# Pointer Re-coding for Creating Definitive MPSoC Models

Pramod Chandraiah
Center for Embedded Computer Systems
University of California, Irvine
pramodc@uci.edu

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

## ABSTRACT

Today's MPSoC synthesis and exploration design flows start from an abstract input specification model captured in a system level design language. Usually this model is created from a C reference code by encapsulating the computation and the communication using behaviors and channels. However, often pointers in the reference code hamper the necessary analysis and transformations. In this paper, we present an automated approach to re-code and eliminate pointers. By re-coding the pointer accesses to the actual variables, MPSoC models with definitive computational blocks that communicate using explicit channels become possible. Our pointer re-coding approach not only increases synthesizeability, analyzeability and verifiability by system tools, but also helps the designer in program comprehension. Our experiments show that this approach is not only feasible, but also effective in creating better models of real-life applications in shorter time.

## Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming

## General Terms

Algorithms, Design, Languages

## Keywords

MPSoC, Pointers, Re-coding, System Specification

## 1. INTRODUCTION

The initial abstract model required by System-on-Chip (SoC) design flows is an executable specification of the design, and often known as specification model [6] or Transaction Level Model (TLM) [7]. This model is typically captured in a System Level Design Language (SLDL), and its quality directly determines the effectiveness of the ensuing synthesis tools and the overall design implementation. Quality characteristics are stipulated by the underlying design flow and the individual tools through specification rules, guidelines and restrictions. Consequently, the system designer invests significant amount of design time in creating this input model.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*CODES+ISSS'07,* September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-824-4/07/0009 ...$5.00.

Today, C reference code obtained from standardizing committees and open-source projects is widely reused in creating this initial specification. Despite this reuse, the creation of the model takes a large amount of time. Re-coding of a C reference code into a MPSoC model in SLDLs such as SpecC [6] and SystemC [9] involves more than just syntactical changes. Separation of computation and communication, exposing parallelism, and other tool-specific optimizations are some of the necessary and time-consuming tasks performed. Re-coding a sequential reference C model into an effective and flexible MPSoC specification poses numerous challenges.

In this paper, we specifically address the problems posed by C pointers. We show how these pointers can be re-coded to create a specification model that is much easier to comprehend for the designer and more suitable for the system synthesis tools. We have integrated the necessary pointer analysis and the code transformations into an interactive SoC specification re-coder. Our results show that our pointer re-coding approach is effective and often eliminates ambiguity created by pointers. This helps significantly in creating a completely contained, definitive specification model suitable for design and exploration.

### 1.1 Motivation

Pointers in a C reference model affect system design mainly in two ways. First, for the designer who is creating the specification model from an alien C model, pointer indirection obscures the variable access information and impairs program comprehension. Secondly, pointers severely limit the effectiveness of system design tools. Pointers make the model less analyzable and hence negatively affect the synthesizeability and verifiability of the system. Many tools expect models without pointers as it significantly simplifies the design of the tools. When the input models do not meet these requirements, depending on the tool, the designer gets a warning, ineffective solution or even a wrong solution. For instance, many High-Level Synthesis (HLS) tools, including SPARK [10, 16], require the input to be free of pointers. Many system design tools expect an unambiguous and statically analyzable TLM specification. Since often C reference models are re-coded into TLM models, any pointer in the TLM poses problems to the synthesis and refinement tools by hindering code and data partitioning tasks. This necessitates the designer to manually eliminate pointers which is time-consuming and error-prone.

## 2. PREREQUISITE: POINTER ANALYSIS

Before performing pointer re-coding, the variable to which a pointer binds needs to be determined. Historically this problem is known as pointer analysis. The pointer analysis

subject is being pursued for the last two decades. Traditionally, it is used by compilers to address data analysis problems like constant propagation and live variables which are needed for program analysis, program optimizations and error detection. Besides these, pointers pose more challenges when programs meant for single-core single-memory architectures are used for creating system models of multi-core multi-memory platforms. In general, precise pointer analysis is undecidable [12, 15, 11] (a precise solution could solve halting problem), and existing algorithms trade-off between run-time efficiency and precision. In the next section, we briefly review different works in this area and then describe our pointer binding approach which is needed for our recoding. Please note that we are not proposing a new pointer analysis algorithm here. Pointer analysis is just a prerequisite to our pointer recoder.

