ETH 263-2210-00L Computer Architecture, Fall 2021

HW 4: Emerging Technologies, Parallelism, Multiprocessors Bottleneck Acceleration, Cache Coherence, Memory Consistency (SOLUTIONS)

Instructor: Prof. Onur Mutlu
TAs: Juan Gómez Luna, Mohammed Alser, Jisung Park, Lois Orosa Nogueira, Gagandeep Singh, Haiyu Mao, Behzad Salami, Nour Almadhoun Alserr, Mohammad Sadr, Hasan Hassan, Can Firtina Geraldo Francisco De Oliveira Junior, Abdullah Giray Yaglikci, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Rakesh Nadig, João Dinis Ferreira, Haocong Luo, Roknoddin Azizibarzoki

Given: Monday, Nov 29, 2021
Due: **Monday, Dec 13, 2021**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture21/`. Please, check your inbox, you should have received an email with the password you should use to login. If you did not receive any email, 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 (2-11).** You should upload your answers to the Moodle Platform (`https://moodle-app2.let.ethz.ch/course/view.php?id=15536`) as a single PDF file.
- If you have any questions regarding this homework, please ask them the Moodle forum (`https://moodle-app2.let.ethz.ch/mod/moodleoverflow/view.php?id=675695`).
- Please note that the handin questions have a hard deadline. However, you can submit your paper reviews till the end of the semester.

1. Critical Paper Reviews [1,000 points]

You will do at least 6 readings for this homework, out of which 5 are tagged as **REQUIRED** papers (two are 2-page short papers). You may access them by *simply clicking on the QR codes below or scanning them.*

Required 1 Required 2 Required 3 Required 4 Required 5

Write an approximately one-page critical review for the readings (i.e., papers from #1 to #5 **and** *at least* 1 of the remaining 19 papers, from #6 to #24). If you review a paper other than the 5 mandatory papers, you will receive 250 BONUS points on top of 1,000 points you may get from paper reviews (i.e., each additional submission is worth 250 BONUS points with a possibility to get up to 6000 points).

Please read the guideline slides for reviewing papers and watch Prof. Mutlu's guideline video on how to do a critical paper review. We also provide you with sample reviews which you can access using the QR code. 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. **We will give out extra credit that is worth 0.5% of your total course grade for each good review.**

Guideline Slides Guideline Video Sample Reviews

1. **(REQUIRED, two pages)** 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
2. **(REQUIRED, two pages)** 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
3. **(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
4. **(REQUIRED)** A. G.Yaglikci, M.Patel, J.S.Kim, R.Azizi, A.Olgun, L.Orosa, H.Hassan, J.Park, K.Kanellopoulos, T.Shahroodi, S.Ghose, O.Mutlu, "*BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows*, in HPCA 2021. https://people.inf.ethz.ch/omutlu/pub/BlockHammer_preventing-DRAM-rowhammer-at-low-cost_hpca21.pdf
5. **(REQUIRED)** R.Bera, K.Kanellopoulos, A.Nori, T.Shahroodi, S.Subramoney, O.Mutlu, "*Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning*, in MICRO 2021. https://people.inf.ethz.ch/omutlu/pub/Pythia-customizable-hardware-prefetcher-using-reinforcement-learning_micro21.pdf
6. 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
7. L.Orosa, Y.Wang, M.Sadrosadati, J.S.Kim, M.Patel, I.Puddu, H.Luo, K.Razavi, J.G.Luna, H.Hassan, N.M.Ghiasi, S.Ghose, O.Mutlu, "*CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations*, in ISCA 2021. https://people.inf.ethz.ch/omutlu/pub/CODIC-DRAM-internal-timing-control-substrate_isca21.pdf
8. H.Hassan, Y.C.Tugrul, J.S.Kim, V.Veen, K.Razavi, O.Mutlu, "*Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications*, in MICRO 2021. https://people.inf.ethz.ch/omutlu/pub/U-TRR-uncovering-RowHammer-protection-mechanisms_micro21.pdf
9. L.Orosa, A.G.Yaglikci, H. Luo, A.Olgun, J.Park, H.Hassan, M.Patel, J.S.Kim, O.Mutlu, "*A Deeper Look into RowHammers Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses*, in MICRO 2021. https://people.inf.ethz.ch/omutlu/pub/ADeeperLookIntoRowhammer_micro21.pdf
10. M.Patel, G.F.Oliveira Jr., O.Mutlu, "*HARP: Practically and Effectively Identifying Uncorrectable Errors in Memory Chips That Use On-Die Error-Correcting Codes*, in MICRO 2021. https://people.inf.ethz.ch/omutlu/pub/HARP-memory-error-profiling_micro21.pdf
11. 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
12. 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
13. 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
14. 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
15. 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

