Compiler Instruction Scheduling Leveraging the LLVM Infrastructure

Yucheng Chen
RISC-V International Opensource Laboratory
Tsinghua University
chen-yc21@mails.tsinghua.edu.cn

***Abstract*—In compiler backend, instruction scheduling is an important technique to generate efficient assembly code by reordering instructions to increase instruction level parallelism. In the past decades, there are two mainstream solution for instruction scheduling. One is heuristic way, which use list scheduling algorithm to determine instruction arrangement according to topology and heuristic value. Another is combinatorial optimization approach, which use integer programming, constraint programming and so on to generate optimal solution in cost model.**

In this report, we focus on instruction scheduling in LLVM infrastructure. We use LLVM tablegen to generate a cost model and we use machine instruction scheduler to run heuristic on the cost model graph. We run embench(www.embench.org) on the performance model of RISC-V rugrats to evaluate the effectiveness of the scheduling.

We achieve average of 8.5% cycle number reduction comparing with not scheduled code, and we achieve average of 3.8% cycle number reduction comparing with the LLVM generic scheduler. In some particular cases, we respectively achieve up to 44% cycle number reduction compared with the unscheduled code and up to 18% reduction compared with LLVM generic scheduler.

***Index Terms*—Compiler, Instruction Scheduling, LLVM, TableGen**

I. Introduction

Compiler Backend: The compiler backend takes the intermediate representation (IR) of the program and generates assembly code for a specific processor. The main tasks of the backend are instruction selection, register allocation and instruction scheduling. Instruction selection explains abstract operations using processor instructions. Register allocation map temporaries to physical registers or memory address. Instruction scheduling reorders instructions to make better use of processor pipeline and improve the latency and throughput.

Dependency: Dataflow among instruction forms dependency, which is a DAG(Directional Acyclic Graph). A node in the graph represent instruction. An edge in the graph from i-th to j-th node means the output of i-th instruction is the input of the j-th instruction. The weight of the edge is assigned as the latency of the first cycle that j can be execute minus the cycle that i start to execute.[1]

AntiDependency: AntiDependency between i-th instruction and j-th instruction is when the input/output of j instruction is the input of the i instruction. This means j-th instruction can't schedule before i-th instruction. In this case, an edge of weight 0 is connected from i-th instruction to j-th instruction.

Processor Resource: Processor resources refer to functional unit and data bus in processor. Each processor resource has its capacity, to represent number of instruction it can holds. Also, an instruction also has its occupation, to represent what resources it occupies.

Scope of Instruction Scheduling: According to spatial distribution of a program, instruction scheduling can be classified as three types:

1) Local instruction scheduling: Instruction arrangement is considered within a basicblock. For example, LLVM list scheduling algorithm.
2) Regional instruction scheduling: Instruction arrangement is considered within a set of basicblock that is closely related. For example, in superblock scheduling, instructions are scheduled within a collection of consecutive basic blocks.[2]
3) Global instruction scheduling: Instruction arrangement is considered within all the basicblock in a function.[3]

Mainstream Solution:

1) Heuristic approaches:
   - The traditional back-end uses a customized heuristic algorithm to solve each problem in isolation. The algorithm makes a series of greedy decisions based on local optimization criteria. This method makes the traditional backend faster, but it makes problem solving suboptimal and it is complex and irregular to build.
2) Combinatorial approaches:
   - Combinatorial methods can solve the compiler back-end problem optimally according to the model, but it will increase the compilation time. The accuracy of combinatorial method in modeling its problems is important because the calculated solution is optimal with respect to the model rather than the problem itself.

LLVM LLC: the LLC tool of LLVM is the compiler backend of llvm. It consume LLVM-IR as input and produce assembly code. LLC tool is consisted of a set of passes and a pass manager. Generally, LLC will first input the LLVM-IR and parse it to its own data structure. Then, the pass manager will perform several passes on it. Every pass will either lower the abstract level of code or optimize the code. Instructure scheduler in LLVM is also implement as the form of passes.

LLVM TBLGEN: llvm-tblgen is a program that translates compiler-related target description (.td) files into C++ code and other output formats. It's used as a bridge from human high level description to large amount low level machine c++ record description. In this report, we focus on use tablegen to break instruction into input/output microoperation, and define the dependency between the microoperation. Tablegen is also used to define processor resource and instruction occupation.

