Elementos cuantitativos de diseño de computadoras

José Galaviz Casas

Facultad de Ciencias, UNAM

Índice general

1. Introducción **2**
1.1. La computadora como ente programable 2
1.2. Binario 4
1.3. Transistores 6
1.4. Lenguajes y niveles 11
1.5. Conjunto de instrucciones, arquitectura y organización 12
1.6. Reloj 13
1.7. La ecuación de desempeño de CPU 14
1.8. Tecnología y diseño 16

2. Midiendo el desempeño **23**
2.1. Pruebas de desempeño 32
2.2. Ley de Amdahl 35
2.3. Costo–beneficio 37
2.4. Malas métricas 37

Bibliografía **40**

1. Introducción

1.1. La computadora como ente programable

¿Para qué sirve una computadora personal? Bueno, depende, si tiene gabinete de escritorio, horizontal, puede usarse para colocar algunos adornos sobre él, en el frente se pueden pegar notas autoadheribles con las cosas urgentes por hacer; si el gabinete es de torre, en cambio, puede usarse como soporte para libros o como medio para dividir la superficie de la mesa, delimitando espacios de trabajo diferentes, si se coloca debajo del escritorio puede eventualmente ser usada para levantar los pies. En efecto, los usos de la computadora personal son muchos, eso sin considerar las *lap top*, cada día más usuales, cuyas características las convierten en una excelente mesa portátil.

Ciertamente los descritos serían los únicos usos de la computadora personal de contar sólo con los componentes físicos de ella, el *hardware*. Pero claro, todos sabemos que una computadora personal sirve para miles de cosas: conectarse a Internet, consultar correo electrónico y redes sociales, navegar, jugar, escuchar música y ver películas, procesar texto u hojas de cálculo, hacer compras en línea, en fin. Hoy día no hay prácticamente ninguna actividad humana en el mundo civilizado en la no estén involucradas las computadoras.

La versatilidad de nuestras computadoras actuales proviene del hecho de que son entes programables, saben hacer muchas cosas diferentes porque podemos indicarles cómo deben hacerse esas cosas. Una computadora es útil porque ejecuta *software*, programas diferentes para diferentes actividades. Muchas de nuestras comodidades actuales se deben a la sinergia entre el hardware y el software. Seguramente el lector está familiarizado con la programación de computadoras usando lo que se denomina *lenguaje de alto nivel*, como suele hacerse en la actualidad en todo el mundo.

Programar una computadora no siempre ha sido así de sencillo. La primera computadora electrónica de propósito general en la historia fue la

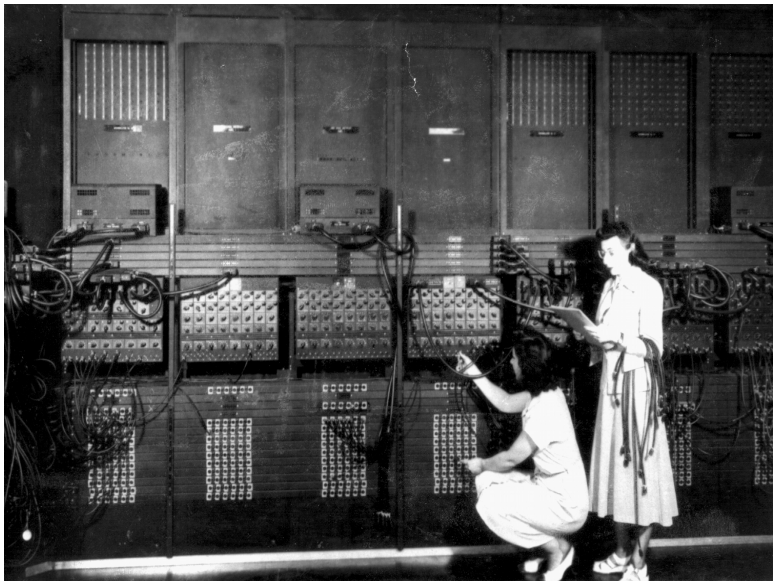

Figura 1.1: Dos programadoras de ENIAC haciendo su labor.

famosa ENIAC (*Electronic Numerical Integrator And Computer*) y programarla era una ardua labor. Realmente había que reconfigurar el hardware, desconectando cables y reconectándolos en otros lugares (Fig. 1.1).

Conscientes de que esta labor era lenta, delicada y propensa el error, Presper Eckert, John Mauchly, Herman Goldstine y John von Neumann discutieron, acerca de una mejor manera de proveer a una computadora del programa a ejecutar. Concluyeron que el programa debía poder almacenarse en la *memoria* de la máquina, para de allí ser leído e interpretado por la entidad encargada de la ejecución, llamada *unidad de control*. Las instrucciones serían entonces ejecutadas, con datos tomados de la misma memoria, por la *unidad aritmético-lógica* (ALU por sus siglas en inglés) de la máquina. Tanto los datos como el programa debían proveerse a la computadora desde el exterior y los resultados, finalmente debían ser también enviados al exterior, así que se requerían también de dispositivos de *entrada/salida*. La idea era usar esta alternativa en la siguiente computadora, la sucesora de ENIAC, a la que se denominó EDVAC (*Electronic Discrete Variable Automatic Computer*). Así, Von Neumann escribió estos conceptos en un reporte que vio la luz en junio de 1945, llamado *First Draft of a Report on the EDVAC*. En el reporte se describe, esencialmente, lo que hoy llamamos *Arquitectura de Von Neumann*, o bien *arquitectura de programa guardado*. En la figura 1.2

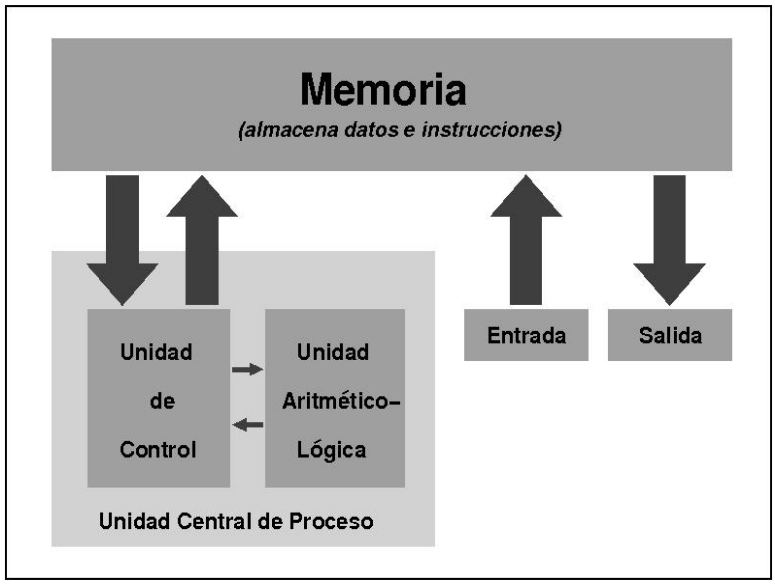

Figura 1.2: Esquema general de la arquitectura de Von Neumann.

se muestra esquemáticamente el concepto.

A la *unidad central de proceso* le llamaremos también *procesador central* y en la literatura se le suele llamar también CPU por las siglas en inglés de *Central Processing Unit*. Invisibles en la figura 1.2 existen otros elementos necesarios. Por ejemplo, si los datos están en la memoria y deben ser operados en el procesador, se requiere tener en éste, lugar para almacenarlos temporalmente. A estas unidades de almacenamiento dentro del procesador se les denomina *registros*.

1.2. Binario

A lo largo del tiempo la labor de programación se ha echo cada vez más simple gracias la construcción de toda una infraestructura para ello. En realidad nuestras computadoras no entienden lo que les decimos en C, en Java o en Scheme, las instrucciones que nuestras unidades centrales de proceso pueden interpretar y ejecutar son mucho más simples que las de un lenguaje de alto nivel y además deben ser dadas exclusivamente en el formato que la computadora puede almacenar y decodificar: en binario.

El lector probablemente haya escuchado antes que "las computadoras trabajan en binario". En efecto, los circuitos electrónicos con los que nues-

tras computadoras se construyen sólo son capaces de distinguir entre dos niveles de voltaje o de carga según el caso, a los que podríamos decirles A y B, Juan y Martha, o.. 0 y 1. Esta última es, por mucho, la mejor opción, porque 0 y 1 son los dígitos posibles en un sistema numérico posicional base 2 o binario[1] así que podemos abstraer lo que realmente se almacena y fluye por los circuitos de nuestras computadoras y representar esto mediante un número y por tanto hacer operaciones aritméticas. También podríamos decirles *verdadero* y *falso* y usarlos como los valores de verdad que utilizábamos en lógica proposicional, esto también nos es útil, porque entonces podemos construir funciones lógicas como la conjunción (Y), la disyunción (O), la implicación (Si.. entonces), y en general todas las que se expresan en tablas de verdad. A estas les llamaremos *funciones de conmutación*, las revisaremos más adelante y esencialmente es lo que, junto con las operaciones aritméticas, se lleva a cabo en la unidad aritmético-lógica ya mencionada. En síntesis, una computadora puede hacer todo lo que se requiere de ella trabajando en binario, con dos valores bastan.

Sin embargo no siempre las computadoras han trabajado en binario, ENIAC era decimal. Pero desde 1962 ya no se han hecho más (aparentemente la última fue la UNIVAC III, hecha por la compañía que originalmente formaron Eckert y Mauchly, los diseñadores de ENIAC). Para explicar esto es conveniente un ejemplo: imagine el lector dos dianas de tiro al blanco, ambas circulares y del mismo tamaño; supóngase que una de ellas, a la que llamaremos A, está dividida en 10 bandas concéntricas y la otra (B) en sólo dos. Si ahora se pone a una persona con un dardo, se le pide que diga en voz alta a cuál banda pretende darle y se le indica que dispare, ¿con cuál de las dianas tiene mayor probabilidad de atinar? Bueno, ciertamente en la diana de sólo dos bandas, esta tolera un mayor rango de error, la precisión del tirador puede ser menor que en el caso de la de 10. De igual forma nuestros dispositivos electrónicos toleran un mayor rango de variación en voltajes, intensidades o cargas sí sólo se consideran dos valores, son más confiables si trabajan en binario.

