# Encoding of Pointers for Hardware Synthesis

Luc Séméria
lucs@azur.stanford.edu

Giovanni De Micheli
nanni@galileo.stanford.edu

Computer System Laboratory, Stanford University
Stanford, CA 94305

## ABSTRACT

*In the recent past, subsets of C and C++ have been defined to model and synthesize electronic systems as well as reusable IP blocks. In order to synthesize as much as possible of the C syntax, we have been researching the problem of synthesizing and optimizing code with pointers. In this paper, we focus on encoding the pointers' values to minimize the size of the circuits implementing the address translations. After defining the encoding problem, we present a solution based on heuristics and graph-embedding techniques. The algorithm has been integrated in our SpC flow which synthesizes C code with pointers.*

## 1. INTRODUCTION

### 1.1 Synthesis from C

For years, designers have been writing system-level models using programming languages, such as C and C++, to estimate the system performance and verify the functional correctness of the design. However, to implement parts of their design in hardware using synthesis tools, they must manually translate their code into a synthesizable subset of hardware description language (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 and IP components using C. C-based component modeling would improve design reuse and retargeting to hardware and software implementations.

However, the tasks of synthesizing hardware from C turns out to be particularly complicated because of dynamic memory allocation, function call, recursion, type casting and pointers. In our research, we have been focusing on the problem of synthesizing code with pointers.

### 1.2 Synthesis of Pointers

The problem of synthesizing pointers has been already introduced in [13] and [6]. The idea in [13] is to use pointer analysis to define the *point-to set* (sets of variables the pointer may point to) of each pointer in the program. Then the loads (...=*p) and stores (*p=...) can be replaced by case statements in which the variables of the point-to set are explicitly referenced. Furthermore, to reduce the size of the decoder associated with these case statements and to minimize the number of register used, the values of the pointers are encoded. For a pointer p, we call its encoded value its *tag*, noted p_tag.

***Example 1.*** *Let us consider a pointer* p *that may point to the variables* a *or* b *(result of the pointer analysis). Consider the following line of code which implements a* load*:*

```
out=*p;
```

*This load instruction can be replaced by a case statement:*

```
switch(p_tag) {
   case 0: out=a; break;
   case 1: out=b; break;
}
```

However, after encoding the pointers' values in the final hardware implementation, some combinational circuit may be needed to translate the values of the pointers involved in assignments or comparisons.

***Example 2.*** *Consider the assignment* p=q*, where the pointers* p *and* q *may point to* a *or* b*. For* p, a *(resp.* b*) is associated with the value 0 (resp. 1), whereas for* q, a *(resp.* b*) is associated with 1 (resp. 0).*

*After encoding, the assignment* p=q *is replaced by the following code segment:*

```
switch(q_tag) {
   case 0: p_tag = 1; break;
   case 1: p_tag = 0; break;
}
```

*In this case, a better encoding can be found by having the same encoding for both* p *and* q*. Then* p=q *would directly be replaced by* p_tag=q_tag*.*

To minimize the corresponding combinational circuit, the codes have to be equal or subfield of one another. This paper presents a solution for the encoding of pointers' values. In Section 2, we are going to formulate the problem. In order to reduce the complexity, we show in Section 3 how the encoding problem can be simplified and solved using graph-embedding techniques. In Sections 4 and 5, we present two optimization techniques called *folding* and *splitting*. The algorithm is then described in Section 6. Finally, in Section 7, we show how the encoding of pointers has been integrated in our tool for the synthesis of pointers in C (SpC) and present some results.

## 2. ENCODING OF POINTER

### 2.1 Definition of the Problem

***Definition 1.*** *We define a pointer-dependence graph as a graph in which the nodes are the pointers and the edges are the relations between the pointers. An edge between two nodes is defined when the two corresponding pointers are assigned or compared.*

***Example 3.*** *Consider the following code segment:*

```
int *r1, *r2, *r3, *q1, *q2;
...
if(i==0)
   { r1=&a;  r2=&b;  r3=&c; }
else
   { r1=&b;  r2=&c;  r3=&d; }

if(j==0)
   { q1=r1;  q2=r2; }
else
   { q1=r2;  q2=r3; }
...
```

*In this example, we consider the pointers {*r1, r2, r3, q1, q2*} and the variables {*a, b, c, d*}. The pointers are defined as follows:* r1 *may point to the variables* a *or* b, r2 *may point to the* b *or* c *and* r3 *may point to* c *or* d. *Then,* q1 *may take the value of* r1 *or* r2 *and* q2 *may take the value of* r2 *or* r3. *Consequently, q1 points to* a, b *or* c, *and q2 points to* b, c *or* d.

