On improving the HLS compatibility of large C/C++ code regions

33rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2025)

May 4-7, Fayetteville, Arkansas, USA

Tiago Santos [1], João Bispo [1], João M. P. Cardoso [1], James C. Hoe [2]

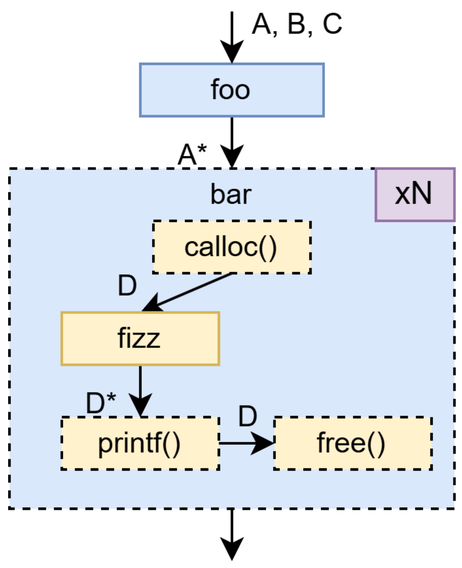
[1] Faculty of Engineering of the University of Porto and INESC-TEC, Porto, Portugal

[2] Carnegie Mellon University, Pittsburgh, USA

[1] {tiagolascasas, jbispo, jmpc}@fe.up.pt, [2] jhoe@andrew.cmu.edu

Context & Motivation

C/C++ applications can be accelerated, on a hybrid CPU-FPGA system, by offloading code regions onto the FPGA via High-level Synthesis (HLS) tools. Although modern FPGA accelerators can hold increasingly large and complex designs, the size and variety of potential code regions remains constrained by the limitations of synthesis tools (e.g., no support for dynamic memory allocation, and system calls).

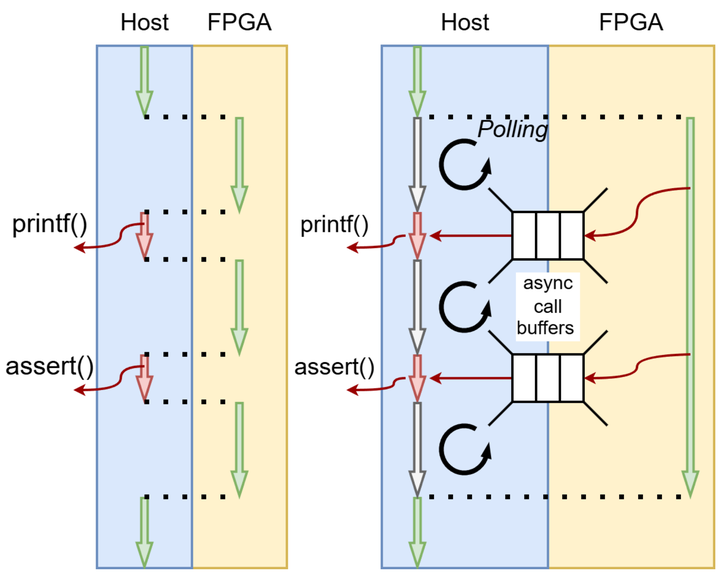


Therefore, to take advantage of modern accelerators, and to enable novel hardware/software partitioning algorithms, we need to remove constraints and expand code regions for being evaluated as a possible hardware partition