Supongamos que necesitamos calcular una función de tres variables independientes, digamos x, y y z, las tres con valores en el conjunto de los dígitos decimales $\{0, \ldots, 9\}$. ¿Cuantas posibles combinaciones de valores tenemos para las tres variables? Esencialmente habría que considerar todos los números decimales de tres cifras, desde el 000, hasta el 999, es decir mil valores. En general, trabajando con n dígitos decimales como variables, se tienen 10^n combinaciones. En binario en cambio, si se tienen n variables, el

[1] En inglés diríamos que son los *BInary digiTS*, de donde proviene la palabra *bits*.

número de combinaciones posibles es 2^n, bastante menos, de hecho 5^n veces menos. Como la complejidad de un circuito electrónico digital para hacer el cálculo está, en general, en función directa al número de combinaciones que pueden tener los valores de las variables independientes, resulta que la complejidad de los circuitos digitales es menor si estos trabajan en binario.

En síntesis, nuestras computadoras electrónicas digitales de hoy día trabajan en binario porque:

- Todo lo que queremos hacer se puede hacer en binario.
- La operación de los componentes electrónicos es más confiable.
- Es mucho más simple diseñar los circuitos.

Podemos pensar entonces que en la memoria de nuestras computadoras se almacenan datos e instrucciones en binario, y esas cadenas de ceros y unos se transfieren al procesador central para ser interpretadas como instrucciones o para ser considerados como los operandos con los que se deben hacer las operaciones aritméticas y lógicas. En nuestras computadoras actuales los datos e instrucciones se almacenan, como hemos dicho, dentro del procesador en lo que se denominan registros. Cuando se dice que un procesador es de 32 o de 64 bits esta nomenclatura se refiere al tamaño de almacenamiento que tienen los registros en los que se guardan los operandos de su ALU.

1.3. Transistores

Toda función de nuestro interés, es decir, toda función computable, puede representarse usando sólo ceros y unos para la entrada, para la salida y para especificar las operaciones elementales que la constituyen. Así que nos basta con el conjunto $\{0, 1\}$ como alfabeto para representarlo todo. Hemos dicho además que esto resulta conveniente porque hace nuestros dispositivos más confiables, más tolerantes: los obliga a distinguir sólo entre dos valores posibles y no entre diez. Fue Claude Shannon quien, en su tesis de maestría[2] de 1937, demostró que usando álgebra booleana y aritmética binaria, era posible hacer circuitos de relevadores[3] que calcularan funciones.

[2]Shannon, Claude Elwood, *A symbolic analysis of relay and switching circuits*, Thesis (M.S.), Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1940. También: Shannon, C. E., "A symbolic analysis of relay and switching circuits", *Transactions of the American Institute of Electrical Engineers*, No. 57, 1938, pp. 713-723.

[3]Un relevador es un interruptor operado eléctricamente. Es decir un dispositivo de conmutación electromecánico. Se les utilizaba en los conmutadores de la red telefónica y fueron utilizados en las primeras computadoras electromecánicas, como la Mark I de Harvard (1944).

A	**B**	**A+B**	**AB**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Tabla 1.1: Tabla de verdad de la disyunción y la conjunción.

Lo esencial para construir una computadora basándose en los conceptos de Shannon, es poseer un dispositivo capaz de conducir o interrumpir el paso de la corriente eléctrica: un interruptor. Usando interruptores es posible calcular cualquier operación de las que conocemos de la lógica proposicional. En la figura 1.3 se muestran, por ejemplo dos disposiciones diferentes para conectar un par de interruptores: en paralelo o en serie. Digamos que un interruptor puede estar cerrado (dejando fluir la corriente eléctrica), en cuyo caso diremos que está en posición 1, o abierto (impidiendo el paso de corriente) lo que denotaremos como 0. Diremos que un foco puede estar encendido (1) o apagado (0). Así que en la siguiente tabla están todos los posibles estados de los interruptores A y B de la figura 1.3 y el correspondiente estado del foco.

Por supuesto las funciones calculadas por nuestros circuitos son la *disyunción* y la *conjunción* de la lógica proposicional, que el lector recordará de cursos previos. Como éstas, todas las funciones de la lógica proposicional pueden calcularse usando circuitos de interruptores. Lo mismo ocurre con las funciones aritméticas, cualquiera de ellas, si representamos los números en base 2, puede verse como una tabla similar a la que hemos puesto y puede calcularse usando interruptores. Por cierto, basta poder calcular la disyunción, la conjunción y una función más, una que convierta un 0 en un 1 y viceversa. A esto le llamamos *negación*. Estas tres operaciones forman lo que llamaremos un *conjunto completo* de operaciones, con ellas nos basta para poder expresar cualquier otra función que pueda llevarse a cabo con interruptores.

Por supuesto no queremos que los interruptores tengan que ser operados de forma manual, sino automática. Se necesita poder construir una red de

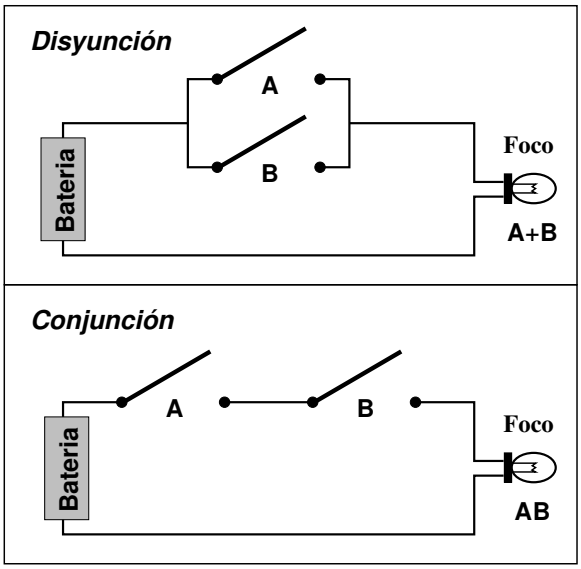

Figura 1.3: Cálculo de la disyunción y la conjunción del cálculo proposicional usando interruptores.

esos interruptores, interconectándolos para que unos determinen el comportamiento de otros. Al principio los relevadores electromecánicos fueron la única alternativa, dada la tecnología disponible. Pero después fueron reemplazados por dispositivos puramente electrónicos. Los primeros de este tipo fueron las válvulas de vacío, conocidos coloquialmente como "bulbos". La desventaja era su velocidad de operación y el hecho de que debían calentarse para funcionar, lo que significa un gran consumo de energía que se disipa en forma de calor.

Las cosas comenzaron a cambiar cuando, a fines de los 40, Bardeen, Brattain y Shockley, basándose en los experimentos de Edgar Lilienfeld de 1923, inventaron el transistor en los laboratorios Bell. Este dispositivo tiene dos propiedades interesantes para la electrónica: puede servir como amplificador de corriente o como interruptor. Por supuesto en el ámbito de la computación digital es ésta segunda la que es fundamental.

En la naturaleza los materiales pueden dividirse, de acuerdo con su conductividad, en dos grupos fundamentales: los conductores y los aislantes. Los primeros dejan que la corriente eléctrica fluya por ellos, mientras que los segundos no lo permiten; un conductor tiene cargas eléctricas móviles, un aislante no. Esencialmente esto depende de cuantos electrones libres posean los átomos del material. Los conductores tienen muchos, los aislantes

muy pocos. Pero existen también materiales que, en estado puro, son aislantes, pero cuando contienen cierto tipo y concentración de impurezas de otro pueden, en cierta circunstancia, conducir la corriente eléctrica. A estos materiales se les denomina *semiconductores*.

El silicio, por sí solo es un aislante: su resistividad al paso de la corriente eléctrica es alta. Sin embargo si se le agregan átomos de algún otro elemento puede convertirse en conductor. Si se le agrega, por ejemplo, arsénico o fósforo se convierte en un material donante de electrones libres, con carga negativa, por lo que se le llama tipo N; si se le agrega, en cambio, galio o boro, se convierte en un material aceptador, lleno de huecos, lo que le confiere una carga positiva, por lo que se le llama tipo P. Un transistor es un sándwich hecho de una capa de material P flanqueada por dos capas de material N, a lo que se le llama *unión NPN* o bien, a la inversa, una capa de material N flanqueada por capas P, a lo que se le llama *unión PNP*.

En una unión NPN, los electrones libres de una capa N no podrían fluir hacia el otro extremo en la otra capa N, pasando a través de la capa P, porque se los tragarían los huecos de ésta. La capa P actúa como esponja. Si la capa P es suficientemente delgada en contraste con las capas N y se crea un voltaje, entre la capa P y una de las capas N, lo que significa que hay un "empuje" de cargas desde la N a la P, entonces los huecos se llenan del todo y la carga puede fluir de un extremo a otro del transistor. Es como si la carga eléctrica fuera agua, la regiones N son entonces secciones de un tubo llenas de agua y la región P un tapón de esponja en medio de ellas; el voltaje es equivalente a la acción de una bomba en uno de los extremos, si se bombea con suficiente potencia el agua satura la esponja P y el agua termina fluyendo a través del tubo. La única diferencia entre un transistor de unión NPN y uno PNP es la polaridad.

Si un transistor recibe corriente eléctrica por el emisor y no recibe nada por la base entonces es como un interruptor abierto, por el colector no sale nada. Si, en cambio, recibe suficiente corriente por la base, entonces la carga del emisor es "bombeada" a través de la base y del colector. En la figura 1.4 se muestra el corte transversal de un transistor plano, como los que se utilizan en los circuitos integrados actuales.

