ETH 263-2210-00L Computer Architecture, Fall 2020

HW 5: Parallelism, Mutiprocessors, Bottleneck Acceleration, Prefetching, Cache Coherence, Memory Consistency(SOLUTIONS)

Instructor: Prof. Onur Mutlu

TAs: Mohammed Alser, João Dinis Ferreira, Rahul Bera, Geraldo Francisco De Oliveira Junior, Can Firtina, Juan Gómez Luna, Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Jeremie Kim, Nika Mansouri Ghiasi, Haiyu Mao, Lois Orosa Nogueira, Jisung Park, Minesh Patel, Gagandeep Singh, Kosta Stojiljkovic, Abdullah Giray Yaglikci

Given: Monday, Dec 7, 2020

Due: **Monday, Dec 21, 2020**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture20/`. Please check your inbox. You should have received an email with the password you should use to login. If you did not receive any emails, contact comparch@lists.inf.ethz.ch. In the first page after login, you should click in "Computer Architecture Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2–8).** You should upload your answers to the Moodle Platform (`https://moodle-app2.let.ethz.ch/course/view.php?id=13549`) as a single PDF file.

1. Critical Paper Reviews [1000 points]

Please read the guidelines for reviewing papers and check the sample reviews. We also assign you a **required reading** for this homework. You may access them by *simply clicking on the QR codes below or scanning them.* If you review a paper other than the REQUIRED papers, you will receive 250 BONUS points on top of 1000 points you may get from paper reviews (i.e., each additional submission is worth 250 BONUS points with a possibility to get up to 3000 points).

Guidelines

Sample reviews

Required Reading 1

Required Reading 2

Required Reading 3

Required Reading 4

Write an approximately one-page critical review for the following required reading (i.e., papers from #1 to #4) and earn *bonus* points for the remaining twelve papers (i.e., papers from #5 to #16). A review with bullet point style is more appreciated. Try not to use very long sentences and paragraphs. Keep your writing and sentences simple. Make your points bullet by bullet, as much as possible.

1. **(REQUIRED)** M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," In *ASPLOS*, 2009. `https://people.inf.ethz.ch/omutlu/pub/acs_asplos09.pdf`
2. **(REQUIRED)** O. Mutlu, J. Stark, C. Wilerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processor," In *HPCA*, 2003. `https://people.inf.ethz.ch/omutlu/pub/mutlu_hpca03.pdf`
3. **(REQUIRED SHORT)** G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," In *AFIPS*, 1967. `https://safari.ethz.ch/architecture/fall2018/lib/exe/fetch.php?media=lecture1-amdahl.pdf`
4. **(REQUIRED SHORT)** L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE TC*, 1979. `https://safari.ethz.ch/architecture/fall2019/lib/exe/fetch.php?media=how-to-make-a-multiprocessor-computer-that-correctly-executes-multiprocess-programs.pdf`
5. J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," In *ASPLOS*, 2012. `https://people.inf.ethz.ch/omutlu/pub/bottleneck-identification-and-scheduling_asplos12.pdf`
6. N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," In *ISCA*, 1990. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=1-jouppi.pdf`
7. S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," In *HPCA*, 2007. `https://people.inf.ethz.ch/omutlu/pub/srinath_hpca07.pdf`
8. E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," In *HPCA*, 2009. `https://people.inf.ethz.ch/omutlu/pub/bandwidth_lds_hpca09.pdf`
9. C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," In *ISCA*, 2001. `https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=luk-isca-2001.pdf`
10. K. Gharachorioo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," In *ICPP*, 1991. `https://courses.engr.illinois.edu/cs533/sp2019/reading_list/gharachorloo91two.pdf`
11. M. S. Papamarcos and J. H. Patel, "A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories," In *ISCA*, 1984. `https://course.ece.cmu.edu/~ece447/s12/lib/exe/fetch.php?media=wiki:papamarcos84.pdf`
12. A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K.T. Malladi, H. Zheng, and O. Mutlu, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," In *ISCA*, 2019. `https://people.inf.ethz.ch/omutlu/pub/CONDA-coherence-for-near-data-accelerators_isca19.pdf`
13. R. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," In ISCA, 1999. `https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=chappell-isca-1999.pdf`
14. C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," In *MICRO*, 2008. `https://people.inf.ethz.ch/omutlu/pub/prefetch-dram_micro08.pdf`
15. D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," In *ISCA*, 1997. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=18-2-joseph-prefetching.pdf`
16. O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engine," In *ISCA* 2005, *MICRO TOP PICKS* 2006. `https://users.ece.cmu.edu/~omutlu/pub/mutlu_ieee_micro06.pdf`

2. 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.

