4 Trabajo a realizar

Nos interesaría experimentar con algún ejemplo que permita obtener ventaja sobre `gcc`, y que al mismo tiempo sea lo suficientemente sencillo como para estudiarlo y programarlo en pocas sesiones de prácticas. O aún mejor, que no requiera estudio adicional, porque ya lo hayamos estudiado.

Estas condiciones las cumple por ejemplo el cálculo del peso Hamming o *"population count"*, ya visto en clase de teoría, para el cual existe una instrucción SSSE3 `pshufb` y otra SSE4.2 `popcount` cuyo uso `gcc` no podrá deducir a partir de nuestro código C.

Se trata por tanto de programar varias versiones de una función que sume los pesos Hamming (nº de bits activados) de todos los elementos de un *array*, con y sin ensamblador en-línea, y comprobando siempre la corrección del resultado calculado. Para esto último, podemos consensuar algunos ejemplos pequeños de prueba, cuyo resultado correcto pueda calcularse a mano. Pero para que el tiempo de medición sea apreciable (y reproducible) tendremos que usar *arrays* de mayor tamaño, y para conocer entonces el resultado correcto hará falta una fórmula aplicable a los datos de entrada utilizados.

Iremos sugiriendo las distintas versiones, dando pistas sobre la idea general que debe implementar el código C y/o el tramo de código ASM, y sobre las instrucciones máquina y restricciones a utilizar, al objeto de guiar, orientar y acelerar tanto la localización y lectura de información en los apuntes de clase y en el manual del repertorio de instrucciones como la programación de los tramos `asm()`.

Cada vez que se desarrolle una nueva versión debería comprobarse su corrección con los ejemplos de tamaño pequeño y con el de tamaño grande, procediendo a su revisión y/o depuración si no produjera el resultado correcto. Ofreceremos sugerencias de compilación condicional y los correspondientes comandos del *shell bash* para automatizar esta tarea lo máximo posible.

Tras finalizar el desarrollo de las versiones, deberíamos cronometrarlas de forma justa y equitativa. Repetiremos 10 veces la ejecución del ejemplo grande para promediar los tiempos de ejecución de cada versión, y repetiremos el estudio para distintos niveles de optimización. Los tiempos promediados se pueden presentar en una gráfica del paquete ofimático disponible en Ubuntu (hoja de cálculo Calc). Daremos también sugerencias de compilación condicional y los correspondientes comandos del *shell bash* para automatizar las mediciones. También daremos indicaciones sobre el formato gráfico deseado.

Se propondrá comenzar con las funciones C estudiadas en clase (adaptadas a calcular el *popcount* del *array* completo), primero la más costosa, después la más eficiente. Propondremos entonces una versión ASM intentando mejorarla, como ejercicio preparatorio. Seguiremos con una ingeniosa propuesta de nuevo en lenguaje C, y por último recurriremos a las citadas instrucciones del repertorio multimedia.

Según la temporización de cada curso, se procurarán realizar guiadamente (como Seminario Práctico) los Ejercicios 1-6 aproximadamente (incluso 7-9 si sobrara tiempo). Aunque no diera tiempo a tanto, comprender los programas mostrados y ejercitarse en el uso de las herramientas son competencias que cada uno debe conseguir personalmente.

Para aprender el funcionamiento de nuevas instrucciones (sean o no del repertorio SSE) basta con leer el manual de Intel y probarlas en la propia sentencia ASM *inline* (y depurarlas con `gdb -tui`, si no produjeran el resultado esperado).

Al objeto de facilitar el desarrollo progresivo de la práctica, se sugiere realizar en orden las siguientes tareas:

4.0 Repasar los apuntes de clase

Esta práctica es posterior o simultánea al estudio en clase de teoría de diversos conceptos relevantes [1], como por ejemplo: códigos de condición (Tema 2.2, tr. 3-11), bucles (tr. 22-33), estructura de la pila (Tema 2.3, tr. 8-11), convenciones de llamada (tr. 19-21, tr. 44-49), punteros y variables locales (tr. 50-51), declaración y acceso a *arrays* (Tema 2.4, tr. 3-14). Se recomienda su estudio detallado.

4.1 Calcular la suma de bits de una lista de enteros sin signo

Utilizar el programa `suma_09` de la Figura 11 como esqueleto para cronometrar diversas versiones de una función que sume los bits (peso Hamming, *popcount*) de los elementos de una lista de N enteros. Notar que la suma puede llegar a ser 32*N, si todos valieran 2^{32}-1 (y por consiguiente tuvieran activados todos los bits). Concluir que basta calcular la suma en un entero, para cualquier valor práctico de N. *¿Cómo de grande puede ser N en dicho peor caso?*

Para tener alguna posibilidad de detectar errores en nuestro código, lo comprobaremos con algunos ejemplos sencillos como los siguientes:

- `unsigned lista[SIZE]={0x80000000, 0x00400000, 0x00000200, 0x00000001};`
- `unsigned lista[SIZE]={0x7fffffff, 0xffbfffff, 0xfffffdff, 0xfffffffe, 0x01000023, 0x00456700, 0x8900ab00, 0x00cd00ef};`
- `unsigned lista[SIZE]={0x0 , 0x01020408, 0x35906a0c, 0x70b0d0e0, 0xffffffff, 0x12345678, 0x9abcdef0, 0xdeadbeef};`

Los resultados deberían ser 4, 156, 116, respectivamente. Notar que la lista se declara sin signo para que los desplazamientos (que previsiblemente tendremos que utilizar) no dupliquen ningún bit de signo. *¿Por qué?*

Para poder comparar los tiempos de ejecución de distintos programas (de distintos usuarios) en el laboratorio (con los mismos ordenadores), y también de un mismo programa (de un estudiante) en distintos ordenadores (laboratorio y portátil personal), necesitaremos acordar algún ejemplo de tamaño mayor, como por ejemplo SIZE=2^{20} elementos, inicializados en secuencia desde 0 hasta SIZE-1. Calcular qué peso Hamming tiene ese ejemplo (en función de SIZE) y modificar la fórmula del programa `suma_09` acordemente. Para ello podemos (en realidad debemos, si es que queremos calcularlo con una fórmula) aprovechar razonamientos específicos para el *array* en cuestión. Pista: ¿se puede aplicar algún razonamiento al bit 0 de todos los elementos? ¿Y al bit 19? ¿Y a los intermedios? *¿Qué fórmula sale?*

