

# 15-740/18-740 Computer Architecture

## Homework 2

Due Wednesday, September 21, at 12:00 PM

### 1. Amdahl's Law I

A particular program  $P$  running on a single-processor system takes time  $T$  to complete. Let us assume that 40% of the program's code is associated with "data management housekeeping" (according to Amdahl) and, therefore, can only execute sequentially on a single processor. Let us further assume that the rest of the program (60%) is "embarrassingly parallel" in that it can easily be divided into smaller tasks executing concurrently across multiple processors (without any interdependencies or communications among the tasks).

- (a) Calculate  $T_2, T_4, T_8$ , which are the times to execute program  $P$  on a two-, four-, eight-processor system, respectively.

Assume the time taken to execute on a single processor system is $T$ . $T_2 = 0.4T + (0.6/2)T = 0.7T$ $T_4 = 0.4T + (0.6/4)T = 0.55T$ $T_8 = 0.4T + (0.6/8)T = 0.475T$
---

- (b) Calculate  $T_\infty$  on a system with an infinite number of processors. Calculate the speedup of the program on this system, where *speedup* is defined as  $\frac{T}{T_\infty}$ . What does this correspond to?

$T_\infty = 0.4T + (0.6/\infty)T = 0.4T$ $Speedup = (T/T_\infty) = T/0.4T = 2.5$ This corresponds to the speedup if "only" the serial portion had been executed.
--

### 2. Amdahl's Law II

Amdahl compares and contrasts the performance of three different machines in his paper (Machines A, B, C). For this problem, consider Machines X, Y, Z configured as follows.

- **Machine X** A scalar processor with 1 arithmetic unit, running at frequency  $2f$ .
- **Machine Y** An array processor with 4 arithmetic units, running at frequency  $f$ .
- **Machine Z** A VLIW processor with 4 arithmetic units, running at frequency  $f$ .

Additionally, we define *instruction-level parallelism* as the degree in which an application's instructions are independent of each other and, therefore, can be executed concurrently.

- (a) If a large portion of an application's code has very low instruction-level parallelism, on which machine would it run the fastest, if any, and why?

An application with low instruction-level parallelism may run fastest on <b>Machine X</b> , as its instructions may not be aggressively executed in anything other than sequential order and the additional frequency provided by this machine would help complete single instructions more quickly.
--

- (b) Describe the characteristics of an application which would perform better on Machine Y than on Machine Z.

An application which must <b>operate on large vectors of data</b> may benefit from the additional throughput Machine Y would have to offer over Machine Z.
--

- (c) Describe the characteristics of an application which would perform better on Machine Z than on Machine Y.

An application which has **high instruction-level parallelism** with scalar instructions, in a way which coincides with the available arithmetic units on Machine Z, may perform better on Machine Z than on Machine Y.

### 3. Moore's Law

- (a) At each generation, silicon fabrication technology is commonly referred to by its feature size (or technology node), which can be loosely thought of as the smallest resolution at which transistors can be "drawn" on a silicon die (called "dimensional tolerances" by Moore). Cutting-edge silicon chips of today are fabricated using a feature size of 32nm. Two years ago, it was 45nm. Four years ago, it was 65nm. According to these three numbers, is Moore's Law still in effect? Explain why or why not.

Moore's Law states that the density of components on a chip doubles roughly every two years. Given the feature sizes in the problem, the area required to "draw" a transistor is  $32^2 = 1024 \text{ nm}^2$ ,  $45^2 = 2025 \text{ nm}^2$ , and  $65^2 = 4225 \text{ nm}^2$ . Thus, each technology node is roughly twice as dense as the one before it.

- (b) Consider two approaches of doubling the number of transistors: halving the size of a single transistor while maintaining constant die area (Moore's Law) versus maintaining the size of a single transistor while doubling the die area. List some reasons why the first approach is superior to the second approach.

**Possible reasons include:** maintaining a standardized die size for backward compatibility with older hardware, voltage and frequency scaling benefits which come with the use of smaller components (assuming leakage current is not an issue), savings in the cost of fabricating a certain number of chips due to smaller amount of silicon required, and increased yield.

### 4. Compiler Potpourri

- (a) Wulf motivates his principles of architecture design from a cost perspective. Yet in most implemented architectures, where cost is a key design constraint, it is possible to find many violations of Wulf's principles. Why do you think this is so? Pick three principles and list the trade-offs of abiding by each of them in terms of, for example, performance, power, reliability, and design and verification cost.