*This leads to the pointer-dependence graph on Figure 1.*

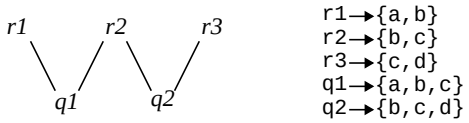


Figure 1: example of pointer-dependence graph

Our goal is to encode each pointer with the minimum number of bits. And, when a pointer is assigned or compared to another pointer, we want the corresponding tags to be equal (e.g. p_tag=q_tag) or as close as possible to each other. If two tags have different number of bits, one tag can be equal to a subfield of the other. Assignments would then be performed by concatenating or removing bits, whereas comparisons would only be executed on the bits common to the two codes. This reduces the size of the circuit that translates or compares the tags while keeping the number of bits to a minimum.

The encoding problem can be formulated as follows. 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 of encoding the symbols in the sets. The constraints on the encoding are two: 1) the supercube[1] of the symbols in each set must have 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 size of the combinational circuit implementing the pointer assignment and comparison.

***Example 4.*** *In Example 3, the pointers* r1, r2 *and* r3 *may point to two different variables and* q1, q2 *may point to three different variables.*

*We want to code* r1, r2 *and* r3 *on 1 bit and* q1, q2 *on 2 bits. Then we want the code of* r1 *and* r2 *to be subfields of the code of* q1 *and the code of* r2 *and* r3 *to be subfields of the code of* q2. *An encoding verifying these properties is shown on Figure 2.*

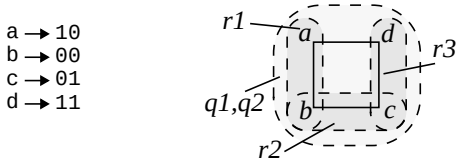


Figure 2: example of encoding

1. The *supercube* of a set of cubes is the smallest cube containing all the cubes in the set [8].

*For* r1, *value* 0 *is assigned to* b *and value* 1 *to* a. *For* r2, 1 *will be assigned to* c *and* 0 *to* b. *As a result,* q1=r1 *will be replaced by* q1_tag={r1_tag,0} *and* q1=r2 *will be replaced by* q1_tag={0,r2_tag} *(where* {,} *is the concatenation operator).*

## 2.2 Problem Formulation

Let's define $P$ pointers $\{p_1, p_2, \ldots, p_P\}$. We define the *pointer dependence relation* as follows. For two pointers, $p_i$ and $p_j$, $r(p_i, p_j) = 1$ if and only if the two pointers are assigned or compared, $r(p_i, p_j) = 0$ otherwise.

For each pointer $p_i$ we define its point-to set $\Pi_i$ to be the set of variables the pointer may point to. The point-to set $\Pi_i$ is a set of $N_i$ symbols $\{s_1^i, s_2^i, \ldots, s_{N_i}^i\}$, where each symbol is associated with a variable. After encoding, we define $E_i$, the set of the encoded symbols of the point-to set $\Pi_i$. The encoded values of the symbols in each set are noted $\{e_1^i, e_2^i, \ldots, e_{N_i}^i\}$.

***Definition 2.*** *Two sets* $\Pi_i$ *and* $\Pi_j$ *are said to be* dependent *if their associated pointers are assigned or compared (i.e.* $r(p_i, p_j) = 1$*).*

Our first goal is to minimize the number of registers as well as the size of the decoders required to store and decode the pointers values. We want the minimize the dimension of the supercube of the encoded symbols in each set. This minimum is achieved when the sum of the dimensions of the supercubes is also minimized:

$$min\left(\sum_{i=1}^{P} dim(supercube(E_i))\right) \quad (1)$$

When two pointers are assigned or compared, we also want to minimize the size of the circuit implementing the translation of the codes. For this purpose, the distance between encoded symbols in two dependent sets has to be minimum:

$$min\left(\sum_{i=1}^{P}\sum_{j=1}^{P} r(p_i, p_j) dist(E_i, E_j)\right) \quad (2)$$

where *dist( )* is the distance between the two encoded sets. When the pointers have the same point-to set, *dist( )* is defined as:

$$dist(E_i, E_j) = \sum_{k=1}^{N} H(e_k^i, e_k^j) \quad (3)$$

where N is the number of symbols in the point-to set and $H(e_k^i, e_k^j)$ is the Hamming distance between the codes of the symbols $s_k^i = s_k^j$.

In general, the sizes of the codes and the point-to sets may differ. The computation of the distance is then more complex: we have to consider the different subfields and the symbols associated with the same variables in the two sets.