16. 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
17. 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
18. 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
19. 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
20. 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
21. H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "*Row Buffer Locality Aware Caching Policies for Hybrid Memories"* in ICCD 2012. https://people.inf.ethz.ch/omutlu/pub/rowbuffer-aware-caching_iccd12.pdf
22. J. Ren, J. Zhao, S. Khan, J., Y. Wu, and O. Mutlu, "*ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems*, in MICRO 2015. https://people.inf.ethz.ch/omutlu/pub/ThyNVM-transparent-crash-consistency-for-persistent-memory_micro15.pdf
23. J.A.Joao, M.A.Suleman, O. Mutlu, and Y.N.Patt, "*Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs*, in ISCA 2013. https://people.inf.ethz.ch/omutlu/pub/utility-based-acceleration-acmp_isca13.pdf
24. M. A.Suleman, O.Mutlu, J.A.Joao, Khubaib, and Y.N.Patt, "*Data Marshaling for Multi-Core Architectures*, in ISCA 2010. https://people.inf.ethz.ch/omutlu/pub/dm_isca10.pdf

2. Emerging Memory Technologies [150 points]

Computer scientists at ETH developed a new memory technology, ETH-RAM, which is non-volatile. The access latency of ETH-RAM is close to that of DRAM while it provides higher density compared to the latest DRAM technologies. ETH-RAM has one shortcoming, however: it has limited endurance, i.e., a memory cell stops functioning after 10^6 writes are performed to the cell (known as cell wear-out).

A bright ETH student has built a computer system using 1 GB of ETH-RAM as main memory. ETH-RAM exploits a perfect wear-leveling mechanism, i.e., a mechanism that equally distributes the writes over all of the cells of the main memory.

(a) This student is worried about the lifetime of the computer system she has built. She executes a test program that runs special instructions to bypass the cache hierarchy and repeatedly writes data into different words until **all** the ETH-RAM cells are worn-out (stop functioning) and the system becomes useless. The student's measurements show that ETH-RAM stops functioning (i.e., all its cells are worn-out) in one year (365 days). Assume the following:

- The processor is in-order and there is no memory-level parallelism.
- It takes 5 ns to send a memory request from the processor to the memory controller and it takes 28 ns to send the request from the memory controller to ETH-RAM.
- ETH-RAM is word-addressable. Thus, each write request writes 4 bytes to memory.

What is the write latency of ETH-RAM? Show your work.

$t_{wear\_out} = \frac{2^{30}}{2^2} \times 10^6 \times (t_{write\_MLC} + 5 + 28)$
$365 \times 24 \times 3600 \times 10^9 \text{ns} = 2^{28} \times 10^6 \times (t_{write\_MLC} + 33)$
$t_{write\_MLC} = \frac{365 \times 24 \times 3600 \times 10^3}{2^{28}} - 33 = 84.5\text{ns}$

Explanation:

- Each memory cell should receive 10^6 writes.
- Since ETH-RAM is word addressable, the required amount of writes is equal to $\frac{2^{30}}{2^2} \times 10^6$ (there is no problem if 1 GB is assumed to be equal to 10^9 bytes).
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to $t_{write\_MLC} + 5 + 28$.

(b) ETH-RAM works in the multi-level cell (MLC) mode in which each memory cell stores 2 bits. The student decides to improve the lifetime of ETH-RAM cells by using the single-level cell (SLC) mode. When ETH-RAM is used in SLC mode, the lifetime of each cell improves by a factor of 10 and the write latency decreases by 70%. What is the lifetime of the system using the SLC mode, if we repeat the experiment in part (a), with everything else remaining the same in the system? Show your work.

$t_{wear\_out} = \frac{2^{29}}{2^2} \times 10^7 \times (25.35 + 5 + 28) \times 10^{-9}$
$t_{wear\_out} = 78579686.3\text{s} = 2.49 \text{ year}$

Explanation:

- Each memory cell should receive $10 \times 10^6 = 10^7$ writes.
- The memory capacity is reduced by 50% since we are using SLC: $Capacity = 2^{30}/2 = 2^{29}$
- The required amount of writes is equal to $\frac{2^{29}}{2^2} \times 10^7$.
- The SLC write latency is $0.3 \times t_{write\_MLC}$: $t_{write\_SLC} = 0.3 \times 84.5 = 25.35\,\text{ns}$

3. BossMem [150 points]

A researcher has developed a new type of nonvolatile memory, BossMem. He is considering BossMem as a replacement for DRAM. BossMem is 10x faster (all memory timings are 10x faster) than DRAM, but since BossMem is so fast, it has to frequently power-off to cool down. Overheating is only a function of time, not a function of activity. An idle stick of BossMem has to power-off just as frequently as an active stick. When powered-off, BossMem retains its data, but cannot service requests. Both DRAM and BossMem are banked and otherwise architecturally similar. To the researcher's dismay, he finds that a system with 1GB of DRAM performs considerably better than the same system with 1GB of BossMem.

