Adding a Deterministic Finite Automaton Based Packetizer to LLVM

Anshuman Dasgupta

Summary: *This document presents an overview of the deterministic finite automaton (DFA)-based packetizer in LLVM*

1. Introduction

Hexagon is a Very Long Instruction Word (VLIW) architecture. On such an architecture, the assembly code explicitly represents the instruction level parallelism present in a program. Specifically, the compiler is responsible for mapping instructions to functional units. [1] On Hexagon, this mapping process is called *packetization*. To conduct packetization, the compiler must gather information on the hardware resources required for each emitted instruction. It must then use this information to ensure that there are no hardware conflicts in the packetized code. The compiler can packetize either during the process of instruction scheduling or after scheduling has occurred.

2. LLVM and Packetization

There are 3 compilation phases in LLVM: the front-end that converts source code into bitcode, the bitcode-to-bitcode optimizer, and the backend that converts bitcode into a target-specific representation. The LLVM compiler infrastructure conducts instruction selection followed by instruction scheduling. During scheduling, the bitcode is lowered into a directed acyclic graph (DAG). The edges of the DAG are annotated with latency information supplied by the target architecture. LLVM uses this information to create a schedule that respects the hardware characteristics of the target. LLVM's scheduling pass currently lacks any infrastructure for packetization. While packetization can be added to the LLVM scheduler, we decided not to modify the scheduling pass for two reasons. First, LLVM does not have the backend hooks required to integrate a packetization pass with the scheduler. Therefore adding this feature would involve extensive modification of the scheduler code. Second, we did not want to introduce architecture-specific code in the scheduler.

We thus decided to split the packetization pass into two components:

1. A machine-independent *machine resource* component that computes the resources required for each instruction. This information is automatically generated from an LLVM architecture-specific table-description file. The machine resource component implements a DFA that tracks the resources consumed by instructions issued by the compiler. This component has been designed to be reused by any VLIW architecture that wants to support a packetization pass.

[1] This contrasts with dynamically-scheduled super-scalar architectures in which the hardware is responsible for extracting ILP from a program by mapping instructions to functional units at program execution time.

2. A Hexagon-specific *packetization* component that queries the *machine resource* DFA to assign Hexagon instructions to packets.

3. Machine Resource Component: A DFA for Packetization

The machine resource constructs a DFA that models the resource consumption of a stream of instructions. On Hexagon, each instruction can be assigned to one or many functional units. To simplify the description of the machine resource component, the remainder of the text will focus on a hypothetical VLIW machine. Consider a VLIW machine that has 3 functional units: a Load/Store unit (LSUNIT) and two arithmetic units (ALU1 and ALU2). Arithmetic instructions can be scheduled on any functional unit. Memory instructions can be scheduled only on LSUINT

3.1 Instruction classes

VLIW architectures typically use the notion of instruction classes – a set of instructions that share the same resource consumption. For instance, on our hypothetical machine, an add and a subtract instruction can both be assigned to any of the 3 functional units. Thus they share the same instruction class: ALU_CLASS. Similarly a load and a store instruction can be assigned only to the LSUINT and share the same class: LS_CLASS.

3.2 Description of the DFA

A DFA consists of three major elements: states, inputs, and transitions. In the DFA constructed by the machine resource component, a state represents the consumption of machine resources by packetized instructions. Inputs in the DFA consist of the set of instruction classes. A transition in the DFA occurs when an instruction is added to a packet. We use existing infrastructure in the LLVM machine description to represent the state of functional units in the DFA – each unit is assigned a unique bitstring with exactly one bit set. Consider the DFA constructed for our hypothetical machine:

Functional Units:

Functional Unit	Bitstring representation
LSUNIT	0x1
ALU1	0x2
ALU2	0x4

Instructions:

Instruction	**Possible Functional Units**	**Instruction Class**
ADD	LSUNIT or ALU1 or ALU2	ALU_CLASS
SUB	LSUNIT or ALU1 or ALU2	ALU_CLASS
LOAD	LSUNIT	LS_CLASS
STORE	LSUNIT	LS_CLASS

