Finding heap-bounds for hardware synthesis

B. Cook MSR · A. Gupta MPI-SWS · S. Magill CMU · A. Rybalchenko MPI-SWS · J. Simsa CMU · S. Singh MSR · V. Vafeiadis MSR

***Abstract*—Dynamically allocated and manipulated data structures cannot be translated into hardware unless there is an upper bound on the amount of memory the program uses during all executions. This bound can depend on the *generic parameters* to the program, *i.e.*, program inputs that are instantiated at synthesis time. We propose a constraint based method for the discovery of memory usage bounds, which leads to the first-known C-to-gates hardware synthesis supporting programs with non-trivial use of dynamically allocated memory, *e.g.*, linked lists maintained with `malloc` and `free`. We illustrate the practicality of our tool on a range of examples.**

I. Introduction

C-to-gates synthesis promises to bring the power of hardware based acceleration to mainstream programmers and to radically increase the productivity of digital designers [17]. However, today's C-to-gates synthesis tools do not support one of the most powerful and widely used features of high-level programming in C—dynamically allocated data structures. Thus, with today's tools we usually cannot synthesize gates from off-the-shelf C-based software. The support for dynamic memory abstraction remains an on-going research problem because of the need to efficiently and accurately determine a bound on heap consumption.

This paper advances the state-of-the-art in hardware synthesis by providing support for programs that dynamically allocate, deallocate, and manipulate heap-based data structures. Our technical contribution is a constraint-based method for finding a symbolic bound on the maximum heap size at compile time. This symbolic bound is expressed as a function on the *generic parameters* to the circuit description.[1] With our method for computing symbolic bounds we can then automatically translate C programs with dynamic memory usage into equivalent programs that operate over statically allocated arrays. That is, when circuit descriptions are instantiated in their surrounding designs, the symbolic bounds can be used to compute concrete bounds for use during synthesis.

Our method significantly increases the expressive power available to the users of synthesis systems. For example, with our new C-to-gates synthesis flow, a designer can think in terms of a tree-based data structure (*e.g.* as used in a Huffman encoder), yet generate hardware that operates on a flat fixed sized array. Furthermore, off-the-shelf libraries can now be used as subroutines by digital designers. This leads to better reuse, as well as new avenues of adapting software verification techniques for use in hardware systems.

[1]The term generic parameter is used in hardware design languages to describe variables whose values will be known at compile-time.

Related work. C-to-gates synthesis is a maturing field with notable systems—see [6], [7], [13], [18], [21], [26], [32], [33]. Some existing C-to-gates synthesis systems already support pointers and pointer aliasing, see *e.g.* [31], but they do not deal with dynamically allocated data structures.

Synthesis tools for other general purpose programming languages also exist (*e.g.* tools supporting Scheme [29], or Haskell [3]). In a few rare instances (*e.g.* [5]) tools have been used not only to generate hardware but also the circuit's correctness proof as well. These tools usually require the user to estimate the maximal amount of memory allocated by the program and take this quantity as an input parameter to the synthesis routine. Thus, the results of our work can perhaps be used with these existing tools.

In the domain of pure functional programming languages, the topic of heap-bounds analysis has been extensively investigated, see *e.g.* [19]. For imperative programs, [20] develops a type system which tracks memory consumption. The Java memory-bounds tool described in [1] uses a heap abstraction and applies heuristics based on arithmetic simplification to find a memory bound. In contrast, our method uses a more precise numerical abstraction for dealing with heap, as we keep track of the size of intermediate list segments identified by the shape analysis when dissecting the heap, which was crucial for dealing with our examples. Furthermore, instead of using heuristics for finding the bound expression, we apply a constraint based boundedness analysis which is complete for linear bound expressions provable using linear invariants.

The semi-manual technique proposed in [4] uses the Daikon [11] to collect likely program invariants—including facts about memory consumption—and uses them to derive an initial set of bound candidates.

In principle, the existing techniques for proving computational complexity, *e.g.* [14], can be used as a basis to design an algorithm for discovery of memory usage bounds. However, since we are only interested in bounds expressed over generic parameters, a major challenge is to bias the bound discovery method towards such well-formed bounds. Our constraint based procedure solves this challenge.

Our approach for finding symbolic bounds uses several known methods and tools as sub-procedures, such as shape analysis (*e.g.* [10], [23], [25]) and abstraction methods based on the introduction of new variables (*e.g.* [22], [24]). Our new constraint-based method draws influence from previously developed methods for invariant generation and rank function synthesis (*e.g.* [9], [30]).

```
void prio(int n,in_signal i,out_signal o) {
  LINK *tmp,*c,*buffer;
  assert( n>0 );
  while (1) {
    buffer = NULL;
    // Build up an n-sized sorted buffer
    for (int k=0;k<n;k++) {
      buffer = sorted_insert(input(i),buffer);
    }
   // Send the sorted list to the output and
   // deallocate the buffer as we walk it
   c=buffer;
   while(c!=NULL) {
     output(o,c->data);
     tmp = c;
     c = c->next;
     free(tmp);
   }
  }
}
```

Fig. 1. Priority queue circuit specification in C, using off-the-shelf implementation of `sorted_insert`. The *generic* parameter `n` is assumed to be specified at compile-time.

```
LINK * sorted_insert(int data, LINK *l) {
  LINK * elem = l;
  LINE * prev = NULL;
  LINK * x = (LINK*)malloc(sizeof(LINK));
  assert(x!=NULL);
  x->data = data;
  while (elem != NULL) {
    if (elem->data >= x->data) {
      x->next = elem;
      if (prev == NULL) { l = x; return l; }
      prev->next = x;
      return l;
    }
    prev = elem;
    elem = elem->next;
  }
  x->next = elem;
  if (prev == NULL) { l = x; return l; }
  prev->next = x;
  return l;
}
```

Fig. 2. Off-the-shelf implementation of incremental insertion sort procedure.