Recomendamos usar un esquema de compilación condicional similar al usado en la práctica anterior, para incorporar la definición de los tests y el resultado esperado.

```
#include <stdio.h>            // para printf()
#include <stdlib.h>           // para exit()
#include <sys/time.h>         // para gettimeofday(), struct timeval
#define SIZE (1<<16)          // tamaño suficiente para tiempo apreciable
int lista[SIZE];
int resultado=0;

#ifndef TEST
#define TEST 5
#endif

/* ------------------------------------------------------------------- */
  #if   TEST==1
/* ------------------------------------------------------------------- */
  #define SIZE 4
  unsigned lista[SIZE]={0x80000000, 0x00400000, 0x00000200, 0x00000001};
  #define RESULT 4
/* ------------------------------------------------------------------- */
#elif   TEST==2
/* ------------------------------------------------------------------- */
  ...
/* ------------------------------------------------------------------- */
#elif   TEST==4 || TEST==0
/* ------------------------------------------------------------------- */
  #define NBITS 20
  #define SIZE (1<<NBITS)     // tamaño suficiente para tiempo apreciable
  unsigned lista[SIZE];       // unsigned para desplazamiento derecha lógico
  #define RESULT ( ? * ( ? << ?-1 ) )       // pistas para deducir fórmula
/* ------------------------------------------------------------------- */
#else
       #error "Definir TEST entre 0..4"
#endif
/* ------------------------------------------------------------------- */

int popcount1(unsigned* array, size_t len)
{
    ...
}
```

```
void crono(int (*func)(), char* msg){
    struct timeval tv1,tv2;                 // gettimeofday() secs-usecs
    long           tv_usecs;                // y sus cuentas

    gettimeofday(&tv1,NULL);
    resultado = func(lista, SIZE);
    gettimeofday(&tv2,NULL);

    tv_usecs=(tv2.tv_sec -tv1.tv_sec )*1E6+
             (tv2.tv_usec-tv1.tv_usec);
#if TEST==0
    printf(    "%ld" "\n",      tv_usecs);
#else
    printf("resultado = %d\t", resultado);
    printf("%s:%9ld us\n", msg, tv_usecs);
#endif
}

int main()
{
#if TEST==0 || TEST==4
    size_t i;                               // inicializar array
    for (i=0; i<SIZE; i++)
        lista[i]=i;
#endif

    crono(popcount1 , "popcount1 (lenguaje C -       for)");
    crono(popcount2 , "popcount2 (lenguaje C -     while)");
    crono(popcount3 , "popcount3 (leng.ASM-body while 4i)");
    crono(popcount4 , "popcount4 (leng.ASM-body while 3i)");
    crono(popcount5 , "popcount5 (CS:APP2e 3.49-group 8b)");
    crono(popcount6 , "popcount6 (Wikipedia- naive - 32b)");
    crono(popcount7 , "popcount7 (Wikipedia- naive -128b)");
    crono(popcount8 , "popcount8 (asm SSE3 - pshufb 128b)");
    crono(popcount9 , "popcount9 (asm SSE4- popcount 32b)");
    crono(popcount10, "popcount10(asm SSE4- popcount128b)");

#if TEST != 0
    printf("calculado = %d\n", RESULT);
#endif

    exit(0);
}
```

Figura 12: popcount.c: esqueleto de suma_09 con sugerencias de compilación condicional para los tests

Observar que `lista` es ahora `unsigned`, que las definiciones de `lista`, `SIZE` y `RESULT` varían con el test escogido, que los tests pequeños (1, 2, 3) se definen fácilmente mediante inicializador, y que el test grande (4) es mejor definirlo programáticamente. Observar que se ha definido un test 0 igual al 4 pero que imprime tan sólo el valor numérico del tiempo cronometrado. Posteriormente se explicará por qué. En el esquema quedan tan sólo por rellenar las distintas versiones, que procedemos a sugerir.

Realizar una **primera** y **segunda** versiones C parecidas las vistas en clase (Tema 2.2, tr.30 y 29, pero aplicadas a enteros de 4 bytes, y de hecho a un *array* de ellos), recorriendo en ambas el *array* con un bucle `for`, y recorriendo los bits con bucle `for` (1ª versión, p.30) o con un bucle `while` (2ª versión, p.29), aplicando ***en ambos*** máscara 0x1 y desplazamiento a la derecha (ambas con el cuerpo de la p.29), para ir extrayendo y acumulando los bits (ver [1]). Compararíamos los tiempos para comprobar que a veces se pueden obtener buenas ganancias pensando bien las cosas en C, sin necesidad de usar ASM.

```
#define WSIZE 8*sizeof(long)
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Figura 13: CS:APP [1], T-2.2, tr.30 cuerpo como tr.29

```
long pcount_while(unsigned long x)
{

  long result = 0;
  while (x)
  {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Figura 14: CS:APP [1], Tema 2.2, transparencia 29

Notar que la variable `result` vista en clase puede continuar usándose para seguir sumando los bits de otro elemento. Notar que preferimos llamar "1ª versión" a la del bucle `for` (tr.30), que previsiblemente tendrá peores prestaciones que el bucle `while` (incluso usando el mismo desplazamiento a derecha), porque debe iterar siempre `8*sizeof(int)` veces independientemente del nº de bits activados.

Realizar una **tercera** y **cuarta** versiones traduciendo el bucle interno `while` por un tramo de unas 4-5 ó 5-7 líneas ensamblador respectivamente (sin contar-contando etiquetas) que incluyan la instrucción ADC que ya utilizamos en la práctica anterior. La idea consiste en que como el bit desplazado acaba en el acarreo (consultar el manual de SHR), de ahí mismo lo podemos sumar y nos ahorramos aplicar la máscara. En principio, se pensaría que debe suponer alguna mejora sobre la 2ª versión. El resultado nos decepcionará. Por facilitar el desarrollo de este primer ejemplo, propuesto como ejercicio preparatorio, indicamos unas posibles restricciones:

```
    for (i=0; i<len; i++) {
      x = array[i];
      asm("\n"
"ini3:                \n\t"          // seguir mientras que x!=0
       "shr %[x]      \n\t"          // LSB en CF
       ...
       : [r]"+r" (result)            // e/salida: añadir a lo acumulado por el momento
       : [x] "r" (x)      );         // entrada:  valor elemento
    }
```

