Eliminating Race Conditions in System-Level Models by using Parallel Simulation Infrastructure

Weiwei Chen, Che-Wei Chang, Xu Han, Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, USA
{*weiwei.chen,cheweic,hanx,doemer*}@*uci.edu*

Abstract—**For a top-down system design flow, a well-written specification model of an embedded system is crucial for its successful design and implementation. However, the task of writing a correct system-level model is difficult, as it involves, among other tasks, the insertion of parallelism. In this paper, we focus on ensuring model correctness under parallel execution. In particular, the model must be *free of race conditions* in all accesses to shared variables, so that a safe parallel implementation is possible. Eliminating race conditions is difficult because discrete event simulation often hides such flaws. In particular, the absence of simulation errors does not prove the correctness of the model. We propose to use advanced conflict analysis in the compiler, fast checking in a parallel simulator, and a novel race-condition diagnosis tool, that not only exposes all race conditions, but also locates where and when such problems occur. Our experiments have revealed a number of dangerous race conditions in existing embedded multi-media application models and enabled us to efficiently and safely eliminate these hazards.**

I. Introduction

At the starting point of the electronic system-level (ESL) design flow, a well-defined specification model of the intended embedded system is critical for rapid design space exploration and efficient synthesis and refinement towards a detailed implementation at lower abstraction levels. Typically, the initial specification model is written using system-level description languages (SLDLs), such as SystemC and SpecC. In contrast to the original application sources, which usually are specified as unstructured sequential C source code, a well-defined system model contains a clear structural hierarchy, separate computation and communication, and explicit parallelism.

In this paper, we focus on the correct specification of potential parallelism in the system model. In order to utilize parallel processing for low power and high performance in embedded systems, ESL models must contain explicit and efficient parallelism. Notably, parallelization is a particularly important but also very difficult task in system modeling.

Most reference code for embedded applications is sequentially specified. To parallelize the application, the designer must first identify suitable functions that can be efficiently parallelized, and then recode the model accordingly to expose the parallelism. Because identifying effective thread-level parallelism requires the designer's knowledge and understanding of the algorithm and therefore is a manual task, the model recoding is typically a tedious and error-prone process. Automating the coding, validation, and debugging is highly desirable.

In this paper, we address the problem of ensuring that the functionality of the model remains correct during parallel execution. In particular, the system model must be free of *race conditions* for all accesses to any shared variables, so that a safe parallel execution and implementation is possible.

Ensuring that there are no race conditions proves very difficult for the system designer, especially because the typically used discrete event simulation often does not reveal such mistakes in the model. Even if the simulation fails due to encountered race conditions, these are very hard to debug as the cause for an invalid output value may be hidden deep in the complex model.

We emphasize that the absence of errors during simulation does not imply the absence of any dangerous race conditions in the model. Even parallel simulation on multi-core hosts, such as [11], [12], [4], for which the likelihood is higher, that a race condition leads to an error, cannot guarantee that all situations are exposed.

To solve this race condition problem, we use our parallel simulation infrastructure [5]. Specifically, we propose a combination of (a) advanced static conflict analysis in the compiler, (b) table-based checking in a parallel simulator, and (c) a novel race-condition diagnosis tool. Together, these tools not only discover all race conditions that can potentially affect the concurrent execution, but also provide the designer with detailed source line information of where and when these problems occur.

II. Overview and Approach

In the context of validating a parallel system model, our proposed approach utilizes the compiler and simulator tools from a parallel simulation infrastructure to automatically detect and diagnose potential data hazards in the model. As a result, the system designer can quickly resolve the hazards due to race conditions and produce a safe parallel model.

A. Creating Parallel System Models

Exposing thread-level parallelism in sequential applications models requires three main steps:

1) *Identify the blocks to parallelize*: The first step is to understand the application and its algorithms. With the help of statistics from a profiler, the designer can then identify suitable blocks in the application with high

computational complexity for which parallelization is desirable and likely beneficial.