II. Example

Imagine that we would like to build an `n`-size priority queue circuit that reads integers from an input signal and returns every `n` input integers on an output signal in sorted order—such a circuit is key to the development of a Huffman encoder. See the function `prio` in Fig. 1 for an example of how we might wish to write a specification of the desired hardware in C. Our intention is that the variable `n` in Fig. 1 is a generic parameter, whereas `i` and `o` should be thought of as signal names. Our synthesis tool treats these in a special way as standard C, of course, does not make this distinction. In this example we assume that the circuit uses `input()` and `output()` as primitives for I/O on the signal variables `i` and `o`. `LINK` is a C struct used to represent singly-linked lists (with fields `data` and `next`). We make use of an existing off-the-shelf insertion-sort implementation, `sorted_insert`. See Fig. 2 for the source code of `sorted_insert`.

Note that in order to convert this program into hardware we must first find an *a priori* bound on the amount of heap during the execution of `prio`, for any input or parameter. The problem is that `sorted_insert` does not guarantee a concrete bound on the amount of heap allocated while its executing, instead it preserves a bound–it takes a state where k heap cells have been allocated and returns a state in which $k+1$ have been allocated. Thus we must hope to find a bound on the amount of heap used by `sorted_insert` from states limited to those reachable from `prio`.

If we can find this bound, then we can convert the program's operations on the heap into operations on statically-allocated arrays, thus facilitating synthesis. We aim to find a bound that holds across the entire program, but is expressed symbolically using only the generic parameters to the top-level function (*i.e.* the parameter `n` of the circuit `prio`). This allows us to pre-allocate a shared array when creating instances of the circuit `prio`.

The procedure given later in Section III is designed to find a function f such that it is a program invariant that $f(\texttt{n})$ is larger than the number of heap cells allocated at any given time during its execution. In this case the procedure described later will find the function $f(\texttt{n}) = \texttt{n} * 8$, assuming that $\texttt{sizeof(LINK)} = 8$ in the encoding.

With f we can now re-encode the program using a pre-allocated array. In essence, when we know the valuations to the input parameters we can then pre-allocate an array using f. We then convert dereferences like `*c` into `a[c]`. Field offsets are explicitly encoded: `c->data` is encoded as `a[c+0]`, and `c->next` is encoded as `a[c+4]`.

From this program (and via a translation into VHDL) we then used the Altera Quartus II 9.0 tools to construct an implementation for the Stratix III FPGA architecture. Using default synthesis and implementation options and with $\texttt{n} = 10$, the generated circuit uses 5859 adaptive look-up tables, 4598 logic registers and 8192 block memory.

III. From heaps to arrays

In this section we describe an analysis that automatically discovers symbolic bounds on the heap usage. We will assume that the size parameters passed to `malloc` are fixed constants. Through the use of static analysis, we annotate each call to `free` with the amount of memory the call is freeing. For example, we would transform the call `free(tmp)` from Fig. 1 to `free(tmp,sizeof(LINK))`. For simplicity of presentation we will assume that programs allocate and free heap cells of a single fixed size. We can support multiple size allocations through the use of compile-time partial evaluation, but at the cost of complexity in the notation in this section. We currently do not support arbitrary DAGs or hash-tables, due to the limitations of existing separation logic based shape analysis tools [8], [10], [23], [25] of which we are dependent.

Our procedure is divided into the following steps.

a) Numerical heap abstraction: First, we augment the program with a new variable h, which is used to track the amount of heap that is currently allocated. The variable h is incremented when malloc is called, and decremented when free is called. For memory-safe programs such behavior of h is correct. We use the shape analysis tool THOR [25] to determine the shape of the data structures used during the program's execution, and to prove memory safety. Using techniques from [24], THOR can be used to produce a new program without heap that is a sound abstraction of the original program—additional integer variables are added by THOR to summarize the sizes of data-structures. Thus, bounds found on h in the abstraction imply bounds in the original program. Note that the new program variables range over integers of arbitrary size (*i.e.* they cannot be represented in 32 or 64 bits).

b) Numerical bounds analysis: Next, we apply our constraint-based boundedness analysis to the numeric program to find a symbolic bound f on the maximum value of h. For improved scalability we combine our constraint-based synthesis approach with a counterexample-guided method of checking and refining candidate bounds.

c) Array-based heap management and synthesis: Once we have computed a symbolic bound (assuming that a bound can be found) we throw away the abstraction and then convert the original program into an array-based program operating over a pre-allocated shared array and then apply off-the-shelf synthesis tools to produce a gate-level design. Note that, although we may sometimes compute a conservative over-approximation for a bound on memory usage, it is often the case that a downstream synthesis tool can perform further pruning to yield a gate level implementation that does indeed have a better (or even ideal) bound. A simple case of this scenario is when a list is used to represent a bit-vector which is used in arithmetic expressions with known range at synthesis time allowing some of the upper bits to be pruned.

