SpC: Synthesis of Pointers in C

Application of Pointer Analysis to the Behavioral Synthesis from C

Luc Séméria
lucs@azur.stanford.edu

Giovanni De Micheli
nanni@galileo.stanford.edu

Computer System Laboratory, Stanford University
Stanford, CA 94305

ABSTRACT

As designers may model mixed software-hardware systems using a subset of C or C++, we present SpC, a solution to synthesize and optimize a C model with pointers. In hardware, a pointer is not only the address of data in memory, but it may also reference multiple variables mapped to registers, ports or wires. Pointer analysis is used to find the point-to-set of each pointer in the program. In this paper, we address the problem of synthesizing and optimizing pointers to multiple variables and array elements. Temporary variables are defined to optimize loads and stores by minimizing the number of live variables. The combinational logic can also be reduced by encoding the pointers values. An implementation using the SUIF framework is presented, followed by some case studies such as the synthesis of a 2D IDCT.

1. INTRODUCTION: SYNTHESIS FROM C

Different languages have been used as an input to behavioral synthesis. Hardware Description Languages (HDLs), such as Verilog HDL and VHDL, are the most commonly used. However, designers often write system-level models using programing languages, such as C or C++, to estimate the system performance and verify the functional correctness of the design. To implement some parts of the design in hardware using synthesis tools, they must manually translate these parts into a synthesizable subset of HDL. This process is both time consuming and error-prone.

The use of C or a subset of C to describe both hardware and software would accelerate the design process and facilitate the software/hardware migration. Designers could describe their system using C. The system would then be partitioned into software and hardware blocks, implemented using synthesis tools.

In order to help designers refine their code for hardware synthesis, we are trying to synthesize the full ANSI C standard [5]. This task turns out to be particularly difficult because of dynamic memory allocation, function calls, recursion, type casting and pointers. In this paper we will focus on the problems related to the use of pointers in C.

Different subsets of C and C-like HDLs have been defined for synthesis. First, HardwareC [7] is a language with a C-like syntax and a cycle-based semantic. It doesn't include pointers, recursion and dynamic memory allocation. Cones [14] from AT&T Bell Laboratories is an automated synthesis system that takes behavioral models written in a C-based language [2] and produces gate level implementations. Here, the C model describes circuit behavior during each clock cycle of sequential logic. This subset is very restricted and doesn't contain unbounded loops or pointers. More recently, Compilogic [12] proposes a solution for translating C into an RTL description in Verilog. For synthesis, they assume that pointers are either memory references or parameters passed by reference without aliasing. Finally, SCENIC [8] from Synopsys is a synthesizable subset of C which uses C++ constructs. Even though any C or C++ code can be included in the SCENIC environment, the synthesis of code with pointers and dynamic memory allocation has not been addressed to date.

For hardware-software codesign, the CoWare system [1] uses C/C++ as a language base for system specification. Additional constructs have been introduced for defining blocks and concurrency. This description is used to synthesize the interfaces between the blocks but the actual synthesis of the blocks into hardware is left to the user. COSYMA [4] uses C*, another superset of C with processes and timing constraints. During hardware synthesis, functions are inlined and pointers are only treated as memory references.

In software, pointers represent addresses in memory. For example, they are used to pass parameters by reference, access array elements or address dynamically allocated memory. Data-flow-analysis problems such as *reaching definition* and *live-variables analysis* are widely used to optimize and parallelize programs. They all rely on knowing which variables are accessed at each statement. In order to analyze a program involving pointers, it is necessary to have information about what each pointer points to. Different pointer-analysis techniques (e.g [16], [17], [11], [13]) exist for computing the point-to information. For hardware synthesis, we also need to know which variables are accessed at each statement. Therefore, pointer analysis could be used for the behavioral synthesis of C models.

In this paper, we present a novel application of pointer analysis to the synthesis and optimization of C models with pointers. In

ICCAD98, San Jose, CA, USA.

Section 2, we define our synthesizable subset of C and how pointers can be removed. In Section 3, we discuss different techniques for optimizing the code. Then in Section 4, we present our implementation which synthesizes and optimizes C code with pointers using SUIF and Behavioral Compiler. Some results are given for different examples and an implementation of a 2-dimensional inverse discrete cosine transform (IDCT).

2. SYNTHESIS OF POINTERS

In software, the semantics of pointers is the address of an element in memory. This definition implies that the C program is targeted to a virtual architecture composed of one memory in which everything is stored. Even though `register` declaration may allow programers to specify the variables to place in registers, the assignment of variables to registers is generally done by the compiler. The notion of caches, memory pages, are transparent to programers.