2) *Restructure and recode the model*: Through partitioning of functional blocks and encapsulating them into SLDL modules, the application is transformed into a system-level model with proper structure and hierarchy [2]. In particular, parallel execution is exposed explicitly.
With parallelism inserted into the model structure, affected variables may need to be recoded appropriately to ensure correct functionality of the model. For example, the variables involved in the restructuring may need to be duplicated, relocated into appropriate scope (localized), or wrapped in channels with explicit communication. Here, proper data dependency analysis is a critical component in resolving access conflicts due to parallelization. Performed manually, this is a tedious and error-prone task especially if the model is of modest or large size. The designer must locate affected variables, identify their all their read and write accesses, and make sure that no invalid accesses exist due to race conditions.

3) *Validate the model*: The correct functionality of the model is typically validated through simulation. However, regular simulators hide many potential access conflicts due to their sequential execution. Parallel simulators, such as [4], [12], execute on multi-core CPUs in parallel and can thus expose some access conflicts and race conditions. If these lead to invalid simulation results, this tells the designer that the model has a problem, but not where it is. Most often it is then very difficult to locate the cause and correct the problem.
Nevertheless, no existing simulation technique can prove the absence of race conditions. We will address this short-coming in this paper by an extension of a parallel simulation infrastructure.

B. Parallel Discrete Event Simulation (PDES)

System-level models written in SLDLs are typically validated by discrete event (DE) simulation which is driven by event notification and time advances. Traditional DE simulation, which is implemented by the reference simulators for both SystemC [9] and SpecC [7], uses a cooperative multithreading model where only one thread can run at any time.

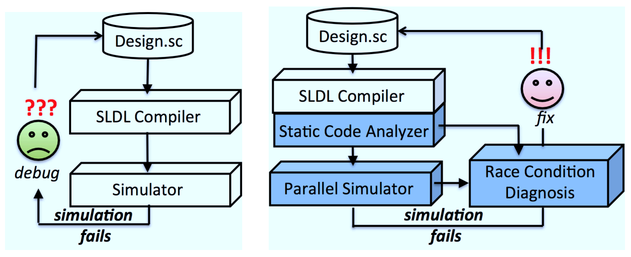
Synchronous PDES approaches, such as proposed in [11], [12], [4], allow multiple threads to run in parallel on multiple cores in today's host PCs. The SLDL simulators are extended to run threads in parallel in the same simulation cycle, i.e. same *delta* and *time*. However, synchronous PDES imposes a total order on simulation cycle advances, making them absolute barriers for thread execution. Available CPU cores remain idle while waiting for the threads mapped to other cores to reach the cycle barrier.

Out-of-order PDES [5] is an advanced simulation approach which breaks the global time and cycle barrier and issues multiple threads in parallel even if they are in different simulation cycles. It relies on static conflict analysis at compile time to generate information about hazards which is then used by the simulator for safe but fast scheduling decisions. Out-of-order PDES fully preserves simulation semantics and timing accuracy, albeit being aggressive in issuing threads in order to run as many as possible as early as possible.

We will reuse the advanced static analysis of out-of-order PDES in this paper for detecting and diagnosing race conditions.

C. Shared Variables and Race Conditions

Variables shared among parallel modules in the system model can cause data hazards, i.e. read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) conflicts. Thus, invalid parallel accesses to shared variables in the system model must be prevented.



(a) Manual Approach (b) Proposed Approach

Fig. 1. Validation and debugging flow for parallel system models.

As discussed earlier, designers traditionally design the specification model and handle shared variables manually. As shown in Figure 1(a), model validation is then performed by compiling and simulating the model using a traditional DE simulator. If the simulation fails, the designer needs to identify the problem, locate it in the code, and revise the model for another iteration. This debugging is typically a lengthy and error-prone process.

Moreover, even if the model is incorrect due to race conditions regarding shared variables, the simulation may actually succeed when using traditional sequential DE simulation. This might lead the designer to believe the model is correct, whereas in fact it is not. In other words, traditional simulation can lead to false validation of parallel models.

