Práctica de optimización para asignaturas de Estructura de los Computadores

M. Anguita, F. J. Fernández, A. F. Díaz, A. Cañas, A. Prieto
Dpto. de Arquitectura y Tecnología de Computadores
Universidad de Granada
18071 Granada
e-mail: manguita@atc.ugr.es

Resumen

Con el objetivo de motivar a los estudiantes en el estudio de materias de Arquitectura y Tecnología de Computadores, estamos proponiendo prácticas en las que los alumnos comprueban que pueden mejorar prestaciones en sus programas aplicando los conocimientos que adquieren en estas materias. Aquí presentamos una práctica propuesta para asignaturas de estructura. En estas asignaturas, se estudia la arquitectura *abstracta* del procesador (repertorio de instrucciones, conjunto de registros, tipos de datos, modos de direccionamiento, lenguaje máquina, ensamblador). En la práctica propuesta, los estudiantes, utilizando ensamblador dentro de código de alto nivel, obtienen mejoras de prestaciones en cuatro ejemplos: cálculo del mínimo, copia, inicialización y búsqueda. Los estudiantes comprueban, que utilizando conocimientos sobre arquitectura abstracta, pueden reducir el tiempo de ejecución de estos ejemplos en un 25%, un 50%, o en más de un 75%. La reducción depende del ejemplo, del tamaño de las entradas, y de la arquitectura *concreta* del procesador en el que se ejecute el código.

1. Introducción

La *motivación* es uno de los aspectos que más influye en el proceso de aprendizaje, de hecho el rendimiento y el aprendizaje de un alumno motivado se aproxima al máximo de sus posibilidades. Para favorecer la motivación es esencial proporcionar un *sentido de utilidad* a los contenidos que se imparten. Los estudiantes que prevén dedicar su vida laboral a la programación de aplicaciones, a menudo, no se enfrentan al estudio de las asignaturas de arquitectura y de estructura de computadores suficientemente motivados, lo que les puede llevar a fracasar en estas materias. Con objeto de incrementar su motivación, podríamos demostrarles, que pueden mejorar las prestaciones de sus programas, aplicando los conocimientos que adquieren en materias de arquitectura y estructura de computadores. Para ser más convincentes, deberían obtener las mejoras ellos mismos, o se les debería mostrar resultados que estimen que pueden obtener ellos mismos, como por ejemplo resultados logrados por otros estudiantes de la propia titulación. Otro objetivo que perseguimos, es que los estudiantes tomen conciencia de que pueden usar estos conocimientos como argumentos a su favor a la hora de *competir por un trabajo* de programador, con estudiantes de otras titulaciones superiores o estudios medios, en las que no se abarca el hardware de los computadores, o no se profundiza en su estudio en la misma medida en la que se hace en las titulaciones de informática. Las empresas de software que aprovechen las características de las arquitecturas disponibles, se verán beneficiadas dentro de un mercado competitivo.

Para alcanzar estos objetivos, por una parte, hemos propuesto trabajos fin de carrera, en los que los estudiantes comprueban que pueden obtener una buena relación entre las prestaciones y el coste de desarrollo en sus programas, aplicando los conocimientos sobre arquitectura y estructura de computadores que han adquirido en la titulación [8]. Los resultados obtenidos en estos proyectos, se muestran a los estudiantes de las asignaturas de arquitectura y de estructura.

Por otra parte, se han introducido prácticas en asignaturas tanto de arquitectura como de estructura, en las que los estudiantes comprueban

por si mismos, las mejoras en prestaciones que pueden conseguir aplicando contenidos que están estudiando (como en [3]). En asignaturas de la materia troncal Tecnología y Estructura de Computadores (Real Decreto 1459/1990 del 26 de octubre, sobre *directrices generales propias* comunes de los planes de estudio Ingeniero en Informática BOE 20/11/1990), se introdujo una práctica sobre mejora de prestaciones en el curso 2002-2003, concretamente en la asignatura Estructura de los Computadores de 2º de Ingeniero Técnico en Informática de Sistemas. Aquí se presenta la práctica desarrollada en el curso 2003-2004, que amplía la práctica del curso anterior. Un guión similar se ha estado utilizando como práctica introductoria para la asignatura Arquitecturas Especializadas desde el inicio de esta asignatura (curso 1998-1999) hasta 2002.

