Designing Hardware with Dynamic Memory Abstraction

Jiri Simsa
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jsimsa@cs.cmu.edu

Satnam Singh
Microsoft Research
7 J J Thompson
Cambridge, CB3 0FB, UK
satnams@microsoft.com

ABSTRACT

Recent progress in program analysis has produced tools that are able to compute upper bounds on the use of dynamic memory. This opens up a space for the use of dynamic memory abstraction in high-level synthesis. In this paper, we explain how to design hardware using C programs with `malloc()` and `free()`. A compilation process is outlined for transforming C programs with heap operations into a hardware description language. As demonstrated by our experiments, this approach is feasible. Further, automatic parallelization of the generated circuits improves by a factor up to 1.9 in terms of clock frequency and a factor up to 2.7 in terms of clock cycles over the previous work.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Automatic Synthesis*

General Terms

Experimentation, Measurement, Performance

Keywords

C to Gates, High-Level Synthesis, Dynamic Memory, Parallel Execution, Bluespec

1. INTRODUCTION

One of the approaches to hardware design and prototyping is to use a high-level programming language to describe the desired functionality and a compiler that synthesizes this description into a circuit. This approach, commonly referred to as high-level synthesis, has been adopted by a number of tools using a variety of high-level programming languages such as C [7], Handel-C [10], Scheme [16], or Haskell [1].

This approach provides for a faster hardware design cycle by automating the process of mapping a high level description to a low level implementation. It also also makes hardware based co-processing more accessible to software engineers that can continue to use familiar software programming languages.

The languages C and C++ have inspired the creation of a number of languages for used in high-level synthesis [2, 3, 5, 7, 9, 13, 17, 18]. The programming abstraction that these C-like languages use is easy to adopt and provides for a smooth transition between software and hardware design. In fact, some C programs can be directly synthesized to gates.

However, until recently, the existing tools had no or very limited support for one of the key features of C – pointers, or more generally, the dynamic memory abstraction. In [4] authors have shown how to use shape analysis [12] and invariant generation [6] to compute an upper bound on the size of dynamic memory ever needed by a C program. Whether this bound is computed by a tool or a designer, it can be used to translate the use of dynamic memory away, for example by replacing the heap with static arrays. Note that this approach works only when such a bound exists.

To demonstrate the practicality of this approach the authors of [4] wrote a simple C to VHDL compiler that for a given C program produces a synthesizable VHDL code. A notable imperfection of this compiler is that it produces VHDL code that executes the original program sequentially. Furthermore, the discussion of the design of such a compiler is missing.

This paper improves on previous work by presenting a description of a compiler that transforms C programs with with heap operations into a hardware description language. In addition to that, the compiler presented here uses sophisticated control and data flow analyses to detect parallelism and includes a back-end for both VHDL and Bluespec [14]. The contribution of this paper thus include: 1) a description of a novel C to hardware description language compiler that supports the use of dynamic memory abstraction, 2) improved performance of generated circuits through the use of a parallel execution model, 3) the use of Bluespec language as an intermediate target in the C to gates synthesis.

2. EXAMPLE

Imagine you would like to design the following system. First the system inputs a sequence of *n symbols* and their *weights*. Then, the system builds a Huffman encoder [8] using the inputted data. Finally, the system enters an infinite loop in which it inputs *queries* and outputs their binary encoding. A software implementation of such a system in C might look similar to the one of Figure 1.

The function `huffman` has five arguments. The first argument represents the number of symbols to be read. The

```
void huffman(
  int n,
  FIFO symbol, FIFO weight, FIFO query,
  FIFO result
) {
    int k, query;
    List *list;
    Tree *node,*node_1,*node_2;

    assert(n>0);

    // Build up an n-sized sorted list
    list = NULL;
    for (int k=0;k<n;k++) {
      node = new_node(deq(symbol),deq(weight));
      list = sorted_insert(node,list);
    }

    // Create Huffman encoder
    while (list->next != 0) {
      node_1 = first(list);
      list = remove_first(list);
      node_2 = first(list);
      list = remove_first(list);
      node = combine_nodes(node_1,node_2);
      list = sorted_insert(node,list);
    }

    // Start answering queries
    while(1) {
      query = deq(query);
      enq(result,encode(list->data,query));
    }
  }
```

Figure 1: SW implementation of Huffman encoder

following three arguments are FIFOs for inputting the symbols, weights, and queries respectively. The last argument is a FIFO for outputting the binary encoding. The interface of these FIFOs consists of two methods – `int deq(FIFO fifo)` and `void enq(FIFO fifo, int data)` – with the standard functionality. For simplicity, the implementation of these FIFOs is omitted.

