LSU EE 4720 **Homework 6** Solution **Due: Not Collected**

To answer the questions below you need to use the PSE dataset viewer program. PSE (pronounced see) runs on Solaris and Linux; you can use the computer accounts distributed in class to run it, a Linux distribution may also be provided for running it on other systems.

Procedures for setting up the class account and using PSE are at `http://www.ece.lsu.edu/ee4720/proc.html`; preliminary documentation for PSE is at `http://www.ece.lsu.edu/ee4720/pse.pdf`.

Problem 1: The code in `http://www.ece.lsu.edu/ee4720/2002f/hw6.pdf` includes two routines to perform a linear search, `lookup_array` and `lookup_ll`. Routine `lookup_array(aws,foo)` searches `aws` for element `foo`. The list itself is an ordinary C array, structure `aws` (array with size) includes the array and its size. Routine `lookup_ll(head,foo)` searches for `foo` in the linked list starting at `head`.

The code calls the search routines under realistic conditions: Before the linked list is allocated dynamic storage is fragmented and before the searches are performed the level-1 cache is flushed. See the code for more details.

The code was executed on a simulated 4-way superscalar dynamically scheduled machine with a 64-entry reorder buffer and a two-level cache. The simulation was recorded in `hw6.ds`; view this dataset file using PSE to answer the questions below.

The code initializes the lists with identical data and then calls the search routines looking for the same value. Answer the following questions about the execution of the two lookup routines. When browsing the dataset be aware that the time spent in the lookup routines is dwarfed by the time needed for setting everything up and so only the last few segments need to be examined.

(*a*) Would increasing the ROB size improve the performance of the linked list routine, `lookup_ll`? Explain.

No. The speed of the linked list routine depends upon how often the load hits the level-one cache. In the first PED below there are many misses, in the second there are many hits.

Without resorting to a dataflow graph one can conclude that a larger ROB won't help by noting that in the first case the only instruction that executes early (execution is shown in yellow), `inc`, does not provide a value needed by the instructions blocking the head of the reorder buffer. So, with a larger ROB `inc` would be fetched earlier but that would do nothing for the instructions blocking the head.

In the second case the ROB does not fill, and so a larger ROB will make no difference.

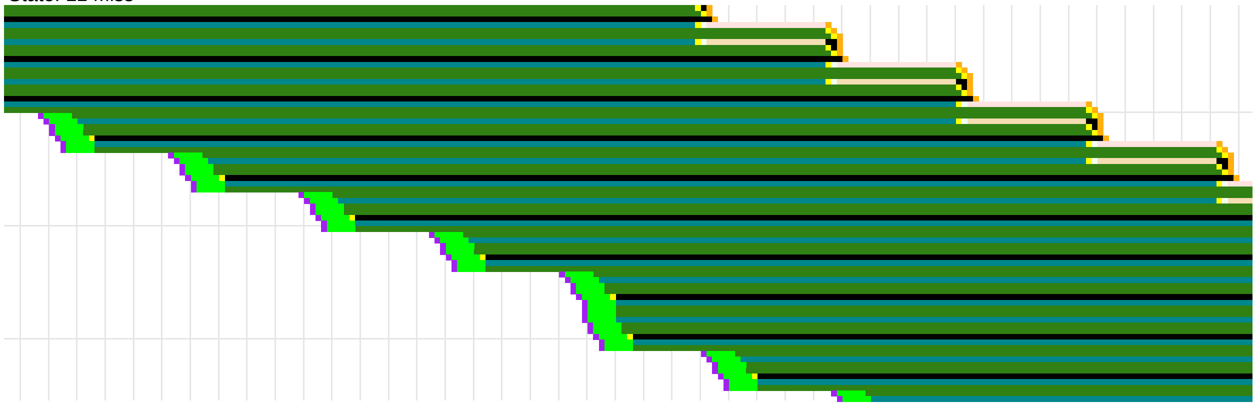
Using a dataflow graph (or just eyeballing the seven instructions) one finds that the critical path for an iteration of this loop is one instruction long, `0x1083c ld[o0+4,o0]`. (If the critical path were measured in instructions rather than cycles then `inc` would also be a critical path, but it only takes one cycle. The other instructions are not on the critical path because they do not produce a value that is needed in the next cycle. Note that if branches weren't predicted they would be on the critical path.) If the load hits the level 1 cache then the critical path is just two cycles long (one cycle for the address, here address computation is on the critical path unlike the unoptimized π routine from Homework 5), if the load misses it is much longer, 23 cycles.

