An Optimizing Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolation

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

***Abstract*—Electronic system-level (ESL) design relies on fast discrete event (DE) simulation for the validation of design models written in system-level description languages (SLDLs). An advanced technique to speedup ESL validation is out-of-order parallel DE simulation which allows multiple threads to run early and in parallel on multi-core hosts. To avoid data hazards and ensure timing accuracy, this technique requires the compiler to statically analyze the design model for potential data access conflicts. In this paper, we propose a compiler optimization that improves the data conflict analysis by exploiting instance isolation. The reduction in the number of conflicts increases the available parallelism and results in significantly reduced simulation time. Our experimental results show up to 90% gain in simulation speed for less than 6% increase in compilation time.**

I. Introduction

Modern System-Level Description Languages (SLDLs), such as SystemC [1] and SpecC [2], support the abstract modeling of systems with both hardware and software for true ESL design. The validation of system models described in SLDLs is typically based on discrete event (DE) simulation. Traditional DE simulation expresses the parallelism in the design model as concurrent user-level threads within a single process. The multi-threading model used is cooperative (i.e. non-preemptive) [3], which simplifies the communication between the threads, but is an impediment against the utilization of parallel computation resources available in todays' multi-core PCs. Recent works [4], [5], [6] propose to use OS kernel threads with additional synchronization in order to issue multiple threads in parallel at each scheduling step so that the available resources in a multi-core host CPU can be utilized. However, the number of parallel threads that can actually run at each scheduling step is often very limited. The inner loops for delta-cycles and time update in the DE simulation algorithm severely restrict the usable parallelism in the model.

Out-of-order (OoO) parallel DE simulation [7] is an advanced approach that improves the parallel simulation by agressively attacking the temporal barriers in the traditional DE simulation algorithm. Here, the compiler performs static code and data analysis in the design model and generates "look-ahead" information, i.e. potential data conflicts among threads and future occurrences of scheduling, and passes this information to the simulator. The simulator, in turn, can then easily decide at runtime whether or not it is safe to issue a set of threads in parallel. Moreover, by using thread-local simulation times, the simulator can even run threads with different timestamps in parallel while ensuring SLDL-compliant behavior with full timing accuracy.

In this work, we observe that it is critical to take the instance path for certain shared variables into consideration in order to decide whether or not multiple threads can be allowed to run in parallel. The access conflict information generated at compile time is sometimes overly conservative and may lead to false conflicts when, for example, modules that are instantiated multiple times in the design model, share segments of the same source code. This, in turn, prevents the simulator from utilizing higher parallelism as it needs to guarantee the correctness of the simulation results. We also discover that certain modifications in the design's SLDL description can help the compiler to reduce the amount of detected conflicts. Following these observations, we propose a novel compiler optimization that automatically performs such beneficial model modifications at compile time with very little cost in space and time. While the compile time increases minimally, we gain a significant reduction in detected data access conflicts. The more precise "look-ahead" information passed to the simulator then results in a much more efficient out-of-order parallel simulation.

After a brief review of out-of-order parallel DE simulation in Section I-B, we explore the effect of instance isolation, a new model improvement technique, on reducing the detected conflicts by the compiler in Section II. Our optimized compiler is then presented in Section III, followed by experimental results in Section IV.

A. Related Work

Parallel Discrete Event Simulation (PDES) has been well explored in [8], [9], [10]. There are two major synchronization paradigms for PDES, namely conservative and optimistic [9]. *Conservative* PDES ensures in-order event execution. In contrast, the *optimistic* paradigm assumes that every event is safe when executed and rolls back when this assumption proves wrong. The temporal barriers in the SLDL model often prevent conservative PDES from achieving real parallel simulation, while rollbacks in the optimistic PDES are expensive in implementation and execution.

DE simulation for SLDLs is driven by events and simulation time advances. In order to interpret the "zero-delay" semantics of SLDLs, the notion of *delta-cycles* is introduced to impose a partial order on the events that happen at the same simulation time [1].