The function `huffman` starts by repeatedly creating new leaf nodes for the inputted symbols and their weights and sorting these nodes by weight into a list. Second, the encoder is built by repeatedly removing the first two nodes from the list, combining the removed nodes into a new one, and sorting the newly created node back into the list. Finally, an infinite loop is entered. Each iteration of this infinite loop inputs a query, computes a binary encoding of the query, and outputs the result. The implementation of subroutines use in the example of Figure 1 is routine and except for the code artifact presented in Figure 2 the details are omitted.

```
List *remove_first(List *list) {
  List *tmp;

  tmp = list;
  list = list->next;
  free(tmp);
  return list;
}
```

Figure 2: `remove_first()` heap operations

Creating such an implementation is easy and the use of dynamic data structures greatly simplifies this task. In comparison to that, a similar implementation in hardware cannot benefit from the use of dynamic memory abstraction.

```
int remove_first(int list) {
  int tmp;

  tmp = list;
  list = ma_list[list].next;
  fl_list_index--;
  fl_list[fl_list_index] = tmp;
  return list;
}
```

Figure 3: `remove_first()` after transformations

That is, unless the hardware implementation is generated automatically from the software implementation.

In order to generate a hardware implementation from a software implementation that makes use of dynamic memory abstraction, one needs to first compute an upper bound on the dynamic memory ever needed by the software implementation. For the example above, one could compute the bound with an automated tool such as [6,12], or do a back of the envelope analysis to arrive at a bound $n \cdot \mathit{list} + (2n-1) \cdot \mathit{node}$, where *list* is the size of the `struct list` data structure and *node* is the size of the `struct node` data structure. To see that this bound is correct one needs to realize that the list for sorting the initial nodes will never have more than n elements and that the encoder is a binary tree with n leaves and thus will have $2n-1$ nodes.

Next, the bound needs to be made concrete. In our example this corresponds to specifying the maximum size of the alphabet that the Huffman encoder supports, say $n = 256$. Now, the use of dynamic memory abstraction can be removed from the program as follows.

First, global arrays `list ma_list[256]`, `tree ma_tree[511]`, `int fl_list[257]` and `int fl_tree[512]` and scalars `int fl_list_index` and `int fl_tree_index` are introduced. Further, code that initializes the arrays `fl_list[]` and `fl_tree[]` with numbers 1 through k, where k is the length of the respective array, is added.

Second, the statement `node = (Tree *) malloc(sizeof(Tree));` in `new_node` is replaced with `node = fl_tree[fl_tree_index];` followed by `fl_tree_index++;`. Other calls to `malloc()` are processed similarly.

Third, the statement `free(tmp);` in `remove_first()` is replaced with `fl_list_index--;` followed by `fl_list[fl_list_index] = tmp;`. Other calls to `free()` are processed similarly.

Fourth, the statement `node->symbol = symbol;` in `new_node()` is replaced with `ma_tree[node].symbol = symbol`. Other cases of pointer dereferencing are processed similarly.

Finally, the declaration `Tree *node;` in `new_node()` is replaced with `int node;`. Other occurrences of a pointer type are processed similarly. Figure 3 depicts the function of Figure 2 after these transformations.

These transformations can be applied to the source code using a simple C to C compiler. Or, as is the case of our C to HDL compiler, these transformations are an inherent part of the method the generates hardware description language code from the intermediate representation.

After using our compiler to produce the Bluespec code for the example of Figure 1, the Bluespec compiler was used to produce Verilog and the Altera Quartus 9.0 synthesis tools were used to synthesize an FPGA design. This resulted in a circuit with 26,123 Altera's look-up tables, 14,005 registers, 2,784 memory bits and clock frequency of 154MHz. For a sample test bench this circuit achieved latency of 525 clock cycles and throughput of 21 clock cycles.