La **tercera** versión consta de otras 3 instrucciones ASM (aparte de la etiqueta e instrucción mostradas, 4-5 líneas en total), ya hemos mencionado que se usa ADC, y es obvio que si hay etiqueta `ini3:` es porque debe haber un salto hacia atrás. Ya hemos estudiado en clase cómo hace `gcc` para comparar con $0, y recomendamos encarecidamente recordar que no se usa CMP. *¿Por qué?* *(ver Tema 2.2 tr.7)*

La **cuarta** versión consta de 5 instrucciones ASM y 1 (ó 2) etiqueta(s) `ini4:` (y `fin4:`), (5-6-7 líneas en total), que se obtienen al intentar evitar la instrucción TEST aprovechando que SHR también afecta al flag ZF. En un primer intento se podría saltar condicionalmente al final tras SHR, y aprovechando que los saltos no modifican los flags (particularmente CF), acumular el acarreo y volver al principio. Eso produce un bucle de 4 instrucciones algo menos eficiente que el anterior. En un segundo intento, podría recordarse la instrucción CLC para comenzar el segmento ASM de la siguiente forma:

```
      asm("\n"
       "clc           \n\t"          // CLC para poder empezar por ADC
"ini4:                \n\t"
       "adc $0, %[r]  \n\t"          // ahora sale bucle de 3 instrucciones
       ...
      );
```

La sentencia ASM tiene las mismas restricciones que la tercera versión. Tanto el primer como el segundo intento necesitan una última instrucción ADC repetida para capturar el último bit. Aunque se obtenga un bucle con un 25% menos de instrucciones, comprobaremos que las prestaciones no mejoran proporcionalmente. De hecho, seguimos sin vencer a `gcc`.

Implementar como **quinta** versión la solución que aparece en el libro de clase [1] 2ª Ed., problema 3.49, resuelto en la página 364 (lenguaje C). Viene resuelto para 64bits (y un único elemento), bastaría con adaptarlo a 32bits (y a un *array* completo). Recordar que preferimos usar `size_t` para los índices.

```
long fun_c(unsigned long x) {
    long val = 0;
    int i;
    for (i = 0; i < 8; i++) {
        val += x & 0x0101010101010101L;
        x >>= 1;
    }
    val += (val >> 32);
    val += (val >> 16);
    val += (val >> 8);
    return val & 0xFF;
}
```

Figura 15: CS:APP2e [1], Problema 3.49, resuelto p.364

El código se basa en aplicar sucesivamente (8 veces) la máscara 0x0101… a cada elemento, para ir acumulando los bits de cada byte en una nueva variable `val` (no podemos acumular los bits de uno en uno en `result` como antes) y sumar en árbol los 4B. Seguramente uno no se espera una mejora tan drástica en las prestaciones, lo cual nos debe hacer reflexionar en lo importante que es escoger bien la idea básica de un algoritmo. Es muy difícil ganar a gcc cuando compila un programa C bien pensado. *¿Qué significa "acumular los bits de cada byte"? ¿Qué significa "sumar en árbol"? ¿Por qué hace falta la nueva variable* `val`*?*

Reflexionando sobre la quinta versión se llega a la solución "*naive*" propuesta por la Wikipedia [5], consistente en reemplazar el bucle lineal inicial `for(i<8)` por más sumas en árbol de granularidad más fina, empezando por bits sueltos. Nuestra **sexta** versión será esta propuesta "*naive*", adaptada de nuevo a *array* de enteros. *¿Qué significa "bucle lineal"? ¿Por qué funciona este algoritmo?*

```
//types and constants used in the functions below
//uint64_t is an unsigned 64-bit integer type (defined in C99 version of C language)
const uint64_t m1  = 0x5555555555555555; //binary: 0101...
const uint64_t m2  = 0x3333333333333333; //binary: 00110011..
const uint64_t m4  = 0x0f0f0f0f0f0f0f0f; //binary:  4 zeros,  4 ones ...
const uint64_t m8  = 0x00ff00ff00ff00ff; //binary:  8 zeros,  8 ones ...
const uint64_t m16 = 0x0000ffff0000ffff; //binary: 16 zeros, 16 ones ...
const uint64_t m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones

//This is a naive implementation, shown for comparison,
int popcount64a(uint64_t x)
{
    x = (x & m1 ) + ((x >>  1) & m1 ); //put count of each  2 bits into those  2 bits
    x = (x & m2 ) + ((x >>  2) & m2 ); //put count of each  4 bits into those  4 bits
    x = (x & m4 ) + ((x >>  4) & m4 ); //put count of each  8 bits into those  8 bits
    x = (x & m8 ) + ((x >>  8) & m8 ); //put count of each 16 bits into those 16 bits
    x = (x & m16) + ((x >> 16) & m16); //put count of each 32 bits into those 32 bits
    x = (x & m32) + ((x >> 32) & m32); //put count of each 64 bits into those 64 bits
    return x;
}
```

Figura 16: Wikipedia, popcount, C "*naive*" implementation [5]

Por un conocimiento de arquitectura que adquiriremos posteriormente (la versión 8 con múltiples instrucciones SSE3 gana a la versión 9 con una única instrucción SSE4) se nos ocurre pensar que también se puede mejorar esta versión leyendo en tamaños superiores (y desenrollando el bucle en un factor 4x). Nuestra **séptima** versión tiene por tanto el siguiente aspecto:

```
// Wikipedia popcount "naive" implementation *** 128b ***
int popcount7(unsigned* array, size_t len)
{
  size_t i;
  unsigned long x1,x2;
  int result=0;

  const unsigned long m1  = 0x5555555555555555; //binary: 0101...
  ...
  const unsigned long m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones

  if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");

  for (i=0; i<len; i+=4)
  {
    x1 = *(unsigned long*) &array[i  ];
    x2 = *(unsigned long*) &array[i+2];

    x1 = (x1 & m1 ) + ((x1 >>  1) & m1 ); //put count of each  2 bits into those  2 b
    ...
    x1 = (x1 & m32) + ((x1 >> 32) & m32); //put count of each 64 bits into those 64 b

    x2 = (x2 & m1 ) + ((x2 >>  1) & m1 );
    ...
    x2 = (x2 & m32) + ((x2 >> 32) & m32);

    result+= x1+x2;
  }
  return result;
}
```

Figura 17: aumento de tipo y desenrollado de bucle para mejorar prestaciones

Para una **octava** versión, podemos buscar con Google qué otros métodos han usado algunos entusiastas para calcular el *popcount*, e implementar alguno de ellos (ver [6], instrucción SSSE3 PSHUFB). Compararíamos con el crono de la versión anterior para ver cuánto se gana por pasar del repertorio normal a SSSE3. La Figura 18 y los párrafos tras ella se dedican a explicar el método [6].

