Designer-in-the-Loop Recoding of ESL Models using Static Parallel Access Conflict Analysis

Xu Han
Center for Embedded Computer Systems
University of California
Irvine, USA
hanx@uci.edu

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

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

ABSTRACT

At the Electronic System Level (ESL), a well-defined design model enables early design space exploration and automatic synthesis on custom multiprocessor platforms. However, the initial design model is usually manually recoded from unstructured and sequential source code. To efficiently create cleanly structured and parallel models, this paper proposes a *designer-in-the-loop* approach on Eclipse platform where the system model is analyzed and *recoded* using automated functions. Particularly, advanced static analysis at compile time can guarantee that the parallelism in the model is safe and free from race conditions. Experiments using the tool with a class of graduate students show significant productivity gains and error reduction in model creation.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Programming Environments—*Integrated environments*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Design

Keywords

Executable Specification, ESL, Recoding, Eclipse

1. INTRODUCTION

Electronic System Level (ESL) design with proper methodology and tools, such as [5, 12], enables early design space exploration, high-level synthesis, and software refinement to cycle-accurate level. Here, a well-structured system-level model is critical as the input to the automation tools. In order to shorten the time to market, quickly creating clean and parallel ESL models is a prerequisite for building cost-effective MPSoCs.

SCOPES '13, June 19-21, 2013, St. Goar, Germany

However, most application codes are specified in sequential C code while an ESL model is typically described in a System Level Description Language (SLDL) such as SystemC or SpecC, with exposed parallelism and block structure. Manually *recoding* original C code to ESL models containing explicit structure, communication, and parallelism is very time-consuming, and the step to expose the parallelism in the application is especially difficult and error-prone.

This paper addresses this problem by a *designer-in-the-loop* approach similar to [6] where the system model is *recoded* and analyzed using automated functions guided by designer decisions. The proposed design flow speeds up the creation of a clean and parallel ESL model from original reference code of an application. In particular, we use static dependency analysis at compile time to guarantee that the *recoded* model is safe and free from parallel access conflicts. A case study with a class of graduate students on parallelizing a Canny Edge Detector example shows that designers not only find the proposed approach useful, but also can complete the recoding faster and with fewer errors.

1.1 Related Work

When time and budget are limited, automated refinement and parallelizing tools are desirable. [9] automatically transforms C programs into parallelized SystemC models based on user-directives in the input program. The parallelism here is limited to task-level pipelining. [10] uses an ILP-based approach on hierarchical task graphs to parallelize applications for embedded systems. The tasks in the application need to be clearly defined and parallelism needs to be exposed from the source code. Advanced parallelizing compilers, such as [1, 13], are still ineffective in exploiting thread-level parallelism in real-life applications. As studied in [14], only an average of 10% of the loops (which cover 12.5% of the total execution time) can be parallelized automatically for the chosen benchmarks using the Intel C++ Compiler 9.1.

In contrast, the proposed approach utilizes designer's knowledge and offers greater flexibility in the types of parallelism to exploit. Some parallelism can only be extracted with the understanding of the algorithms in the application. Also, a clean input model is usually a prerequisite for parallelization tools to work effectively.

Research work has been done to verify system-level models with parallelism in SystemC. Formal approaches, such as model checking, provide comprehensive coverage and guarantees on system descriptions. [17] proposed a trace driven approach to detect race conditions and deadlocks in sys-

tem models. The trace record can grow exponentially when the size of the design is increasing. [2] uses dynamic partial order reduction approach to address the state explosion problem. Exhaustive simulation is needed which is not efficient for checking industrial-size models. A simulation-based approach, such as [16], is also studied to analyze non-deterministic anomalies among parallel logical processes.

The proposed approach in this paper is based on static code analysis which is practically linear to the size of the design. Large designs can be completely checked with negligible compilation cost.

A good amount of research on the topic of 'programmer in the loop' is done in ExCAPE center [15], aiming to synthesize programmer's insights into the actual software implementation. Their approach involves automatic program completion with partial code specified by the programmer, code generation with desired input and output specified by the programmer, and synchronization insertion for parallel code. Instead of targeting general computing, our work focuses on system-level modeling. Our goal is to create the modularity and parallelism required in system-level models efficiently using automatic code analysis and transformations. In our context, the low-level software is generated by back-end ESL tools.

