Gates Synthesis for C Programs with Heap

Byron Cook
Microsoft Research

Ashutosh Gupta
Max Planck Institute for Software Systems

Satnam Singh
Microsoft Research

Viktor Vafeiadis
Microsoft Research

ABSTRACT

Current C to gates synthesis tools do not support programs that make non-trivial use of dynamically-allocated heap (*e.g.* linked-list C programs that call `malloc` and `free`). The problem is that its difficult to determine an *a priori* bound on the amount of heap used during the program's execution, if a bound even exists. In this paper we develop a new method of synthesizing symbolic bounds expressed over generic parameters, thus leading to a C to gates synthesis flow for programs using dynamic allocation and deallocation.

1. INTRODUCTION

C to gates synthesis promises to bring the power of hardware based acceleration to mainstream programmers, as well as to radically increase the productivity of digital designers [10]. C to gates synthesis tools can produce competitive quality of results for systems modeled with an appropriate subset of C. However, several of C's commonly used programming abstractions remain unsupported [17]. For example, we cannot simply take off-the-shelf C-based implementations of queues or trees and synthesize them to gates, as dynamic heap allocation is largely unsupported by current tools.

This paper advances the state-of-the-art in C to gates synthesis with support for programs that dynamically allocate, deallocate, and manipulate heap-based data structures. Our technical contribution is a method of synthesizing a symbolic bound on the maximum heap size at compile time; this symbolic bound is expressed as a function on the generic parameters (in the HDL sense) to the circuit description. With our symbolic bounds we can then automatically translate C programs with dynamic heap usage into equivalent programs that modify a pre-allocated array. That is, when circuit descriptions are instantiated in larger designs, the symbolic bounds can then be used to compute concrete bounds for use during synthesis. This significantly increases the expressive power available to the users of C to gates synthesis systems. For example, with our C to gates synthesis flow, a designer can think in terms of a tree-based data structure (*e.g.* 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, whereas in the past custom array-based re-implementations had to be developed.

Related work. C to gates synthesis is a maturing field with notable systems including Catapult-C [23], CleanC [12], CoCentric [1], SA-C [18], ROCCC [2], SPARK [11], Streams-C [7] and DWARF[24]. Support for C pointers in synthesis systems is not new, *e.g.* [22]. The distinction here is one of dynamic heap versus pointers and aliasing—systems such as [22] support pointer and aliasing, but not dynamic allocation and deallocation. Thus, current tools cannot support off-the-shelf list or tree libraries without first radically modifying the C code to operate over pre-allocated structures. This is due to the fact that typical software libraries are not written with hard memory constraints in mind. Instead, they usually preserve bounds (*e.g.* procedures that take lists and return lists of the same length). Our solution here is to synthesize symbolic bounds expressed in terms of generic parameters that can later be used when circuit parameters are known in order to instantiate finite-state implementations from infinite-state code. Our support for dynamic allocation and deallocation allows us to handle the forthcoming examples described in this paper whereas existing systems cannot.

Our approach for finding symbolic bounds uses several known methods and tools as sub-procedures, such as shape analysis (*e.g.* [4, 8, 14, 16]) and abstraction methods based on the introduction of new variables (*e.g.* [13, 15]). We also draw influence from the constraint-based invariant generation and rank function synthesis tools (*e.g.* [19, 21]).

2. EXAMPLE

Imagine that we would like to build an `n`-size priority queue that reads integers from an input signal and returns every `n` input integers on an output signal in sorted order. See the function `prio` in Figure 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 Figure 1 is a generic parameter, whereas `i` and `o` should be thought of as signal names (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 Figure 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.

```
void prio(int n,in_signal i,out_signal o) {
  LINK *tmp,*c,*buffer;
  assume( 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);
   }
  }
}
```

Figure 1: Priority queue circuit specification in C, using off-the-shelf implementation of `sorted_insert`.

```
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;
}
```

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

The problem is that `sorted_insert` doesn't guarantee a concrete bound on the amount of heap allocate 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 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 synthesis procedure given later in §3 is designed to find a function f such that it is a program invariant that $f(\mathtt{n})$ is larger than the number of heap cells allocated at any given time. In this case the procedure described later will find the function $f(\mathtt{n}) = \mathtt{n} * 8$, assuming that `sizeof(LINK)` = 8 in the encoding. This symbolic bound is synthesized by our tool in 4.3s.

