Computer Architecture (263-2210-00L), Fall 2017
HW 5: Multicore Execution, Runahead Execution, Prefetching
SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Hasan Hassan, Arash Tavakkol, Mohammad Sadr, Lois Orosa, Juan Gomez Luna

Assigned: Sunday, Nov 26, 2017
Due: **Wednesday, Dec 13, 2017**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture/`. Please check your inbox. You should have received an email with the password you can use to login to the paper review system. If you have not received any email, please contact comparch@lists.ethz.ch. In the first page after login, you should click in "Architecture - Fall 2017 Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-6).** Please upload your solution to the Moodle (`https://moodle-app2.let.ethz.ch/`) as a single PDF file. **Please use a typesetting software (e.g., LaTeX) or a word processor (e.g., MS Word, LibreOfficeWriter) to generate your PDF file. Feel free to draw your diagrams either using an appropriate software or by hand, and include the diagrams into your solutions PDF.**

1 Critical Paper Reviews [200 points]

Please read the following handout on how to write critical reviews. We will give out extra credit that is worth 0.5% of your total grade for each good review.

- Lecture slides on guidelines for reviewing papers. Please follow this format. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-f17-how-to-do-the-paper-reviews.pdf`
- Some sample reviews can be found here: `https://safari.ethz.ch/architecture/fall2017/doku.php?id=readings`

(a) Write a one-page critical review for the first paper of the following list and at least **one** of the other 3 papers:

- O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003. `https://people.inf.ethz.ch/omutlu/pub/mutlu_hpca03.pdf`
- M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching" ISCA 2007. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=p381-qureshi.pdf`
- N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=1-jouppi.pdf`
- D. Josephand and D. Grunwald, "Prefetching using Markov Predictors," ISCA 1997. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=18-2-joseph-prefetching.pdf`

2 Running Ahead [200 points]

Consider the following program, running on an in-order processor with no pipelining:

```
LD   R1   ←  (R3)      // Load A
ADD  R2   ←  R4, R6
LD   R9   ←  (R5)      // Load B
ADD  R4   ←  R7, R8
LD   R11  ←  (R16)     // Load C
ADD  R7   ←  R8, R10
LD   R12  ←  (R11)     // Load D
ADD  R6   ←  R8, R15
```

Assume that all registers are initialized and available prior to the beginning of the shown code. Each load takes 1 cycle to execute, and each add takes 2 cycles to execute. Loads A through D are all cache misses. In addition to the 1 cycle taken to execute each load, these cache misses take 6, 9, 12, and 3 cycles, respectively for Loads A through D, to complete. For now, assume that no penalty is incurred when entering or exiting runahead mode.

Note: Please show all your work for partial credit.

(a) For how many cycles does this program run *without* runahead execution?

42 cycles.

(b) For how many cycles does this program run *with* runahead execution?

27 cycles. See Table 1.

(c) How many additional instructions are executed in runahead execution mode?

14 instructions. See Table 1.

(d) Next, assume that exiting runahead execution mode incurs a penalty of 3 cycles. In this case, for how many cycles does this program run *with* runahead execution?

4 runahead periods in total. $27 + 4 \times 3 = 39$ cycles

(e) At least how many cycles should the runahead exit penalty be, such that enabling runahead execution decreases performance? Please show your work.

4 cycles. $(27 + 4 \times X > 42)$

(f) Which load instructions cause runahead periods? Circle all that did:

$\boxed{LoadA}$ $\boxed{LoadB}$ $\boxed{LoadC}$ $\boxed{LoadD}$

For each load that caused a runahead period, tell us if the period generated a prefetch request. If it did not, explain why it did not and what type of a period it caused.

Table 1: Execution timeline for Problem 2(b), using runahead execution with no exit penalty and no optimizations.