Figure 1 shows the resulting DFA constructed for this machine. In the DFA, each state contains the possible consumption of functional units by instructions. Each transition represents the addition of an instruction belonging to a particular instruction class to a packet. The initial state denotes an empty packet with no machine resources consumed. For instance, consider State 1 which models the effect of adding an ALU_CLASS instruction to an empty packet. Since an ALU_CLASS instruction can be assigned to any functional unit, State 1 represents the possible consumption of LSUNIT (0x1), ALU1 (0x2), or ALU2 (0x4). If an instruction class cannot be added to the packet, the DFA does not contain the corresponding transition. For instance, in State 5, the LSUNIT has been consumed by an instruction. Therefore, there is no valid transition from State 5 on an input of LS_CLASS.

3.3 Constructing the DFA in LLVM

The DFA for a target architecture can be constructed while building LLVM. We modified the TableGen component of LLVM to parse the machine fragments that describe the functional units present on a target. The DFA is then constructed iteratively using the algorithm outlined in Figure 2. At the end of the construction process, we generate a sparse representation of a table T with *n* rows and *m* columns along with an API that can be used to query the availability of functional units during packetization. Rows in table T represent states and columns represent inputs to the DFA. Thus the table entry on row *i* and column *j* contains the resulting state if a transition is made from state *i* on input *j*. Since table T typically contains a large number of invalid states, we emit a sparse representation of the table. Specifically, the DFA generator emits two tables – DFAStateEntryTable and DFAStateInputTable. Entry *k* in DFAStateEntryTable is an index into DFAStateInputTable. It indicates which element in DFAStateInputTable contains the first transition for state number *k*. DFAStateInputTable contains a set of tuples *(I*, *m*) where *m* is the new state reached (i.e., the transition) on input *I*.

3.4 Packetization component: Using the DFA to conduct packetization

The generated API in encapsulated in a C++ class – DFAPacketizer.h – and contains the following methods:

- void clearResources(): Resets the DFA to the initial state
- bool canReserveResources(MachineInstr* MI): This method returns true if MI can be added to the packet
- void reserveResources(MachineInstr* MI): Adds instruction MI to the packet and sets the DFA to the corresponding state

We added a Hexagon-specific packetization pass that iterates through the instructions in a basic block after the LLVM DAG scheduling phase has occurred. The packetization algorithm begins with an empty packet and iteratively attempts to add an instruction to the packet by querying the DFA and checking for dependences between the candidate instruction and all other instructions in the packet. If an instruction can be added, the DFA is transitioned to the appropriate state. If a instruction cannot be added, the DFA is reset to the initial state and a new empty packet is created. The algorithm continues until it encounters the end of the basic block.