2. DESIGNER-IN-THE-LOOP RECODING

Creating parallel ESL models in SLDLs from existing sequential applications involves a number of tasks, as shown in Fig. 1:

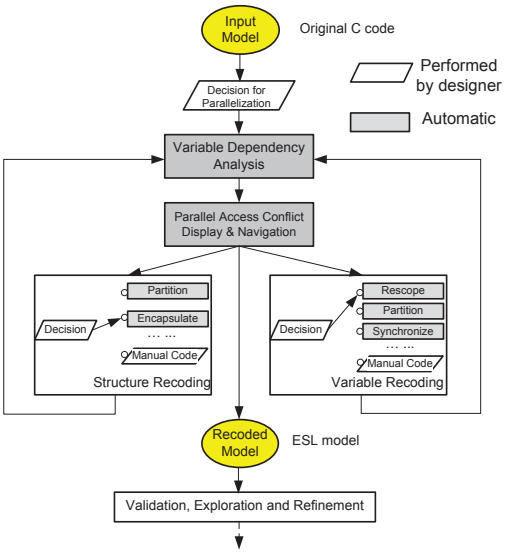


Figure 1: Designer-in-the-Loop Recoding Flow

Decision for Parallelization

The first step requires the designer to identify effective parallelism in the model. The designer also needs to identify the risk and avoid cases when parallelization downgrades the performance due to heavy data dependencies, frequent synchronization, or I/O bottlenecks. For these reasons, this step is kept manual in the design flow.

Variable Dependency Analysis

Resolving data dependencies is one major task in exposing parallelism. To guarantee the variables are in the correct scope when structures are recoded and the implemented parallelism is free from parallel access conflicts, the designer must locate the variables in related statements, and identify their access type and dependency on other statements. Due to its complexity, this process is lengthy and very error-prone if done manually.

In our work, the variable dependency analysis is automated using static code analysis in the compiler. This is the focus of this paper and is presented in detail in Section 3.

Parallel Access Conflicts Display and Navigation

With our extension of the on Eclipse platform (described in detail in Section 4), the designer is provided a monitor and checker for potentially dependent variables for each function or module during model development. The tool can also navigate instantly to any variable in the displayed list based on designer's selection so that a recoding action can be quickly performed.

Structure Recoding

Structure Recoding is needed to explicitly describe the hierarchy and parallelism in SLDL. This involves a number of transformations. *Partitioning* breaks large functional blocks into smaller ones. *Encapsulation* transforms functional blocks into modules (e.g. behaviors in SpecC or modules in SystemC). *Hierarchy* is recoded to expose the parallelism by instantiating and connecting newly created blocks and placing them for parallel execution.

These transformations are applied with the designer's input, especially with the knowledge of the target platform and design constraints. Here, we rely on existing transformations [6, 7]:

- Loop Splitting: With designer-specified parameters of trip count and the number of unrolls, this function creates different incarnations of the loop (with the same loop body) where each split loop iterates over different contiguous subsets of the loop index range.

- Encapsulating Functions: This function creates a new behavior from the body of a function call selected by the designer and a port list of the function interface. The created instance is placed and connected in its parent, and the original function call is replaced with a call to this instance.

Variable Recoding

The purpose of recoding variables is two-fold. (1) The functionality of the model must be maintained as its structure is recoded. The variables involved in restructuring must be recoded according to how they are used in each block by relocating them to proper scope or creating channels for them. (2) Data dependencies must be resolved when parallelism is implemented. In case of data parallelism on a distributed memory platform, data can be partitioned onto the processing elements. If communication is required among parallel tasks, variable accesses must be synchronized in a shared memory platform, or communication channels must be established in a distributed system.

As indicated by the back arrows in the recoding flow (Fig. 1), the designer iteratively recodes the variables in the list provided by the previous step.

The automated variable recoding currently includes [6]:

- Rescoping: Relocating variables to proper local, class, or global scope, is needed to maintain the recoded model, eliminate false dependencies, and indicate memory mapping. This transformation is automated with designer specified target variable and destination scope.
- Array Splitting: When different parallel tasks work on a slice of an array each, the array can be split into sub-arrays automatically with user-specified parameters of index ranges.
- Synchronization through Channels: When dependency exists among parallel tasks, the user specifies the variables used for communication and the dependent behaviors and then the function creates channels and inserts blocking *send* and *receive* calls for those behaviors.

