CSPOT Dataflow Prototype

Rich Wolski

October 17, 2022

1 Introduction

The basic idea is to create a set of primitives that can be used to implement a simple data flow language (textually) that can be executed as a script in some scripting language. The code that is in `cspot/apps/dataflow` in the `release-2.0` branch is a small, proof-of-concept prototype. It is not a general dataflow implementation but rather an experiment that shows that CSPOT can implement dataflow computations expressed as directed, acyclic graphs (DAGS).

The following outline assumes that the reader is familiar with CSPOT as it is described in the CSPOT Technical Report available from `https://www.cs.ucsb.edu/research/tech-reports/2018-01`.

The prototype has the following restrictive properties (compared to a complete implementation).

- All DAG nodes (representing computations) have 2 input edges and 1 output edge.
- The only data type that can be conveyed along a DAG edge is a C double-precision floating point number. Thus, the prototype only implements double-precision floating point computations.
- When a node computation is not commutative with respect to its inputs, the left input to the operator corresponding to a node must be flagged as such.
- The output of a program that has been executed is stored at the end of the program WooF (see below). All print statements from within a handler appear in the output of woofc-namespace.
- The 3 CSPOT dataflow primitives are designed to be called from bash as a bash script.
- The prototype only executes a single DAG (there is no iteration or conditional execution).

The directory for the CSPOT dataflow code includes a single bash script – `quadratic.sh` – that computes one of the roots of the quadratic formula as an example.

2 The CSPOT Dataflow Components

The prototype consists of 4 CSPOT software components: 3 CSPOT programs and 1 CSPOT handler.

- `dfinit.c` is a program that initializes 2 WooFs that are needed to execute a program.
- `dfaddnode.c` is a program that appends a dataflow node to a WooF that contains the state of the dataflow program nodes at any time during its execution.
- `dfaddoperand.c` is a program that appends edges to a WooF that records which edges are ready to transmit their carried operand to a destination node.
- `dfhandler.c` is a CSPOT handler invoked when an edge is added to the operand WooF. It executes an algorithm that moves the dataflow program through its next state transition.

The `dfinit` program initializes or reinitializes the WooFs that are necessary to execute a DAG and, thus, must be called before any nodes or edges are added to these WooFs.

3 Writing a CSPOT Dataflow Program

A CSPOT dataflow program, executed as a script, consists of 3 sections that must be executed in order. The script must

- Call `dfinit` in the CSPOT namespace where the WooFs are to be created (i.e. where the program will execute).
- Call `dfaddnode` once for each node in the DAG (syntax discussed below). Note that the order in which nodes are added does not affect program correctness but might affect program performance.
- Call `dfaddoperand` once for each edge in the DAG that carries an input to the program itself. The order in which edges are added does not affect program correctness and likely does not affect program performance.

Using `quadratic.sh` as an example, the code is

Listing 1: quadratic.sh

```
#!/bin/bash

#usage: quadratic.sh a b c
A="$1"
B="$2"
C="$3"

#double-precision arithmetic operator codes supported by prototype
ADD=1
SUB=2
MUL=3
DIV=4
SQR=5

#init woofs with woofname "test"
./dfinit -W test -s 10000

#add nodes
./dfaddnode -W test -o $DIV -i 1 -d 0 # -b + sqr(b^2 - 4ac) / 2a -> result
./dfaddnode -W test -o $MUL -i 2 -d 1 # 2a -> node 1
./dfaddnode -W test -o $MUL -i 3 -d 4 # -1 * b -> node 4
./dfaddnode -W test -o $ADD -i 4 -d 1 -1 # (-1 * b) + sqr(b^2 - 4ac) -> node 1, order 1
./dfaddnode -W test -o $SQR -i 5 -d 4 -1 # sqr(b^2 - 4ac) -> node 4
./dfaddnode -W test -o $SUB -i 6 -d 5 # b^2 - 4ac -> node 5
./dfaddnode -W test -o $MUL -i 7 -d 6 -1 # b^2 -> node 6, order 1
./dfaddnode -W test -o $MUL -i 8 -d 6 # 4ac -> node 6
./dfaddnode -W test -o $MUL -i 9 -d 8 # ac -> node 8

#add program inputs (including constants)
./dfaddoperand -W test -d 7 -V $B
./dfaddoperand -W test -d 7 -V $B
./dfaddoperand -W test -d 8 -V 4.0
./dfaddoperand -W test -d 9 -V $A
./dfaddoperand -W test -d 9 -V $C
./dfaddoperand -W test -d 3 -V -1.0
./dfaddoperand -W test -d 3 -V $B
./dfaddoperand -W test -d 2 -V 2.0
./dfaddoperand -W test -d 2 -V $A
./dfaddoperand -W test -d 5 -V 1.0 # ignored because this is SQR and need two op
```