## 2.1 Related Work

The general problem of pointer analysis can be divided into two parts, *Points-To* and *Alias* analysis. *Points-to* analysis attempts to statically determine the memory locations a pointer can point to. On the other hand, *alias* analysis attempts to determine if two pointer expressions could point to the same memory location. In the context of pointer re-coding, we are primarily interested in *points-to* analysis. The research in this area over the last two decades is summarized very well in [11]. Different pointer analysis algorithms differ in the precision of the analysis, efficiency of the algorithm, and scalability. Broadly, these algorithms can be classified based on two independent aspects, flow sensitivity and context sensitivity. Flow-insensitive algorithms [17, 20, 1] do not consider the control flow of the program and hence are faster than flow-sensitive algorithms [3], that can potentially offer more precise results. Flow-insensitive analysis can again be broadly differentiated as unification - based [17] or inclusion-based [1], the former being faster but less precise. The accuracy of such algorithms can be improved by adding context-sensitivity. A context-sensitive algorithm [19, 4] considers the effect of calling functions on the callee functions, and vice-versa. On the contrary, a context-insensitive algorithm is conservative and assumes that a callee affects all callers. Besides these two aspects, algorithms differ depending on whether composite data is considered as one object, or individual multiple objects. Further, high precision analysis of dynamically instantiated data structures requires shape analysis techniques [8].

Though the problem of pointer analysis has been widely addressed by the compiler community, the problem of pointer recoding has not been addressed. For sequential compilers, explicit pointer recoding might make little sense. However, pointer recoding becomes critical for MPSoC design, where the sequential specification has to be split and mapped onto multiple processors and multiple memories. The pointers need to be explicitly recoded in the specification so that the individual tools, which are otherwise not capable of handling pointers, can compile/synthesize/analyze/refine the input models.

## 2.2 Our Pointer Analysis

In its concluding note, [11] correctly remarks that pointer analysis must be tailored to meet the accuracy, efficiency and scalability requirements of the client applications. Since our pointer re-coder is interactive (for reasons stated in Section 4), run-time efficiency and scalability are an important concern. So for our re-coder, we choose a flow-insensitive and context-insensitive *points-to* analysis. Specifically, we chose Andersen's algorithm which is inclusion-based [1] over unification-based algorithm [17] as the former gave more precision to our needs than the later. This algorithm offers reasonable precision and performance suitable for our pointer recoder. Unlike Andersen's approach, we have our algorithm implemented on an Abstract Syntax Tree (AST) representation of the SoC model (similar to the approach taken by [2] for source to source transformations). Our points-to analysis determines the set of variables a pointer could point to in its life time. A points-to list generated by our analysis is shown in Figure 1. The algorithm assumes that after incrementing a pointer in an array, the pointer still points to the same array. For instance, pointer *p4* points only to *a* despite being incremented[1]. Depending on the program, the points-to list of a pointer can contain one or more variables. In Figure 1, all pointers except *p2* bind to one variable.

```
1.  int a[50], ab[50][16];
2.  int v1, v2,  x, y;
3.  int *p1,*p2, *p3, *p4, (*p5)[16], p6;

4.  p1 = &x;
5.  *p1 = y+1;
6.  if(condition) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  p3 = &ab[40][10];
10. *p3 = 100;
11. p4 = a;
12. p4++;
13. *p4++ = 1;
14. p5 = &ab[5];
15. p6 = p4+v1;
```

(a) Code with pointers

p1 → x
p2 → v1, v2
p3 → ab
p4 → a
p5 → ab
p6 → p4 → a

(b) Points-to list

**Figure 1: Points-to list generated by pointer analysis**

Our re-coding is performed after all pointers are analyzed and bound to their variables. We perform re-coding only on the pointers which bind to exactly one variable. If there is a possibility that a pointer could point to more than one variable (for example *p2*), then pointer re-coding is not performed. Such pointers are brought to the designer's attention and the decision is left to the designer to resolve this issue. The designer can use his application knowledge and provide accurate binding information to facilitate re-coding.