Actualmente el grosor del canal P es de unos 5 o 6 átomos, lo que ocasiona que, aún cuando el transistor esté en estado de corte, algunas cargas se fugan del emisor al colector. A este fenómeno se le llama *fuga de transistor* o *transistor leaking* en inglés y es importante porque, aún cuando la magnitud de la fuga no hace que se confunda un cero con un uno, sí representa un desperdicio de energía que debe disiparse en forma de calor, lo que es muy importante hoy día cuando deseamos dispositivos móviles cuya batería dure

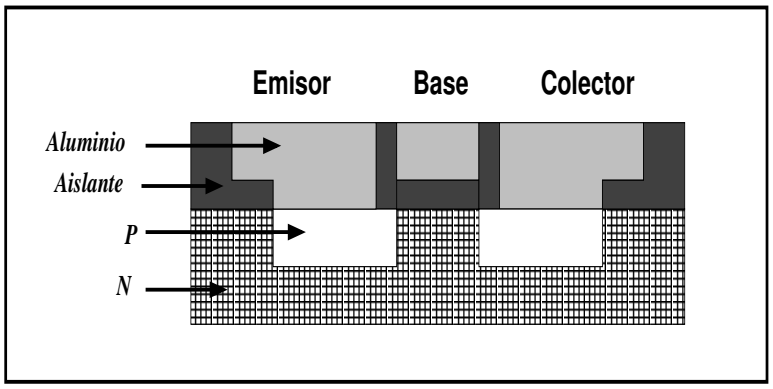

Figura 1.4: Corte transversal de un transistor plano.

mucho tiempo. Sin embargo, recientes avances tecnológicos harán posible fabricar en 2012 chips con tecnología de 22nm y transistores, ya no planos como el mostrado en la figura 1.4, sino tridimensionales, con la base en un plano perpendicular al emisor y al colector. Esto reducirá notablemente la fuga de energía y con ello la duración de la carga de nuestras baterías y reducirá la cantidad de calor producido por los chips. Esta nueva tecnología es llamada Tri-Gate por Intel y FinFET por el resto de la industria.

El siguiente avance luego del uso de transistores no fue, de hecho, un nuevo dispositivo electrónico, sino el tamaño del mismo. Reducir el tamaño de los transistores hizo posible empacar muchos de ellos interconectados para lograr circuitos con alguna función específica. Al empacamiento de circuitos se le llama *integración* y al resultado, por supuesto, *circuito integrado*. Así que el siguiente paso tecnológico en la construcción de computadoras digitales fue el uso de circuitos integrados.

En la literatura suele emplearse el término *generación* para referirse a los estadios del proceso evolutivo de los dispositivos de cómputo. En general se clasifican como sigue:

Primera generación. Basadas en válvulas de vacío.

Segunda generación. Basadas en circuitos de transistores individuales, es decir, sin empacar.

Tercera generación. Basadas en circuitos integrados a baja escala, unos cientos de transistores por paquete.

Cuarta generación. Basadas en circuitos integrados a gran escala (VLSI

o *Very Large Scale Integration*), miles, millones y, hoy día, cientos de miles de millones de transistores por paquete.

1.4. Lenguajes y niveles

Cuando hacemos un programa en lenguaje C, éste tiene que recorrer un camino para convertirse en algo que la unidad central de proceso pueda ejecutar. El programa debe ser traducido a instrucciones de *lenguaje de máquina*. Estas, por lo antes dicho, pueden pensarse como secuencias de ceros y unos. Una instrucción podría ser algo como:

`10100101010001001010011101O`

A lo mejor los primeros cinco bits (`10100` = 20) indican que hay que hacer la operación número 20 del catálogo de operaciones de la ALU, por ejemplo una suma. Los siguientes seis (`101010` = 42) indican que uno de los operandos está almacenado en el registro 42 y que es allí donde debe guardarse el resultado de la suma. Los restantes 16 bits (`0010010100111010` = 9530) podrían representar el otro operando, el número 9530. Así que la operación podría escribirse cómo:

ADD R20, 9530

Donde se ha usado la palabra `ADD` para decir "suma".

Podríamos, de hecho, si tuviéramos la paciencia suficiente, traducir un programa en lenguaje de máquina almacenado en memoria usando palabras y códigos como los anteriores, a los que se les llama *mnemónicos*. Expresar un programa así es lo que se denomina programar en *lenguaje ensamblador*. Esta era un práctica usual hasta los años 70 y siguió siéndolo en el contexto de las computadoras personales hasta la del 80. Al programa que debe traducir de lenguaje ensamblador a lenguaje de máquina se le denomina, precisamente, *ensamblador*. La labor de traducción de un ensamblador es simple, las instrucciones en lenguaje ensamblador corresponden, una a una, con las de lenguaje de máquina.

Para traducir de un lenguaje como C a lenguaje de máquina las tareas son muchas y bastante más complejas. Se requiere generar un equivalente semántico del programa en C en lenguaje de máquina. Cada instrucción en C corresponde a varias instrucciones de máquina, es por eso que se les denomina *lenguajes de alto nivel*, el término se refiere al nivel semántico de las instrucciones.

Los programas encargados de hacer traducciones de lenguaje de alto nivel a lenguaje de máquina se dividen en dos categorías: los que generan un programa completo semánticamente equivalente al escrito en lenguaje de alto nivel, llamados *compiladores* y los que traducen una por una las

instrucciones de lenguaje de alto nivel, conforme leen traducen y alimentan con eso al procesador, a estos se les denomina *intérpretes*. Los compiladores además de traducir, también optimizan el código varias veces conforme el proceso de traducción se acerca más a lenguaje de maquina.

1.5. Conjunto de instrucciones, arquitectura y organización

En vista de lo mencionado en la sección previa, es claro que resulta fundamental definir con claridad cuál es el catálogo de operaciones que la computadora sabe hacer *per se*, el *conjunto de instrucciones* de la máquina. Todo lo que se puede hacer con una computadora se construye sobre esa base. Cualquier compilador o intérprete y cualquier sistema operativo depende en última instancia de ese conjunto de operaciones elementales del procesador.

El diseño del conjunto de instrucciones es lo que se denomina *arquitectura del conjunto de instrucciones* o ISA, por las siglas en inglés de *Instruction Set Architecture*. El diseño del este catálogo es, de hecho, el modelo abstracto del que parte el programador de compiladores (y por tanto implícitamente, el programador en general) para hacer realmente útil a la computadora de acuerdo con lo que se ha dicho al inicio del texto. Originalmente el término *arquitectura de computadoras* se refería justamente a las características del conjunto de instrucciones, el área se dedicaba a diseñar este conjunto con el objetivo de hacerlo suficientemente versátil para el programador y a la vez, suficientemente sencillo de implementar en hardware para hacerlo eficiente. Aparentemente el origen del término "arquitectura de computadora" se lo debemos a IBM, cuando en la década de los 60 lo usó para enfatizar el hecho de que los programas en lenguaje de máquina que se podían ejecutar en una computadora de la familia System/360, podían transportarse a cualquier otra de la misma familia, aún cuando su tamaño, aspecto y capacidades fueran diferentes, y ejecutarse sin problema. El código dependía exclusivamente de la arquitectura del conjunto de instrucciones: la arquitectura S/360.

Cuando hablamos de una implementación de un conjunto de instrucciones entramos al terreno de lo que se denomina *organización de computadoras* o *microarquitectura*. En contraste con la arquitectura del conjunto de instrucciones, cuyo objetivo es determinar *qué* puede hacer la máquina, en la organización el objetivo es determinar *cómo* lo hace, qué partes de hardware se necesitan y cómo deben interactuar para que cada una de las instrucciones del lenguaje de la computadora puedan ejecutarse. El ámbito de la arquitectura es abstracto y el de la organización es concreto, pero la sinergia de ambos es lo que logra computadoras eficientes y versátiles.

El más clara la diferencia original entre arquitectura y organización de

computadoras cuando pensamos, por ejemplo, en un procesador de Intel con el conjunto de instrucciones tradicional llamado X86 y uno de AMD con el mismo conjunto. Ambos procesadores tienen la misma arquitectura de conjunto de instrucciones, pero las ejecutan de manera diferente, tienen una organización distinta.

Hoy en día la diferencia entre arquitectura y organización de computadoras ya no es tan evidente ni hay un acuerdo general en cómo delimitarlas. En buena medida porque actualmente la abstracción necesaria para construir el software sobre el hardware, está formada no sólo por el conjunto de instrucciones, sino por algunos otros elementos: los mecanismos de acceso a memoria y las garantías de consistencia que esta ofrece, el esquema de manejo de entrada y salida y el de condiciones de excepción, por ejemplo. Algunos autores prefieren colocar a la organización de computadoras y a la arquitectura del conjunto de instrucciones como parte de la arquitectura de computadoras.

1.6. Reloj

Nuestros dispositivos electrónicos funcionan de tal forma que, para tener un estado claro, bien definido, se debe esperar un cierto tiempo suministrando corriente para alcanzarlo. Así que una vez que se ha hecho lo necesario para representar un 1 en una celda de un registro, se debe esperar un tiempo para garantizar que, en efecto, allí se almacena un 1. Esto significa que los procesos de transferencia no son instantáneos, nuestras computadoras no pueden operar de manera, formalmente hablando, continua. Operan en pasos discretos de tiempo. Por tanto se requiere que haya una entidad encargada de marcar el tiempo, de indicar cuando se puede operar y cuando hay que esperar, algo como el metrónomo que se utiliza en la enseñanza musical. A esto se le denomina *reloj*[4].

Todas nuestras computadoras actuales operan a un ritmo establecido por un reloj, cuanto más rápido oscile mayor será el número de operaciones aritmético-lógicas o de transferencia de registros que se pueden hacer por unidad de tiempo. La frecuencia del reloj está, cómo en todo lo que oscila, dada en términos de oscilaciones por segundo (Hertz) aunque nuestras computadoras operan a velocidades tan sorprendentes que tenemos que usar múltiplos de ella como los gigahertz (GHz, miles de millones de oscilaciones por segundo). En la tabla 1.2 se listan algunos de los prefijos usuales.

[4]No se debe confundir el término con el de un reloj que indique la hora del día. El reloj al que nos referimos aquí es simplemente un marcador del ritmo, conceptualmente un *tic-tac*.

Prefijo	**Valor**	**Prefijo**	**Valor**
zepto	10^{21}	kilo	10^3 o 2^{10}
atto	10^{-18}	mega	10^6 o 2^{20}
femto	10^{-15}	giga	10^9 o 2^{30}
pico	10^{-12}	tera	10^{12} o 2^{40}
nano	10^{-9}	peta	10^{15} o 2^{50}
micro	10^{-6}	exa	10^{18} o 2^{60}
mili	10^{-3}	zeta	10^{21} o 2^{70}

Tabla 1.2: Prefijos comunes y su valor.