3. Asymmetric Multicore [400 points]

A microprocessor manufacturer asks you to design an asymmetric multicore processor for modern workloads. You should optimize it assuming a workload with 80% of its work in the parallel portion. Your design contains one large core and several small cores, which share the same die. Assume the total die area is 32 units.

- *Large core*: For a large core that is n times faster than a single small core, you will need n^3 units of die area (n is a positive integer). The dynamic power of this core is $6 \times n$ Watts and the static power is n Watts.
- *Small cores*: You will fit as many small cores as possible, after placing the large core. A small core occupies 1 unit of die area. Its dynamic power is 1 Watt and its static power is 0.5 Watts.

The parallel portion executes *only* on the small cores, while the serial portion executes *only* on the large core.

Please answer the following questions. Show your work. Express your equations and solve them. You can approximate some computations, and get partial or full credit.

(a) What configuration (i.e., number of small cores and size of the large core) results in the best performance?

One large core and 24 small cores. The large core will occupy 8 units of die area.

Explanation:
Given that the large core occupies n^3 units, the number of small cores will be $32 - n^3$. Thus, the speedup can be calculated as:
$Speedup = \frac{1}{\frac{0.2}{n} + \frac{0.8}{32-n^3}}$.

Without loss of generality, we assume that the total execution time is:
$t_{total} = t_{serial} + t_{parallel} = \frac{0.2}{n} + \frac{0.8}{32-n^3}$ seconds.

n	$\#small$	t_{serial}	$t_{parallel}$	t_{total}
1	31	0.20	0.03	0.23
2	24	0.10	0.03	**0.13**
3	5	0.07	0.16	0.23

These calculations can be approximated without a calculator:

n	$\#small$	t_{serial}	$t_{parallel}$	t_{total}
1	31	0.20 / 1 = 0.20	$0.02 < 0.80 / 31 < 0.03$	> 0.22
2	24	0.20 / 2 = 0.10	$0.03 < 0.80 / 24 < 0.04$	$<$ **0.14**
3	5	0.20 / 3 = 0.07	0.80 / 5 = 0.16	> 0.22

(b) The energy consumption should also be a metric of reference in your design. Compute the energy consumption for the best configuration in part (a).

$E_{total} = 26 \times t_{serial} + 38 \times t_{parallel} = 3.74$ Joules.

Explanation:
We can calculate the energy consumption as:
$E_{total} = E_{large} + E_{small} =$
$(P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + P_{large\_static} \times t_{parallel}$
$+ (P_{small\_static} \times t_{serial} + (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel}) \times (32 - n^3) =$
$7 \times n \times t_{serial} + n \times t_{parallel} + (0.5 \times t_{serial} + 1.5 \times t_{parallel}) \times (32 - n^3) =$
$14 \times t_{serial} + 2 \times t_{parallel} + 12 \times t_{serial} + 36 \times t_{parallel} =$
$26 \times t_{serial} + 38 \times t_{parallel} = 3.74$ Joules.

This result can be approximated without a calculator:
$E_{total} < 26 \times 0.10 + 38 \times 0.04 = 2.6 + 1.52 = 4.12$ Joules.

(c) For the best configuration obtained in part (a), you are considering to use the large core to collaborate with the small cores on the execution of the parallel portion.

(i) What is the overall performance improvement, compared to the performance obtained in part (a), if the large core collaborates on the parallel portion?