978-1-4673-0772-7/12/$31.00 ©2012 IEEE

PDES with delta-cycle notion has been also been explored. For example, [11], [4] apply PDES to SystemC targeting symmetric multi-processing (SMP) architectures by using the conservative synchronization paradigm. Still, the global simulation time is shared by every thread in the design model which proves as an obstacle in obtaining efficient parallelism.

Compared to the existing approaches of SLDL PDES [6], out-of-order PDES [7] is more aggressive and issues more threads in parallel at each scheduling step by allowing execution *out-of-order*. However, it is still conservative in the sense that it preserves a partial order of events and maintains accurate simulation time. While some events may be executed out of the order, no rollback corrections are needed. We will briefly review this approach, which can efficiently execute any model written in SLDL on any SMP host, in the following section.

B. Out-of-order Parallel DE Simulation

Regular PDES imposes a total order on event handling and time advances which severely limits the potential for parallel execution of threads. In contrast, out-of-order PDES imposes only a *partial order* on time and events so that threads without potential conflicts can run in parallel [7]. Table I lists the major differences between regular and out-of-order PDES.

Static model analysis at compile time is the key to allowing the scheduler at run time to make quick decisions without risks about potential conflicts when issuing threads in parallel. The *risks* here are data hazards, i.e. read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) conflicts for shared variables and events.

At run time, threads switch back and forth between the states of RUNNING and WAITING. When RUNNING, they execute specific *segments* of their code. The following definitions are essential for the thread conflict analysis:

- **Segment** seg_i: portion of code executed by a thread between two scheduling steps.
- **Segment Boundary** v_i: SLDL statements which call the scheduler, i.e. *wait*, *waitfor*, *par*, etc.

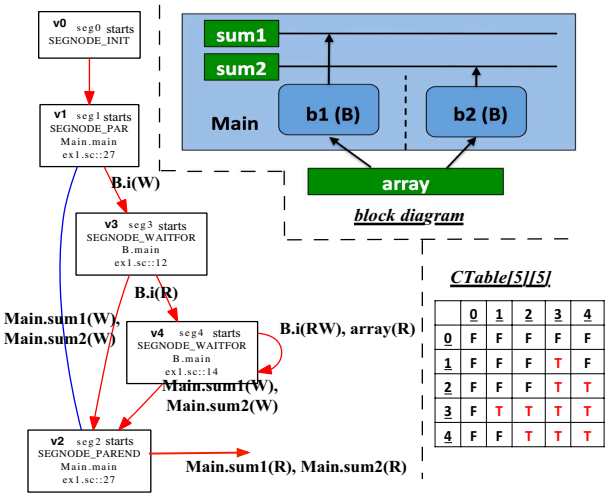
Note that segments seg_i and segment boundaries v_i form a directed graph where seg_i is the segment following the boundary v_i. Every node v_i is connected by segments to other nodes. We formally define:

- **Segment Graph (SG)**: SG=(V, E), where V = $\{v \mid v$ is a segment boundary$\}$, E=$\{e_{ij} \mid e_{ij}$ is the code portion between v_i and v_j, where v_j is reached after $v_i\}$.
- **Segment Conflict Table** $CTable[N][N]$ where N is the total number of segments: $CTable[i][j]$ = *true*, iff there is a data conflict between the segments seg_i and seg_j; otherwise, $CTable[i][j]$ = *false*.

Fig. 1(a) shows a simple example written in SpecC SLDL. The design has two parallel instances `b1` and `b2` of type `B`. Both instances `b1` and `b2` compute the sum of a range of elements stored in a global array `array`. The range is provided at the input ports `begin` and `end`. The result is passed back to the parent via the output port `sum`. The top behavior `Main` prints the results `sum1` of `b1` and `sum2` of `b2` to the screen.

```
1 #include <stdio.h>
2 int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3 behavior B(in int begin,// port variable
4            in int end,  // port variable
5            out int sum) // port variable
6 {
7   int i;                // member variable
8   void main(){
9     int tmp;            // stack variable
10    tmp = 0;            // tmp(W)
11    i = begin;          // i(W), begin(R)
12    waitfor 1;          // segment boundary (waitfor)
13    while(i <= end){    // i(R), end(R)
14      waitfor 2;        // segment boundary (waitfor)
15      tmp += array[i];  // array(R), tmp(RW)
16      i ++;             // i(W)
17    }
18    sum = tmp; }
19 };
20
21 behavior Main()
22 {
23   int sum1, sum2;      // member variables
24   B b1(0, 4, sum1);    // behavior instantiation
25   B b2(5, 9, sum2);    // behavior instantiation
26   int main(){
27     par{               // segment boundary (par)
28       b1.main();
29       b2.main(); }     // segment boundary (par_end)
30     printf("summation 1 is :%d \n", sum1); // sum1(R)
31     printf("summation 2 is :%d \n", sum2); // sum2(R) }
32 };
```