## 2.3 Abstract Syntax Tree

Our pointer analysis and re-coding are performed on the Abstract Syntax Tree (AST) representation of the program. The AST is generated from the input program of the design and captures the complete structure of the program. The AST preserves all structural information including, blocks, functions, channels, ports, statements, expressions, and so on. Figure 2(a) illustrates the amount of information stored in the AST. This data structure is necessary in order to reproduce the program back in its original form. Code generator pays special attention for code formatting, so that we can recreate the same program. Figure 2(b) shows the structure of the AST for a simple code snippet. Our primary pointer re-coding algorithm works at the statement and expression level of this data structure.

[1]Only in erroneous/non-portable programs, arithmetic on pointers can make a pointer point to different variables.

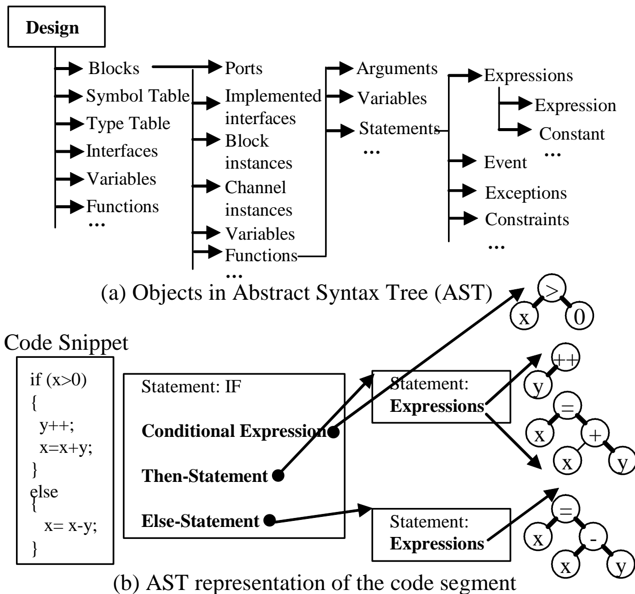



**Figure 2: Overview of Abstract Syntax Tree (AST)**

# 3. POINTER RE-CODING

Pointers are problematic because they implement multiple concepts. A programmer can use a pointer as a *value*, *alias*, *address*, or an *offset*. It is a *value* when the absolute value of the pointer is used, it is an *alias* when it points to more than one variable in its life time, an *address* when it is simply dereferenced, and an *offset* when the pointer points into an array and is manipulated using pointer arithmetic. Pointer recoding can be performed automatically in the latter two cases, that is when the pointer is not aliased or used as an absolute value. Re-coding involves replacing the indirect access to a variable through a pointer with direct access to the variable. Before presenting our algorithm, we outline our general approach to replace pointers:

- A pointer access to a scalar variable is replaced with the actual scalar. In Figure 3, this re-coding applies to variable $x$ accessed through pointer *p1*. Re-coding mainly affects the dereferencing operation of *p1* as shown in lines 4-5. The pointer initialization in line 4 is deleted as it is no longer necessary.
- For every pointer into an array, an integer is created which acts as index into the array. Then, a pointer access to vector variables is replaced with the array access operator ('[ ]') using the actual vector variable and the newly created index variable. In Figure 3, this recoding applies to array variables $a$ and $ab$. The newly created integers, *ip3, ip4, ip5* and *ip6* are used as indices. Arithmetic operations on pointers are replaced with arithmetic on the index variables, as shown in lines 12, 13 and 15 in Figure 3. Pointer initialization is replaced with initialization of the associated index variable with an offset expression (lines 9, 11, 14). We explain later how the offset expression is determined.

Note that, replacing pointer access with array access expression, though at source-level appear to create computation overheads, often are compiled away.

