Introduction to π-WAM Assembly

Mostowski Collapse[1]

[1] XLOG Technologies AG, Switzerland

August 27, 2026

Abstract. With the π-WAM we ventured into an alternative WAM inside the same Dogelog Player Prolog system. The idea consists of having a more specialized WAM besides the general WAM, that allows the execution on parallel CPU and GPU backends. The work progressed pretty quickly from a very simple first prototype that implemented a reduced WAM based on ideas from the famous Hack computer science educational project. It then quickly progressed from 16-bit instruction set to a 32-bit instruction set that integrates into the Dogelog Player surface compilation of Prolog code. The parallel CPU backend already proves that we can leave the MLips regime and enter the GLips regime.

Keywords: Prolog, Concurrent Logic Programming, Turso, Polar

1 Introduction

The report starts with 3 chapters that basically verify our performance claim. These chapters also show that we use a development discipline, where a Prolog emulator is developed as a specification to the later host language native backend executors implementations. Although the programming method is entirely manual.

The last two chapters address more advanced compilation techniques and how they can be realized via 100% Prolog code, especially the usage of Prolog logical variables for labels and gotos. Interestingly we arrive at a welcome integration with the Dogelog Player transform, allowing us to delegate the complications of local cuts.

2 Emulating π-WAM in Dogelog Player

2.1 Overview

We are currently setting sail to give Dogelog Player a more Janus faced backend. Until now we had only one code compilation idea that realizes our Prolog VM. With π-WAM we add a second Prolog VM to the same Prolog system, with the aim to use it for specialized tasks.

2.2 The Machine

The concept of a π-WAM embraces a fusion of a processs (π) calculus a Warren Abstract Machine (WAM). Optimized for speed the π-WAM is very primitive and currently only supports the '$SEQ'/2 control construct, plus rudimentary 32-bit integer arithmethic and between/3.

The machine itself is a variant of Hack by Nisan and Schocken. While Hack has A and C instructions, we always combine an A and C instruction into a combo. A π-WAM AC instruction has an action field, an object field from A and a relative jump field from C:

```
INSTR = (ACT, OBJ, REL)
```

By simply setting REL = 0 the π-WAM will go from the present instruction to the next instruction. Otherwise when REL > 0 or REL < 0 we can let any action continue at relatively computed new program adress. Using relative jumps means also code blocks can be moved around.

2.3 The Compiler

The compiler capitalizes that code blocks are relocatable. We adopt a continuation passing style compilation and pass a code block as a continuation to the compilation. The compilation is written in Prolog itself. Given a goal GOAL and a code block CODE, we can use the notation:

```
CODE' := [[ GOAL | CODE ]]
```

To indicate that CODE' resulted from compiling GOAL with a continuation CODE. The compiler uses Prolog lists to represent code blocks, and utilizes Prolog DCG to emit code blocks. The rudimentary 32-bit integer arithmetic maps straight away to Hack ALU and conditional jumps.

```
[[ (GOAL, GOAL2) | CODE ]] := [[ GOAL | [[ GOAL2 | CODE ]] ]]
```

The above equation shows how we compile $SEQ/2, one could say it's just the associativity law. A major challenge was between/3, but we viewed it as Logical Loops as described by Schimpf, so that we implement it without any choice points and basically as a for loop.

2.4 The Emulator

To test the compiler, we devised an emulator that will execute an abstract π-WAM, again written in Prolog itself. We made a further refinement of the instruction set, again inspired by Hack, and let the action field itself be structured into function, mode and condition as follows:

```
ACT = (FUN, MODE, COND)
```

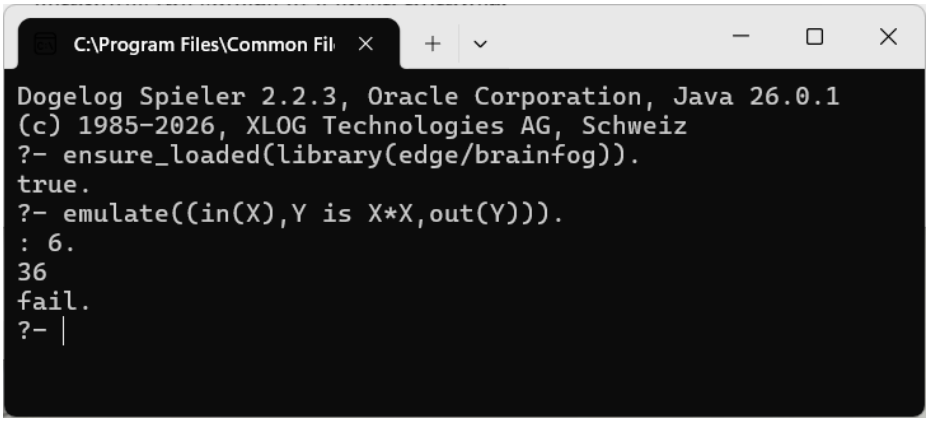
The π-WAM does not only do without choice points, it has also no stack and no trail. The latter two are both superfluous since for the rudimentary 32-bit integer arithmetic the compiler will simply allocate more registers and registers are also used as Prolog variables and just overwritten.

```
sys_run(PC, CODE, ACCU, STATE) :-
   nth0(PC, CODE, INSTR),
   INSTR = (ACT, OBJ, REL),
   ACT = (FUN, MODE, COND),
   sys_run_get(MODE, OBJ, STATE, VALUE),
   sys_run_fun(FUN, ACCU, VALUE, ACCU2),
   sys_run_set(MODE, OBJ, STATE, ACCU2, STATE2),
   sys_run_jump(COND, ACCU2, REL, REL2),
   PC2 is PC+1+REL2,
   sys_run(PC2, CODE, ACCU2, STATE2).
```