Our goal is to minimize Eq. 1 and Eq. 2. There is a trade-off between the storage area (number of registers) and the amount of logic used to translate the codes. For example, one may optimize the size of the pointers keeping the amount of logic minimum by minimizing first Eq. 2 and then Eq. 1. In general, we can cast the problem as follows:

$$min\left(\beta\sum_{i=1}^{P} dim(sup.(E_i)) + \gamma\sum_{i=1}^{P}\sum_{j=1}^{P} r(p_i, p_j) dist(E_i, E_j)\right) \quad (4)$$

where $\beta$ and $\gamma$ are two coefficients.

Since this problem is computationally hard to solve using exact methods, we have been looking at heuristics for solving the encoding of pointers.

# 3. SIMPLIFIED PROBLEM

## 3.1 Formalism for a Global Solution

In the general formulation of the problem presented in Section 2, different codes may be associated with the symbols in each set. Therefore the encoding has to be found *locally*, in each set. The problem can be simplified by constraining all symbols associated with the same variable to share the same code. Eq. 2 is then irrelevant because the distance between the codes of the symbols that correspond to the same variable in the different point-to sets is null (i.e. $dist(E_i, E_j) = 0$ $\forall (i, j) \in \{1, 2, ..., P\}^2$). Our goal becomes to minimize Eq. 1 only. The encoding is then found *globally* for all the symbols which correspond to the same variable in the sets.

We are now considering the symbols (i.e. variables) in the union $\Pi$ of the point-to sets. These symbols will be denoted: $\Pi = \{s_1, s_2, \ldots, s_N\}$. The complexity is reduced: instead of dealing with O(*P*N)* symbols $\{s_j^i$ such that $i \in \{1, 2, ..., P\}$ and $j \in \{1, 2, ..., N_i\}\}$ we only deal with $N$ symbols $\{s_1, s_2, \ldots, s_N\}$, where $N$ is the number of variables.

We introduce now a formalism which has been used to solve other encoding problems [8], [7], [14].

***Definition 3.*** *The* relation matrix ***A*** *is defined as the matrix in which the rows represent the point-to sets and the columns the symbols. The entry $a_{i,j}$ of **A** is 1 if and only if the symbol $s_j$ is in the set $\Pi_i$.*

***Example 5.*** *Let's take the case of Example 3. We can construct the following relation matrix:*

$$\boldsymbol{A} = \begin{array}{c} \begin{matrix} a & b & c & d \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} r1 \\ r2 \\ r3 \\ q1 \\ q2 \end{matrix}$$

*For example, the first row of the matrix shows that* `r1` *may point to* `a` *or* `b`.

We are looking for an encoding matrix ***E***, that satisfies the encoding constraints represented by ***A***. The constraints expressed by the relation matrix ***A*** is the following. Each row in ***A*** corresponds to a point-to set. For each row $\alpha$ of ***A***, we want the supercubes of the rows of ***E*** corresponding to the 1s in $\alpha$ to have minimum size. This corresponds to the constraint expressed in Eq. 1.

This problem differs from the input encoding problem [8]. The *relation* matrix is not a binary constraint matrix, as defined for the general encoding problem [8], in the sense that we don't have the additional constraint that, for each row $\alpha$ of the constraint matrix, the supercube of the rows of ***E*** corresponding to the 1s in $\alpha$ does not intersect any of the rows of ***E*** corresponding to the 0s in $\alpha$. The 0s in the relation matrix can then be considered as *don't care* in the constraint matrix.

## 3.2 General Encoding Algorithm

The problem of input encoding has been extensively studied ([1] [4] [7] [9] [10] [11] [12] [14]). We use an approach reminiscent of MUSTANG [9] and POW3 [1].

***Definition 4.*** *An affinity graph is an undirected weighted graph in which the nodes are the symbols $\Pi = \{s_1, s_2, \ldots, s_N\}$ and the edges are the relations between the symbols in $\Pi$. The weight $w_{i,j}$ on the edge $\{s_i, s_j\}$ is defined as:*

$$w_{i,j} = \sum_{k=1}^{P} a_{k,i} \cdot a_{k,j} \cdot (1 + \lceil Log_2 N \rceil - \lceil Log_2 N_k \rceil) \quad (5)$$

*where $P$ is the number of pointers, $N$ is the total number of symbols, $N_k$ the number of symbols in the set $\Pi_k$, and $a_{i,j}$ is an element of the relation matrix.*