## 3.1 Pointer Expressions

The algorithm to re-code pointers must handle all expression types and take care of various coding scenarios. Given

```
1.  int a[50], ab[50][16];
2.  int v1, v2, x, y;
3.  int *p1,*p2, *p3, *p4, (*p5)[16], p6;

4.  p1 = &x;
5.  *p1 = y+1;
6.  if(condition) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  p3 = &ab[40][10];
10. *p3 = 100;
11. p4 = a;
12. p4++;
13. *p4++ = 1;
14. p5 = &ab[5];
15.  p6 = p4+v1;
```

(a) Code with pointers

```
1.  int a[50], ab[50][16];
2.  int v1, v2, x, y;
3.  int ip3, ip4, ip5, ip6;

4.  //Nothing here
5.  x =y+1;
6.  if(condition) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  ip3 =10;
10. ab[40][ip3] = 100;
11. ip4 = 0;
12. ip4++;
13. a[ip4++] = 1;
14. ip5 = 5;
15. ip6 = ip4+v1;
```

(b) Code with p1, p3, p4, p5, p6 substituted

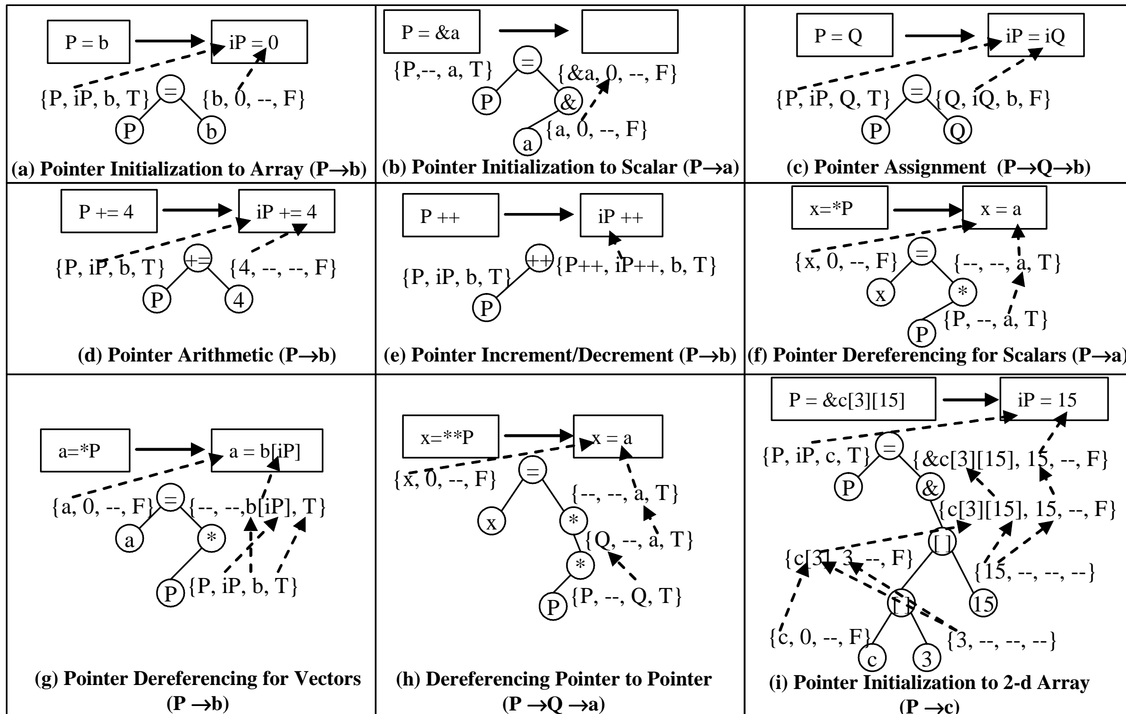
**Figure 3: Pointer recoding example**

a pointer and its type, our algorithm recursively traverses through the AST in a depth-first manner, searching for the specified pointer. Since each node in the AST has only local information, upon traversal, the node returns all possible results to the node above. The nodes above, which have a better global picture, choose the appropriate result from the results returned by the child nodes. Upon traversing a node, the recursive recoding function returns a tuple of 4 elements ($e1$, $e2$, $e3$, $e4$). $e1$ contains the unmodified original expression. $e2$ contains the expression of the index variable (if the expression processed was a pointer), or an offset expression (if the expression processed was a regular variable). If the expression processed is a pointer, $e3$ contains the target symbol to which the pointer is bound to. $e4$ is a boolean indicating if there was a positive pointer match.