Para mejorar prestaciones en una aplicación, se pueden utilizar tanto los conocimientos adquiridos sobre la *arquitectura abstracta* del procesador (repertorio de instrucciones, conjunto de registros, tipos de datos, modos de direccionamiento, lenguaje máquina, ensamblador), como los adquiridos sobre la *arquitectura concreta* del procesador (segmentada, superescalar, VLIW, SIMD, multihebra simultánea o hyperthreading) y de los sistemas de cómputo (multiprocesadores, multicomputadores, cluster). La arquitectura abstracta (incluido programación en ensamblador) es objeto de estudio de asignaturas de la materia troncal Tecnología y Estructura de Computadores, donde también se introduce la arquitectura concreta. Por otra parte, en las asignaturas de la materia troncal Arquitectura e Ingeniería de Computadores, se estudian en detalle la arquitectura concreta de los procesadores y de los computadores actuales, que utilizan paralelismo en diferentes niveles para incrementar prestaciones. En las titulaciones de informática de Granada (Ingeniero Superior, Ingeniero Técnico de Sistemas e Ingeniero Técnico de Gestión), la arquitectura abstracta se incluye dentro de la asignatura Estructura de los Computadores.

Dado este reparto de contenidos entre estructura y arquitectura, las prácticas de mejora de prestaciones en las asignaturas de arquitectura, muestran al estudiante cómo incrementar prestaciones teniendo en cuenta la arquitectura concreta actual de los procesadores y computadores. Por otro lado, con la práctica para estructura que aquí se presenta, se pretende que los estudiantes comprueben la utilidad de usar en sus programas ensamblador. Al programar en ensamblador utilizan sus conocimientos sobre la arquitectura abstracta: repertorios, tipos de registros, tipos de datos, modos de direccionamiento, e incluso en algunos casos lenguaje máquina. Los estudiantes comprueban, con ejemplos muy sencillos, que resulta útil utilizar ensamblador para mejorar *tiempo de ejecución* o también para obtener una *funcionalidad* que de otra forma no obtendrían. Resultados para ejemplos más complejos (aplicaciones completas) se muestran al comienzo del curso a través de los trabajos fin de carrera, como comentamos más arriba.

En la Sección 2 se describe la práctica de estructura propuesta. La Sección 3 presenta resultados que los estudiantes obtienen durante el desarrollo de la práctica. Por último la Sección 4 presenta algunas conclusiones.

2. Descripción de la práctica de mejora de prestaciones

El guión de la práctica se puede descargar de la dirección http://atc.ugr.es/~mancia/ec/prac3.pdf. En la Tabla 1 se puede ver el índice. Esta práctica tiene como objetivo adicional, que los estudiantes aprendan el ensamblador usado tradicionalmente en Linux y la interfaz entre ensamblador y gcc. En prácticas anteriores los estudiantes han utilizado la notación de Intel (usada tradicionalmente bajo DOS y Windows) y han visto cómo manejar ensamblador en línea en compiladores C para dos o Windows. Por este motivo, el apartado 3 del guión, muestra las peculiaridades del ensamblador de Linux.

El apartado 2 del guión presenta al estudiante las herramientas que va ha utilizar para desarrollar la práctica. Al buscar y elegir un compilador para gcc y un entorno de desarrollo, se han tenido en cuenta varios criterios: (1) Gratuidad de las herramientas. (2) Compilador actualizado. (3) Entorno de desarrollo que permita ejecutar paso a paso las instrucciones en ensamblador, además de las instrucciones en alto nivel. (4) Entorno de desarrollo similar o igual al entorno usado en prácticas anteriores, con el fin de no perder tiempo en el aprendizaje de un nuevo entorno. Teniendo en cuenta los criterios enumerados, se eligió el

compilador **djgpp**. Este compilador es una versión de gcc para DOS que se mantiene actualizado a las últimas versiones de gcc, en la dirección web http://www.delorie.com/. El entorno desarrollo **rhide**, que acompaña a djgpp, es similar al entorno de desarrollo de Borland C que los estudiantes utilizan en prácticas anteriores, por lo que no tienen que acostumbrarse a un nuevo entorno.

Tabla 1. Índice de la práctica.