Discrete Event Simulation (DES)	Synchronous Parallel Discrete Event Simulation (SPDES)	Out-of-order Parallel Discrete Event Simulation (Out-of-order PDES)	Race Condition Diagnosis for Parallel Models
Regular SLDL compiler	Regular SLDL compiler Multithreading protection instrumentation	Regular SLDL compiler Multithreading protection instrumentation Static Conflict Analysis	Regular SLDL compiler Multithreading protection instrumentation Static Conflict Analysis
Sequential simulator	Multi-core Parallel simulator	Multi-core Out-of-order Parallel Simulator	Multi-core Parallel Simulator

Fig. 2. Reusing essential tools from a parallel simulation infrastructure to diagnose race conditions in parallel models.

As shown in Figure 2, traditional DE simulation uses a regular SLDL compiler to generate the executable model and then uses sequential simulation for its validation. In comparison, synchronous PDES also uses the regular SLDL compiler, but instruments the design with any needed synchronization protection for true multi-threaded execution. An extended simulator is then used for multi-core parallel simulation.

The advanced out-of-order PDES approach, in contrast, uses the PDES compiler extended by an additional static code analyzer to generate potential conflict information. The corresponding scheduler in the simulator is also extended to utilize the compiled conflict information for issuing threads early and out of the order for faster simulation.

This infrastructure for advanced PDES motivates our idea for dynamic race condition diagnosis. The compiler for out-of-order PDES can analyze the design model statically to generate the needed information about potential data conflicts. Also, the synchronous PDES simulator allows threads in the same simulation cycle to run in parallel. Combining the two, we can therefore pass the conflict information generated by the compiler to the scheduler for dynamic race condition diagnosis among the parallel executing threads. As illustrated in Figure 1(b), the system designer can thus detect and obtain a diagnosis about parallel accesses to shared variables automatically and fix the problem quickly.

III. Automatic Race Condition Diagnosis

Figure 3 shows the detailed tool flow for our proposed race condition diagnosis in parallel design models. Here, we are using a SpecC-based compiler and simulator framework.

The flow starts with an initial design model, i.e. *Design.sc*, as the input to the SLDL compiler. The compiler parses and checks the syntax of the model, builds an internal representation (extended abstract syntax tree) of the model, and then generates a C++ model (*Design.cc* and *Design.h*) that is compiled by the C++ compiler, e.g. *g++*, to produce the executable file for simulation.

In our proposed tool flow, we add a *Static Code Analyzer* which analyzes the internal design representation for potentially conflicting accesses to shared variables during the parallel execution. A *Segment Graph* representing the parallel execution flow in the model and a *Variable Access List* are computed in this step. Using the segment graph and the variable access lists for the segments, the static analyzer constructs then a *Data Conflict Table* that lists any potential access conflicts in the design. This data conflict table is then passed to the simulator via instrumentation into the model.

The model is then validated by a *Parallel Simulator*, a synchronous PDES simulator extended with dynamic conflict checking. Whenever there are two threads running at the same simulation and delta time, the simulator checks the data conflict table for any conflicts between the segments the threads are running in. If there is a conflict, the simulator has detected a race condition and reports this in a *Dynamic Conflict Trace* file (*Design.w*). Note that this conflict checking is based on fast table look-ups which introduces very little simulation overhead.

After the simulation completes, the *Race Condition Diagnosis* tool processes the generated *Variable Access List* and *Dynamic Conflict Trace* files and displays the detected race conditions to the designer. As shown in Figure 3, critical parallel accesses to shared variables are listed with detailed information, including time stamp, access type, variable name and type, and line number and source file location where the variable is defined and where the access occurred. Since there may be many reports for the same race condition due to iterations in the execution, our tool combines these cases and lists the first time stamp and the number of repetitions.

Given this detailed information, the designer can easily find the cause of problems due to race conditions and resolve them.