We will now walk through the procedure to recode different pointer expressions. Figure 4 shows the way the algorithm operates on the AST when it is invoked to recode the pointer $P$ for important pointer usages. The 4-element tuple returned by a node traversal is shown in the curly brackets. In the example, we assume that $P$ points to either one of 4 variables, scalar $a$, 1-d vector $b$, 2-d vector $c$, or a pointer $Q$. At the time of recoding, the pointer binding is already performed and the only variable the pointer points-to is shown at the bottom of each example in Figure 4.

*Pointer initialization* to an array is shown in Figure 4(a). Our recursive recoder starts from the assignment ('=') node and reaches the identifier node $P$. Since $P$ is the pointer to be recoded, along with the original identifier ($P$), the index variable associated with the pointer ($iP$), the target variable the pointer binds to ($b$), and a boolean asserting the pointer match (*True*) are returned. Next, the other child node $b$ is reached. Three elements, the original identifier expression $b$, an integer offset of 0 (instead of an index variable, since $b$ is not a pointer) and a *false* boolean are returned. After returning to the *assignment* node, the pointer assignment is replaced with a new assignment expression formed using the index variable $iP$ and the offset expression *0*. In all examples, when the assignment (=) node receives one or all the 3 results ($e1$, $e2$, $e3$) an appropriate choice is made depending on the *node type* and $e4$. When $e4$ is *false* $e1$ is chosen. If $e4$ is *true* then the *node type* is used to decide between $e2$ and $e3$. If the *node type* is a pointer, $e2$ is chosen over $e3$.

Recoding *pointer initialization* to a scalar variable is shown in Figure 4(b). The pointer assignment expression is completely removed as the index assignment makes sense only for arrays. The necessary binding information ($P \rightarrow a$) is remembered by the recoder.




**(a) Pointer Initialization to Array (P→b)** **(b) Pointer Initialization to Scalar (P→a)** **(c) Pointer Assignment (P→Q→b)**
**(d) Pointer Arithmetic (P→b)** **(e) Pointer Increment/Decrement (P→b)** **(f) Pointer Dereferencing for Scalars (P→a)**
**(g) Pointer Dereferencing for Vectors (P →b)** **(h) Dereferencing Pointer to Pointer (P →Q →a)** **(i) Pointer Initialization to 2-d Array (P →c)**

**Figure 4: Recoding process for different expression types of a pointer *P***

*Pointer assignment* in Figure 4(c) is similarly replaced with an assignment expression of the indices of the two pointers (*iP, iQ*). Pointer arithmetic shown in Figure 4(d,e) is similarly replaced with arithmetic of pointer indices.
While recoding *pointer dereferencing* expressions, 3 main scenarios need to be addressed (Figure 4(f,g,h)) depending on the type of the target variable the pointer points to. If the target is a scalar, the dereferencing node will return just the target scalar ({*–, –, a, T*}) as in Figure 4(f). If the target is an array, an array access expression (*b[ip]*), formed using the target array and the index variable of the pointer, is used as the replacement expression, as shown in Figure 4(g).

If the target variable is another pointer, all 4 elements, the target pointer, the index variable of the target pointer, the variable pointed to by the target pointer and the matching flag ({*Q,–,a,T*}) are returned, as shown in Figure 4(h). The offset expression generated while initializing the pointer to the beginning of an array is simply 0. However, initializing a 1-D pointer to a specific element in a multi-dimensional array (as shown in Figure 4(i)), generates an offset expression based on the assumption that the pointer is used to access elements only within that dimension. This offset is propagated upwards through the AST, as shown by dotted arrows in Figure 4(i), and is used to create the index variable initialization. When the information at the node is limited, as in case of expressions $++, +, -, /, *, \&, \| \dots$, no decision is made about choosing the correct result. Under such circumstances, all the 4 results returned by individual child nodes are combined using the operation type of the current expression, as shown for *P++* in Figure 4(e), and passed to the parent node.

