Quantifying Computational Power

1. Need to define computational power
   1. Need something that can analyze the efficiency of algorithms as well.
   2. Understand limits of RISC architectures, things to keep in mind with all architectures
      1. For example, measure power of subroutines within the CPU or within a computer program; if they are computationally powerful, means ability to handle very possible computable function
   3. Provides a link, even if rather tentative and abstract, between the answers to mathematical functions and the answers to computable functions
2. Turing Machine Weakness
   1. Turing machines can compute every “computable” problem, we accept this theorem as truth
   2. There are algorithms for which the turing machine takes an exponential amount of time to evaluate that DO NOT require exponential time on other systems
      1. Ex: addition time increases exponentially with the number of bits encoding the addend (in binary)
      2. Addition speed on other computers increases in proportion to the log2 of the number of bits in each addend
   3. Note that for every problem a turing machine must use brute force to solve, every other computer would also have to use brute force
3. Measurement of “computational efficiency”
   1. “Computational efficiency” = the overall speed and data usage to evaluate a certain computable function
      1. Computational efficiency is measured as a function of input variable length in bits
   2. “Entropy” = the log2 of the number of useful microstates the object in question can occupy
      1. In a lot of cases, this can be thought of as system state
      2. Can also be thought of as the number of RAM bits used
   3. Propose that efficiency is measured in two aspects: time and entropy, which are measured as a function of the entropy of input data
      1. In most cases, the entropy will correspond to the number of bits in the word size
      2. Time is delta clocks for synchronous computers
      3. Time for asynchronous computers can be measured against anything, for example relative to an external clock or relative to the time one switch takes to change state in the computer
      4. Entropy is used instead of bits because it is a more general term, and this paper seeks to explore classical computation in the broadest sense possible
      5. Also, entropy must consider system state in addition to RAM and ROM. This way, affecting the entropy of the system allows for program branching.
      6. Generally, RAM and ROM correspond to program execution entropy and program definition entropy respectively
      7. RAM and ROM are lumped together in entropy because of the need for including immediate values and the program itself and computer RAM, all of which can be mixed together
      8. Immediate values stored in ROM may be considered both execution entropy and definition entropy
      9. General “efficiency” may be considered the product of time efficiency and entropy efficiency
      10. Let G(M, F, n) be the general efficiency of a computer M executing function F with a variable word size (or input entropy) of n. Let T(M, F, n) be the time efficiency, E(M, F, n) be the entropy efficiency.  
          G(M, F, n) = T(M, F, n)\*E(M, F, n)
      11. Algorithms written to reduce one aspect of efficiency, ie making the execution more efficient in that aspect, generally serve to increase the values of the other two aspects
          1. Examples: addition, factoring
      12. Choosing the most “efficient” algorithm is the process of finding the minimum value of general efficiency, making this a 3-dimensional optimization problem
   4. The biggest problem with extremely simple computers such as turing machines or lambda calculus is that their general efficiencies (without biasing functions) are exponential with respect to n for functions computable with polynomial general efficiency on other systems
4. Families of Fundamental Functions
   1. Functions that do specific things in succession of each other or are nested within each other
      1. Yes, you can even define loops if the functions themselves are just in succession or nested.
      2. Use well behaved rules for which functions execute when if they are nested
   2. The general efficiency of the function family reflects the general efficiency of the computer as a whole
      1. More specifically, if the general efficiency of the function family is non-exponential, so too is the general efficiency of ANY computable function
      2. In other words, every computable function can be defined in terms of the fundamental functions operating in succession without having infinite or exponential repetition of the functions.
   3. A “powerful computer” may be defined as a computer where the general efficiency of the function family is non-exponential with respect to n
      1. Let C(M, F, n) be the general efficiency of any function and B(M, F, n) be the combined (summed) general efficiency of the function family. Let P(n) be a polynomial function of n.
      2. IF (limit as n approaches infinity) B(M, F, n)/P(n) = 0 THEN (limit as n approaches infinity) C(M, F, n)/P(n) = 0
      3. If the function family is non-exponential in general efficiency, then every computable function is non-exponential in general efficiency.
   4. Let C(M, F, n) be the general efficiency of any function and B(M, F, n) be the combined (summed) general efficiency of the function family. Let P(n) be a polynomial function of n.
      1. (limit as n approaches infinity) C(M, F, n)/B(M, F, n) < P(n) << 2^n
