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

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

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)?

(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?

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

(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?

(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?

(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?

(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?

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

(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?

(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?

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?

(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).

(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?

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

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

(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)?

(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).

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.

(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.

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

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.

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

ii) application B?

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

ii) application B. Why or why not?

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.

(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		

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			

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

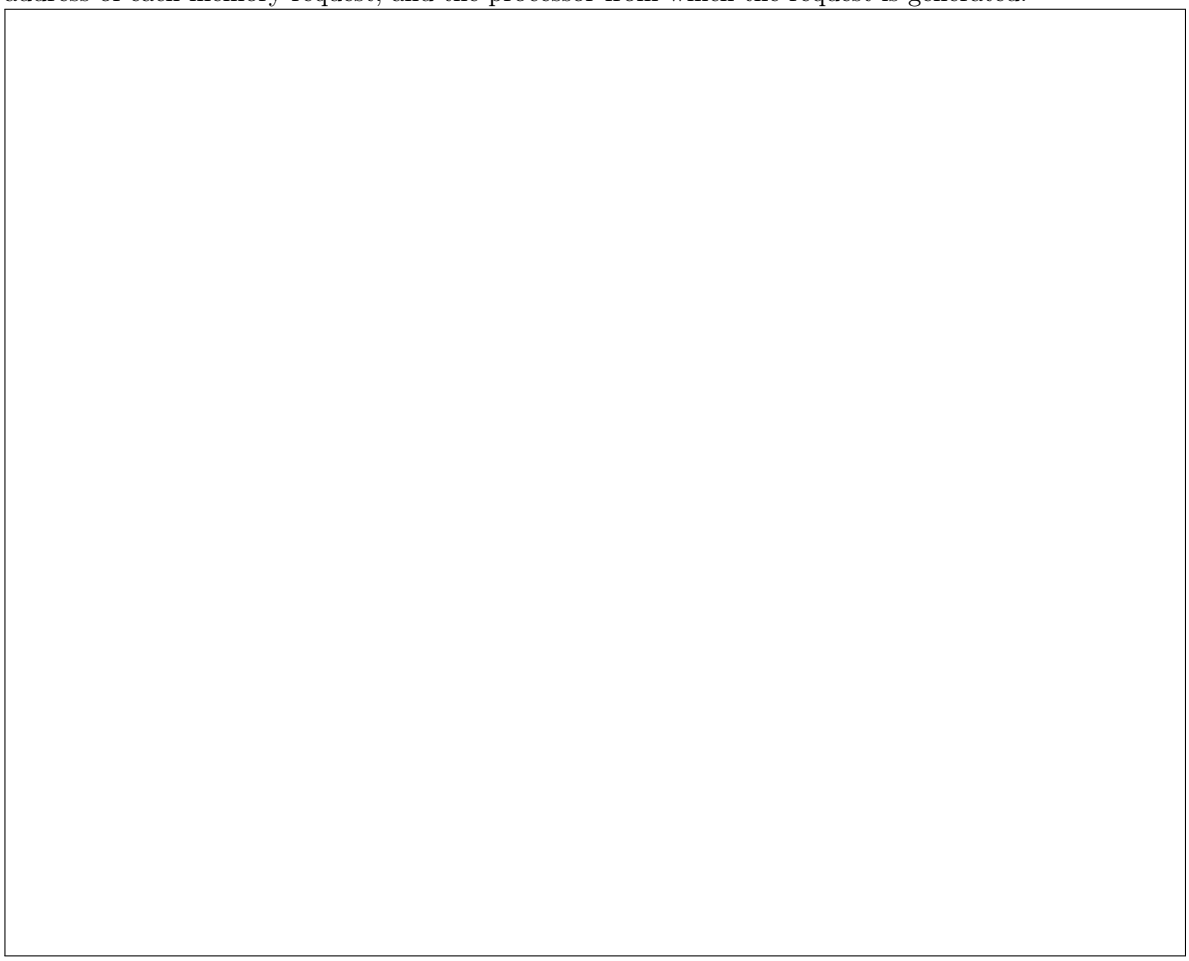
Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	E

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.



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.

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

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.

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.

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.

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

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

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.

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

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

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