55:132/22C:160
High Performance Computer Architecture
Spring 2011

Fifth Homework Assignment
Due: Thursday, March 31, in-class

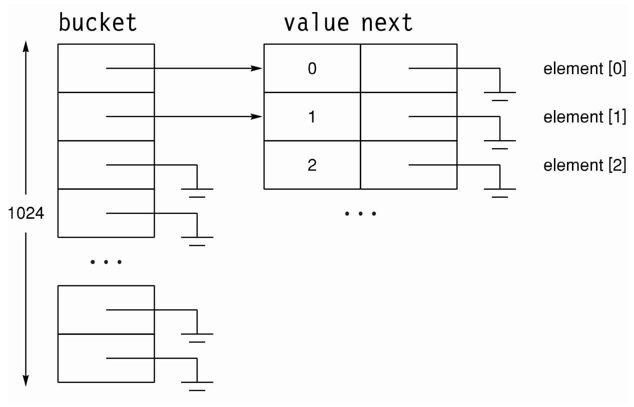
The purpose of this assignment is to investigate the potential for instruction-level parallelism in real programs. You will do this by investigating a simple code example.

A hash table is a popular data structure for organizing a large collection of data items so that one can quickly answer questions such as, “Does an element of value 100 exist in the collection?” This is done by assigning data elements into one of a large number of buckets according to a hash function value generated from the data values. The data items in each bucket are typically organized as a linked list sorted according to a given order. A lookup of the hash table starts by determining the bucket that corresponds to the data value in question. It then traverses the linked list of data elements in the bucket and checks if any element in the list has the value in question. As long as one keeps the number of data elements in each bucket small, the search result can be determined very quickly.

The C source code shown below inserts a large number (N_ELEMENTS) of elements into a hash table, whose 1024 buckets are all linked lists sorted in ascending order according to the value of the elements. The array `element[]` contains the elements to be inserted, allocated on the heap. Each iteration of the outer (for) loop, starting at line 6, enters one element into the hash table.
Line 9 calculates `hash_index`, the hash function value, from the data value stored in `element[i]`. The hashing function used is a very simple one; it consists of the least significant 10 bits of an element’s data value. This is done by computing the bitwise logical AND of the element data value and the (binary) bit mask 11 1111 1111 (1023 in decimal).

```
1   typedef struct _Element {
2       int value;
3       struct _Element *next;
4   } Element;
5   Element element[N_ELEMENTS], *bucket[1024];
    /* The array element is initialized with the items to be
    inserted; the pointers in the array bucket are
    initialized to NULL. */

6   for (i = 0; i < N_ELEMENTS; i++)
    {
7       Element *ptrCurr, **ptrUpdate;
8       int hash_index;
        /* Find the location at which the new element is to
        be inserted. */
9       hash_index = element[i].value & 1023;
10      ptrUpdate = &bucket[hash_index];
11      ptrCurr = bucket[hash_index];
        /* Find the place in the chain to insert the new
        element. */
12      while (ptrCurr &&
13          ptrCurr->value <= element[i].value)
14      {
15          ptrUpdate = &ptrCurr->next;
16          ptrCurr = ptrCurr->next;
        }
        /* Update pointers to insert the new element into
        the chain. */
17      element[i].next = *ptrUpdate;
18      *ptrUpdate = &element[i];
     }
```



The above figure illustrates the hash table data structure used in our C code example. The bucket array on the left side of the figure is the hash table. Each entry of the bucket array contains a pointer to the linked list that stores the data elements in the bucket. If bucket i is currently empty, the corresponding `bucket[i]` entry contains a NULL pointer. In Figure 3.15, the first three buckets contain one data element each; the other buckets are empty.

Variable `ptrCurr` contains a pointer used to examine the elements in the linked list of a bucket. At Line 11, `ptrCurr` is set to point to the first element of the linked list stored in the given bucket of the hash table. If the bucket selected by the `hash_index` is empty, the corresponding bucket array entry contains a NULL pointer.

The while loop starts at line 12. Line 12 tests if there are any more data elements to be examined by checking the contents of variable `ptrCurr`. Lines 13 through 16 will be skipped if there are no more elements to be examined, either because the bucket is empty, or because all the data elements in the linked list have been examined by previous iterations of the while loop. In the first case, the new data element will be inserted as the first element in the bucket. In the second case, the new element will be inserted as the last element of the linked list.