To summarize, the model is iteratively recoded by the designer using automated transformation functions and guided by analysis results. Then the recoded model is used as input for validation, exploration, refinement and further generation of implementation models.

In general, the complex and iterative recoding process cannot be fully automated and requires 'designer-in-the-loop'. First, the task requires information about target platform and implementation constraints which may only be known by the designer. Second, manual coding is still needed when a model is recoded. For example, interfaces need to be defined after code partitions; data merging or collection may be necessary after data partitions. Third, having the designer in control provides flexibility and generally can produce better designs than fully automatically generated models.

3. VARIABLE DEPENDENCY ANALYSIS

This section describes the variable dependency analysis algorithm implemented in our proposed design flow.

At compile time, *variable access lists* (VarAList) are built statically at function level for data dependency checking. Each item in the list is a 3-tuple ***(Symbol, AccessType, InstancePath)***, where:

- ***Symbol***: the identifier of the variable.
- ***AccessType***: the variable access type, i.e. read-only (R), write-only (W), read-write (RW), or pointer access (Ptr).
- ***InstancePath***: the scope of the variable in the design instance hierarchy.

SLDLs support different types of variables that need to be analyzed separately. We distinguish and handle the following cases (variable examples are from the design in Fig. 2(a)):

- **Global variables (G)**, e.g. `array` in line 2: This case is handled directly as tuple *(Symbol, AccessType, NULL)*.
- **Local variables**, e.g. `j` in line 9: Local variables are stored on the function call stack and cannot be involved in any dependency due to parallel execution of the function. Thus, we can ignore them in the access lists.

```
1 #include <stdio.h>
2 int array[10] = {0,1,2,3,4,5,6,7,8,9};
3
4 behavior A(in int start,
5              in int end,
6              inout int sum)
7 {
8   int i;
9   void g(int j){
10    sum += array[j];
11  }
12  void main(){
13    i = start;
14    while(i <= end){
15      g(i ++);
16    }
17  }
18 };
```

```
behavior Main()
{
  int sum1, sum2;
  A a1(0, 4, sum1);
  A a2(5, 9, sum2);
  int main(){
    par{
      a1.main();
      a2.main();
    }
    printf("%d\n", sum1);
    printf("%d\n", sum2);
    return 0;
  }
};
```

(a) Example source code in SpecC.

Functions	Main.main()	A.main()	A.g()
Local Variable Access List	(sum1(M), R, NULL) (sum2(M), R, NULL)	(i(M), RW, NULL) (start(P), R, NULL) (end(P), R, NULL)	(sum(P), RW, NULL) (array(G), R, NULL)
Callee List	A.main()	A.g()	

(b) Function Local VariableAccess Lists

Functions	Main.main()	A.main()	A.g()
Variable Access List	~~(sum1(M), R, NULL)~~ upgrade to RW ~~(sum2(M), R, NULL)~~ upgrade to RW **(i(M), RW, a1)** ~~(0, R, NULL)~~ mapped to constant ~~(4, R, NULL)~~ mapped to constant **(sum1(M), RW, NULL)** **(array(G), R, NULL)** **(i(M), RW, a2)** ~~(5, R, NULL)~~ mapped to constant ~~(9, R, NULL)~~ mapped to constant **(sum2(M), RW, NULL)** ~~(array(G), R, NULL)~~ redundant entry	**(i(M), RW, NULL)** **(start(P), R, NULL)** **(end(P), R, NULL)** **(sum(P), RW, NULL)** **(array(G), R, NULL)**	**(sum(P), RW, NULL)** **(array(G), R, NULL)**

(c) Function Variable Access Lists

Figure 2: Simple design example with the variable access lists.

- **Instance member variables (M)**, e.g. `i` in line 8: A block or channel can be instantiated multiple times and then produce multiple variables with the same identifier. We distinguish these by their *InstancePath* in the *(Symbol, AccessType, InstancePath)* tuple.
- **Port variables (P)**, e.g. `start` in line 4: Ports are mapped to other variables during instantiation. We find the actual mapped variable by tracing up the known instance path and store that variable with its instance path.
- **Pointers**: We currently do not perform pointer analysis. For now, we conservatively mark pointer accesses as potential dependencies and leave the final decision to the designer.