IV. Race Condition Elimination Infrastructure

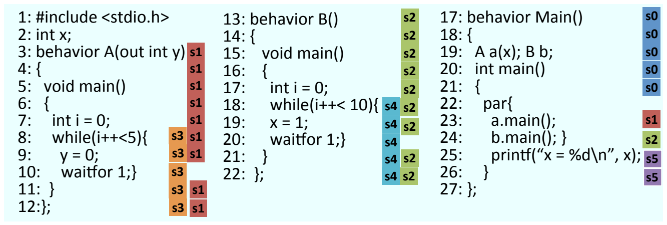
As outlined above, we use (a) advanced static code analysis to generate potential conflict information for a model, and then (b) parallel simulation for dynamic conflict detection.

A. Static Code Analysis

Careful source code analysis at compile time is the key to identify potential access conflicts to shared variables.

During simulation, threads switch back and forth between the states of **RUNNING** and **WAITING**. Each time, threads execute different *segments* of their code. Access hazards exist when two segments contain accesses to the same variables. Except when two segments contain only read accesses (RAR), any write access creates a potential conflict (RAW, WAR, or WAW). These potential conflicts are called *race conditions* when they occur at the same simulation time, i.e. in the same simulation cycle.

Due to the (intended) non-deterministic execution in the simulator, race conditions may or may not lead to invalid values for the affected variables. Since this is often dangerous, race conditions must be eliminated (or otherwise handled) for parallel design models to be safe.

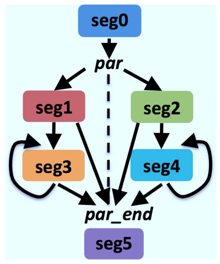


```
#include <stdio.h>
int x;
behavior A(out int y)    s1
{                        s1
  void main()            s1
  {                      s1
    int i = 0;           s1
    while(i++<5){     s3 s1
      y = 0;          s3 s1
      waitfor 1;}     s3
  }                   s3 s1
};                    s3 s1
```

```
behavior B()                  s2
{                             s2
  void main()                 s2
  {                           s2
    int i = 0;                s2
    while(i++< 10){        s4 s2
      x = 1;               s4 s2
      waitfor 1;}          s4
  }                        s4 s2
};                         s4 s2
```

```
behavior Main()               s0
{                             s0
  A a(x); B b;                s0
  int main()                  s0
  {                           s0
    par{
      a.main();               s1
      b.main(); }             s2
      printf("x = %d\n", x);  s5
    }                         s5
};
```

(a) SpecC source code



(b) Segment graph

seg	Access lists
0	
1	x(W)
2	x(W)
3	x(W)
4	x(W)
5	x(R)

(c) Access lists

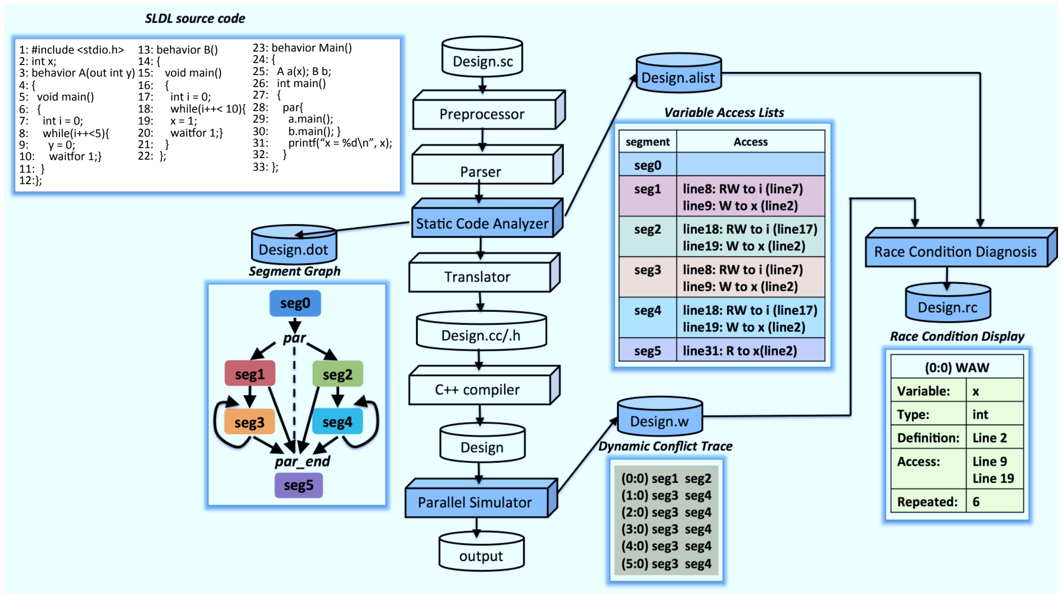
seg	0	1	2	3	4	5
0	F	F	F	F	F	F
1	F	T	T	T	T	T
2	F	T	T	T	T	T
3	F	T	T	T	T	T
4	F	T	T	T	T	T
5	F	T	T	T	T	F

