Jenny Nguyen jtn2497

L1 Miss Rates:

	50x50	100x100	200x200	400x400	800x800	1200x1200	1600x1600	2000x2000
i-j-k	5.74%	6.00%	6.38%	21.54%	19.46%	30.76%	33.63%	45.73%
j-i-k	5.66%	6.24%	7.90%	25.66%	21.47%	29.65%	32.22%	43.81%
j-k-i	4.57%	5.60%	18.90%	29.43%	30.53%	34.99%	38.71%	56.06%
k-j-i	4.26%	5.64%	19.07%	29.30%	30.51%	34.79%	38.17%	55.23%
i-k-j	4.81%	4.95%	4.96%	5.01%	5.02%	9.53%	9.97%	9.96%
k-i-j	5.15%	5.10%	5.01%	5.01%	5.06%	9.44%	9.79%	9.77%

L2 Miss Rates:

	50x50	100x100	200x200	400x400	800x800	1200x1200	1600x1600	2000x2000
i-j-k	1.14%	8.01%	87.99%	22.56%	89.70%	52.07%	50.19%	60.64%
j-i-k	3.98%	2.39%	4.51%	1.94%	80.10%	52.28%	52.70%	59.85%
j-k-i	4.21%	4.28%	21.82%	16.52%	72.31%	61.73%	59.34%	68.80%
k-j-i	2.58%	6.01%	21.72%	17.99%	72.07%	62.79%	57.63%	67.87%
i-k-j	1.28%	4.67%	1.84%	1.84%	1.94%	0.95%	0.98%	0.96%
k-i-j	1.48%	1.90%	4.10%	3.53%	3.16%	2.04%	2.05%	2.10%

Total Load and Store Instructions:

	50x50	100x100	200x200	400x400	800x800	1200x1200	1600x1600	2000x2000
i-j-k	502656	4010158	32040182	256160462	2048644970	6913456272	16386600652	32004087866
j-i-k	502656	4010158	32040194	256160386	2048643794	6913453242	16386589702	32004057556
j-k-i	630158	5020158	40080188	320320848	2561287134	8642908982	20485188520	40008132408
k-j-i	630156	5020158	40080210	320321060	2561287324	8642906222	20485183674	40008126640
i-k-j	630156	5020158	40080174	320320308	2561281388	8642884542	20485131808	40008021650
k-i-j	630156	5020158	40080174	320320326	2561281402	8642885018	20485133256	40008023334

Total Floating Point Instructions:

	50x50	100x100	200x200	400x400	800x800	1200x1200	1600x1600	2000x2000
i-j-k	1005292	8079736	65599354	543839139	4544616422	15469828262	37253424647	72473671542
j-i-k	1001473	8012822	64903713	536723608	4569870479	15455959270	37244519834	72808572847
j-k-i	875468	7004663	56577922	532396403	4273265069	14332473071	34401443217	67365291501
k-j-i	875317	7010273	56531165	538468216	4354980669	14589002900	34953302461	68328361069
i-k-j	875108	7002841	56033834	448176241	3586162749	12103768206	28687455346	56028296642
k-i-j	875210	7007120	56031235	448202327	3587553428	12108364348	28705414022	56063362514

For the smallest matrix size, do the L1 and L2 miss rates vary for the different loop-order variants? Do they vary for the larger matrix sizes? Is there any difference in behavior between the different problem sizes? Can you explain intuitively the reasons for this behavior?

For the smallest matrix size 50x50, L1 and L2 miss rates don't vary by much. For L1, they range from 4.26% to 5.74%, and for L2, they range from 1.14% to 4.21%. There is likely low variation among these numbers because the matrix is small enough to fit within the L1 and L2 caches, leading to less cache misses overall. However, as the matrix size grows, there is much more variation between the different loop-order variants. For example, for 2000x2000, L1 miss rates for i-k-j and k-i-j variants are around 10%, while j-k-i and k-j-i are around 55-56%. The L2 miss rate for i-k-j is only 0.96%, while the miss rate for j-k-i is 68.80%. This behavior is probably due to spatial locality. For i-k-j and k-i-j, j is the innermost loop. Because of this, the whole cache line is fetched, resulting in low miss rates. On the other hand, for j-k-i and k-j-i, i is the innermost loop. The code jumps across rows for every iteration, and by the time the code goes back to the next element in a previous row, it has already been evicted from the cache, resulting in higher miss rates.

