Finding heap-bounds for hardware synthesis

B. Cook+
A. Gupta#
S. Magill*
A. Rybalchenko#
J. Simsa*
S. Singh+
V. Vafeiadis+

*CMU
#MPI-SWS
+MSR

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

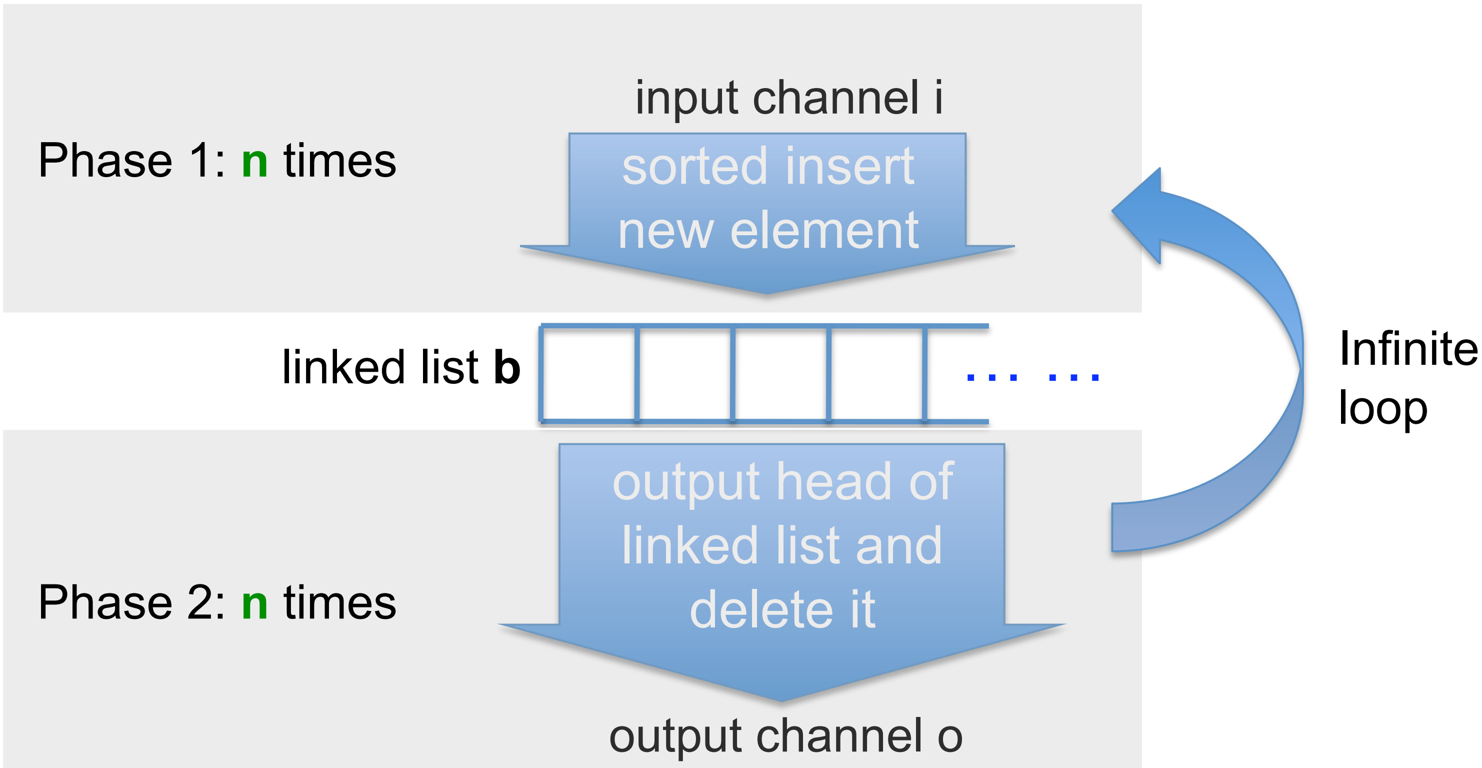
- **An example**
- Our solution
- Experimental results & conclusion

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



Example: priority queue

```
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);
       }
     }
}
```

