminated with further analysis.
- **H.264 video encoder**: H.264 video encoder with parallel motion estimation distortion calculation. Among the 151 variables reported by SSAD, 64 are pointer variables that can be eliminated with further pointer analysis. This model is under development. For this industrial-size model with more than 70k lines of code, SSAD narrows the size of the potential conflicts set down to 87 which need further investigation for a safe parallel model.

V. CONCLUSIONS AND FUTURE WORK

Writing well-defined and safe ESL models with explicit parallelism is difficult. Parallel accesses to shared variables pose an extra challenge as they are often hidden deep in the model and cause problems that are difficult to capture during simulation.

In this paper, we propose the *may-happen-in-parallel* analysis for discrete event execution semantics by using the segment graph of a system-level design. The approach enables the fast yet complete detection of potential conflicts due to parallel accesses to shared variables. It helps the designer to target dangerous shared variables with very few false positives and ensures a safe design. Our experimental results show the effectiveness of revealing risky shared variables in existing industrial-sized embedded applications in very short execution time.

In future work, we plan to integrate pointer analysis support for variables of pointer types and develop automatic algorithms to assist in resolving the reported conflicts.

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] W. Chen, X. Han, and R. Dömer, "Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment," *IEEE Design and Test of Computers*, vol. 28, pp. 20–31, May/June 2011.

[2] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis*, pp. 241–246, 2010.

[3] E. Schonberg, "On-the-fly detection of access anomalies," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[4] V. Balasundaram and K. Kennedy, "Compile-time detection of race conditions in a parallel program," in *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, pp. 175–185, 1989.

[5] G. Naumovich and G. S. Avrunin, "A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel," *ACM SIGSOFT on Software Engineering Notes*, vol. 23, pp. 24–34, Nov. 1998.

[6] Y. Lin, "Static Nonconcurrency Analysis of OpenMP Programs," in *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, pp. 36–50, 2008.

[7] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 237–252, Oct. 2003.

[8] N. Blanc and D. Kroening, "Race Analysis for SystemC using Model Checking," *ACM Transactions on Design Automation of Electronic Systems*, vol. 15, June 2010.

[9] A. Sen, V. Ogale, and M. S. Abadir, "Predictive Runtime Verification of Multi-processor SoCs in SystemC," in *Proceedings of the Design Automation Conference (DAC)*, 2008.

[10] C. Schumacher, J. Weinstock, R. Leupers, and G. Ascheid, "Scandal: SystemC Analysis for Nondeterminism Anomalies," in *Forum on Specification and Design Languages*, 2012.

[11] X. Han, W. Chen, and R. Doemer, "Designer-in-the-loop recoding of esl models using static parallel access conflict analysis," in *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2013.

[12] W. Chen, C.-W. Chang, X. Han, and R. Dömer., "Eliminating Race Conditions in System-Level Models by using Parallel Simulation Infrastructure," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, 2012.

[13] R. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," *Acta Informatica*, vol. 19, no. 1, pp. 57–84, 1983.

[14] W. Chen, R. Dömer, and X. Han, "Out-of-Order Parallel Simulation for ESL Design," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.