Una **novena** versión consistiría en sustituir todo el bucle interno `while` por la instrucción SSE4 POPCNT. Compararíamos con el crono de la versión 8 para ver cuánto se gana por pasar del repertorio SSSE3 a SSE4. Como hemos comentado anteriormente, los resultados nos hacen sospechar de leer sólo 32 bits.

La **décima** y última versión intenta mejorar la anterior realizando dos lecturas de 64 bits (como en la versión 7) y dos *popcount*, realizando por tanto la cuarta parte de iteraciones. Todos los ejemplos con repertorio multimedia se ofrecen prácticamente resueltos.

```
// Versión SSE4.2 (popcount)
int popcount9(unsigned* array, size_t len){
    size_t i;
    unsigned x;
    int val, result=0;


    for (i=0; i<len; i++)
    {
      x = array[i];
      asm("popcnt %???, %???"


       : [val] "=r" (val)
       :   [x]  "r" (x)

      );
      result += val;
    }
    return result;
}
```

```
// popcount 128bit p/mejorar prestaciones
int popcount10(unsigned* array, size_t len){
    size_t i;
    unsigned long x1,x2;
    long val; int result=0;
    if (len & 0x3) printf(
"leyendo 128b pero len no múltiplo de 4\n");
    for (i=0; i<len; i+=4) {
      x1 = *(unsigned long*) &array[i  ];
      x2 = *(unsigned long*) &array[i+2];
      asm("popcnt %[x1], %???? \n\t"
          "popcnt %[x2], %???? \n\t"
          "add    %????, %[val]\n\t"
       : [val]"=&r" (val)
       :  [x1]  "r" (x1),
          [x2]  "r" (x2)
      );
      result += val;
    }
    return result;
}
```

```
// Versión SSSE3 (pshufb) web http:/wm.ite.pl/articles/sse-popcount.html
int popcount8(unsigned* array, size_t len){
    size_t i;
    int val, result=0;
    int SSE_mask[] = {0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f};
    int SSE_LUTb[] = {0x02010100, 0x03020201, 0x03020201, 0x04030302};
                     //    3 2 1 0     7 6 5 4    1110 9 8    15141312
    if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");
    for (i=0; i<len; i+=4)    {
    asm("movdqu    %[x], %%????\n\t"
        "movdqa  %%xmm0, %%???? \n\t"  //    x: two copies xmm0-1
        "movdqu    %[m], %%???? \n\t"  // mask: xmm6
        "psrlw      $4 , %%???? \n\t"
        "pand    %%xmm6, %%xmm0 \n\t"  //; xmm0 - lower  nibbles
        "pand    %%xmm6, %%xmm1 \n\t"  //; xmm1 - higher nibbles

        "movdqu    %[l], %%???? \n\t"  //; since instruction pshufb modifies LUT
        "movdqa  %%xmm2, %%???? \n\t"  //; we need 2 copies
        "pshufb  %%xmm0, %%xmm2 \n\t"  //; xmm2 = vector of popcount lower nibbles
        "pshufb  %%xmm1, %%xmm3 \n\t"  //; xmm3 = vector of popcount upper nibbles

        "paddb   %%????, %%xmm3 \n\t"  //; xmm3 - vector of popcount for bytes
        "pxor    %%????, %%xmm0 \n\t"  //; xmm0 =  0,0,0,0
        "psadbw  %%????, %%xmm3 \n\t"  //; xmm3 = [pcnt bytes0..7|pcnt bytes8..15]
        "movhlps %%????, %%xmm0 \n\t"  //; xmm0 = [       0      |pcnt bytes0..7 ]
        "paddd   %%????, %%xmm0 \n\t"  //; xmm0 = [  not needed  |pcnt bytes0..15]
        "movd    %%xmm0, %[val]     "
        : [val]"=r" (val)
        :  [x]  "m" (array[i]),
           [m]  "m" (SSE_mask[0]),
           [l]  "m" (SSE_LUTb[0])
      );
         result +=  val;
    }
    return result;
}
```

Figura 18: popcount8: función para cálculo SSSE3 del peso Hamming (algunos registros XMM omitidos)

Para comprender el método SSSE3 propuesto en la web [6] conviene consultar el dibujo que acompaña a la página de manual de PSHUFB, la operación de ***baraje*** más corta del repertorio SSSE3. Los registros XMM (XMM0-XMM7) son de 128bits, y están pensados para almacenar en paralelo varios elementos, por ejemplo 4 enteros de 32bits (4ints x 2^5 bits/int = 2^7 bits = 128bits), 8 shorts, o 16 chars (2^4 x 2^3). La operación de ***baraje*** permite "barajar" esos elementos (como si fueran cartas de una baraja), indicando en un primer argumento el baraje deseado (en cada posición se indica el nº del dato deseado en esa posición) y en un segundo argumento los datos a barajar. Es *fundamental* advertir que en el baraje no se indica a qué posición va cada elemento (podríamos equivocarnos y dejar huecos), sino qué elemento termina en esa posición. Por fijar conceptos, la instrucción `pshufb %xmm1, %xmm2` baraja los 16 bytes de XMM2, colocando el byte i (i=0..15) en todos los bytes de XMM1 donde ponga i. Esto nos permite repetir elementos y que otros se queden fuera del resultado, lo cual puede parecer anti-intuitivo y poco relacionado con barajas de cartas. Notar también que los datos de baraje (XMM2) son sobrescritos. Conviene re-leer este párrafo junto con el dibujo del manual hasta comprender la operación de baraje.

La idea para acelerar el cálculo del *popcount* consiste en pre-calcular cuántos bits tiene activados cada número (hasta un límite dado, por ejemplo de 8 bits: 0 tiene 0bits, 1 y 2 tienen 1bit, 3 tiene 2bits… hasta 255, que tiene 8 bits activados), y usar el propio número como índice en una tabla (más o menos grande según el límite impuesto) en donde se almacenan esos resultados pre-calculados. El *popcount* de un elemento `x=array[i]` (supongamos `x`=255) es entonces `Tabla[x]` (=8). A este tipo de tabla, donde el dato disponible indexa el resultado deseado, se les suele llamar *Tabla de Consulta* (*Look-Up Table, LUT*). Por ejemplo, una paleta de colores indexados es una LUT, porque el código del color se usará como índice.

