Exam in EDAF15 Algorithm Implementation

May 31, 2018, 14-19

The results will be announced through email on Saturday, June 2 at 15:00

Inga hjälpmedel!

You may answer in English, på svenska, auf Deutsch, или по-русски.

Examiner: Jonas Skeppstedt

30 out of 60p are needed to pass the exam and your grade is $\lfloor$your score/10$\rfloor$.

In this course there is always exactly one thread of execution and no parallelization is allowed.

1. (10p) In a pipelined processor it can sometimes happen that multiple iterations of a loop are executing concurrently. Consider a loop with two floating point instructions i_1 and i_2 which have a true data dependence between them: i_1 produces a values which i_2 consumes, or, in other words i_1 writes to a register r, and i_2 reads the value of r. Assume there are no cyclic dependences (i_1 does read the output from i_2 from the previous iteration) i.e. that each iteration is independent except for a loop index variable (which you safely can ignore in this question).

   Which three essential hardware features make it possible to have two or more loop iterations with i_1 and i_2 executing concurrently, and what does each of these do?

2. (10p) Cache memories

   (a) (2p) What is the purpose of having sets in a cache — why are caches not "simplified" by having only one set?

   For the remaining questions, assume a cache is divided into sets where each set contains four cache blocks.

   (b) (1p) What is such a cache called?

   (c) (2p) Can the data at a memory address A be put in any set? Why or why not?

   (d) (2p) In any of the four cache blocks of a set? Why or why not?

   (e) (1p) What is meant by cache levels?

   (f) (2p) Consider a simple five-stage pipelined RISC processor. What is the reason for having separate first level caches for data and instructions? That is relevant in superscalar processors as well but easier to see in a simpler processor.

3. (20p) Implement a circular double linked list where an empty list is represented by `NULL`, with the functions declared below.

```
#include <stdlib.h>
typedef struct list_t   list_t;
struct list_t {
        void            data;
        list_t*         succ;
        list_t*         pred;
};

/* (2p) create a new list node with this data. */
list_t* new_list(void* data);
```

```
/* (3p) deallocate entire list but not any data pointer. */
void free_list(list_t* list);


/* (1p) return the number of nodes in the list. */
size_t length(list_t* list);

/* (3p) insert data first in the list. */
void insert_first(list_t** list, void* data);

/* (3p) insert data last in the list. */
void insert_last(list_t** list, void* data);

/* (3p) if the list is empty, return NULL, otherwise
 * remove (and free) the first node in the list
 * and return its data pointer.
 */
void* remove_first(list_t** list);

/* (5p) Use malloc to allocate memory for an array and
 * return a pointer to an array with the contents (data
 * pointers) of the list, and write the length of the
 * list in the variable pointed to by size. If the size
 * is zero, NULL should be returned.
 *
 * Note: the word array above is used in the sense that
 * memory should be allocated for a number of elements
 * in contiguous memory locations and not as in array
 * declaration (i.e. not like this: int a[10];)
 *
 */
void** list_to_array(list_t* list, size_t* size);
```

4. (10p) After profiling a big application program a colleague discovers that calls to `malloc` and `free` related to the lists take up a significant fraction of the execution time, and asks you what to do about it. After discussing how the lists are used, you two conclude that:

   - The most time consuming use (use = creating, inserting, removing etc) of lists is in a part of the code which you are not allowed to modify.
   - The second most time consuming code uses `list_to_array` for lists usually with fewer than 400 list nodes, and very rarely more than 1000 list nodes, but in principle there is no limit to the lengths of the lists. However, you are allowed to modify the code which uses `list_to_array`.

   - (5p) How can you help your colleague for the first case by modifying your list implementation? Write C code!
   - (5p) Make a new function similar to `list_to_array` but more suitable for the function `work` below, and modify `work` to exploit it.

```
void work(list_t* list)
{
        void**          a;
        size_t          n;

        a = list_to_array(list, &n);
        process_array(a, n);
        free(a);
}
```

5. (10p) Consider the following code:

```
void f(unsigned int a, signed int b)
{
        g(a / 16);
        h(b / 16);
}
```

   Assume you are using a very bad C compiler which is unable to optimize `f` and you decide to manually optimize `f` instead. Explain why and how >> easily can be used in the call to `g` but not so easily in the call to `h`. Also assume that for `signed int`, >> is arithmetic shift as in Java (and not logical shift as Java's >>>).