In hardware, at the behavioral level, designers want to have control on where data are stored and want to optimize the locality of the storage. Typically, a design contains multiple memory banks, register files, registers and wires. Pointers may be used to reference any variable no matter where its information is available. As a result, pointers must be considered as references: references to memory elements, registers, wires or ports. In particular, pointers can be used to read and write data. In this paper we call the action of reading data using a pointer a *load*. Subsequently, a *store* is the action of writing data using a pointer.

The synthesis of pointers consists of synthesizing the appropriate logic for accessing data. For this purpose, we want to change the addresses into numbers and replace *loads* and *stores* by some assignments involving regular variables.

Example 1. *Consider a pointer* `p` *that points to* `a` *or* `b`. *If we associate the value* `0` *with* `a` *and* `1` *with* `b`, *we can remove the pointer. First, for the addresses, instead of* `p=&a` *and* `p==&b`, *we can write* `p=0` *and* `p==1`.

Then a load `(c=*p)` *can be replaced by:*

```
if(p==0) c=a; /* case p==&a */
else c=b;     /* case p==&b */
```

Finally a store `(*p=c)` *can be replaced by:*

```
if(p==0) a=c; /* case p==&a */
else b=c;     /* case p==&b */
```

In Example 1, after associating a number with each variable the pointer may reference, we remove *loads* and *stores* by inserting branching (e.g. `if`) statements. This requires to know at compile-time the *point-to-set* of each pointer (i.e. the set of variable the pointer may point to). For this purpose, we will be using pointer analysis, also called alias analysis.

2.1 Pointer Analysis

Pointer analysis is a compiler technique to identify at compile-time the potential values of the pointers in the program. This information is used to determine the set of variables the pointer may point to. For synthesis, in the case of *loads* and *stores*, we want to synthesize the logic to access or modify the variable referenced by the pointer. For this purpose, the point-to information must be both *safe* and *accurate*: *safe* because we have to consider all variables the pointer *may* reference and *accurate* because the smaller the point-to-set is, the less logic we have to generate. We can distinguish two types of analyses:

- *flow- and context-insensitive:* the analysis [13] doesn't distinguish the order in which the statements are executed (*flow-insensitivity*) and the different calls of a function (*context-insensitivity*). This interprocedural analysis has an almost-linear complexity. It can be used to analyze very large programs but the point-to information is rather inaccurate. Within a procedure, flow-insensitive analysis gives global information (valid for all references in the code) rather than the information specific to each reference. Similarly, in the case of function calls, context-insensitive analysis propagates the information from the call-site, through the called function, and back to *all* call sites. With flow- and context-sensibility, we are expecting more precise results.
- *flow- and context-sensitive:* this analysis provides more accurate results. It distinguishes the different paths of control within the program and the different calls of a function. One implementation [16], [17] by Wilson and Lam, within the SUIF framework, can efficiently support the full-featured ANSI C with good accuracy. Even though the complexity of the analysis can be exponential, it is not a limitation for hardware synthesis because we deal with rather small and simple programs. Beside, most of the inaccuracy comes from features such as dynamic memory allocation, type casting, recursion and recursive data structures. And, a priori, these features won't be used for modeling hardware.

The second type of analysis is more appropriate for hardware synthesis. In our case, the complexity of the analysis is not an issue, and the coding style for modeling hardware leads to rather accurate results. We are especially interested in the representation of arrays, structures and variables. Here, the analysis doesn't distinguish the different elements within the array but it distinguishes the different instantiations of variables and structures. This makes sense since all elements of an array are usually alike.

Our implementation uses a flow- and context-sensitive analysis. The aliasing information is then used to encode the pointers value and to generate the appropriate logic for accessing the data.

2.2 Resolution of Pointers

Let us start with the definition of our synthesizable subset of C. Our subset contains pointers to variables which can be stored in multiple memories, registers or wires. Pointer arithmetic is only allowed for pointers to array elements. Since memory blocks are instantiated at compile time, pointers to dynamically-allocated memory whose size is unknown at compile time are not allowed. This implies that, in general, `malloc`, `free` and recursions are not supported. Nevertheless, `malloc` followed by `free` could be allowed (treated as local variables in function calls) as well as tail recursion. The problem of pointers to functions and type casting is not addressed either.

The resolution of pointer can be done in three steps. First we analyze the pointers in the program to know which variables are referenced. Then we replace the *loads* and *stores*. Finally we encode the pointers value.