Siguiendo con el ejemplo `Tabla[array[i]]`, se tarda menos en acceder al elemento 255 de la tabla (obteniendo resultado=8bits) que hacer 8 desplazamientos, máscaras y acumulaciones. El inconveniente es que una tabla tan grande no cabe en un registro XMM. Pero si la limitamos a elementos de 4bits (medio byte, un *nibble*), sí que podemos almacenarla en un registro XMM, en donde caben 16B. De hecho nos sobra más de la mitad de cada byte, porque la LUT sólo necesita 16 elementos de 3bits: 16 porque calcularemos *popcount* de 4bits, y 3 porque el máximo son 4bits activados (0b100). Pero en SSSE3 no existe operación de baraje con 32 nibbles. La operación de baraje más corta opera sobre 16B, y nosotros aprovecharemos sólo la mitad de cada byte.

Se puede recorrer por tanto el *array* de 4 en 4 elementos, cargando 4 enteros en un registro XMM de 128bits (16B), repartiendo sus nibbles entre dos registros XMM (para que todos los índices salgan entre 0..15), barajando con la tabla pre-calculada (LUT) para obtener cuántos bits hay activados en cada nibble, y sumando todos esos números. La máscara y tabla se pueden consultar en la Figura 18.

Explicado paso a paso, el tramo ASM carga 4 enteros en un registro XMM y saca copia en otro XMM. Carga una máscara para quedarse con nibbles inferiores. Desplaza 4b una de las copias, de manera que al aplicar la máscara a ambas copias, resulten separados los nibbles inferiores y superiores en su correspondiente registro XMM. Se cargan entonces dos copias de la LUT y se barajan usando como índices los nibbles, obteniendo los popcount respectivos, como se explicó anteriormente.

El último tramo sirve para acumular todos esos popcount en `val`. PADDB es una suma de bytes, que se usa para reunir las cuentas de nibbles inferiores y superiores. Sumar horizontalmente esas cuentas es más complicado, debiéndose usar PSADBW (instrucción pensada para sumar valores absolutos de diferencias), que produce 2 resultados de 16b, uno en la parte menos significativa y otro en el centro del registro XMM. MOVHLPS sirve para llevar el resultado central a la parte inferior de otro registro XMM, y PADDD sirve para sumar ambos. El resultado final se puede mover a un registro de 32b con MOVD.

Notar que casi todas las restricciones se han indicado en memoria, encargándonos nosotros del movimiento explícito a registros (con MOVDQU, para evitar problemas si los *arrays* resultan no estar alineados a 16B). De esta forma el tramo ASM es virtualmente idéntico al de la web [6]. Se puede usar MOVDQA para mover entre registros XMM. Notar por último que el *array* se recorre de 4 en 4 elementos, y que dicho recorrido y la acumulación son las únicas tareas que se realizan en lenguaje C. Sólo la restricción para `val` se ha indicado en registro de 32b, para optimizar su suma con `result`. El movimiento de los 32b inferiores de un registro XMM a uno de 32b se puede realizar con MOVD.

Mediciones: cronometrar las distintas versiones con –O0, –O1 y –O2

Como también nos interesa saber cómo se comporta gcc según el nivel de optimización (`-O0`, `-Og`, `-O1`, `-O2`), repetiremos 10 mediciones de tiempo para esos 4 niveles. Se trata por tanto de recompilar 4 veces, repetir 10 mediciones, y organizar los resultados en una gráfica de paquete ofimático (Calc), mostrando los promedios de cada versión (1ª - 10ª) para cada nivel de optimización (0 - 2), tal vez con un gráfico de barras con abscisas bidimensionales versión-optimización (Calc lo denomina *"columnas 3D"*). Conviene que cambie el color de las columnas con la versión de la función, no con la optimización.

En principio se esperaría que cada versión fuera progresivamente mejor, y dentro de cada una, se mejorara con el nivel de optimización. Si alguna versión no siguiera esta tendencia, convendría probar con otro modelo de CPU para ver si es una característica propia del modelo usado. Si varios modelos reprodujeran la anomalía, sería conveniente cambiar de orden las mediciones o realizar otras modificaciones en el código al objeto de descubrir si somos nosotros quienes estamos creando la anomalía. Si no se descubre ningún motivo, se debería consultar el código ensamblador generado para intentar explicar dicho comportamiento.

Recomendaciones

Recordar que siempre se debe comprobar que el resultado es correcto. Una optimización que produce un resultado distinto sólo tiene tres explicaciones: o está mal el programa optimizado, o está mal el original, o están mal ambos.

El esquema de compilación condicional que sugerimos en la Figura 12 nos facilita tanto la comprobación de los ejemplos pequeños definidos al principio, como la realización de las mediciones y su incorporación a una hoja Calc. Considerar los siguientes comandos *shell bash* añadidos al principio del código fuente:

```
// gcc popcount.c -o popcount -Og -g -D TEST=1
/*
  === TESTS === ____________________________________
  for i in 0 g 1 2; do
      printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
    for j in $(seq 1 4); do
      printf "__TEST%02d__%48s\n" $j "" | tr " " "-"
      rm  popcount
      gcc popcount.c -o popcount -O$i -D TEST=$j -g
        ./popcount
    done
  done
  === CRONOS === ___________________________________
  for i in 0 g 1 2; do
      printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
      rm  popcount
      gcc popcount.c -o popcount -O$i -D TEST=0
    for j in $(seq 0 10); do
      echo $j; ./popcount
    done | pr -11 -l 22 -w 80
  done
  ________________________________________________
 */

#include <stdio.h>            // para printf()
#include <stdlib.h>           // para exit()
#include <sys/time.h>         // para gettimeofday(), struct timeval

int resultado=0;

#ifndef TEST
#define TEST 5
#endif

...
```

Figura 19: popcount.c: autodocumentando el sistema de test y medición

El comando que aparece como primer comentario sería útil para depurar alguna versión que produjera resultado incorrecto con el primer ejemplo (TEST=1). Pudiera incluso suceder que alguna versión sólo produjera resultados erróneos en algún nivel de optimización concreto. Se debería entonces recompilar con ese nivel de optimización (e información de depuración `-g`), comprobar que el error sigue manifestándose, y proceder a depurar con `gdb -tui`.