      2. This whole thing is concerned with asymptotic behavior
      3. The less than symbol between the limit and P(n) is because the limit might have an asymptote that involves transcendental aspects such as log2(n), but all of these aspects will grow more slowly than a polynomial as n tends to infinity
   5. Framing in terms of P vs NP:
      1. All problems that should be evaluated in a P way can be evaluated in a P way. The computer is still able to solve NP problems, but they will require atrocious amounts of energy. If a computer is able to compute the function family, it is guaranteed to behave in the specific way I just mentioned
      2. Because NP is exponential and equivalent to look up table, for the functions in P to be “complete:”
      3. \*actually, NP can be a lot worse than a look up table? Maybe? Because have to evaluate to the end -> but np is where you don’t know the answer but can find by brute force; look-up table requires you to know beforehand -> “look up table” is the wrong terminology here i think, not the NP thing
      4. Limit as x goes to infinity of C0(x)/ax = 0 for all a > 1 and lim as goes to infinity of C1(x)/ax = 0 for all a > 1. C1 is the efficiency function on any computer, and C0 is the efficiency on the computer you are trying to measure. Basically, the set of all functions that can be evaluated in a non exponential worst case efficiency on any classical computer can be evaluated in the same manner on the computer you are trying to test.
      5. Furthermore, if the function family can be completed with N efficiency on a system, then all classically computable problems can be computed with N efficiency on that system.
   6. Note that these functions can determine the efficiency of anything that performs computation, meaning they can also apply to things like programming languages, not just physical computers.
   7. It is important to note that, because this definition is made to apply to physical systems with finite entropy, the functions are considered on the basis of how they would behave as you increase the entropy capacity of the system.
      1. Example: If you keep increasing the size of RAM, but you begin to encounter a point where you cannot arbitrarily index without exponential efficiency, then the overall index efficiency of the system should be considered exponential
      2. Note which parts of the system are taken to infinity and which parts are not: Things are only taken to infinity on the base level. Consider a system in which there are 4 caches and 4 address registers for those caches. The size of the caches and address registers will be taken to infinity, and its behavior analyzed as a function of the word size. HOWEVER, the number of caches and registers will not be taken to infinity, because these are one level above the base level entropy properties. In other words, having 4 registers and 4 caches is intrinsic to the system architecture, but the size of each cache or register is not intrinsic.
      3. Examining the behavior of system efficiency as properties one level above base level entropy are taken to infinity is possible; it simply involves a reframing of the base entropy, and yields interesting results (ie. efficiency increases at a MUCH faster rate, leading to constant or linear big O behavior).
      4. Only concerned with taking base level entropy to infinity because this is all that is required to be able to run ANY computable program.
5. Three Basic Functions as a Representation of Computation
   1. These functions are a function family that satisfies the requirements detailed in the previous section.
   2. Arbitrary Index, Arbitrary Read/Write, Logic
      1. Arbitrary Index
         1. Access any range of data/entropy using data at a predefined location as a pointer
         2. Necessary for manipulation of any position in entropy.
         3. Necessary to expand computer functionality to its entire domain.
         4. You need the ability to infinitely defer a web.
         5. Note that this includes indexing by use of immediate values, because the entropy of the system includes ram, rom, and system state
      2. Arbitrary Read/Write (eg transfer)
         1. Transfer one range of entropy into another as accessed by arbitrary index
         2. This also includes program branching, because branching can be described as just transferring a range of entropy into the positional state of the system.
         3. Necessary to utilize entropy at all. Computers fundamentally use data to produce a result.
         4. Without being able to read data, the computer would not be able to produce multiple outputs. Every output state would be the same. However, we know that computers produce multiple outputs based on input because we accept this as an axiom of computation. Therefore, because it would be a logical inconsistency for computers to not read data, data reads are one of the fundamental functions.
         5. Very similar line of reasoning to the arbitrary read for arbitrary write.
         6. An axiom of computation is that computers do things. The only way they can do things is by changing something about the world, which is what the umbrella term “arbitrary write” is used to describe.
         7. Arbitrary Program Branch
            1. Included as a side effect of the data transfer and pointers; because entropy includes system state as well, and branching is editing the state of the position of the ROM read head, metaphorically speaking
            2. Power sets of program branching (similar to number sets; irrationals contain rationals contain integers)