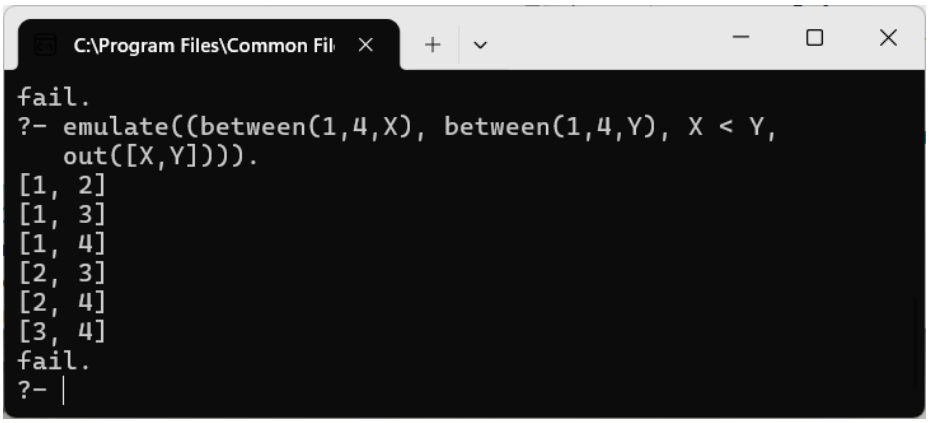
The above shows the emulator main routine. The parameters are program counter, code block, accumulator and state, i.e. the array of registers. No single fetch and execution logic dispatch as in the usual WAM is found. The execution logic is distributed across get, fun, set and jump.

2.5 The Results

The emulator dispatches π-WAM in/1 and out/1 commands to the usual Prolog read and write predicates. This can be used to test the emulator beyond running a statically given goal. Here is a simple utility that computes the square of a given number:



As unorthodox as our machine may sound, it shares many threats with the famous Intel 4004 from 1971. The first commercial CPU developed for electronic calculators, but we want to show more. Here is a backtracking example with two between/3, giving an idea what the π-WAM can do:



2.6 Summary

Optimized for speed the π-WAM is very primitive. The compiler capitalizes that code blocks are relocatable. The emulator dispatches π-WAM in/1 and out/1 commands to the usual Prolog read and write predicates. We could demonstrate a is/2 example and a between/3 example.

3 Executing π-WAM with Dogelog Player

3.1 Overview

A π-WAM combines ideas from π-calculus and Warren Abstract Machine. We already made a compiler and emulator, 100% written in Prolog, and we could demonstrate is/2 and between/3. In this post we introduce executors, written in the Dogelog Player target host languages.

3.2 Machine Code

The π-WAM executor is basically a CPU backend, that runs the Hack machine code via a host language written machine code interpreter. The JIT-ing of the host language will automatically translate the interpreter into native code, but not the π-WAM machine code itself.

The machine code is run first by having the abstract machine code itself packed into concrete 32-bit words. This is done in Prolog itself, before the code is passed to the machine code interpreter. Packing is done by coding enumerates as integer values. Here is an example:

```
% sys_pack_mode(+Atom, -Integer)
sys_pack_mode(get, 0).
sys_pack_mode(cst, 1).
sys_pack_mode(set, 2).
sys_pack_mode(out, 4).
sys_pack_mode(in, 5).
```

Since Prolog systems usually feature first argument indexing, the coding of enumerates via facts in the above argument layout will typically capitalize on indexing. Packing for the π-WAM is further done by combining multiple integer values, into a single integer value. Here is an example:

```
% sys_pack_act(+Triple, -Integer)
sys_pack_act((FUN, MODE, COND), ACT) :-
   sys_pack_fun(FUN, VAL),
   sys_pack_mode(MODE, VAL2),
   sys_pack_jump(COND, VAL3),
   ACT is (VAL<<8) \/ (VAL2<<4) \/ VAL3.
```

As can be seen we make use of the bitwise operation (<<)/2 shift left and the bitwise operation (\/)/2 bitwise or. These arithmetic functions are usually found in Prolog systems that implements the ISO core standard. So, generating machine code for the π-WAM is again relatively portable.

3.3 Virtual Machine

Since the π-WAM machine code interpreter is not a bare matal thing, it is custom to call it a virtual machine. The virtual machine can be relatively straight forward derived from the Prolog written emulator. While Prolog uses sys_run/4 tail recursively, the virtual machine has a while loop:

```
private static void run(PiWam pi) {
    int pc = 0;
    int accu = 0;
    while (pc < pi.code.length) {
        int instr = pi.code[pc++];
        int value = run_get(pi, instr);
        accu = run_fun(instr, accu, value);
        run_set(pi, instr, accu);
        pc += run_jump(instr, accu);
    }
}
```

We use a class PiWam to encapsulate the code and the registers. What was dispatching by enumerates in the emulator is again straight forward to translate. We only have to use the integer values from the instruction packing and can then switch along these values:

```
private static int run_get(PiWam pi, int instr) {
    int act = instr>>20;
    int obj = ((instr>>10) & 0x03FF) - 512;
    switch ((act>>4) & 0x0F) {
        case 0: /* get */
            return pi.state[obj];
        case 1: /* cst */
            return obj;
        case 2: /* set */
            return 0;
        case 4: /* out */
            return 0;
```

```
        case 5: /* in */
            System.out.print(": ");
            System.out.flush();
            return readBig().intValue();
        default:
            throw new IllegalArgumentException("illegal mode");
    }
}
```

The unpacking of machine code is again done by bitwise operations. Now in real-time while the virtual machine executes, by means of the bitwise operations from the host language. To support negative OBJ respective REL values, we subtract 512, which was added while packing.