The weight $w_{i,j}$ in the affinity graph increases with the number of sets that contain both $s_i$ and $s_j$: when two variables are in many point-to sets, we want their code to be close. This is even more important for small point-to sets. For example, if we have $N_k = 2$ symbols in the point-to set $\Pi_k$, their code must be next to each other to minimize the dimension of the supercube of the encoded set $E_k$, whereas if we have $N_k = 10$ symbols in the point-to set $\Pi_k$, the Hamming distance between the encoding of the symbols in the point-to set can be as much as $\lceil Log_2(N_k) \rceil = 4$. Therefore, the weight $w_{i,j}$ is the sum of the contributions of the point-to sets that contain both $s_i$ and $s_j$, where the contribution of each point-to set $\Pi_k$ is $(1 + \lceil Log_2 N \rceil - \lceil Log_2 N_k \rceil)$.

The pointer encoding problem can be solved as an embedding of the affinity graph in the Boolean hyperspace as done in [10],[9],[1].

***Example 6.*** *The relation matrix presented in Example 5 can be used to generate the affinity graph on Figure 3.*

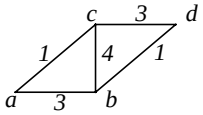


Figure 3: Example of Affinity Graph

*Let's look at $w_{a,b}$ the weight on the edge {*`a`,`b`*}. The variables* `a` *and* `b` *are both in the point-to sets of* `r1` *and* `q1`*. The weight $w_{a,b}$ is 3, sum of 2, contribution from r1, and 1, contribution from q1.*

*After graph-embedding, the encoding presented in Figure 2 can be found.*

These algorithms however do not consider the fact that two symbols can share the same code. We are going to use this property in Section 4 for a technique called *folding*. One symbol can also have multiple codes. The notion of *splitting* presented in Section 5 will be based on this property.

# 4. ENCODING WITH FOLDING

***Definition 5.*** *We define as* folding *the action of assigning the same code to two symbols.*

***Proposition 1.*** *Two symbols can be folded if and only if they are not both in the same point-to set and not in two dependent point-to sets.*

The rationale for this proposition is that we want to distinguish each symbol inside a point-to set and, in the case of a comparison, we want to distinguish the symbols in the two dependent point-to sets.

In the relation matrix ***A***, folding the symbols $s_i$ and $s_j$ is equivalent to replacing the columns *i* and *j* by one column *k* such that:

$$a_{k,l} = a_{i,l} \vee a_{j,l} \text{ for } l \text{ in } \{1, 2, ..., N\}. \quad (6)$$

In the affinity graph, folding is done by merging (or fusing[1]) the nodes corresponding to the symbols $s_i$, $s_j$ into one new node corresponding to $s_k$. The weights on the edges incident to this new node corresponding to $s_k$ are then defined as:

$$w_{k,l} = w_{i,l} + w_{j,l} \text{ for } l \text{ in } \{1, 2, ..., N\}. \quad (7)$$

Graph-embedding techniques can be modified to incorporate folding. In Section 6, we present a column-based method with folding.

***Example 7.*** *Let's consider the pointer-dependence graph on Figure 4 where* `r1`, `r2`, *and* `r3` *point respectively to {*`a,b,c`*}, {*`b,c,d`*} and {*`c,d,e`*}. The relation matrix and the associated affinity graph are represented in Figure 5.*

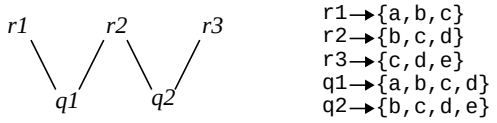


Figure 4: Pointer-dependence graph

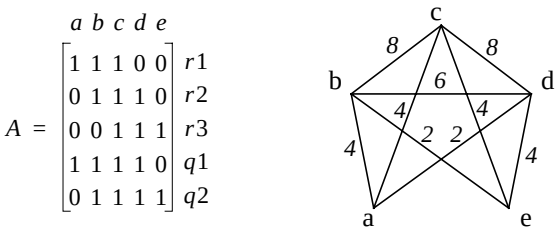


Figure 5: Encoding problem before folding

*The number of variables in each point-to set is either 3 (for* `r1`, `r2`, *and* `r3`*) or 4 (for* `q1` *and* `q2`*). Therefore, we want to code the symbols associated with the variables on 2 bits. However, since we have 5 symbols, an encoding with less than 3 bits cannot be found without folding.*

*The symbol* `a` *is in the point-to set of* `r1` *and* `q1`*, whereas the symbol* `e` *is the point-to set of* `r3` *and* `q2`*. According to the pointer-dependence graph, these point-to sets are not dependent. The symbols associated with* `a` *and* `e` *can be folded. After folding we end up with the graph on Figure 6.*