A las oscilaciones del reloj que marca el ritmo de trabajo en la computadora se les llama *tics* o *clocks*.

1.7. La ecuación de desempeño de CPU

Considerando la frecuencia de operación de una CPU es posible calcular el tiempo efectivo de operación del procesador en la ejecución de un programa. Con T denotaremos el tiempo total de ejecución de un programa, con C el número de ciclos necesarios para ejecutarlo y con D la duración de cada ciclo. Así:

$$T = C \cdot D \tag{1.1}$$

Por ejemplo un programa que tarda en ejecutarse 7 millones de ciclos en una computadora cuyo ciclo dura 2 ns. tarda: $T = 7{\times}10^6 \cdot 2{\times}10^{-9} = 0.014\ s$.

Otra manera de medir el tiempo de CPU es, evidentemente:

$$T = \frac{C}{F} \tag{1.2}$$

donde $F = \frac{1}{D}$ es la frecuencia de operación del reloj, los gigahertz.

Un programa que tarda 17 millones de ciclos en una máquina a 2 GHz, tarda $T = (17 \times 10^6)/(2 \times 10^9) = 0.0085\ s$ de tiempo de CPU.

Pero es difícil saber cuantos ciclos en total toma la ejecución de un programa, así que podemos contar mejor el número de instrucciones (I). Si sabemos el tiempo promedio de ciclos por cada instrucción[5] (R).

$$R = \frac{C}{I} \tag{1.3}$$

Esta es una medida de que tan meritorio es un procesador, que tan baratas, en ciclos de reloj, son sus instrucciones.

De 1.3:

$$C = R \cdot I$$

en 1.1:

$$T = R \cdot I \cdot D \tag{1.4}$$

o bien en 1.2:

$$T = \frac{R \cdot I}{F} \tag{1.5}$$

Hagamos un análisis de unidades de la expresión 1.4.

$$\frac{ciclos}{instrucción} \times \frac{instrucciones}{programa} \times \frac{segundos}{ciclo} = \frac{segundos}{programa}$$

En síntesis el desempeño depende de:

1. Frecuencia del reloj.
2. Cantidad de ciclos por cada instrucción.
3. Contador de instrucciones por programa.

y depende en la misma medida de cada uno de los tres elementos, así que una mejora del 10 % en alguno de los rubros mencionados genera una mejora del 10 % en el desempeño total.

¿De qué dependen estos factores?

Frecuencia. Hardware, tecnología y organización del procesador (calor generado, densidad de componentes).

[5] El término usado en inglés es *clocks per instruction* y frecuentemente se denota con CPI.

Ciclos por instrucción. Organización, arquitectura del conjunto de instrucciones.

Instrucciones por programa. Arquitectura del conjunto de instrucciones y eficiencia del compilador.

Algunas veces es útil contar el número de ciclos de reloj como sigue:

$$C = \sum_{i=1}^{n} R_i \cdot I_i$$

i es el índice de la instrucción en el catálogo de instrucciones de la máquina, I_i es el número de veces que la instrucción i se ejecuta en el programa y R_i es el número de ciclos de reloj que tarda en ejecutarse la instrucción i. De donde:

$$T = \left(\sum_{i=1}^{n} R_i \cdot I_i \right) \cdot D \tag{1.6}$$

Esto porque, de acuerdo a la definición de valor esperado de una variable aleatoria (en este caso en número de clocks por instrucción): $E(X) = \sum_{espacio\ muestral} xp(x)$. Como nuestro valor de R es, de hecho, un promedio:

$$R = \sum_{i=1}^{n} R_i \times \frac{I_i}{I}$$

Si sustituimos esto en 1.4 nos da 1.6.

1.8. Tecnología y diseño

En 1975, la desaparecida Digital Equipment Corporation[6], introdujo al mercado su modelo PDP-11/70. Ésta costaba $72,650 dólares, venía con 256KB de memoria RAM (expandibles a 2MB), consumía 6000 Watts, podía hacer unas 300,000 operaciones por segundo y ocupaba al menos, si no se tenían muchos periféricos, el mismo volumen que un refrigerador de 18 pies cúbicos. Hoy día por unos $1000 dólares uno puede adquirir una computadora laptop con 4GB de RAM, capaz de hacer más de una decena de millar de millón de operaciones por segundo, que consume 95 Watts, que pesa menos de dos kilogramos y cabe en un portafolio.

[6]En 1998 DEC pasó a ser parte de *Compaq*, que a su vez fue comprada por Hewlett-Packard en 2002.

Figura 1.5: DEC PDP-11/70 de 1975 y una laptop de la segunda década del siglo XXI, unas 30,000 veces mas poderosa y cuesta 2 centésimas partes de lo que costaba la primera.

Muchos factores han contribuido a salvar la enorme distancia entre estos dos equipos de cómputo. Primero el avance tecnológico, el hecho de que los transistores con los que se fabrican nuestras computadoras se han podido hacer cada vez más pequeños y de que se hayan podido poner millones de ellos en un sólo circuito integrado (chip). Luego el viraje del mercado, el hecho de que hoy día existen millones de usuarios de computadoras en todo el mundo y cada día se añaden más. También han contribuido, y de manera fundamental, las innovaciones en el diseño de las unidades centrales de proceso (a las que llamaremos también procesadores) siempre en la búsqueda de un mayor desempeño y de obtener la mejor relación costo-beneficio posible.

Cuando en 1971 Intel desarrollo el 4004, el primer microprocesador de la historia, se pudieron poner sus 2300 transistores en una superficie de $12mm^2$. La tecnología de la época permitía hacer una celda de memoria de 10 micras[7]. Un Intel Xeon de 2011 de la serie 7500 tiene 2300 millones de transistores en ocho núcleos distribuidos en una superficie de $684mm^2$, la tecnología usada para fabricarlo permite construir celdas de memoria de

[7]También suele decirse micrómetros, la millonésima parte de un metro o la milésima de un milímetro

45nm (nanómetros, o sea 45 millonésimas de milímetro). Al momento de escribir estas notas (2017) el estándar es de 10nm probablemente en 2021 veremos unos de 4nm. Además este crecimiento en la densidad del chip no ha traído consigo un incremento considerable en el costo del chip.

Aparentemente el primero en notar y describir esta tendencia en 1965 fue Gordon E. Moore, co-fundador de Intel, por lo que se le ha llamado *la ley de Moore*. En la figura 1.6 se muestra una gráfica del logaritmo del número de transistores por chip contra el tiempo en los principales procesadores de Intel. El hecho de que los datos observados sean bien aproximados por una línea recta, dada la escala logarítmica en el eje de las abscisas, significa que el crecimiento del número de transistores es exponencial. En efecto Moore estimo que, si se fijaba un costo constante, el número de transistores que es posible poner en un chip a ese costo se duplicaba cada dos años, algunos más precisos suelen decir que cada 18 meses. El efecto es el mismo, mucha mayor densidad de transistores significa también mucho mayor poder de cómputo; aunque no precisamente en la misma proporción. Cuando se dobla el número de transistores se obtiene, en promedio, un 40 % más de poder de cómputo, que de cualquier forma es bastante.

Otros componentes de cómputo siguen aproximadamente leyes similares: la capacidad de los discos duros, el número de pixeles en las cámaras digitales o la capacidad de las memorias flash, por ejemplo.

La ley de Moore no es gratis, la inversión tecnológica necesaria para producir chips a densidades cada vez más altas también crece, sin embargo se ha mantenido. En buena medida por el principal factor que amortiza el costo de producción del equipo de cómputo: el consumidor.

Existen "leyes" similares a la de Moore que son, hoy día, probablemente más importantes. El llamado *escalamiento de Dennard* por ejemplo, expresado por Robert H. Dennard en 1974, establece que el consumo de potencia de un transistor (MOSFET en el artículo original) es proporcional al área. Esto es similar a la ley de Moore pero reemplazando el costo por el área. Es decir el número de transistores que se pueden alimentar con una potencia dada crece exponencialmente. Lo que a su vez está relacionado con la más reciente *ley de Koomey* (Jonathan Koomey, 2010) que establece que el desempeño por watt consumido crece exponencialmente duplicandose cada 1.57 años.

El usuario típico de un sistema de cómputo en la década de los 50 del siglo XX era un tipo de bata blanca con anteojos y un libro de física bajo el brazo, hoy día el usuario típico es cualquier persona. Las cosas comenzaron a cambiar, primero lentamente, cuando las computadoras se incorporaron al sector público y al privado y ya no sólo le pertenecían a la academia o

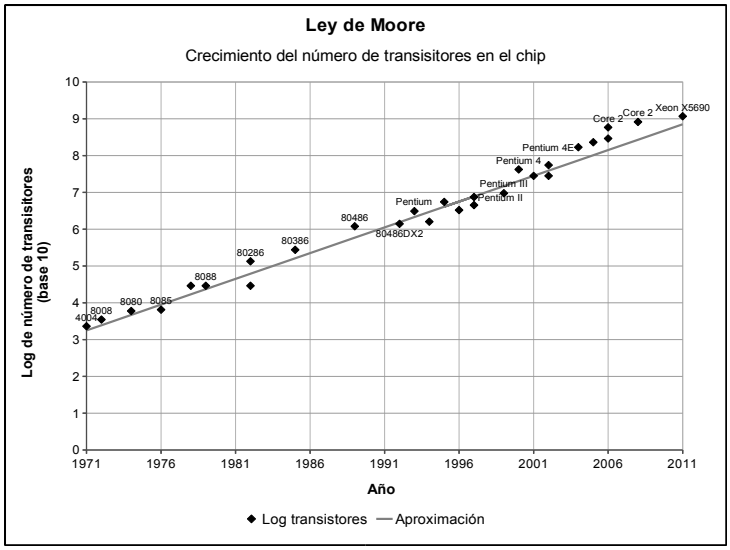

Figura 1.6: Evolución del número de transistores en el chip de un procesador.

la milicia. Pero el cambio radical ocurrió en los 80, la popularización de la computadora personal trajo consigo un enorme segmento de mercado. Por supuesto un segmento impensable si las aplicaciones de las computadoras permanecieran estáticas, la mayoría de las personas no suelen hacer cálculos de mecánica cuántica en la sala de su casa. Pero hoy casi todo mundo juega, oye música, se comunica o trabaja en su computadora personal usando programas de aplicación muy variados. Un Ferrari es caro porque el costo de producción tiene que amortizarse entre los 20 o 30 clientes que encargan uno al año, un modelo compacto de línea de las principales armadoras es mucho más barato porque el costo se amortiza con los miles de unidades que se venden. Lo mismo ocurre con el equipo de cómputo: más usuarios, muchos más, significa, mucho menor precio al consumidor final.