(i) What can the researcher change or improve in the core (he can't change BossMem or anything beyond the memory controller) that will make his BossMem perform more favorably compared to DRAM, realizing that he will have to be fair and evaluate DRAM with his enhanced core as well? (15 words or less)

Solution: Prefetcher degree or other speculation techniques so that misses can be serviced before memory powered off.

(ii) A colleague proposes he builds a hybrid memory system, with both DRAM and BossMem. He decides to place data that exhibits low row buffer locality in DRAM and data that exhibits high row buffer locality in BossMem. Assume 50% of requests are row buffer hits. Is this a good or bad idea? Show your work.

Solution: No, it may be better idea to place data with high row buffer locality in DRAM and low row buffer locality data in BossMem since row buffer misses are less costly.

(iii) Now a colleague suggests trying to improve the last-level cache replacement policy in the system with the hybrid memory system. Like before, he wants to improve the performance of this system relative to one that uses just DRAM and he will have to be fair in his evaluation. Can he design a cache replacement policy that makes the hybrid memory system look more favorable? In 15 words or less, justify NO or describe a cache replacement policy that would improve the performance of the hybrid memory system more than it would DRAM.

Solution: Yes, this is possible. Cost-based replacement where cost to replace is dependent on data allocation between DRAM and BossMem.

(iv) In class we talked about another nonvolatile memory technology, phase-change memory (PCM). Which technology, PCM, BossMem, or DRAM requires the greatest attention to security? What is the vulnerability?

Solution: PCM is nonvolatile and has potential endurance attacks.

(v) Which is likely of least concern to a security researcher?

Solution: DRAM is likely least vulnerable, as BossMem also has nonvolatility concerns.

4. Emerging Memory Technologies [150 points]

4.1. Phase Change Memory [40 points]

Indicate whether each of the following statements is true or false. *Note: we will subtract 1.5 points for each **incorrect** answer. (The minimum score you can get for this question is 0 point.)*

(a) Phase Change Memory (PCM) is more technology-scalable than DRAM.

1. True 2. False

(b) PCM read/write operations are more energy efficient than DRAM read/write operations.

1. True 2. False

(c) PCM provides shorter access latency but has lower endurance compared to NAND flash memory.

1. True 2. False

(d) Row-buffer hit latencies of DRAM and PCM are comparable.

1. True 2. False

(e) Row-buffer miss penalty is smaller in PCM than in DRAM, since PCM commonly employs a small row buffer.

1. True 2. False

4.2. SAFARI-RAM [110 points]

Researchers in the SAFARI Research Group developed a new non-volatile memory technology, SAFARI-RAM. The read and write latency of SAFARI-RAM is close to that of DRAM while providing higher memory density compared to the latest DRAM technologies. However, SAFARI-RAM has one shortcoming: it has limited endurance, i.e., a memory cell fails after 10^7 writes are performed to the cell (known as *cell wear-out*).

A bright ETH student has built a computer system using SAFARI-RAM as the main memory. SAFARI-RAM exploits a perfect wear-leveling mechanism, i.e., a mechanism that equally distributes the writes across all of the cells of the main memory.

The student wants to estimate the worst-case lifetime of SAFARI-RAM when used as main memory. The student executes a test program to wear out the entire SAFARI-RAM *as quickly as possible*. The test program runs special instructions to bypass the cache hierarchy and repeatedly writes data into different pages until *all* the SAFARI-RAM cells are worn out. The students measurements show that SAFARI-RAM stops functioning (i.e., all its cells are worn-out) in 2.5 years. Assume the following:

- The processor is in-order.
- There is no memory-level parallelism (i.e., there is a single bank in the memory system).
- It takes 32 ns to send a memory request from the processor to the memory controller.
- It takes 52 ns to send the request from the memory controller to SAFARI-RAM.
- The write latency of SAFARI-RAM is 172 ns.
- Each write request is fully serialized, i.e., there are three steps of write requests: (1) memory request from CPU to controller, (2) write request from controller to SAFARI-RAM, and (3) data write to SAFARI-RAM cells. None of the steps can be pipelined.
- SAFARI-RAM requests are issued at *page* granularity, where the page size is 4,096 bytes (4 KiB).
- SAFARI-RAM adopts a quad-level cell (QLC) technique that stores four bits in a single memory cell.