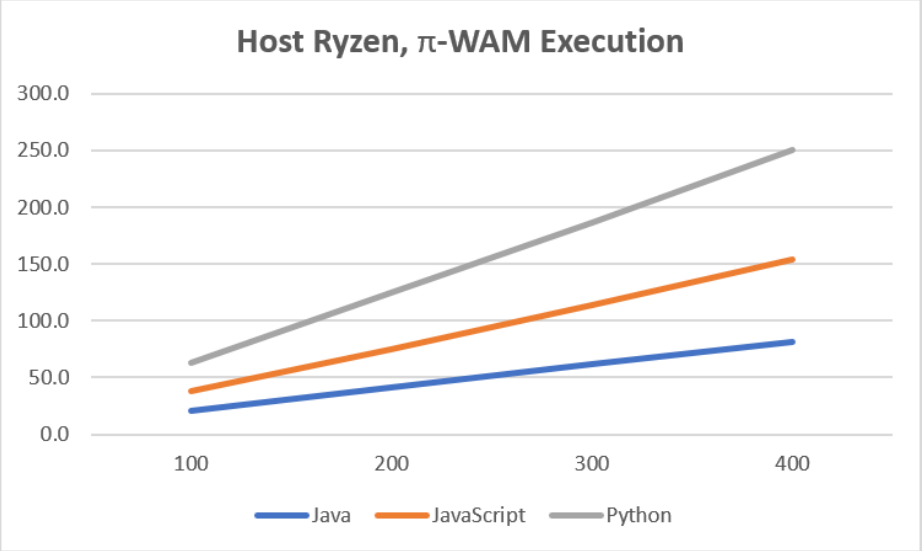
3.4 Host Languages

The Java host language offers the fasted realization of a π-WAM virtual machine. The test case is seen below; it is run for N=100,200,300,400 where the execute/1 predicate is from the library(edge/brainfog). The predicate does the π-WAM compile and then the CPU backend execute:

```
:- ensure_loaded(library(edge/brainfog)).

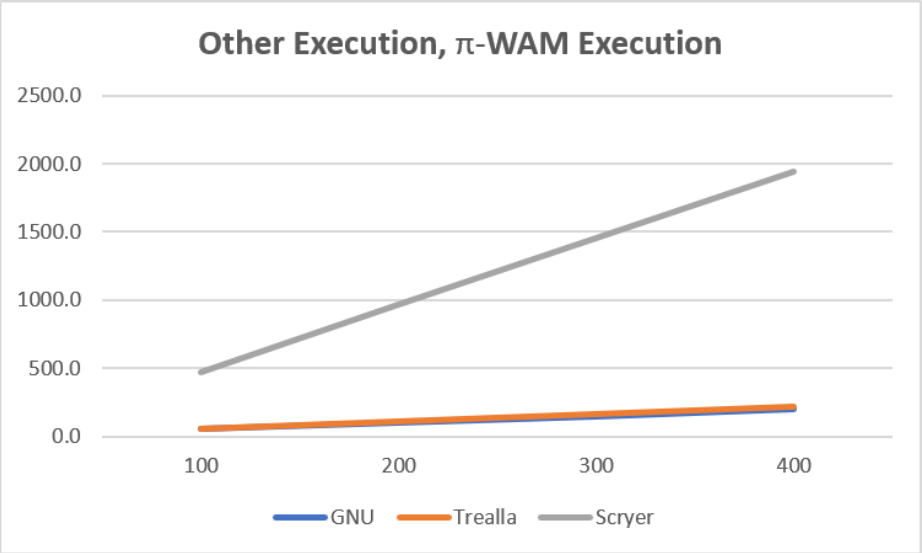
test(N) :-
   execute((between(1,N,X), between(1,100,Y), A is X+Y,
                between(1,100,Z), _ is A*Z))).
```

Our benchmarking then showed that the ratio between Java and JavaScript based π-WAM virtual machine execution involves a slowdown of factor 2, while the Python based π-WAM shows a slowdown of factor 3. The following diagram shows our finding again:



3.5 Other Prologs

We were then curious how ordinary Prolog systems execute the same test case. It turns out that not only the Java π-WAM virtual machine beats GNU Prolog, also already the JavaScript π-WAM virtual machine with 154 ms for N=400 is 50% faster than GNU Prolog with 200 ms:



Similar results can be obtained when testing against Trealla Prolog, since the present test case has close outcomes for GNU Prolog and Trealla Prolog. An outlier is Scryer

Prolog, we don’t know the culprit, but the 2000 ms are more than 10x time slower than the Java π-WAM virtual machine.

3.6 Summary

On the Prolog side we added packing the machine code into 32-bit instructions. On the host targets side, for Java, JavaScript and Python, we added a π-WAM virtual machine. Interestingly the Java based machine can execute our testcase in 80 ms, while GNU Prolog takes 200 ms.

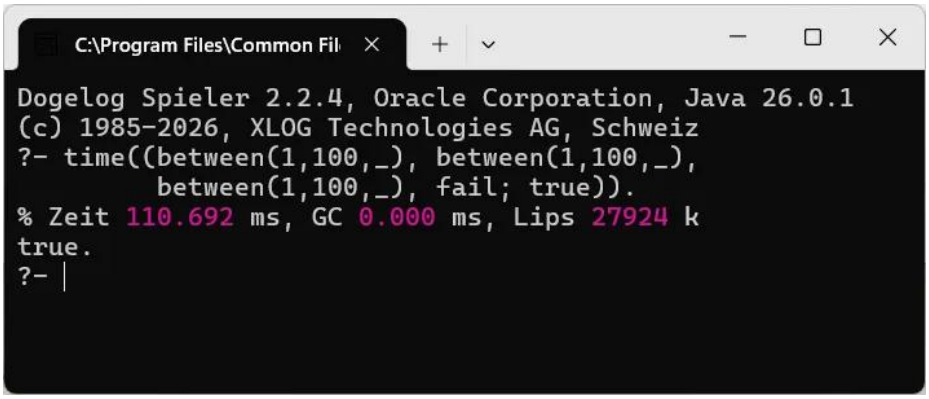
4 Parallel π-WAM: 1.7 Giga Lips on a CPU

4.1 Overview

We recently implemented a parallel π-WAM on a GPU backend and could demonstrate an estimated 11.4 Giga Lips. In this post we report a further experiment, this time presenting a parallel π-WAM on a CPU backend, that can lift specialized Prolog, currently to 1.7 Giga Lips performance.