(a) Example source code in SpecC.



	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	T	F
2	F	F	F	T	T
3	F	T	T	T	T
4	F	F	T	T	T

(b) Example block diagram and segment graph (SG).

Fig. 1. A simple design example with segment graph and conflict table.

In the corresponding SG shown in Fig. 1(b), there are five segment nodes connected by red arrows indicating the possible control flow between the nodes. From the initial node, the control flow reaches the `par` statement (line 27) which is represented by the two nodes *v1* and *v2* connected by a blue line. Here, the simulator will fork two parallel threads for `b1` and `b2`. Since both are of the same type, both will reach `waitfor 1` (line 12) via the same arrow *v1*→*v3*. From there, the control flow reaches either `waitfor 2` (line 14) via *v3*→*v4*, or will skip the `while` loop (line 13) and complete the thread execution at the end of the `par` statement via *v3*→*v2*. From within the `while` loop, control either loops around *v4*→*v4*, or ends the loop and the thread via *v4*→*v2*. Finally, after *v2* the result is printed. [1]

For each segment, the variables accessed are analyzed. Local variables (e.g. `tmp`) and port variables driven by constant

[1] An algorithm to construct the segment graph (SG) from the design's control flow graph (CFG) is given in [7].

TABLE I
COMPARISON OF REGULAR VS. OUT-OF-ORDER PARALLEL DISCRETE EVENT SIMULATION(PDES) FOR SLDLS.

	Regular PDES, e.g. [11], [4], [6]	Out-of-Order PDES , e.g. [7]
Simulation Time	One global time tuple (t, δ) shared by every thread	Local time for each thread th as tuple (t_{th}, δ_{th}). A total order of time is defined with the following relations: **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 $t_1 = t_2$, $\delta_1 > \delta_2$.
Event Description	Events are identified by their ids, i.e, event (id).	A timestamp is added to identify every event, i.e. event (id, t, δ).
Simulation Thread Sets	**READY**, **RUN**, **WAIT**, **WAITFOR**, **JOINING**, **COMPLETE**	Threads are organized as subsets with the same timestamp (t_{th}, δ_{th}). Thread sets are the union of these subsets, i.e, **READY** = $\cup$**READY**$_{t,\delta}$, **RUN** = $\cup$**RUN**$_{t,\delta}$, **WAIT** = $\cup$**WAIT**$_{t,\delta}$, **WAITFOR** = $\cup$**WAITFOR**$_{t,\delta}$ $(\delta = 0)$, **JOINING** = $\cup$**JOINING**$_{t,\delta}$, **COMPLETE** = $\cup$**COMPLETE**$_{t,\delta}$, where the subsets are ordered in increasing order of time (t, δ).
Run Time Scheduling	Event delivery in-order in delta-cycle loop. Time advance in-order in outer loop. Threads at same time run in parallel. Limited parallelism, inefficient SMP utilization.	Event delivery out-of-order if no conflicts exist. Time advance out-of-order if no conflicts exist. Threads at same time or with no conflicts run in parallel. ***More parallelism, efficient SMP utilization.***
Compile Time Analysis	No conflict analysis needed.	Static conflict analysis derives Segment Graph (SG) from CFG, analyzes variable and event accesses, passes conflict table to scheduler. ***Compile time increases.***

values (e.g. `begin` and `end`) cannot cause any hazards and are therefore ignored. However, global variables (e.g. `array`), member variables (e.g. `B.i`), and connector variables (e.g. `Main.sum1` and `Main.sum1`) can create conflicts. Fig. 1(b) shows the conflicts for the example annotated at the arrows and listed in the conflict table.