To obtain the variable access information, we propose a three-phase code analysis algorithm:

- **Phase 1:** Function preprocessing.
  In this phase, we add the variables that are accessed in the statements of a function (or method) definition to a local VarAList. Instead of following the function call tree, we simply add the functions that are called in the function body to the list of callees. Fig. 2(b) shows the local VarALists for the functions in the example. Since we just scan the code and do not follow the function call tree, as shown in Algorithm 1, the time complexity of this phase is linear to the size of the design.

- **Phase 2:** Build Variable Access Lists.
  In this phase, we build the complete variable access lists for each function (or method). The complete *VarAList* contains not only the variables accessed locally in a function, but also those that are accessed indirectly by called functions. The Variable Access Lists get updated in this phase as shown in Fig. 2(c).

  Algorithm 2 shows that the compiler starts from the top of the function call tree and traverses down to each leaf function. This analysis could grow exponentially if each callee function is followed and there are frequent deep function calls. We avoid this by *caching* the variable access information for each function (line 15-20 in Algorithm 2). Instead of calling *BuildFctAccessList* to analyze the callee function *f*, we use the cached data if *f* is already analyzed. The time complexity of this phase is therefore also linear to the code size.

- **Phase 3:** Get Instance-Specific Variable Access Lists.
  In this phase, we build the complete InstancePath for the variables. In phase 2, the variable access lists are at function level and instance member variables cannot be distinguished. For instance in Fig. 2, *i* is a member variable of *behavior* A, and after phase 2, we have the entry *(i, RW, NULL)* in the *VarAlist* of *A.g()*. However, we cannot identify *Main.a1.i* or *Main.a2.i* at that level.

  In phase 3, we prepend the instance path to the variables so as to have the instance-specific entries in the VarAList, such as *(i, RW, Main.a1)* or *(i, RW, Main.a2)*.

  Note that not all variables are straightforward to analyze. The SLDL actually supports ports which are connected by port maps in the structural hierarchy of the design model. We need to trace the instance path for the mapping of the port variables so as to identify the real variables that are accessed. For example, *start* and *end* are ports of *behavior A*. Their real mapping are the two constants 0 and 4 for instance *Main.a1*. Therefore, the real variables accessed in *Main.main* are constants 0 and 4, not *Main.a1.start* and *Main.a1.end*, respectively.

  The time complexity of this phase is linear to the size of the *VarAList*.

Note that **Phase 1** and **Phase 2** are performed automatically by the compiler, while **Phase 3** is executed when the designer launches the analysis and recoding functions by clicking on the Eclipse interface (explained in the following section).

4. INTEGRATION WITH ECLIPSE

We have implemented the proposed design flow supporting SpecC as an Eclipse [4] plug-in. Fig. 3 shows the interface of our SpecC-extended Eclipse which integrates the recoding and analysis functions together with many other features, including:

Automatic Compiling

Whenever the user edits and saves a working model, the model is compiled in the background. At this time, **Phase 1** and **Phase 2** (described in Section 3) execute to build the data structures needed for parallel access conflict analysis.

Algorithm 1 Function Preprocessing to Build Variable Access Lists

```
 1: BuildFctLocalAccessList(fct)
 2: {
 3:   fct.localVarAList = BuildStmntAccessList(fct.topstmnt);
 4: }
 5:
 6: newStmntVarAList = BuildStmntAccessList(stmnt)
 7: {
 8: newVarAL = new VAList;
 9: switch (stmnt.type) do
10: case  STMNT_COMPOUND:
11:   for all subStmnt ∈ Stmnt do
12:     newVarAL += BuildStmntAccessList(subStmnt);
13:   end for
14: case  STMNT_IF_ELSE:
15:   newVarAL += ExtendAccess(stmnt.conditionVar, NONE);
16:   newVarAL += BuildStmntAccessList(subIfStmnt);
17:   newVarAL += BuildStmntAccessList(subElseStmnt);
18: case  STMNT_WHILE:
19:   newVarAL += ExtendAccess(stmnt.conditionVar, NONE);
20:   newVarAL += BuildStmntAccessList(subWhileStmnt);
21: case STMNT_EXPRESSION:
22:   if stmnt is a function call f() then
23:     Add f() to fct.callee_list;
24:   else
25:         newVarAL += ExtendAccess(stmnt.expression, ACCESS_NONE);
26:   end if
27: case  ...: /* other statements omitted for brevity */
28: end switch
29: return newVarAL;
30: }
31:
32: newExprVarAList = ExtendAccess(expression, ACCESSTYPE)
33: {
34: newVAL = new VAList;
35: switch (stmnt.type) do
36: case  EXPR_ASSIGNMENT: /*assignment*/
37:   newVAL += ExtendAccess(left_expr, W);
38:   newVAL += ExtendAccess(right_expr, R);
39: case  EXPR_ADD_ASSIGN: /*augmented assignment*/
40:   newVAL += ExtendAccess(left_expr, RW);
41:   newVAL += ExtendAccess(right_expr, R);
42: case  EXPR_DIVIDE: /*binary operation*/
43:   newVAL += ExtendAccess(left_expr, R);
44:   newVAL += ExtendAccess(right_expr, R);
45: case  EXPR_NOT: /*unary operation*/
46:   newVAL += ExtendAccess(left_expr, R);
47: case  EXPR_POST_INCREMENT:
48:   newVAL += ExtendAccess(left_expr, RW);
49: case  ...: /* other expressions omitted for brevity */
50: end switch
51: return newVAL;
```