2.2.1 Replacing the Loads and Stores

After pointer analysis, the first task is to remove *loads* and *stores*. For this purpose, given a pointer `p`, we define the following variables:

- `star_p`: the value of the variable the pointer `p` points to (i.e. `*p`).

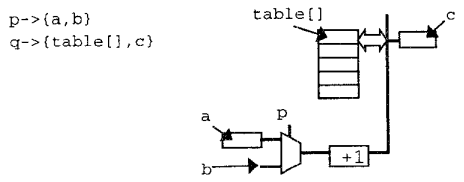

Figure 1: Implementation of *q=*p+1

- p_index: the offset within the array (defined only in the case of a pointer to an array element).

With the previous restrictions on the subset, *loads* and *stores* can be replaced by case statements at compile time. When the data are stored in registers, the case statement corresponding to a *load* will be implemented using a multiplexor controlled by the pointer's value. In the case of a *store*, some control logic will be generated to update the proper variable. We mention here that the value of the pointers will be encoded in a second pass. In the case of references to array elements in a memory, *loads* and *stores* are simply treated as memory accesses.

Example 2. *Consider the code segment* *q=*p+1 *where* p *points to* a *or* b *and* q *points to either an element of* table[] *or* c. *We create the variables* star_p, star_q *and* q_index. *The* loads *and* stores *are then replaced by the following code:*

```
switch p:
    case &a: star_p = a;
    case &b: star_p = b;
star_q = star_p + 1;
switch q:
    case table: table[q_index]=star_q;
    case &c:    c = star_q;
```

This code cannot be directly synthesized. A second pass is necessary to remove the addresses '&'.

The removal of *loads* and *stores* can be done in one pass. For each *load* (...=*p), we look at the point-to-set of the pointer at this instruction and generate the case statements that defines star_p according to the value of p and p_index. The *load* instruction is then replaced by an assignment from star_p. For each *store* (*p=...), we also look at the point-to-set of p at this instruction. The *store* is then replaced by an assignment to star_p and case statements are inserted to update the value of the variables p points to. In the case of a pointer to an array, pointer arithmetic is supported by changing the value of the index: the value of p_index is initialized when p gets the address of the array element. Then, the index is modified instead of p.

2.2.2 Encoding the Addresses

During a second pass, the value of the pointers are encoded and the addresses are removed. We define a new variable to store the encoded value of a pointer p:

- p_tag: encoded value of the pointer. Its size is given by $\lceil \log_2(\text{size\_of\_point-to-set}) \rceil$.

A simple encoding technique is to look at all variables a pointer may point to in the program and associate a number with each of them. In the case of an assignment (p=q) or comparison (p==q), some circuit must convert the values of the pointers in the hardware implementation. This leads to the possibility of further optimization presented in Section 3.3.

Example 3. *In the previous example* (*q=*p+1), p *points to* a *and* b. *The value of* p *is encoded in one bit stored by* p_tag *and* a *(resp.* b*) is associated with* 0 *(resp.* 1*). The value of* q *is encoded as well. We end up with the following code for* *q=*p+1*:*

```
switch p_tag:
    case 0: star_p = a;
    case 1: star_p = b;
star_q = star_p + 1;
switch q_tag:
    case 0: table[q_index]=star_q;
    case 1: c = star_q;
```

The architecture generated from this code segment is presented in Figure 1. We can see that the load is implemented using a 2-input multiplexor controlled by p_tag.

Example 4. *consider the assignment of pointers* (p=q), *where* p *points to* a, b *or* c *and* q *points to* b *or* c. *In order to remove the pointers, we create* p_tag *and* q_tag. *For* p_tag *we associate the value* 0 *with* a, 1 *with* b *and* 2 *with* c. *For* q_tag, *we associate* 0 *with* b *and* 1 *with* c. *The following code is generated for* p=q*:*

```
switch q_tag:
    case 0: p_tag=1;
    case 1: p_tag=2;
```

Now if b (resp. c) was associated with the value 0 (resp. 1) of p_tag, p=q *would have been replaced by:*

```
p_tag=q_tag;
```

This shows that the logic generated for the assignment is directly related to the encoding of the pointers.

In this section, we have presented simple techniques to transform a C code with pointers into a code without pointers. *Loads* and *stores* are removed using the temporary variable star_p and case statements. The values of the pointers are then encoded. The encoding could also be flow-sensitive depending of the point-to-set at the current line in the program. For this purpose, an explicit static single-assignment (SSA) representation [11] of the aliasing information would be appropriate.