If the large core collaborates with the small cores in the parallel portion, the best-case speedup can be calculated as:
$Speedup = \frac{1}{\frac{0.2}{n} + \frac{0.8}{32 - n^3 + n}}$.

Without loss of generality, we assume that the total execution time is:
$t_{total} = t_{serial} + t_{parallel} = \frac{0.2}{n} + \frac{0.8}{32 - n^3 + n}$ seconds.

The execution time of the serial part t_{serial}, which takes significantly longer than the parallel part (about 3 times longer), does not change. By using the large core to collaborate in the parallel portion, the execution time of the parallel part $t_{parallel}$ decreases from $\frac{0.8}{24}$ to $\frac{0.8}{24+2}$, i.e., a speedup of $\frac{13}{12}$, which is less than 10%. Thus, the overall performance improvement from using the large core to collaborate in the parallel portion is negligible.

(ii) What is the overall energy change, compared to the energy obtained in part (b), if the large core collaborates on the parallel portion?

If the large core collaborates in the parallel portion, we calculate the energy consumption as:

$E_{total} = E_{large} + E_{small} =$
$(P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + (P_{large\_dynamic} + P_{large\_static}) \times t_{parallel}$
$+ (P_{small\_static} \times t_{serial} + (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel}) \times (32 - n^3) =$
$7 \times n \times t_{serial} + 7 \times n \times t_{parallel} + (0.5 \times t_{serial} + 1.5 \times t_{parallel}) \times (32 - n^3) =$
$14 \times t_{serial} + 14 \times t_{parallel} + 12 \times t_{serial} + 36 \times t_{parallel} =$
$26 \times t_{serial} + 50 \times t_{parallel} \simeq 2.6 + 2.0 = 4.6$ Joules.

We assume that $t_{parallel}$ has a very small change, as discussed above. If we compare this equation to the energy equation in part (b), we observe that the energy consumption increases by $P_{large\_dynamic} \times t_{parallel} = 6 \times n \times t_{parallel} = 12 \times t_{parallel}$ Joules. Since the energy consumption of the parallel portion is $38 \times t_{parallel}$ Joules in part (b), there is an energy increase in the parallel portion of more than 30% (i.e., $\frac{12}{38}$). The overall energy increase is more than 11%.

(iii) Discuss whether it is worth using the large core to collaborate with the small cores on the execution of the parallel portion.

It is not really worth using the large core in the parallel part. While the performance improvement is negligible, the overall energy consumption increases by more than 11%.

(d) Now assume that the serial portion can be optimized, i.e., the serial portion becomes smaller. This gives you the possibility of reducing the size of the large core, and still improving performance. For a large core with an area of $(n-1)^3$, where n is the value obtained in part (a), what should be the fraction of serial portion that would lead to better performance than in part (a)?

10%.

Explanation:
We call t_{total} the total execution time with a large core with $n = 2$, as obtained in part (a), and t'_{total} for a smaller core with $n = 1$. We can obtain the new parallel fraction p from the following equation:

$t_{total} > t'_{total}$;

$0.13 > \frac{1-p}{n-1} + \frac{p}{32-(n-1)^3}$;

$0.13 > \frac{1-p}{1} + \frac{p}{31}$;

$p > 0.90$.

The serial portion should be *at most* 10%.

(e) Your design is so successful for desktop processors that the company wants to produce a similar design for mobile devices. The power budget becomes a constraint. For a maximum of total power of 20W, how much would you need to reduce the dynamic power consumption of the large core, if at all, for the best configuration obtained in part (a)? Assume again that the parallel fraction is 80% of the workload. (Hint: Express the dynamic power of the large core as $D \times n$ Watts, where D is a constant).

We have to reduce the dynamic power consumption of the large core by *at least* 20×.

Explanation:
We calculate the total power as the total energy divided by the total execution time:
$P_{total} = \frac{E_{total}}{t_{total}}$ Watts;

$P_{total} = \frac{E_{large}+E_{small}}{t_{total}} \leq 20$ Watts;

We express the dynamic power of the large core as $D \times n$. From part (a) we know n, t_{serial}, $t_{parallel}$ and t_{total}, from part (b) we know E_{small}:

$\frac{(D+1)\times n\times t_{serial}+n\times t_{parallel}+E_{small}}{t_{total}} = \frac{(D+1)\times 2\times 0.10+n\times 0.03+2.00}{0.13} \leq 20$ Watts;

$D \leq 0.3$.