1. Resumen de objetivos.
2. Herramientas
3. Ensamblador de GNU.
Diferencias entre el ensamblador GNU y el ensamblador de Intel.
Ensamblador en línea en gcc o g++.
4. Ejemplos de programas con ensamblador en línea.
5. Utilidad del ensamblador en línea.
6. Trabajo a desarrollar
Acelerar el cálculo del mínimo.
Instrucciones de manejo de cadenas para mejorar prestaciones.
7. Referencias.

El apartado 4 del guión presenta ejemplos de programas gcc con ensamblador en línea. Para que el estudiante se habitúe a la interfaz entre ensamblador y gcc, se presentan ejemplos de interfaces incorrectas y sus versiones correctas. Precisamente estos ejemplos muestran errores habituales que se cometen cuando se usa por primera vez la interfaz con gcc.

Dentro de los contenidos de estructura, como se mencionó más arriba, se encuentran el estudio de la arquitectura abstracta de los procesadores actuales, incluido su lenguaje máquina y su lenguaje ensamblador. Como apoyo a estos contenidos, se suele estudiar el repertorio de instrucciones de algún procesador particular y se realizan prácticas en las que se programa en ensamblador el procesador. En las titulaciones de informática en Granada, se estudian como ejemplo los procesadores de la familia 86, tanto en asignaturas de estructura como de arquitectura. Hay varias razones que nos motivan a utilizar como ejemplo la familia 86: (1) Los estudiantes en el primer curso de la titulación, ya se han familiarizado con los conceptos relacionados con la arquitectura abstracta estudiando un procesador sencillo, el CODE [7]. (2) Son las arquitecturas disponibles en las aulas de prácticas, por lo no se familiarizan con la arquitectura a través de simuladores. (3) Los procesadores con la arquitectura abstracta de la línea 86 dominan el mercado del PC (Athlon, Pentium 4, Pentium M), y se están extendiendo a otros mercados, como el mercado de productos embebidos, de estaciones y servidores (Xeon, Opteron), e incluso en el mercado supercomputadores (hay cluster en la lista de supercomputadores TOP500 basados en procesadores de la línea 86. (4) Destacamos también, que Intel ofrece actualmente la arquitectura abstracta del propio 8086 (1978) para productos embebidos (escáner, impresoras, fotocopiadoras) a través del 80186 [5].

En el apartado 5 del guión se da al estudiante argumentos a favor de usar los conocimientos que adquieren en estructura en sus programas de alto nivel. Deben tener en cuenta que hay instrucciones en los repertorios de los procesadores, que los compiladores no generan. Dentro de estas instrucciones están lógicamente las que se han incorporado recientemente a las arquitecturas. Los compiladores que aprovechan una arquitectura aparecen generalmente después ésta. Pero además, hay instrucciones incorporadas con anterioridad, que los compiladores no generan, o no lo hacen en todos los casos donde resultan rentables, o las generan en casos o de forma que no son rentables. Generalmente, los compiladores advierten al programador, que debe comprobar si el código generado usando alguna opción de optimización realmente mejora prestaciones. Estas instrucciones que mencionamos pueden ser rentables para disminuir tiempo de ejecución, o para aprovechar una determinada funcionalidad.

Para poder beneficiarse de estas instrucciones, si el compilador no lo hace, el programador puede usar ensamblador. Además, si el ensamblador disponible tampoco conoce la instrucción, puede usar el código máquina de la instrucción. Otra alternativa para aprovechar estas instrucciones, es buscar librerías de funciones que las utilicen, lo que puede encarecer demasiado el coste por tiempo (de búsqueda, de entrenamiento) y dinero (para adquirirlas inicialmente y para mantenerlas) de desarrollo del software. Hay que tener en cuenta que las librerías que aprovechan una arquitectura aparecen en el mercado después de la arquitectura y probablemente con un alto precio.

Además, las funciones de librería son generales, no proporcionan las mismas prestaciones que una implementación que se adapte específicamente a las necesidades de la aplicación. Por otro lado, algún informático debe dedicarse a programarlas.

Instrucciones de los procesadores de la línea 86 de Intel cuyo uso explícito en ensamblador nos pueden permitir mejorar prestaciones, teniendo en cuenta los compiladores actuales, son por ejemplo: las instrucciones de manejo de cadenas (*movs, scas, cmps, stos*), instrucciones de precaptación de memoria, instrucciones que saltan caches, instrucciones con procesamiento SIMD (MMX, SSE, SSE2, 3DNow!), instrucciones de movimiento condicional (*cmov*) o las instrucciones *setcc* ([4], [1]).