3. OPTIMIZATION

In the previous section, we have seen how pointers can be removed using the information of pointer analysis. Now, we will optimize the code for hardware synthesis. First we will present techniques to minimize the number of live-variables (i.e. the number of register used) before *loads* and *stores*. Then we will show an optimal encoding of the pointers value which reduces the amount of logic for comparisons and assignments.

3.1 Optimization of Loads

We are trying to reduce the number of live-variables before *loads*. By definition, a *load* may read any variable of the point-to-set. This implies that all of these variables are live before the load as well as the pointer. However, only one variable is accessed: the-variable the pointer points to. Then, for a pointer p, the only value a *load* really needs is star_p, the value p points to.

Example 5. *In Figure 2, the* load (out=*p) *where* p *points to* a, b, *or* c, *is replaced by an assignment from* star_p. *The number of live-variables before the load goes from 4* {a, b, c, p} *to 1*

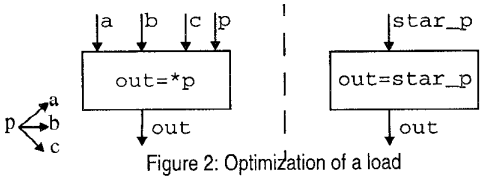

Figure 2: Optimization of a load

{star_p}, *assuming that none of these variables are live after the load.*

If we define star_p as early as possible in the program (i.e move the assignment to star_p up in the program), we can reduce the number of live-variable before the *load* by, at most, the number of variables in the point-to-set. In our implementation, we define star_p each time p or any variable in the point-to-set is modified. Then we use dead-code elimination to remove the useless assignments.

However, the early definition of star_p may increase the number of live-variables. When all variables of the point-to-set are live, star_p is just a copy of one of these variables and therefore is not necessary. So, in order to minimize the number of live-variables, star_p should be *killed* when all variables of the point-to-set are live. Here is the algorithm to optimize *loads*:

- Update star_p when p, or any variable of the point-to-set changes.
- Do live-variable analysis [10] (backward dataflow analysis).
- Insert definition of star_p when all variable of the point-to-set are live.
- Do dead-code elimination.

Example 6. *Let us take the following code segment, before and after optimization, where the pointer* p *points to* a, b *or* c.

```
/* original code */ |/*code after optimization */
  a=in;                    a=in;
                         | // if (p_tag==0) star_p=a;
  clk++;                   clk++;
                         | switch p_tag
                         |   case 0: star_p=a; break;
                         |   case 1: star_p=b; break;
  temp=a+b+c;              temp=a+b+c;
  clk++;                   clk++;
  out=*p+temp;           | out=star_p+temp;
```

We assume that none of the variables are live after the last line. During the first pass, we replace *p *by* star_p, *and update* star_p *after* a=in. *Then, because of* temp=a+b+c, a, b *and* c *are live when the clock (*clk*) is incremented the first time. After live-variable analysis we add the* case *statements which define (i.e. kill)* star_p. *Finally dead-code elimination will remove the assignment to* star_p *at the beginning of the code. The number of live-variables before the load has been reduced from 5* {a,b,c,p,temp} *to 2* {star_p, temp}.

This optimization can drastically decrease the number of live variables before *loads*. Nevertheless, it increases the number of branching statements which correspond to combinational steering logic to control the value of star_p. Therefore there is a trade-off here between the number of live-variables (i.e registers) and the amount of steering logic in the hardware implementation.

3.2 Optimization of Stores

For hardware synthesis, functions can either be inlined or implemented as components. When a function is inlined and one of its parameters passed by reference is both read and written, we end up with a *load* followed by a *store*. Here, the number of live variables between the *load* and the *store* can be reduced by one. The reason is that the *store* needs all variables of the point-to-set except the variable p points to. For this purpose, given a pointer p and the size of its point-to-set pts_size, we define the following class of variables:

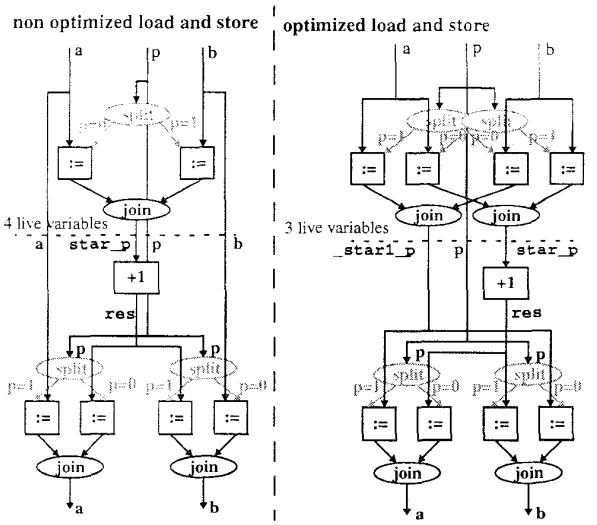

Figure 3: CDFG for *p=*p+1 with p->(a,b)

- _starN_p: (stands for "not star p") value of the set of variables of the point-to-set p does ***not*** point to, where N goes from 1 to (pts_size-1).