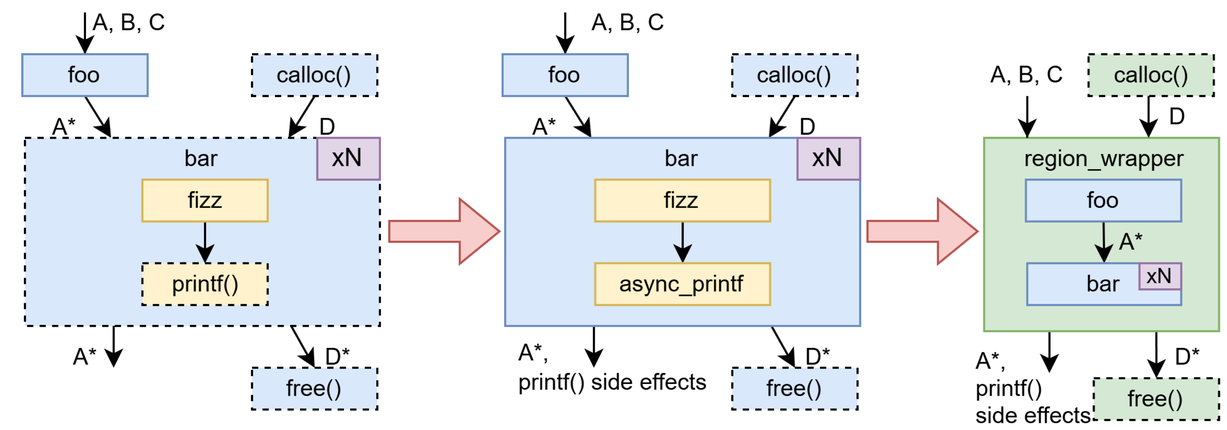
Our Approach

We propose a set of automated source-to-source code transformations to address synthesizability issues, implemented as part of the Clava C/C++ source-to-source compiler

- Struct flattening, to address the presence of indirect pointers in struct fields and to expose memory allocations
- Array flattening, to bring all N-dimensional arrays to 1D, improve AST-level analysis of array indexes, and backend compatibility
- Hoisting of `malloc()`, `calloc()` and `free()` up the call hierarchy, to remove memory de/allocations out of the region (subjected to having no reallocations)
- Converting dynamic allocations to static arrays when the size is statically known (subjected to FPGA resource usage)
- Specializing user-space C library functions to remove variadic arguments (e.g., `sscanf()`) and function pointers (e.g., `qsort()`)
- Implementing calls to kernel-space functions with unused or nonexistent return values (e.g., `printf()`, `exit()`) through asynchronous calls from the FPGA to the host, through a fire-and-forget mechanism



Example: expanding a cluster by making tasks synthesizable



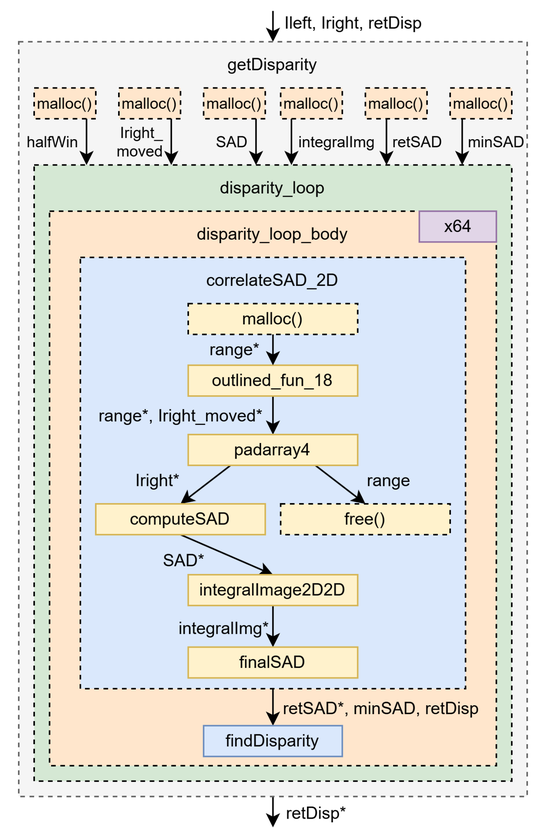
We can increase the cluster of synthesizable tasks by hoisting calls to calloc() and free(), and by implementing the call to printf() as an asynchronous host call