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]`. See Figure 3 for an example. In this case we assume that `n`= 7.

From this program (and via a translation into VHDL) we then used the Xilinx ISE 10.1.3 tools to construct an implementation for the Virtex-5 XC5VLX110T-2FF1136 FPGA. Using default synthesis and implementation options and with `n` = 7, the generated circuit uses 1,058 slices with the memory implemented in distributed RAM with a critical path of 7.024ns.

Local heaps and custom allocation management. Note that we can use general purpose implementations of `malloc` and `free`, but we may be able to do better. Many Separation Logic [20] based shape analysis tools (*e.g.* [4, 8, 16]) are capable of determining a symbolic representation of the memory footprint of each command in the program. Using this information we can potentially determine that no command beyond those in `prio` and `sorted_insert` access the heap encoded in the shared array `a`. Thus, when `prio` is used in a larger design (as we will do later when discussing Huffman encodings) we are able to make `a` local only to the gates implementing `prio`, thus improving clock speed. In the case of a Xilinx Virtex-5 XC5VLX110T FPGA (which is used on the proto-type RAMP BEE3 board and is the target platform for our experiments), we have 148 36Kb dual-ported memories distributed across the circuit, thus one near the gates implementing an instance of `prio` can be dedicated to the array `a`.

3. BOUND SYNTHESIS

In this section we describe the analysis that automatically synthesizes 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 Figure 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 current separation logic based shape analysis tools [3, 4, 8, 14, 16] (of which we are descendant). Finally, note that our procedure is designed to try and solve an undecidable problem. Thus, like property verification tools for infinite-state systems, our tool must necessarily fail to find bounds in some cases. The ultimate goal can only be to make it succeed in all practical cases.

Our procedure is divided into the following the three steps:

Shape analysis: We use an off-the-shelf shape analysis to determine the shape of the data structures used during the program's execution. As a by-product of the

```
bit a[7 * 8];

void prio_a_7(int n,in_signal i,out_signal o) {
  char tmp,c,buffer;
  assert(n==7);
  while (1) {
    buffer = 0;
    // Build up an n−sized sorted buffer
    for (int k=0;k<n;k++) {
      buffer = sorted_insert_a_7(input(i),buffer);
    }
    // Send the sorted list to the output and
    // deallocate the buffer as we walk it
    c=buffer;
    while(c!=0) {
      output(o,a[c]);
      tmp = c;
      c = a[c+4];
      free(tmp);
    }
  }
}
```

Figure 3: Re-encoding of example from Figure 1 without heap. We assume that the value of parameter `n` = 7. The procedure `sorted_insert_a_7` is like `sorted_insert` but specialized to work over the array `a` instead of the heap.

shape-analysis, we prove the program's memory safety—meaning that invalid memory is never dereferenced and allocated memory cells are never leaked.

Instrumentation: Using abstraction techniques from [15] we use the output of the shape analysis to produce a new program without heap that is a sound abstraction with respect to the original program. Thus, bounds synthesized on the abstraction imply bounds for the original program. Note that the new program has variables that range over integers of arbitrary size (*i.e.* they cannot be represented in 32 or 64 bits). These new variables are used to track the sizes of data structures pointed to by stack variables in the original program. We also instrument the abstraction with an extra variable (over unbounded integers) to track the amount of heap currently allocated.

Bounds synthesis: We then apply our constraint-based synthesis approach to find a symbolic bound f on the maximum value of heap cells allocated during the program's execution.

Array conversion 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 perhaps 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 which have a known range at synthesis time allowing some of the upper bits to be pruned.

In the remainder of this section will we define some preliminary terminology and then discuss the above procedures in more technical detail.