Remark that each _starN_p may only store the value of one of two variables.

Example 7. *If* p *may point to* a, b *or* c. *We define:*

```
- _star1_p=(p!=&a)?a:b;
- _star2_p=(p!=&b)?b:c;
```

Example 8. *Let us look at the example of* (*p=*p+1) *where* p *points to* a *or* b. *Figure 3 shows the control/data-flow graph (CDFG) before and after optimization using* _star1_p. *The code corresponding to* (*p=*p+1) *after optimization is the following:*

```
if(p==0) {
    star_p = a;
    _star1_p = b; }
else {
    star_p = b;
    _star1_p = a; }

star_p = star_p + 1;

if(p==0) {
    a = star_p;
    b = _star1_p; }
else
    b = star_p;
    a = _star1_p; }
```

The definition of the temporary variables has been inserted before the load, and the variables of the point-to-set are updated after the store. We can verify that the number of live-variables between the load *and* store *has been reduced from 4* {a,b,p,star_p} *to 3* {star_p,_star1_p,p}.

To perform this optimization, let us first consider an adaptation of the algorithm described in Section 3.1. Indeed, one could imagine an algorithm where the _starN_p variables are used at each store and defined when p or any variable of the point-to-set is modified. Since each _starN_p variable can only store the value of one of two variables of the point-to set, they should be *killed* each time one of the variables of the point-to-set is live. This cre-

ates a lot of logic to control their value, which turns out not to be very practical.

In our implementation, we focussed on the case of a *load* followed by a *store*. For a pointer p, the algorithm is the following:

- List the *stores* dominated by *loads* from the same pointer (forward dataflow analysis).
- List the *loads* post-dominated by *stores* from the same pointer (backward dataflow analysis).
- Do live-variable analysis assuming that the *stores* (*p=...), which are in the list, kill all variables in the point-to-set.
- If, for all the *loads* in the list, none of the variables in the point-to-set are live:
  - define star_p and the _starN_p variables before the *loads* and when p, or any variable of the point-to-set changes between *loads* and *stores*;
  - use star_p and the _starN_p variables to update the values of variables in the point-to-set after the *stores*.

Even though this optimization reduces the number of live variables before *stores* by at most one, it helps reducing the number of registers while calling functions. This optimization can be performed while optimizing the *loads*, as we will see in Section 4.

3.3 Encoding of Pointers

In software, the values of the pointers represent addresses in memory. These values can then be assigned (p=q) or compared (p==q). In hardware, as we have seen in Example 4, we have to add case statements to "translate" the values of the pointers by means of some combinational circuit. We can use encoding technique to minimize the size of this circuit.

First, we want to encode each tag with a minimum number of bits. Moreover, when a pointer is assigned or compared to another pointer, we would like the corresponding tags to be equal (e.g. p_tag=q_tag) or as close as possible to each other. If the tags have different number of bits, one tag can be equal to a subfield of the other.

The encoding problem can be formulated as follow. For each pointer we define a set of symbols corresponding to the variables the pointer may point to. As a result we have an ensemble of sets of symbols and the dependencies among the sets. The problem consists in encoding the symbols in the sets. The constraints on the encoding are two: 1) the supercube of the symbols in each set must have a minimum size. 2) the symbols that correspond to the same variable in two dependent sets must be encoded as close as possible. The reason for the first constraint is to minimize the number of bits to store while the reason for the second one is to reduce the combinational logic implementing the pointer assignment and comparison.