No branching, eg linear counter

if/else if/else

For loops without breaks

For loops with first order breaks (eg what happens in c)

For loops with n order breaks

Recursion of functions via the stack

Goto’s, where entropy is stored in position instead of the stack in a non-FIFO fashion; this is similar to recursion, but doesn’t have to employ the use of the stack to keep track of where you are trying to branch

Self-editing code, which is the king of branching, because everything can be re-defined

* + - * 1. Must be able to transcend FIFO branching
    1. Logic
       1. Bitwise NAND 2 ranges to output another range using arbitrary index and arbitrary read/write
       2. This is necessary because it allows for the implementation of all logical functions.
       3. Every logic gate can be built from 2-input nands on ranges
       4. Can be used to simulate any computer system and any logical function in conjunction with the arbitrary indexing and arbitrary read/write
  1. I/O functionality is not included because this paper is concerned with the internal execution of a computer
     1. If a computer is able to arbitrarily read/write and arbitrarily index, then a certain portion of entropy may be set aside for use as the “input” and “output” if desired
  2. Note that peripherals are not considered when analyzing the efficiency of a computer. If the computer is to be used with peripherals, then they should be considered part of the system itself. If analyzing a computer with a cpu, gpu, keyboard, monitor, and drivers to interface between those, consider all 5 types of components to constitute the computer.
  3. Note that this paper regards “computers” as machines which operate on data to produce data; no regard for such things as “exact timing” and “sensors” is given.
  4. Each function performs its function SOLELY by performing binary logic operations on the entropy of the system. Example:
     1. Let the range [a,b] represent a range of system bits
     2. Consider a system where bits 0 thru 10 are the specific system state (eg the program address and the values of the registers, etc) and bits 11 thru 100 are the values of read/write memory
     3. An example of the arbitrary data read and arbitrary data write operation combined with the arbitrary index function would be to write the result of ([0,10] OR [30,40]) to [40,50]. In this example, the arbitrary read/write allows the values of ranges to be read and edited in their entirety, and the arbitrary index function allows access to those locations in the first place. The arbitrary logic function is used to generate the OR using a base logic gate only, in this case with the three fundamental functions it is NAND
  5. Propose the conjecture that EVERY computable function on a computer architecture can be written as a procedure of the three basic functions operating on that architecture, where each function can call other functions.