El segundo comando (etiquetado TESTS en el listado de la Figura 19) barre con dos bucles anidados los cuatro niveles de optimización y los cuatro ejemplos de test, ejecutando para cada combinación el programa recompilado con esas opciones. Haber cuidado la alineación de los resultados al redactar el código fuente nos permite ahora revisar a una formidable velocidad la corrección de todas las versiones de *popcount*, con los diversos tests que habíamos acordado, para los distintos niveles de optimización.

Considerar ahora el siguiente commando *shell bash* para un ejecutable recompilado con TEST=0.

```
for i in $(seq 0 10); do echo $i ; ./popcount; done | pr -11 -l 22 -w 80
```

Consultando la página de manual de `pr`, se deduce que este comando permite realizar las deseadas 10 mediciones y dejarlas escritas en pantalla en un formato listo para hacer *copy-paste* fácilmente a la hoja de cálculo mostrada en el Apéndice 1. En realidad ese comando lanza 11 ejecuciones, por si la primera (o alguna) sale claramente peor que el resto, y pagina los 110 números (10 versiones x 11 mediciones) en las mismas 11 columnas de la hoja Calc. Se han escogido 22 líneas de 80 caracteres como tamaño de página. El comando etiquetado CRONOS en el listado de la Figura 19 permite realizar todas las mediciones a la misma formidable velocidad.

Los comandos para recompilar y lanzar la medición se podrían anotar también en la propia hoja Calc como recordatorio para cuando se desee repetir el experimento.

5 Entrega del trabajo desarrollado

Los distintos profesores de teoría y prácticas acordarán las normas de entrega para cada grupo, incluyendo qué se ha de entregar, cómo, dónde y cuándo.

Por ejemplo, puede que en un grupo el profesor habilite que los estudiantes que cumplan el régimen de asistencia puedan entregar en SWAD, hasta medianoche del domingo de la última semana de esta práctica, el código fuente realizado (`popcount.c`), listo para compilar y reproducir las mediciones usando el comando CRONOS incluido en el propio fuente con la técnica mostrada más arriba, y otro fichero (`popcount.pdf`) con las gráficas de las mediciones de tiempo, con el formato indicado en este guión de prácticas. Puede que en otro grupo el profesor de prácticas proporcione otras instrucciones distintas.

En cualquier caso, **el examen de prácticas** a final del curso (Test TP de 4 puntos) **es el mismo para todos los grupos**, lo cual significa que independientemente del profesor de prácticas, cada estudiante es responsable de comprender todo lo explicado en estos **guiones de prácticas (que son comunes a todos los grupos y son materia de examen)** y desarrollar las habilidades que se ejercitan en los mismos (que también son materia de examen).

6 Bibliografía

[1] Apuntes y presentaciones de clase, y particularmente
Programación Máquina II: Control
sección “Códigos de condición”, instrucciones `test/setcc`, p.7-11
sección “Bucles”, p.22-33
Libro CS:APP 2ª Ed., Problema 3.49, p.364.
Randal E. Bryant, David R. O’Hallaron: “Computer Systems: A Programmer’s Perspective”, 2nd-3rd Ed., Pearson, 2011-2016. http://csapp.cs.cmu.edu/

[2] Manuales de Intel sobre IA-32 e Intel64, en concreto el volumen 2: “Instruction Set Reference”
https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf

[3] Wikipedia, convenciones de llamada http://en.wikipedia.org/wiki/Calling_convention
X86 calling conventions http://en.wikipedia.org/wiki/X86_calling_conventions
WikiBook http://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions
[4] Wikipedia, extensiones x86 MMX/SSE: http://en.wikipedia.org/wiki/X86#Extensions
MMX http://en.wikipedia.org/wiki/X86#MMX
SSE, SSE2, SSE3, SSSE3, SSE4 http://en.wikipedia.org/wiki/X86#SSE
[5] Wikipedia, popcount, C naive implem. https://en.wikipedia.org/wiki/Hamming_weight#Efficient_implementation
[6] Código SSSE3 para fast popcount http://0x80.pl/articles/sse-popcount.html
Copia rescatada de http://web.archive.org/web/20100701222327/http://wm.ite.pl/articles/sse-popcount.html
[7] GAS manual http://sourceware.org/binutils/docs/as/index.html
9.13: 80386 depend.features http://sourceware.org/binutils/docs/as/i386_002dDependent.html
[8] GCC manual v.7.3 (la del laboratorio) https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/
6: Extensions to C Language https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/#toc-Extensions-to-the-C-Language-Family
6.32.1: Common Variable attributes https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Common-Variable-Attributes.html
6.45 How to Use Inline Assembly https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Using-Assembly-Language-with-C.html
6.45.2 Extended Asm https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Extended-Asm.html
6.45.3 Constraints https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Constraints.html
[9] GCC Inline Assembly HOWTO http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
6: More about constraints http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s6
[10] Linux Assembly HOWTO http://tldp.org/HOWTO/Assembly-HOWTO/index.html
3.1: GCC inline assembly http://tldp.org/HOWTO/Assembly-HOWTO/gcc.html
Brennan's Guide to inline asm http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html
5.1: Linux calling conventions http://tldp.org/HOWTO/Assembly-HOWTO/linux.html
[11] Sourceforge tutorials http://asm.sourceforge.net/resources.html#tutorials
Using asm in Linux http://asm.sourceforge.net/articles/linasm.html#InlineASM
Inline asm x86 – IBM http://www.ibm.com/developerworks/linux/library/l-ia
Otra Brennan's – SETI@Home http://setiathome.ssl.berkeley.edu/~korpela/djgpp_asm.html
Miyagi's intro – texto http://asm.sourceforge.net/articles/rmiyagi-inline-asm.txt

Apéndice 1. Ejemplo de gráfica

A continuación se muestra un ejemplo del tipo de hoja de cálculo deseada, en donde se anota el modelo de CPU y el comando usado para recompilar el programa, lanzar las mediciones y preparar los números para poder hacer *copy-paste* fácilmente. En la parte dedicada a anotar las mediciones, la media se calcula sobre las columnas 1-10, ignorando la columna 0 como se explica en el texto recordatorio. La medición 0 suele salir claramente mal. Si fuera otra la que sale mal, se podría intercambiar con la 0.