4.2 Single Baseline

Let's recall the single-threaded baseline before we begin. Its a simple Prolog program that does a triple loop. In the original Dogelog Player VM, we can execute it in around 100 ms, and it shows us ca. 27 MLips. We used this baseline already in the GPU estimate:



For the CPU estimate we can first hook into our new library(edge/brainfog) introduced with Dogelog Player 2.2.3, which already provides a single- threaded CPU backend. We run the same CPU backend from there in an isolated class Laika with the following harness:

```
vm.state = new int[]{0,0,0};
for (int j = 0; j < 3; j++) {
    long tm = System.currentTimeMillis();
    for (int i = 0; i < 100; i++) {
        vm.run();
    }
    System.out.println(vm.state[2] + " /* "+
            (double) (System.currentTimeMillis() - tm) / 100 +
            " ms */");
}
```

The harness does a little bit more than what we did in our GPU test, since single-threaded is much faster in CPU than in GPU. So we average out 100 runs, and show 3 of them. By the results of ca. 10 ms we see that π-WAM is ca. 10x times faster than the original Dogelog Player VM:



4.3 Memory Organization

To move to a multi-threaded solution, we adopted our GPU design, in that we used a flat memory organization. The state of the multiple Hack VM is just a stretched array where each Hack VM occupies an index range. The modified Laika2 class reads as follows:

```
public class Laika2 implements Runnable {
    public int[] code;
    public int[] state;
    public int offset;

    public void run() {
        run(offset);
    }

    public void run(int offset) {
        int pc = 0;
        int accu = 0;
        while (pc < code.length) {
            int instr = code[pc++];
            int value = run_get(instr, offset);
            accu = run_fun(instr, accu, value);
            run_set(instr, accu, offset);
```

```
            pc += run_jump(instr, accu);
        }
    }
```

We then launch platform threads, equaling the number of desired Hack VM instances. The default thread size for a platform thread is usually 1 MB on 64-bit JVMs, so that running 128 instances should not cause an out of memory error. The launcher now first allocates the Hack VMs:

```
int NUM_THREADS = 128;
int[] state = new int[3 * NUM_THREADS];
Laika2[] vms = new Laika2[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++) {
    Laika2 vm = new Laika2();
    vm.code = code;
    vm.state = state;
    vm.offset = 3 * i;
    vms[i] = vm;
}
```

Since in this model the number of threads equals the number of Hack VMs, the next step is to launch 128 different threads, and then wait for their execution to finish. This replicates the roundtrip measurement we had in the case of our GPU testing, not backported to the CPU:

```
for (int j = 0; j < 3; j++) {
    long tm = System.currentTimeMillis();
    Thread[] threads = new Thread[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        Thread thread = new Thread(vms[i]);
        threads[i] = thread;
        thread.start();
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        Thread thread = threads[i];
        thread.join();
    }
    System.out.println(/* join(state, ", ") + */ " /* "+
            (double) (System.currentTimeMillis() - tm) + " ms */");
}
```

Our laptop, a Ryzen AI 7 350 has 8 physical cores, with hyperthreading, giving it 16 logical cores. Extrapolating from 10 ms , 128 threads in the ideal scenario should run in 10 ms * 128 / 16 = 80 ms. What we measured shows quite some congestion, we only arrived at around 300 ms:



4.4 Smallstep Scheduler

Having an excess number of threads is obviously a bad idea, since we will delegate a lot of context switching to the operating system. What if we do some context switching on our own? Is this the secret of the GPU work group concept? We tried running Hack VMs round robin:

```
public class Laika3 implements Runnable {
    public int[] code;
    public int[] state;
    public int offset;
    public int[] pc;
    public int[] accu;
    public int active;

    public void run() {
        while (active > 0) {
            for (int i = 0; i < pc.length; i++) {
                run(i);
            }
        }
    }
```

The round robin will loop, until there are no more active Hack VM. Since we now reenter inactive Hack VMs, to keep the round robin logic simple. The Hack VM logic

needs not only load and store of the pc and accu, but also a slightly more elaborated termination logic:

```
public void run(int id) {
        int pc = this.pc[id];
        int accu = this.accu[id];
        int offset = this.offset + id * 3;
        for (int k = 0; k < 64; k++) {
            if (pc < code.length) {
                int instr = code[pc++];
                int value = run_get(instr, offset);
                accu = run_fun(instr, accu, value);
                run_set(instr, accu, offset);
                pc += run_jump(instr, accu);
            } else if (pc < code.length + 1) {
                active -= 1;
                pc++;
                break;
            } else {
                break;
            }
        }
        this.pc[id] = pc;
        this.accu[id] = accu;
    }
```

With this approach and only 16 threads time came down to ca. 200 ms:



To avoid pc/accu load store dominance, the work groups have instruction burst parameter 64. Since Hack VM to Dogelog Player VM is 100 ms : 10 ms, the single-

threaded Hack VM shows 270 MLips. Since single to multi was 10 ms : 200 ms, the multi-threaded gives 270 MLips * 128 / 20 = 1.7 GLips.

4.5 Summary

Having an excess number of threads is a bad idea. What if we do context switching on our own? With this approach we could bring down the execution time of 128 Hack VMs by 33%. We estimate for the test which had 11.4 GLips on the GPU, that we reach 1.7 GLips on the CPU.

5 π-WAM Assembly: Comfortable Labels and Goto

5.1 Overview

What Hamlet is to English language, is Hack to Compiler Construction. The playbook of Hack contains every drama that a Compiler Construction will face. In the following we show how we realized Project 6: Assembler from the Nand to Tetris journey via a little Prolog DSL.

5.2 Motivational Example

Our first take on π-WAM falls into the category quick and dirty prototype. The main goal was to have a working between/3, illustrating some ideas of our π-WAM such as no trail, no choice point and no stack. And the goal was less to have a large scale project with good maintainability.

Unfortunately, the below compiler fragment caused some headache recently. The compiler protocol was such that it takes an already compiled continuation CODE, which is assumed to be relocatable. And then determines its length via length/2 and reinserts it via phrase/1.

```
/* between/3 compiled as for loop */
sys_comp_goal(between(TERM,TERM2,VAR), _, _, CODE) --> !,
   {length(CODE, REL)},
   sys_comp_build(TERM), [((acc,set,true),VAR,0)],
   {REL2 is REL+3},
   sys_comp_build(TERM2), [((sub,get,gr),VAR,REL2)],
   {REL3 is -REL2-2}, phrase(CODE),
   [((val,get,true),VAR,0),
    ((add,cst,true),1,0),
    ((acc,set,true),VAR,REL3)].
```

This double buffering causes some overhead, but allowed us to juggle with code sizes, and compute derived codes sizes such as REL2 and REL3. The use of constants such as -2 is extremely brittle, since it makes assumptions about the number of instructions generated by sys_comp_build/1.

5.3 Assembly DSL

In the Nand to Tetris journey, the assembly phase juggles with labels and provides strategies to keep symbol tables registering these labels, and to associate these labels with locations later in the binary code of Hack. We can do without symbol tables but with Prolog logical variables.