(a) What is the capacity of SAFARI-RAM? Show your work. *Hint:* 2.5 years $\approx 8 \times 10^{16}$ ns.

128 GB

Explanation:

- Each memory cell should serve 10^7 writes.
- Since SAFARI-RAM performs writes at page granularity, the required number of writes is equal to $\frac{capacity}{2^{12}} \times 10^7$.
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to 32+52+172 = 256 ns.

$$\therefore \frac{capacity}{2^{12}} \times 10^7 \times 256 = 8 \times 10^{16}$$

$$capacity = \frac{2^{15} \times 10^{16}}{2^8 \times 10^7}$$

$$= 2^7 \times 10^9$$

(b) The student decides to improve the lifetime of SAFARI-RAM cells by using the single-level cell (SLC) mode in which a single memory cell stores a single bit. When SAFARI-RAM operates in the SLC mode, each cell's endurance increases by a factor of 10 while the write latency of SAFARI-RAM decreases to p% of the write latency of the QLC mode. To measure the lifetime of SAFARI-RAM in the SLC mode, the student repeats the same experiment performed in part (a) while keeping everything else in the system the same. The result shows that the lifetime of SAFARI-RAM increases by 2× in SLC mode compared to the QLC mode. Formulate p using C_{QLC}, which denotes the capacity of SAFARI-RAM in the QLC mode.

$p = \frac{1}{172} \times (\frac{2^{18} \times 10^{10}}{C_{\text{QLC}}} - 8400)$

Explanation:

- Each memory cell should receive $10 \times 10^7 = 10^8$ writes.
- The memory capacity is reduced by 4× in the SLC mode compared to the QLC mode. $\therefore C_{\text{SLC}} = \frac{C_{\text{QLC}}}{2^2}$.
- The required number of writes to wear out the entire SAFARI-RAM in the SLC mode is equal to $\frac{C_{\text{SLC}}}{2^{12}}$.
- The total latency of each memory access in the SLC mode is equal to $(32+52+172 \times \frac{p}{10^2})$.
- The lifetime in the SLC mode is 5 years = 16×10^{16} (double the lifetime in the QLC mode).

$$\therefore \frac{C_{\text{SLC}}}{2^{12}} \times 10^8 \times (84 + 172 \times \frac{p}{10^2}) = 16 \times 10^{16}$$

$$172 \times \frac{p}{10^2} = \frac{2^{18} \times 10^8}{C_{\text{QLC}}} - 84$$

$$p = \frac{1}{172} \times (\frac{2^{18} \times 10^{10}}{C_{\text{QLC}}} - 8400)$$

5. Asymmetric Multicore [150 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.

6. Cache Coherence [150 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 and b.2, 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 and b.2) 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. Cache Coherence (II) [150 points]

We have a system with 4 byte-addressable processors {P0, P1, P2, P3}. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. All caches are connected to and actively snoop a global bus, and cache coherence is maintained using the MESI protocol, as we discussed in class. Note that on a write to a cache block in the S state, the block will transition directly to the M state. Accessible memory addresses range from `0x00000` – `0xffffff`.

Each processor executes the following instructions in a *sequentially consistent* manner:

P0	
0	`st r0, 0x1ff40`
-	
-	

P1	
1	`st r0, 0x110c0`
2	`st r1, 0x11080`
3	`ld r2, 0x1ff00`

P2	
4	`ld r0, 0x1ff40`
5	`ld r1, 0x110f0`
-	

P3	
-	
-	
-	

After executing the above 6 memory instructions, the *final* tag store state of each cache is as follows:

Final Tag Store States

Cache for P0		
	Tag	*MESI state*
Set 0	`0x1ff`	`S`
Set 1	`0x1ff`	`S`
Set 2	`0x110`	`I`
Set 3	`0x110`	`I`

Cache for P1		
	Tag	*MESI state*
Set 0	`0x1ff`	`S`
Set 1	`0x1ff`	`I`
Set 2	`0x110`	`M`
Set 3	`0x110`	`M`

Cache for P2		
	Tag	*MESI state*
Set 0	`0x10f`	`I`
Set 1	`0x1ff`	`S`
Set 2	`0x10f`	`M`
Set 3	`0x110`	`I`

Cache for P3		
	Tag	*MESI state*
Set 0	`0x133`	`E`
Set 1	`0x000`	`I`
Set 2	`0x000`	`I`
Set 3	`0x10f`	`I`

(a) Fill in the following tables with the *initial* tag store states (i.e., *Tag* and *MESI* state) before having executed the six memory instructions shown above. Answer X if a tag value is unknown, and for the *MESI* states, write in *all possible values* (i.e., `M`, `E`, `S`, and/or `I`).