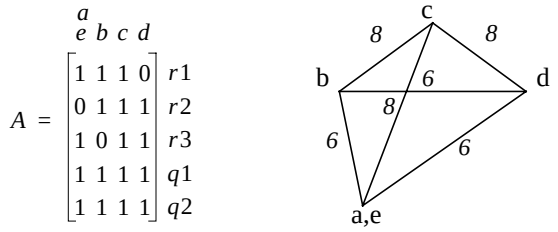


Figure 6: Encoding problem after folding `a` and `e`

*This leads to an encoding that requires only two bits:*

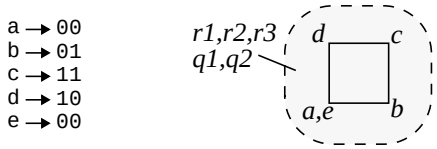


Figure 7: Result of the encoding after folding `a` and `e`

1. A pair of vertices `a`, `b` in a graph are said to be fused (merged or identified) if the two vertices are replaced by a single vertex such that every edge that was incident on either `a` or `b` or on both is incident on the new vertex [2].

# 5. SPLITTING TECHNIQUE

***Definition 6.*** *We define as splitting the action of assigning two or more codes to one variable (or symbol).*

In Section 3 and 4, each variable was associated with a unique symbol that was encoded. After splitting, one variable may be associated with more than one symbol: splitting a symbol $s_i$ is equivalent to creating a new symbol $s_i'$ which corresponds to the same variable. The original symbol $s_i$ and the newly created $s_i'$ are then encoded into $e_i$ and $e_i'$ respectively.

***Proposition 2.*** *A point-to set $\Pi_k$ that contains a symbol $s_i$ may, after splitting $s_i$, contain the newly created symbol $s_i'$ if and only if there is no code equal to $e_i'$ in the encoded set $E_k$ or in any encoded set depended of $\Pi_k$.*

As described in Section 2.2 the symbols in each set can have different codes. Therefore, to minimize the dimension of the supercube of the encoded symbols in a set (i.e. Eq. 1), we can create new symbols associated with the same variables, for this set. Note that, if we split the symbols for each point-to set, we end up with a local encoding scheme as presented in Section 2.2. However, to limit the increase in complexity, we are trying to split as few symbols as possible.

In the relation matrix ***A***, splitting is done by adding a column $i'$ relative to $s_i'$. For each row $\alpha_k$ relative to the point-to set $\Pi_k$, the entry $a_k^{i'}$ can be set to 1 if Proposition 2 is verified. The point-to set $\Pi_k$ may then contain $s_i$, $s_i'$ or both $s_i$ and $s_i'$.

In order to minimize Eq. 1, for set that may contain $s_i$ or $s_i'$, the following expression may be considered:

$$\min_{E'}(dim(supercube((E_k - \{e_i\}) \cup E'))) \quad (8)$$

where $E'$ is either $\{e_i\}$, $\{e_i'\}$ or $\{e_i, e_i'\}$.

For example, if Eq. 8 is minimum for $E' = \{e_i'\}$, the dimension of the supercube of the encoded symbols in the point-to set $\Pi_k$ is minimum when $a_k^i$ is set to 0 and $a_k^{i'}$ is set to 1.

The new affinity graph can then be recomputed from the relation matrix. Splitting as well as folding can be incorporated in our graph-embedding algorithm as presented in Section 6.

***Example 8.*** *Let's consider the pointer-dependence graph on Figure 8 where* `r1`, `r2` *and* `r3` *may respectively point to* `{a,b}`, `{b,c}` *and* `{a,c}`.

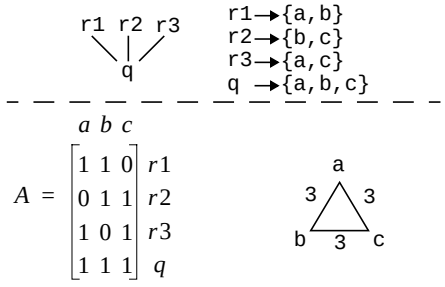


Figure 8: encoding problem before splitting

*We would like to encode* `r1`, `r2` *and* `r3` *with 1 bit and* `q` *with 2 bits. We also want the codes of* `r1`, `r2` *and* `r3` *to be subfields of the code of* `q`.

