An N-ary decision tree generation algorithm for Dyninst aarch64 instruction decoder

Steve Song

July 22, 2018

Abstract

RISC architecture CPUs are considered as an alternative to CISC CPUs by scientists and engineers when they design next generation high performance computing servers. The performance gap between RISC and CISC architecture CPUs continue to be filled, especially when the applications are throughput sensitive rather than latency sensitive. Besides that, RISC is even optimized to be power efficient. In such cases, the cost of performance per unit will be the dominant factor[1]U̇nder this context, Dyninst seeks to support the aarch64 ISA that is considered as one of the options used in new servers stacked by RISC CPUs. Among the many features Dyninst supports, Instruction API is the one most coupled to the architecture and needs to be redesigned, especially the interface to identify aarch64 isa instructions. This paper conceptually describes how a C++ code generator for Dyninst aarch64 instruction decoder is designed.

1 Background - Dyninst instruction API

The manual for Dyninst instruction API can be found in [2]. As one of the steps to understand target processs semantics, Dyninst during runtime read the target processs binaries from its stacks or heaps section in memory and parse them to corresponding assembly so that it can understand the semantics on the machine instruction level. In this step, the key component in Dyninst is the instruction decoder, which helps Dyninst identify the instructions given binaries.

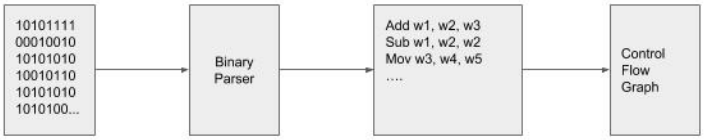

Figure 1: Dyninst decoding process

2 Identify the instruction from binary

2.1 Decoding Strategy - Design A Decision Tree

The decoding strategy is an intuitive procedure. It is mostly like how we can decide an instruction given the an instruction in binary. The root node of the decision tree examines the bit field that will differentiate the instructions into sub groups and send the binary to the corresponding child node for further examination until to the leaf node, where it can decisively tells us which assembly instruction it is.

In more detail, each non-leaf node has a bit mask used to categorize the input encoding to 2 sub-groups. First, the input binary to the node is denoted as I; the mask, e.g. 0010, is denoted as M; to return the result starting from base 0, it finally shift by S. For this particular case, S is 1; the output, also called as Branch Index, as BI. Hence, the bit field that we are interested in is the 2nd from LSB. Then the formula, F, to get the BI from I, M and S is

$$BI = f(M, I, S) = I\&M \gg S$$

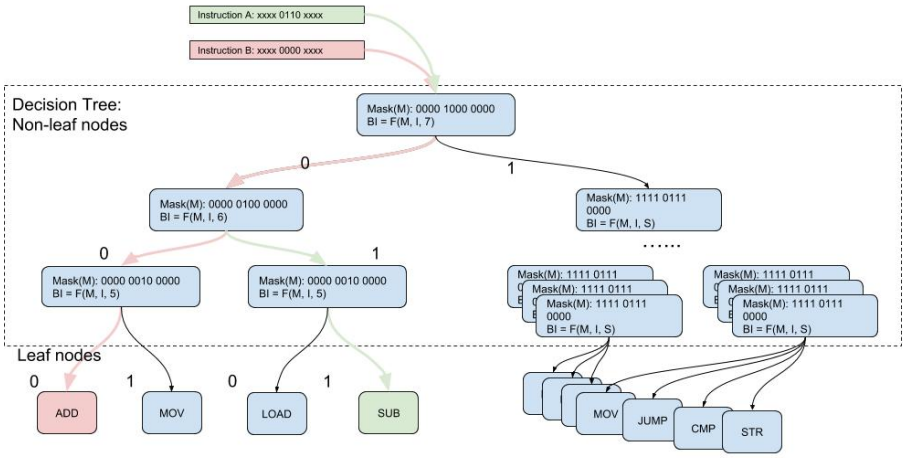

Figure 2: Binary Decision Tree

Therefore, the result BI is also either 0 or 1, which can tell which child node to go next for further examination. Obviously, the amount of shift varies on different sub comparison groups.

2.2 Tree Splitting(Optimization) - N-ary Decision Tree

Note that for some of node, we categorize the sub group into 2 is not efficient. For instance, the left sub-tree can totally use the 2 bits together as the index of

the leaf nodes. These cases are identified when all instructions in the sub group shares the same bit fields that have not yet been used for comparisons.

By this way, it saves branch jumps by indexing more fields at one node, which makes the tree even denser with shorter heights. I didn't measure the performance. However, it will apparently help reduce memory consumption and fewer jumps on the tree.