Otras instrucciones nos ofrecen alguna funcionalidad que podemos necesitar, como por ejemplo *xchg* o *rdtsc* en los procesadores de la línea 86. Cuando se ejecuta la instrucción de intercambio *xchg* con un dato en memoria, si éste dato está en la cache del procesador, el hardware de mantenimiento de coherencia impide que cualquier otro componente del sistema pueda modificar el dato. Si el dato no está en cache se activa la línea de salida del procesador LOCK. Mientras esta línea está activa ningún componente puede acceder al bus del sistema. Esta característica de la instrucción hace que sea útil para implementar primitivas de sincronización. La instrucción *rdtsc*, devuelve el número de ciclos de reloj desde que se inició el sistema. Es útil para obtener el tiempo de ejecución con precisión de ciclos de reloj, y para poder realizar una comparativa de prestaciones entre arquitecturas, independiente de la frecuencia de reloj, lo que nos permite comparar características de la arquitectura concreta (superescalar, SIMD, multihebra simultánea, segmentada[1]).

Si alguna de las instrucciones se utiliza frecuentemente, se puede definir dentro de una macro. En el apartado 5 del guión se dan al estudiante algunas macros que pueden resultar de utilidad.

En el apartado 6 se describe al estudiante el trabajo que ha de realizar. El trabajo se ha dividido en dos apartados. En el primero el estudiante comprueba la utilidad de las instrucciones de movimiento condicional para mejorar prestaciones, con un ejemplo sencillo: el cálculo del mínimo de una lista. El estudiante compara los ciclos de reloj que supone el cálculo del mínimo usando una instrucción de movimiento condicional y usando el código que genera el compilador. En gcc no se ha incorporado aún la posibilidad de generar instrucciones de movimiento condicional para mejorar prestaciones. En compiladores comerciales, se ha incorporado esta posibilidad recientemente, a pesar de ser instrucciones incluidas en la línea 86 a partir del PentiumPro (1995). Así por ejemplo en el compilador Visual C++ de Microsoft se ha incorporado en la versión .NET 2003 la opción arch:SSE, que permite al compilador buscar puntos en los que resulta rentable usar la instrucción *cmov* e instrucciones incluidas en el repertorio multimedia SSE [6]. No obstante, el compilador no es capaz de usar estas instrucciones (incluida *cmov*) en todos los puntos donde resultan rentables, y, como hemos podido comprobar, no siempre el uso de esta opción mejora prestaciones.

Para medir ciclos de reloj el estudiante utiliza una de las macros que se presenta en el apartado 5 del guión, que utiliza la instrucción *rdtsc*.

Con el segundo apartado dentro de "Trabajo a desarrollar" se ilustra con tres ejemplos, la utilidad de las instrucciones de manejo de cadenas para mejorar prestaciones: (1) copia de datos de 32 bits de una zona a otra de memoria, (2) inicializar una lista de componentes de 32 bits a un valor, y (3) buscar un dato de 32 bits en una lista. Las instrucciones de tratamiento de cadenas estaban ya incluidas en el repertorio de instrucciones del 8086. En estos ejemplos también se miden ciclos de reloj usando la macro propuesta en el apartado 5. Hay funciones de librería de los compiladores que se suelen implementar en ensamblador para utilizar estas instrucciones de manejo de cadenas con el fin de mejorar prestaciones (por ejemplo las operaciones sobre listas de caracteres memchr, memcmp, memset o memcpy de la librería string.h).

3. Trabajo del estudiante

La práctica tiene asignada tres sesiones de dos horas, aunque la mayor parte de los estudiantes las realizan en dos sesiones. En la primera sesión ejecutan los ejemplos del apartado 4 del guión y

[1] No obstante, la segmentación se relaciona también con la frecuencia de reloj

comienzan el primer apartado de "Trabajo a desarrollar", cálculo del mínimo. El resto del tiempo asignado lo dedican al segundo apartado: copia, iniciar, y búsqueda.

Aquí presentaremos algunos resultados que obtienen los estudiantes para que se pueda percibir en qué medida comprueban los estudiantes que se mejoran prestaciones. Además indicaremos algunos de los comentarios que se dan a los alumnos durante el desarrollo de las secciones para relacionar los resultados obtenidos con los conceptos estudiados en clase. Los resultados mostrados aquí se han obtenido en tres computadores con diferentes procesadores que se encuentran en las aulas de prácticas: Pentium III (PIII), Pentium 4 (P4), y el Pentium M (PM) que incorporan los portátiles Centrino. Se recomienda a los estudiantes tomar datos para diferentes procesadores (diferentes aulas de prácticas) para poder así realizar comparativas entre procesadores. En la Tabla 2 se pueden ver algunas características de los procesadores utilizados.