Cycle	Operations		Enter/Exit Runahead Mode	Runahead Mode?	Is Additional Runahead Instruction?
1	Load 1 Issue		enter RA		
2	Add 1			Yes	Yes
3	Add 1			Yes	Yes
4	Load 2 Issue			Yes	Yes
5	Add 2			Yes	Yes
6	Add 2			Yes	Yes
7	Load 1 Finish	Load 3 Issue	exit RA	Yes	Yes
8	Add 1				
9	Add 1				
10	Load 2 Issue, already issued		enter RA		
11	Add 2			Yes	Yes
12	Add 2			Yes	Yes
13	Load 2 Finish	Load 3 Issue, already issued	exit RA	Yes	Yes
14	Add 2				
15	Add 2				
16	Load 3 Issue, already issued		enter RA		
17	Add 3			Yes	Yes
18	Add 3			Yes	Yes
19	Load 3 Finish	Load 4 Issue, dependent on R11 from Load 3. Load 3 finishes at the end of cycle 19, so R11 isn't available now.	exit RA	Yes	Yes
20	Add 3				
21	Add 3				
22	Load 4 Issue		enter RA		
23	Add 4			Yes	Yes
24	Add 4			Yes	Yes
25	Load 4 Finish		exit RA	Yes	
26	Add 4				
27	Add 4				

Load A:	This period generated prefetches
Load B:	Short runahead period. / Overlapping runahead periods.
Load C:	Dependent loads. / Short runahead period.
Load D:	No loads to prefetch in this period. (Useless runahead period)

(g) For each load that caused a runahead period that did not result in a prefetch, explain how you would best mitigate the inefficiency of runahead execution.

Load A:	N/A. Prefetch was generated.
Load B:	Do not enter runahead mode if only a few cycles are left before the corresponding cache miss will be filled. Do not enter runahead mode if the subsequent instruction(s) have already been executed during a previous runahead period.
Load C:	Predict the value of cache-miss address loads.
Load D:	Predict if a period will generate useful cache misses, and do not execute that period if it's deemed useless.

(h) If all useless runahead periods were eliminated, how many additional instructions would be executed in runahead mode?

Since only the first runahead period generated prefetches, only that runahead period will be executed. According to Table 1, 6 instructions are executed in the first runahead period.

How does this number compare with your answer from part (c)?

Less.

(i) Assume still that the runahead exit penalty is 3 cycles, as in part (d). If all useless runahead execution periods were eliminated (i.e., runahead execution is made *efficient*), for how many cycles does the program run with runahead execution?

One runahead period, so $27 + 1 \times 3 = 30$ cycles.

How does this number compare with your answer from part (d)?

Less.

(j) At least how many cycles should the runahead exit penalty be such that enabling *efficient runahead execution* decreases performance? Please show your work.

16 cycles. $(27 + 1 \times X > 42)$

3 Prefetching [150 points]

An architect is designing the prefetch engine for his machine. He first runs two applications A and B on the machine, with a stride prefetcher.

Application A:

```
uint8_t a[1000];
sum = 0;
for (i = 0; i < 1000; i += 4)
{
    sum += a[i];
}
```

Application B:

```
uint8_t a[1000];
sum = 0;
for (i = 1; i < 1000; i *= 4)
{
    sum += a[i];
}
```

i and sum are in registers, while the array a is in memory. A cache block is 4 bytes in size.

(a) What is the prefetch accuracy and coverage for applications A and B using a stride prefetcher. This stride prefetcher detects the stride between two consecutive memory accesses and prefetches the cache block at this stride distance from the currently accessed block.