The following sections discuss the above procedures in more detail.

IV. NUMERICAL HEAP ABSTRACTION

A shape analysis tool is designed to take a program and compute an invariant for each program location describing the shape of the heap. The invariant describes the data structures stored in the heap during the program's execution. Shape analysis tools are based on symbolic simulation together with abstraction techniques.

Using techniques described in [24], the shape analysis tool THOR can be used to introduce new variables which soundly track the sizes of data structure shapes inferred by the shape analysis. In the example of the function `prio`, THOR would introduce a variable $\mathtt{k_b}$ recording the length of the linked list starting from `buffer`. At the command `buffer = NULL`, we initialize $\mathtt{k_b}$ to zero. At the lines `prev->next = x`

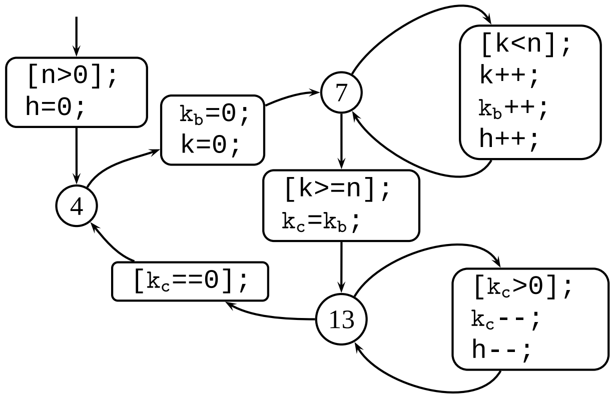

Fig. 3. Numerical abstraction of procedure `prio` shown from Fig. 1. Commands of the form `[`e`];` denote assume statements.

within `sorted_insert`, the length of that linked list is increased; therefore the abstraction will increment $\mathtt{k_b}$. Similarly, THOR will introduce another variable $\mathtt{k_c}$ recording the length of the linked list from `c`. Corresponding to the assignment `c=buffer`, the abstraction will set $\mathtt{k_c}$=$\mathtt{k_b}$, and at the assignment `c=c->next`, the abstraction decrements $\mathtt{k_c}$. Also, when we exit the `while(c!=NULL)` loop, we know that `c==0`, and hence also $\mathtt{k_c}$=0.

Fig. 3 shows the control-flow graph (CFG) of the resulting abstraction of `prio`. The CFG contains three nodes corresponding to the three loops in the `prio` function. These nodes are connected by the edges which are annotated with the code occurring between the locations. The transitions between locations come in two forms: assignments v=e`;` and assumption checks `[`e`];`. The assumptions prune executions in which the condition e does not hold.

For brevity, calls to the function `sorted_insert` in Fig. 3 have been summarized as the transition $\{\mathtt{k_b}$`++;h++;`$\}$ from location 7 to 7, but our technique is designed to work for a fully expanded CFG of the code.

V. NUMERICAL BOUNDS ANALYSIS

Preliminaries. Our shape analysis procedure produces a program $\mathcal{P} = (V, h, P, \mathcal{L}, \ell_{init}, \mathcal{T})$ that consists of a set of variables V, a heap consumption variable $h \in P$, a set of generic parameters $P \subseteq V \setminus \{h\}$, a set of locations $\mathcal{L}$, an initial location $\ell_{init} \in \mathcal{L}$ and a set of abstract transitions $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is given by a tuple (ℓ, ρ, ℓ') where $\ell, \ell' \in \mathcal{L}$ and ρ is a constraint over $V \cup V'$, where the variables in V' represent the values of variables V after the transition is executed. Each transition relation preserves the values of generic parameters, *i.e.*, for each $(\ell, \rho, \ell') \in \mathcal{T}$ we have

$$\forall V\ \forall V' : \rho \rightarrow P' = P\ .$$

A *state* s is a valuation of V. A *computation* is a sequence of location and state pairs $(\ell_1, s_1), (\ell_2, s_2), \ldots$ such that ℓ_{init} is the initial location, *i.e.*, $\ell_1 = \ell_{init}$, and for each consecutive pair (ℓ_i, s_i) and (ℓ_{i+1}, s_{i+1}) there is a transition $(\ell_i, \rho, \ell_{i+1}) \in$

$\mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A state s is reachable at a location ℓ if the pair (ℓ, s) appears in some computation.

