Compilation Techniques for VLIW Architectures

Dietmar Ebner and Florian Brandner

VU2 185.32

Assignment 2

Please deliver all answers via e-mail with the subject “[VLIW] <matr.nr..> - <name>”. For changes to the sourcecode and/or makefile, use the diff utility, e.g.,

```
diff -Nupr ../reference .
```

Example 1: VLIW Toolchain (10P)

Using the VEX toolchain (`http://www.hpl.hp.com/downloads/vex`), perform the following experiments with the MPEG2 decoder available on the course website. The makefile assumes that the vex toolchain is installed in “`$(HOME)/vex`”. You can use the phony target “`sim`” to run the decoder and compare the results to reference data. The VEX documentation can be found in the `./share/doc` subdirectory.

a) **[2P]** The VEX toolchain supports custom machine and memory architecture models. Compile and and run the mpeg2 decoder for the risc, default, and vex8 machine description. Compare the results in respect to execution time, IPC, stall cycles, and nops.

b) **[2P]** Create profiling data and re-compile the application with profiling. Compare the results (for the default model only!) tho those without profiling from the previous run.

c) **[6P]** Use `gprof` and the `rgg` utility to examine the profile for the example stream. Where (in which function) does the decoder spend most of its time? Use compiler switches and pragmas to improve the performance critical parts of the application. What is the best cycle count you can obtain? You are free to use inline assembler and/or assembler modules.

The winner gets a small prize ;-)

Example 2: Branch Delay Slots (20P)

The VEX architecture does not expose branch delay slots. Assume a 4-way architecture where load and multiply operations have a latency of 3 instructions, and branches (jumps, return, call instructions) expose a delay slot of 4 cycles. All other instructions have a latency of one cyle. There are no multiway branches, i.e, each instruction must not contain more than a single jump which can be scheduled in the first issue slot only. Use the provided VEX parser template to write a simple list scheduler that produces a correct schedule using explicit nop instructions. Don't care about memory deallocation or efficiency.

a) **[10P]** Implement the function `sched::computeDDG()` to compute a data dependence graph for the given list of vex instructions. The first node is an artificial start vertex with a dummy edge to each node without predecessors. Likewise, the node with id 2 is an end vertex with an edge leading from each node without successors. Data structures are already defined in `sched.h`. Use the `risc.mm` machine model to generate test data. Label the edges with the type of the dependence (e.g., control, raw, ...) and the edge weight. Branch delays can be expresses with negative weights, e.g., the weight of an edge among a simple add and a return instruction is -3. A weight of 0 means that the source node can be scheduled within the same VLIW instruction but not afterwards. You can assume that there are no branch targets apart from the labels present in the instruction stream. Call instructions can be considered to be a barrier.

b) **[10P]** Use the DDG implemented in a) to write a simple list scheduler (`sched::schedule()`)that produces a correct stream of operations. Insert explicit nops where necessary. Each VLIW instruction consists of 4 successive operations. NOPS are expressed by setting the instr pointer in the result list to NULL.

Again, the student with the best results earns a little surprise!