## 3.2 Pointer to Multi-dimensional Array

Recoding a pointer to a multi-dimensional array requires more attention. When a 1-dimensional pointer is used to access a multi-dimensional array, properly replacing the pointers with the actual array variable requires separate index variables for each dimension. However, this would result in additional overhead, because initialization and arithmetic on pointers will be translated into multiple initializations and arithmetic operations involving each index variable. To avoid this, we associate only one index variable with the pointer, based on the assumption that the pointer is used to point to only the elements across one dimension[2]. For example, the pointer *P* in Figure 4(i), is assumed to point only to 20 elements in row 3 of the array *c*. Thus, the index variable *iP* can only range from 0-19. More specifically, *P* is bound to sub-array *c[3]* and is used to replace any dereference expression of *P*. Figure 3 shows such a recoding for pointer *p3*.

## 3.3 Pointer Dependents

If it is determined that two pointers are dependent on each other, then recoding one of them requires recoding the other. For instance, if we are recoding pointer *P* and *P* depends on *Q* (as in expression $P = Q+4$), then this requires recoding *Q* along with *P*. A preparation stage identifies such dependent pointers and creates a list. The original pointer and these dependent pointers are then iteratively recoded as in Listing 1.

**Listing 1: Algorithm: Main pointer recoder**

```
MainRecodePointer (Ptr, Type) {
  While(Ptr)
     assert(PointerBindsToSingleVariable(Ptr))
     RecursivePointerRecoder(Ptr, Type)
     Ptr = GetNextDependentPointer(Ptr)
}
```

[2]Please note that this is a safe assumption for proper ANSI-C code, even if pointer arithmetic is used that crosses from one dimension to the next.

```
1.  int A, B,  *P, *Q;
2.  char* R;
3.  void* S;
4.  P = (int*) malloc(10*sizeofint())
5.  if (P)  // P cannot be recoded
6.  { // Code …}
7.  Q = &A;
8.  *Q = 1;
9.  Q++;  //Q cannot be recoded. It points to a scalar and is being incremented
10.R = (char*) (&B) //R cannot be recoded. A char* points to an Integer
11.*R = 0; R++; *R=0;
12.S = (int*)(&B) //S cannot be recoded. A void* points to an Integer
```

**Figure 5: Pointers that cannot be recoded.**

## 3.4 Pointer as Function Arguments

Pointers that appear as function arguments also need recoding. A pointer argument is replaced with the target variable and the index variable of the pointer. A dereferencing pointer argument is replaced with just the variable it points to. Besides recoding the arguments, the corresponding function parameter must also be recoded to change the function signature. This recoding is scheduled and is recoded later along with the other dependent pointers.

Note that by replacing the pointer arguments with actual variables, a call-by-reference is changed to a call-by value. Hence immediately after this recoding, the program, though syntactically correct, semantically is not the same anymore. However, this is just an intermediate step. By converting these functions to modules/behaviors and parameters into ports, the proper semantics will be restored as the ports contain direction information (in, out, inout). Transformations to create behaviors and ports is outside the scope of this paper.

## 3.5 Restrictions

Though most practical pointer usages can be recoded by our approach, there are some limitations. We cannot recode pointers under circumstances as listed in Figure 5.

- Our recoder is meant only for pointers to static or stack variables, not for dynamically allocated memory.
- It is not possible to recode pointers whose values are being read for absolute use, for example $P$ in Figure 5 which is read in line 5.
- It is not possible to recode pointer to scalars, if an arithmetic operation on the pointer is performed. Pointer $Q$ in Figure 5 shows this case. Note that such operations are not ANSI-C compliant.
- Operations involving different pointer types are not recoded. For example, pointer $R$, a character pointer and $S$, a void pointer, are being used to point to an integer.

Despite these restrictions, we find it possible to recode the large majority of coding scenarios in practical sources. Most often pointers are used in the C model for the sake of convenience. Such pointers can be recoded.