(d) Data conflict table

Fig. 4. A parallel design example with a simple race condition.

For our proposed race condition analysis, we formally define the following terms:

- **Segment** seg_i: statements executed by a thread between two scheduling steps.
- **Segment Boundary** b_i: SLDL primitives which call the scheduler, e.g. *wait*, *wait-for-time*, *par*.



```
1: #include <stdio.h>
2: int x;
3: behavior A(out int y)
4: {
5:   void main()
6:   {
7:     int i = 0;
8:     while(i++<5){
9:       y = 0;
10:     waitfor 1;}
11:  }
12:};
13: behavior B()
14: {
15:   void main()
16:   {
17:     int i = 0;
18:     while(i++< 10){
19:     x = 1;
20:     waitfor 1;}
21:   }
22:  };
23: behavior Main()
24: {
25:   A a(x); B b;
26:   int main()
27:   {
28:     par{
29:       a.main();
30:       b.main(); }
31:       printf("x = %d\n", x);
32:     }
33: };
```

segment	Access
seg0	
seg1	line8: RW to i (line7) line9: W to x (line2)
seg2	line18: RW to i (line17) line19: W to x (line2)
seg3	line8: RW to i (line7) line9: W to x (line2)
seg4	line18: RW to i (line17) line19: W to x (line2)
seg5	line31: R to x(line2)

```
(0:0) seg1 seg2
(1:0) seg3 seg4
(2:0) seg3 seg4
(3:0) seg3 seg4
(4:0) seg3 seg4
(5:0) seg3 seg4
```

(0:0) WAW	
Variable:	x
Type:	int
Definition:	Line 2
Access:	Line 9 Line 19
Repeated:	6

Fig. 3. Tool flow for automatic race condition diagnosis among shared variables in parallel system models.

Here, segment boundaries b_i start segments seg_i. Thus, a directed graph is formed by the segments, as follows:

- **Segment Graph (SG)**: **SG=(V, E)**, where **V** = $\{v \mid v_i$ is segment seg_i started by segment boundary $b_i\}$, **E**=$\{e_{ij} \mid e_{ij}$ exists if seg_j is reached after $seg_i\}$.

From the control flow graph of a design model, we can derive the corresponding segment graph [5].

For example, Figure 4(a) and (b) show a simple system model written in SpecC SLDL and its corresponding segment graph. Starting from the initial segment seg_0, two separate segments seg_1 and seg_2 represent the two parallel threads after the *par* statement in line 22. New segments are created after each segment boundary, such as *waitfor 1* (lines 10 and 20). The segments are connected following the control flow of the model. For instance, seg_3 is followed by itself due to the *while* loop in lines 8-10.

Given the segment graph, we next need to analyze the segments for statements with potentially conflicting variable assignments. We first build a variable access list for each segment, and then compile a conflict table that lists the potential conflicts between the N segments in the model:

- **Variable Access List:** $segAL_i$ 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*).
- **Data Conflict Table (CT[N,N])**:

$$CT[i,j] = \begin{cases} true & \text{if } seg_i \text{ has data conflict with } seg_j \\ false & \text{otherwise} \end{cases}$$

Note that $CT[N,N]$ is symmetric and can be built simply by comparing pairs of the variable access lists.