> **Application A's prefetch accuracy is 248/249 and coverage is 248/250.**
> Application A accesses a[0], a[4], a[8], ... a[996]. It has 1000/4 = 250 accesses to memory.
> The first two accesses to a[0] and a[4] are misses. After observing these two accesses, the prefetcher learns that the stride is 4 and starts prefetching a[8], a[12], a[16] and so on until a[1000] (on the access to a[996], a[1000] is prefetched; however, it is not used). In all, 249 cache blocks are prefetched, while only 248 are used.
> Hence, the prefetch accuracy is 248/249 and coverage is 248/250.
>
> **Application B's prefetch accuracy is 0 and coverage is 0.**
> Application B accesses a[1], a[4], a[16], a[64] and a[256]. It has five accesses to memory.
> However, there isn't a constant stride between these accesses, as the array index is multiplied by 4, rather than being incremented/decremented by a constant stride. Hence, the stride prefetcher cannot prefetch any of the accessed cache blocks and the prefetch accuracy and coverage are both 0.

(b) Suggest a prefetcher that would provide better accuracy and coverage for
i) application A?

> A next block prefetcher would always prefetch the next cache block. Hence, the cache block containing a[4] would also be prefetched. Therefore, the prefetch accuracy would be 249/250 and coverage would be 249/250.

ii) application B?

> Most common prefetchers such as stride, stream, next block would not improve the prefetch accuracy of application B, as it does not have an access pattern for which prefetchers are commonly designed. Some form of pre-execution, such as runahead execution or dual-core execution could help improve its prefetch accuracy.

(c) Would you suggest using runahead execution for
i) application A. Why or why not?

No. While runahead execution could still provide good prefetch accuracy even for a regular access pattern as in application A, it might not be possible to prefetch all cache blocks before they are accessed, as runahead execution happens only when the application is stalled on a cache miss. A stride prefetcher or next block prefetcher, on the other hand could possibly prefetch all cache blocks before they are accessed, as prefetching happens in parallel with the application's execution.

ii) application B. Why or why not?

Yes. Application B's memory access pattern does not have a regular access stride. Commonly used prefetchers are not designed to prefetch an access pattern like this. However, runahead execution could potentially prefetch some cache blocks, especially as the address of the cache blocks does not depend on the data from a pending cache miss.

4 More Prefetching [200 points]

A processor is observed to have the following access pattern to cache blocks. Note that the addresses are **cache block addresses**, not byte addresses. This pattern is repeated for a large number of iterations.

`Access Pattern P:  A,  A + 3,  A + 6,  A,  A + 5`

Each cache block is 8KB. The hardware has a fully associative cache with LRU replacement policy and a total size of 24KB.

None of the prefetchers mentioned in this problem employ confidence bits, but they all start out with empty tables at the beginning of the access stream shown above. Unless otherwise stated, assume that 1) each access is separated long enough in time such that all prefetches issued can complete before the next access happens, and 2) the prefetchers have large enough resources to detect and store access patterns.

(a) You have a stream prefetcher (i.e., a next-N-block prefetcher), but you dont know the prefetch degree (N) of it. However, you have a magical tool that displays the coverage and accuracy of the prefetcher. When you run a large number of repetitions of access pattern P, you get 40% coverage and 10% accuracy. What is the degree of this prefetcher (how many next blocks does it prefetch)?

Next 4 blocks.
40% coverage with a stream prefetcher for this pattern means blocks A+3 and A+6 are prefetched. Possible N at this point are 3 and 4. Accuracy 10% = 2/(N*5), so N is 4.

(b) You didnt like the performance of the stream prefetcher, so you switched to a PC-based stride prefetcher that issues prefetch requests based on the stride detected for each memory instruction. Assume all memory accesses are incurred by the *same* load instruction (i.e., the same PC value) and the initial stride value for the prefetcher is set to 0.

Circle which of the cache block addresses are prefetched by this prefetcher:

`A,  A + 3,  A + 6,  A,  A + 5`
`A,  A + 3,  A + 6,  A,  A + 5`
`A,  A + 3,  A + 6,  A,  A + 5`
`A,  A + 3,  A + 6,  A,  A + 5`

Explain:

This prefetcher remembers the last stride and applies that to prefetch the next block from the current access.

(c) Stride prefetcher couldnt satisfy you either. You changed to a Markov prefetcher with a correlation table of 12 entries (assume each entry can store a single address to prefetch, and remembers the most recent correlation). When all the entries are filled, the prefetcher replaces the entry that is least-recently accessed.

