Finding heap-bounds for hardware synthesis
Coding hardware in advanced languages

- Use of advanced languages simplifies development process
- Advanced data structures are easy to use (lists, tree etc.)
- Advanced languages dynamically allocate memory
Problem with dynamic allocation

- Hardware has limited amount of memory
- Unbounded heap usage prevents compilation into hardware
Generic heap-bound

- In parameterized design, program has two kinds of input
  - Generic inputs
  - Input signals

- **Generic heap-bound** is a function over generic inputs that bounds heap usage

  Generic inputs are set to a constant during synthesis

  If generic heap-bound exists then heap is bounded by a constant during synthesis
Can we infer generic heap bound at compile time?
Outline

• An example

• Our solution

• Experimental results & conclusion
Example: priority queue

- Reads infinite series of integers at input channel \( i \)
- Sort the inputs in batches of \( n \)
- Push out sorted batch at output channel \( o \)
Example: priority queue

Linked list \( b \) is initialized to be empty

Phase 1: \( n \) times

- Sorted insert new element

Phase 2: \( n \) times

- Output head of linked list and delete it

Input channel \( i \)

Output channel \( o \)

Infinite loop
Example: priority queue

```c
prio(int n, in_sig i, out_sig o){
    Link *b, *c, *tmp;
    assume(n > 0);
    while(1) {
        b = NULL;
        for(int k=0; k<n; k++) {
            b = sorted_insert(in(i), b);
        }
        c = b;
        while(c != NULL) {
            out(o, c->data);
            tmp = c;
            c = c->next;
            free(tmp);
        }
    }
}
```

- Generic input
- Input signals

There is a single call to `alloc` in `sorted_insert`

Can we infer a generic heap bound for this program?
Outline

• An example

• Our solution

• Experimental results & conclusion
Inferring *generic heap-bound* & compiling

• The following 3 steps can do the job
  1. Track heap usage at each possible run of program
  2. Estimate maximum heap usage over all runs
  3. Translate to non-dynamic allocating heap program

💡 We supply the following **solution** for above steps
  1. Numerical heap abstraction (shape analysis)
  2. Numerical bound analysis (invariant generation)
  3. Array based heap management
Finding heap-bounds for hardware synthesis

Numerical heap abstraction
- Generates an abstract program that tracks heap usage (shape analysis)

Numerical bound analysis
- Computes generic heap bounds for the abstract program (Invariant generation)

Array based heap management
- Translates to non-dynamic allocation program
Finding heap-bounds for hardware synthesis

- Numerical heap abstraction
  - Generates an abstract program that tracks heap usage (shape analysis)

- Numerical bound analysis
  - Computes generic heap bounds for the abstract program (Invariant generation)

- Array based heap management
  - Translates to non-dynamic allocation program
Numerical heap abstraction

Input program is translated into an abstract numerical program using shape analysis

- Each data structure is replaced by a set of integers
- Actions on data structures are replaced by actions on the integers
- A new variable is introduced to represent heap usage
Example: numerical heap abstraction

- shape analysis recognizes \( c \) as a pointer to a linked list
- An integer \( k_c \) is introduced to represent length of the linked list \( c \)
- An integer \( h \) is introduced to represent the amount of heap used

```c
while( c != NULL ){
    out( o, c->data );
    tmp = c;
    c = c->next;
    free(tmp);
}
```

```c
while( k_c >= 0 ){
    skip;
    skip;
    k_c = k_c - 1;
    h = h - 1;
}
```
Abstract numerical program

- Abstract numerical program consists of
  - variables
  - control locations
  - transition relations between locations
  - generic inputs
  - a variable to represent heap usage
Abstract numerical program: priority queue

$\rightarrow$ $k_b$ is a new variable which represents length of linked list $b$
Finding heap-bounds for hardware synthesis

Generates an abstract program that tracks heap usage (shape analysis)

Computes generic heap bounds for the abstract program (Invariant generation)

Heap-bound == generic heap-bound

Translates to non-dynamic allocation program
Numerical bound analysis

- For each location \( p \), we find a heap bound \( B_{nd_p} \) such that \( h \leq B_{nd_p}(\text{generic inputs}) \)

- Example:
Heap-bound from Invariant

- **Invariant** is an assertion that is true at all reachable states

- **Invariant** may imply a heap-bound that bounds heap usage

"We solve following problem:

- For each location $p$, find an invariant $\text{Inv}_p$ such that for some heap-bound $\text{Bnd}_p$

  \[ \text{Inv}_p \rightarrow h \leq \text{Bnd}_p(n) \]

We extend constraint solving method for invariants
Heap-bounds via constraint solving

- A template is substituted for each invariant $\text{Inv}_p$ and heap-bound $\text{Bnd}_p$
  - Template = parameterized assertion over program variables