3. EXPERIMENTAL EVALUATION

This section demonstrates the practicality of the approach presented in this paper and evaluate the efficiency of the hardware designs generated by our compiler. In particular, the designs produced by our compiler are compared to the results produced by the compiler of [4].

For the purpose of the experimental evaluation a number of simple hardware designs was described in C using the dynamic memory abstraction. Our compiler was then used to generate a Bluespec and VHDL versions of these designs. The Bluespec compiler was then used to generate a Verilog version of the designs. Finally, both Xilinx and Altera tools were used to synthesize FPGA designs for the Verilog and VHDL. The performance of the designs was evaluated using a test bench.

In previous work, VHDL has been used the output of the back-end. It turns out that such a choice introduces several problems for the compilation process; for example, data select and unit scheduling. We noticed that these problems can be solved elegantly and efficiently by generating Bluespec code and letting the Bluespec compiler to deal with such problems. Further, there is a nice match between the rule-based scheme of Bluespec and our internal representation of the C program to be synthesized. Finally, Bluespec proved to have a highly tuned Verilog netlist generator which generates Verilog code that passes nicely through synthesis tools. All these points motivated our decision to add Bluespec as a back-end to our compiler.

Next, we describe the examples adopted from [4] that were used in the experimental evaluation.

Huffman Encoder – This is our running example of Figure 1. The design has three inputs and one output. The implementation inputs n symbols and their weights, and builds a Huffman encoder using this data. It then enters an infinite loop in which it inputs a symbol and outputs its encoding. For the purpose of experimental evaluation we chose $n = 10$.

Batched Priority Queue – This example implements a data structure for sorting elements in a batch. The design has one input and one output. The implementation repeatedly inputs n elements, and outputs them in a sorted order. For the purpose of experimental evaluation we chose $n = 10$.

Merger – This example implements a merger of two sorted sequences. The design has two inputs and one output. The implementation repeatedly receives n_1 sorted elements on the first input and n_2 sorted elements on the second input. Using the merge phase of the merge sort algorithm it combines the two sequences into one sorted sequence, which is then outputted. For the purpose of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Packet Sorter – This example implements a simple network element. The design has two inputs and one output. The implementation repeatedly inputs packet data on the first input and packet identifier on the second input. It sorts these packets in a queue by their identifiers, ignoring duplicates, until it fills the queue with n packets. It then outputs the packets in the sorted order. For the sake of experimental evaluation we chose $n = 10$.

Program	Latency	Throughput
Huffman	525 cycles	21 cycles
Prio	198 cycles	3 cycles
Merger	92 cycles	4 cycles
Packet	332 cycles	3 cycles
BST	173 cycles	21 cycles

Table 1: Bluespec back-end

Program	Latency	Throughput
Huffman	813 cycles	30 cycles
Prio	196 cycles	3 cycles
Merger	125 cycles	4 cycles
Packet	353 cycles	3 cycles
BST	166 cycles	21 cycles

Table 2: New VHDL back-end

Program	Latency	Throughput
Huffman	1410 cycles	42 cycles
Prio	343 cycles	5 cycles
Merger	192 cycles	7 cycles
Packet	589 cycles	5 cycles
BST	282 cycles	21 cycles

Table 3: Original VHDL back-end

Binary Search Tree Dictionary – This example implements a data structure for storing a set of elements with a lookup operation. The design has two inputs and one output. The implementation inputs n elements on the first input and builds a binary search tree out of them. This is followed by an infinite loop in which a query is received on the second input and the result of the lookup is produced on the output. For the purpose of experimental evaluation we chose $n = 10$.

For each example, we generated three descriptions in a hardware description language – Bluespec description using our compiler, VHDL description using our compiler, and VHDL description using the original compiler of [4]. The Bluespec descriptions were compiled to Verilog using the Bluespec compiler (version 2008.11.C).

Further, a test bench was written for each example to evaluate the latency and throughput of the generated designs. Tables 1, 2, and 3 give the result obtained by simulating the generated designs using the Bluespec, new VHDL, and old VHDL back-end respectively.