Pero ese no fue el único cambio importante durante la década de los 80.

Desde el surgimiento de los primeros sistemas de cómputo y hasta la década de los 70 la memoria principal de los sistemas de cómputo era de núcleo magnético, un tipo de tecnología costoso y de acceso muy lento. Como los programas requieren estar en memoria para poder ser ejecutados, era necesario hacer programas breves; cuanto más cortos, mejor, para hacer el menor uso de memoria posible. Sin embargo el desarrollo de compiladores no había alcanzado los niveles de optimización que podemos apreciar hoy día,

así que los programas buenos debían ser hechos en lenguaje ensamblador por programadores expertos. Para hacer un buen programa se requería conocer a la perfección la arquitectura de la máquina y su conjunto de instrucciones de lenguaje de máquina. Un programador así de especializado cobra mucho, así que los programas buenos eran caros. En respuesta a esto los diseñadores de hardware debían facilitarles la vida a los programadores en la medida de lo posible. Así que optaron por darle soporte en hardware a casi todas las necesidades imaginables de los programadores en ensamblador: si éste requería una instrucción para obtener la raíz cuadrada real de un número, se proporcionaba una instrucción de lenguaje de máquina que lo hacía y, claro está, se diseñaba el hardware necesario para ejecutarla en el procesador. Con el tiempo esto generó enormes conjuntos de instrucciones, el catálogo de lo que el procesador podía hacer en lenguaje de máquina era enorme y además contenía instrucciones muy complejas en su ejecución, extraer una raíz cuadrada no es trivial, al procesador le tomaba decenas de ciclos de ejecución.

Haciendo referencia a la ecuación 1.4, se pretende mejorar el desempeño disminuyendo el número de instrucciones necesarias para construir el programa. Dadas las circunstancias tecnológicas vigentes en los 70 y antes, sólo eso se podía hacer. Cuanto menor la longitud de programa mejor, tanto para el desempeño, como para el costo del software.

A lo largo de la década de los 70 las cosas ya eran diferentes, las memorias de semiconductores comenzaron a usarse más, eran más rápidas y más baratas, empezaron a surgir estándares de facto en la industria, como Unix y el lenguaje de programación C. Esto les concedió libertad a los diseñadores de hardware, el soporte para el programador de ensamblador ya no era fundamental, el hardware podía ser diferente si a fin de cuentas la cara que habría que presentar era la misma: Unix y un compilador de C. Con este nuevo panorama algunos arquitectos de computadoras se cuestionaron seriamente la premisa de que un programa bueno es un programa corto. Algo que además era mas asequible gracias a las optimizaciones que los compiladores ya hacían en el código que generaban. A lo mejor el programa podía ser más largo siempre y cuando el hardware que lo ejecuta sea más rápido. Si ya no es necesario dar al programador en ensamblador todo lo imaginable, ¿qué es lo que se debe dar?, ¿qué instrucciones debe tener el lenguaje de máquina?, ¿cuáles son las esenciales?

David Patterson en Berkeley y John Henessy en Stanford, cada uno por su cuenta comenzaron a hacer diseños nuevos, con conjuntos de instrucciones muy pequeños, mínimos, con instrucciones de tamaño fijo, formato fijo y con unas cuantas maneras diferentes de acceder a los datos, demasiado restrictivo

para lo que el mundo estaba acostumbrado, pero funcionó y muy bien. La decisión acerca de qué instrucciones y cuales modos de acceder a los datos debían incluirse en el conjunto se tomó con base en extensos y detallados estudios estadísticos acerca de la frecuencia promedio con que eran usados en los programas. La premisa, que después retomaremos con detalle era: hacer mejor lo más frecuente. El resultado fue lo que se denominó RISC (*Reduced Instruction Set Computer*). Por supuesto luego de elegir este nombre, a los diseños con el paradigma previo se les llamó CISC (*Complex Instruction Set Computer*). En la ecuación de desempeño 1.4, el factor en el que se enfoca el diseño CISC es el conteo de instrucciones por programa. En el diseño RISC, en cambio, el factor a mejorar es el número de ciclos por instrucción.

Durante los 80 los diseños RISC se adueñaron del mercado de estaciones de trabajo, computadoras más robustas que una PC, pero menos que un mainframe o una minicomputadora (como la PDP con la que empezamos el capítulo). Luego superaron con creces el desempeño de estas últimas y de hecho las desplazaron ocupando su nicho. Las máquinas más poderosas de los 80 y buena parte de los 90 (sin mencionar supercomputadoras), fueron diseños RISC. Hoy día la mayoría de los procesadores actuales para equipo de cómputo poseen aún muchos de los conceptos RISC subyaciendo en una arquitectura mucho más compleja. Ciertamente la complejidad resurgió, pero de manera mucho más acotada.

El paradigma RISC hizo crecer el desempeño, aproximadamente un tercio más de lo que se hubiera obtenido sólo gracias a la tecnología de integración de circuitos, añadiendo hardware. La simplificación de las instrucciones y su número mínimo, su formato fijo, hizo posible, como veremos más adelante, procesar varias instrucciones en simultáneamente, lo que se denomina *paralelismo a nivel de instrucción* o ILP por las siglas en inglés de *Instruction Level Parallelism*.

La ley de Moore se ha mantenido hasta ahora y no se prevé que vaya a dejar de funcionar en un futuro cercano, sin embargo hacer crecer la densidad de nuestros chips trae consigo problemas serios. Miles de millones de transistores, todos trabajando al unísono, todos hacinados en una muy pequeña área. El calor producido por todo esto es tal que hoy en día no es posible ver el chip del procesador de una computadora, está enterrado en enormes placas de aluminio de formas caprichosas diseñadas para disipar calor y bajo varios ventiladores que ayudan en la tarea. Alrededor del año 2006 el *escalamiento de Dennard* dejó de ser válido, antes de ello los fabricantes podían aprovechar el descenso en el consumo que traía consigo la reducción de tamaño y subir la frecuencia de operación del procesador. A partir de 2006, subir la frecuencia eleva el consumo (y con él la temperatura)

drásticamente, se llegó a un cierto límite de tamaño en el que existen fugas de energía importantes.

Por otra parte es difícil lograr mayor paralelismo del que se posee actualmente, si el nivel semántico al que se procesa es el de las instrucciones. El procesador a fin de cuentas sólo puede ver, a lo más, unas decenas de instrucciones. Dentro de ese limitado ámbito reacomoda, distribuye, retrasa o adelanta la ejecución de las instrucciones para lograr un máximo nivel de paralelización, para que, en la medida de lo posible, todas las partes del procesador estén siempre trabajando. Es por esto que la tendencia durante las últimas décadas ha sido asignar cada vez mayor responsabilidad al compilador en materia de desempeño. El compilador posee una visión mucho más general y puede, por tanto hacer optimizaciones que al procesador le serian imposibles, donde el procesador tendría que adivinar, por ejemplo, el compilador puede ocasionalmente tener la certeza de lo que ocurrirá.

Pero desde los primeros años del siglo XXI se ha estado trabajando en un frente diferente para lograr aun mayor desempeño, paralelizando, ya no sólo a nivel de instrucción, sino a un nivel semántico superior: ejecutando tareas completas en paralelo; a lo que se ha llamado *paralelismo a nivel de hilo de ejecución* (TLP, por las siglas en inglés de *Thread Level Parallelism*). Esto, como veremos más adelante implica varios problemas.

2. Midiendo el desempeño

Toda decisión que tome un arquitecto de computadoras tiene impacto en dos rubros: el costo y el desempeño del equipo. Así que, como en el resto de los ámbitos que componen nuestra experiencia en el mundo, la mejor decisión es aquella con la mejor relación costo-beneficio. Un primer requisito para tomar una buena decisión es, entonces, poseer una clara idea de cómo las diferentes alternativas disponibles tienen impacto en el desempeño y esto significa *cuantificarlo*.

Existen dos tipos diferentes de medidas de desempeño:

- Aquellas directamente relacionadas con el tiempo que tarda un sistema en terminar una tarea determinada. Esto es, el *tiempo de respuesta* del sistema y, claro está, un desempeño es tanto mejor cuanto menor sea el tiempo de respuesta, así que es una de las medidas que suelen clasificarse como *mejor cuanto menor* o LIB por la siglas de *Lower Is Better*.

- Las relacionadas directamente con el número de labores realizadas por unidad de tiempo. Es decir el *rendimiento* (*throughput*) del sistema. En este caso un desempeño mejor se asocia con un número mayor de tareas realizadas, así que es una medida del tipo *mejor cuanto mayor* o HIB por las siglas de *Higher is Better*.

En general los usuarios de un sistema de cómputo están más interesados en conocer el desempeño del sistema en términos del primer grupo de medidas y los administradores o proveedores de servicios prefieren conocer una medida del segundo tipo. Por supuesto en general si se tiene una medida del primer tipo es posible transformarla en una del segundo usando su inverso multiplicativo y viceversa.

Necesariamente para medir el desempeño de un sistema de cómputo se requiere de un instrumento de medida y dado que nuestras computadoras lo son porque ejecutan programas, se requiere entonces de uno o varios programas que pongan a trabajar al sistema y medir los tiempos de ejecución.