Example 9. *Consider 3 pointers, p, q, r and s. The dataflow is represented in Figure 4. The pointers are defined as follow:*

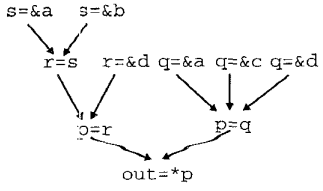

Figure 4: Datalow graph defining the pointers p, q, r and s

- s *is equal to* &a *or* &b;
- r *is equal to* s *(that points to* a *or* b*) or* &d;
- q *is equal to* &a, &c *or* &d;
- p *is equal to* r *(that points to* a, b *or* d*) or* q *(that points to* a, c *or* d*).*

*After encoding, the pointers are replaced by the tags (*p_tag, q_tag *and* r_tag*). We define the sets of symbols ({*q_a, q_b, q_d*}, {*r_a, r_b, r_d*}, {*s_a, s_b*} ...) where* q_a *is the value of* q_tag *when* q *points to* a*. We want to find an encoding for the symbols in each set.*

*In the example the 3 assignments (*p=q, p=r *and* r=s*) define the dependencies. To minimize the size of the supercubes, p_tag, q_tag and r_tag require 2 bits to encode 3 or 4 values and s_tag requires 1 bit to encode 2 values.*

If we assign s_a=0 *and* s_b=1 *then we can derive* r_a=00, r_b=01, r_c=11 *(or* 10*).*

We can encode the other pointers in a similar vein, as shown in Figure 5.

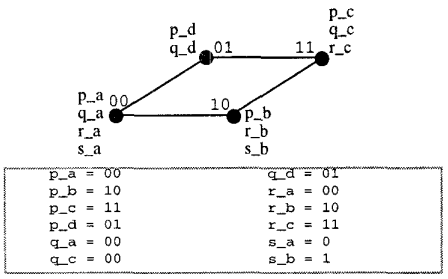

Figure 5: Encoding of pointer's values

With this encoding, no additional cost is incurred in translating the pointers values.

The encoding problem can be solved by a specialized algorithm or cast into an encoding problem for symbolic tables [3]. In particular NOVA [15] can be used to find the required encoding by constructing a symbolic table (to be interpreted by NOVA) which groups the symbols in each set and minimize the distance between the codes of the pointers that are assigned or compared to each other. Despite the fact that the pointer encoding problem differs from symbolic table encoding problem [3], the use of NOVA can be viewed as a heuristic to achieve optimal pointer encoding.

4. IMPLEMENTATION AND RESULTS

We have implemented the different algorithms using the SUIF environment [18]. The toolflow is presented on Figure 6. Our implementation takes a function with pointers in C and generates a module in Verilog. This module can then be synthesized using the Behavioral Compiler™ of Synopsys. For hardware synthesis, the timing information is expressed in the C model: clk++ in C will be translated into @(posedge clk) in Verilog. The ports and the data types are defined in a separate header file. The translation from C to Verilog consists of different passes. After the front-end, we inline the functions and perform the pointer analysis [16]. Then the aliasing information is used to remove and optimize pointers in the following order:

- define the point-to-set of each pointer;
- replace the *loads* and *stores* (insert star_p);
- optimize load 1: define star_p when p or any variable of the point-to-set change;

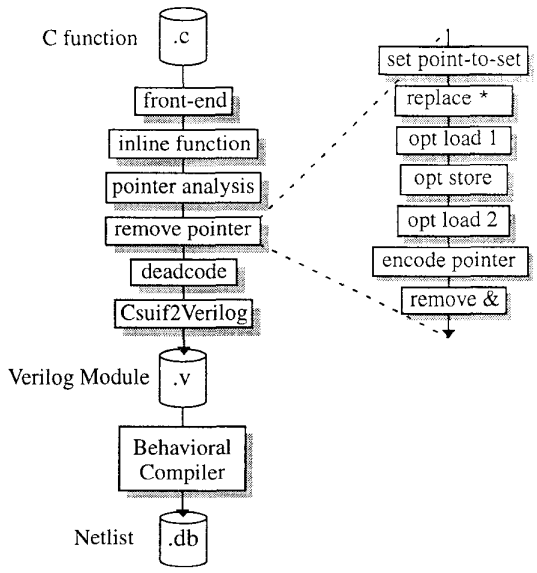

Figure 6: Toolflow for the Synthesis of Pointer in C

- optimize *loads* followed by *stores*: create the `_starN_p` variables;
- optimize load 2: kill `star_p` when all variables of the point-to-set are live;
- encode pointers value using NOVA [15];
- dead-code elimination.

The intermediate code without pointer is then translated into Verilog using Csuif2Verilog.