Circle which of the cache block addresses are prefetched by this prefetcher:

`A,  A + 3,  A + 6,  A,  A + 5`
`A,  A + 3,  A + 6,  A,  A + 5`
`A,  A + 3,  A + 6,  A,  A + 5`
`A,  A + 3,  A + 6,  A,  A + 5`

Explain:

All entries are filled after the first repetition, except the entry for A+5. Accesses to A+3 and A+5 thrash each other for block As next-block entry, so they cannot be correctly prefetched.

(d) Just in terms of coverage, after how many repetitions of access pattern P does the Markov prefetcher from part (c) start to outperform the stream prefetcher from part (a), if it can at all? Show your work.

5 repetitions.
Coverage for Markov prefetcher from part (c) is (0+2+3*(N-2))/(5N). This is equal to 40% when N=4, so it outperforms 40% at the 5th repetition. We gave full credit if you wrote either 4 or 5, depending on the work shown

(e) You think having a correlation table of 12 entries makes the hardware too costly, and want to reduce the number of correlation table entries for the Markov prefetcher. What is the minimum number of entries that gives the same prefetcher performance as 12 entries? Similar to the last part, assume each entry can store a single next address to prefetch, and remembers the most recent correlation. Show your work.

4 entries.
With 4 different accesses, we need at least 4 entries.

(f) Your friend is running the same program on a different machine that has a Markov prefetcher with 2 entries. The same assumptions from part (e) apply.

Circle which of the cache block addresses are prefetched by the prefetcher:

```
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
```

Explain:

NONE. Not enough entries. Entries will be evicted before they are used again.

(g) As an avid computer architect, you decide to update the processor by increasing the cache size to 32KB with the same cache block size. Assume you will be only running a program with the same access pattern

P for a large number of iterations (i.e., one trillion), describe a prefetcher that provides smaller memory bandwidth consumption than the baseline without a prefetcher.

Explain:

No prefetcher. Since all lines will sit in the cache after the first iteration, the prefetcher needs to achieve 100% accuracy on the first iteration in order to consume the same amount of memory bandwidth as the baseline, which is not possible without any training on the prefetcher beforehand.
Note: We leave it up to you to think whether or not you can ever design a prefetcher that leads to less memory consumption than the baseline without a prefetcher.

5 Markov Prefetchers vs. Runahead Execution [100 points]

(a) Provide two advantages of runahead execution over Markov prefetchers.

(i) Does not require recording of miss addresses.
(ii) More accurate, as it executes actual code.
(iii) (Mostly) Uses pre-existing hardware.
(vi) Less bandwidth wasted. Markov will issue N prefetches/miss.
(v) Can prefetch compulsory misses (i.e. does not need to see an address previously as a miss to be able to prefetch it)

(b) Provide two advantages of Markov prefetchers over runahead execution.

(i) Is not limited by the branch prediction accuracy as it is not pre-execution based.
(ii) Can prefetch dependent misses.
(iii) Does not need to execute instructions to generate prefetches.

(c) Describe one memory access pattern in which runahead execution performs better than Markov prefetchers. Show pseudo-code.

If the access pattern did not repeat, runahead execution would perform better, as Markov prefetchers cannot avoid compulsory misses.

```
uint8_t a[1000];
sum = 0;
for (i = 0; i < 1000; i ++)
{
sum += a[i];
}
```

(d) Describe one memory access pattern in which runahead execution performs worse than Markov prefetchers. Show pseudo-code.

Linked list traversal that happens multiple times. Runahead cannot prefetch dependent misses whereas markov is able to trigger the miss for the next node when the previous node is visited (assuming that the linked list was traversed previously and the correlation table still contains the correlations between node addresses).

```
while (node != NULL) {
...
node = node->next;
}
```

6 Parallel Speedup [200 points]

You are a programmer at a large corporation, and you have been asked to parallelize an old program so that it runs faster on modern multicore processors.