*Using the encoding technique without splitting symbols, we can find the encoding on Figure 9.*

*In this case* `r1` *and* `r2` *are encoded on 1 bit but the encoding of* `r3` *requires 2 bits.*

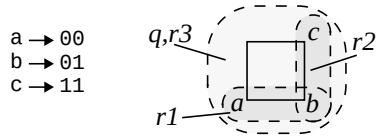


Figure 9: encoding without splitting

*After splitting the symbol* a*, we end up with the two symbols* a *and* a′*. We can find the encoding on Figure 10 where the symbol* a *is in the point-to set of* r1, r2 *and* q *and* a′ *in the point-to set of* r3 *and* q.

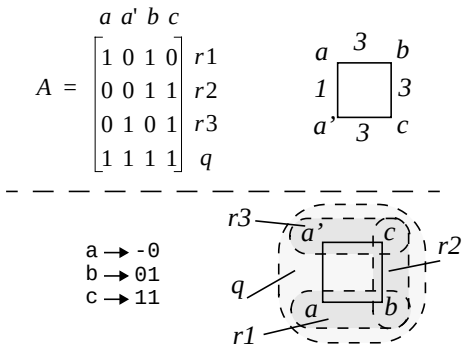


Figure 10: result after splitting symbol a

*The encoding in Figure 10 is optimal: r1, r2 and r3 are encoded on 1 bit and the assignments to* q *(*q=r1, q=r2, q=r3*) don't require any additional logic.*

# 6. ALGORITHM

We propose a column-based approach (which means that the encoding matrix can be found column by column [3]). Our algorithm without folding and splitting is similar to the one used in [1]. A pseudo code of the algorithm with folding and splitting is presented on Figure 11.

We consider one bit of the code at the time. For a symbol $s_i$ associated with the code $e_i$, we consider the bits $e_i^k$ for $k$={1, 2, ..., $n$} where $n$ is typically $n = \lceil \log_2(N) \rceil$. At each iteration $k$, we construct the k[th] column of the encoding matrix ***E***: the value of $e_i^k$ (0 or 1) is assigned in a way that minimizes the distance between the encoded symbol $e_i$ and its neighbors in the affinity graph.

The assignment is done for the symbols on every edge starting with the edges with highest weights. For the symbols incident to the edges $\{s_i, s_j\}$, we are trying to assign the same value to both $e_i^k$ and $e_j^k$. However, if the symbols $s_i$ and $s_j$ are also incident to other edges whose weights are higher than $w_{i,j}$, the values of $e_i^k$ and $e_j^k$ may already be assigned to different values. Moreover, at each iteration of $k$, the number of symbols having the same code is also limited.

***Definition 7.*** *An* edge *$\{s_i, s_j\}$ is said to be* violated *at iteration k if the values $e_i^k$ and $e_j^k$ associated with the two symbols incident to the edge, have different values.*

***Definition 8.*** *A* class violation *is defined at iteration k, when more than $2^{n-k}$ symbols have the same code so far. (At iteration k, we are only considering the k first bits of the codes, since the other ones haven't been assigned yet).*

The rationale for defining class violation is that we ultimately want to distinguish all the symbols. Therefore, in our greedy algorithm, we have to make sure that, at each iteration of $k$, we have less than $2^{n-k}$ symbols associated with the same code. For example for $k$=($n$-1), we cannot have more than 2 symbols with the same code.

***Proposition 3.*** *An edge $\{s_i, s_j\}$ is violated at iteration k if either one of the following conditions applies:*

- *there is class violation (and therefore, $e_i^k$ and $e_j^k$ need to have different values),*
- *the values $e_i^k$ and $e_j^k$ associated with the two symbols incident to the edge have already been assigned to different values (by Definition 7).*

In the case of a class violation, we try to fold one of the symbols on the edge $\{s_i, s_j\}$ with any of the previously assigned symbols. At this stage, two symbols are folded if Proposition 1 is verified and if they have the same partial code so far.