II. Methodology and procedures adopted

We first used SchedReadWrite, to break down instruction into a set of use operands and one def operand. Then we define the processor resource according to the specific architecture functional unit. We use WriteRes to bind the functional unit with the def operands and use ReadAdvance to implement the forwarding mechanism. After binding operands to WriteRes and ReadAdvance, we define the SchedMachineModel to describe the general information of the architecture and register it as a subtarget of LLVM RISC-V backend.

We leverage the machine scheduler and run a set of heuristics hierarchically on the machine scheduler with the cost model. Machine Scheduler arrange the instruction in topology order and pick candidate node based on the heuristics result.

SchedReadWrite: Every instruction can be decompose as a set of input operands and output operands. The operands are defined as a SchedRead or SchedWrite class, to indicate an in arc or out arc in the instruction dependency DAG. Take vmacc

```
vmacc. vd rs1 rs2 vm
```

as example, vmacc is belong to VMAC_MV_V_X operation cluster(vmadd,vnmsub,vnmsac are also belong to this cluster).

```
VMAC_MV_V_X:
    V : VALUrVV<WriteVIMulAddV,
        ReadVIMulAddV, ReadVIMulAddV,
        ReadVMask>;
    X : VALUrVX<WriteVIMulAddX,
        ReadVIMulAddV, ReadVIMulAddX,
        ReadVMask>;
```

This VMAC_MV_V_X cluster is discussed as 2 types, one for vector operand input, another for scalar operand input.

- WriteVIMulAddV: represent a mul-add vector input type's output destination operand.
- WriteVIMulAddx: represent a mul-add scalar output type's destination operand.
- ReadVIMulAddV: represent a mul-add type vector input source oeprand.
- ReadVIMulAddX: represent a mul-add type scalar input source operand.
- ReadVMask: represent the vector mask.

All input operands are defined as SchedRead while all output operands are defined as SchedWrite.

ProcResource: We model functional unit of rugrats as the ProcResource. There are 2 ALU(one for mul,add,branch, another for mul,add,CSR), one LSU with buffer size of 4 entries. Divide unit and branch unit are with a buffer size of 2 entries. WriteRes is later used to bind the output operand with the processor resource.

WriteRes: We define WriteRes for each output operand to bind the ProcResource with the write latency.

```
let Latency = 1 in {
    def : WriteRes<WriteVIMulAddV,
                   [VEXEMACC]>;
    def : WriteRes<WriteVIWMulAddX,
                   [VEXEMACC]>;
}
```

In above code, vmacc instruction goes through VEXEMACC in the subtarget with 1 cycle latency. We use writeRes to bind the ProcResource VEXEMACC with the output operands.

ReadAdvance: We define ReadAdvance for each input operand to deduct the forwarding cycle bonus for input operands. Consider following example code snippet:

```
%div = sdiv i32 %a %b
%add = add i32 %div %c
```

Suppose div latency is 7, in the scheduler view it would be:

```
SU(1): Latency = 7
    %div = sdiv i32 %a %b

SU(2): Latency = 7
    %add = add i32 %div %c
```

Suppose input operand ReadALU has 1 cycles forwarding bonus, by defining ReadAdvance, we could have the 1 cycles deduct to the div latency.

```
def : ReadAdvance<ReadALU, 1>;
--------
SU(1): Latency = 7
    %div = sdiv i32 %a %b

SU(2): Latency = 6
    %add = add i32 %div %c
```

SchedMachineModel: SchedMachineModel defines the records for MCScheduler class in LLVM and register a subtarget for LLVM.

```
def Xmodel:SchedMachineModel {
    let MicroOpBufferSize = 64
    let IssueWidth = 2
    let LoadLatency = 3
    let MispredictPenalty = 10
}
```

- MicroOpBufferSize is the number of instruction the processor could buffer for out of order execution. This is a estimate of machine specific feature such as the reorder buffer and register renaming pool, while the estimation itself is independent from subtarget.

- IssueWidth is the max instructions number scheduled in the same per-cycle group. It is used for modeling bottleneck between the reservation station and decoder.
- LoadLatency is the load instructions expected latency.
- MispredictPenalty is the typical cycles number that the processor needs to take to recover from a branch misprediction. The number of cycles the processor needs to recover varies. Generally it should set to an average number of the cycles that recovery takes.