B. Dynamic Race Condition Checking

We detect race conditions dynamically at runtime when the simulator schedules the execution of the threads. Figure 5 shows the scheduling algorithm for synchronous PDES extended with the needed checking. The simulator performs the regular discrete event scheduling on the right side of Figure 5 in order to deliver events and increment simulation time, following the usual *delta* and *time* cycles. On the left, the algorithm issues suitable threads to run in parallel as long as CPU cores are available.

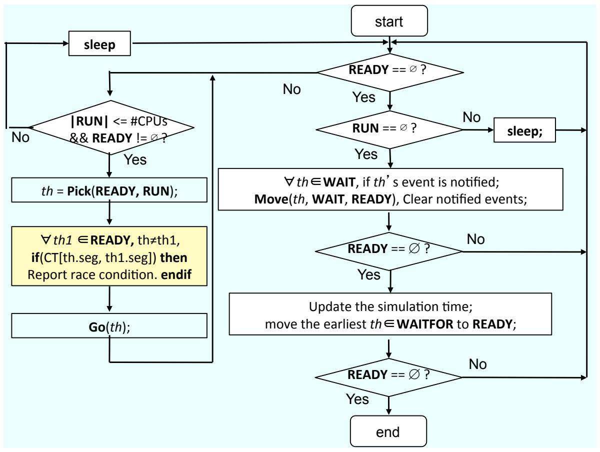

start
sleep
READY == ∅ ?
No
Yes
|RUN| <= #CPUs && READY != ∅ ?
No
Yes
th = Pick(READY, RUN);
∀th1 ∈READY, th≠th1, if(CT[th.seg, th1.seg]) then Report race condition. endif
Go(th);
RUN == ∅ ?
No
sleep;
Yes
∀th∈WAIT, if th' s event is notified; Move(th, WAIT, READY), Clear notified events;
READY == ∅ ?
No
Yes
Update the simulation time; move the earliest th∈WAITFOR to READY;
READY == ∅ ?
No
Yes
end

Fig. 5. Parallel simulation algorithm with dynamic race condition checks.

Whenever it issues a thread for execution, the scheduler consults the data conflict table provided by the compiler in order to report detected race conditions. As shown on the left side of Figure 5, the scheduler checks every thread for conflicts with the other threads that are **READY** to run. Again, we emphasize that these checks are simple table look-ups so that the overhead of race condition detection is minimal.

While we are using our parallel simulator for speed reasons here, we should note that the same detection approach can also be integrated in a traditional sequential DE simulator.

V. Experiments and Results

We have implemented the proposed tool set for race condition elimination in parallel models in a SpecC-based system-level design environment [6]. In this section, we report the results of using our approach on several embedded application examples. We describe how the tool set helped us to detect and diagnose a number of race conditions. Several reports on invalid parallel accesses to shared variables turned out to be the actual cause of simulation errors which we then could fix. Other reports could be ruled out for various reasons described below. All experiments have been performed on the same host PC with a 4-core CPU (Intel$^{(R)}$ Core$^{(TM)}$2 Quad) at 3.0 GHz.

A. Case study: A Parallel H.264 Video Decoder

During the development of our parallel H.264 video decoder model [10], we used the regular sequential SLDL simulator to validate the functionality. This showed 100% correct outputs. However, when we later used parallel simulation, the model *sometimes* ran through a few frames and then terminated with various assertion failures, or even terminated immediately with a segmentation fault. This was frustrating because such non-deterministic errors are very hard to debug.

We used the new race condition diagnosis tool to check the model, resulting in reports on 40 different shared variables. The model designer went through the variable list one by one to eliminate problems caused by race conditions. As reported in Table I, half of the reported variables were defined in channels and therefore protected from access conflicts by the SpecC execution semantics (implicit locks). No change was necessary for these.