An *invariant* at a location $\ell \in \mathcal{L}$ is a superset of all reachable states at ℓ. We represent invariants by formulas over the variables V. An *invariant map* Inv assigns an invariant to each location. In particular, we have $\mathit{Inv}(\ell_{init}) = \mathit{true}$, *i.e.*, every state is reachable at the initial location. We will use primed notation $\mathit{Inv}(\ell)'$ for $\mathit{Inv}(\ell)[V'/V]$. An invariant map Inv is *parametric* if it does not restrict the values of program variables besides the generic parameters and the heap consumption variable, *i.e.*, for each $\ell \in \mathcal{L}$ we have

$$\forall V : \mathit{Inv}(\ell) \leftrightarrow (\exists V \setminus (P \cup \{h\}) : \mathit{Inv}(\ell)) \ .$$

An invariant map Inv is *inductive* if for each program transition $(\ell, \rho, \ell') \in \mathcal{T}$ we have

$$\forall V \ \forall V' : \ (\mathit{Inv}(\ell) \wedge \rho) \rightarrow \mathit{Inv}(\ell')' \ .$$

We are interested in a parametric invariant map Bnd that bounds the heap consumption. Formally, we will search for Bnd such that for each $\ell \in \mathcal{L}$ we have

$$\forall P \ \exists c \in \mathbb{N} \ \forall V \setminus P : \mathit{Bnd}(\ell) \rightarrow h \leq c \ .$$

Then, the maximal value of the constant c among all program locations determines the maximal amount of memory that is dynamically allocated during the program computation.

For proving that Bnd is valid we will need an inductive invariant map Inv. Formally, we require that for each $\ell \in \mathcal{L}$ holds

$$\forall V : \mathit{Inv}(\ell) \rightarrow \mathit{Bnd}(\ell) \ .$$

Bounds analysis algorithm. Fig. 4 presents our constraint-based procedure BOUND for discovering heap consumption bounds. The procedure takes as parameters a program $\mathcal{P}$, an invariant template map Inv^T, and a bound template map Bnd^T. It returns either a valid bound map or an exception if no such map can be found.

The template maps used by BOUND are reminiscent of those used in constraint-based invariant generation [9], [30] and rank function synthesis [27]. A template map assigns an assertion over program variables and *template* parameters to each program location. The template map Inv^T may use a template of the form

$$\alpha_1 v_1 + ... + \alpha_n v_n \leq \alpha \wedge \beta_1 v_1 + ... + \beta_n v_n \leq \beta \ ,$$

which is a conjunction of two linear inequalities with the template parameters $\alpha_1, ..., \alpha_n, \alpha, \beta_1, ..., \beta_n, \beta$ and program variables $V = \{v_1, \ldots, v_n\}$.

The bound template map Bnd^T given to BOUND as input assigns to each program location a bound template of the form

$$h \leq \delta_1 p_1 + \cdots + \delta_m p_m + \delta \ ,$$

where $\delta_1, \ldots, \delta_m, \delta$ are *template* parameters and $P = \{p_1, \ldots, p_m\}$ are *generic* parameters. Since Bnd^T only refers to P and h, it guarantees to yield parametric bound invariants only.

```
procedure BOUND
input
  $\mathcal{P} = (V, h, P, \mathcal{L}, \ell_{init}, \mathcal{T})$: program
  $\mathit{Inv}^T$: invariant template map
  $\mathit{Bnd}^T$: bound template map
var
  $Q$: template parameters in $\mathit{Inv}^T$ and $\mathit{Bnd}^T$
  $\Psi$: auxiliary constraint over $Q$
begin
1   $\Psi := \mathit{true}$
2   foreach $\ell \in \mathcal{L}$ do
3     $\Psi := \Psi \wedge \forall V : \mathit{Inv}^T(\ell) \rightarrow \mathit{Bnd}^T(\ell)$
4   foreach $(\ell, \rho, \ell') \in \mathcal{T}$ do
5     $\Psi := \Psi \wedge \forall V \ \forall V' : (\mathit{Inv}^T(\ell) \wedge \rho) \rightarrow \mathit{Inv}^T(\ell')'$
6   $Q$ := free variables in $\Psi$
7   if exists $M$ such that $\Psi(M)$ then
8     return $\mathit{Bnd}^T[M/Q]$
9   else
10    raise "no bound found"
end
```

Fig. 4. BOUND discovers bounds on the value of the variable h, which keeps track of the amount of dynamically allocated memory.

BOUND collects a conjunction of constraints Ψ over template parameters for both template maps in lines 1–5. These constraints encode the condition that the computed bounds must be valid. Lines 2–3 state that the bounds hold for all reachable states, which are represented by an invariant map induced by the invariant template map Inv^T. Lines 4–5 encode the condition that Inv^T in fact represents all reachable program states.

We collect all template parameters in line 6. If our constraint solving procedure can find a satisfying assignment to Ψ, then this assignment defines a bound map in line 8. Otherwise, BOUND raises an exception.

The transition relations in the program $\mathcal{P}$ produced during the shape analysis phase are conjunctions of linear inequalities over V and V'. For our templates consisting of linear inequalities, we eliminate the universally quantification over V and V' in lines 3 and 5 of BOUND by applying a standard technique, see *e.g.* [9], based on Farkas' lemma [12]. The resulting constraint Ψ is a conjunction of non-linear inequalities and can be efficiently solved using the existing tools, *e.g.* [15], [16].

The soundness and completeness of BOUND is formalized in the following theorem.