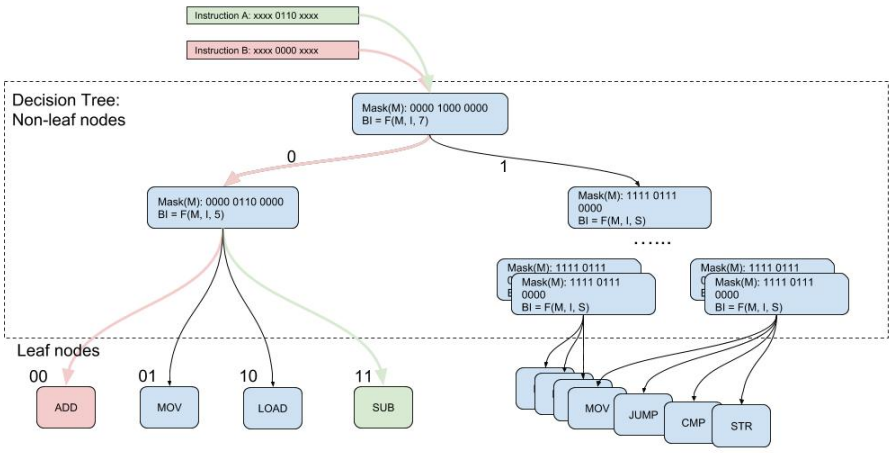

Figure 3: N-ary Decision Tree

3 Learning A Decision Tree

So far, we have figured out a decision tree based algorithm to decode the binaries from memory. However, there are hundreds of instructions in aarch64 isa. It is too trivial to implement the decision tree manually. It is even an error prone process. What is more important, we would like to always stay updated with official ARM ISA in time and be reliable low level lib. Hence, we don't want to use existing code from non official libraries. Under such situations, the best way is to generate the decision tree automatically from reliable sources. Fortunately, we have worked with ARM company and obtained the aarch64 isa manual in XML legally. This will become our ground truth.

3.1 Decoder Generator

Combined the resources with have and the existing Dyninst Instrution API lib interface, the whole process is hard: since what we want is the decision tree based decoder in C++ in the Dyninst lib, it is able to write a script to generate the decoder offline/pre-compilation. It is generally divided into two phas-

es/components with borrowing ideas from building a common compiler[3]: 1. Pre-processing: XML parser 2. Decoder generation: Decision Tree Generator.

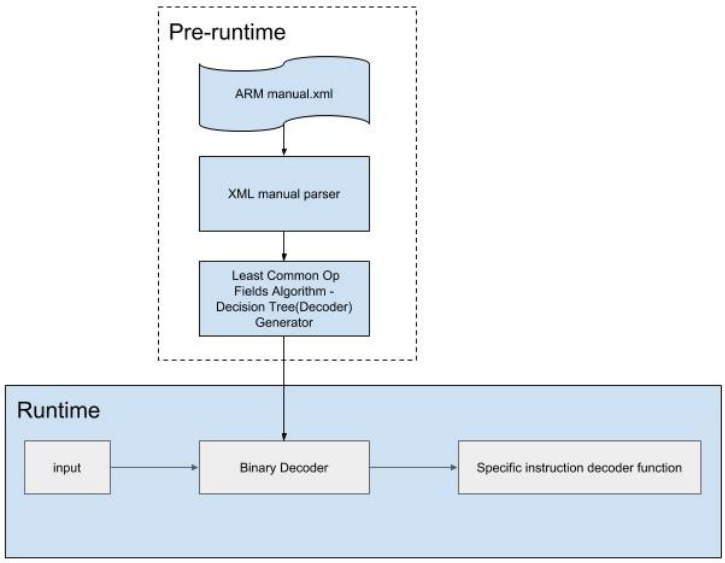

Figure 4: DecoderGen

3.2 Pre processing: XML parser

The official XML contains too much information. Most of them are not interested to us but engineers as references. To implement the decoder, the necessary information for each instruction includes:

1. Instruction semantics
2. Operation(Op) fields and encoding
3. Operand fields and allowed encoding(range)
4. Instruction Mnemonics
5. Misc fields and allowed encoding

This part is easy and trivial. I just iterate through the whole xml document and parse the related fields by identifying the tags. Finally, record the useful information and put them in a table. The table is after pre-processing feed to the Decoder Generator.

3.3 Least Common Op Fields Algorithm

Once after pre-processing, the decoder generator is feed with the instruction data table. It contains all the necessary information to decode a binary code to an instruction. The actual aarch64 ISA can be found in the ARM manual[4]. To begin with, let's take a look at what the information we have, the encoding fields for each instruction. To illustrate the problem, presuming we have four fake instructions, A, B, C and D with numbered bit position:

```
    0123 4567 890a
A:  xxxx 01xx xxxx
B:  xxxx 00xx xxxx
C:  xxx0 1110 xxxx
D:  xxx1 1110 xxxx
```

Some explanation on this representations. The x bit represent variable fields, like operands or flags. This can be set to either 0s or 1s in runtime. Fields with 0s or 1s are Opcode fields. These fields tells us which instruction it is, which is the most important information for us.