# 4. INTERACTIVE SOURCE RECODER

Though some pointers can be recoded fully automatically, some of the design decisions can only be made by the designer. For instance, since the underlying pointer analysis algorithm is not precise in all situations, it may not be possible to automatically recode all pointers. Designer attention becomes necessary to resolve some of the ambiguity due to pointers. Besides the limitations in the pointer analysis algorithms, some occasional pointer usage scenarios described in Section 3.5 are difficult to be recoded. Ideally, the designer would want to recode only the pointers that interfere in creating a specific model. The designer also wants to have flexibility on the program scope over which the recoding is performed. Recoding all the pointers in the program will also negatively affect the original readability of the program. Moreover, since this pointer recoding is used in the context of MPSoC specification generation, the designer would want to generate a model most suitable for her/his design flow and underlying architecture. Meeting these requirements necessitates a designer-controlled environment, where the designer can make the design decisions and automation is available to perform the tedious recoding. To aid the designer in coding and re-coding, we have integrated our pointer recoding into a source *re-coder*. The source re-coder is a controlled, interactive approach to implement analysis and recoding tasks. It is an intelligent union of editor, compiler, and powerful transformation and analysis tools. The re-coder supports re-modeling of C-based SLDL models at all levels of abstraction. It consists of 5 main components:

- Textual editor maintaining textual document object
- Abstract Syntax Tree (AST) of the design model
- Preprocessor and Parser to convert the document object into AST
- Transformation and analysis tool set
- Code generator to apply changes

With our setup the designer binds all pointers with a single click of a button, following which she/he invokes the pointer recoder on individual pointers. The source code transformations are performed and presented to the designer *instantly*. The designer uses his application knowledge to resolve any unresolvable pointer ambiguities interactively. The designer can also make changes to the code by typing and these changes are applied to the AST *on-the-fly*, keeping it updated all the time. This intelligent mix of application knowledge and the automation of the recoding makes our pointer recoder very effective.

# 5. EXPERIMENTS AND RESULTS

The main advantage of recoding pointers is to enhance program comprehension for the designer and to make the model conducive for tools with limited or no capability to handle pointers. Our interactive source recoder makes pointer recoding feasible and enables it to be useful on real-life embedded source codes. To show this, we obtained the openly available embedded benchmarks listed in Table 1. For each example, the table lists the number of pointers that we could recode. Since operations, such as file I/O, typically become part of the testbench, we examined the above examples in the context of the kernel functions listed. Some of the pointers required user intervention. For example, in case of *FFT*, all the 4 pointers were being used as *value* and could not be recoded (as explained in Section 3.5). Overall, our pointer recoder was effective in recoding 83% of the pointers in the listed examples.

To demonstrate the productivity gains, we applied the source recoder on industrial strength design examples, such as a MP3 audio decoder and a GSM vocoder, each spanning, thousands of lines of code. From these C codes we created a specification model in SLDL suitable for design exploration and synthesis using a top-down system synthesis tool-set. The number of behaviors in the resulting specification models are given in Table 2. The design tools performed explo-

**Table 1: Pointer re-coding on different benchmarks**

| Example | Applicable functions | Re-coded pointers |
|---|---|---|
| adpcm [13] | *adpcm_coder(), adpcm_decoder()* | 6/6 |
| FFT [13] | *fft_float()* | 0/4 |
| sha [13] | *sha_transform()* | 1/1 |
| blowfish [13] | *BF_encrypt(), BF_cfb64_encrypt()* *BF_cbc_encrypt()* | 10/10 |
| susan [13] | *susan_corners(), susan_principle()* *susan_edge()* | 13/17 |
| Float-MP3 decoder [14] | *decodeMP3()* | 14/16 |
| Fix-MP3 decoder [5] | *III_decode(), synth_full()* | 22/23 |
| GSM | Across the program | 17/17 |