Terminology. We represent a program $\mathcal{P}$ as a tuple $\mathcal{P} = (X, \mathcal{L}, \mathcal{T}, l_0)$, consisting of a set X of variables, a set $\mathcal{L}$ of program locations, an initial location $l_0 \in \mathcal{L}$ and a set $\mathcal{T}$ of transitions. Each transition $\tau \in \mathcal{T}$ is a tuple (l, ρ, l') where $l, l' \in \mathcal{L}$ and ρ is constraints over $X \cup X'$ (In this notation, adding ' to any object means that the program variables appearing in the object are renamed by adding ' in their name.). A valuation of variables X is said to be a *state* of the program $\mathcal{P}$. An execution of $\mathcal{P}$ is a sequence of location and state pairs $(l_0, s_0), (l_1, s_1)$... such that l_0 is initial location and for each consecutive pair (l_i, s_i) and (l_{i+1}, s_{i+1}) there is a $(l_i, \rho, l_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$.

An *invariant* at a program location $l \in \mathcal{L}$ is a superset of all reachable states at l, which is represented by an assertion over X. A map Φ of invariants over each program location is *inductive invariant map* if $\Phi(l_0)$ is superset of all initial states and $\forall (l, \rho, l') \in \mathcal{T}.\ \forall (X \cup X').\ \Phi(l) \wedge \rho \Rightarrow \Phi'(l')$.

The variables X of an input program will contain a special variable $\mathcal{H}$ denoting the heap (a partial function from memory addresses to values).

Shape analysis. The first step in our bounds synthesis procedure is to run an off-the-shelf separation logic based shape analysis (*e.g.* [3, 4, 8, 16]) on the program $\mathcal{P}$. Shape analysis 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 is based on symbolic simulation together with abstraction techniques. It begins execution using the initial state and symbolically executes the program to calculate an assertion about the resulting state. Loops are handled by executing the body of the loop multiple times until a fixpoint is found. After each loop iteration, various abstraction heuristics are applied to ensure that the fixpoint calculation converges. These fixpoints are loop invariants: assertions that hold at the beginning of the loop, and that are preserved by each loop iteration.

As an example, consider Figure 1. When performing shape analysis on this code, at the beginning of `prio`, we assume that no data structures are allocated. In other words, the heap is empty. We then enter the infinite `while` loop, and record that `buffer==0`. Next, the shape analysis tool considers the `for` loop. After the first iteration, symbolic execution works out that the heap consists of a single memory cell at address `buffer` whose `next` field is `NULL`. After the second iteration, we can similarly work out that `buffer` points to a linked list of length 2. At this point, abstraction notices the repeated pattern and weakens the assertion to say that `buffer` points to a linked list of arbitrary length. After symbolically executing the loop another time, the heap again consists of a linked list starting at `buffer`: we have reached a fixpoint, and thus have calculated a loop invariant.

Similarly, the loop invariant of the next loop of `prio` is that the heap consists of a singly-linked list from `c` to `NULL`. Hence, when we exit the loop, since `c==NULL`, we known that the heap is empty. Thus, the assertion that the heap is empty is a loop invariant for the outer loop of `prio`.

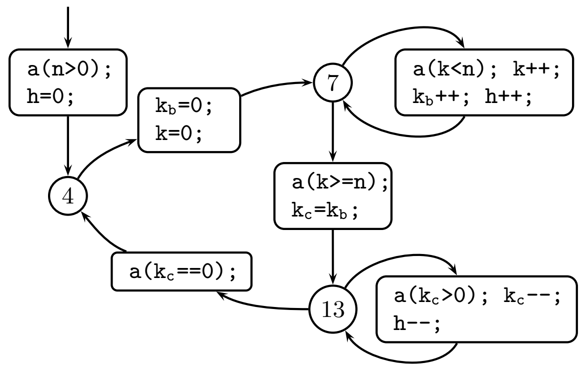

Figure 4: Arithmetic abstraction of procedure `prio` shown from Figure 1. Commands of the form a(e) have special meaning similar to a conditional check: executions through the command in which the Boolean condition e does not hold are simply ignored. In this case the problem of bound synthesis is to find a function f over the input parameters to `prio` such that $f(\mathtt{n}) > \mathtt{h}$ is a program invariant.