After registering the SchedMachineModel, compile the tablegen to c code and included it into the llvm.

MachineScheduler: In machine scheduler, a machine basic block is divided into multiple schedule region according to list schedule algorithm is used to pick candidate instruction node according to a topology arrangement. To decide which candidate to pick, a set of heuristic is run hierarchically in tryCandidate():

1) physical register copies: copy def physical register to their use to shorten register live range.
2) register pressure (excess): avoid excess the target's register pressure limit.
3) register pressure (critical pressure): avoid increasing max critical pressure in the schedule region.
4) acyclic latency: aggressive schedule for loop.
5) clusters: Keep node with weak edge cluster together for later pass to do peephole optimization.
6) register pressure (current max): avoid increasing max pressure of schedule region.
7) critical resource: avoid critical resource consumption.
8) latency: avoid long latency instruction chain together.
9) source order: use original source assembly order.

The latency relevant heuristic calculate depth of the schedule unit(SU). SU with less height will be selected. When calculte height, the longest chain of the dependency graph is calculated. Weight of each edge is set from the SchedMachineModel class generated from tablegen.

The tryCandidate examine heuristic in above order hierarchically, when it finds one heuristic return true, it will return true to tell that the candidate SU is better than the current SU.

III. Results and discussion

In this section, we present analysis on two cases and a table of performance. The experiment is run on rugrats out of order performance c model. Scheduled binaries and not scheduled binaries are both compiled in O3 environment. The result is shown in TABLE I.

A. Performance on Benchmark

Performance Table:

We run embench as experiments on unscheduled code, LLVM generic scheduled code and our subtarget specific scheduled code. We achieve average of 8.5% cycle number reduction comparing with not scheduled code, and we achieve average of 3.8% cycle number reduction comparing with the LLVM generic scheduler. In some particular cases, we respectively achieve up to 44% cycle number reduction compared with the unscheduled code and up to 18% reduction compared with LLVM generic scheduler. Program with more dependency, longer the basic block critical path can benefit more from the compiler scheduling (e.g. tarfind.c, edn.c, slre.c). Program with more branches, can hardly benefit with the scheduling or even get worse result (e.g. statement.c), since the machine scheduler only reorder instruction within basic block.

B. Case Analysis

Critical Resource Case: Consider a program (div.c) with some back-to-back divide and some other independent calculation:

```
for(i=3;i<10000;i++){
    ...
    d = c / i; e = d/ i;
    d = e / i; e = d /i;
    d = e / i; e = d /i;
    b = b *a; b = b +c;
    b = b *a; b = b +c;
    b = b *a; b = b +c;
    b = b *a; b = b +c;
    b = b *a; b = b +c;
    b = b *a; b = b +c;
    ...
}
```

Compile it with default O3 and scheduled O3 respectively:

```
llc div.ll -o div.exe -O3
llc {mcpu=RIOSDJIModel easy.ll -o div.sched \
    -debug-only=machine-scheduler \
    -verify-misched -enable-misched
```

In div.exe, div instructions are arranged consecutively while in div.sched, div instructions are arranged separately. By viewing the debug log of machine scheduler:

```
Executed 34c
Critical: 33c, 33 IDIVUNIT
```

we could know div instructions are arranged separately because heuristics try to avoid adding pressure to critical processor resources. .

Dependency Case: Consider following machine instruction IR without scheduling:

```
SU0  %2:gpr = COPY %23:gpr
SU1  %1:gpr = COPY %22:gpr
SU2  %9:gpr = ADD %1:gpr, %2:gpr
SU3  %23:gpr = nsw MUL %9:gpr, %2:gpr
SU4  %10:gpr = nsw ADD %23:gpr, %2:gpr
SU5  %11:gpr = nsw MUL %10:gpr, %23:gpr
SU6  %12:gpr = ADD %21:gpr, %9:gpr
SU7  %13:gpr = ADDI %12:gpr, 3
SU8  %14:gpr = nsw MUL %13:gpr, %2:gpr
SU9  %15:gpr = nsw MUL %14:gpr, %13:gpr
SU10  %22:gpr = nsw ADD %15:gpr, %11:gpr
```