Theorem 1. *The procedure* BOUND *is complete for bound expressions in linear arithmetic provable using linear arithmetic invariants,* i.e., *in this case it computes a bound map. The procedure* BOUND *is also sound,* i.e., *it computes a bound map that represents an upper bound on the memory usage.*

Example. Consider the program in Fig. 3 over the variables `n`, h, `k`, $\mathtt{k_b}$, and $\mathtt{k_c}$. The only generic parameter is the variable `n`.

We consider a template map Inv^T that assigns to each program location a conjunction of two linear inequalities. For example, for the location ℓ_7 we have

$$\begin{aligned} Inv^T(\ell_7): \quad & \alpha_{\mathtt{n}}\mathtt{n} + \alpha_h h + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{k_b}}\mathtt{k_b} + \alpha_{\mathtt{k_c}}\mathtt{k_c} \leq \alpha \land \\ & \beta_{\mathtt{n}}\mathtt{n} + \beta_h h + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{k_b}}\mathtt{k_b} + \beta_{\mathtt{k_c}}\mathtt{k_c} \leq \beta \land \\ & \gamma_{\mathtt{n}}\mathtt{n} + \gamma_h h + \gamma_{\mathtt{k}}\mathtt{k} + \gamma_{\mathtt{k_b}}\mathtt{k_b} + \gamma_{\mathtt{k_c}}\mathtt{k_c} \leq \gamma \end{aligned}$$

The bound template at this location is

$$Bnd^T(\ell_7): \quad h \leq \delta_{\mathtt{n}}\mathtt{n} + \delta \ .$$

Next, BOUND creates a conjunction of constraints Ψ over the template parameters from all program locations. We only present two constraints from Ψ that are created at lines 3 and 5 for the location ℓ_7 and the loop transition at the location ℓ_7 respectively. The first constraint is the implication

$$\forall \mathtt{n}\ \forall h\ \forall \mathtt{k}\ \forall \mathtt{k_b}\ \forall \mathtt{k_c} : Inv^T(\ell_7) \to Bnd^T(\ell_7) \ .$$

The second constraint involves the transition relation of the loop:

$$\begin{aligned} & \forall \mathtt{n}\ \forall h\ \forall \mathtt{k}\ \forall \mathtt{k_b}\ \forall \mathtt{k_c}\ \forall \mathtt{n}'\ \forall h'\ \forall \mathtt{k}'\ \forall \mathtt{k_b}'\ \forall \mathtt{k_c}' : \\ & \quad (Inv^T(\ell_7) \land \\ & \quad \mathtt{k} < \mathtt{n} \land \mathtt{n}' = \mathtt{n} \land h' = h + 1 \land \mathtt{k}' = \mathtt{k} + 1 \land \\ & \quad \mathtt{k_b}' = \mathtt{k_b} + 1 \land \mathtt{k_c}' = \mathtt{k_c}) \to \\ & \quad Inv^T(\ell_7)' \end{aligned}$$

We solve Ψ and obtain $\delta_{\mathtt{n}} = 1$ and $\delta = 0$ for the bound template parameters occurring in the location ℓ_7, *i.e.*, we have

$$Bnd^T(\ell_7) = (h \leq \mathtt{n}) \ .$$

The corresponding invariant map assigns $h \leq \mathtt{k_b} \land \mathtt{k_b} \leq \mathtt{k} \land h \leq \mathtt{n}$ to the location ℓ_7. In our example, the bound occurs in the corresponding inductive invariant; in general, however, this need not be the case.

Incremental bounds analysis. The constraint-based procedure BOUND performs an expensive computation—non-linear constraint solving—and does not scale beyond medium-sized programs. We improve the scalability of BOUND by performing the boundedness analysis in an incremental fashion using the idea of path invariants [2]. We apply the expensive, constraint-based procedure only to certain program fragments, which are determined automatically.

Fig. 5 presents our BOUND-based procedure INCBOUND for an incremental discovery of heap consumption bounds for the full program from its fragments. Initially, the bound map states that no heap consumption takes place, see line 3. Then, this claim is verified in lines 6–7 using a verification tool for proving program safety. Such a tool is applied on an augmented program that is obtained from $\mathcal{P}$ by adding a distinguished error location ℓ_{err} that is reachable if the heap bound claimed by Bnd is not valid. In the case of a false bound, the algorithm will return a counterexample in the form of a sequence of transitions π that leads to heap consumption beyond the claimed bound.

```
procedure INCBOUND
input
  $\mathcal{P} = (V, h, P, \mathcal{L}, \ell_{init}, \mathcal{T})$: program
  $Inv^T$: invariant template map
  $Bnd^T$: bound template map
var
  $Bnd$ : bound map
  $\ell_{err}$: distinguished error location
  $\mathcal{T}_\mathcal{E}$: transitions for bound assertion checking
function PATHPROGRAM
input
  $\pi$ : sequence of transitions
begin
1   return $(V, h, P, \mathcal{L}, \ell_{init},$
2           $\{\tau \mid \tau = (\ell, \rho, \ell')$ occurs in $\pi$ and $\ell' \neq \ell_{err}\})$
end;
begin
3   $Bnd$ := $\lambda \ell \in \mathcal{L}. h \leq 0$
4   repeat
5     $\mathcal{T}_\mathcal{E}$ := $\{(\ell, \neg Bnd(\ell) \land V' = V, \ell_{err}) \mid \ell \in \mathcal{L}\}$
6     if exists $\pi \in (\mathcal{T} \cup \mathcal{T}_\mathcal{E})^*$ from $\ell_{init}$ to $\ell_{err}$
7       such that $\rho_\pi \neq \emptyset$ then
8       $\mathcal{P}_\pi$ := PATHPROGRAM($\pi$)
9       try
10        $Bnd_\pi$ := BOUND($\mathcal{P}_\pi$, $Inv^T$, $Bnd^T$)
11      catch
12        return "unbounded consumption path $\pi$"
13      $Bnd$ := $\lambda \ell \in \mathcal{L}. Bnd(\ell) \lor Bnd_\pi(\ell)$
14    else
15      return "bound assertion map $Bnd$"
16  done
end
```