Instrumentation. We begin the instrumentation step by introducing a variable `h`, which we increment at calls to `malloc` and decrement at calls to `free`. This keeps track of the amount of memory allocated. (To deal with the case where memory cells of multiple sizes are allocated and deallocated, we introduce one such variable for each size.)

Then, using techniques described in [15], we automatically introduce new variables which soundly track the sizes of data structure shapes inferred by the shape analysis. In the example of the function `prio`, we 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` within `sorted_insert`, the length of that linked list is increased; therefore we increment $\mathtt{k_b}$. Similarly, we introduce another variable $\mathtt{k_c}$ recording the length of the linked list from `c`. At the assignment `c=buffer`, we set $\mathtt{k_c}$=$\mathtt{k_b}$, and at the assignment `c=c->next`, we decrement $\mathtt{k_c}$. Also, when we exit the `while(c!=NULL)` loop, we know that `c==0`, and hence also $\mathtt{k_c}$=0. After adding variables to track the sizes of data structures we abstract the original program—as described in [15]—leaving us with a program that only manipulates arithmetic variables.

Figure 4 shows the control-flow graph (CFG) of 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 occurs between the locations. The transitions between locations come in two forms: assignments v=e and assumption checks a(e). The assumptions prune executions in which the condition e does not hold.

For brevity, calls to the function `sorted_insert` in Figure 4 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.

Bounds synthesis. We call X_p (which are a subset of X) the *parametric variables*. Normally we let X_p be the set of generic parameters to the top-level function (*e.g.* `n` from `prio`), but we are free to make other choices. The family of functions from which we search for a desired bound are expressed over X_p. Lets assume E is set of all arithmetic expressions over X. An arithmetic expression $e \in E$ is associated with each program location, which represents the heap consumption at the program location. A *bound specification* for an arithmetic program $\mathcal{P}$ is a pair $(\mathcal{E}, X_p)$, where $\mathcal{E} : \mathcal{L} \rightarrow E$ and $X_p \subset X$. For a given bound specification, the objective of *bound synthesis* analysis is to find an invariant $\mathcal{E}(l) \leq f(X_p)$, at each location l.

Our method of synthesizing bounds proceeds as follows:

- An invariant template similar to those described in [19, 21] is assumed with *unknown parameters* at each program location, which we call template map Γ. For example at some program location, we may assume $\alpha_1 x_1 + ... + \alpha_n x_n \leq \alpha \ \wedge \ \beta_1 x_1 + ... + \beta_n x_n \leq \beta$ as a template which has a *structure* of conjunction of two linear inequality with the unknown parameters $\alpha_1, ..., \alpha_n, \alpha, \beta_1, ..., \beta_n, \beta$ and over the variables $x_i \in X$. We also create extra templates using bound specification $(\mathcal{E}, X_p)$. For each location l, we construct a *bound template* $\mathcal{E}(l) \leq \gamma_1 x_1 + ... + \gamma_m x_m + \gamma$ where $\gamma_1, ..., \gamma_m, \gamma$ are unknown parameters and $x_i \in X_p$. We generate a new template map Γ_b by conjoining Γ and bound templates, i.e., $\Gamma_b(l) = \mathcal{E}(l) \leq \gamma_1 x_1 + ... + \gamma_m x_m + \gamma \wedge \Gamma(l)$. An invariant can be produced by choosing a value of unknown parameters. The values of the parameters of bound template determines the bound expressed over parametric variables.

- This template map Γ_b is used to encode inductive invariant constraints by replacing invariant map Φ with Γ_b, i.e., $\forall (l, \rho, l') \in \mathcal{T}.\ \forall (X \cup X').\ \ \Gamma_b(l) \wedge \rho \Rightarrow \Gamma'_b(l')$. Only the parameters of templates are not quantified in this formula. In result, it is a constraint over the parameters.

- Satisfying assignments to the parameters in the constraint will give an inductive invariant map. A constraint solver is used to find satisfying values. These values of parameters are placed in templates to compute an inductive invariant map.