II. MOTIVATION

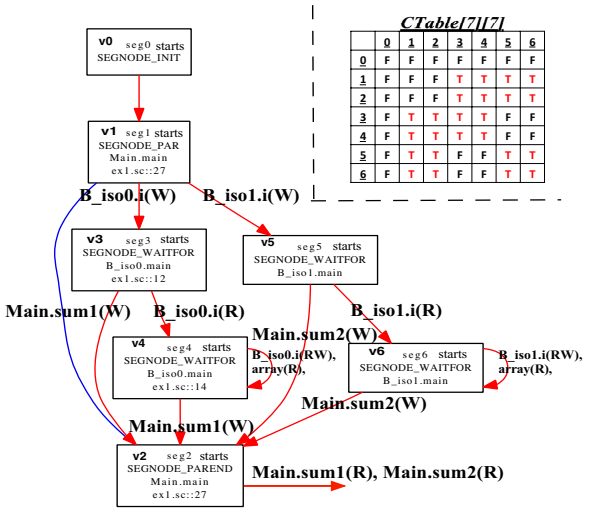
To be safe, the static analysis sometimes is overly conservative and generates *false conflicts* which prevent the out-of-order scheduler from issuing threads in parallel that do not pose any real hazard. For example, in Fig. 1 both *seg3* and *seg4* contain write (W) accesses to variables `Main.sum1` and `Main.sum2`. Consequently, the analysis reports a conflict between *seg3* and *seg4*. In turn, the thread th_{b2} cannot execute *seg4* in parallel with thread th_{b1} in *seg3* (or vice versa). In reality, however, the execution is safe when th_{b1} and th_{b2} are in *seg3* and *seg4* at the same time since instance `b1` will only access `Main.sum1` and `b2` only modifies `Main.sum2`. Thus, the conflict does not really exist.

Looking closer at this *false conflict*, we observe that the compiler cannot tell that the access of `Main.sum1` in *seg3* and *seg4* will only happen in thread th_{b1} but not in th_{b2}, because both `b1` and `b2` share the same definition (and same segments/code). Following this, we can resolve the false conflict if `b1` and `b2` have separate definitions, such as `B_iso0` and `B_iso1` in Fig. 2(a). Here, two different segments, *seg3* and *seg5*, start from `waitfor 1` in instances `B_iso0` and `B_iso1`, respectively. Now *seg3* only writes `Main.sum1` and *seg5* only writes `Main.sum2`. Consequently, the scheduler detects that it is safe for th_{b1} to execute *seg3* in parallel to th_{b2} in *seg5*. The same argument holds for *seg4* and *seg6*, as indicated in the extended *CTable* in Fig. 2(b).

In general, the fewer conflicts are detected by the analysis, the more threads can be issued in parallel by the out-of-order scheduler, and the higher the simulation speed will be. On the other hand, if the code analyzer reports conflict between all the segments, the out-of-order simulator will downgrade to a regular in-order simulator.

```
 3 behavior B_iso0(...)     behavior B_iso1( )
 4 {                        {
     ...                    // same definition as B
19 };                       };
20
21 behavior Main()
22 {
23   int sum1, sum2;
24   B_iso0 b1(0, 4, sum1);
25   B_iso1 b2(5, 9, sum2);
26   int main(){
27    par{
28     b1.main();
29     b2.main(); }
30   ...}
32 };
```

(a) Isolated instances in SpecC.



(b) Segment graph (SG) and conflict table with isolated instances.

Fig. 2. Simple design example with isolated instances.

Definition: We introduce the term **Instance Isolation** to describe the source code modification demonstrated above. *Isolating* an instance means to create a unique copy of the instance definition (behavior or channel) so that the instance has its own statements (and segments).

We will now show that instance isolation has a major impact on out-of-order PDES. Using the example of a H.264 video decoder[2], Table II shows the impact of instance isolation in

[2]See the experimental section for details on our H.264 model.