In mobile devices, the dynamic power of the large core has to be $\leq 0.3 \times n$ Watts (given the assumptions in the question). Since the dynamic power of the large core is $6 \times n$ Watts in the desktop processor, we have to reduce the dynamic power consumption of the large core by *at least* 20× for mobile devices.

4. Runahead Execution [200 points]

Assume an in-order processor that employs Runahead execution, with the following specifications:

- The processor enters Runahead mode when there is a cache miss.
- There is no penalty for entering and leaving the Runahead mode.
- There is a 64KB data cache. The cache block size is 64 bytes.
- Assume that the instructions are fetched from a separate dedicated memory that has zero access latency, so an instruction fetch never stalls the pipeline.
- The cache is 4-way set associative and uses the LRU replacement policy.
- A memory request that hits in the cache is serviced instantaneously.
- A cache miss is serviced from the main memory after X cycles.
- A cache block for the corresponding fetch is allocated *immediately* when a cache miss happens.
- The cache replacement policy does *not* evict the cache block that triggered entry into Runahead mode until after the Runahead mode is exited.
- The victim for cache eviction is picked at the same time a cache miss occurs, i.e., during cache block allocation.
- ALU instructions and Branch instructions take one cycle.
- Assume that the pipeline *never stalls* for reasons *other than data cache misses.* Assume that the conditional branches are always correctly predicted and the data dependencies do not cause stalls (except for data cache misses).

Consider the following program. Each element of Array `A` is one byte.

```
for (i = 12; i < 100; i++) { \\ 2 ALU instructions and 1 branch instruction
  int m = A[i*16*1024] + 1;  \\ 1 memory instruction followed by 1 ALU instruction
  ...                        \\ 26 ALU instructions
}
```

(a) After running this program using the processor specified above, you find that there are 66 data cache hits. What are **all** the possible values of the cache miss latency X? You can specify all possible values of X as an inequality. Show your work.

$61 < X < 93$.

Explanation. The program makes 100 memory accesses in total. To have 66 cache hits, a cache miss needs to be followed by 2 cache hits. Hence, the Runahead engine needs to prefetch 2 cache blocks. After getting a cache miss and entering Runahead mode, the processor needs to execute 30 instructions to reach the next LD instruction. To reach the LD instruction 2 times, the processor needs to execute at least 62 instructions (2*(29 ALU + 1 Branch + 1 LD)) in Runahead mode. If the processor spends more than 92 cycles in Runahead mode, then it will prefetch 3 cache lines instead of two, which will cause the number of cache hits to be different. Thus, the answer is $61 < X < 93$.

(b) Is it possible that *every* memory access in the program misses in the cache? If so, what are **all** possible values of X that will make all memory accesses in the program miss in the cache? If not, why? Show your work.

Yes, for $X < 31$ and $X > 123$.

Explanation. When X is equal to or smaller than 30 cycles, the processor will be in Runahead mode for insufficient amount of time to reach the next LD instruction (i.e., the next cache miss). Thus, none of the data will be prefetched and all memory accesses will get cache miss.
When X is larger than 123 cycles, the processor will prefetch 4 cache blocks. Since the prefetched cache blocks will map to the same cache set, the latest prefetched cache block will evict the first prefetched cache block in Runahead mode (note that the cache block that triggered Runahead execution remains in the cache due to the cache block replacement policy). This will cause a cache miss in the next iteration after leaving the Runahead mode. Thus, the accesses in the program will always miss in the cache.

(c) What is the *minimum* number of cache misses that the processor can achieve by executing the above program? Show your work.

25 cache misses.

Explanation. When $92 < X < 124$, the Runahead engine will prefetch exactly 3 cache blocks that will be accessed after leaving the Runahead mode. It is the minimum number of misses that could be achieved since all cache blocks accessed by the program map to the same cache set and the cache is 4-way associative.