Hierarchy View and Non-local Variables View

After the model compiled, its structure and hierarchy is visualized in *Hierarchy View*. The user is able to analyze the dependent variables for a single behavior instance or function by clicking it in the view window. Then based on the item selected by user, an *InstancePath* is computed, **Phase 3** runs in the background to build a corresponding Variable Access List which is immediately visualized in *Non-local Variables View*. Another use case is when the user selects multiple parallel candidates in *Hierarchy View*, their access lists are compared and potential parallel access conflicts are highlighted. This feature is best for checking parallel access conflicts during model development.

Parallel Access Conflict Analysis and Browser

An action **Parallel Access Conflict Analysis** is implemented to locate all potential parallel access conflicts in a model and display them in a browser. The action takes the intermediate representation after *Automatic Compiling* as

Algorithm 2 Build Variable Access Lists

```
1: BuildDesignAccessLists()
2: {
3:   /*Preprocess functions in the design*/
4:   for all fct ∈ Design do
5:     BuildFctLocalAccessList(fct);
6:   endfor
7:   /*traverse the function call tree from the top*/
8:   BuildFctAccessList(Main.main);
9: }
10:
11: BuildFctAccessList(fct)
12: {
13:   fct.VarAList = fct.localVarAList; /*add local access list*/
14:   for all f ∈ fct.callee_list do /*traverse the function call tree*/
15:     if (f.cached == true) then
16:       add fct.VarAList += f.VarAList; /*use the cache*/
17:     else
18:       BuildFctAccessList(f); /*build the function var access list*/
19:       f.cached = true;       /*and cache the information*/
20:     endif
21:   endfor
22: }
```

input, locates all parallel statements in the model, and applies *Algorithm 2* to all parallel executing instances to collect their variable access lists. Then it compares the variable access lists for each parallel statement, and displays matching entries (the same variable symbol and *InstancePath*) with writing conflicts.

Note that the parallelism in SpecC SLDL is described explicitly with *parallel statements*. Therefore, this analysis is conservative by examining all parallel statements in the design and thus covers all potential access conflicts (including false positives).

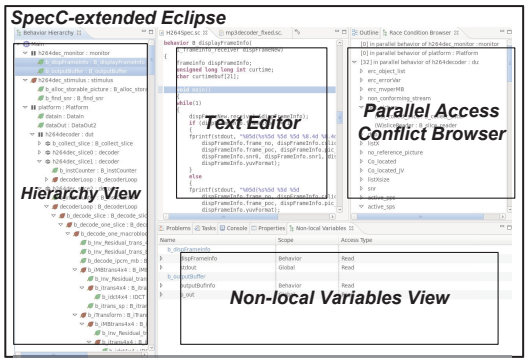


Figure 3: SpecC-extended Eclipse Platform

In summary, our Eclipse extension supports the designer in monitoring dependent variables for a behavior or function, checking potential conflicts among parallel blocks, checking parallel access conflicts over the entire design, and quickly making recoding decisions to the variables in question.

5. CASE STUDY: EDGE DETECTOR

As an example application, we use a Canny Edge Detector to demonstrate the proposed designer-in-the-loop recoding flow in a close-to-real-world setting [18]. The Canny Edge Detector is an image processing application which applies five successive functions to detect edges in an image. Starting from a sequential model of Canny (based on the C reference code from [3]), we follow the recoding flow (Fig. 1) to create a parallel system model.