En todos los ejercicios, se trabaja con listas con componentes de 32 bits. El estudiante realiza ejecuciones para diferentes tamaños en las listas: 16, 32, 64, 128, 256, 512, 1024, 2048 y 4096. Se llega a un tamaño en byte de 16KB. Así los estudiantes pueden observar el comportamiento para tamaños pequeños y la tendencia para tamaños grandes. Obtienen, para cada código a evaluar, el número de ciclos que consume su ejecución, y los ciclos que supone el procesamiento de una componente. Para cada tamaño obtienen la media de cuatro ejecuciones. Con estos valores rellenan una tabla, en la que también deben indicar la ganancia en prestaciones conseguida para cada tamaño. Además, deben representar los datos obtenidos en gráficas, para poder así examinar e interpretar cómodamente los resultados.

Tabla 2. Procesadores utilizados en la evaluación. Familia obtenida con CPUID.

Procesador	Familia	cache L1 datos	cache L1 inst.	cache L2
Pentium III	6	16 KB	16KB	256KB
Pentium 4	15	8 KB	12Kμop.	512KB
Pentium M	6	32KB	32KB	1MB

En el primer apartado de la sección "Trabajo a desarrollar" del guión de prácticas, se da al estudiante dos programas que calculan el mínimo en un vector de enteros positivos de 32 bits. Uno de los programas está escrito con código C de alto nivel, calculando el mínimo con el ciclo:

```
for (i=0;i<num;i++) {
   if (minnum>vector[i]) minnum = vector[i];  }
```

Con el fin de evitar el salto condicional que el compilador genera para implementar la sentencia *if*, el segundo programa emplea, usando ensamblador en línea dentro del código gcc, una secuencia de instrucciones como esta:

```
cmpl    vector[i], minnum
cmoval  vector, minum
```

La variable *minnum* estará en un registro. En esta secuencia se utiliza una instrucción de comparación (*cmp*), y una instrucción de movimiento condicional (*cmova*- mover si mayor). La instrucción, *cmp*, compara los dos operandos: *minnum* y la siguiente componente del vector. Si como resultado de la comparación se obtiene que el mínimo actual (*minnum)* es mayor que vector[i], entonces se actualiza el mínimo con el valor de vector[i]. En las Figuras 1, 2 y 3 se pueden ver los resultados de ejecución que obtendría los estudiantes en los tres procesadores de la Tabla 2 tanto para el código generado por gcc (C), como para el programa gcc con ensamblador en línea (S). En estas gráficas, se muestran, para cada tamaño, los ciclos que como media requiere el procesamiento de una componente (eje Y izquierda) y la ganancia en prestaciones que se consigue (relación C/S) para los diferentes tamaños de las listas (eje Y derecha). Representar los ciclos por componente, en lugar del total de ciclos, permite gráficas más compactas.

En los dos programas se genera aleatoriamente la lista, antes de pasar al cálculo del mínimo. Esto hace que la lista esté completa en la cache de nivel 2 antes de comenzar los cálculos. Por tanto, no se observa la influencia de las faltas de cache en L2.

Los estudiantes observan mejoras de prestaciones en los tres procesadores, la mejora es menor en el P4 que el PIII y el PM. En el PIII, el tiempo se reduce a más de la mitad usando ensamblador en línea (ganancia 2,1). Mientras que en el P4 el programa con ensamblador requiere, para tamaños grandes, aproximadamente un 86% del tiempo del código generado por gcc (ganancia 1,2). Por otra parte, el P4 consume más ciclos de reloj. El mayor número de etapas del cauce

(*pipeline*) del P4 se pone especialmente de manifiesto para tamaños pequeños del vector. Con tamaños pequeños afecta más la latencia que la productividad del cauce. La latencia es mayor en la versión C.