Infinite Loop

allocation loop

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

de-allocation loop

- Generic input
- Input signals

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)**

↓

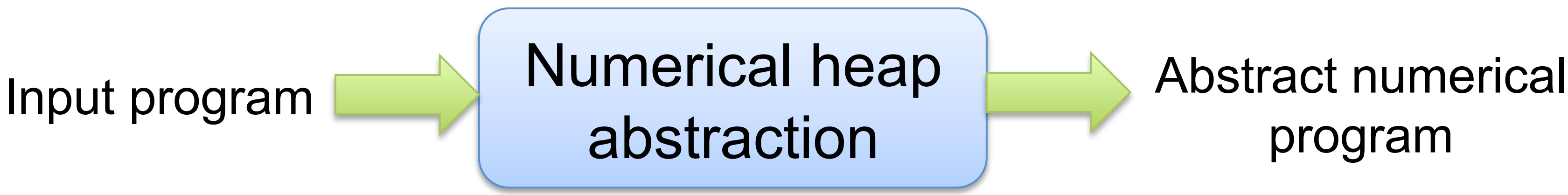
Numerical bound analysis — Computes generic heap bounds for the abstract program (Invariant generation)

↓

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

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

Input program

```
while( k_c >= 0 ){
   skip;
   skip;
   k_c = k_c - 1;
   h = h – 1;
}
```