TABLE II
EXPERIMENTAL RESULTS FOR H.264 DECODER MODELS WITH DIFFERENT DEGREE OF ISOLATION.

Model	#bhvr	#chnl	formatted lines of code	Regular PDES		Out-of-Order PDES		#seg	#conflicts	#total issues	#OoO issues
				cmpl [sec]	sim [sec]	cmpl [sec] (speedup)	sim [sec] (speedup)				
iso0	54	11	55258	11.86	99.42	18.76 (-36.8%)	96.97 **(+2.5%)**	38	322 (22.3%)	927583	152816 (16.5%)
iso1	54	13	55330	11.93	101.01	17.57 (-32.1%)	96.49 **(+4.7%)**	41	336 (20.0%)	933222	146711 (15.7%)
iso2	58	16	55558	12.07	99.25	17.74 (-32.0%)	83.20 **(+19.3%)**	48	388 (16.8%)	913225	147541 (16.2%)
iso3	62	19	55786	12.23	99.68	17.87 (-31.6%)	72.72 **(+37.1%)**	55	440 (14.6%)	920001	166065 (18.1%)
iso4	66	24	56160	12.46	100.95	18.13 (-31.3%)	60.33 **(+67.3%)**	67	512 (11.4%)	923017	179581 (19.5%)

terms of model and code size, as well as compilation and simulation time. We have manually created 5 design models with an increasing number of isolated instances. *iso0* is the initial model without instance isolation, *iso1*, *iso2* and *iso3* are partially isolated, and *iso4* is a fully isolated model where each instance has its own definition. As expected and shown in Table II, the more instances we isolate, the more behaviors (*#bhvr*) and channels (*#chnl*) exist in the model. The number of lines of code increases as well, as does the compile time for out-of-order PDES due to the larger segment graph.

On the other hand, the out-of-order simulation gains significant speedup (up to 67.5%) due to the decreasing ratio of detected conflicts (*#conflicts*) among the increasing number of segments (*#seg*). This tendency is also clearly visible in the number of threads issued in parallel by the out-of-order PDES (*#OoO issues*). We conclude that instance isolation can significantly improve the run-time efficiency of out-of-order PDES. Next, we will automate this technique and address the overhead of larger designs and longer compilation time at the same time.

III. OPTIMIZED STATIC CODE ANALYSIS FOR OUT-OF-ORDER PARALLEL DE SIMULATION

We propose an optimized algorithm for the static code analysis needed by out-of-order PDES which automatically isolates the instances in the design "on-the-fly" without actually creating additional class definitions (no duplicated source code) and with only minimal compile time increase.

Isolation essentially creates additional scheduling statements (i.e. *par*, *wait*, *waitfor*) for specific instances. Different segments are then generated for these statements and variable access information is separated. Textually, isolation creates an additional class with a different name (e.g. *B_iso1* in Fig. 2(a)) but all statements remain the same as in the original definition (e.g. `B` in Fig. 1(a)). Our compile-time optimization uses the same scheduling statements (no duplication), but still creates separate segments for different instances and attaches the variable access information accordingly. Instances are distinguished by unique identifiers (i.e. *instID*) which the compiler maintains and passes to the simulator. At run time, we then use the instance identifiers to distinguish the segments (e.g. in Fig. 1, *v3(seg3 starts)* for `waitfor 1` in line 12 of `b1`, and *v5(seg5 starts)* for `waitfor 1` in the same line 12 of `b2`).

Moreover, when processing a function call in a segment, we analyze the function body to obtain its variable access information since these variables also affect the same segment. We also need to process the definition of a called function to connect the segment graph correctly if additional segments are generated due to segment boundary statements inside the function (e.g. we process `b1.main` to find `waitfor 1` and `waitfor 2` in Fig. 1). This analysis could grow exponentially with the size of the design if there are frequent deep function calls. We avoid this by caching the information obtained during the function analysis, so that we can reuse this data the next time the same function is called. The time complexity of our static analysis is therefore practically linear to the size of the code.