In the case where there are still more elements to be examined, line 13 tests if the current linked list element contains a value that is smaller than or equal to that of the data element to be inserted into the hash table. If the condition is true, the while loop will continue to move on to the next element in the linked list; lines 15 and 16 advance to the next data element of the linked list by moving `ptrCurr` to the next element in the linked list. Otherwise, it has found the position in the linked list where the new data element should be inserted; the while loop will terminate and the new data element will be inserted right before the element pointed to by `ptrCurr`.

The variable `ptrUpdate` identifies the pointer that must be updated in order to insert the new data element into the bucket. It is set by line 10 to point to the bucket entry. If the bucket is empty, the while loop will be skipped altogether and the new data element is inserted by changing the pointer in `bucket[hash_index]` from NULL to the address of the new data element by line 18. After the while loop, `ptrUpdate` points to the pointer that must be updated for the new element to be inserted into the appropriate bucket.
After the execution exits the while loop, lines 17 and 18 finish the job of inserting the new data element into the linked list. In the case where the bucket is empty, `ptrUpdate` will point to `bucket[hash_index],` which contains a NULL pointer. Line 17 will then assign that NULL pointer to the next pointer of the new data element. Line 18 changes `bucket[hash_table]` to point to the new data element. In the case where the new data element is smaller than all elements in the linked list, `ptrUpdate` will also point to `bucket[hash_table],` which points to the first element of the linked list. In this case, line 17 assigns the pointer to the first element of the linked list to the next pointer of the new data structure.

In the case where the new data element is greater than some of the linked list elements but smaller than the others, `ptrUpdate` will point to the next pointer of the element after which the new data element will be inserted. In this case, line 17 makes the new data element to point to the element right after the insertion point. Line 18 makes the original data element right before the insertion point to point to the new data element. The reader should verify that the code works correctly when the new data element is to be inserted to the end of the linked list.

Now that we have a good understanding of the C code, we will proceed with analyzing the amount of instruction-level parallelism available in this piece of code. We will focus on the amount of instruction-level parallelism available to the *run time hardware scheduler* under the most favorable execution scenarios (the ideal case). Specifically, assume that the hash table is initially empty. Suppose there are 512 new data elements, whose values are numbered sequentially from 0 to 511, so that each goes in its own bucket (this reduces the problem to a matter of updating known array locations!). Figure 3.15 shows the hash table contents after the first three elements have been inserted, according to this "ideal case." Since the `value` of `element[i]` is simply `i` in this ideal case, each element is inserted into its own bucket.

Further assume that each line of source code takes one execution cycle and, for the purposes of computing ILP, takes one instruction. These assumptions greatly simplify bookkeeping in solving the following questions. Note that the for and while statements execute on each iteration of their respective loops, to test if the loop should continue. In this ideal case, most of the dependences in the code sequence are relaxed and a high degree of ILP is therefore readily available.

Further suppose that the code is executed on an "ideal" processor with infinite issue width, unlimited renaming, "omniscient" knowledge of memory access disambiguation, branch prediction, and so on, so that the execution of instructions is limited only by data dependence. Consider the following in this context:

a. Describe the data (true, anti, and output) and control dependences that govern the parallelism of this code segment, as seen by a run time hardware scheduler. Indicate only the *actual* dependences (i.e., ignore dependences between stores and loads that access different addresses, even if a compiler or processor would not realistically determine this). Draw the *dynamic* dependence graph for six consecutive iterations of the outer loop (for insertion of six elements), under the ideal case. Note that in this dynamic dependence graph, we are identifying data dependences between dynamic instances of instructions: each static instruction in the original program has multiple dynamic instances due to loop execution. *Hint:* The following definitions may help you find the dependences related to each instruction:
   Data true dependence: On the results of which previous instructions does each instruction immediately depend?
   Data antidependence: Which instructions subsequently write locations read by the instruction?
   Data output dependence: Which instructions subsequently write locations written by the instruction?
   Control dependence: On what previous decisions does the execution of a particular instruction depend (in what case will it be reached)?

b. Assuming the ideal case just described, and using the dynamic dependence graph you just constructed, how many instructions are executed, and in how many cycles?

c. What is the average level of ILP available during the execution of the for loop?

d. For simplicity, assume that only variables `i, hash_index, ptrCurr,` and `ptrUpdate` plus the pointer to the base of `element[]` array need to occupy registers. Assuming general renaming and that registers are recycled as soon as their contents are no longer needed, how many registers are necessary to achieve the maximum achievable parallelism in part (b)?

e. Assume that in your answer to part (a) there are 7 instructions in each iteration. Now, assuming a consistent steady-state schedule of the instructions in the example and an issue rate of 2 instructions per cycle, how is execution time affected?

f. Finally, calculate the minimal instruction window size needed to achieve the maximal level of parallelism in part (a).