A la evaluación del desempeño de un sistema se le denomina *benchmark* en inglés, en español se suele traducir como *prueba de desempeño*, *análisis comparativo* o simplemente *comparativa*, lo que es bastante adecuado dado que, en efecto, medir es comparar. Por supuesto nos interesa poder comparar el desempeño de un sistema con el de otro. Algo como "El rendimiento del sistema A es 2.5 veces mayor que el del sistema B", formalmente:

$$\frac{\mathrm{R}end(A)}{\mathrm{R}end(B)} = 2.5$$

o equivalentemente, si el tiempo de respuesta es representado por T:

$$\frac{T(B)}{T(A)} = 2.5$$

Comparar el desempeño de dos sistemas es una labor delicada. Observemos, por ejemplo, los datos de tiempo de ejecución de dos sistemas mostrados en la tabla 2.1. Se puede decir, sin faltar a la verdad, que el sistema B es mejor que A ejecutando los programas 1 y 2. Se puede decir, también, que A es mejor ejecutando los programas 3, 4 y 5. ¿Cómo zanjar el conflicto?

Lo primero que se requiere es obtener una única medida de desempeño. Un sólo número que refleje la evaluación hecha. Una buena opción podría ser considerar el tiempo total de ejecución, listado en el último renglón de la tabla. Considerándolo como una medida de síntesis del desempeño, podríamos decir que tanto A como B son igualmente buenos. En general el tiempo total de ejecución puede ser una manera correcta de sintetizar el desempeño, pero quisiéramos tener un parámetro estadísticamente más significativo. Que no dependa ni de cuantos programas se ejecutan ni de cuantas veces se ejecutan, que nos proporcione una estimación del desempeño del equipo en su uso cotidiano. Lo que queremos es, de hecho, lo que los estadísticos denominan una *medida de tendencia central*: un número alrededor del cual, en general, se encontrará las más de las veces el desempeño de la máquina.

Existen varias medidas de tendencia central usadas en estadística. La *moda* es una de ellas, definida como el valor de una muestra de datos que más se repite, puede ni siquiera existir. La *mediana*, definida como el punto medio entre los valores extremos de la muestra, tiene la desventaja de dejarse llevar fácilmente por valores excesivamente altos o excesivamente bajos en la muestra. Necesitamos medidas de tendencia central mucho más robustas. Por fortuna tenemos aún mucho de donde escoger. Si $D = \{\delta_1, \delta_2, \ldots, \delta_n\}$ es una muestra de n datos, se definen:

Programa	**Computadora A**	**Computadora B**
Prog1	20	1
Prog2	15	2
Prog3	10	15
Prog4	10	27
Prog5	10	20
Total	**65**	**65**

Tabla 2.1: Tiempos de ejecución en milisegundos de cinco programas para dos sistemas de cómputo diferentes.

Media aritmética

$$\mathrm{A}(D) = \frac{1}{n}\sum_{i=1}^{n}\delta_i \tag{2.1}$$

Media armónica

$$\mathrm{H}(D) = \frac{n}{\sum_{i=1}^{n}\frac{1}{\delta_i}} \tag{2.2}$$

Media geométrica

$$\mathrm{G}(D) = \sqrt[n]{\prod_{i=1}^{n}\delta_i} \tag{2.3}$$

La media aritmética es lo que normalmente llamamos el *promedio* de la muestra de datos y es, con mucho, la medida de tendencia central más usada en la vida cotidiana, además de ser la más simple de calcular. Sin embargo puede no ser la mejor en toda circunstancia. Es bien conocida, por ejemplo, su indeseable tendencia a dejarse llevar por valores excesivamente grandes respecto a la mayoría de los datos (*outliers*) de una muestra.

Dados un conjunto de datos, la media aritmética siempre será la mayor de las tres, la armónica la menor y la geométrica estará siempre entre ambas. La media armónica, por cierto, es el inverso de la media aritmética de los

Velocidad (tareas/min)
70
30
40
60

Tabla 2.2: Las cuatro velocidades a las que puede operar un sistema de cómputo. La media aritmética es 50 tareas/minuto, la media armónica es 44.8 tareas/minuto.

inversos de los elementos de la muestra, por lo que también se puede sesgar indeseablemente ante valores relativamente alejados del resto de la muestra. La media geométrica, por su parte, tiene un significado interesante: es el tamaño que debería tener el lado de un hipercubo para tener un volumen igual al hiperprisma cuyos lados son los elementos de la muestra. Sí, por ejemplo, tenemos $\{3, 5, 4\}$ como muestra y consideramos estos como las medidas en centímetros del ancho, alto y largo de un prisma cuadrangular, el volumen del prisma sería 60 cm^3. La media geométrica de la muestra es 3.915, si suponemos que esto es el largo en centímetros del lado de un cubo, entonces el volumen del cubo es igual al del prisma previo.

Cada media es útil por sí misma como medida de tendencia central en ciertas circunstancias. No todas son utilizables a ultranza. Supongamos, por ejemplo, que tenemos un sistema que puede realizar cierto tipo de tareas en cuatro velocidades diferentes como se muestra en la tabla 2.2. La media aritmética de las velocidades es de 50 tareas/min, mientras la media armónica es de 44.8 tareas/min.

Supongamos ahora que ponemos a funcionar este sistema durante dos minutos en cada uno de los regímenes de velocidad, haciendo entonces un total de 8 minutos. En la tabla 2.3 se muestran los resultados. El número total de tareas ejecutadas es 400. Si usamos la media armónica (44.8 tareas/min) como medida de tendencia central y decimos que ese fue la tasa a la que se ejecutaron tareas durante 8 minutos, entonces obtenemos: $44.8 \times 8 = 358.4$ tareas ejecutadas, lo que discrepa de las 400 que obtenemos sumándolas. Si en cambio usamos la media aritmética (50 tareas/min) como la tasa de ejecución durante los 8 minutos obtenemos: $50 \times 8 = 400$ tareas, lo que es apegado a la realidad.

Vel (tareas/min)	**Trabajo (min)**	**#tareas**
70	2	140
30	2	60
40	2	80
60	2	120
Total	8	400

Tabla 2.3: Resultado de ejecutar tareas durante dos minutos en cada régimen en el sistema de ejemplo. La suma del total de tareas ejecutadas es 400.

Vel (tareas/min)	**Tareas**	**Tiempo (min)**
70	14	0.2
30	14	0.47
40	14	0.35
60	14	0.23
Total	56	1.25

Tabla 2.4: Resultado de ejecutar 14 tareas en cada régimen en el sistema de ejemplo. La suma del total de tiempo es 1.25 minutos.

Si, en cambio, la carga de trabajo fija es el número de tareas realizadas en cada régimen, como se muestra en la tabla 2.4, el tiempo total de ejecución (suma de la tercera columna) es de 1.25 minutos. Si consideramos la media aritmética como la tasa a la que se ejecutan las 56 tareas totales, entonces el tiempo estimado sería: $56/50 = 1.12$ minutos, muy distinto de los 1.25 minutos. Por el contrario, si consideramos la media armónica, entonces obtenemos: $56/44.8 = 1.25$ minutos, tal como debe ser.

Así que cuando la cantidad que se desea medir es directamente proporcional a la métrica de las muestras se debe usar la media aritmética. Por el contrario, si la relación es inversamente proporcional, se debe usar la media armónica. Es también conveniente usar la media armónica cuando tenemos una muestra grande de valores uniformemente distribuidos y unos pocos va-

Producción (ton)	**Factor de crecimiento**
100	
130	1.3000
180	1.3846
240	1.3333
305	1.2708
345	1.1311
M. arit.	1.284
M. geom.	1.281
M. armo.	1.278

Tabla 2.5: Producción anual, en toneladas, de una empresa productora de cereales durante los últimos seis años.

lores excesivamente altos (la media armónica siempre es menor o igual a la aritmética).

La media geométrica tiene también su propio ámbito de uso adecuado. Supongamos que la tabla 2.5 muestra la producción anual de una empresa agrícola productora de cereales.

Si consideramos la media aritmética del factor de crecimiento como la tasa promedio del mismo, obtendríamos: $100 \times 1.284^5 = 348.98$ toneladas de producción en el último año, mayor que la producción real. Si usáramos la media armónica obtendríamos $100 \times 1.278^5 = 340.91$ lo que es inferior a la producción real. Pero si usamos la media geométrica tendríamos: $100 \times 1.281^5 = 345$ lo que realmente coincide con la producción del último año. En este caso la media geométrica es la medida de tendencia central que se debe usar. La métrica que usamos es, realmente, una razón sin unidades, comparamos una magnitud con otra de la misma métrica. La media geométrica es lo adecuado para obtener medidas de tendencia central en razones de proporción o *tasas* (lo que suele llamarse *ratio* en inglés). Más adelante veremos una cualidad adicional muy útil de esta media.

En el conjunto de programas que se utilizan en una prueba de desempeño habrá algunos más representativos que otros. Para juzgar correctamente

habría que asignar a cada programa un cierto peso que fuera indicativo de qué tan estadísticamente representativo es. Podríamos pensar entonces en usar una media aritmética en la que se asocie un peso diferente a cada uno de los programas en función de su representatividad. La medida en cuestión es la *media aritmética ponderada*. Dada una muestra $D = \{\delta_1, \delta_2, \ldots, \delta_n\}$ de n datos y un conjunto de *pesos* $W = \{w_1, w_2, \ldots, w_n\}$, tales que $\sum_{i=1}^{n} w_i = 1$, la media aritmética ponderada se define como:

$$\mathrm{P}(D, W) = \sum_{i=1}^{n} w_i \, \delta_i \tag{2.4}$$

Así la media aritmética convencional resulta ser un caso particular de la ponderada en el que los pesos de los programas son todos iguales a $1/n$.

Tendríamos ahora la dificultad de elegir adecuadamente los pesos de los programas. Otra opción es fijar una máquina como patrón de medida. Decidir que hay una cierta computadora p que posee una arquitectura suficientemente general que le concede un comportamiento representativo y referir los tiempos de ejecución a esa máquina. Los tiempos a considerar deben ser entonces normalizados, divididos por los tiempos en la computadora de referencia. Así, el valor a considerar para el i-ésimo programa sería:

$$\hat{t}_i = \frac{t_{i,c}}{t_{i,p}} \tag{2.5}$$

En este contexto la medida de tendencia central adecuada sería la media geométrica, dado que los valores de la muestra son cocientes de tiempos, razones. Utilizar la media aritmética, por ejemplo, puede entregar resultados inconsistentes, en función de la máquina usada como referencia.