NAME	Unsched Cycle	Generic Cycle	%(to Unsched)	Subtarget Cycle	%(to Unsched)	%(to Generic)
matmul	1376481500	1391046500	-1.058132637	1373297500	0.231314406	1.275945844
crc	12995500	12983000	0.096187142	12995500	0	-0.09627975
edn	4080614500	2439718500	40.21198278	2282464000	44.06567932	6.445600179
md5	1271242000	1317197000	-3.614968668	1067669000	16.01370943	18.94386337
slre	655546000	642183500	2.03837717	647227500	1.268942225	-0.785445282
sha256	1255991500	1255328500	0.052786981	1095342000	12.79065185	12.74459235
huffbench	1248933500	1218718500	2.419264116	1217016500	2.555540387	0.139654892
tarfind	539200500	470616000	12.7196655	452528000	16.0742618	3.843473235
statement	420813000	432281000	-2.72520098	452223500	-7.464241837	-4.613318652
ud	945310000	945309500	5.28927E-05	945309500	5.28927E-05	0
AVERAGE:			5.01400143		8.553591047	3.789808619

TABLE I

TABLE 1: CYCLE NUMBER OF UNSCHEDULE CODE, LLVM GENERIC SCHEDULING CODE, AND OUR SUBTARGET SCHEDULING CODE

```
SU11   %21:gpr = nuw nsw ADDI %21:gpr, 1
SU12   BNE %21:gpr, %17:gpr, %bb.1
```

We calculate the critical dependency DAG from IR above as Fig.1, in which each instruction node would only retain edges that support its depth. Scheduler will select the node according

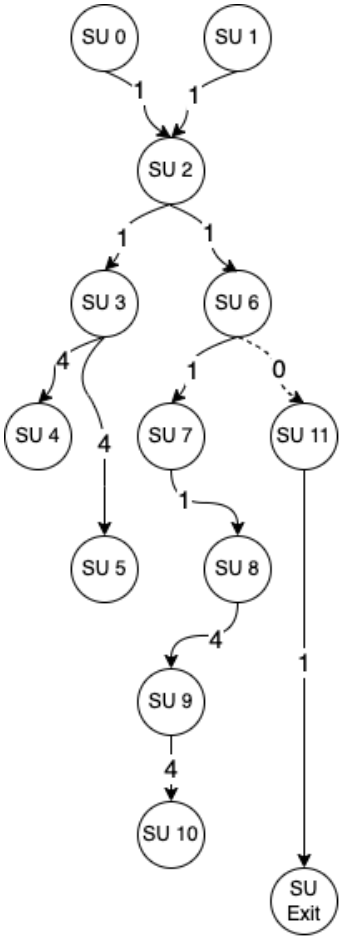

Fig. 1. Dependency Critical DAG

to the topology order. When multiple candidates are available, the node with maximum depth will be selected. In this way, instruction on the critical path will have higher priority to be scheduled, thus the overall cycle number can be reduce in this muladd case.

IV. CONCLUSION

In this report, we focus on instruction scheduling in LLVM infrastructure. We use LLVM tablegen to generate a cost model by

- breaking down instruction into use/def operands.
- binding def operand with subtarget processor resources and defining its latency.
- defining Readadvance to give latency bonus of forwarding to use operand.
- defining the subtarget schedule machine model and register it to LLVM machine scheduler.

Then we use machine instruction scheduler to run several heuristics in order of their priority hierarchically on the cost model graph. The machine scheduler visit the instruction DAG in topology order and choose candidate of best heuristic result each time.

We run embench as experiments on unscheduled code, LLVM generic scheduled code and our subtarget specific scheduled code. We achieve average of 8.5% cycle number reduction comparing with not scheduled code, and we achieve average of 3.8% cycle number reduction comparing with the LLVM generic scheduler. In some particular cases, we respectively achieve up to 44% cycle number reduction compared with the unscheduled code and up to 18% reduction compared with LLVM generic scheduler.

REFERENCES

[1] K. Wilken, J. Liu, and M. Heffernan, "Optimal instruction scheduling using integer programming," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 121–133. [Online]. Available: https://doi.org/10.1145/349299.349318

[2] G. Shobaki, K. Wilken, and M. Heffernan, "Optimal trace scheduling using enumeration," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 4, mar 2009. [Online]. Available: https://doi.org/10.1145/1498690.1498694

[3] S. Winkel, "Optimal versus heuristic global code scheduling," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 43–55.