The LATENCY column identifies the number of cycles after which the first value appeared on the output. The THROUGHPUT column identifies the number of cycles after which the second value appeared on the output.

Note that the THROUGHPUT value is much smaller than that of LATENCY not because of pipelining, but because all the examples operate either in a batched mode – Priority Queue, Merger, Packet Sorter – or include an initialization phase – Binary Search Tree Dictionary, Huffman Encoder.

The high level message of the measurements presented in Tables 1, 2, and 3 is twofold. First, the parallel nature

Program	ALUTs	Registers	Mem Bits	Speed
Huffman	26,123	14,005	2,784	154MHz
Prio	4,687	3,459	928	170MHz
Merger	4,356	3,368	1,856	211MHz
Packet	6,942	6,556	1,856	165MHz
BST	10,861	7,484	1,856	196MHz

Table 4: Altera – Bluespec back-end

Program	ALUTs	Registers	Mem Bits	Speed
Huffman	12,723	9,840	3,072	96MHz
Prio	1,824	1,775	1,024	104MHz
Merger	1,724	1,837	2,048	111MHz
Packet	2,922	3,329	2,048	107MHz
BST	1,819	2,243	2,048	150MHz

Table 5: Altera – Original VHDL back-end

of the execution improves the latency of the generated circuits by a factor of 1.7 to 2.7 and the throughput of the generated circuits by a factor of 1 to 2. Second, the circuits generated through the Bluespec back-end perform on examples roughly the same (Prio and BST) or better (Huffman, Merge, Packet) then the circuits generated through the VHDL back-end. This demonstrated the improvement over the previous work and justifies the design decision to introduce a Bluespec back-end.

Finally, the Altera Quartus II 9.0 and Xilinx ISE 11.2 tools were used to synthesize the generated VHDL and Verilog, targeting the Stratix III FPGAs and Virtex-6 FPGAs respectively. Both tool sets produced consistent results and only the result for the Altera tools are presented.

The results for the Bluespec back-end and the original VHDL back-end are shown in Tables 4 and 5 respectively. The ALUTs (Altera's adaptive look-up tables) column indicates the number 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 Mem Bits column indicates how many memory bits were mapped onto embedded memory blocks. The last column shows the maximum speed. In all cases the tools automatically picked the smallest EP3SL50F484C2 FPGA.

Although all of the designs generated using the new VHDL back-end simulate correctly, some of them cause the Xilinx tools to run out of memory and crash and the Altera tools to produce an erroneous empty circuit (which may also be a side effect of memory exhaustion). An alternative formulation of the generated VHDL may side-step this problem, which seems to be related to the size and access patterns used for the array representing the heap. This alternative has not been pursued as the Verilog designs generated through Bluespec proved to be an acceptable alternative.

4. CONCLUSION

This paper furthers the practicality of the use of dynamic memory abstraction in the process of designing hardware. It presents a detailed description of a compiler that inputs a C program and creates a parallel execution model of the program. This model is then used to generate a description in a hardware description language. The compiler currently includes back-ends for VHDL and Bluespec.

The circuits generated by the presented compiler represent an improvement over the previous work – the combined decrease in latency and increase in clock frequency result in 5x improvement. Also, to the best of our knowledge this paper is the first to offer a detailed description of compiling a C program with `malloc()` and `free()` to a hardware description language.

As an on-going work the authors of this paper investigate the use of heap-based program analyses [11, 15] for automated localization and pipelining of hardware designs based on C programs with dynamic memory abstraction. Also, alternatives for the current parallel execution model are considered, that would allow execution of several iterations of a loop at a time.

Acknowledgement.

The authors would like to thank the Bluespec team for providing them with access to the Bluespec compiler and the Xilinx team for help with troubleshooting.

5. REFERENCES

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

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

[3] 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.

[4] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. Accepted to FMCAD 2009. Available on Byron Cook's webpage, 2009.

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

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

[7] 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.

[8] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

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

[10] C. Inc. Handel-C language overview. *Web page* `http://www.celoxica.com`, 2004.

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

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

[13] 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.

[14] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

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

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

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

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