lscpu:	CPU(s): 4 Nombre del modelo: Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz Virtualización: VT-x Caché L3: 6144K

POPCOUNT:

```
for l in 0 g 1 2; do
  printf "__OPTIM%1c__%48s\n" $l "" | tr " " "="
  rm popcount
  gcc popcount.c -o popcount -O$l -D TEST=0
 for j in $(seq 0 10); do
  echo $j; ./popcount
 done | pr -11 -l 22 -w 80
done
```

Ignorar medición 0, repetir columna si alguna medición se sale demasiado de la media

Zona para reprodu
recordar que se ig

Optimización -O0	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	88237	74662	81400	88886	82354	88627	86607	83895	85103	88977	85817	84633
popcount2 (lenguaje C - while):	41198	40171	39885	39797	39798	39754	39746	39849	39684	39715	39824	39822
popcount3 (leng.ASM-body while 4i):	12618	12458	12462	12442	12459	12409	12473	12429	12422	12424	12465	12444
popcount4 (leng.ASM-body while 3i):	11165	11039	10986	10989	10973	10984	11051	10967	10966	10984	11010	10995
popcount5 (CS:APP2e 3.49-group 8b):	20001	19986	19993	19963	19880	19881	19941	19892	19930	19863	19920	19925
popcount6 (Wikipedia- naive - 32b):	9050	9072	9053	9042	9027	9032	9025	9015	9112	9030	9052	9046
popcount7 (Wikipedia- naive -128b):	4706	4710	4719	4702	4712	4676	4704	4729	4707	4695	4706	4706
popcount8 (asm SSE3 - pshufb 128b):	830	809	809	834	810	811	835	810	810	814	832	817
popcount9 (asm SSE4- popcount 32b):	2733	2760	2772	2730	2733	2730	2774	2733	2763	2753	2773	2761
popcount10(asm SSE4- popcount128b):	842	869	868	844	844	844	844	843	847	846	845	849

media	0	1
87532	88724	82391
39966	40976	39744
12477	12781	12648
10992	11050	11024
19898	19987	19966
9044	9075	9082
4710	4703	4725
810	827	811
2746	2732	2756
855	845	844

Optimización -Og	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	21631	21455	21008	21000	20995	21095	21022	20923	20923	20945	21030	21040
popcount2 (lenguaje C - while):	8015	8379	7995	8064	8010	8584	8008	8002	7986	7988	7983	8097
popcount3 (leng.ASM-body while 4i):	12472	12745	12486	12516	12494	12493	12516	12526	12459	12510	12482	12520
popcount4 (leng.ASM-body while 3i):	9610	9781	9558	9664	9621	9605	9565	9572	9974	9621	9594	9656
popcount5 (CS:APP2e 3.49-group 8b):	6219	6219	6232	6227	6214	6248	6223	6219	6241	6227	6228	6228
popcount6 (Wikipedia- naive - 32b):	2641	2650	2617	2688	2640	2618	2645	2625	2617	2661	2638	2640
popcount7 (Wikipedia- naive -128b):	1547	1549	1573	1550	1549	1572	1551	1573	1586	1553	1576	1563
popcount8 (asm SSE3 - pshufb 128b):	395	392	391	437	391	393	417	391	392	393	392	399
popcount9 (asm SSE4- popcount 32b):	466	502	463	469	462	467	466	465	464	514	465	474
popcount10(asm SSE4- popcount128b):	313	313	312	314	312	313	313	313	313	315	313	313

media	0	1
21000	21214	21267
8146	7990	8384
12525	12507	12807
9618	9639	9719
6245	6254	6397
2638	2615	2651
1557	1582	1548
396	390	394
465	467	467
313	313	315

Optimización -O1	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	13697	14204	13640	13634	13667	13719	13694	13719	13675	13618	13642	13721
popcount2 (lenguaje C - while):	12531	13102	12555	12523	12545	12540	12531	12544	12623	12553	12550	12607
popcount3 (leng.ASM-body while 4i):	10111	10370	10032	10150	10090	10074	10151	10165	10221	10103	9930	10129
popcount4 (leng.ASM-body while 3i):	9649	9716	9631	9640	9584	9622	9642	9579	9603	9559	9548	9612
popcount5 (CS:APP2e 3.49-group 8b):	4571	5148	5199	4663	4357	4815	4726	4630	4481	5513	4487	4802
popcount6 (Wikipedia- naive - 32b):	2650	2658	2631	2633	2636	2638	2656	2615	2616	2633	2615	2633
popcount7 (Wikipedia- naive -128b):	1418	1419	1438	1418	1417	1419	1419	1442	1600	1461	1443	1448
popcount8 (asm SSE3 - pshufb 128b):	392	428	416	391	390	393	393	391	450	394	391	404
popcount9 (asm SSE4- popcount 32b):	452	455	453	453	451	473	473	451	461	454	452	458
popcount10(asm SSE4- popcount128b):	314	311	312	313	312	313	326	311	319	314	312	314

media	0	1
13671	14236	13651
12552	12763	12549
10159	10344	10216
9656	9604	10077
4913	5184	4371
2632	2614	2654
1456	1465	1637
406	390	514
458	453	462
317	312	323

Optimización -O2	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	18567	18047	17847	18173	17391	18290	17996	18097	18314	17820	18076	18006
popcount2 (lenguaje C - while):	8245	8545	8439	8253	8589	8577	8576	8601	8531	8564	8368	8504
popcount3 (leng.ASM-body while 4i):	10268	10395	10252	10252	10288	10263	10264	10316	10238	10315	10215	10280
popcount4 (leng.ASM-body while 3i):	9543	9835	9554	9587	9544	9651	9551	11301	9572	11709	9631	9994
popcount5 (CS:APP2e 3.49-group 8b):	4622	4937	4443	4544	4486	4475	4692	4760	4617	4875	4707	4654
popcount6 (Wikipedia- naive - 32b):	2340	2345	2327	2305	2325	2303	2303	2346	2302	2327	2348	2323
popcount7 (Wikipedia- naive -128b):	1330	1320	1315	1360	1317	1338	1360	1316	1315	1314	1315	1327
popcount8 (asm SSE3 - pshufb 128b):	392	395	392	393	391	389	391	390	414	390	391	394
popcount9 (asm SSE4- popcount 32b):	462	461	475	455	454	454	453	455	455	455	456	457
popcount10(asm SSE4- popcount128b):	313	317	311	312	312	312	312	311	330	350	312	318