Experimental Case Studies

Benchmark	Variant	Description	#Statements	#Tasks	Critical Path Length
disparity	D1	5-task sequence	153	5	5
	D2	findDisparity	27	1	1
	D3	disparity_loop	217	11	11
	D4	disparity_loop + retSAD static	218	11	11
spam-filter	S1	no async calls	79	7	7
	S2	printf() + assert()	83	7	10
llama2	L1	llama_iteration	650	35	22
	L2	llama_loop (async disabled)	804	48	34
	L3	llama_loop	808	48	37

1. Disparity Map Application from CortexSuite

- Original synthesizable region is limited by a `malloc()` and `free()`
- Hoisting `malloc()` and `free()` enables entire region up to disparity_loop to be synthesized
- getDisparity can also be added to the cluster by hoisting some `malloc()` calls, and turning others into static arrays (converting one `malloc()` causes BRAM usage to increase from 0.22% to 28.40% of total available BRAMs)



2. Debugging Use Case

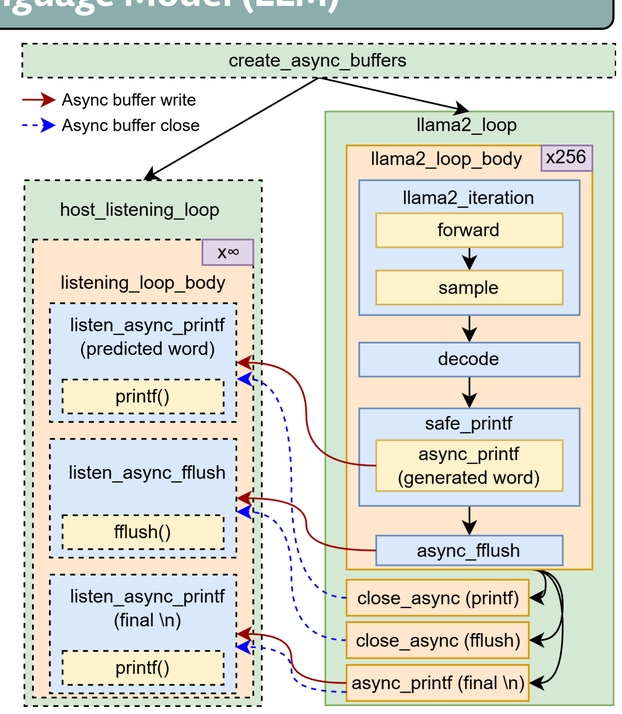
```
void SgdLR_sw(...,
  int8_t *printf_buf,
  async_kernel_info *printf_info,
  int8_t *assert0_buf,
  async_kernel_info *assert0_info,
  int8_t *assert1_buf,
  async_kernel_info *assert1_info) {

  for (epoch = 0; epoch < EPOCHS; epoch++) {
    for (id = 0; id < N; id++) {
      // main computation
    }
    int64_t printf_args[1] = {epoch};
    async_printf(printf_buf, printf_info, false,
                                    printf_args, 1);
  }
  close_async(printf_info);
  bool c0 = fabs(theta[0] - THETA0) <= ERR;
  bool c1 = fabs(theta[1023] - THETA1023) <= ERR;
  async_assert(assert0_buf, assert0_info, true, c0);
  async_assert(assert1_buf, assert1_info, true, c1);
}
```

- We use the *spam-filter* application from Rosetta, which is already 100% synthesizable
- Uses reference implementation using AMD's Xilinx Runtime (XRT) targeting a ZCU102 CPU-FPGA embedded system
- Negligible impact on latency when measured on the board

3. *llama2* Large Language Model (LLM)

- *llama2* alternates between predicting a word and printing it by calling `printf()` and `fflush()`, limiting the region to a recurrent kernel that only does prediction
- By replacing them with asynchronous calls, we can have a single kernel that predicts all words in one invocation, without incurring in additional communication overheads (at the small cost of only 4 additional BRAMs and 2% more FFs and LUTs)



Tiago Santos was supported by the CMU Portugal Visiting Students Program and by Fundação para a Ciência e Tecnologia (FCT) PhD scholarship 10.54499/2021.07324.BD