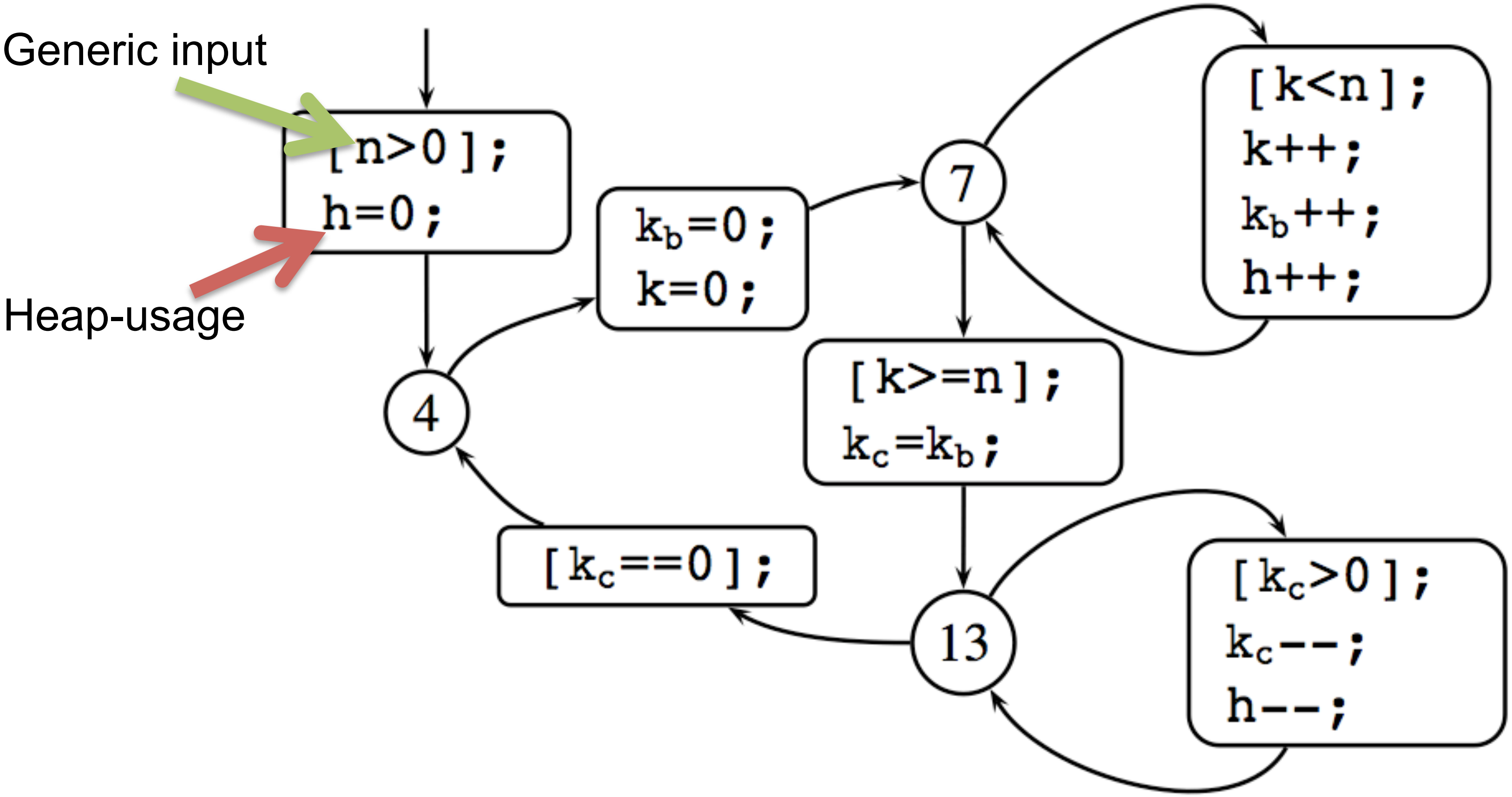
Abstract numerical program

- shape analysis recognizes **c** as a pointer to a linked list
- An integer $\mathbf{k_c}$ is introduced to represent length of the linked list **c**
- An integer **h** is introduced to represent the amount of heap used

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



➔ k_b is a new variable which represents length of linked list **b**

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)

Heap-bound == generic heap-bound

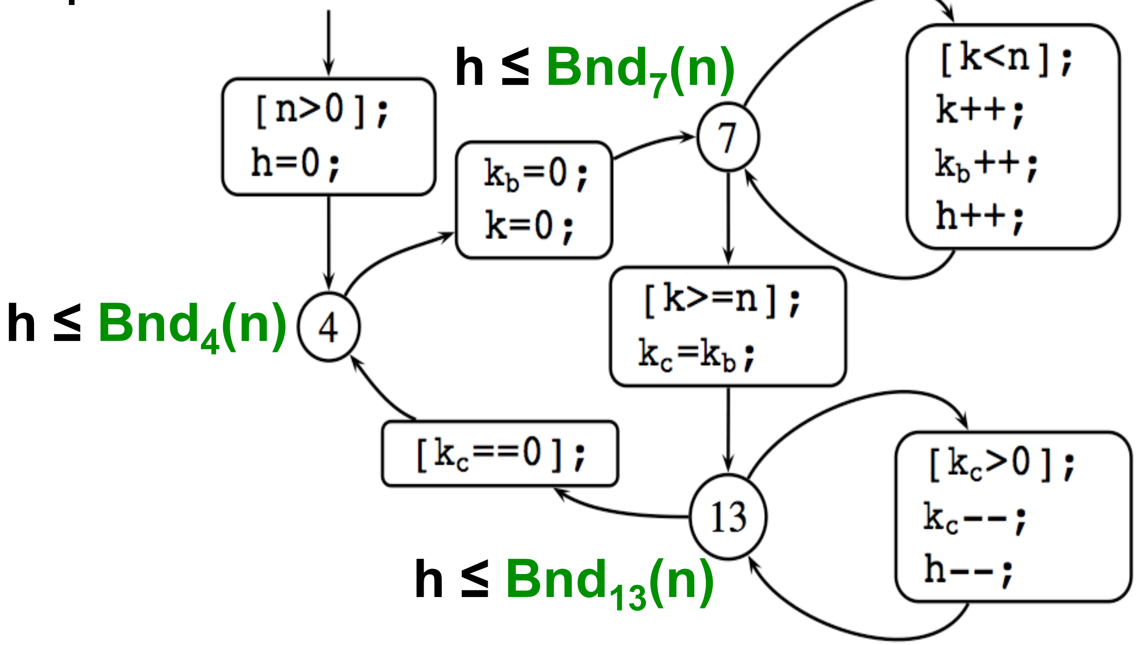
Array based heap management — Translates to non-dynamic allocation program