- Build constraints using numerical program and these templates

- Solve the constraints and get the heap-bounds
Template for invariant

• Template for invariant = parameterized assertion over program variables

• Example: template for invariant
  \[ a + a_n \cdot n + a_h \cdot h + a_k \cdot k + a_c \cdot k_b + a_c \cdot k_c \leq 0 \]
  – a, a_n, a_h, a_k, a_b and a_c are parameters
  – n, h, k, k_b, and k_c are program variables

• A template specifies a space of assertions
• We search for an invariant in this space
Template for heap-bounds

• Template for heap-bound = parameterized function over generic inputs

• Example: template for heap-bound
  \[ b_n \cdot n + b \]
  - \( b_n \) and \( b \) are parameters
  - \( n \) is generic input
Template Maps

- **Inv** = map from location to templates for invariants
- **Bnd** = map from location to templates for heap-bounds

Example:

**Inv**$_4$: \(a + a_n \times n + \ldots + a_c \times k_c \leq 0\)

**Bnd**$_4$: \(b_n \times n + b\)
Constraints

Build constraints that encode two conditions:

1. Inductive argument
   - If program state is in invariant and program runs then state remains in invariant

2. Invariant implies heap-bound that bounds heap usage

Example:
- For transition location 7 to 13
  \[ \text{Inv}_7 \land \text{trans}(7,13) \rightarrow \text{Inv'}_{13} \]
- For bound at location 7
  \[ \text{Inv}_7 \rightarrow h \leq \text{Bnd}_7 \]
Solving and getting heap-bound

- The built constraints are solved over the parameters
- Placing the solution of parameters in templates produces the heap-bounds

Example:
- Template for bound at location 7
  \[ \text{Bnd}_7: \ b_n \times n + b \]
- Solution of parameters, \( b_n = 1 \) and \( b = 0 \)
  \[ \text{Bnd}_7: \ 1 \times n + 0 \]
Finding heap-bounds for hardware synthesis

- Numerical heap abstraction
  - Generates an abstract program that tracks heap usage (shape analysis)

- Numerical bound analysis
  - Computes generic heap bounds for the abstract program (Invariant generation)

- Array based heap management
  - Translates to non-dynamic allocation program
Array based heap management

• Following are introduced in the input program
  – an array h of size the heap-bound and initialize it: \( \forall i. \ h[i]=i+1 \)
  – a variable m and initialize it with 0

• m will act as a head of linked list of available cells

• Example translation
  – \( x = \text{alloc}(); \) \( \Rightarrow \) \( x=m; \) \( m=h[m]; \)
  – \( \text{free}(x); \) \( \Rightarrow \) \( m=h[x]; \) \( m=x; \)
Finding heap-bounds for hardware synthesis

- Numerical heap abstraction
  Generates an abstract program that tracks heap usage (shape analysis)

- Numerical bound analysis
  Computes generic heap bounds for the abstract program (Invariant generation)

- Array based heap management
  Translates to non-dynamic allocation program
Outline

• An example

• Our solution

• Experimental results & conclusion
We have **developed** a tool-chain from C to gates

1. Numerical heap abstraction (shape analysis)
   - THOR

2. Numerical bound analysis (invariant generation)
   - ARMC and InvGen

3. Array based heap management
## Experimental results

### Computed bounds

<table>
<thead>
<tr>
<th>Program</th>
<th>Bound</th>
<th>C LOC</th>
<th>VHDL LOC</th>
<th>Bound synthesis time</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>$8 \times n_1 + 8 \times n_2$</td>
<td>80</td>
<td>1927</td>
<td>600m</td>
</tr>
<tr>
<td>prio</td>
<td>$8 \times n$</td>
<td>56</td>
<td>1475</td>
<td>4s</td>
</tr>
<tr>
<td>packet</td>
<td>$12 \times n + 8$</td>
<td>95</td>
<td>2430</td>
<td>6s</td>
</tr>
<tr>
<td>bst_dict</td>
<td>$24 \times n_1$</td>
<td>142</td>
<td>2703</td>
<td>80s</td>
</tr>
</tbody>
</table>

### Synthesis and implementation results

<table>
<thead>
<tr>
<th>Program</th>
<th>ALUTs</th>
<th>Registers</th>
<th>Block Mem</th>
<th>Blocks</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>5,157</td>
<td>4,694</td>
<td>8,192</td>
<td>2</td>
<td>90MHz</td>
</tr>
<tr>
<td>prio</td>
<td>5,859</td>
<td>4,598</td>
<td>4,096</td>
<td>1</td>
<td>83MHz</td>
</tr>
<tr>
<td>packet</td>
<td>9,413</td>
<td>9,158</td>
<td>8,192</td>
<td>2</td>
<td>76MHz</td>
</tr>
<tr>
<td>bst_dict</td>
<td>5,786</td>
<td>5,660</td>
<td>8,192</td>
<td>2</td>
<td>125MHz</td>
</tr>
</tbody>
</table>
Conclusion

• A new method to compute heap-bounds using
  – shape analysis
  – invariant generation (constraint solving)

• An attempt to bring the following together
  – agility of software development and
  – speed of raw gates

Thank you!