En el segundo apartado de la sección "Trabajo a desarrollar" del guión de prácticas, los estudiantes utilizan instrucciones de manejo de cadenas combinadas con prefijos de repetición (*rep*, *repnz*) para mejorar prestaciones. Se pretende ilustrar a los estudiantes sobre el uso de estas instrucciones, y sobre su utilidad para mejorar prestaciones. Utilizan en concreto: ***rep movs*** para copiar una lista de componentes de 32 bits en otra**,** ***rep stos*** para inicializar un vector de componentes de 32 bits a un valor, y ***repnz scas*** para encontrar un valor en un vector de 32 bits**.**

Como ejemplo, mostraremos aquí los resultados que obtienen los estudiantes para *rep movs*. El código de alto nivel que ejecutan es:

```
for (i=0;i<num;i++) destino[i]=fuente[i];
```

Para mejorar el código máquina generado por gcc, aprovechando *rep movs*, utilizan la siguiente sentencia *asm* que incluye además la interfaz con gcc:

```
asm ( "movl %0,%%esi     \n\t
      "movl %1,%%edi     \n\t"
      "movl %2,%%ecx     \n\t"
      "cld               \n\t"
      "rep   movsl       \n\t"
::"m"(fuente),"m"(destino),"m"(num)
:"%ecx", "%edi","%esi","memory");
```

Es conveniente que los estudiantes realicen tanto ejecuciones en las que los datos estén en cache L2 antes de pasar a ejecutar el código a evaluar, como ejecuciones en las que los datos no estén en cache. De esta forma pueden ver cómo afectan los fallos de cache.

En las Figuras 4, 5 y 6, se pueden ver los resultados que obtendrían los estudiantes para este ejemplo en los tres procesadores de la Tabla 2.

En la gráfica se presentan los ciclos por componente para cuatro ejecutables. Hay dos ejecutables con código máquina generado por gcc (C y C-Ca), y dos ejecutables con parte del código del programador (S y S-Ca). La parte de código del programador para la copia, es la que se acaba de especificar más arriba. En las versiones de código C-Ca y S-Ca los componentes de las dos listas se generan aleatoriamente antes de pasar al cálculo. De esta forma los dos vectores estarán en la cache L2. En las versiones C y S sólo se genera la lista fuente. En las gráfica se representa la ganancia para la versión sin lista destino en cache (C/S), y la ganancia para la versión con las dos listas en cache (C/S-Ca).

Se debe indicar a los estudiantes que el hecho de que los datos estén o no en cache al realizar el cálculo, dependerá de la aplicación concreta en la que se utilice el código, del hardware de precaptación que incluya el procesador, y también de la utilización o no de instrucciones ensamblador de precaptación.

Con estos ejecutables se observa la efectividad del hardware de precaptación a cache de nivel 2 incluido en el P4 y en el PM. El hardware de precaptación incluido en los P4, PM, y en los PIII fabricados con tecnología de 0.13µm (no es el caso del PIII usado en los resultados presentados), precapta datos al nivel 2 de cache en caso de que se detecte un acceso a memoria secuencial con salto constante. El mecanismo se dispara concretamente cuando se detecta en un acceso regular 2-4 faltas consecutivas.

Como se puede ver en las gráficas, en todos los casos la versión con ensamblador en línea (S) mejora prestaciones con respecto a su correspondiente versión sin código ensamblador (C). En las versiones en las que todos los vectores están en cache antes del cálculo (versiones etiquetadas con Ca) los estudiantes observan mayor ganancia. La mayor ganancia se debe a que la latencia en el acceso L2 no oculta la mejora de las instrucciones de manejo de cadenas.

Los resultados permiten comprobar la componente no determinista que la jerarquía de memoria puede introducir en el tiempo de ejecución, y la utilidad del hardware incorporado en los procesadores para evitar la penalización en el acceso a memoria principal. Los tiempos de las versiones con vectores en cache son muy reproducibles, especialmente en el PM que tiene una cache L2 de 1MB. También son muy reproducibles en el PM los tiempos para las versiones sin vectores en cache, debido a la eficiencia de su hardware de precaptación. En el PIII, la falta de hardware de precaptación hace muy notables las diferencias entre las versiones sin vectores en cache (C y S) y las versiones con vectores en cache (C-Ca y S-Ca).

Las gráficas también reflejan la arquitectura segmentada. Conforme se aumenta el tamaño de las listas, se decrementa el número de ciclos que requiere el procesamiento de una componente.