**Table 2: Productivity factor**

| Quantities | GSM | Fix-Point MP3 | Floating-Point MP3 |
|---|---|---|---|
| Lines of C code | 13K | 8.7K | 3.6K |
| Functions in C Model | 163 | 67 | 30 |
| Behaviors in Spec. Model | 70 | 54 | 43 |
| Interfering pointers | 17 | 23 | 16 |
| Pointers recoded | 17 | 22 | 14 |
| Automatic Re-coding time | $\approx$ 1.5 min | $\approx$ 1.5 min | $\approx$ 1 min |
| Estimated Manual time | 170 mins | 220 mins | 140 mins |
| Productivity factor | 113 | 146 | 140 |

ration by mapping code and data partitions in the model to different processors and memories. This required that the input model is free of pointers which otherwise would negatively interfere in partitioning code and data. As shown in Table 2, in case of floating-point MP3 code, there were totally 16 pointers that interfered in creating the unambiguous specification. Out of 16, 14 could be recoded using our pointer re-coder. Two pointers could not be recoded because of two limitations (mentioned in Section 3.5), (a) the absolute value of the pointer was being read, (b) the pointer could point to more than 1 variable at run-time.

Using our source recoder, the pointers were eliminated in a matter of minutes. This automatic re-coding time is shown in Table 2. In the absence of our pointer recoder, designer performs this step manually. To obtain the manual time, we first manually recoded different pointers in different examples using Vim [18], an advanced text editor with block editing capability, and arrived at a mean-time of 10 manual minutes per pointer. The estimated manual time shown in Table 2 is obtained using this average time. Clearly, using our source recoder to recode pointers results in productivity gains[3] in the order of hundreds.

# 6. CONCLUSIONS

Due to the large availability of C reference models, the design of today's embedded systems often starts from a C-reference code typically obtained from open-source projects and standardizing committees. These C models are reused to create a system model in the desired system level design language. Though this code reuse speeds up the design process, it poses numerous challenges. Presence of pointers in the input C models is one such issue. The ambiguity introduced by the use of pointers in the C model presents problems to the exploration and synthesis design flows. Most of today's synthesis and verification tools are not designed to handle pointers, as their primary goal is to address other challenging tasks. These tools either cannot handle pointers, or result in inefficient solutions in presence of pointers. To overcome this limitation in the tools, designers invest significant time and effort to create definitive unambiguous input models by recoding pointers.

In this paper, we proposed pointer recoding that can replace the pointers in the input models with actual variables. Resolving pointer ambiguity is a hard problem and a complete solution in general is not available. Hence, to be effective on real life examples, the pointer recoder needs to be interactive. By making our pointer recoder available in the form of an intelligent editor, the designer can selectively recode selected pointers in the desired scope and realize the code transformations *on-the-fly*. Pointers, that cannot be analyzed, can be manually resolved through the editor. Because of this interactive nature of our recoder, the recoder is very effective on real-life design examples.

We have shown the effectiveness of our recoder in recoding pointers in embedded examples and the resulting productivity gain in the order of hundreds. Existing tool chains, which have limited or no capability to handle models with pointers, will immensely benefit from our pointer recoder.

[3] In general, measuring productivity is a difficult task. Factors such as designer's experience and tools used must be considered for accurate measurement of productivity gains. Since our improvements show multiple orders of magnitudes, small adjustments to measurement accuracy will not make any significant difference.

# 7. REFERENCES


[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] M. Buss, S. A. Edwards, B. Yao, and D. Waddington. Pointer analysis for source-to-source transformations. In *SCAM*, 2005.

[3] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.

[4] M. Fahndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.

[5] MAD fix point mp3 algorithm implementation. http://sourceforge.net/projects/mad/.

[6] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[7] F. Ghenassia. *Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*. Springer-Verlag, 2006.

[8] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for c. In *Languages and Compilers for Parallel Computing*, 1995.

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

[10] S. Gupta et al. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 2004.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.

[12] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4), 1992.

[13] MiBench, A free, commercially representative embedded benchmark suite. http://www.eecs.umich.edu/mibench/.

[14] MPG123. http://www.mpg123.de/mpg123/mpg123-0.59r.tar.gz.

[15] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5), 1994.

[16] Restrictions on Input "C" Code for SPARK. http://mesl.ucsd.edu/spark/download/docs/featureList.html.

[17] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[18] Vim, advanced text editor. http://www.vim.org/index.php.

[19] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN PLDI*, 1995.

[20] S. Zhang et al. Program decomposition for pointer aliasing: a step toward practical analyses. In *SIGSOFT*, 1996.