In the case when different values have been assigned to the two symbols $s_i$, $s_j$ (i.e. $e_i^k \neq e_j^k$), we try to split the symbols on the edge. One symbol can be split if the newly created symbol does cause any class violation or can be folded with another symbol. In our algorithm, for a symbol $s_i$, we create a new symbol $s_i'$ associated with a code $e_i'$ such that $e'^l_i = e_i^l$ for l>k and $e'^k_i = e_j^k$. In case of a class violation, we try to fold this new symbol as done previously. If folding cannot be done, the symbol is not split.

```
encode pointer() {
 /* construct matrix E one column at a time */
 for k=1 to n
    assign_code(k);
}

assign_code(k) {
 sort edges by weight in decreasing order;
 foreach edge {s_i, s_j} {
    if(e_i^k and e_j^k not assigned) {
       e_i^k = e_j^k = select_bit(s_i, s_j);
       if(class violation) {
          try_fold(s_i);   try_fold(s_j); }
    }
    if(s_i or s_j not assigned) {
       s_h=unassigned(s_i, s_j); s_l=assigned(s_i, s_j);
       e_h^k = e_l^k;
       if(no class violation) {
          try_fold(s_i);   try_fold(s_j); }
    }
    else /* s_i and s_j assigned */
       if(e_h^k != e_l^k)
          violated_edges->add({s_i, s_j})
 }

 sort violated edges by weight in decreasing order;
 foreach violated edge {s_i, s_j} {
    try_split(s_i);   try_split(s_j);
 }
}

try_split(s_i) {
    create s_i'
    e_i'=e_i xor(1<<k);
    if(class violation)
       try_fold(s_i');
 }

try_fold(s_i) {
    if(∃s_j s.t. Proposition 1 verified and e_i==e_j){
       fold(s_i, s_j);
       remove s_i;
 }
```

Figure 11: algorithm with splitting and folding

***Example 9.*** *Consider the problem presented on Figure 12. The associated relation matrix and affinity graph are presented on Figure 13.*

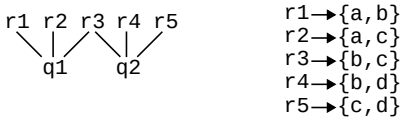


Figure 12: encoding problem

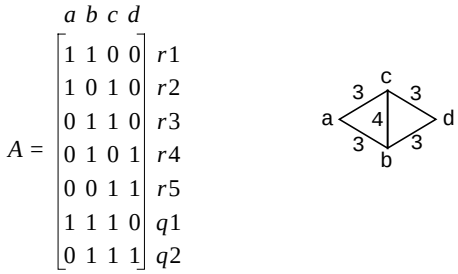


Figure 13: relation matrix and affinity graph

*Since we have 4 symbols, we want to encode them on n=2 bits.*

*At iteration k*=1, *we assign the value* 0 *to* b,c *and* 1 *to* a,d. *The violated edges are {*a,b*}, {*a,c*}, {*d,c*} and {*d,b*} but folding and splitting cannot be done. For example, for the edge {*a,b*},* a *cannot be folded with* b *(resp.* c*) because both symbols are in the point-to set of* r1 *and* q1*(resp.* r2 *and* q1*).*

*At iteration k*=2, *we assign* 00 *to* b, 01 *to* c, 10 *to* a *and* 11 *to* d. *All the edges are violated and some symbols can be split. The variable* a *can be split and the new symbol* a′ *can be folded with* d. *The variable* d *can also be split and the new symbol* d′ *can be folded with* a.

*Finally, we end up with the encoding on Figure 14 in which all the constraints are verified.*

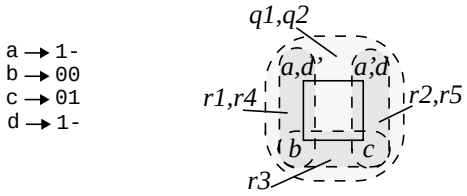


Figure 14: encoding after splitting and folding

## 7. IMPLEMENTATION IN SPC

In [13], we presented SpC a solution for the synthesis of pointers in C. The toolflow is presented on Figure 15. 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. The translation from C to Verilog consists of different passes. After the front-end, we inline the functions and perform the pointer analysis. 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*;
- optimize *loads* and *stores*;
- encode pointers;
- dead-code elimination.

The intermediate code without pointers is then translated into Verilog.

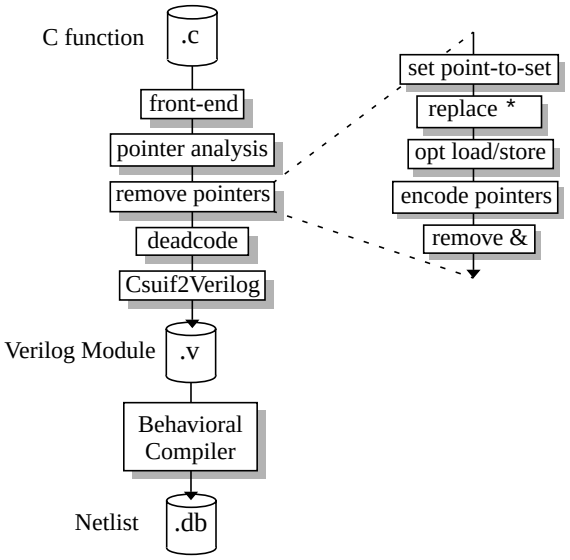


Figure 15: Toolflow for the Synthesis of Pointers in C

Today, SpC supports only pointers to variables and array elements. Nevertheless, our algorithm for pointer encoding could be used for pointers to pointers, pointers to functions, and recursive data structures as well.