5. Prefetching [200 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.

6. Cache Coherence [300 points]

We have a system with 4 byte-addressable processors. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. Coherence is maintained using the Illinois Protocol (MESI), which sends an invalidation to other processors on writes, and the other processors invalidate the block in their caches if *the block is present* (NOTE: On a write hit in one cache, a cache block in Shared state becomes Modified in that cache).

Accessible memory addresses range from 0x50000000 − 0x5FFFFFFF. Assume that the offset within a cache block is 0 for all memory requests. We use a snoopy protocol with a shared bus.

Cosmic rays strike the MESI state storage in your coherence modules, causing the state of a *single* cache line to instantaneously change to another state. This change causes an inconsistent state in the system. We show below the initial tag store state of the four caches, *after* the inconsistent state is induced.

Inital State

Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	I

Cache 1		
	Tag	*MESI state*
Set 0	0x522222	I
Set 1	0x510000	S
Set 2	0x5FFFFF	S
Set 3	0x533333	S

Cache 2		
	Tag	*MESI state*
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFF	S
Set 3	0x533333	S

Cache 3		
	Tag	*MESI state*
Set 0	0x5FF000	E
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(a) What is the inconsistency in the above initial state? Explain with reasoning.

Cache 2, Set 1 should be in S state. Or Cache 3, Set 1 should be in I state.

Explanation. If the MESI protocol performs correctly, it is *not* possible for the same cache line to be in S and E states in different caches.

(b) Consider that, after the initial state, there are several paths that the program can follow that access different memory instructions. In b.1-b.4, we will examine whether the followed path can potentially lead to incorrect execution, i.e., an incorrect result.

b.1) Could the following path potentially lead to incorrect execution? Explain.

order	**Processor 0**	**Processor 1**	**Processor 2**	**Processor 3**
1^{st}			ld 0x51110040	
2^{nd}	st 0x5FFFFF40			
3^{rd}				st 0x51110040
4^{th}		ld 0x5FFFFF80		
5^{th}		ld 0x51110040		
6^{th}		ld 0x5FFFFF40		

No.

Explanation. The 3^{rd} instruction (st 0x51110040 in Processor 3) will invalidate the same line in Processor 2, and the whole system will be back to a consistent state (only one valid copy of 0x51110040 in the caches). Thus, the originally-inconsistent state does not affect the architectural state.

b.2) Could the following path potentially lead to incorrect execution? Explain.

order	**Processor 0**	**Processor 1**	**Processor 2**	**Processor 3**
1^{st}				ld 0x51110040
2^{nd}	ld 0x5FFFFF00			
3^{rd}			ld 0x51234540	
4^{th}	st 0x5FFFFF40			
5^{th}				ld 0x51234540
6^{th}	ld 0x5FFFFF00			

Yes.

Explanation. The 1^{st} instruction could read invalid data. This would be the case if the cosmic-ray-induced change was from I to S in Cache 3 for cache line 0x51110040.

After some time executing a particular path (which could be a path *different* from the paths in parts b.1-b.4) and with no further state changes caused by cosmic rays, we find that the final state of the caches is as follows.

Final State

Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	E

Cache 1		
	Tag	*MESI state*
Set 0	0x5FF000	I
Set 1	0x510000	S
Set 2	0x5FFFFF	S
Set 3	0x533333	I

Cache 2		
	Tag	*MESI state*
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFF	S
Set 3	0x533333	I

Cache 3		
	Tag	*MESI state*
Set 0	0x5FF000	M
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(c) What is the *minimum* set of memory instructions that leads the system from the initial state to the final state? Indicate the set of instructions in order, and clearly specify the access type (ld/st), the address of each memory request, and the processor from which the request is generated.

The minimum set of instructions is:

(1) st 0x533333C0 // Processor 0
(2) ld 0x5FFFFFC0 // Processor 0
(3) ld 0x5FF00000 // Processor 1
(4) st 0x5FF00000 // Processor 3

Alternatively, as instructions (1)(2) and instructions (3)(4) touch different cache lines, we just need to keep the order between (1)(2), and between (3)(4). These are valid reorderings: (3)(4)(1)(2), (1)(3)(2)(4), (3)(1)(4)(2), (1)(3)(4)(2) or (3)(1)(2)(4).