Fig. 5. INCBOUND performs an incremental boundedness analysis using guidance from spurious counterexamples.

In case a counterexample π is found, we identify a fragment of $\mathcal{P}$ that is traversed by the transitions occurring in π. This code fragment is defined by a path program $\mathcal{P}_\pi$ for π [2], see lines 1–2. In particular, the path program $\mathcal{P}_\pi$ traverses the same loops of $\mathcal{P}$ that are visited by π.

We compute an adjustment Bnd_π for the bound map by apply the procedure BOUND on the path program, see line 10. The adjustment is used to weaken the claimed bound, see line 13.

This sequence of incremental adjustments continues until either the full program $\mathcal{P}$ satisfies the claimed bound map or a path that for which no heap consumption bound can be found is discovered.

The soundness and completeness properties of INCBOUND are inherited from the procedure BOUND and the notion of path invariants.

Theorem 2. *The procedure* INCBOUND *is complete for bound expressions in linear arithmetic provable using linear arithmetic invariants,* i.e., *in this case it computes a bound map and terminates. The procedure* INCBOUND *is also sound,* i.e.,

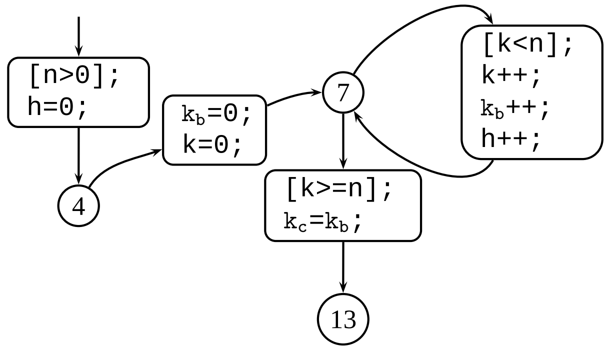

Fig. 6. Path program for the program from Fig. 3 and a path consisting of transitions between the locations (ℓ_{init}, ℓ_4), (ℓ_4, ℓ_7), (ℓ_7, ℓ_7), and (ℓ_7, ℓ_{13}).

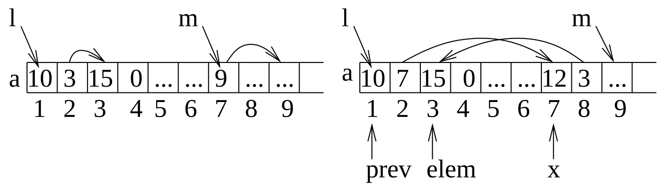

Fig. 7. Creation of a new `LINK` structure in the array-based heap implementation.

it computes a bound map that represents an upper bound on the memory usage.

Example. Consider finding a bound for h in the program from Fig. 3. In the algorithm from Fig. 5 we start with a candidate bound $h \leq 0$ at each location. We can then attempt to prove that $h \leq 0$ at every location using a symbolic model checker (this corresponds to lines 5-7 of Fig. 5. In this case $\mathtt{h} \leq 0$ is not necessarily true at location 7 in Fig. 3, in which case the symbolic model checker will return a witness counterexample path. Imagine that we get the path $\pi = 4 \rightarrow 7$. In this case PATHPROGRAM(π) will return a sub-program of Fig. 3, as found in Fig. 6. We can then find a bound on this sub-program, resulting in $\mathtt{h} \leq \mathtt{n}$. Thus, we refine the candidate whole-program bound to be $\mathtt{h} \leq 0 \vee \mathtt{h} \leq \mathtt{n}$. Repeating the steps from lines 5-7 allows us to prove that $\mathtt{h} \leq 0 \vee \mathtt{h} \leq \mathtt{n}$ is a valid bound for the whole program. After simplification, we return $\mathtt{h} \leq \mathtt{n}$.

VI. ARRAY-BASED HEAP MANAGEMENT

Numerical boundedness analysis computes a bound on the maximal amount of memory that is dynamically allocated during program computation, and represents this bound as a function of generic parameters. When synthesizing a hardware implementation, the generic parameters are instantiated. Hence we obtain a concrete bound, say N.

Next, we replace all heap operations in the program $\mathcal{P}$ by operations on a statically allocated array `a` of size N. Each pointer to the heap becomes an array index. Field accesses are converted into arithmetic operations over array indices. For example, the statement `c = c->next;` from the program in Fig. 1 becomes `c = a[c+4];`.