Esto es debido a que conforme se aumenta el tamaño, se va tendiendo a la productividad que proporciona el cauce segmentado del procesador para el código que se ejecuta cíclicamente. La ganancia en prestaciones se incrementa conforme se aumenta el tamaño, poniendo de manifiesto la mejor productividad de las instrucciones de tratamiento de cadenas frente al código que genera el compilador. Esta tendencia se observa con menos claridad en el PIII sin datos precargados en L2, debido a la penalización de las faltas de cache. En estas experiencias realizadas para tamaños de hasta 4096 componentes, en la copia, se observan ganancias de 3,7 en P4, 4 en PM y por encima de 4 en PIII.

Así pues, los resultados familiarizan además al estudiante con otros conceptos que han abordado en la asignatura y que se amplían en asignaturas posteriores, como la jerarquía de memoria, o la segmentación del procesador. Observan la influencia de la jerarquía en las prestaciones, lo que supondría su ausencia, y los efectos de las faltas de cache. También observan los beneficios de la segmentación del procesador y la degradación de prestaciones si hay *paradas* en el cauce, debidas a instrucciones de salto (cálculo del mínimo) o a esperas de datos (faltas de cache).

4. Conclusiones

Para incrementar la motivación en el estudio de las materias de arquitectura y tecnología de computadores, estamos proponiendo a los estudiantes de informática, prácticas en las que verifican por sí mismo que pueden mejorar las prestaciones de sus programas utilizando contenidos de estas materias. Aquí concretamente se ha presentado una práctica para asignaturas de Estructura de los Computadores. En esta práctica los estudiantes, utilizando ensamblador en línea dentro de código de alto nivel, mejoran las prestaciones de cuatro ejemplos sencillos: (1) cálculo del mínimo, (2) copia de una lista de componentes de 32 bits, (3) inicializar una lista de componentes de 32 bits a un valor, y (4) buscar un dato de 32 bits en una lista. Para mejorar prestaciones se utilizan instrucciones que se introdujeron en el 8086 (1978) y en el Pentium Pro (1995). Con estos ejemplos sencillos, pretendemos ilustrar e introducir a los estudiantes en la mejora de prestaciones teniendo en cuenta las características de las arquitecturas. También presentamos a los estudiantes mejoras en prestaciones en aplicaciones completas aprovechando los conocimientos de arquitectura, mostrándoles resultados obtenidos por estudiantes de informática en trabajos fin de carrera.

Los estudiantes comprueban, que con un único procesador, usando sus conocimientos sobre arquitectura *abstracta*, pueden conseguir para estos ejemplos, ganancias en velocidad entre 1,2 y 4,9. La ganancia en prestaciones depende del ejemplo, de la arquitectura *concreta* del procesador, y del tamaño de las listas de entrada.

Creemos conveniente que los estudiantes perciban en los primeros cursos, la importancia de que un ingeniero en informática que va a desarrollar software, conozca la arquitectura actual para poder y saber aprovecharla.

Referencias

[1] “AMD Athlon™ Processor x86 Code Optimization Guide”, ttp://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_1274_3734^3748,00.html

[2] Djgpp home, http://www.delorie.com/djgpp/

[3] J. Flisch, J. Real, J.C. Cano y J. Sahuquillo. Prácticas de ensamblador. VI Jornadas de Enseñanza Universitaria de Informática (JENUI 2000), Alcalá de Henares. Madrid.

[4] “Intel® Pentium® 4 Processor Optimization Reference Manual”, http://www.intel.com/design/pentium4/manuals/248966.htm

[5] Intel 186 processor http://www.intel.com/design/intarch/intel186/index.htm?iid=ipp_browse+embed_186process&

[6] M. Lacey, “Optimizing Your Code with Visual C++” , Microsoft Corporation, 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchOptimizingYourCodeWithVisualC.asp

[7] A. Prieto, F.J. Pelayo, F. Gómez-Mula, J. Ortega, A. Cañas, A. Martínez, F.J. Fernández. “Un computador didáctico elemental (CODE-2)” ”, JENUI , Cáceres.

[8] Padilla, J.A; Anguita, M., Fernández, F.J.; Díaz, A.F.; Cañas, A.; Prieto, A. “Optimización de una implementación JPEG teniendo en cuenta la arquitectura actual de los procesadores (JENUI 2003), Cádiz.