In summary, we optimize the static analysis by using instance identifiers (instead of separate source code statements) and limiting the algorithm complexity by caching the data analyzed for function calls.

Before we present our algorithm in detail, we need to introduce a few definitions:

- *cacheMultiInfo*: a boolean flag for caching multiple sets of information for different instances; *true* if segment boundary nodes are created or interface functions[3] are called in this function; otherwise, *false* (default).
- *cachedInfo*: set of information for different instances:
  - *instID*: instance identifier, e.g. `Main.b1`;
  - *dummyInSeg*: a dummy segment as the initial segment of this set of cached information.
  - *carryThrough*: a boolean flag; *true* when the input segment carries through *fct* and is part of the output segments; otherwise, *false*, e.g. for `Main.b1`, `B.main`.*carrythrough* = *false* since *v1(seg1)* is connected to *v3(seg3)* and will not carry through `B.main`.
  - *outputSegments*: segments (except the input ones) that will be the output after analyzing this function *fct*, e.g. for `Main.b1`, `B.main`. *outputSegments*={*seg3*, *seg4*}.
  - *segAccessLists*: a list of segment access lists; the segments here are a subset of the global segments of the design.; this list only contains the segments that are accessed by *instID.fct*.

  Note that, if *cacheMultiInfo* is *false*, there is only *one* set of information for all the instances that call this function.

During the analysis, when a statement is processed, there is always an input segment list and an output segment list. For example, for the `while` loop (line 13), the input segment list is {*seg3*} and the output segment list is {*seg3*, *seg4*}. To start, we create an initial segment (i.e. *seg0*) for the design as the input segment of the first statement in the program entrance function, i.e. `Main.main()`.

We now present the optimized static analysis algorithm with

[3]Interface function definitions differ for different types of instances.

four phases:

- **Input:** Design model (e.g. design.sc)
- **Output:** Segment Graph, Segment Conflict Table.
- **Phase 1:** Create the *global* segment graph.
  As listed in detail in Algorithm 1, the complexity of this phase is $O(n)$ where n is the number of statements in the design, when no function needs to be cached for multiple instances.
- **Phase 2:** For each function, build a *local* segment graph with segment access lists. Here, we only add variables accessed in this function definition to the segment access lists. We do not follow function calls in this phase.
  As shown in detail in Algorithm 2, the complexity of this phase is $O(n_s)$ where n_s is the number of statements in the function definition. As shown for our example in Fig. 3(a), we do not follow function calls to `B.main` from `Main.main` but connect a *dummyInSeg* node instead. We also create two sets of `main` function cache information, Fig. 3(b) and Fig. 3(c), for the different instances of `B` since segment boundary nodes are created when calling `B.main`. Note that we do not know the instance path of the member variables in this phase. Therefore, we use port variable `sum` instead of its real mapping `Main.sum1` and `Main.sum2` here in Fig. 3(b) and Fig. 3(c), respectively.
- **Phase 3:** Build the *complete* segment access lists for each function.
  That is, we propagate function calls and all accessed variables are added to the cached segment access lists cascaded with proper instance paths. For our example, `b1` is cascaded to the instance paths of the member variables accessed in segment `B.main.dummyInSeg` for instance `b1` when analyzing `Main.main` if necessary[4]. We also use the function caching technique here to reduce the complexity of the analysis. The complexity of this phase is $O(n_g)$ where n_g is the total size of the segment graphs for each function.
- **Phase 4:** Collect the access lists for each segment in `Main.main` and add them to the global segment access lists. Since `Main.main` is the program entry (or root function), all the segments are in its local segments and the member variables have complete cascaded instance paths in the segment access lists. The real mapping of port variables can be found according to their instance paths. The complexity of this phase is $O(n_l)$ where n_l is the size of the local segment access list of `Main.main`.

As we intended, the proposed algorithm generates more precise segment conflict information without any modification of the design model. The complexity is, for practical purposes, linear to the size of the analyzed design. At this time, we do not support the analysis of recursive function calls.