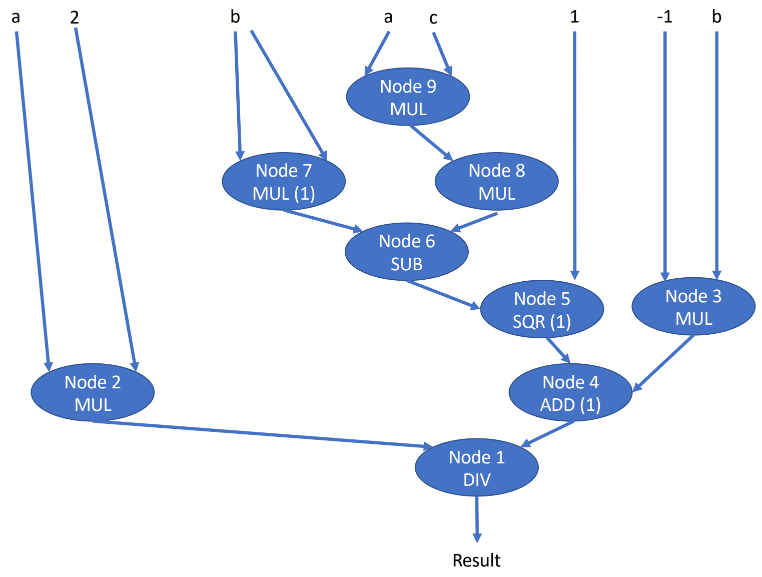

Figure 1: Graphical representation of quadratic formula (the +SQR() root only).

The key-value arguments to `dfinit` are

- `-W <WooF name prefix>` The program will create a WooF with the prefix and `.dfprogram` as an extent and a second WooF with the same prefix and the extent `.dfoperand`. The `.dfprogram` WooF will contain DAG nodes and the `.dfoperand` WooF will contain edge records.
- `-s size` The `size` parameter specifies how many WooF entries should be configured for both WooFs before they wrap around. It needs to be large enough to hold at least $5x$ the number of program nodes in the graph.

The key-value arguments to `dfaddnode` are

- `-W woofname` Gives the prefix for the `.dfprogram` and `.dfoperand` WooFs.
- `-i node-id` Each node in the DAG needs to be given a unique integer identifier which is specified by the `-i` argument.
- `-o operation code` Each operation that can be represented by a node in the DAG is represented by an integer "op code." When a node is "fired" the "op code" indexes a set of functions that perform corresponding functions.
- `-d dest-node-id` After a node has "fired" (its operation has been performed) the `-d` argument indicates the node-id that will receive its output as an input.
- `-1 <optional>` If the node that will receive the output of the this node is not commutative, then make the output the first (left) input of the destination node.

A `dest-node-id` of 0 indicates the result of the computation.

Recall, that `dfoperand` specifies edges that are inputs to the program (i.e. edge that do not have a program node as a source). The key-value arguments for `dfaddoperand` are

- `-W woofname` Gives the prefix for the `.dfprogram` and `.dfoperand` WooFs.
- `-d dest-node-id` The nodes that will receive the edge value
- `-1 <optional>` If the node that will receive the output of the this node is not commutative, then make the output the first (left) input of the destination node.

For example, Figure 1 shows the DAG representation of the textual representation shown in Listing 1. In the figure, inputs to the program (each defined by a `dfaddoperand` command) are shown along the top. For clarity, each input is shown above where it is consumed by a node and not with fan-out. That is, the "a" value and the "b" value are inputs to multiple nodes. Also, the (1) indicates non-commutative input. For example, $Node1$ (which is a division operation) divides its first input (from $Node4$ indicated by (1)) by its second input (from $Node2$).