We use a list of array indices that is embedded into the array `a` to keep track of free array cells. Each list element is an index of a free cell. We introduce a global variable `m` that stores the head of the list, and hence the cell at index `m` is free. Then, the value of `a[m]` is the next list element, which is the index of the second free cell stored in the list. We obtain the third element by accessing `a[a[m]]` and so on. Initially $\mathtt{m} = 0$ and the array `a` is initialized in the following way:

$$\forall 0 \leq i < N : \mathtt{a}[i] = i + 1 \ .$$

A call to `malloc()` consumes the head of the list. That is, `x = malloc()` is implemented by the sequence of instructions `x = m; m = a[m];`, where the first assignment delivers the free cell and the second assignment ensures that the subsequent call to `malloc` will return the next free cell in the list. We do not need to check whether the free list empty because the boundedness analysis guarantees that it will never happen, i.e., we have $m \leq N$.

Fig. 7 illustrates the array-based treatment of `malloc`. We assume that the heap stores data structure `LINK`, whose size is two integers, and that each array cell is of size one integer. The array on the left is free starting at the index 7, as represented by the valuation $\mathtt{m} = 7$, $\mathtt{a}[9] = 9$, etc. After executing `x = malloc(2);`, assigning `x->data = 12;`, the cell at index 7 is no longer free. It stores the data value 12. The next free cell becomes the first one available, i.e., we have $\mathtt{m} = 9$. After identifying the predecessor and successor of `x`, i.e., inserting x into the sorted heap, we obtain the array show on the right in Fig. 7.

A call to `free(x)` pushes `x` onto the free list. That is, this call translates to a pair of statements `a[x] = m; m = x;`. The last freed cell will be the first free cell in the list of free cells, i.e., the subsequent call to `malloc` will return the last freed cell.

VII. EXPERIMENTAL RESULTS

In this section we discuss the results of our experiments with the proposed synthesis procedure on a number of real-world examples. The original input C programs and the resulting outputs are available at http://www.cs.cmu.edu/~jsimsa/c2vhdl.tar. Before discussing the outputs of our tool, we first describe the problems solved by the C-based software models.

Priority queue – This is our running example from Figure 1. The design has one input signal and one output signal. The implementation repeatedly inputs n elements, sorts them, and outputs them in a sorted order. For the sake of experimental evaluation we chose $n = 10$.

Merge sort – This example implements a merger of two sorted sequences. The design has two input signals and one output signal. The implementation repeatedly receives n_1 sorted elements through the first input signal and n_2 sorted elements through the second input signal. Using the merge sort it combines the two sequences into one sorted sequence, which is then output. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Packet sorting – This example implements a simple network element. The design has two input signals and one output signal. The implementation repeatedly inputs packet data through the first input signal and packet identifier through the second input signal. It inserts these packets into a buffer while ignoring duplicate identifiers, until it fills a buffer with n packets. It then sorts the received packets by their identifier and outputs them. For the sake of experimental evaluation we chose $n = 10$.

Binary search tree dictionary – This example implements a data structure for storing a set of elements with a test for membership. The design has two input signals and one output signal. The implementation repeatedly inputs n_1 elements through the first input signal and builds a binary search tree out of them. This is followed by receiving n_2 queries through the second input signal and producing the correct response through the output signal. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Huffman encoder – This example implements a data structure for encoding symbols. The design has three input signals and one output signal. The implementation repeatedly inputs n_1 symbols through the first input signal, their frequencies through the second input signal, and builds a Huffman encoder using this data. This is then followed by receiving n_2 symbols through the third input signal and producing their binary encoding through the output signal. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Each of these models was succesfully run through the sequence of procedures described in this paper: shape analysis, bounds analysis, and array transformation.

Table I lists the symbolic bounds for our examples in bytes[2]. These symbolic bounds were then concretized using the aforementioned values and run through our translation tool which inputs a C program and a concrete bound and generates a functionally equivalent VHDL program. The Table I also lists lines of code (LOC) for both the hand-written C models and their automatically generated VHDL counterparts. Note that, due to last minute changes to the bounds search algorithm, we were unable to produce a consistent set of run-times for all of the examples in time for this submission: the circuits were computed on various machines with different architectural specifications. The run-times range from minutes to hours depending on the example.

Our VHDL generation step is carefully crafted to work well with FPGA synthesis tools. The generated VHDL files were synthesized using the Altera Quartus II 9.0 tools (build 184 04/29/2009 SP1 SJ Web Edition) targeting Stratix III FPGAs. The results are shown in Table II. The ALUT (Altera's adaptive look-up tables) column gives an indication of the size of the combinational elements in the generated design. The registers column indicates how many flip-flops in the logic fabric were used for registers. The block mem column indicates how many memory bits in the generated design were implemented using embedded memory blocks and the following column shows how many independent memories were synthesized. The last column shows the maximum speed. In all cases the tools automatically picked the smallest EP3SL50F484C2 FPGA and package and the timing results are given for this part.

[2]Data types size and structure alignment of a 32-bit architecture (e.g. 4-byte pointers) is assumed.