With a critical path of 23 cycles per iteration and seven instructions per iteration, the processor is already executing the loop as fast as it can. The ROB just fills with mostly waiting instructions. A larger ROB won't help.

When there are lots of hits, near the end of the loop, the critical path is just two cycles per iteration. As noted earlier, the ROB is less than half full, so increasing its size won't help. This wasn't asked, but since we're here we might as will determine if this code is executing as fast as it can. The ideal CPI for this code is $\frac{2}{7} \approx 0.286$, which is attainable on a 4-way superscalar processor. The code is actually executing at $\frac{3}{7} \approx 0.429$, the problem is fetch inefficiency. That is, because the three instructions starting at address `0x10848` lie on two aligned groups it takes three cycles to fetch the 7-cycle loop.

Rank: 1/122   Pos. 120/122
0.14 IPC over 219 cycles.
State: L1 Miss

First Instruction:.LLM10 lookup_ll+3 hw6.c:74
ld [%o0], %g2

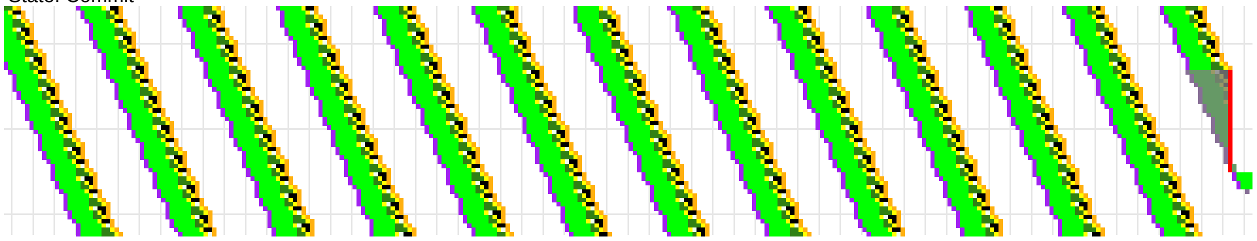


Time 238,182 Tag 425,274 PC 0x00010830

Grid 20 insn X 5 cyc

Rank: 46/122   Pos. 121/122
2.31 IPC over 292 cycles.
State: Commit

First Instruction:.LLM10+2 lookup_ll+5 hw6.c:74
bne,a   0x10848 {lookup_ll+9}

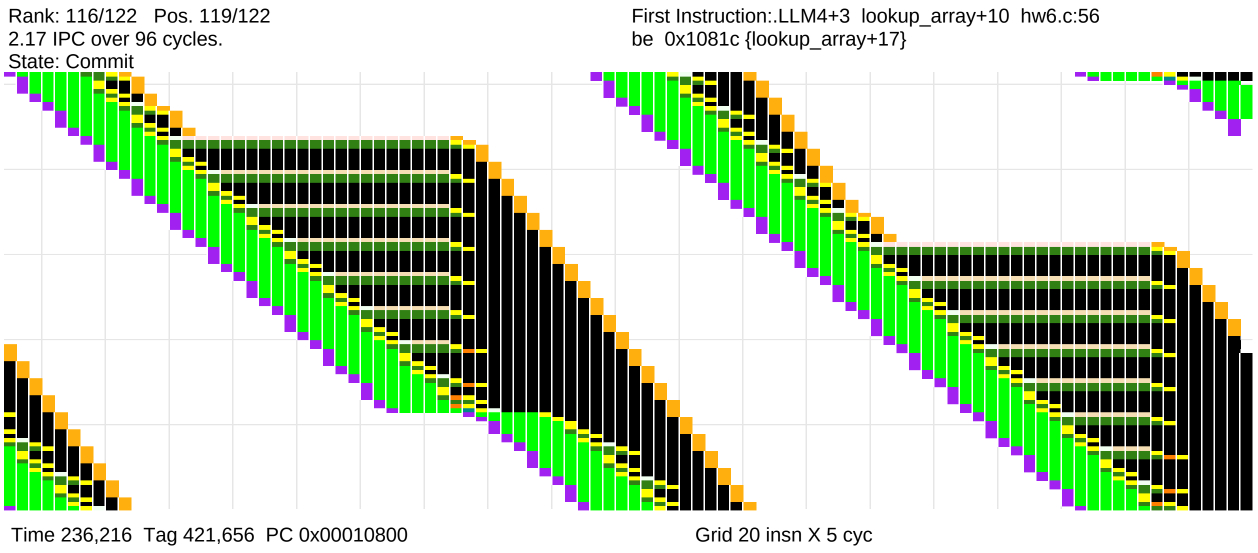


Time 241,199 Tag 427,936 PC 0x00010838

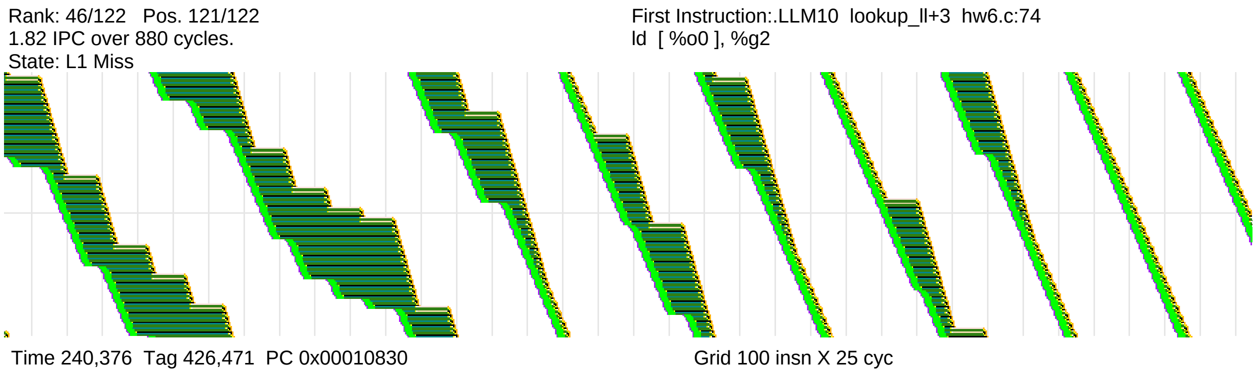
Grid 20 insn X 5 cyc

(*b*) Would increasing the ROB size improve the performance of the array routine, `lookup_array`? Explain.

Yes, because the instructions allowed in with a larger reorder buffer would be able to execute. As can be seen from the PED below, there are many instructions that execute (shown in yellow) right after the pre-ready state. When the ROB fills there are instructions that could execute but aren't being fetched because of the full ROB, and so a larger ROB would help these. Unlike the linked list routine, the load that misses the cache is not on the critical path, so a load miss does not leave the processor with little to do.



(*c*) As can be seen viewing the PED plots (for example, the one below), the array routine follows a regular pattern while the linked list code starts off slowly but as it nears completion it runs much faster. Why does the linked list code speed up like that?