4 The CSPOT Dataflow Implementation

The code that this section describes is contained in `df.h`, `dfhandler.c`, and `dfoperation.c`. The latter contains the function `dfOperation()` that maps node op codes to double precision arithmetic operations which it then performs and returns.

Each element of the `.dfprogram` WooF has the following C-language structure:

Listing 2: C structure for node WooF

```
#define WAITING (0)
#define CLAIM (1)
#define DONE (2)

struct df_node_stc
{
        int opcode;       /* this op code */
        int ready_count;/* how many have we received */
        double value;     /* value of first operand to arrive */
        int order;        /* is waiting first or second in non-commute */
        int node_no;      /* which node is it */
        int dst_opcode;   /* next node type */
        int dst_no;       /* next node address */
        int dst_order;    /* for non-commutative operations */
        int state;        /* waiting, claimed, done */
};
```

The constants `WAITING`, `CLAIM`, and `DONE` are the values that the `state` field. The program `dfaddnode` initializes and then appends to the program WooF one of these structures for each node in a program. The initial value of `ready_count` is 0 and the initial state is `WAITING`. The put operation in `dfaddnode` that appends each structure does not trigger a handler.

The operand structure is shown in the following listing:

Listing 3: C structure for operand WooF

```
struct df_operand_stc
{
        double value;             /* operand value */
        int dst_no;               /* dest node address */
        int order;                /* for non-commute */
        char prog_woof[1024];     /* name of woof holding the program */
};
```

The program `dfaddoperand` appends one of these structures for each program input to the `.dfoperand` WooF. Each put operation in `dfaddoperand` specifies the handler `dfhandler` which implements dataflow execution cycle.

The `dfhandler` CSPOT handler is invoked on the `dfoperand` WooF every time an operand is appended to the `.dfoperand` WooF. On handler entry, it initializes a node structure for the destination node and the state set to `CLAIM` that it then appends to the program WooF. The purpose of this node is to set the sequence number (held in the `c_seqno` variable) that defines the place in the log that begins a backward scan of the program WooF.

The handler then scans backwards (from `c_seqno-1`) through the program WooF looking for a node that has a `node_id` equal to the `node_id` in the `CLAIM` node (which is the same as the `dest_node_id` in the operand). If it finds a matching node, then the `ready_count` value is either 0 (and the state is `WAITING`) indicating that the node has yet to receive either input or the `ready_count` is 1 (state also `WAITING`) indicating that the node is "partially fired" by having one of its two inputs delivered.

If `ready_count` is 1 (and the `node_id` matches), then the node has already received its first operand and the handler must fire the node. To do so, it takes the value (passed in the operand structure to the handler) and the value stored in the node when it was partially fired passes the two of them to the function `dfOperation()` (possibly ordering the inputs if the flag indicating a non-commutative operation is set) which applies the op code in the node to the two inputs and returns the double precision result. It then calls the `dfFireNode()` function to fire the node and pass its result to its destination node.

The `dfFireNode()` function sets the node's state to `DONE` and its `ready_count` to 2 (since both inputs are present). It puts this "DONE" node into the node WooF largely for debugging purposes since a node marked `DONE` has a `ready_count` of 2 and hence will be skipped by the handler in any node WooF scan.

The `dfFireNode()` function also initializes an operand structure for the result computed by the node, sets the `dst_no` id to be the node id of the destination node that is to receive this result as an input, and appends the new operand to the operand WooF, specifying `dfhandler` in the put (triggering a handler invocation for this operand).

Each append to the operand WooF, then, triggers another handler that moves the destination node through the execution cycle.

The `dfFireNode()` function is only called when the second node input arrives. When the first input arrives, the handler looks for a matching node that has a state set to `CLAIM` and a sequence number (in the program WooF) smaller than `c_seqno`. If it finds such a record, then an "older" claim is underway (another handler is running for the node with `node_id`). The logic is that this older handler invocation should continue and then rescan from the new end of the program WooF looking for the younger `CLAIM` node. That is, if two handlers run for the same node, their respective `CLAIM` records will append to the WooF with one occurring before the other. The first one (the one with a smaller sequence number) will take the `ready_count` from 0 to 1 creating a partially fired node *and then* scan backwards looking for the younger `CLAIM` record. If it finds a second `CLAIM` then the older handler completes the node's firing. Thus, the handler appending the younger `CLAIM` simply exits, leaving the older hander to handle both the partial fire and the complete fire of the node.