En la tabla 2.6 se muestran los tiempos de ejecución en milisegundos para tres diferentes programas en tres diferentes computadoras: A, B y C. En las tablas 2.7, 2.8 y 2.9 se muestran los tiempos normalizados usando a A, B y C como máquina de referencia, respectivamente.

Como puede observar el lector, si la medida de desempeño usada es el tiempo normalizado total o su media aritmética, entonces la computadora ganadora es justo la que se elije como referencia. La única medida consistente es la obtenida usando la media geométrica. En ese caso la máquina ganadora es siempre la C, lo que es congruente con el tiempo total de ejecución de la prueba de la tabla 2.6. Este comportamiento resulta del hecho de que la media geométrica de los cocientes de dos muestras, es igual al cociente de las medias geométricas de las muestras, es decir:

$$G\left(\frac{X}{Y}\right) = \frac{G(X)}{G(Y)}$$

Programa	**A**	**B**	**C**
Prog1	5	13	35
Prog2	730	250	45
Prog3	1200	230	50
Total	**1935**	**493**	**130**

Tabla 2.6: Tiempos de ejecución (ms) para tres programas en tres diferentes máquinas.

Programa	**A**	**B**	**C**
Prog1	1	2.6	7
Prog2	1	0.34	0.06
Prog3	1	0.19	0.04
Total	**3**	**3.13**	**7.1**
M. arit.	**1**	**1.04**	**2.37**
M. geom.	**1**	**0.55**	**0.26**

Tabla 2.7: Desempeño de los tres sistemas de ejemplo, usando los tiempos de A para normalizar.

Programa	**A**	**B**	**C**
Prog1	0.38	1	2.69
Prog2	2.92	1	0.18
Prog3	5.22	1	0.22
Total	**8.52**	**3**	**3.09**
M. arit.	**2.84**	**1**	**1.03**
M. geom.	**1.8**	**1**	**0.47**

Tabla 2.8: Desempeño de los tres sistemas de ejemplo, usando los tiempos de B para normalizar.

Programa	**A**	**B**	**C**
Prog1	0.14	0.37	1
Prog2	16.22	5.56	1
Prog3	24	4.6	1
Total	**40.37**	**10.53**	**3**
M. arit.	**13.46**	**3.51**	**1**
M. geom.	**3.82**	**2.12**	**1**

Tabla 2.9: Desempeño de los tres sistemas de ejemplo, usando los tiempos de C para normalizar.

La desventaja de usar la media geométrica de tiempos normalizados es que perdemos la noción intuitiva que deseábamos en principio: poder decir, "el sistema X tiene un desempeño 3.5 veces mejor que el sistema Y". Como lo que se maneja son razones de tiempos y se pasan por la media geométrica ya no hay una noción intuitiva de nada. El hecho de que la media aritmética "se deje llevar" por valores grandes o pequeños, significa, desde el punto de vista positivo, que se toma en cuenta la información que se recibe, si deseamos que nuestra medida no se sesgue tan fácilmente es porque, implícitamente, algo de la información se pierde. Todo tiene un costo.

Las ventajas, sin embargo son grandes: no se sesga ante valores extrañamente grandes o pequeños; se puede usar un conjunto de valores normalizados sin que importe la elección de los valores de referencia. Es por esto que la media geométrica fue la elegida para sintetizar las pruebas de desempeño más científicas que poseemos y que nos ocuparán en la siguiente sección.

Es conveniente señalar el hecho interesante de que, las tres medias: armónica, aritmética y geométrica, pueden ser vistas como una única media: la aritmética, sólo que con métricas diferentes. Si tomamos los valores tal cual, es decir los transformamos usando la identidad, la media aritmética es justo ella misma. Si en cambio transformamos los valores tomando sus inversos multiplicativos entonces la media aritmética se transforma en la media armónica. Por último, si los valores son transformados usando sus logaritmos, entonces sumarlos es como tomar su producto, la media aritmética se transforma en la geométrica.

2.1. Pruebas de desempeño

Al conjunto de programas usados para evaluar dicho desempeño se le llama *carga de trabajo* o *workload* en inglés y, evidentemente, es fundamental la elección de esta carga de trabajo para dar validez a la prueba de desempeño. A fin de cuentas queremos una medida de desempeño para poder predecir cómo se comportará un sistema cuando sea usado en el mundo real, así que la carga de trabajo usada en la prueba debe ser diseñada de tal forma que permita estimar, tan exactamente como sea posible, el desempeño del equipo cuando esté en operación.

Existen varias opciones para diseñar la carga de trabajo:

1. Programas reales. Ejemplo: correr un conjunto de programas populares en dos computadoras diferentes, ejecutando las mismas tareas y comparar los tiempos de ejecución.

2. Kernels. Son trozos de programas reales que aíslan el desempeño de ciertas características individuales. Por ejemplo, el famoso *Linpack* que prueba el desempeño en tareas relacionadas con álgebra de matrices o *Livermore loops* para medir la eficiencia en la ejecución de ciclos. Los kernels no son útiles por sí mismos.

3. Benchmarks de juguete. De 10 a 100 lineas de código que produce resultados conocidos. Por ejemplo la tradicional criba de Eratóstenes o Quicksort. Era usual hace algunas décadas reportar el desempeño de un sistema diciendo cuanto tiempo tardaba obteniendo los primeros n números primos o cuanto tardaba ordenando una secuencia de enteros.

4. Benchmarks sintéticos. No son programas útiles por si mismos, en general es un programa diseñado para tratar de simular la frecuencia promedio de las instrucciones que se ejecutan. Algunos fabricantes de computadoras comenzaron a hacer modificaciones a sus compiladores de tal forma que al compilar un benchmark de este tipo se hicieran optimizaciones no estándares para obtener mejores resultados en el benchmark falsificando los resultados[1]. Los más famosos son *Whetstone* y *Dhrystone*.

Otra opción es hacer algo que mezcle varias de estas opciones en aras de construir una prueba que sea a la vez, apegada a la realidad, representativa y general. Esa es la opción que eligió SPEC (siglas de *Standard Performance Evaluation Corporation*) en su prueba comparativa que es, con mucho, la más científica y reproducible con la que contamos. La prueba general de SPEC evalúa el desempeño en dos rubros: aritmética entera y de punto flotante (conocidas como CINT y CFP, respectivamente), usando dos métricas diferentes: una basada en el tiempo de respuesta y la otra en el rendimiento (denominada con el sufijo *rate*). El conjunto de prueba de SPEC está constituido por varios programas de uso común como el compilador de C de GNU, algunos programas de compresión de datos como `gzip` o `bzip2` en la parte de aritmética entera o algunos programas de modelación científica en la parte de punto flotante. Todos ellos se ejecutan con una entrada determinada, usando banderas específicas para compilarlos.

Las baterías de prueba (*benchmark suites*) son colecciones de programas o fragmentos de ellos, al estilo de los kernels. La intención es proporcionar resultados reproducibles y que realmente den una idea del desempeño del sistema en condiciones reales respecto a otros sistemas. Los programas elegidos

[1] De hecho es posible optimizar tirando a la basura el 25 % del código de Dhrystone, uno de los benchmarks más populares.

deben ser entonces, estadísticamente significativos y las medidas obtenidas, más que dar un resultado que sea, *per se*, indicativo, proporcionan un criterio para comparar el sistema evaluado con otros. Las baterías más famosos y usuales son las de SPEC (*Systems Performance Evaluation Corporation*) una organización internacional no lucrativa entre cuyos miembros se cuentan las principales compañías fabricantes de hardware y software, universidades, centros de investigación y compañías consultoras.

SPEC ha elaborado diferentes baterías de pruebas para evaluar diferentes aspectos de los sistemas. Algunas resultan entonces mucho más útiles que otras en cierto contexto. Hay una batería para evaluar servidores web, otra para estaciones de trabajo de capacidades gráficas, otra para servidores de correo electrónico y otra, para evaluar el desempeño del sistema en general. Esta última es la prueba de mayor utilidad general y es, por tanto, la más usual. Se encuentra dividida en dos partes: la que evalúa el desempeño del sistema ejecutando tareas que involucran aritmética entera (las más usuales), llamada CINT2006 en su versión mas reciente (2011) y la que evalúa el desempeño del sistema utilizando aritmética de punto flotante, CFP2006. Cada una de estas partes evalúa el desempeño usando los dos tipos de métricas mencionados: tiempo de respuesta (SPECint2006 y SPECfp2006), llamada *speed* en la documentación de SPEC y rendimiento (SPECint_rate2006 y SPECfp_rate2006), llamada *throughput* por SPEC.

Cualquier persona u organización puede ejecutar las baterías de SPEC y reportar los resultados obtenidos. Pero para evitar que, atendiendo a intereses comerciales, las compañías modifiquen la batería o manipulen el ambiente en el que se ejecuta la prueba para obtener resultados mejores, SPEC fija todos los parámetros de control de la prueba, por ejemplo las banderas del compilador y los datos de entrada a los programas de la prueba. Cualquier reporte de desempeño que se diga acorde con SPEC está obligado a contener los resultados atendiendo a las restricciones de SPEC, a lo que se le llama reporte de desempeño *base*. Adicionalmente, si la entidad que hace el reporte lo desea, puede relajar las restricciones y reportar lo que en terminología de SPEC se denomina máximo desempeño *peak performance*. La intención es dar a los fabricantes la libertad de decir qué tan bien se desempeñan sus sistemas pero sin pretender engañar al público. El reporte base, tanto en punto flotante como en aritmética entera, es lo que el sistema en condiciones estándar puede lograr y el público podría, si quisiera, reproducir el experimento y obtener los mismos números. Es un punto de partida confiable. El reporte de desempeño máximo puede o no, ser tomado en cuenta por el público.

2.2. Ley de Amdahl

Toda decisión en el ámbito del diseño de computadoras (como en la mayoría de los ámbitos de la vida en general), está regida por una relación costo–beneficio. Así que necesariamente se debe poder evaluar, en particular, el beneficio que se obtendría de adoptar una decisión particular que mejora, de alguna manera, el estado actual de las cosas. En 1967 Gene Amdahl, entonces arquitecto de computadoras en IBM, formuló lo que hoy conocemos como la *ley de Amdahl* con el propósito de evaluar la mejora en el desempeño en computadoras de procesamiento paralelo. La regla encontrada por Amdahl es, sin embargo, aplicable de manera general para cuantificar el beneficio obtenido al introducir una cierta mejora en un sistema preexistente.