- **Decision for Parallelization**: Using a profiling, the functional block *gaussian_smooth* is identified containing the highest amount of computation (~50%). Thus, the application can benefit most if we focus on *gaussian_smooth*. Its source code, listed in Fig. 4(a), shows that the function first creates a Gaussian kernel used to blur the image (this task is called 'Prep'), then blurs the image horizontally by filtering each pixel using its neighbors in X-direction (BlurX), and finally blurs the image vertically by filtering each pixel using its neighbors in Y-direction (BlurY). To reduce communication among parallel tasks, our parallelization strategy is to run BlurX on horizontal slices of the image in parallel, and BlurY on vertical slices the same way.
- **Variable Dependency Analysis**: There are 15 variables used in *gaussian_smooth*. Instead of tedious line by line manual inspection, our variable dependency analysis tool acquires all variable access information instantly, as shown on the right in Fig. 4(a). The designer can also view potential parallel access conflicts among parallel task candidates.
- **Structure Recoding**: To expose parallelism, the designer applies the partitioning and encapsulating recoding functions listed in Fig. 1 to recode *gaussian_smooth* into 3 SpecC behaviors, namely Prep, BlurX, BlurY. According to the available processing elements, new parallel statements are created to call multiple BlurX and BlurY instances. Some manual code is needed to adjust the loop index in order to take in different range of the input image array.
- **Variable Recoding**: The 15 variables identified are recoded iteratively based on designer's decisions. For example, counter variables *r,c,cc* are duplicated and localized in BlurX and BlurY. Input data *imgin* is recoded into ports and port maps. Some variables have multiple options of recoding. For example, *imgin* and temporary image *tempim* can be partitioned to separate processing elements to reduce communication load and memory usage. Alternatively, *imgin* and *tempim* can be accessed as a whole, if we assume shared memory. In the latter case, the designer can choose to ignore the (false) read and write conflicts for *tempim* among parallel tasks because these accesses are actually to non-overlapping regions.

The recoded models then serves as input to the ESL design flow. Here the recoded model shown in Fig. 4(b) for 4 processing elements speeds up the simulation by 1.6X on a quad-core host, and it has also been refined to a bus-functional TLM model using SCE tool suite [12] for design space exploration.

6. EXPERIMENTS AND RESULTS

We evaluate the designer-in-the-loop recoding approach in Eclipse on parallel SpecC models of 5 embedded applications. These models are developed in-house based on standard reference code.

6.1 Parallel Access Conflict Analysis on Embedded Applications

Table 1 summarizes the results of our *Parallel Access Conflict Analysis*. All models except JPEG encoder show potential parallel access conflicts. The designer needs to decide

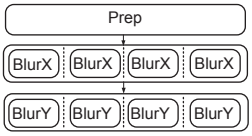
```
gaussian_smooth (unsigned char *imgin, int rows,
int cols, float sigma, short int *smoothedimg)
{  int r, c, rr, cc, windowsize, center;
   float tempim[SIZE], kernel[WINSIZE], dot, sum;
   /* Create a 1-dimensional gaussian smoothing kernel */
   make_gaussian_kernel(sigma, kernel, &windowsize);
   center = windowsize / 2;
   /* Blur in the x - direction */
   for(r=0;r<rows;r++){
      for(c=0;c<cols;c++){
         dot = 0.0;
         sum = 0.0;
         for(cc=(-center);cc<=center;cc++){
            if(((c+cc) >= 0) && ((c+cc) < cols)){
               dot += (float)imgin[r*cols+(c+cc)] * kernel[center+cc];
               sum += kernel[center+cc];}}
         tempim[r*cols+c] = dot/sum;
      }}
   /* Blur in the y - direction */
   for(c=0;c<cols;c++){
      for(r=0;r<rows;r++){
         sum = 0.0;
         dot = 0.0;
         for(rr=(-center);rr<=center;rr++){
            if(((r+rr) >= 0) && ((r+rr) < rows)){
               dot += tempim[(r+rr)*cols+c] * kernel[center+rr];
               sum += kernel[center+rr]; }}
         smoothedimg[r*cols+c] =
                     (short int)(dot*FACTOR/sum + 0.5); }}
}
```

behavior Prep
Reads: *sigma, imgin, windowsize*
Writes: *center*
RW: *kernel*

behavior BlurX
Reads: *cols, rows, kernel, image, center*
Writes: *tempim*
RW: *r, c ,cc, dot, sum*

behavior BlurY
Reads: *cols, rows, kernel, image, center, tempim*
Writes: *smoothedim*
RW: *r, c ,rr, dot, sum*