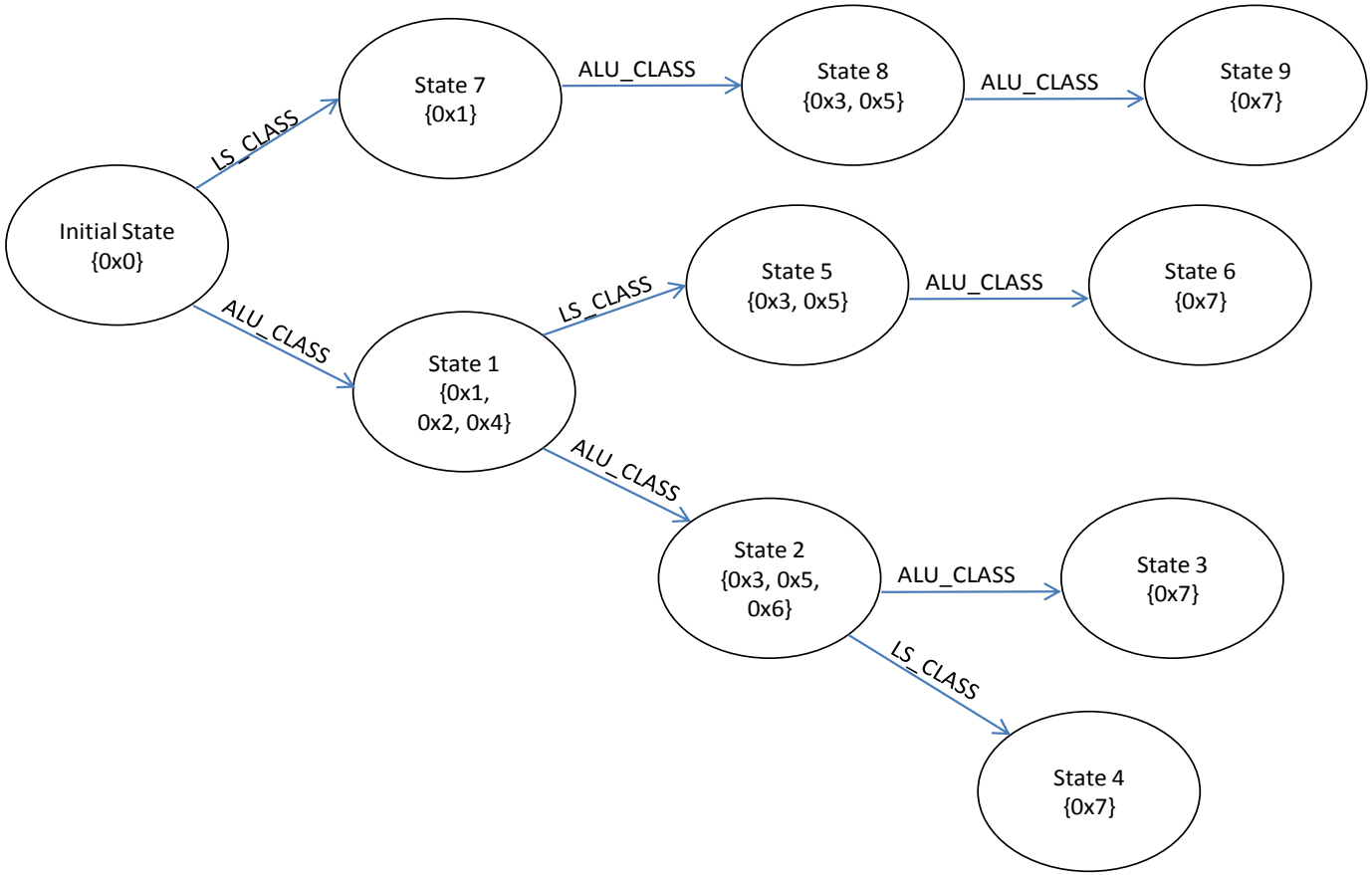

Figure 1: DFA for a hypothetical machine with 3 functional units

```
Function CreateDFA()
   for all instruction classes C:
      Record the resources consumed by class C in list R

Create a new DFA D with an initial State I
Worklist = {I}
while Worklist is not empty:
   CurrentState = Pop first element from Worklist
   if (Transition from CurrentState on InsnClass does not exist):
     if (ValidTransition(CurrentState, InsnClass, NewConsumption)):
       Add a new state, NewState, to DFA D with resource consumption = NewConsumption
       Add a new transition to DFA D from CurrentState to NewState on Input InsnClass
       Add NewState to the end of Worklist


Function ValidTransition(CurrentState, InsnClass, NewConsumption)
NewConsumption = {}
for all resource states R in CurrentState:
   s = sizeof(InsnClass) * 8
   for i from 0 to s:
      if ((0x1 << i) & InsnClass):
         Consumption = R | (0x1 << i)
         if (Consumption != R)
            This indicates that at least one resource can be used to accommodate an
            instruction of class InsnClass in CurrentState
            if (we haven't encountered Consumption before):
               Add Consumption to NewConsumption
```

Figure 2: Pseudo-code for DFA construction algorithm