Algorithm 1 Code analysis for out-of-order PDES, Phase 1

```
1:  Traverse the control flow graph (CFG) of the design.
2:  Create a segment boundary node for each scheduling statement
    and connect them accordingly.
3:  for all function fct do
4:    if fct is first called then process the definition of fct.
5:      if new segment nodes are created in fct then
6:        set fct.cacheMultiInfo = true;
7:        cache function information with current instID;
8:      else
9:        cache function information without instID; endif
10:   else
11:     if fct.cacheMultiInfo = false then
12:       use the cached information of fct;
13:     else
14:       if current instID is cached then
15:         use the cached information of fct.cacheinfo[instID];
16:       else
17:         process the definition of fct;
18:         cache function information with current instID; endif
19:     endif
20:   endif
21: end for
```

Algorithm 2 Code analysis for out-of-order PDES, Phase 2

```
1:  for all function fct do
2:    Traverse the control flow of fct.
3:    Create and maintain a local segment list localSegments.
4:    Use fct.dummyInSeg as the initial input segment.
5:    For each statement, add variables with their access type into
      proper segments.
6:    for all function calls inst.fct or fct do
7:      Do not follow the function calls. Just register
        inst.fct.dummyInSeg or fct.dummyInSeg as the input
        segments of the current statement and indicate that inst.fct
        or fct is called in the input segments.
8:      Add the output segments of the function call to localSeg-
        ments and use the output segments as the input of the next
        statement in the current function.
9:    end for
10: end for
```

IV. Experiments and Results

To demonstrate the effects of our optimization, we have implemented the regular and out-of-order PDES algorithms for the SpecC SLDL[5]. We have run two sets of experiments and have measured the results on the same host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz.

The first experiment uses a JPEG image encoder design (Fig. 4(a)) which encodes the three color components (Y, Cb, Cr) of the image in parallel since they are data independent. It performs the *DCT*, *Quantization* and *Zigzag* modules for each color component concurrently, and uses a sequential Huffman encoder at the end. The size of our input BMP image is 3216x2136 pixels.

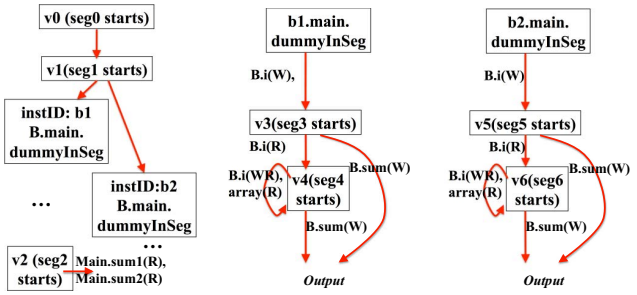
The second experiment uses a parallelized video decoder model (Fig. 4(b)) based on the H.264/AVC standard [12]. An H.264 video frame can be split into multiple independent slices during encoding. Our model uses four parallel slice decoders to decode the separate slices in a frame simultaneously. The H.264 stimulus module reads the slices from the input stream and dispatches them to the four slice decoders for parallel processing. A synchronizer block at the end completes the decoding of each frame and triggers the stimulus to send the

[4]Regular member variables accessed in different instances will not cause data hazards. Only port variables and interface member variables need the instance path for tracing the real mapping later.

[5]Our results should be equally applicable to SystemC SLDL.

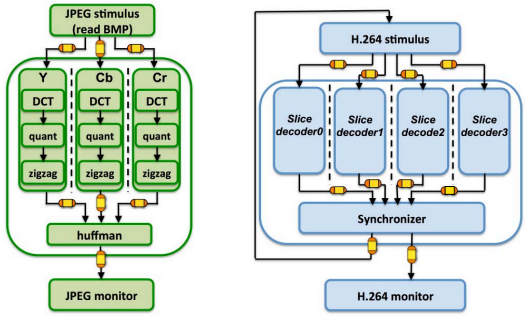
TABLE III
EXPERIMENTAL RESULTS FOR A SET OF JPEG IMAGE ENCODER AND H.264 VIDEO DECODER MODELS.