A line can hold multiple linked list elements. An L1 miss brings in the linked list element which is currently needed but also brings in others that will be needed later. As the search proceeds more and more of the elements will be found in the cache.

(*d*) How could one determine the line size from the PED plots? Be specific and use numbers from the dataset. (The line size can be found two other ways, if you come upon them by all means use them to check your answer that is based on the PED plot.)

The array routine accesses memory consecutively and each element is four bytes. It is accessed by the load instruction at `0x107f8`. Looking at the PED above it is easy to find load misses. Since the array is not in the level-one cache when the routine starts every access to a new line must miss. To find the line size count the number of loads between misses. There are 16, and so the line size must be 64 bytes.

(*e*) Before people stopped replacing $2,500 computers every six months computer engineers would loose sleep worrying about The Memory Wall, the growing gap in performance between processors and memory (*e.g.*, the number of instructions that could have been executed while waiting for memory). What is it about the array routine that lets it sail over the memory wall while the linked list routine is stopped dead? The answer should take into account certain load instructions and the critical path. Discuss how the performance of the routines change as the L1 miss time gets longer and longer.

The reason the linked list suffers is because the load for the second element can't start until the load for the first element finishes. In contrast, since an array is laid out sequentially in memory, there is no need to load the first element to find out where the second is located. The array code enjoys two advantages here. First, a miss to a single line will bring several array elements, second, it is possible to overlap misses to two lines (if the ROB is large enough, which it isn't). Therefore, even if the gap between memory and processor speed continues to widen, the performance of the array code won't suffer much as long as the ROB is made larger and the memory can handle multiple overlapping misses.

(Linked lists do have their advantages, but should not be used where an array would suffice.)