1. Demonstration of Power
   1. VM of any machine
      1. You can literally just draw the circuit diagram for any classical computer in existence using the three basic functions.
      2. By default, you get:
         1. C program simulation
         2. Simulation of different programming paradigms
      3. Note that virtual machines created on a system will suffer the same inefficiencies. If the base system has extremely inefficient indexing, the virtual machine will as well, even if it is simulating a system where indexing is extremely fast. The slowness is forced to either manifest in the speed of operating the virtual machine or the speed of SIMULATING the operation of the virtual machine.
      4. Let M be the stronger machine and m be the weaker machine simulating M. Let T(M, F, n) be the time efficiency of any function F on the computer architecture M as a function of number of bits n; let E(M, F, n) be the entropy counterpart. Let B be the 3 basic functions. Let k0 and k1 be positive constants, eg. they are members of (0,infinity).  
         T(m, M(F), n) <= k0\*E(m, M(F), n)\*T(m, B, n)\*T(M, F, n)  
         E(m, M(F), n) <= k1\*E(M, F, n)\*E(m, B, n)  
         Note the parallel between the time equation and the chain rule in calculus.
   2. Differential equation evaluation efficiency
      1. Even analog computers have the same sort of efficiency as described by the three basic functions
      2. Analog computers are used for solving differential equations faster, but, because time is still critical to the operation and it uses physical parts, they still have the same type of efficiency stuff
   3. Infinite multithreading
      1. Simply use self-vm and then just switch between virtual machines, simulating shared memory. This creates the illusion of multiple cores (and also the reality of multiple cores, just operating a lot slower)
      2. Note that this process, while expensive, only increases in efficiency cost proportionally to product of <number of threads / cores> and <efficiency of self virtual machine>
   4. Families of Deference
      1. Arrays and stuff
      2. As long as you are able to defer infinitely many times, or at least up to the limit of your memory, it is fine
      3. In a system with addr register and ram, you need 2 general registers to perform every possible deference. In the worse case, you simply allot temporary immediate locations for each pointer you are using. The 2 general registers allow you to buffer the data and the pointer from immediate registers.
      4. As long as data transfer from an immediate pointer to a single order deference AND data transfer from a single order deference to an immediate is possible, the computer can perform all possible data indexes. Every data transfer scheme boils down to one of the two above functions (or immediate pointer to immediate pointer, which is a subset of either function)
      5. If you have less than 2 registers, shit gets painful, and you have to perform transfers bitwise using the entropy of the system; actually, you can’t really do anything because the entirety of the address has to be known, so you can only do the bitwise transfer with 1 register. 0 registers, you are shit outta luck
   5. In terms of arithmetic
      1. Distributed ALUs
      2. Mask writes with rotations up and down is ONLY PROPORTIONALLY slower than a computer with an ALU. Maybe also talk about how you still have to evaluate undecidable problems or whatever for the max amount of time on both machines; there is no algorithm to make it faster. -> oh shit but you might be able to make faster algorithms -> ok but the “better” computer can be simulated by programming on the “worse” computer, just replacing the “add” instructions and things like that with simpler stuff
      3. As efficiency of binary operations increases, all ALUs will eventually have to perform sub-optimal methods of operation because the computer cannot alter its physical self
      4. As long as the logic allows for creation of all logic gates from all possible sets of data and can write anywhere, it will be non-exponential
      5. The only way things have to be exponential is if a truth table is required to describe the logic, which is not the case if logic can be constructed as a function of entropy ranges, or if the function must be evaluated from start to finish to know the answer, but this is true for every computer and will not change if architecture changes.
      6. Discuss arithmetic families
      7. “Or” logic on its own is enough to create all logic gates. -> this means masked “or” writes and using jump based on the value of “or”-ing the bits of a word together
      8. Increasing the complexity of the ALU or computer architecture in general only increases speed of defined functions; there will always be functions that must be explicitly defined down to a bitwise level.
         1. You will have to implement a VM to get around this, where the VM provides the virtual circuitry to evaluate the new function
2. Analysis of Some Various Systems
   1. No computer will be capable of performing every function with maximum efficiency (maybe this should be an fpga???????) because the complexity blows up and the complexity of the computer is constant.
   2. Path Indexing
      1. Abuse entropy of the system state in order to transfer data in a system with no registers
      2. Extremely inefficient; proportional speed to the number of bits
   3. Return Matrix
      1. Not inefficient in entropy since the return position would have to be stated anyways; you just collect them in a different position
      2. It has log2(n) time efficiency, where n is the number of function calls
      3. Allows for pseudo-self editing code in a machine where the next address cannot be assigned manually
   4. Positional State Machine
      1. Example of a machine with exponential efficiency
   5. Edge detector
      1. Used for succession
      2. log2(n) efficiency where n is number of bits
      3. The fastest edge detector boils down to the speed of log2(n) because it has to hone in on the edge, even in physical systems
   6. Function degrees
      1. Zeroth degree is a macro, first degree involves placing things on the stack and calling (can handle entropy efficiency up to recursion); second degree allows for full manipulation of entropy beyond just recursion.
3. Flakiness Constant
   1. Just the product of <# of board connections>\*<# instructions required to complete program>\*<probability of failure at a physical joint during the execution of one instruction>
   2. Can be used to describe physical systems, or stuff that you are programming and trying to bug check. Basically, just establishes direct relation involving points of failure, clocks, and probability of failure at a single point
4. Depth of programming languages
   1. Depth of programming languages and a rigorous definition; eg windows operating system is the integral of python which is the integral of c which is the integral of assembly which is the integral of machine code which is the integral of cpu circuitry design which is the integral of electrical engineering which is the integral of applied mathematics which is the integral of number theory which is the integral of logical axioms that we assume to be true.
5. Two pcb layers is always enough.
6. DISTRIBUTED ALU
7. LOOK AT COMPUTER ARCHITECTURE AS A THIRD EFFICIENCY MEASURE?
8. APPLY THEORIES TO FILE SYSTEMS, OSes, PROGRAMS
9. LIMIT AS ARITHMETIC COMPLEXITY GOES TO INFINITY
10. GO AHEAD COMPUTING
11. System seizures?
12. Decentralized logic gates
13. Talk about how the computer still has to wait to access ROM and RAM, so maybe it should be considered in the general efficiency?
14. Microinstructions