For our application we assume a conjunction of linear inequality as templates. The choice of the number of conjuncts is specified as an option to our tool. More conjuncts in the template may facilitate the finding of more expressive invariants, at the cost of performance. Thus, the drawback of our template-based technique is that if the bounds for the invariants that are needed to prove the bounds are not expressible in the given template structure then the tool fails. Experimentally we have observed that templates with only 2 conjuncts suffice.

Note that the abstraction's transition relation is formed from constraints expressed only over arithmetic. This restriction and choice of the structure of template allows us to reduces the induction condition $\Gamma_b(l) \wedge \rho \Rightarrow \Gamma'_b(l')$ into polyhedral entailment check using techniques based on Farkas lemma [6] and non-linear constraint solving as described in [9].

Consider, for example, Figure 4. We aim to compute a symbolically expressed bound over `h` using the generic parameters to `prio` that holds at all locations. Note that the variables `i` and `o` have been lost in the process of abstraction, so we do not need to treat them differently in the synthesis process. The bound specification for this example is $(\{4 \rightarrow \mathtt{h}, 7 \rightarrow \mathtt{h}, 13 \rightarrow \mathtt{h}\}, \{\mathtt{n}\})$.

We assume a template map Γ for the set of locations that has structure of conjunction of 2 linear inequalities. For example, we assume

$$\Gamma(7) = \alpha_{\mathtt{n}}\mathtt{n} + \alpha_{\mathtt{h}}\mathtt{h} + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{k_b}}\mathtt{k_b} + \alpha_{\mathtt{k_c}}\mathtt{k_c} \leq \alpha \ \wedge$$
$$\beta_{\mathtt{n}}\mathtt{n} + \beta_{\mathtt{h}}\mathtt{h} + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{k_b}}\mathtt{k_b} + \beta_{\mathtt{k_c}}\mathtt{k_c} \leq \beta$$

as template at location 7. We also construct a bound template $\mathtt{h} \leq \gamma_{\mathtt{n}}\mathtt{n} + \gamma$ at location 7. So, $\Gamma_b(7) = \mathtt{h} \leq \gamma_{\mathtt{n}}\mathtt{n} + \gamma \ \wedge \ \Gamma(7)$, which has the set of parameters $\{\alpha_{\mathtt{n}}, \alpha_{\mathtt{h}}, \alpha_{\mathtt{k}}, \alpha_{\mathtt{k_b}}, \alpha_{\mathtt{k_c}}, \alpha, \beta_{\mathtt{n}}, \beta_{\mathtt{h}}, \beta_{\mathtt{k}}, \beta_{\mathtt{k_b}}, \beta_{\mathtt{k_c}}, \beta, \gamma_{\mathtt{n}}, \gamma\}$.

In the second step, we build the constraint. Lets pick the loop at location 7 to demonstrate the constraints. The transition relation ρ for the loop is $(\mathtt{k} < \mathtt{n} \wedge \mathtt{n}' = \mathtt{n} \wedge \mathtt{h}' = \mathtt{h} + 1 \wedge \mathtt{k}' = \mathtt{k} + 1 \wedge \mathtt{k}'_{\mathtt{b}} = \mathtt{k_b} + 1 \wedge \mathtt{k}'_{\mathtt{c}} = \mathtt{k_c})$. (Primed version of variables represents the value of the variable after the execution of the transition.) The constraint for inductive invariant for this loop transition is $\forall (X \cup X')$. $\Gamma_b(7) \wedge \rho \Rightarrow \Gamma'_b(7)$, whose expanded form is