To generate the decision tree, generator first find the *least common op bit fields* so that we can divide the instruction into 2 sub groups. In this case, this least common op bit fields is 4 and 5. Since to begin with, we would like to divide sub group into 2, we take the first bit within 4, 5. Then it is 4. So it is easily to categorize instruction A & B into sub group (G1) and instruction C & D into sub group (G2). So put the mask M1 into the root node of the decision tree.

```
       0123 |4|567 890a
G1 A:  xxxx |0|1xx xxxx
   B:  xxxx |0|0xx xxxx
-------------------------
G2 C:  xxx0 |1|110 xxxx
   D:  xxx1 |1|110 xxxx
-------------------------
M1     0000  1 000 0000
```

Then we pass the sub group to the child node in the decision tree, and we have, take sub group G1 for example, instruction A and B. Since bit field 4 has been used, we mark it with *u*. Re-run the categorizing process again, we have 5 as our index bit field. Generator put the mask M2 into the child node. As one more level down the decision tree, with this information, it can uniquely identify the instruction already, the algorithm terminates and generate leaf node in the decision tree. At the meantime, insert an instruction function into the instruction function table and put the index into the leaf node.

```
    0123 4|5|67 890a
A:  xxxx u|1|xx xxxx
B:  xxxx u|0|xx xxxx
-------------------
```

M2 0000 0 1 00 0000

Similarly, we can re-run the whole process to figure the decision tree node masks and child nodes for the sub group G2. A more generalized algorithm is described in the pseudocode below.

Algorithm 1 decision tree generation algorithm

```
1:  procedure treeGen(instructionArray, node)
2:      if length of instructions = 1 then
3:          node.childs append genLeafNode(instructions[0])
4:          return

5:      maskBits = init()
6:      for each i in instructions do
7:          maskBits = i

8:      shift ← getShift(maskBits)
9:      subInstructionsGrups ← init(maskBits)
10:     for each i in instructions do
11:         bi ← f(i, mask, shift)
12:         subInstructionsGroups[bi] appends i

13:     newNode ← genNewNode(mask, shift)
14:     node.childs append newNode
15:     for each g in subInstructionsGroups do
16:         treeGen(g, newNode)
```

3.4 Implementation Data Structure

In reality, I use arrays as the decision tree. Each entry in the array represent a single node in the decision tree with the comparison meta data, like mask and shift length, and child nodes indexes in the array. The leaf node of the tree doesn't have any child node information but the index of the index to the specific instruction function in the instruction function array. By this way, it is more space efficient since the actual size of the instruction numbers are small enough to occupy a really small portion of memory[5]. Instruction Function is a specific type of functions in Dyninst Instruction API. Each instruction has a corresponding Instruction Function to parse the instruction binary and store the information in the instruction class. Once the instruction is identified, the corresponding Instruction Function is invoked, and store information for use by next stage in the lib.

4 Design Problems

4.1 Aliased Instruction

One problems I saw during running the algorithm, I found that there are some instructions with different mnemonics share the same op encoding. There are actually the same instruction with aliases. Most of them share the same encoding, but with some specific values in the operand/flag fields, they are given different names. Such aliases is just useful for human reading. They don't affect how the processor execute the instructions. However, since we are decoding the binaries, it might be good to be able to handle them. With borrowing the idea from computer architecture area to analyze this problem[6], I provide two different solutions to this problem: weak aliasing handling and strong aliasing handling.

With weak aliasing handling, during the algorithm running, we just chose the more generic instruction which covers the most range of aliased instructions with the same op code. The reason is that like I explained above, the execution path and semantic are the same.

With strong aliasing handling, decoder should be able to recognize them. The solution is during tree generation, once we collected all the aliased instructions (this is possible since aliased instruction by definition always are categorized into same sub gropus), they are sort from the most strict encoding to most generic encoding. And during comparison, the aliased instrution will be picked up by the stronger mask first before they are identified as the generic instructions.

5 Summary

In this paper, I described an least common op fields algorithm to generate decision tree based decoder for the Dyninst lib. This is used as part of the porting to ARM aarch64 architecture project. The source code can be found on the git hub repository `https://github.com/dyninst/dyninst/blob/arm64_dev/instructionAPI/aarch64_manual_parser.py`. The algorithm works well for an open source library Dyninst. Hopefully, this algorithm will be useful for some ones as references when they have similar requirements or problems.

References

[1] "https://www.toptal.com/back-end/arm-servers-armv8-for-datacentres."

[2] P. Lab, *Dyninst - Instruction API*.

[3] "https://www2.adacore.com/gap-static/gnat_book/html/node6.htm."

[4] *ARM Architecture Reference Manual ARMv8*.

[5] "https://en.wikipedia.org/wiki/binary_tree#arrays."

[6] “https://en.wikipedia.org/wiki/aliasing_(computing).”