Initial Tag Store States

Cache for P0		
	Tag	*MESI state*
Set 0	`0x1ff`	`M, E, S`
Set 1	`X`	`M, E, S, I`
Set 2	`0x110`	`M, E, S, I`
Set 3	`0x110`	`M, E, S, I`

Cache for P1		
	Tag	*MESI state*
Set 0	`X`	`M, E, S, I`
Set 1	`0x1ff`	`M, E, S, I`
Set 2	`X`	`M, E, S, I`
Set 3	`X`	`M, E, S, I`

Cache for P2		
	Tag	*MESI state*
Set 0	`0x10f`	`I`
Set 1	`X`	`M, E, S, I`
Set 2	`0x10f`	`M`
Set 3	`X`	`M, E, S, I`

Cache for P3		
	Tag	*MESI state*
Set 0	`0x133`	`E`
Set 1	`0x000`	`I`
Set 2	`0x000`	`I`
Set 3	`0x10f`	`I`

(b) In what order did the memory operations enter the coherence bus?

time →					
0	4	5	1	2	3

8. Memory Consistency [150 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;`

9. Memory Consistency (II) [150 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	`X[0] = 1;`	Instr. T1.0	`X[0] = 0;`
Instr. T0.1	`X[0] += 1;`	Instr. T1.1	`flag[0] = 1;`
Instr. T0.2	`while(flag[0] == 0);`	Instr. T1.2	`b = X[0];`
Instr. T0.3	`a = X[0];`		
Instr. T0.4	`X[0] = a * 2;`		

`X` 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 could be possible final values of `a` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0, 1, or 2.

Explanation:
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[0]`. Thread 0 will remain in instruction T0.2 until `flag` is set by T1.1. There are *at least* three possible sequentially-consistent orderings that lead to *at most* three different values of `a` at the end:
Ordering 1: T1.0 → T0.0 → T0.1 → T0.3 - Final value: `a` = 2.
Ordering 2: T0.0 → T1.0 → T0.1 → T0.3 - Final value: `a` = 1.
Ordering 3: T0.0 → T0.1 → T1.0 → T0.3 - Final value: `a` = 0.

(b) What could be possible final values of `X[0]` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0, 2, or 4.

Explanation:
The value of `X[0]` is twice the value of `a`:
Ordering 1: T1.0 → T0.0 → T0.1 → T0.3 → T0.4 - Final value: `X[0]` = 4.
Ordering 2: T0.0 → T1.0 → T0.1 → T0.3 → T0.4 - Final value: `X[0]` = 2.
Ordering 3: T0.0 → T0.1 → T1.0 → T0.3 → T0.4 - Final value: `X[0]` = 0.

(c) What could be possible final values of `b` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

0, 1, 2, or 4.

Explanation:
Because there are no specific instructions to enforce the execution ordering of T1.2, `b` can have any of the values that `X[0]` can have during the execution of the two threads.

(d) The programmer wants `a` and `b` to have the same value at the end of the execution of both threads. The final value of `a` and `b` should be the same value as in the original program (i.e., the possible final values of `a` that you found in part (a)). What *minimal* changes should the programmer make to the program?
(Hint: You can use more flags if necessary.)

She needs two more flags to enforce ordering.

Explanation:
Since the final value should be the same as in the original program, we have to maintain the `flag[0]` in T1.1 and T0.2. Then, `b` should not be updated until `X[0]` has the value that will be stored in `a`. Thus, either before or after T0.3, we need to set a new flag (`flag[1]`) that will be checked by Thread 1 before updating `b`. Finally, we cannot update `X[0]` until `b` has its final value. Thread 1 will set `flag[2]` only after `b` is updated. The modified code will be as follows:

	Thread T0		**Thread T1**
Instr. T0.0	`X[0] = 1;`	Instr. T1.0	`X[0] = 0;`
Instr. T0.1	`X[0] += 1;`	Instr. T1.1	`flag[0] = 1;`
Instr. T0.2	`while(flag[0] == 0);`	Instr. T1.2	**`while(flag[1] == 0);`**
Instr. T0.3	`a = X[0];`	Instr. T1.3	`b = X[0];`
Instr. T0.4	**`flag[1] = 1;`**	Instr. T1.4	**`flag[2] = 1;`**
Instr. T0.5	**`while(flag[2] == 0);`**		
Instr. T0.6	`X[0] = a * 2;`		

10. Parallel Speedup [150 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.