(a) *gaussian_smooth* source code



(b) recoded structure

Figure 4: Recoded *gaussian_smooth* module

whether the data sharing is indeed safe or not. After examining the reported variable lists, we found that the small-size models (Canny, JPEG and MP3) are already safe from parallel access conflicts, but medium/large models (GSM vocoder and H.264 decoder) actually contain dangerous variables which need to be recoded. Note that these reported conflicts did not show any problems during simulation, but would create errors in the real implementation. Thus, the proposed tool prevented a costly failure in the final implementation.

A dynamic race condition analysis method is presented in [8]. If we compare the conflicting variables reported for the same applications, higher numbers are reported in this work than those in [8] (subtracting the protected channel variables). This is because our static analysis covers the variables which are not in the execution path of the dynamic method. More importantly, the method in this work is conservative guaranteeing that all possible conflicts are included (including some false positives).

Table 1 also shows that the preprocessing (Phase 1 and Phase 2) and analysis (Phase 3 and list comparison) are sufficiently fast for the user. Results are reported instantly, except for the case of the H.264 decoder where the analysis needs less than 2.5 seconds on our 3.0 GHz workstation.

6.2 Classroom Experiment using Canny Edge Detector

To evaluate the effectiveness of the proposed design flow in a close-to-real-world setting, we have assigned the recoding task of the Canny Edge Detector described in Section 5 to a class of 68 graduate students [11].

6.2.1 Setup

Students in the class were instructed on embedded system design methods and trained in SpecC SLDL modeling before the experiment. Then each student was assigned the task to expose parallelism in a sequential model of canny. We offered the SpecC-extended Eclipse as an *optional* tool to them and provided them the corresponding instructions. The students were free to use any of their preferred editors as alternatives. Note that the SpecC-extend Eclipse contains all features described in Section 4 except *Parallel Access Conflict Brower* which was not available to the students at that time.

Meanwhile, we conducted an anonymous survey asking the students to report the tool they had used and their time needed to complete the assignment. If they reported having used our SpecC-extended Eclipse for parts of the assignment, we further asked for ratings on a scale of 1('did not use') to 5 ('very useful') about the usefulness of the tool's features. All students were given one week to do the assignment. After they finished, the course instructor graded their submissions by verifying the parallel simulation results against a golden model.

6.2.2 Results

After evaluating the survey, we found that not all students participated. 22 of them did not leave valid responses. For those who participated, 23 students used the SpecC-extended Eclipse, the remaining 23 students used other editors, such as vi, emacs and gedit. For the reason why some students chose not to use SpecC-extended Eclipse, 39% reported in the survey that their network connection was too slow or unstable (the tool was only accessible remotely), 9% stated that they do not like to use IDE or GUI, and others did not state any reason. According to the survey results, we classify the students into groups of 'Eclipse users', 'non-Eclipse users' and 'hybrid users' (for students who reported having used both Eclipse and other editors).

The reported working time and successful simulation from the instructor for all 3 types of users are summarized in Table 2. These not only indicate that Eclipse users spent 22% less time than non-Eclipse users on average, but also show that *all* students using SpecC-extended Eclipse produced successful models, while only 60.87% of non-Eclipse users did so. Thus, Eclipse usage reduced the recoding time significantly and improved the quality of the design.

	non-Eclipse	Eclipse	hybrid
Number of Students	23	18	5
Average Working Time [minutes]	281.30	219.39	227.40
Successful Model Simulation	60.87%	100.00%	100.00%

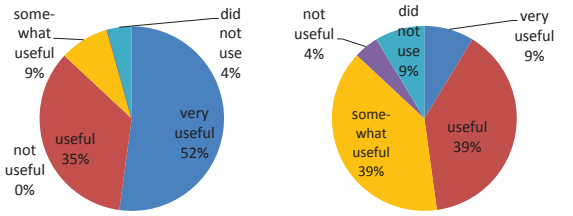
Table 2: Working Time and Successful Simulation