Another 15 variables were determined as storing values that are constant to each frame and thus can be shared safely when the decoder processes the frames in parallel. One report was about a complex structure that is actually accessed by parallel tasks only to non-overlapping members. Another variable was a global counter used purely for debugging. Again, no change to the model was necessary for these cases.

However, the remaining three of the reported variables actually caused the simulation problems. The model designer resolved the race conditions for them by relocating the variables from global to class scope. This way, each parallel unit has its own copy of these variables. As a result, the dangerous race conditions were eliminated and the model now simulates correctly also in the parallel simulator.

B. Case study: A Parallel H.264 Video Encoder

As a second large application example, we have converted the reference C source code of a H.264 encoder into a SpecC model [3]. To allow parallel video encoding, we restructured the model for parallel motion estimation distortion calculation. When using the regular sequential simulator, the simulation result of the parallelized model matched the reference implementation. However, after we switched the simulator with a newly developed parallel simulator, the encoding result became inconsistent. That is, the encoding process finished properly, but the encoded video stream differed from time to time.

At the beginning of debugging, we had no idea about the cause of the encoding errors. Moreover, we were not even sure whether the problem was caused by the model itself or by our new parallel simulator. Literally thousands of lines of code, in both the model and the simulator, were in question.

At this point, the advanced static code analysis used in our new out-of-order PDES simulator [5] sparked the idea of using it to attack such debugging problems. After some preparation, a first version of the race condition diagnosis tool described in this paper was implemented.

When we used this tool to analyze the H.264 video encoder model, we indeed found a total of 68 variables accessed in race conditions by the parallel motion estimation blocks. Specifically, the encoding malfunction was caused by read/write accesses to 14 global variables which stored the intermediate computation results in the parallelized behaviors. After localizing those global variables to local variables on the stack of the executing thread, the encoding result was correct, matching the output of the reference code.

For the remaining reported variables, a listed in Table I, there was no immediate need to recode them. For example, variables which remain constant during the encoding process in our model and for our fixed parameters (for example *height_pad*), we decided to leave these unchanged for now (future work).

C. Additional Embedded Applications

For this paper, we have used the proposed tool set also to double-check embedded application models that had been developed earlier in-house based on standard reference code.

The first application is a fixed-point MP3 decoder model for two parallel stereo channels [1]. As shown in Table I, our diagnosis tool reports 7 shared variables that are subject to race conditions, out of a total of 82 conflict trace entries. We have looked at these variables and found that they are all member variables of channel instances which are protected by implicit locks for mutual exclusive accesses to channel resources. Thus, we successfully confirmed that this model is free of race conditions.

The second application is another parallel MP3 decoder model based on floating-point operations [1]. Our diagnosis tool lists 9 shared variables out of 75 trace file entries. Eight of those variables are channel variables which are free from data hazards. The remaining variable, namely *hybrid_blc*, is an array of 2 elements. Each element is used separately by the two stereo channels, so there is no real conflict. We can resolve the race condition report by splitting this array into two instances for the two channels. Thus, this parallel model is also free of data hazards.

The third embedded application is a GSM Vocoder model whose functional specification is defined by the European Telecommunication Standards Institute (ETSI) [8]. Only two variables are reported by our diagnosis tool for race condition risks. The first one, *Overflow*, is a Boolean flag used in primitive arithmetic operations. It can be resolved by replacing it with local variables on the stack of the calling thread. The second one, *old_A*, is an array which stores the previous results

TABLE I
EXPERIMENTAL RESULTS ON AUTOMATIC RACE CONDITION DIAGNOSIS FOR EMBEDDED MULTI-MEDIA APPLICATIONS.