We extend the DCG code generation by threading pairs:

```
% sys_label(-Integer, -List, +List)
sys_label(STEP, STEP-CODE, STEP-CODE).

% sys_instr(+Term, -List, +List)
sys_instr(INSTR, STEP-[INSTR|CODE], STEP2-CODE) :-
   STEP2 is STEP+1.
```

The above means that our assembly DSL has two commands. The command sys_label/3 will read the current binary code location, and return it in the given Prolog logical variable. The command sys_instr/2 will emit an instruction and increase the binary code location counter.

We can demonstrate its usage comparing old fail/0 realization:

```
/* Without Assembly DSL and with Double Buffering */
sys_comp_goal(fail, _, _, CODE) --> !, {length(CODE, REL)},
   [((val,cst,true),0,REL)], phrase(CODE).
```

With the new fail/0 realization using the assembly DSL:

```
/* With Assembly DSL and without Double Buffering */
sys_comp_goal(fail, STACK, BUFFER, CONT) --> !,
   sys_instr(((val,cst,true),0,THERE-HERE)),
   sys_label(HERE),
   sys_comp_goal(true, STACK, BUFFER, CONT),
   sys_label(THERE).
```

There are couple of differences. First of all, we don't need double buffering anymore, which is a big win for memory and time efficiency. Instead of passing an already pre-compiled continuation CODE, we can pass the original Prolog continuation CONT.

Since there is no more compiled CODE, we then remember what Edmund Landau was teaching us, in his booklet Grundlagen der Analysis about Z. We simply represent the relative address as a position difference, i.e. the compound THERE-HERE from the two labels HERE and THERE.

5.4 Between Reconsidered

That we don't need to apply double buffering has a couple repercussions for the compilation of control flow. Since continuations are now defined as Prolog goals and not as compiled code, we lose a little bit of our freedom. But we can also streamline some of the connectives.

Take the old realization of true/0 and (,)/2 conjunction:

```
sys_comp_goal(true, _, _, CODE) --> !,
   phrase(CODE).
sys_comp_goal((GOAL, GOAL2), STACK, BUFFER, CODE) --> !,
   {sys_comp_goal(GOAL2, STACK, BUFFER, CODE, CODE2, [])},
   sys_comp_goal(GOAL, STACK, BUFFER, CODE2).
```

The new realization does completely without double buffering:

```
sys_comp_goal(true, _, _, []) --> !.
sys_comp_goal(true, STACK, BUFFER, [GOAL|CONT]) --> !,
   sys_comp_goal(GOAL, STACK, BUFFER, CONT).
sys_comp_goal((GOAL, GOAL2), STACK, BUFFER, CONT) --> !,
   sys_comp_goal(GOAL, STACK, BUFFER, [GOAL2|CONT]).
```

We arrive at a form of continuation passing, where the continuation list can be used to deactivate goals temporarily form compilation. And then "true" tries to reactive goals again for compilation. We can also solve the dynamic instruction width problem of between/3 very elegantly:

```
/* between/3 compiled as for loop */
sys_comp_goal(between(TERM, TERM2, VAR), STACK, BUFFER, CONT) --> !,
   sys_comp_build(TERM),
   sys_instr(((acc,set,true),VAR,0)),
   sys_label(LOOP),
   sys_comp_build(TERM2),
   sys_instr(((sub,get,gr),VAR,THERE-HERE)),
   sys_label(HERE),
   sys_comp_goal(true, STACK, BUFFER, CONT),
   sys_instr(((val,get,true),VAR,0)),
   sys_instr(((add,cst,true),1,0)),
   sys_instr(((acc,set,true),VAR,LOOP-THERE)),
```

```
   sys_label(THERE).
```

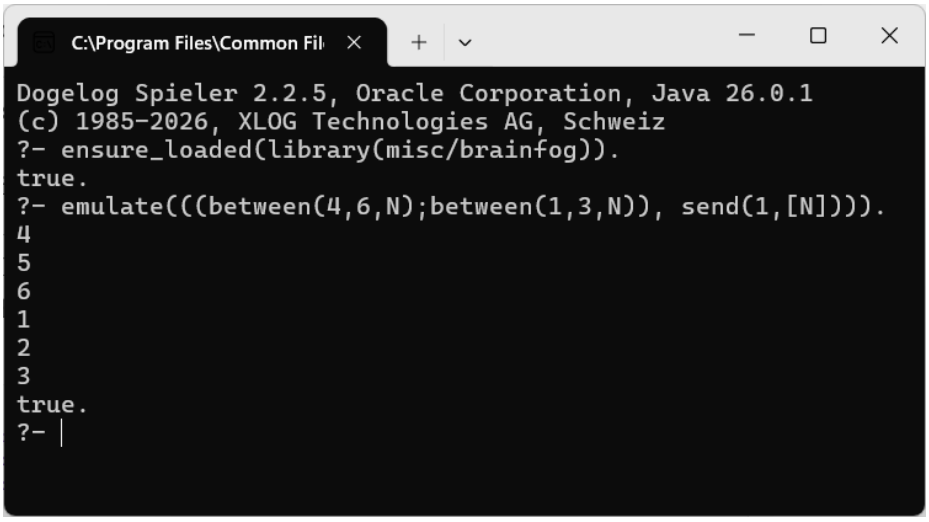
5.5 Disjunction Outlook

Seeing a label LOOP in the compilation of between/3 makes it more clear how we approach backtracking in π-WAM. A pre-compiled continuation CODE or a continuation CONT, does neither succeed or fail. It just exhausts it solutions and then continues.

The compilation of between/3 just adds a further exhaustion loop to a given continuation. It stats by assigning the value TERM to VAR. And then loops and calls the continuation and increments VAR, as long as we have VAR =< TERM2. This gave us the idea to implement disjunction as:

```
sys_comp_goal((GOAL; GOAL2), STACK, BUFFER, CONT) --> !,
   sys_comp_goal(GOAL, STACK, BUFFER, CONT),
   sys_comp_goal(GOAL2, STACK, BUFFER, CONT).
```

Interestingly this exhaustion-based compilation scheme works:



The above shows the result of the 100% Prolog emulator emulate/1. But we get the same result via execute/1 on the CPU backend and via expedite/1 on the GPU backend. The only problem is that CONT is generated twice, so we are currently working a more compact compilation scheme.