Chip designers may violate Wulf's principles because, though they may save cost in the long term for compiler writers and for backwards compatibility, when focusing on short term costs, it may make more sense to disobey the laws to allow for a simpler design.

Some examples of trade-offs include:

- **Regularity:** Consider a scenario where an ISA designer finds that many programs which run on their architecture routinely perform a shift right logical on the same data as a preceding add instruction and this prompts them to introduce a new, combined add and shift right logical instruction. If they were to obey Wulf's principles, they would need to implement new instructions which are shift right logical flavors of the basic arithmetic instructions. This would increase the complexity to decode the instructions which could add to the critical path, reducing performance; in addition, it may require more transistors to build the more complex decoder which will increase the cost of power; reliability could be affected because more transistor area would allow for more faults to occur due to transistor wear out; and design and verification cost would increase because the new, complex decoder's design would need to be vetted.
- **Orthogonality:** Consider an ISA designer who determines that two data values are often loaded from memory and then added together, and so introduces a new add instruction which takes two memory addresses as its operands and stores its result to a register. If this were to be done for every arithmetic instruction, it would suffer from some of the same drawbacks as the first principle.
- **Environment support:** Wulf offers some examples of ways to support software concepts in hardware, such as support for uninitialized variables. If an ISA designer supported such a feature, it would require the ISA to be exposed to the complexities of variables in the programming language. Lots of storage may be required to track metadata about variables when, in actuality, the programmer or compiler could have been diligent and would not need such support. This could decrease performance by requiring variable metadata to be stored with variables (effectively reducing the amount of information which could be stored in the caches, for example); power might increase due to the cost of transferring additional bits of data from one place to another (in terms of driving the wires or increasing the number of wires used); reliability would be a concern because more data would be susceptible to bit errors; and designing for every type of variable would be difficult to do and time consuming to test.

- (b) At the time, Wulf claimed that “designing irregular structures at the chip level is *very* expensive.” With an abundance of transistors available on-chip today, do you think this statement still applies? In what cases would it make sense and what cases would it not?

**Possible answer:** The statement still applies as the less regular a structure, the more difficult it would be to design and verify. However, if the benefit of such a structure can be justified, it would make sense to implement it in the available area on chip.

- (c) Colwell et al. discuss some of the merits and definitions of RISC versus CISC architectures. In your opinion, what the key strengths and weaknesses of RISC architectures compared to CISC architectures, as defined in the paper?

**Possible answers:**

- **Strengths:** simplifies the hardware, enables the compiler to optimize code by increasing its ability to reorder operations.
- **Weaknesses:** possibly worse performance (recall the original MIPS design had floating point operations emulated in software), increased code (program) size.

- (d) Colwell et al. also describe the concept of the “semantic gap”—a notion of how easily higher level language concepts can be expressed in the instruction set. Consider an ISA which tries to shorten the semantic gap with a `FOREACH x y` instruction which takes the location of a null-terminated array, `x` and the location of some code to execute on each element of the array, `y`. What are the trade-offs for the programmer,

compiler designer, and hardware designer, of implementing such an instruction? What do you think Wulf would have to say about such an instruction (specifically, which of his principles might it violate)?

**Possible answers:**

- **Programmer:** The programmer may not notice much of a difference in the way he or she constructs his or her code. The compiler would take care to translate a higher-level language to the ISA anyway.
- **Compiler designer:** The compiler designer, on the other hand, could choose to optimize their compiler to recognize portions of code which could benefit from using the new instruction. This would take time to implement and could reduce code size.
- **Hardware designer:** The hardware designer bears the burden of the implementation of such an instruction as he or she must modify the various components responsible for decoding and executing the instruction. This will take time to design and ensure the correctness of operation.

Such an instruction might violate the principles of **regularity** (because the instruction only operates on null-terminated arrays, not other data structures); **orthogonality** (because the instruction expects memory addresses as opposed to, say, registers containing memory addresses, or the contents of registers themselves); **composability** (as it obeys neither regularity nor orthogonality); **one vs. all** (because the use of the instruction is specific to null-terminated arrays); and **provide primitives, not solutions** (because the instruction could easily be decomposed to simpler operations).

## 5. SIMD and Multi-Threading

*List some examples of when a SIMD engine would be more beneficial than multi-threading. When would multi-threading be more beneficial to use than a SIMD engine?*

**Possible answers:** SIMD might be more beneficial than multi-threading when a program cannot be broken into separate threads (or separating it into multiple threads would incur high synchronization costs among the threads). If a program does not exhibit much data parallelism, but several copies of it can be run simultaneously (as in a web server), multi-threading might be more beneficial.