Para evaluar el beneficio de introducir una mejora en un proceso es buena idea comparar, por ejemplo, el tiempo que tomaba la realización del proceso sin la mejora con el tiempo luego de introducirla. Conviene, entonces usar una expresión del tipo:

$$\frac{Tiempo_{sin}}{Tiempo_{con}} \tag{2.6}$$

Son más bien raras las situaciones en las que, en un proceso constituido de varias etapas, es posible introducir una mejora que tenga impacto directo en todas y cada una de ellas. Generalmente la mejora incide sobre una de ellas. Claro que esto tiene impacto sobre el desempeño total a la largo del proceso, pero sólo indirectamente. Convendría entonces utilizar una evaluación como la anterior, en diferentes niveles: para cuantificar el impacto directo de la mejora en la etapa para la que fue diseñada y después para evaluar su impacto indirecto a nivel global sobre todo el proceso. Llamaremos a estas *ganancia bruta* y *ganancia neta*, respectivamente.[2] Aclaremos mediante un ejemplo.

Imaginemos que debemos hacer, tan rápido como sea posible, un recorrido a campo traviesa que comprende dos diferentes tipos de terreno: una parte en terreno irregular, pedregoso o cubierto de hierba entre el bosque y otra sobre nieve no muy firme. Si hacemos todo el recorrido usando unas botas de *trekking* nos desempeñaremos razonablemente bien en la parte boscosa del recorrido, pero tendremos problemas para andar sobre nieve blanda cuando se nos undan los pies y nuestro avance sea muy lento. Una mejora

[2]Se ha utilizado el término *ganancia* para denotar lo que en la literatura en inglés suele llamarse *speedup*. Por una parte la traducción literal es "aceleración" lo que no resulta estrictamente correcto y además hace pensar que la ley de Amdahl es aplicable sólo a mejoras en el tiempo.

posible es llevar un par de raquetas de nieve o *snowshoes* que distribuyen el peso en una superficie mayor e impiden el hundimiento de los pies. Digamos que, sobre la nieve, cuando usamos las raquetas podemos ir al doble de velocidad que si no las usamos, es decir, la ganancia (g), *específicamente en el tramo en el que pueden usarse las raquetas*, es de 2. Formalmente diríamos:

$$g = \frac{Tiempo_{sin}}{Tiempo_{con}} = 2$$

Esta es una evaluación de la ganancia bruta obtenida al usar la mejora. Porque no es cierto que la mejora pueda usarse durante toda la ejecución de la tarea encomendada. Supongamos que, de los 30 km del recorrido, sólo 4 km son sobre nieve. Es decir el 13.33 % del recorrido es sobre nieve y el restante 86.66 % sobre terreno "normal". Podríamos decir, entonces, que la fracción F del recorrido en que es posible usar las raquetas es: 0.1333, mientras que no es posible usarlas en la fracción: $1 - F = 0.8666$.

Si suponemos que, usando sólo botas de *trekking*, el recorrido completo lo hacemos en un tiempo $T_0 = 12$ horas, podríamos decir que, usando las raquetas en el tramo en que es posible usarlas, el tiempo que usaríamos en el recorrido completo debe ser:

$$T_m = T_0 \left(0.8666 + \frac{0.1333}{2}\right) = 0.9333\, T_0 \approx 11.2$$

lo que podemos leer "nos tardamos el tiempo usual cuando no podemos usar la mejora y la mitad cuando sí podemos usarla".

Ahora podemos hacer un cociente análogo al expresado en 2.6 pero entre el los tiempos totales y no sólo en la fracción en que se puede usar la mejora. Esta es, entonces, una evaluación cuantitativa de la ganancia neta:

$$G = \frac{T_0}{T_m} = \frac{12}{11.2} = 1.07$$

lo que significa que, en general, el uso de raquetas de nieve en el trayecto mejora el tiempo de recorrido en un 7 %.

Generalizando lo hecho en el ejemplo. Si g es la ganancia bruta de una mejora introducida en una fracción F de un proceso P, T_0 el tiempo total de ejecución de P sin usar la mejora, T_m el tiempo total de ejecución de P usando la mejora en la parte en que es posible; entonces:

$$T_m = T_0 \left[(1 - F) + \frac{F}{g}\right]$$

de donde, la *ganancia neta* obtenida por la introducción de la mejora es:

$$G = \frac{T_0}{T_m} = \frac{1}{(1-F) + \frac{F}{g}} \tag{2.7}$$

Esta es una medida HIB y, dado que $T_m \leq T_0$, es una fracción impropia: la mejora es tanto más insignificante cuanto más cercano a uno es el cociente de 2.7. La regla coloquial que podemos extraer de la ley de Amdahl es: *hay que hacer mejor lo más común*. La ecuación 2.7 justamente nos permite cuantificar que tan mejor es lo "mejor" (factor g) y que tan común es lo "común" (factor F).

2.3. Costo–beneficio

En general, como hemos dicho, las decisiones que un arquitecto de hardware toma, están regidas por dos factores fundamentales: el desempeño del equipo y el costo de lo necesario para obtenerlo. Es conveniente, por cierto, aclarar que por "costo" no nos referimos ahora sólo al costo cuantificado en términos monetarios. A fin de cuentas se traducirá justamente en eso, pero no es ni necesario, ni recomendable, usar sólo el costo monetario como medida de lo necesario para obtener un cierto desempeño.

En este rubro es útil usar medidas como por ejemplo:

- (Desempeño SPEC) / precio.
- (Instrucciones / segundo) / (área en el chip).
- (Transacciones / segundo) / precio.
- (Frecuencia de reloj) / temperatura.
- (Tareas ejecutadas / segundo) / (Watt).

2.4. Malas métricas

Son populares algunas métricas para evaluar el desempeño y que, en general, no proporcionan una idea clara del comportamiento del sistema en su uso cotidiano. Parecen prácticas e intuitivas, pero al usarlas arbitrariamente se corre el riesgo de terminar con una idea completamente equivocada de la realidad.

Un ejemplo clásico de esto es lo que suele llamarse MIPS (Millones de Instrucciones Por Segundo).

$$MIPS = \frac{\text{Número de instrucciones}}{\text{Tiempo de ejecución} \times 10^6} = \frac{F}{R \times 10^6}$$

donde R y F tienen el significado que les fue atribuido en la ecuación fundamental de desempeño (1.5), es decir ciclos por instrucción (*clocks per instruction*) y frecuencia de operación, respectivamente. Por ejemplo, en una máquina de 700 MHz cuya tasa de ejecución sea de 6 ciclos por instrucción:

$$MIPS = \frac{700,000,000}{6,000,000} = 116.6$$

La ventaja de usar esta medida de desempeño es que es fácil de entender, sobre todo por personas no muy avezadas en el área, es intuitiva.

Las desventajas son que:

- Depende del conjunto de instrucciones de la máquina. Una sola instrucción en una arquitectura particular, puede equivaler a toda una rutina en otra.
- No todas las instrucciones tardan lo mismo, así que dependiendo de las instrucciones que utilice un programa la medida en MIPS varía. Entonces ¿cuales instrucciones del conjunto total del procesador se deben usar? ¿las más usuales? ¿todas? ¿las más económicas en tiempo?

Incluso puede ocurrir que una máquina A con mejor desempeño real que otra B resulte con una medida en MIPS menor. Por ejemplo, si A tiene unidad de punto flotante (*floating point unit* o FPU) y tarda 8 ns. en una multiplicación mientras B tiene que emular la multiplicación a través de una rutina de 200 instrucciones tardándose 16 ns. resultará que $MIPS(B) > MIPS(A)$.

También ocurre que no se considera la "calidad" de las operaciones realizadas. 100 sumas de punto flotante tardan menos que 100 divisiones enteras, pero siguen siendo 100 instrucciones.

Aún el tiempo de ejecución está sujeto a diversas interpretaciones. Supongamos que se ejecuta un programa en una computadora, el tiempo de ejecución puede ser considerado como:

- el tiempo que tarda en completarse el programa incluyendo entrada/-salida, accesos a memoria, etc.

- el tiempo efectivo en el que el procesador está ejecutando las instrucciones del programa.

En el primer caso estamos hablando del tiempo que le tomó al sistema completo (incluyendo dispositivos de e/s, sistema operativo, etc.) la ejecución del programa. Estamos midiendo el desempeño del sistema. En el segundo caso estamos midiendo el desempeño de la unidad central de proceso (CPU) únicamente.

La popularidad del sistema operativo Linux puso en uso una unidad similar llamada *BogoMIPS*. Linus Torvalds mismo, quien introdujo la medida en la versión 0.99.11 del kernel de Linux en 1993, dice haberla nombrado BogoMIPS en referencia a la palabra "bogus", algo que es falso. En efecto, la medida en BogoMIPS de una computadora es sólo un truco técnico para poder sincronizar adecuadamente el hardware con el software, para determinar tiempos de espera adecuados para verificar el estado de algunos dispositivos, no para evaluar el desempeño del sistema.

En algunos lugares se utiliza también la frecuencia de operación del reloj del sistema como medida de desempeño. Claro, esto sólo es válido en las mismas circunstancias que los MIPS, cuando se comparan procesadores de la misma arquitectura de conjunto de instrucciones. Sólo si las instrucciones ejecutadas son exactamente las mismas, es valido usar el ritmo de trabajo como medida de la rapidez.

En los sistemas de alto desempeño, orientados a cómputo científico suele usarse una medida similar llamada MFLOPS (megaflops), la cantidad de millones de instrucciones de punto flotante ejecutadas por segundo. En este caso particular la medida no es del todo injusta: se comparan máquinas orientadas a las mismas tareas, cuantificando cuantas instrucciones, de un cierto tipo específico, se ejecutan por unidad de tiempo.

Bibliografía

[1] Hennessy, J. y D. Patterson, *Computer Architecture: A Quantitative Approach*, 6a Ed. Morgan Kaufmann, 2017.

[2] Jain, Raj, *The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons Inc, 1991.