5.6 Summary

Our initial quick and dirty prototype deploys double buffering to determine code lengths and is brittle in examples such as between/3. Our assembly DSL does without symbol tables but with Prolog logical variables only. We could fix the between/3 glitch and give a first disjunction (;)/2 prototype.

6 Jump and Link for Tseytin Transform

6.1 Overview

We can report some progress in the π-WAM compilation using the recently introduced assembly with labels. While we already showed disjunction (;)/2 without resorting into TRY_ME_ELSE, RETRY_ME_ELSE or TRUST_ME. In this take we show how to do if-then-else (->)/2 .

6.2 Binary Disjunction

It all began with some ideas how to compile disjunction (A; B) in the presence of a continuation C. Distributivity of conjunction over disjunctions allows us to state ((A; B), C) = ((A, C); (B, C)). The compilation scheme itself is based on an exhaustive reading of continuations:

```
/* Binary disjunction with duplicate continuation */
sys_comp_goal((GOAL; GOAL2), REGS, BUFS, LABEL, CONT) --> !,
   sys_comp_goal(GOAL, REGS, BUFS, LABEL, CONT),
   sys_comp_goal(GOAL2, REGS, BUFS, LABEL, CONT).
```

The exhaustive reading of a continuation C says that we compile it as if it were (C, fail; true). So, reaching the end of a compilation of a continuation C, indicates the completion of an exhaustive enumeration of C. This explains the above compilation, but has the defect that the C code is duplicated.

A solution to avoid this duplication, is to pack the continuation into a subroutine by performing a Tseytin transform. We basically compile ((A; B), C) = (p <- C, (A, p; B, p)) where p is a fresh propositional variable. In compilation a fresh propositional introduction is a new label ENTRY:

```
/* Binary disjunction without duplicate continuation */
sys_comp_goal((GOAL; GOAL2), [TEMP|REGS], BUFS, LABEL, CONT) --> !,
   sys_comp_goal(GOAL, [TEMP|REGS], BUFS, LABEL, ['$JSR'(ENTRY)]),
   sys_comp_goal(GOAL2, [TEMP|REGS], BUFS, LABEL, ['$JSR'(ENTRY)]),
   sys_instr((bra,mov,EXIT)),
   sys_label(ENTRY),
   sys_instr((set,nop,TEMP)),
   sys_comp_cont(CONT, REGS, BUFS, LABEL),
   sys_instr((jmp,mov,TEMP)),
   sys_label(EXIT).
```

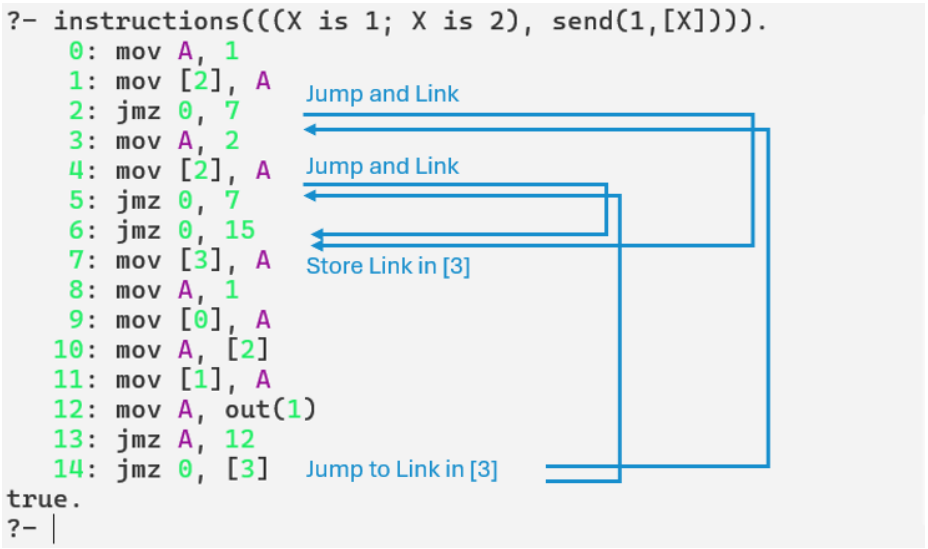
The propositional introduction p <- C has a opening and a closing. In the opening the accumulator is stored in a register TEMP. In the closing we jump to what was stored in TEMP. This allows simple subroutine calls without the need for a stack. The '$JSR'/1 call deploys a jump and link:

```
/* Calling the Tseytsin transform */
 sys_comp_goal('$JSR'(ENTRY), REGS, BUFS, CONT) --> !,
   sys_instr((bra,mov,ENTRY)),
   sys_comp_cont(CONT, REGS, BUFS).
```

Our bra and jmp instructions are RISC (ARM / RISC-V) inspired. They jump to the argument address to run a block of code, and they save a return address in the accumulator so the called subroutine knows how to get back. This is unlike CISC (x64) where a runtime stack is needed.

6.3 Global Cuts

For simplicity in the Hack VM the jump instructions always have the condition "branch if zero" on them and always perform linking. Unconditioned jumps are realized by zero. The argument can then be an immediate address, like 7, or an indirect address, like [3]:

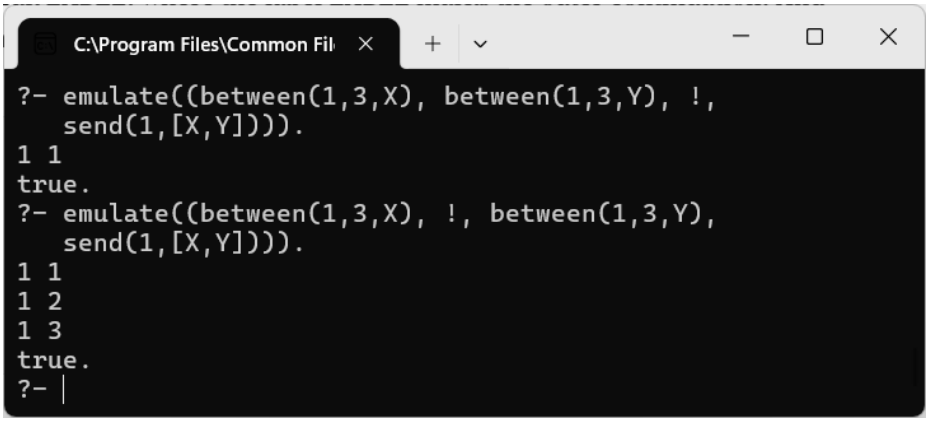


The global cut !/0, is then conceived as jump out of whatever backtracking created by disjunction or between for-loops. It is implemented as a deferred break LABEL, where the label LABEL marks the main block. And since it is a break, the jumps goes after the main block:

```
sys_comp_body(GOAL, REGS, BUFS, CONT) -->
   sys_comp_goal(GOAL, REGS, BUFS, LABEL, CONT),
   sys_label(LABEL).

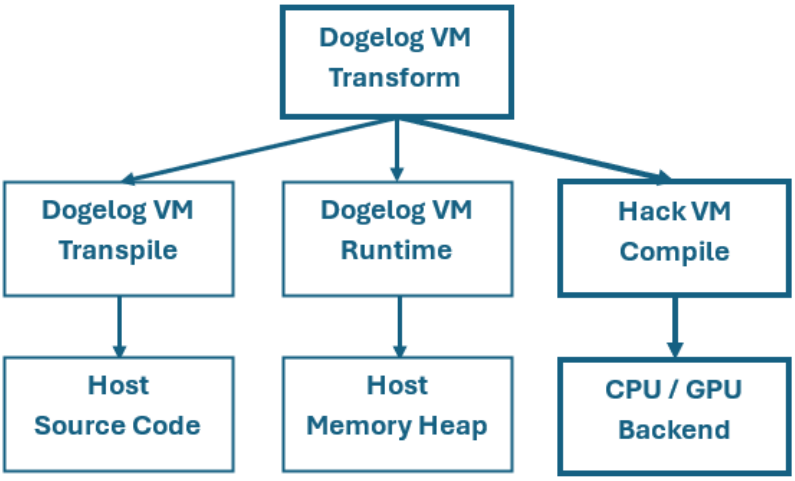
sys_comp_goal(!, REGS, BUFS, LABEL, CONT) --> !,
   sys_comp_cont(CONT, REGS, BUFS, LABEL),
   sys_instr((bra,mov,LABEL)).
```

The global cut works fine, even for nested backtracking cases:

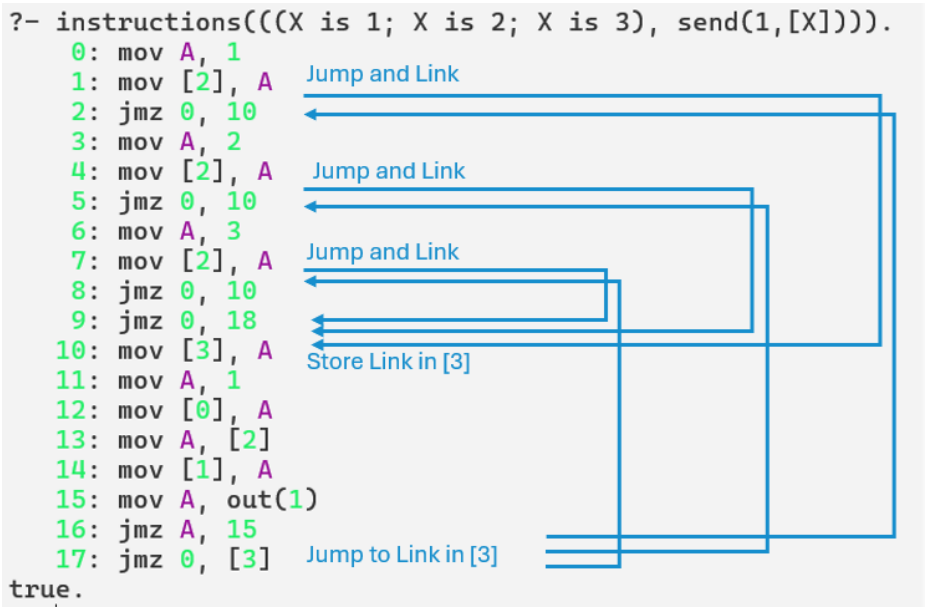


6.4 Multiway Disjunction

Only compiling binary disjunction would blow up the code. Since the N-th disjunction in a multiway disjunction would have (N-1)-many Tseytin transforms and equally many '$JSR'/1 calls for N>1. To arrive at a better compilation schema, we tap into the front Dogelog VM transform:



The transform replaces multiway disjunction by an '$ALT'/1 call with a list argument of the alternatives in the form of '$SEQ'/2. We can readily compile this list by iterating through it, and only use a single '$JSR'/1 call for either if-then-else (->)/2 alternatives or ordinary alternatives:



But multiway disjunction is not the only things we get for free in the new setting. The compilation also emits '$CUT'/1 goals and we didn't need to reinvent the wheel to

compile alternatives that are if-then-else (->)/2. The latter is now a byproduct of using the front Dogelog VM transform.