We have written several simple models to test the functionality and the efficiency of our implementation. Table 1 and Table 2 show the examples and the results after pointer resolution with and without optimization.

example	C lines	area (no optimization)		area (with optimization)	
		combinational	non-combinational	combinational	non-combinational
load	43	1527	2334	2076	1523
load/store	48	5319	1427	5324	1042
encoding	58	272	834	162	834

Table 1: Results after synthesis and optimization using target library lsi_10k: combinational area and non-combinational area in gates.

example	C lines	time (no optimization)	time (with optimization)
load	43	46 ns	51 ns
load/store	48	86 ns	88 ns
encoding	58	7.5 ns	5.9 ns

Table 2: Results after synthesis and optimization using target library lsi_10k: cumulative timing in ns.

Table 1 and Table 2 show the area and cumulative timing for the examples. They illustrate each feature of the optimizer. The first model tests the optimization of *loads*. It contains one pointer that may point to 3 integers stored in registers. After the definition of the pointer, we have two paths and then a *load*. In one path, none of the variables of the point-to-set are used. In the other path, all variables of the point-to-set become live. Without any optimization we have 5 32bit registers (i.e. 2300 gates of non-combinational area). After optimization we only have 3 registers (i.e. 1500 gates of non-combinational area). Notice the increase of the combinational area and of the cumulative time caused by adding steering logic to update the value of `star_p`. There is a trade-off between the number of registers and the size of the steering logic.

In the second example, we have a pointer that may point to two integer variables stored in registers. This pointer is used as a parameter in a function call. After inline the function, we end up with a *load* followed by a *store*. Here the optimization saves one register with a little increase of the amount of steering logic.

Finally, we implemented the model in Example 9. Here the encoding of the pointers value reduces the combinational logic by 40%. Since the design is simpler, the circuit is also faster.

Our implementation can also deal with larger designs and arrays. We have taken a regular implementation of a two-dimensional inverse discrete cosine transform (2D IDCT) [9] in C and synthesized it. The 2D IDCT is widely used in image compression standards such as JPEG, MPEG and H263. The 2D IDCT implemented consists of two one-dimensional IDCTs (1D IDCTs). For this purpose, we use 3 different memories: the input buffer (in_table), the intermediate buffer that stores the result of the first 1D IDCT (buf_table) and the output buffer (out_table). To access these memories, we use pointers and pointer arithmetic. In the 1D IDCT pointers are also used to reference two register banks (buff1 and buff2).

Pointers can also be used to optimized the circuit. In particular, the 2D IDCT can be implemented using only one call to 1D IDCT:

```
2d_idct() {
  int i, *p_in, *p_out;
  for(i=0; i<2; i++) {
  if(i==0) {
      p_in = in_table;
      p_out = buf_table;
  } else {
    p_in = buf_table;
     p_out = out_table;
  }
   1d_idct(p_in, p_out);
  }
}
```

Here pointers are not only used to access memories, but they are also used to specify resource sharing: in this case only one `1d_idct` is instantiated.

test	cpu time	nb ptr	C lines	Verilog lines	area (in gates)		cycles at 20MHz
					combinational	non-combin.	
idct	7.8s	6	176	221	38172	12910	1088

Table 3: Result of the synthesis of the IDCT running at 20MHz using target library LSI10k.

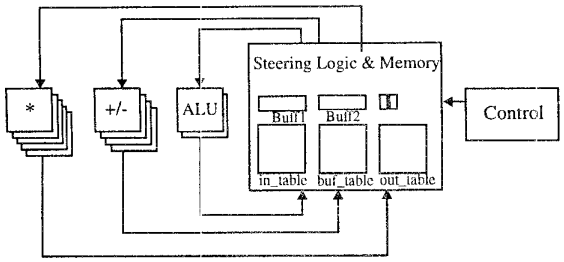

Figure 7: architecture of the 2D IDCT

The results of the synthesis is presented on Table 3. The CPU time for translating the C model into Verilog was calculated on SunUltra2. The Verilog module was synthesized with behavior compiler without unrolling loops. The architecture of the IDCT is presented in Figure 7. The design consists of 5 multipliers, 4 adders and 2 ALUs. Other implementations can be found by changing the timing and resource constrains.

5. CONCLUSION AND FUTURE WORK

We have presented an extension of the synthesizable subset of C to pointers to variables. Pointers resolution is not only used to synthesize C models with pointers, it also allows designers to further optimize their code with explicit resource sharing. Our implementation takes a C function with pointers and generates a Verilog module without pointers. The code of this module can then be synthesized by commercial tools such as the Behavioral Compiler™ of Synopsys.