(a) You parallelize the program and discover that its speedup over the single-threaded version of the same program is significantly less than the number of processors. You find that many cache invalidations are occuring in each core's data cache. What program behavior could be causing these invalidations (in 20 words or less)?

> Cache ping-ponging due to (inefficient or poorly-designed) data sharing.

(b) You modify the program to fix this first performance issue. However, now you find that the program is slowed down by a global state update that must happen in only a single thread after every parallel computation. In particular, your program performs 90% of its work (measured as processor-seconds) in the parallel portion and 10% of its work in this serial portion. The parallel portion is perfectly parallelizable. What is the maximum speedup of the program if the multicore processor had an infinite number of cores?

> Use Amdahl's Law: for n processors, Speedup(n) = $\frac{1}{0.1+\frac{0.9}{n}}$
> As $n \to \infty$, Speedup(n) $\to 10$

(c) How many processors would be required to attain a speedup of 4?

> 6 processors.
> Let Speedup(n) = 4 (from above) and solve:
> $4 = 1/(0.1 + \frac{0.9}{n})$
> $0.25 = 0.1 + \frac{0.9}{n}$
> $0.15 = \frac{0.9}{n}$
> $n = 6$

(d) In order to execute your program with parallel and serial portions more efficiently, your corporation decides to design a custom heterogeneous processor.

- This processor will have one large core (which executes code more quickly but also takes greater die area on-chip) and multiple small cores (which execute code more slowly but also consume less area), all sharing one processor die.
- When your program is in its parallel portion, all of its threads execute **only** on small cores.
- When your program is in its serial portion, the one active thread executes on the large core.
- Performance (execution speed) of a core is proportional to the square root of its area.
- Assume that there are 16 units of die area available. A small core must take 1 unit of die area. The large core may take any number of units of die area n^2, where n is a positive integer.
- Assume that any area not used by the large core will be filled with small cores.

(i) How large would you make the large core for the fastest possible execution of your program?

> 4 units.
> For a given large core size of n^2, then the large core yields a speedup of n on the serial section, and there are $16 - n^2$ small cores to parallelize the parallel section. Speedup is thus $1/(\frac{0.1}{n}+\frac{0.9}{16-n^2})$. To maximize speedup, minimize the denominator. One can find that for $n = 1$, the denominator is 0.16. For $n = 2$, the denominator is 0.125. For $n = 3$, the denominator is 0.1619 (this can be approximated without a calculator: 0.0333 plus $0.90/7 > 0.12$ is greater than 0.15, thus worse than $n = 2$.) Hence, $n = 2$ is optimal, for a large core of $n^2 = 4$ units.

(ii) What would the same program's speedup be if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

$Speedup = 1/(0.1 + \frac{0.9}{16}) = 6.4$

(iii) Does it make sense to use a heterogeneous system for this program which has 10% of its work in serial sections?
Why or why not?

Yes.
The serial portion of the program is large enough that speedup of the serial portion with the large core speedup outweighs loss in parallel throughput due to the large core.

(e) Now you optimize the serial portion of your program and it becomes only 4% of total work (the parallel portion is the remaining 96%).

(i) What is the best choice for the size of the large core in this case?

4 units.
Same as above, we can calculate n^2. Now the speedup is $1/(\frac{0.04}{n} + \frac{0.96}{16-n^2})$. Again, $n = 2$ maximizes the speedup.

(ii) What is the program's speedup for this choice of large core size?

$1/(\frac{0.04}{2} + \frac{0.96}{12}) = 10$

(iii) What would the same program's speedup be for this 4%/96% serial/parallel split if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

$1/(0.04 + \frac{0.96}{16}) = 1/0.1 = 10$

(iv) Does it make sense to use a heterogeneous system for this program which has 4% of its work in serial sections?
Why or why not?

No.
The heterogeneous system is more complex to design, but the performance is the same for this program.