Outline for stuff:

1) The efficiency at which a computer executes a function can be regarded as a function of the input word size. In general, the analysis is mostly concerned with the asymptotic behaviour of the function.

2) General efficiency is measured as the product of [efficiency of memory and system state usage (entropy efficiency)] as well as [the efficiency in time (time efficiency)]. Increasing the efficiency of one aspect will generally decrease the efficiency of the other aspect and vice-versa. Lower efficiencies correspond with a more powerful system.

3) Consider a family of functions that may either be nested or executed in succession and which describe operations on the entirety of a computer system, for example system state, addressing, etc. Note that this implies the ability to affect changes on the function script itself, as the script is considered part of the computer system. If a function can be executed in polynomial general efficiency on a sufficiently powerful computer architecture, then it can be constructed in polynomial general efficiency using the function family and rules for combining the functions as described. I provide an in-depth explanation of this in the paper as well as establishing what I consider the most basic family of functions (the "three basic functions") satisfying this power. I prove this using a virtual machine argument among other explanations.

4) Various mathematical relationships that describe relationships between different measures of efficiency on different computers. I also provide a method to calculate bounds on efficiencies of different architectures using techniques not dissimilar to the chain rule of derivatives.

5) Limit equations to describe the fact that, if a system can execute the "three basic functions" in polynomial general efficiency, then, with polynomial general efficiency, it can execute every function that can have polynomial general efficiency. Similarly, if the "three basic functions" take exponential general efficiency on the computer in question, then there will be certain functions that take exponential general efficiency on that computer but would only take polynomial general efficiency on a more powerful computer. This second point is rather obvious because the definition of the "three basic functions" proves it.

6) Analysis of general efficiency as different factors besides input word size are manipulated. For example, the general efficiency as the number of registers in the computer is increased and approaches infinity.

7) A way to describe the branching power of a computer architecture or programming language. I established a set of tiers as part of this analysis.

8) Define a degree system to describe arithmetic and deference functions based on the computational power required.

9) Analysis of various extremely minimalistic system architectures in terms of their general efficiency.

10) Analysis of algorithms which buffer data through the state of the system instead of simply placing the values in registers.

11) Decentralized ALUs and the general efficiency discrepancies between different ALUs. I analyze the asymptotic behaviour of ALU families as the complexity of the arithmetic operations approaches infinity.

12) Analysis from the point of view of the function, where the computer architecture itself is considered another manipulated variable and is considered in the general efficiency equation.

13) Definition of the "flakiness equations," describing a way to determine the flakiness of a real life system with less-than-ideal conditions.

14) Define a system to analyze the "degree" of functions based on their interaction with the stack in RAM (or something analogous to RAM).

15) Ways to apply these theories to programs and operating systems instead of physical computers (most theories translate with minimal technical effort).

16) Various unconventional computer architectures, such as go-ahead, asynchronous architectures without a clock.

17) Different, unconventional types of logic gates. Also discuss the way in which gates can be decentralized and manifest themselves as the interaction between different computer sections.

18) Multi-threading on single core computers. I analyzed this in terms of general efficiency.

19) Different ways to schedule micro-instructions and ways to balance work between software and hardware.