Flow Control Optimizations

D. Jiménez, E. Morancho and À. Ramírez

March 24, 2016

Index

Previous Work

Tools

1. Using `oprofile`, answer the following questions:
   (a) Which event you have to use to figure out the routines with most of the conditional branch executed instructions?
   (b) Which event you have to use to figure out the routines with most of the miss prediction conditional branches?
2. You can figure out the number of executed branch instruction using `oprofile` and `pin` tools.
   (a) Which tool gives more accurate results?
   (b) Is it possible to get miss predicted branch information with `pin` tool?

1

Inlining

1. In this exercise we will analyze the performance impact of the *inlining* optimization in the ORIGINAL VERSION of the `pi.c` program (again given in the lab4_session.zip file).

   (a) Perform timing of the `pi.c` compiled with O3 optimization level and without *inlining* (Do `make pi.3ni` to compile the `pi.c` program with `-fno-inline` flag, that is, without inlining).

   (b) Perform timing of the `pi.c` compiled with O3 optimization level. In that case, the compiler applies *inlining*. Which is the speedup compare to the previous execution?

   (c) Indicate which routines are inlined by the compiler.

   (d) Analyze if the compiler could do other optimizations once it inlined those routines, and enumerate which optimizations has been done if that is the case. Hint: Use `pin` tool and check if there has been any significant change on the number of executed instructions of different types of instructions as divide, multiply and shift operations.

   (e) Re-write `DIVIDE` as a macro and compile the program with `O3` optimization level but not *inlining* (compile with `-fno-inline`). Compare the execution time of this new version to the original code, compiled with O3 optimization level.

   (f) Use the best strategy of *inlining* for the next steps of optimizations of `pi.c`.

2

Loop unrolling

2. In this exercise you will analyze the performance impact of applying *loop unrolling* to `matriu4x4.c` program. This program generates two 4x4 matrices and multiply them. **In order to avoid that compiler makes *loop unrolling*, compile your program with O2** (`make matriu4x4.2` generates the executable compiling with O2 optimization level).

   (a) **Make *inlining* of the routine `multiplica` by hand (or using the `__attribute__((always_inline))` directive)** .

      i. Do timing of the program.
      ii. Profile the program execution with `oprofile` and `gprof`.
      iii. Looking at the assembler, compute the approximated number of instructions done to multiply two matrices.

   (b) Make full *unrolling* of the inner loop of the code (Advice: use macros).

      i. Do timing of the program. Compute speedup compared to previous versions.
      ii. Profile the program execution with `oprofile` and `gprof`.
      iii. Looking at the assembler, compute the approximated number of instructions done to multiply two matrices.

   (c) Make full *unrolling* of the next inner loop of the code.

      i. Do timing of the program. Compute speedup compared to previous versions.
      ii. Profile the program execution with `oprofile` and `gprof`.
      iii. Looking at the assembler, compute the approximated number of instructions done to multiply two matrices.

   (d) Make full *unrolling* of the three loops of the code.

      i. Do timing of the program. Compute speedup compared to previous versions.
      ii. Profile the program execution with `oprofile` and `gprof`.
      iii. Looking at the assembler, compute the approximated number of instructions done to multiply two matrices.

   (e) After doing those experiments above, which is the *unrolling* degree that gives best performance? Why? Justify your answer based on what you have seen in the profiling and assembler code (Hint: pin tool can help you to understand the timing results and the assembler code).

   (f) Now, apply to the original code (with no inlining) the full *unrolling* of the three loops, compile it with O2 too, and do timing. Why this version is not as fast as the inline version? (Hint: pin tool can help you to understand the timing results and the assembler code).

3. Apply *loop unrolling* to your best version of the `pi.c` program (pi program with memoization). Find out the best *unrolling* degree for this program and be careful with the inlining (hint: look at the loops with a small loop body).

3

Loop fusion

Remember that the performance improvements achieved with loop fusion are also due to a better exploitation of the memory hierarchy in some cases, and not only due to the reduction of the loop overhead.

4. Use *loop fusion* in your best version of the `pi.c` program. Note that loops to be fusioned may be in different routines.

4

Removing Conditional Branches

5. In this exercise you will analyze the substituton of the conditional branch in `pi.c` program using bithacks and memoization.

   (a) Profile your best version of `pi.c` with `gprof` and `oprofile` (with the appropiated events to measure number of conditional branches and misspredited branches).

   (b) Optimize the SUBSTRACT routing so that the problematic conditional branch is removed using memoization.

   (c) Optimize the SUBSTRACT routing so that the problematic conditional branch is removed using bithacks. Hint: look at the absolute value and max value bithacks to figure out how to do it.

   (d) Do timing of both optimized versions.

   (e) Profile both versions with `oprofile` checking that we have reduced the number of miss predicted branches.

   (f) Compute the speedup of both versions compare to the previous version with conditional branch.

6. Optimize `LONGDIV` under point of view of branches and taking into account that we need and have 32-bit integer representation. Which is the speedup that you achieve?