Application	Lines of Code	# Variables / # Trace Entries	Resolved Race Conditions	Unresolved Race Conditions	Tool Execution Time [sec]				
					Compiler, diagnosis off/on		Simulator, diagnosis off/on		Diagnosis
H.264 video decoder	40k	40 / 1201	3 resolved, localized to class scope 15 store values constant to each frame, safe to share 1 structure accessed without overlap, safe 1 debugging value, temporarily used only 20 in channels, safely protected	0	12.51	14.05	18.51	19.56	1.04
H.264 video encoder	70k	68 / 712911	14 resolved, localized to stack variables 15 constant with current parameters, OK 1 identical in all parallel blocks, OK 1 array variable resolved by splitting the array 37 in channels, safely protected	0	29.52	29.72	110.58	111.72	13.97
MP3 decoder, fixed point	7k	7 / 82	7 in channels, safely protected	0	1.14	1.19	4.34	4.44	0.08
MP3 decoder, floating point	14k	13 / 75	12 in channels, safely protected 1 array variable resolved by splitting the array	0	3.63	3.89	13.82	13.87	0.34
GSM vocoder	16k	2 / 253	1 resolved, duplicated for parallel modules 1 resolved, localized to stack variable	0	3.90	4.00	1.46	1.50	0.07
JPEG encoder	2.5k	3 / 66	9 in channels, safely protected	0	4.01	4.09	1.54	1.56	0.02

for an unstable filter. This variable is incorrectly shared by two parallel modules. We can resolve this situation by duplicating the variable so that each parallel instance has its own copy. We should note that these two bugs have been undetected for more than a decade in this in-house example.

The last application is a JPEG encoder for color images. There are three variables reported as potential race conditions out of 253 entries in the trace log. Since all three are members of channel instances which are implicitly protected by locks, it is safe to have them in the parallel model.

In summary, Table I lists all our experimental results, including the size of the models and the performance of the tools. While our application examples are fairly large design models consisting of several thousand lines of code, the overhead of race condition diagnosis is negligible for both compilation and simulation. Also, the diagnosis tool itself runs efficiently in less than a few seconds.

VI. Conclusions and Future Work

Writing well-defined and correct system-level design models with explicit parallelism is difficult. Race conditions due to parallel accesses to shared variables pose an extra challenge as these are often not exposed during simulation.

In this paper, we proposed an automatic diagnosis approach that enables the designer to ensure that a developed model is *free of race conditions*. The infrastructure of our proposed tool flow includes a compiler with advanced conflict analysis, a parallel simulator with fast dynamic conflict checking, and a novel race-condition diagnosis tool. This flow provides the designer with detailed race condition information that is helpful to fix the model efficiently when needed.

The proposed approach has allowed us to reveal a number of risky race conditions in existing embedded multi-media application models and enabled us to efficiently and safely eliminate these hazards. Our experimental results also show very little overhead for race condition diagnosis during compilation and simulation.

In future work, we plan to provide better analysis support for variables of array and pointer types, and to develop recoding functions to automate the steps in resolving race conditions.

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] P. Chandraiah and R. Dömer. Specification and design of an MP3 audio decoder. Technical Report CECS-TR-05-04, Center for Embedded Computer Systems, University of California, Irvine, May 2005.

[2] P. Chandraiah and R. Dömer. Computer-aided recoding to create structured and analyzable system models. *ACM Trans. Embed. Comput. Syst.*, 11S(1):23:1–23:27, June 2012.

[3] C.-W. Chang and R. Dömer. System Level Modeling of a H.264 Video Encoder. Technical Report CECS-TR-11-04, Center for Embedded Computer Systems, University of California, Irvine, 2011.

[4] 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*, 28(3):20–31, May/June 2011.

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

[6] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(647953):13 pages, 2008.

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

[8] A. Gerstlauer, S. Zhao, D. D. Gajski, and A. M. Horak. Design of a GSM vocoder using SpecC methodology. Technical Report ICS-TR-99-11, Information and Computer Science, University of California, Irvine, March 1999.

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

[10] X. Han, W. Chen, and R. Dömer. A Parallel Transaction-Level Model of H.264 Video Decoder. Technical Report CECS-TR-11-03, Center for Embedded Computer Systems, University of California, Irvine, 2011.

[11] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, 2009.

[12] 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*, pages 241–246, 2010.