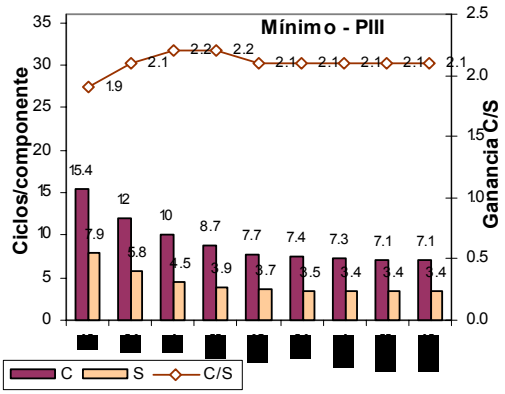

Figura 1. Mínimo Pentium III

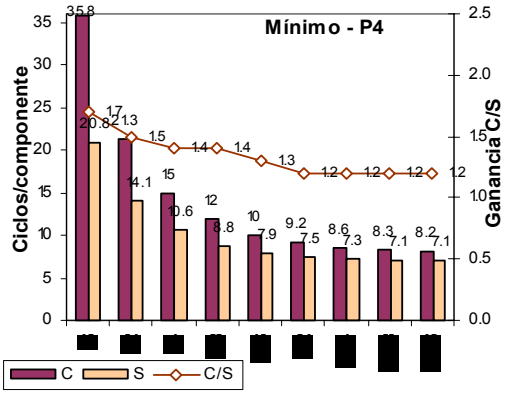

Figura 2. Mínimo Pentium 4

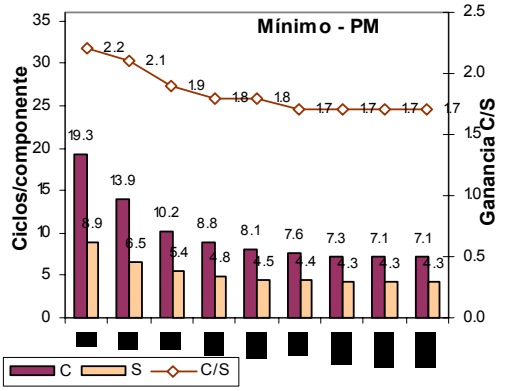

Figura 3. Mínimo Pentium M

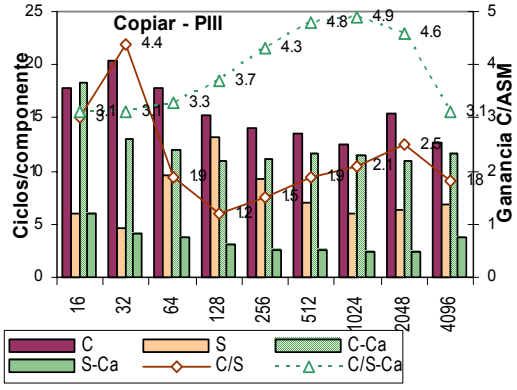

Figura 4. Copia Pentium 3

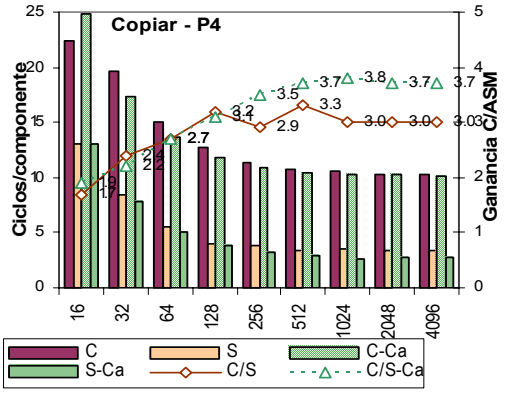

Figura 5. Copia Pentium 4

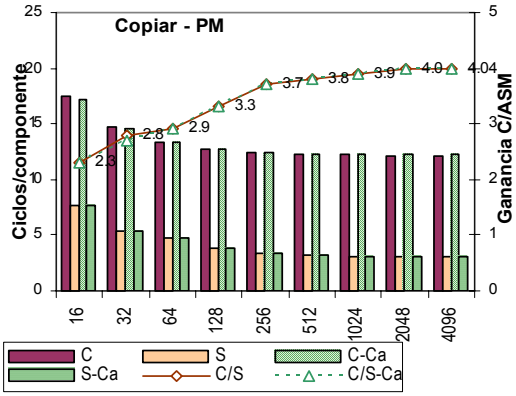

Figura 6. Copia Pentium M