Numerical bound analysis

- For each location *p*, we find a heap bound **Bnd$_p$** such that

  h ≤ **Bnd$_p$(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 $\mathbf{Inv_p}$ such that for some heap-bound $\mathbf{Bnd_p}$

$$\mathbf{Inv_p} \rightarrow \mathbf{h} \leq \mathbf{Bnd_p(n)}$$

We extend constraint solving method for invariants

Heap-bounds via constraint solving

- A template is substituted for each invariant **Inv_p** and heap-bound **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

$$a+a_n*n+a_h*h+a_k*k+a_c*k_b+a_c*k_c \leq 0$$

  - 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

$$\mathbf{b_n * n + b}$$

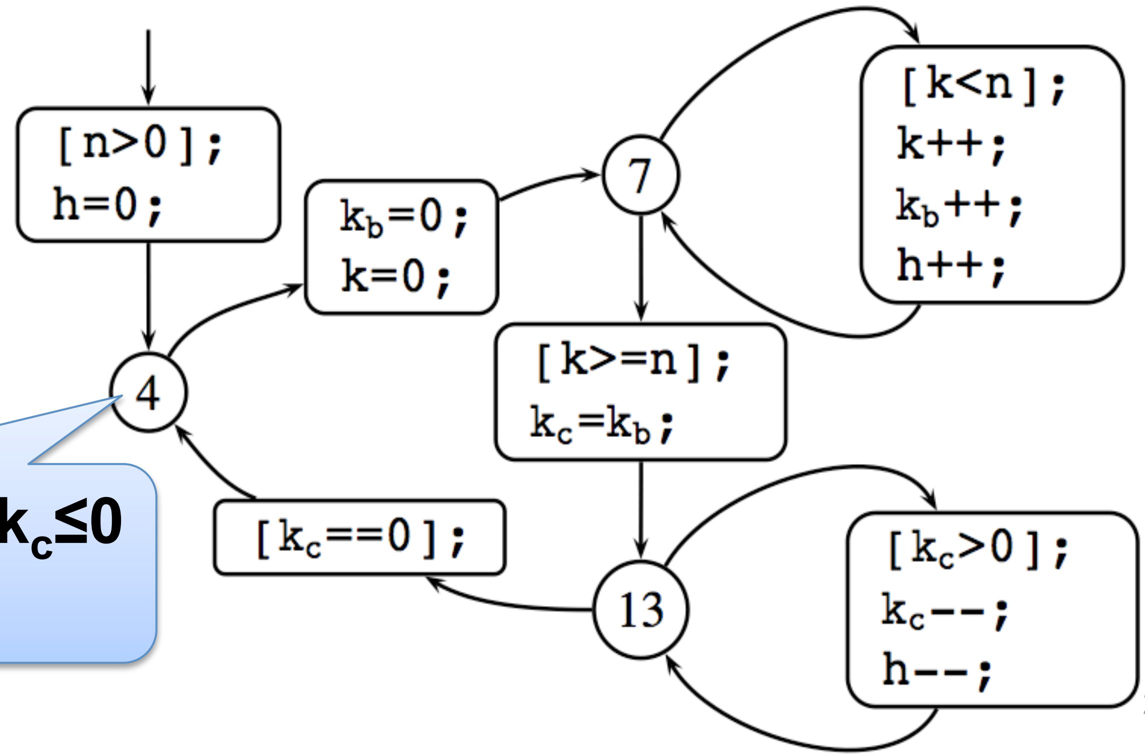
  - $\mathbf{b_n}$ and $\mathbf{b}$ are parameters
  - $\mathbf{n}$ is generic input

Template Maps

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

---

Example:



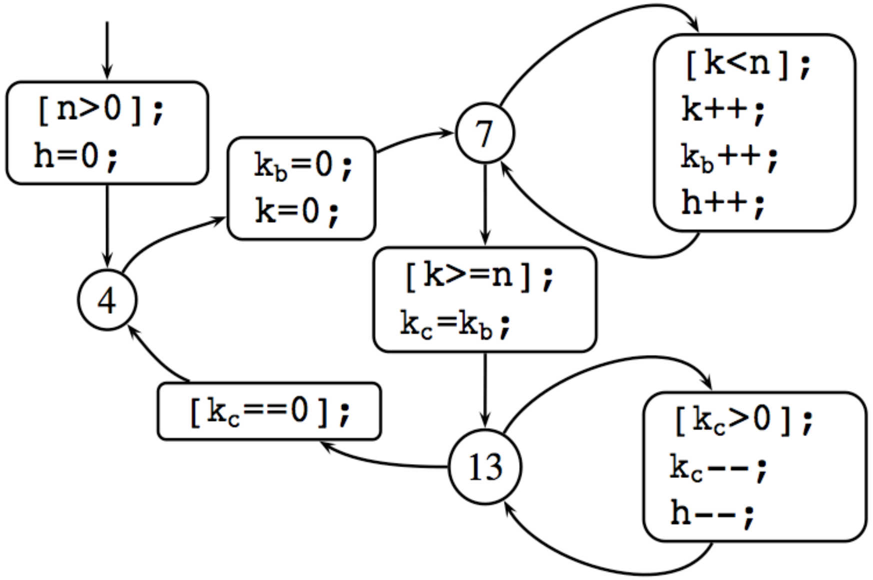
Inv$_4$: $a+a_n*n+\ldots+a_c*k_c \leq 0$

Bnd$_4$: b_n*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

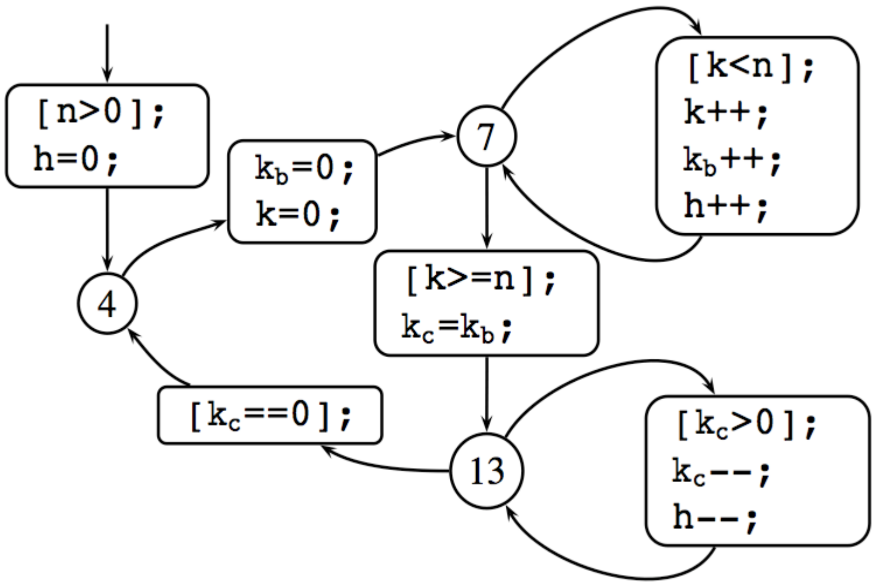


Example:

- For transition location 7 to 13
  $\mathbf{Inv_7} \wedge \mathbf{trans(7,13)} \rightarrow \mathbf{Inv'_{13}}$
- For bound at location 7
  $\mathbf{Inv_7} \rightarrow \mathbf{h \leq Bnd_7}$

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

  Bnd_7: b_n*n+b

- Solution of parameters, b_n=1 and b=0

  Bnd_7: 1*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 = alloc(); ➔ x=m; m=h[m];
  - free(x); ➔ 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**

Implementation

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

Program	Bound	C LOC	VHDL LOC	Bound synthesis time
merge	$8 * n_1 + 8 * n_2$	80	1927	600m
prio	$8 * n$	56	1475	4s
packet	$12 * n + 8$	95	2430	6s
bst_dict	$24 * n_1$	142	2703	80s

Synthesis and implementation results

Program	ALUTs	Registers	Block Mem	Blocks	Speed
merge	5,157	4,694	8,192	2	90MHz
prio	5,859	4,598	4,096	1	83MHz
packet	9,413	9,158	8,192	2	76MHz
bst_dict	5,786	5,660	8,192	2	125MHz

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!