The heuristic algorithm described in Section 6 has been implemented and applied to some test cases. The results are illustrated in Table 1 and have been obtained as follows. The register file as well as the logic necessary to translate the values of the pointers has been synthesized using Synopsys Design Compiler™ and the tsmc.35 library. We present the results for three different schemes.

First we present the results for a straightforward minimum-length encoding of the symbols. In this suboptimal encoding, each variable in each point-to set is simply associated with a number (0 for the first variable, 1 for the second variable, etc...). The number of bits used to encode each tag is then minimum but the size of the circuit which translates the values of the pointers is not.

The second scheme is the implementation of the algorithm without splitting and folding. The size of the circuit translating the values of the pointers is then reduced. However, the number of bits used for the encoding is not always minimal, which leads to larger decoding circuits (combinational area) and more registers (non-combinational area).

Finally, the last column shows the results for the algorithm with folding and splitting. The length of the codes is then close from the minimum and the size of the combinational circuit is reduced, which gives better results.

| example | P | N | min. length | | simple algorithm | | spit and fold | |
|---|---|---|---|---|---|---|---|---|
| | | | combin. | non-c. | combin. | non-c. | combin. | non-c. |
| test1 | 5 | 5 | 1215 | 2756 | 1211 | 3307 | 1069 | 2756 |
| test2 | 7 | 4 | 1119 | 2447 | 1346 | 2723 | 1006 | 2447 |
| test3 | 9 | 7 | 3747 | 4960 | 3666 | 5236 | 3325 | 4960 |

Table 1: results after synthesis and optimization using tsmc.35 library: combinational area and non-combinational area in library units.

## 8. CONCLUSION

We have presented the problem of encoding pointers for the synthesis of hardware. After simplification, the problem can be solved using graph-embedding techniques. These techniques can be further optimized: two symbols can share the same code (*folding*) and one symbol can have multiple codes (*splitting*). The algorithm presented has been successfully implemented for the synthesis of C code with pointers. Our technique could also be applied to other languages and object-oriented programs.

## 9. ACKNOWLEDGMENTS

This research is supported by DARPA under grant DATB63-95-C0049 and by Synopsys Inc. We would also like to thank Maurizio Damiani for its precious comments and suggestions.

## 10.REFERENCES

[1] L.Benini and G.De Micheli, *"State assignment for low power dissipation"* Custom Integrated Circuits Conference, 1994, pp. 136-139.

[2] Narsingh Deo *"Graph Theory with applications to Engineering and Computer Science"*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[3] T.A Dolotta and E.J. McCluskey. *"The Encoding of Internal States of Sequential Machines."* in IEEE Transactions on Electronic Computers, volume EC-13, pp. 549-562, October 1964.

[4] C. Duff, *"Codage d'automates et théorie des cubes intersectants,"* These, Institut National Polytechnique de Grenoble, France, March 1991.

[5] Evgueii Goldberg, Tiziano Villa, Robert Brayton, Alberto Sangiovanni-Vincentelli, *"Theory and Algorithms for Face Hypercube Embedding"* Transaction on CAD, volume 17(6), June 1998.

[6] Hwayong Kim, Kiyoung Choi, *"Transformation from C to Synthesizable VHDL"* in Proceedings of Asia Pacific Conf. on HDL APCHDL'98, July 1998.

[7] Giovanni De Micheli. *"Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-Level Logic Macros."* IEEE Transaction on CAD, volume 5(4), pp. 597-616, 1986.

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

[9] A. R. Newton, S. Devaras, H-keung Ma and A. Sangiovanni-Vincentelli. *"MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations."* IEEE Transaction on CAD, volume 7(12), pp. 1290-1300, 1988.

[10] G. Saucier, *"State Assignment of Asynchronous Sequential Machines using Graph Techniques"*, IEEE Transaction on Computer, March 1972.

[11] G. Saucier, C. Duff, F. Poirot, *"State assignment using a new embedding method based on intersecting cube theory"*," in Proceeding of Design Automation Conf., pp. 321-326, June 1989.

[12] G. Saucier, M.C. Depaulet and P. Sicard, *"ASYL: A rule-based system for controller synthesis"*, IEEE Transactions on CAD, volume CAD-6, pp. 1088-1097, November 1987.

[13] Luc Séméria, Giovanni De Micheli, *"SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C"*, proceeding of the 1998 ICCAD, pp. 340-346, November 1998.

[14] 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.