Students who have used SpecC-extended Eclipse also provided satisfaction ratings of its features. Note that the student used the features of *Hierarchy View* and *Non-local variables View* for dependency and parallel access conflict analysis during model development. The results are summarized in Fig. 5. Although *Hierarchy View* received much higher scores than *Non-local variables View*, overall, 95% of students consider both features 'somewhat useful', 'useful' or

Table 1: **Experimental results on *Parallel Access Conflict Analysis* for embedded applications.**

Application	Lines of Code	Max. # of Active Threads	Reported Conflicts	Resolved Conflicts	Preprocessing Time [sec]	Analysis Time [sec]
Edge Detector	1k	6	2	2 array accessed without overlap, safe	0.002	0.006
JPEG Encoder	2.5k	7	0	N/A	0.004	0.006
MP3 Decoder	7k	5	1	1 debugging value, safe	0.029	0.100
GSM Vocoder	16k	3	2	1 global value used to indicate overflow, safe 1 duplicated in parallel modules, resolved	0.024	0.018
H.264 Video Decoder	40k	8	50	45 store values constant to each frame, safe 1 structure accessed without overlap, safe 1 debugging value, temporarily used only 3 localized to class scope, resolved	0.210	2.442

'very useful'.



(a) Hierarchy View (b) Non-local Variable View

Figure 5: Survey Ratings of SpecC-extended Eclipse Features

7. CONCLUSIONS

In this paper, we proposed a designer-in-the-loop recoding flow to transform original C code into parallel executable ESL specification models. Our goal is to free the designers from manual code transformations and error-prone dependency analysis so that a clean and safe ESL model can be efficiently built.

Our static analysis and automated recoding functions supporting SpecC SLDL are integrated in Eclipse. Experiments with several embedded applications show that the parallel access conflict analysis is fast and effective. We also evaluated the proposed design flow with a class of graduate students. The results show that the proposed approach increases productivity significantly and avoids mistakes in system modeling.

In the future, we plan to improve and extend the designer-in-the-loop flow with additional recoding functions and further integration with other features of the Eclipse platform. We plan to reduce the false positives reported by the parallel access conflict analysis by analyzing synchronization mechanisms in the model. We also plan to apply the recoding methodology to SystemC.

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.

8. REFERENCES

[1] Automatic Parallelization with Intel Compilers. http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/.

[2] N. Blanc and D. Kroening. Race Analysis for SystemC using Model Checking. *ACM Transactions on Design Automation of Electronic Systems*, 15(3), June 2010.

[3] Canny Edge Detector. `ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src`.

[4] Eclipse CDT. http://www.eclipse.org/cdt/.

[5] W. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor SoC Platforms: a Component-based Design Approach. *IEEE Design and Test of Computers*, 19, Nov.-Dec. 2002.

[6] P. Chandraiah and R. Dömer. Code and Data Structure Partitioning for Parallel and Flexible MPSoC Specification Using Designer-Controlled Recoding. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 27(6):1078 –1090, June 2008.

[7] P. Chandraiah and R. Dömer. Computer-Aided Recoding to Create Structured and Analyzable System Models. *ACM Transactions on Embedded Computing Systems*, 11S(1), June 2012.

[8] W. Chen, C.-W. Chang, X. Han, and R. Doemer. Eliminating race conditions in system-level models by using parallel simulation infrastructure. In *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, pages 118–123, 2012.

[9] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl. SPRINT: A Tool to Generate Concurrent Transaction-Level Models from Sequential Code. *EURASIP Journal on Advances in Signal Processing*, 2007(1), 2007.

[10] D. Cordes, P. Marwedel, and A. Mallik. Automatic Parallelization of Embedded Software using Hierarchical Task Graphs and Integer Linear Programming. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[11] R. Dömer. EECS222A System-on-Chip Description and Modeling Assignment 4. `https://eee.uci.edu/12s/18422/Assignment4.pdf`.

[12] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer, 2001.

[13] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S.

Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[14] M. M. Islam. On the Limitations of Compilers to Exploit Thread-Level Parallelism in Embedded Applications. *IEEE/ACIS International Conference on Computer and Information Science*, 2007.

[15] U. of Pennsylvania. ExCAPE. `https://excape.cis.upenn.edu/`.

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

[17] 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.

[18] R. D. X. Han, Y. Samei. System-level modeling and refinement of a canny edge detector. Technical Report CECS-TR-12-13, Center for Embedded Computer Systems, University of California, Irvine, Nov. 2012.