Explanation.

(1) The instruction sets the line 0x533333C0 to M state in Cache 0, and invalidates the line 0x533333C0 in Cache 1 and Cache 2.
(2) The instruction evicts 0x533333C0 from Cache 0, and sets the line 0x5FFFFFC0 to E state in Cache 0.
(3) The instruction sets the line 0x5FF00000 to S state in Cache 1, as well as in Cache 3.
(4) The instruction sets the line 0x5FF00000 to M state in Cache 3, and it invalidates the line 0x5FF00000 in Cache 1.

7. Memory Consistency [300 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

Thread T0		**Thread T1**	
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`	Instr. T1.1	`*flag = 1;`
Instr. T0.2	`while(*flag == 0);`	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

`X`, `Y`, and `flag` have been allocated in main memory, while `a` and `b` are contained in processor registers. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a *single* instruction.

(a) What is the final value of `Y[0]` in the SC processor, after both threads finish execution? Explain your answer.

2.

Explanation. `Y[0]` is set equal to 1 by instruction T1.0. Then, it will be incremented by instruction T0.3. The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag`. Thread 0 will remain in instruction T0.2 until `flag` is set by T1.1, i.e., after `Y[0]` is initialized. So, instruction T0.3 must be executed after instruction T1.0, causing `Y[0]` to be first set to 1 and then incremented.

(b) What is the final value of `b` in the SC processor, after both threads finish execution? Explain your answer.

0 or 1.

Explanation. There are *at least* two possible sequentially-consistent orderings that lead to *at most* two different values of `b` at the end:
Ordering 1: T1.0 $\rightarrow$ T0.1 - Final value = 1.
Ordering 2: T0.1 $\rightarrow$ T1.0 - Final value = 0.

With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called RC, that implements *weak consistency.* As discussed in the lecture, the weak consistency model has no need to guarantee a strict order of memory operations. For this question, consider a very weak model where there is *no* guarantee on the ordering of instructions as seen by different cores.

(c) What is the final value of `Y[0]` in the RC processor, after both threads finish execution? Explain your answer.

1 or 2.

Explanation. Since there is no guarantee of a strict order of memory operations, as seen by different cores, instruction T1.1 could complete before or after instruction T1.0, from the perspective of the core that executes T0. If instruction T1.1 completes before instruction T1.0, from the perspective of the core that executes T0, instruction T0.3 could complete before or after instruction T1.0. Thus, there are three possible weakly-consistent orderings that lead to different values of `Y[0]` at the end:
Ordering 1 (from the perspective of T0): T1.0 $\rightarrow$ T1.1 $\rightarrow$ T0.3 - Final value = 2.
Ordering 2 (from the perspective of T0): T1.1 $\rightarrow$ T1.0 $\rightarrow$ T0.3 - Final value = 2.
Ordering 3 (from the perspective of T0): T1.1 $\rightarrow$ T0.3 $\rightarrow$ T1.0 - Final value = 1.

After several months spent debugging her code, the programmer learns that the new processor includes a `memory_fence()` instruction in its ISA. The semantics of `memory_fence()` is as follows for a given thread that executes it:

1. Wait (stall the processor) until *all* preceding memory operations from the thread complete in the memory system and become visible to other cores.
2. Ensure *no* memory operation from any later instruction in the thread gets executed before the `memory_fence()` is retired.

(d) What *minimal* changes should the programmer make to the program above to ensure that the final value of `Y[0]` on RC is the same as that in part (a) on SC? Explain your answer.

Use memory fences before T1.1 and after T0.2.