$$\forall \{\mathtt{n}, \mathtt{h}, \mathtt{k}, \mathtt{k_b}, \mathtt{k_c}, \mathtt{n}', \mathtt{h}', \mathtt{k}', \mathtt{k}'_{\mathtt{b}}, \mathtt{k}'_{\mathtt{c}}\}.$$
$$(\ \mathtt{h} \leq \gamma_{\mathtt{n}}\mathtt{n} + \gamma \ \wedge \ \alpha_{\mathtt{n}}\mathtt{n} + \alpha_{\mathtt{h}}\mathtt{h} + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{k_b}}\mathtt{k_b} + \alpha_{\mathtt{k_c}}\mathtt{k_c} \leq \alpha \ \wedge$$
$$\beta_{\mathtt{n}}\mathtt{n} + \beta_{\mathtt{h}}\mathtt{h} + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{k_b}}\mathtt{k_b} + \beta_{\mathtt{k_c}}\mathtt{k_c} \leq \beta) \wedge (\mathtt{k} < \mathtt{n} \wedge \mathtt{n}' = \mathtt{n} \ \wedge$$
$$\mathtt{h}' = \mathtt{h} + 1 \wedge \mathtt{k}' = \mathtt{k} + 1 \wedge \mathtt{k}'_{\mathtt{b}} = \mathtt{k_b} + 1 \wedge \mathtt{k}'_{\mathtt{c}} = \mathtt{k_c}) \Rightarrow$$
$$\mathtt{h}' \leq \gamma_{\mathtt{n}}\mathtt{n}' + \gamma \ \wedge \ \alpha_{\mathtt{n}}\mathtt{n}' + \alpha_{\mathtt{h}}\mathtt{h}' + \alpha_{\mathtt{k}}\mathtt{k}' + \alpha_{\mathtt{k_b}}\mathtt{k}'_{\mathtt{b}} + \alpha_{\mathtt{k_c}}\mathtt{k}'_{\mathtt{c}} \leq \alpha \ \wedge$$
$$\beta_{\mathtt{n}}\mathtt{n}' + \beta_{\mathtt{h}}\mathtt{h}' + \beta_{\mathtt{k}}\mathtt{k}' + \beta_{\mathtt{k_b}}\mathtt{k}'_{\mathtt{b}} + \beta_{\mathtt{k_c}}\mathtt{k}'_{\mathtt{c}} \leq \beta.$$

We build such constraints for all transitions using templates at all locations. The satisfiability of the conjunction is checked using a constraint solver. One possible solution for the parameters of the template at location 7 is

$$\alpha_{\mathtt{n}} = \alpha_{\mathtt{k}} = \alpha_{\mathtt{k_c}} = \alpha = \beta_{\mathtt{n}} = \beta_{\mathtt{h}} = \beta_{\mathtt{k_c}} = \beta = \gamma = 0;$$
$$\gamma_{\mathtt{n}} = \alpha_{\mathtt{h}} = \beta_{\mathtt{k_b}} = 1; \qquad \alpha_{\mathtt{k_b}} = \beta_{\mathtt{k}} = -1.$$

Placing above values in the template $\Gamma_b(7)$ results invariant $\mathtt{h} \leq \mathtt{n} \wedge \mathtt{h} - \mathtt{k_b} \leq 0 \wedge \mathtt{k_b} - \mathtt{k} \leq 0$ at location 7. This technique finds following invariant map:

$$\Gamma_b(4) = \ \mathtt{h} \leq 0 \wedge (\mathtt{h} \leq \mathtt{n})$$
$$\Gamma_b(7) = \ \mathtt{h} \leq \mathtt{n} \wedge (\mathtt{h} \leq \mathtt{k_b} \wedge \mathtt{k_b} \leq \mathtt{k})$$
$$\Gamma_b(13) = \ \mathtt{h} \leq \mathtt{n} \wedge (\mathtt{h} \leq \mathtt{k_c})$$

which implies a bound over `h` at locations 4, 7, and 13 as 0, `n`, and `n` respectively. Note that the tightest bound for `h` is found for each location. The other computed invariants are discovered during the bound search.

Note that algorithms designed to find inductive invariants can in cases find trivial assignments—*e.g.* in the context of templates the solver could assign the value 0 to all of the parameters resulting in the trivial invariant *true.* We avoid this problem here as our bound templates instantiate the coefficient to `h` to be 1, thus forcing the solver to find non-trivial solutions.

Array conversion and synthesis. Having computed a symbolic bound for the number of dynamically allocated memory cells, when instantiating a parametrized circuit with concrete values, we introduce a fixed shared array `a` whose size is the computed bound of the dynamically allocated memory cells. Dynamically allocated pointers are represented as offsets into this array. Field differences are converted to the respective array operations. For example, `c = c->next` (from Figure 1) becomes `c = a[c+4]` in Figure 3.