Model		Regular PDES		Unoptimized Out-of-Order PDES				Optimized Out-of-Order PDES	
				Original Design		Isolated Design			
		cmpl [sec]	sim [sec]	cmpl [sec] (speedup)	sim [sec] (speedup)	cmpl [sec] (speedup)	sim [sec] (speedup)	cmpl [sec] (speedup)	sim [sec] (speedup)
JPEG Encoder	spec	0.95	1.84	0.96 (-1.0%)	1.84 (+0.0%)	1.13(-15.9%)	0.92 (+100.0%)	1.01 **(-5.9%)**	0.96 **(+91.7%)**
	arch	1.24	1.82	1.25 (-0.8%)	1.70 (+7.1%)	1.37 (-9.5%)	0.95 (+91.6%)	1.28 **(-3.1%)**	0.92 **(+97.8%)**
	sched	1.30	1.80	1.31 (-0.8%)	1.72 (+4.7%)	1.43 (-9.1%)	0.94 (+91.5%)	1.34 **(-3.0%)**	0.96 **(+87.5%)**
	net	1.50	2.33	1.52 (-1.3%)	2.01 (+16.0%)	1.63 (-8.0%)	1.25 (+86.4%)	1.55 **(-3.2%)**	1.24 **(+87.9%)**
H.264 Decoder	spec	13.12	96.91	18.76(-30.1%)	96.97 (-0.1%)	18.13 (-27.6%)	60.33 (+60.6%)	12.25 **(+7.1%)**	60.25 **(+60.8%)**
	arch	12.20	99.86	17.89 (-31.8%)	100.30 (-0.4%)	18.46 (-33.9%)	60.77 (+64.3%)	12.77 **(-4.5%)**	59.78 **(+67.1%)**
	sched	18.34	99.80	24.23 (-24.3%)	99.44 (+0.4%)	24.80 (-26.1%)	60.96 (+63.7%)	18.85 **(-2.7%)**	60.29 **(+65.5%)**
	net	19.07	104.77	25.71 (-25.4%)	104.56(+0.2%)	26.06 (-26.8%)	66.25 (+58.1%)	19.54 **(-2.4%)**	66.00 **(+58.7%)**



(a) Main.main (b) B.main (Main.b1) (c) B.main (Main.b2)

Fig. 3. Function local SGs and segment access lists for the example in Fig. 1.



(a) JPEG image encoder. (b) H.264 video decoder.

Fig. 4. Design examples for experimental PDES algorithm comparison.

next one. Note that this design model is of industrial-size and consists of about 40k lines of code. We use a test stream of 1079 video frames with 1280x720 pixels per frame.

Table III lists our experimental results for both design models at four different abstraction levels (*spec*, *arch*, *sched*, *net*). We measure and compare the out-of-order PDES algorithms in compile and simulation time against the *regular PDES* implementation as reference.

The results shown in Table III clearly support the two main contributions of this paper. First, instance isolation is very effective in improving the simulation speed, as shown in the columns for the unoptimized out-of-order PDES. Second (and more important), our new analysis algorithm for optimized out-of-order PDES not only shows high speedup in simulation due to the automatic isolation "on-the-fly", it also shows only an insignificant increase in compile time of less than 6%.

V. CONCLUSION

Out-of-order parallel DE simulation (PDES) is an advanced technique for fast multi-core validation of ESL design models. In this paper, we exploit the idea of instance isolation and show that this technique significantly improves the efficiency of the static code analysis required by out-of-order PDES. Our optimized out-of-order PDES algorithm isolates instances in the design model automatically without any model modification and, for practical purposes, is linear in complexity due to the careful caching of analysis results for functions. Our experimental results show high gains in simulation speed for an insignificant increase in compile time.

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] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer, 2002.
[2] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Kluwer, 2000.
[3] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-Core Parallel Simulation of System-Level Description Languages," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 311–316, 2011.
[4] 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.
[5] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, pp. 606–609, 2010.
[6] 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.
[7] 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.
[8] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 440–452, Sept 1979.
[9] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 30–53, Oct 1990.
[10] D. Nicol and P. Heidelberger, "Parallel Execution for Serial Simulators," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, pp. 210–242, July 1996.
[11] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pp. 80–87, 2009.
[12] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, pp. 560 –576, July 2003.