Explanation. The memory fence before instruction T1.1 stalls thread 1 until instruction T1.0 has completed, i.e., ensures that `Y[0]` is initialized to 1 before the flag is set. Thread 0 waits in the loop T0.2 until the flag is set. The memory fence after instruction T0.2 ensures that instruction T0.3 will not happen until the memory fence is retired. Thus, instruction T0.3 will also complete *after* the flag is set. The modified code will be as follows:

	Thread T0		**Thread T1**
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`		**memory_fence();**
Instr. T0.2	`while(*flag == 0);`	Instr. T1.1	`*flag = 1;`
	memory_fence();	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

8. BONUS: Building Multicore Processors [250 points]

You are hired by Amdahl's Nano Devices (AND) to design their newest multicore processor. Ggl, one of AND's largest customers, has found that the following program can predict people's happiness.

```
for (i = 12; i < 2985984; i++) {
  past = A[i-12];
  current = A[i];
  past *= 0.37;
  current *= 0.63;
  A[i] = past + current;
}
```

A is a large array of 4-byte floating point numbers, gathered by Ggl over the years by harvesting people's private messages. Your job is to create a processor that runs this program as fast as possible.

Assume the following:

- You have magically fast DRAM that allows infinitely many cores to access data in parallel. We will relax this strong assumption in parts (d), (e), (f).
- Each floating point instruction (addition and multiplication) takes 10 cycles.
- Each memory read and write takes 10 cycles.
- No caches are used.
- Integer operations and branches are fast enough that they can be ignored.

(a) Assuming infinitely many cores, what is the maximum steady state speedup you can achieve for this program? Please show all your computations.

The cycle counts are extra information that are not necessary.
Instead, the loop body is sequential, while 12 consecutive iterations are parallel. Notice that the 13th iteration is dependent on the 1st iteration.
Therefore $\frac{1}{12}$ of the program is serial. Using Amdahl's law, this solves to a maximum speedup of 12.

(b) What is the minimum number of cores you need to achieve this speedup?

12 cores are needed.

(c) Briefly describe how you would assign work to each core to achieve this speedup.

Assign iteration `i` to processor `i%12`.

It turns out magic DRAM does not exist except in Macondo[1]. As a result, you have to use cheap, slow, low-bandwidth DRAM. To compensate for this, you decide to use a private L1 cache for each processor. The new specifications for the DRAM and the L1 cache are:

- DRAM is shared by all processors. DRAM may only process one request (read or write) at a time.
- DRAM takes 100 cycles to process any request.
- DRAM prioritizes accesses of smaller addresses and write requests. (Assume no virtual memory)
- The cache is direct-mapped. Each cache block is 16 bytes.
- It takes 10 cycles to access the cache. Therefore, a cache hit is processed in 10 cycles and a cache miss is processed in 110 cycles.

All other latencies remain the same as specified earlier. Answer parts (d), (e), (f) assuming this new system.

(d) Can you still achieve the same steady state speedup as before? Circle one: YES NO
Please explain.

No. Notice that the serial bottleneck is now caused by the DRAM. In steady state, only one memory access is performed for every four iterations. So four iterations take 100 + 200 = 300 cycles. 12 iterations requires 900 cycles.
In order to achieve 12x speedup, each core needs to complete each iteration in 75 cycles. However, fetching from memory alone requires 100 cycles. Therefore, it is not possible.

By just saying that it is now slower than before is not enough, since the single core baseline is also slower.

(e) What is the minimum number of cores your processor needs to provide the maximum speedup?

3 cores are needed to take the maximum advantage of each cache hit, and a maximum speed up of almost 3 can be achieved.
Core 1 calculates iterations i+0, i+1, i+2, i+3, core 2 calculates iteration i+4, i+5, i+6, i+7, and core 3 calculates iteration i+8, i+9, i+10, i+11.

(f) Briefly describe how you would assign work to each core to achieve this speedup.

3 cores are needed to take the maximum advantage of each cache hit, and a maximum speed up of almost 3 can be achieved.
Core 1 calculates iterations i+0, i+1, i+2, i+3, core 2 calculates iteration i+4, i+5, i+6, i+7, and core 3 calculates iteration i+8, i+9, i+10, i+11.

[1]An imaginary town featured in *One Hundred Years of Solitude* by the late Colombian author Gabriel García Márquez (1927-2014).