To implement `malloc` and `free`, we can use a singly-linked free list. We introduce shared variable `m` containing the offset of the first free location, which contains the offset of the next free location, and so on. We initialize the array so that the free list contains all the available memory cells. Calls to `malloc()` just pop an element off the free list. In other words, `x = malloc()` is implemented as `x = m; m = a[m]`. Note that we do not need to check whether the free list empty because the bound synthesis guarantees that it will always be non-empty. Similarly, `free(x)` pushes `x` onto the free list: i.e., `a[x] = m; m = x`.

4. ADDITIONAL EXAMPLES

In this section we discuss some of the other example circuits that we have synthesized from heap-based C programs. Note that, as no previously known tool supports these programs, a comparative evaluation is not possible.

Huffman encoder. Consider a an implementation of Huffman encoding using binary trees: The component of a Huffman encoder that builds up the code-tree is naturally expressed as a loop which takes two items out of a priority queue (as discussed in §2) and replaces it with one combined tree. This is could be written as:

```
huffman_node *n1, *n2 ;
while (l->next != NULL) {
  n1 = remove_node (&l) ;
  n2 = remove_node (&l) ;
  l = sorted_insert(combine_nodes (n1, n2), l);
}
```

There are several data-structures in the Huffman encoding example that are easier to express as dynamic data structures which for a given symbol size N can be automatically transformed into fixed size array-based code:

- The Huffman trees are expressed as a tree data structure which can be automatically transformed into an array representation with $2N - 1$ elements.
- The priority queue is expressed as a list data structure which can be automatically transformed into an array of size N.
- The Huffman codes themselves are variable length bit-strings which are expressed as lists of bits. The overall data-structure has a maximum size N for the worst case code-word of all 0s or all 1s.

These symbolic bounds were synthesized in under 300s (~280s for the shape analysis and instrumentation, and ~1s for the bounds synthesis). The bounds synthesis in this case is for the reason that only a few of branches in the Huffman tree manipulation code actually allocate new memory—the majority simply modify existing heap-based data structures and thus do not have effect on the instrumented variable tracking heap allocation in the abstraction.

SAT learned clauses. We are currently investigating the use of C to gates synthesis to produce hardware versions of compute intensive algorithms typically implemented only in software, such as SAT solvers. However, the average

SAT-solver's use of dynamic data structures and control-oriented computations leads to difficulty when developing custom VHDL or Verilog. Our technique allows us to synthesize hardware implementations of some of these key data structures. Consider, for example, the set of learned clauses. A linked structure is desirable because it allows us to easily append and remove learned clauses as the SAT solver learns from conflicts. Although a linked structure in software does not produce the best performance (C++ implementations typically uses resizable vectors) our technique allows the programmer to think more directly in terms of a list of clauses and then the system can automatically transform the linked-list implementation into a fixed-array version. In the case of learned clauses we can add a simple case-split depending on the size of the list and thus use a hardware implementation in the common case—note that the same source code would be used in the software and hardware versions. In the case of MiniSAT [5], if we add an explicit case split to the clause simplification code then the bound is easily propagated (in 2s), and a symbolic heap usage bound is discovered.

Examples of failure. Our approach for symbolic bounds synthesis can fail in many ways. For example, as mentioned before, 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 lists certainly exists, but our tool cannot work out what this bound is.

5. CONCLUSION

C to gates synthesis aims to bring together the ease of software development with the speed of raw gates. However, current C to gates synthesis systems are lacking support for some important software abstractions, including non-trivial dynamic allocation/deallocation on the heap. This paper has introduced a new method that synthesizes symbolic 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.

6. REFERENCES

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

[2] 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.

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

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

[5] N. Een and N. Sörensson. An extensible SAT-solver. *SAT*, 2003.

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

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

[8] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. *PLDI*, 2007.

[9] A. Gupta, R. Majumdar, and A. Rybalchenko. An efficient invariant generator. *TACAS*, 2009.

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

[11] 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.

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

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

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

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

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

[17] G. De Micheli. Hardware synthesis from C/C++ models. *DATE*, 1999.

[18] 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.

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

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *LICS*, 2002.

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

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

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

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