Program	Bound	C LOC	VHDL LOC
merge	$8 * n_1 + 8 * n_2$	80	1927
prio	$8 * n$	56	1475
packet	$12 * n + 8$	95	2430
huffman	$52 * n_1 - 12$	202	5855
bst_dict	$24 * n_1$	142	2703

TABLE I
BOUNDS AND LINES OF CODE

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
huffman	20,678	11,116	12,288	3	76MHz
bst_dict	5,786	5,660	8,192	2	125MHz

TABLE II
SYNTHESIS AND IMPLEMENTATION RESULTS

Most of the synthesized circuits occupy only a small portion of the smallest Stratix-III FPGA. The largest design is huffman which utilizes 55% of the combinational ALUTs but less than 1% of the available block memory and only 29% of the available logic registers. The smallest design is prio which occupies 15% of the available combinational ALUTs, 12% of the available logic registers and less than 1% of the available block memory. The operating frequency of these circuits is in a range which is typical for FPGA circuits used as co-processing circuits. We have tested several of our examples running on a Cyclone II FPGA on the Altera DE2 board. For example, the priority encoder circuit was synthesized, implemented and run on the Altera Cyclone II EP2C35F672C6 FPGA (supporting 33,216 logic elements) and we have observed the correct behavior on actual hardware using the SignalTap logic analyzer. Our conclusion from these preliminary results is that we have identified a viable approach for translating heap-based C programs into VHDL designs which have acceptable area utilization and performance.

Examples of failure. Our approach for symbolic bounds synthesis can fail in various ways. For example, the input program might operate over DAGs (*e.g.* BDDs) or hash tables;

in which case, we would currently fail to produce an arithmetic abstraction. Note that—even in the case of programs with simple linked data structures—improving the scalability and accuracy of shape analysis is an area of active research. When we successfully generate arithmetic abstractions, our constraint-based synthesis algorithm can also fail. The abstraction may be too coarse, or the problem may be too complex (*e.g.* highly non-linear). Consider the case of a "watcher list" for a literal ℓ in a SAT solver, which tracks the clauses in the clause database in which ℓ appears. A bound on the size of this list certainly exists, but our tool cannot work out what this bound is.

VIII. Conclusion

C-to-gates synthesis aims to bring together the agility of software development with the speed of raw gates. Until now, C-to-gates synthesis systems were lacking support for non-trivial dynamic allocation and deallocation on the heap, thus limiting the wider applicability of these tools. This paper has introduced a new method that synthesizes symbolic heap bounds expressed on generic parameters. The method uses computed shape invariants and abstractions together with a constraint solving based approach to find a symbolic expression representing the bound. Our system facilitates the use of common software abstractions and libraries (potentially with no memory bounds) within C-to-gates synthesis systems. Thus, designers can potentially use high-level abstractions (*e.g.* dynamically allocated trees and lists) when designing circuits.

Future work. Using techniques described in [28], and with some modification to our shape analysis tool, we can determine the symbolic footprint of each command with respect to the global heap—allowing us to break a monolithic computation over a single memory into several smaller computations that work on independent memories. We can perhaps exploit this independence to build more parallel circuits, as well as to find potential energy savings.

In this paper we have focused on the application of our heap-bounds procedure to the problem of hardware synthesis, though it may also have application in other areas. As future work it might be fruitful to investigate its application to problems such as compilation for embedded systems, or model checking for infinite-state systems.

References

[1] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM*, 2007.

[2] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM Press, 2007.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ICFP*, 1998.

[4] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 2006.

[5] B. C. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. *Technical Report 86*, `http://www.cli.com`, 1994.

[6] F. Bruschi and F. Ferrandi. Synthesis of complex control structures from behavioral SystemC models. *DATE*, 2003.

[7] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Int. Workshop On Applied Reconfigurable Computing*, 2006.

[8] B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[9] M. A. Colon, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[10] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. *TACAS*, 2006.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[12] J. Farkas. Uber die theorie der einfachen ungleichungen. *Journal fur die Reine und Angewandte Mathematik*, 124:1–27, 1902.

[13] M. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. *FCCM*, 2000.

[14] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[15] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *TACAS*, 2009.

[16] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, 2009.

[17] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14, Apr. 1997.

[18] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *VLSI Conference*, 2003.

[19] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.

[20] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *ESOP*, 2006.

[21] IMEC. CleanC analysis tools. `http://www.imec.be/CleanC/`, 2008.

[22] R. Iosif, M. Bozga, A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.

[23] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. *SAS*, 2000.

[24] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. *SAS*, 2006.

[25] S. Magill, M. Tsai, P. Lee, and Y. Tsay. THOR: A tool for reasoning about shape and arithmetic. *CAV*, 2008.

[26] W. A. Najjar, A. P. W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8), 2003.

[27] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[28] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, 2009.

[29] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard: a Scheme to hardware compiler. In *Workshop on Scheme and Functional Programming*, 2006.

[30] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. *SAS*, 2004.

[31] L. Semeria and G. D. Micheli. SpC: synthesis of pointers in C. *ICCAD*, 1998.

[32] A. Takach, B. Bower, and T. Bollaert. C based hardware design for wireless applications. *DATE*, 2005.

[33] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench automated reconfigurable VHDL generator. *FPL*, 2007.