Re-instrument your code by removing PAPI calls, and using clock_gettime with CLOCK_THREAD_CPUTIME_ID to measure the execution times for the six versions of MMM and the eight matrix sizes specified above. How do your timing measurements compare to the execution times you obtained from using PAPI? Repeat this study using CLOCK_REALTIME. Explain your results briefly.

The clock_gettime measurements for both CLOCK_THREAD_CPUTIME_ID and CLOCK_REALTIME pretty much match up with the PAPI measurements. For the matrix size 2000x2000, the i-k-j variant finishes in approximately 33 seconds, and it has an L1 miss rate of 9.96% and L2 miss rate of 0.96%. The j-k-i variant finishes in approximately 223 seconds, and it has an L1 miss rate of 56.06% and L2 miss rate of 68.80%. Based on these measurements, the execution time seems to be closely related to the cache miss rates. The results also show that CLOCK_THREAD_CPUTIME_ID and CLOCK_REALTIME were nearly identical.

What is Moore's Law? What is Amdahl's Law? Which of these is an empirical observation and which of these is a mathematical fact?

Moore's Law states that the number of transistors on a microchip doubles approximately every two years, and Amdahl's Law states that the performance improvement is limited by the lower components. Moore's Law is an empirical observation, and Amdahl's is a mathematical fact.

In the earliest ISA's, memory could only be accessed using the absolute addressing mode. What problems arise in implementing loops with such an ISA? How do we get around these problems in today's ISA's?

With absolute addressing, memory references had to use a literal, hardcoded address. Each memory address had to be written explicitly. For loops, it would cause problems if you're indexing into an array with a loop counter or iterating through an array. Today's ISA's use a base plus an offset for addressing, so you can now access memory relative to the address.

What are *data and control dependences?* Give simple examples to illustrate these concepts.

Data dependences occur when an instruction needs to use the result of a previous one. For example:

```
ADD X1, X2, X3
SUB X4, X1, X5.
```

The SUB instruction uses the X1 register, so it can't complete until the X1 register is ready from the ADD instruction.
Control dependences occur when the execution of an instruction depends on the outcome of a branch. For example:

```
CMP X1, X3
BEQ foo
```

Whether the instructions at foo are executed depends on the outcome of the CMP instruction.

Explain *out-of-order execution* and *in-order retirement/commit*. Why do high-performance processors execute instructions out of order but retire them in order? What hardware structure(s) are used to implement in-order retirement?

Out-of-order execution means that instructions aren't executed in the order that they're written in to keep the CPU busy without having to wait on instructions that have dependencies. In-order retirement/commit means that the registers and memory are still updated in program order even if they are executed out of order. So, high-performance processors execute instructions out of order to improve performance but retire them in order to ensure that register and memory updates follow program order and prevent younger instructions from overwriting architectural state before the older ones finish. Hardware structures that are used to implement in-order retirement include the reorder buffer (ROB), which stores the results of finished instructions until it's their turn to retire.

Consider the invariants for retirement in OOO execution with renaming shown in the lecture slides. Why do we need to check the condition "(R3.PR# = ROB[n].PR#") before updating R3.v ? Explain what would go wrong if we did not check this condition before updating R3.v.

We need to check the condition to make sure we write the correct and newest value into register R3 when an instruction retires. "R3.PR# = ROB[n].PR#" checks if the retiring instruction is the latest one that writes to R3, and if it is, it updates R3.v. If the condition is false, it doesn't update it because there is a newer instruction that changed R3. If we didn't check this condition before updating R3.v, then the physical register might contain the value of an older instruction even if a younger instruction writes to the same register.

What is the typical branch prediction accuracy in modern processors? In lecture, we said "A correctly predicted branch is essentially a NO-OP as far as performance is concerned." Explain this statement in a few sentences.

The typical branch prediction accuracy in modern processors is approximately 95%. The statement means that when the branch is correctly predicted, the instruction pipeline continues with no interruptions. The correct future instructions have already been fetched and begun executing, so it just continues from here as if the branch logic was never even there. Because of this, no performance is lost from delays or “bubbles” caused by a mispredicted branch.