Thus, when a partial fire occurs (when the original node WooF scan finds a `ready_count` set to 0) it puts a node to the program WooF with the `ready_count` set to 1 (indicating a partial fire). It then scans the node WooF backwards from the sequence number for the partial fire to the sequence number for the older `CLAIM` node (contained in `c_seqno`). If it finds a another `CLAIM` in the sequence number range between the partial fire and the original `CLAIM` then a second handler executed later and exited. This handler, then fires the node by calling `dfOperation()` to compute the result, and `dfFireNode()` as described above. If a second `CLAIM` node is not found during this scan, then the handler exits and a subsequent handler invocation will find the partial fire in the node WooF and complete the node's firing.

5 Some Thoughts

The prototype demonstrates that it is possible to implement dataflow relatively efficiently using the current definition of CSPOT. The log scan for a node with `ready_count = 0` is $O(n)$ where there are n program nodes, however if the nodes are added to the program WooF from top to bottom, the actual scan is much less. Scans for partially fired nodes are $O(k)$ where k is the make span of the program graph (assuming the DONE nodes are removed). Finally, handlers that are not associated with dependent nodes can execute in parallel. However the prototype is otherwise not terribly useful. There are several extensions that are possible that are likely fruitful areas of study.

One extension is to change the arity of the nodes from 2 to some arbitrary maximum. The handler would need to become more complex since a partially fired node would be any node with at least 1 input available that is not DONE. However, this change would allow a compiler or runtime system to "coalesce" fine-grained nodes into coarser nodes. That is, the prototype does a great deal of work to execute a single double-precision arithmetic operation. A "true" system would bundle multiple such operations together to save the overhead.

A second, related, extension is to allow for arbitrary functional operations (as opposed to double-precision arithmetic). Here the issue is that the operand WooF will need to have elements that are sized by the C union of *all* possible node inputs. There are several ways to implement this extension, but all of them make the simple API (consisting of `dfinit`, `dfaddnode`, and `dfaddoperand`) considerably more complex. Still, this complexity is properly part of the intermediate representation and could be managed by a full-fledged language system.

Another set of extensions is to add iteration and conditionals to the dataflow language. A simple way to do so is to embed this functionality in bash or what ever scripting language is calling the dataflow API. The challenge with this approach is primarily loop-carried dependencies. That is, the output of a node in one iteration may be an input to a node in another and there is not a way in the API to express this kind of dependency.

One way to achieve this functionality is to change the current prototype so that the program nodes for each loop body each go in a new program WooF. Similarly, the start of each loop body would also require a new operand WooF that carries program inputs and also dependencies from previous loop bodies.

Architecturally, the change would likely split the program WooF into a "code" WooF (that contains the program nodes) and a "state" WooF that contains partially fired nodes, CLAIM records, etc. There would be one code WooF for the loop body, but each iteration gets its own (or a reinitialized) state WooF. Each loop body will also require its own operand WooF which would need to be created as soon as a loop body result is generated.

Lastly, the current system is not distributed. That is, the `dfinit` function creates an operand WooF and a program WooF in the same CSPOT namespace. The operand WooF does have the program WooF's URI in it so that nodes could be fired ion other namespaces, but the dataflow API does not have an option for adding a namespace URI to each node. If the API allowed a URI for every destination node, then it would be possible to execute functions remotely from the location of the operand WooF.

Distributing the operand WooFs is slightly more complex. The initial operands would specify target namespaces but the API would need a URI for the output of each node specifying an operand WooF which would then carry a separate URI for the target node.

In short, the model is distributed, but implementing it using the current prototype makes the prototype dataflow API quite unruly due to the need to specify URIs for nodes and operands. Most likely, the approach in the prototype would ultimately result in an IR that a language system or visual programming system would emit. If that is true, than a redesign of the dataflow API (to make it more of an IR) is probably the right thing to do.