Exam in EDAF15 Algorithm Implementation

May 30, 2011, 14-19

Inga hjälpmedel!

Examinator: Jonas Skeppstedt, tel 0767 888 124

30 out of 60p are needed to pass the exam.

1. (10p) Explain the following terms and why they are essential in fast pipelined processors which can start executing multiple instructions every clock cycle, i.e. superscalar processors.

   - (4p) Branch-prediction
   - (3p) Rename registers
   - (3p) Reorder buffer

   Answer *See book.*

2. (10p) Compare a direct mapped and a 2-way set associate cache. Why is not a processor's data cache fully associative?

   Answer *See book about direct mapped vs associative cache. The reason a processor's data cache is not fully associative is that too many comparators would be required which would be a waste of chip area (transistors). It is* **not** *that it would take too much time to search through the entire cache since that is not done due to the replication of comparators.*

3. (10p) Implement a stack of integers of type `int`. Your implementation should have the following functions:

   - (2p) a function `new_stack` to create a new and empty stack.
   - (2p) a function `free_stack` to deallocate the stack.
   - (2p) a function `push` to push a new integer on the top of the stack.
   - (2p) a function `pop` to remove and return the value on the top of the stack.
   - (2p) a function `top` to return the value on the top of the stack without removing it.

   Both `pop` and `top` must check that there is something to return, and call `error` otherwise. Your implementation should use `malloc` and `free` but

not every time an integer is pushed or popped. If `malloc` returns `NULL` you should also call `error`.

For example, if you have created a new stack, and then `push` first 11, then 13, and finally 44, then `top` should return 44. If you then call `pop` once it should return 44, and if you call `pop` again it should return 13. If you call `pop` again it should return 11. If you then call `pop`, it should call `error`.

Answer *Since* `malloc` *is not allowed to be called on every push, and not* `free` *on every pop, you can for instance represent the stack as a list and rely on using a free-list to full the requirement above. Alternatively the stack can be represented using an array such as in the book on page 99.*

4. (10p) Suppose you are part of a team which needs to define a struct which contains quite a number of both scalar variables and arrays. Your team's C program, which needs to be very fast, will use many such structs in many different functions. In the discussion on in which order the members of the struct should be declared, one person says:

   "We can just let the compiler optimize the layout!"

   Your remark starts with *"No, that's impossible, because..."*

   - (2p) How would you continue that sentence?

     Answer *C compilers are not permitted to change the order of attributes in a struct. It is forbidden by the language definition.*

   - (8p) What would instead be your opinion about how to proceed?

     Answer *We should instead check which attributes are used at approximately the same time so we can benefit from both temporal and spatial locality. Furthermore, to reduce the amount of padding between variables, we should put variables (i.e. attributes) that are going into the same cache block in an order so that the padding is minimized — for instance in an order of descending size. We should first read all source code and then use the gprof and gcov tools to profile the program to get more insights into what the program does, and then use cachegrind or oprofile to measure the number of cache misses for different proposed struct layouts. Then we pick the best struct layout.*

5. (6p) Suppose you have a C program which needs to be very fast, which, however, it currently is not. Your profiling shows it spends more than 30% of the execution time in `malloc` and `free`. Under which circumstances would you consider using each of the following functions/approaches, and why?

- (2p) `alloca`

  Answer *The data size is reasonable and it needs not to be returned from the allocating function.*

- (2p) one or more arenas

  Answer *We can find objects which can be deallocated (i.e. not needed any more — we cannot deallocate individual objects from an arena, unless it was the most recently allocated object) at approximately the same time such as after processing some work. Then the entire arena can be deallocated after that work is completed. If the objects will be needed at unpredictable lengths of time, arenas are less suitable since they can contain lots of unused objects but not be deallocated since some object still is needed.*

  If we can classify objects into classes which can be thrown away at specific points in the program then with multiple arenas, each arena can take care of one such class.

- (2p) free-lists of recently used objects

  Answer *If it often happens that objects of the same size are allocated and deallocated then they can be put on a free-list instead of deallocating them using `free`. If the objects are of different sizes or only deallocated near the end of an execution, free-lists are not useful.*

6. (4p) If you have coded the inner loop the best you can, timed it, but cannot understand why it still is much slower than you had expected, what would you do?

   Answer *Assuming the inner loop was coded the best you can means you have already taken the cache into account, the best thing to do is to read the assembler code and see if the compiler produces inefficient code. If you still cannot understand the reason, it's time to use detailed pipeline simulators which will show you what actually happened.*

7. (10p) The C code on the next page is the initialization of a program, which calls a function `work` that is not shown. Assume that the only optimization your C compiler is allowed to perform is register allocation of local variables whose address has not been taken. Improve the speed of the code as much as you can for a RISC processor, on which all instructions except the following take one clock cycle (i.e. one clock cycle in the execute pipeline stage). Note that C compilers need to use several instructions for the `%` operator (remainder).

Instruction	**Clock cycles**
Divide	30
Multiplication	4
Memory load	2
Mispredicted branch	4

The function `rand` generates a random number (which you must use).

```
#include <stdio.h>

unsigned int x = 0;

void work(void);
void error(const char*);

static unsigned int f(unsigned int a, unsigned int b)
{
        unsigned int    c = 1;

        while (b-- > 0)
                c = c * a;
        return c;
}

static void g(unsigned int s, unsigned int r)
{
        x = 15 * x + r % s;
        x += x / s;
}

int main(int argc, char** argv)
{
        unsigned int    i;
        unsigned int    s;
        unsigned int    n;

        if (argc != 2 || 1 != sscanf(argv[1], "%u", &n))
                error("expected a number");

        for (i = 0; i < n; ++i) {
                s = f(2, i);
                g(s, rand());
        }

        work();

        return 0;
}
```

Answer *See the file 20110530.c for original and optimized versions. Remarks on the optimized code:*

- *Both* `f` *and* `g` *can be inlined by the programmer.*
- *The global variable* `x` *can be copied to a local variable* `y` *which can be register allocated — according to the question the global variable could not be register allocated.*
- *The function* `f` *is not really needed and what it calculates can be done more efficiently, as can seen.*