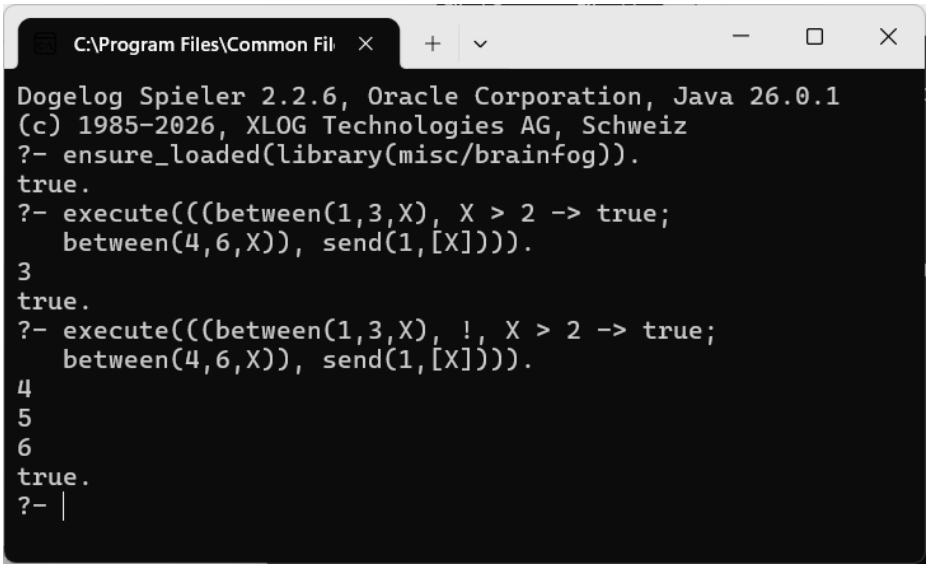
6.5 Local Cuts

The front Dogelog VM transform pairs '$CUT'/1 calls with either '$SEQ'/2 elements or '$MARK'/1 calls. We can use the first argument of both to carry our break LABEL. The '$CUT'/1 then invokes this LABEL, only the LABEL is now an argument of '$CUT'/1 itself, and not a compilation parameter:

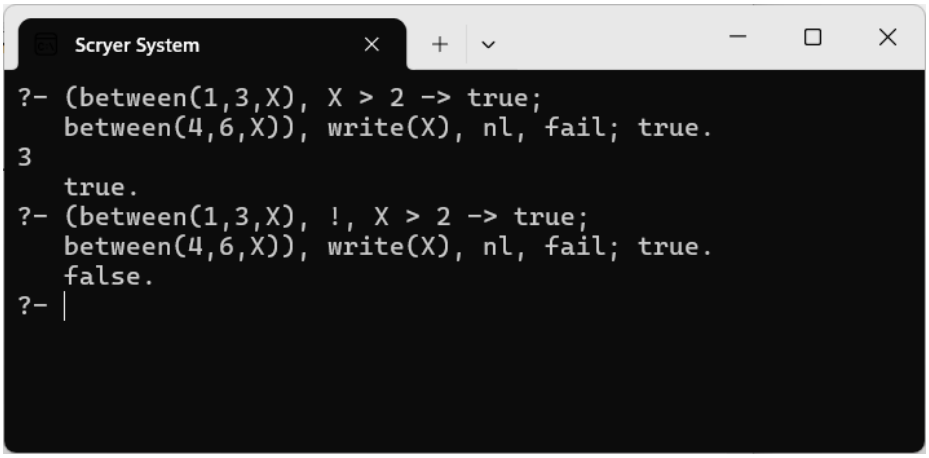
```
sys_comp_goal('$MARK'(LABEL), REGS, BUFS, CONT) --> !,
   sys_comp_cont(CONT, REGS, BUFS),
   sys_label(LABEL).

sys_comp_goal('$CUT'(LABEL), REGS, BUFS, CONT) --> !,
   sys_comp_cont(CONT, REGS, BUFS),
   sys_instr((bra,mov,LABEL)).
```

So far this means we could completely abstain from the classic WAM approach that sees a disjunction leaving a choice point at the its entrance. On the other hand our approach from the π-WAM focuses on the dual end, namely the exit. It leads to a similarly correct behavior:



Some Prolog systems erroneously make the if-condition cut transparent:



We could rejoice don't need choice points, and hence have a different take than C_OR from SWI-Prolog. But while SWI-Prologs light weight choice points store a link address, we need to do the same to call the fresh Tseytin proposition. But we do so with Jump and Link from RISC.

6.6 Summary

To aid in the compilation of (->)/2 we use the front Dogelog VM transform that introduces '$MARK'/1 and '$CUT'/1. While SWI-Prologs light weight choice points store a link address, we need also to do the same to call the fresh Tseytin proposition. But we do so with Jump and Link from RISC.

7 Conclusions

The report documents what has been done in its own scope, namely mastering the compilation of Prolog backtracking as found in between/3 backtracking and the logic of if-then-lese (->)/2 into a primitive virtual machine. This machine then shows excellent sequential and parallel performance on a CPU backend. The work can be compared to SQL Turso, although we don't rely on LLVM or to Palantir Polar, although we do not cover vector or matrix instructions. But since π-WAM also refers to π-calculus, the compilation and execution challenges are much larger. In particular the matters of channels and the GPU backend are covered elsewhere.

References

1. Nisan, N. and Schocken, S.: *The Elements of Computing Systems*, June 15, 2021 https://mitpress.mit.edu/9780262539807/the-elements-of-computing-systems/

Appendix

Prolog text of the disassembler:

```
% instructions(+Goal)
instructions(GOAL) :-
   sys_trans_body(GOAL, nothing, MARK, 1, CONT, []),
   (sys_comp_prog(MARK, CONT, CODE, _, _),
       sys_disasm_code(0-CODE, _-[]),
       fail;
       true).

% sys_disasm_code(+Pair, -Pair)
sys_disasm_code --> sys_label(STEP), sys_instr((MODE, FUN, OBJ)), !,
   sys_disasm_instr(MODE, FUN, OBJ, LIST),
   {format('~t~d~5|: ~w ~p, ~p', [STEP|LIST]), nl},
   sys_disasm_code.
sys_disasm_code --> [].

% sys_disasm_instr(+Atom, +Atom, +Integer, -List, +Pair, -Pair)
sys_disasm_instr(cst, FUN, OBJ, [FUN, 'A', OBJ]) --> [].
sys_disasm_instr(get, FUN, OBJ, [FUN, 'A', [OBJ]]) --> [].
sys_disasm_instr(ext, FUN, _, [FUN, 'A', OBJ]) -->
   sys_instr(OBJ).
sys_disasm_instr(out, FUN, OBJ, [FUN, 'A', out(OBJ)]) --> [].
sys_disasm_instr(in, FUN, OBJ, [FUN, 'A', in(OBJ)]) --> [].
sys_disasm_instr(set, mov, OBJ, [mov, [OBJ], 'A']) --> [].
sys_disasm_instr(bra, mov, OBJ, [jmz, 0, OBJ]) --> [].
sys_disasm_instr(brz, mov, OBJ, [jmz, 'A', OBJ]) --> [].
sys_disasm_instr(jma, mov, OBJ, [jmz, 0, [OBJ]]) --> [].
sys_disasm_instr(jmz, mov, OBJ, [jmz, 'A', [OBJ]]) --> [].
```