In particular, we synthesize and optimize a C model with pointers to multiple variables. These variables can be mapped to registers, wires, ports or elements of different memories. Pointer arithmetic is also allowed for pointers to array elements. The logic is then optimized. In particular, we minimize the number of registers before *loads* and between *loads* and *stores*. The values of the pointers are also encoded in order to minimize both their size and the circuit generated for the assignments and comparisons of pointers.

In the future, we are planing to extend this work to pointers to pointers and dynamic memory allocation. We are also planning to work on pointers to functions. This could be used for object-oriented synthesis and reconfigurable hardware.

6. ACKNOWLEDGMENT

The work presented in this paper is supported by ARPA under contract DABT63-95-C-0049 and by Synopsys Inc. We would also like to thank David Heine from Stanford, for its help with the implementation using SUIF, Joachim Kunkel, Abhijit Ghosh from Synopsys Inc. for their comments and their support.

7. REFERENCES

[1] Ivo Bolsens, Hugo J. De Man, Bill Lin, Karl Van Rompaey, Steven Vercauteren, Diederik Verkest, "Hardware/Sofware Co-Design of Digital Telecommunication Systems", Proceedings of the IEEE, Vol 85, No. 3, pp.391-418, March 97.

[2] C.T.Bye, M.R. Lightner and D.L. Ravenscroft, *"A Functional Modeling and Simulation Environment based on ESIM and C"*, Proceeding of the 1984 ICCAD, pp.51-53, November 84.

[3] Giovanni De Micheli *"Synthesis and Optimization of Digital Circuits"*, Mc Graw Hill, Hightstown, NJ, 1994.

[4] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny, *"The COSYMA Environment for Hardware/Software Cosynthesis of Small Embedded Systems"* Microprocessors and Microsystems 20(3),pp.159-166, May 1996.

[5] Brian Kernighan, Dennis Ritchie, "The C Programming Language", Prentice Hall Software Series, Englewood Cliffs, NJ, 1988.

[6] David Knapp, *"Behavioral Synthesis: Digital System Design Using the Synopsys Design Compiler"*, Prentice Hall, Upper Saddle River, NJ, 1996.

[7] David Ku and Giovanni De Micheli, *"High-Level Synthesis of ASICs under Timing and Synchronization Constraints"*, Kluwer Academic Publishers, Boston, MA 1992.

[8] Stan Liao, Steve Tjiang, Rajesh Gupta, *"An Efficient Implementation of Reactivity for Modeling Hardware in the SCENIC Design Environment"*, Design Automation Conference DAC97, pp.70-75.

[9] Elliot Linzer, Ephraim Reig, *"New Scaled DCT Algorithms for Fused Multiply/Add Architectures"*, International Conference on Acoustics, Speech, and Signal Processing, Proceedings ICASSP '91, Vols.1-5, pp.2201-2204, 1991.

[10] Steven S. Muchnick *"Advanced Compiler Design & Implementation"*, Morgan Kaufmann Publishers, San Francisco, Ca, 1997.

[11] Cytron, Ron, and Reid Gershbein. *"Efficient Accomodation of May-Alias Information in SSA form"*, Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation, pp.36-45, June 1993.

[12] Donald Soderman, Yuri Panchul, *"Implementing C Designs in Hardware: A Full-Featured ANSI C to RTL Verilog Compiler in Action"*, http://www.compilogic.com/

[13] Bjarne Steensgaard *"Point-to Analysis by Type Inference of Programs with Structures and Unions"*, Proceedings of the 1996 International Conference on Compiler Construction, pp.136-150, April 1996.

[14] Charles Stoud, Ronald Munoz, David Pierce, *"Behavioral Model Synthesis with Cones"*, IEEE Design & Test of Computers, Vol 5 No3, pp.22-30, June 88.

[15] Tiziano Villa, Alberto Sangiovanni-Vincentelly, *"NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation"*, IEEE Transactions on Computer-Aided Design, Vol. 9, pp.905-924, September 1990.

[16] Robert Wilson, *"Efficient, Context-Sensitive Pointer Analysis For C Programs"*, PhD Dissertation, Stanford University, 1997.

[17] Robert Wilson, Monica Lam, *"Efficient Context-Sensitive Pointer Analysis for C Programs"*, Proceeding of the ACM SIGPLAN'95 Conference on Programming Languages Design and Implementation, pp.1-12, June 95.

[18] R.P.Wilson et al. *"Suif: An Infrastructure for Research on Parallelizing and Optimizing Compilers"*, ACM SIPLAN Notices 28(9), pp.67-70, Sept. 1994.