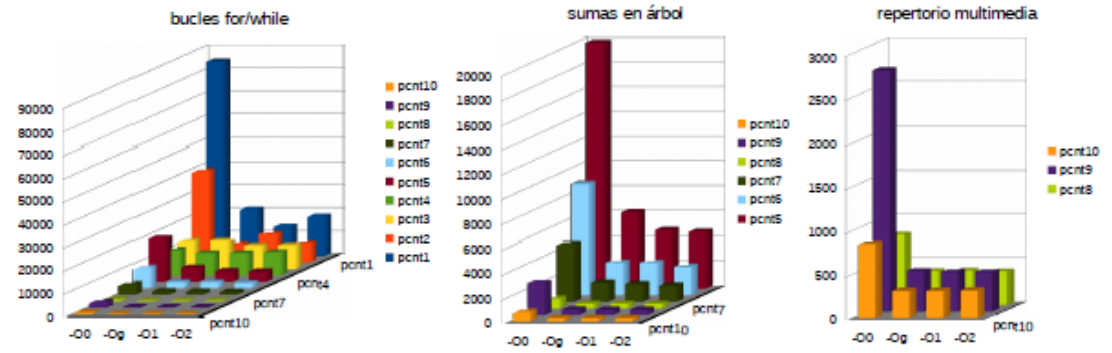
media	0	1
17748	18399	18186
8464	8044	8517
10320	10223	10761
9838	9581	9816
4790	4711	4850
2319	2327	2304
1335	1316	1318
395	393	431
461	457	458
312	336	318

POPCOUNT:	-O0	-Og	-O1	-O2
pcnt1	84633	21040	13721	18005
pcnt2	39822	8097	12607	8504
pcnt3	12444	12520	10129	10280
pcnt4	10995	9656	9612	9994
pcnt5	19925	6228	4802	4654
pcnt6	9046	2640	2633	2323
pcnt7	4706	1563	1448	1327
pcnt8	817	399	404	394
pcnt9	2751	474	458	457
pcnt10	849	313	314	318

GainInEff	-O0	-Og	-O1	-O2
pcnt1			1.00	
pcnt2		1.69		
pcnt3			1.35	
pcnt4			1.43	
pcnt5				2.95
pcnt6				5.91
pcnt7				10.34
pcnt8				34.86
pcnt9				30.00
pcnt10		43.82	43.66	43.16

Comentario

comparado con el for más rápido
el while es un 70% más rápido
ASM se queda en un 35%
o en un 43%
sumar en grupos 8b sale 3x más rápido
sumar en árbol 6x
lectura 128b sube a 10x
SSSE3 sube a 35x más rápido
SSE4 sólo 30x por leer 32b
SSE4 128b sube a 44x



Notar que las tablas grises son otra medición completa, por si acaso varios números de una fila estuvieran claramente mal y necesitáramos sustituirlos por otros correctos. Si sólo uno estuviera mal, se podría intercambiar con la columna 0. La media se hace de las mediciones 1-10, excluyendo la medición 0, que estadísticamente suele salir peor que el resto.

Más habitualmente, repetiremos dos veces el comando, anotando unos resultados a la izquierda y otros a la derecha (tablas grises). Las gráficas se calculan a partir de la información a la izquierda, y se ha procurado que las medias de ambas partes queden enfrentadas para poderlas comparar sin esfuerzo. Si salen medias muy parecidas, ambas mediciones está bien hechas y ni siquiera nos molestaremos en comprobar si alguna medición está subiendo demasiado la media y querríamos sustituirla por la medición 0, o por alguna de la parte derecha.

Notar que ha sido necesario copiar y etiquetar las medias (para que queden adyacentes, ver tabla roja) de forma que sea fácil generar la gráfica a partir de la tabla roja. El asistente de Calc para generar gráficas necesita que los datos estén de partida dispuestos bidimensionalmente, y puede aprovechar las cabeceras para etiquetar los ejes de la gráfica y la leyenda adecuadamente.

Para que los gráficos de columnas 3D resulten intuitivos, se ha procurado que cambie el color de columna con la versión, y se mantenga para los distintos niveles de optimización. También se ha procurado que al fondo aparezcan las mediciones más lentas, para que no tapen a las más rápidas. Otro detalle para facilitar la comprensión consiste en añadir un texto que describa aproximadamente lo que se hacía en cada versión, de manera que se pueda recordar de qué versión estamos hablando. El texto descriptivo no se incluye a la hora de etiquetar los ejes.

Se añaden otras gráficas de detalle (*zoom*), copias de la primera, eliminando las versiones anteriores para apreciar mejor las diferencias entre distintas versiones posteriores. La mejora combinada es tan grande que las últimas versiones parecen tardar todas 0 cuando se visualizan a la escala de la primera versión.

Notar que se resalta el modelo de CPU usado, y si se estaba usando virtualización. En un equipo de sobremesa como los disponibles en el L-2.9 (como el usado para esta gráfica) las mediciones saldrán seguramente bien a la primera. Se tarda más en hacer *copy-paste* que en realizar la medición propiamente dicha. En un equipo portátil seguramente se tendrán activadas en la BIOS diversas opciones de protección termal y ahorro de energía que justamente reducen la velocidad del reloj cuando el procesador empieza a trabajar fuerte y calentarse, impidiendo obtener mediciones fiables. En un portátil con Windows ejecutando Linux en una máquina virtual, las mediciones son adicionalmente impedidas. Aun así, es posible que cualitativamente se obtengan resultados sensatos.

En el laboratorio L-2.9, como muestra la gráfica, se han obtenido ganancias (comparando con el mejor `for`) de alrededor de 1.7 para el bucle `while`, alrededor de 3, 6 y 10 para algoritmos de suma en árbol (no se obtuvo ganancia intentando hacer el `while` mejor que `gcc`), y alrededor de 35, 30 y 44 para el repertorio SSSE3 y SSE4.

Recordar siempre que no se debe medir la velocidad de un programa sin haber comprobado previamente que produce el resultado correcto. Si se siguió la recomendación de autodocumentar el método de test y cronometraje (Figura 19), la comprobación se puede realizar en segundos.