Timed Hardware Computation

Proposal

MATTHEW AHRENS, Tufts University, USA

Additional Key Words and Phrases: DSL,Embedded Systems

The domain. Two imperative functions define many small hardware programs: a setup and a loop body. Consider the example program in figure 1. On a device that supports a hardware interface such as GPIO (General Purpose Input Output), this program initializes pins and allocates memory during setup, and it reads data from hardware input, performs data transformations, and writes data to hardware output during the loop body.

The current practice. These hardware inputs and outputs often have timing constraints. A timing constraint could look like a specified rate on input to avoid appearing laggy, a specified rate on output to avoid burning out hardware actuators, or restricting intermediate computation to avoid both dropping data and redundantly consuming data. The example program in figure 2 shows what a programmer would write to satisfiy timing constrains where an input needs to run every 500 milliseconds and an output needs to run every 2500 milliseconds.

The problem. In the last example program, the constraint on the output can only be seen indirectly from the constraint on the input and the implementation of the loop – e.g. 5 * 500. The constraint of 2500 milliseconds itself does not manifest in the program. Also, this magic number 5 appears in both the control flow and memory allocation; where did this magic number come from? The programmers solved for it given the constraints on the hardware input and output. If the programmers did this calculation wrong, the hardware might not run at the expected rate, and the data in the buffer might be read from redundantly or overwritten without ever being read. For simple programs, solving these constraints appears trivial, but complexity grows as programs contain data dependencies that are not 1-1. For example, data from multiple inputs could be transformed together to control an output.

Solution. A DSL could assist writing these imperative programs. Given information about how often hardware should run, the DSL performs a static analysis at compile time to ensure that the delays in the program respect the specified rates. To enable this analysis, the programmers must express computation in terms of constant iterations, like the magic number 5, and not variable or dynamic control flow. The DSL will not handle variable nor dynamic rate constraints. In exchange, the DSL will solve for delay and loop constraints for users if they leave "holes" in the program for the compiler to fill in. Lastly, at runtime, the DSL will instrument the programs with special delays that track timing errors in the case runtime code does not allow for the timing constraints the user expects. For example, if a transformation function takes a long time to compute a result, that time will impact how often hardware runs. If the time the transformation function takes is less than the delay statements, however, the program could subtract the time spent from the amount to delay at runtime thus not breaking the constraint.

Author's address: Matthew Ahrens, Computer Science, Tufts University, 161 College Ave, Medford, Ma, 02189, USA, matthew.ahrens@tufts.edu.

2019. 2475-1421/2019/10-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
int inputPin = 1;
int outputPin = 2;

void setup(){
  pinMode(inputPin, READ);
  pinMode(outputPin, WRITE);
}

void loop(){
  int value = read(inputPin);
  write(outputPin);
  delay 500; //milliseconds
}
```

Fig. 1. An imperative C-like setup-loop program that illustrates how programmers program small hardware devices with input, output, and delay primtives.

```
int inputPin = 1;
int outputPin = 2;

int dataBuffer[5]
void setup() {
  pinMode(inputPin, READ);
  pinMode(outputPin, WRITE);
}

void loop(){
  for(int i = 0; i < 5; i++) {
    dataBuffer[i] = read(inputPin);
    delay 500; //milliseconds
  }
  write(outputPin, transform(dataBuffer));
}

int * transform(int[5] buttonPresses); //some transformation function
```

Fig. 2. This program shows that, even for a simple 1-input-1-output scenario, the programmer must solve for magic numbers – e.g. 5 – to implement timing constraints (input: 500ms, output: 2500 ms), and the magic number appears in multiple places (control flow, memory allocation) in the program.